Table of Contents

JVM Anatomy Quarks: 2021-07-31 Snapshot

About, Disclaimers, Contacts

"JVM Anatomy Quarks" (https://shipilev.net/jvm/anatomy-quarks/) is the on-going mini-post series, where every post is describing some elementary piece of knowledge about JVM. The name underlines the fact that the single post cannot be taken in isolation, and most pieces described here are going to readily interact with each other.

The post should take about 5-10 minutes to read. As such, it goes deep for only a single topic, a single test, a single benchmark, a single observation. The evidence and discussion here might be anecdotal, not actually reviewed for errors, consistency, writing 'tyle, syntaxtic and semantically errors, duplicates, or also consistency. Use and/or trust this at your own risk.

Aleksey Shipilëv, JVM/Performance Geek
Shout out at Twitter: @shipilev (http://twitter.com/shipilev); Questions, comments, suggestions: aleksey@shipilev.net (mailto:aleksey@shipilev.net)

This is a rolling release with all posts in one, generated at 2021-07-31.

JVM Anatomy Quark #1: Lock Coarsening and Loops

Do these play together at all?

Question

It is known that Hotspot does lock coarsening optimizations (https://en.wikipedia.org/wiki/Java_performance#Escape_analysis_and_lock_coarsening) that can effectively merge several adjacent locking blocks, thus reducing the locking overhead. It effectively converts this:

```java
synchronized (obj) {
  // statements 1
}
synchronized (obj) {
  // statements 2
}
```

...into:

```java
synchronized (obj) {
  // statements 1
  // statements 2
}
```

Now, the interesting question that was posed today is, does Hotspot do this optimization for *loops*? E.g. having:

```java
for (...) {
  synchronized (obj) {
    // something
  }
}
```

...could it optimize into this?

```java
synchronized (this) {
  for (...) {
     // something
  }
}
```

Theoretically, nothing prevents us from doing this. One might even see the optimization like a loop unswitching (https://en.wikipedia.org/wiki/Loop_unswitching) on steroids, only for locks. However, the downside is potentially coarsening the lock so much, the particular thread would hog the lock while executing a fat loop.

Experiment

The easiest way to approach answering this is to find the positive evidence current Hotspot does optimization like it. Luckily, it is pretty simple with JMH (http://openjdk.java.net/projects/code-tools/jmh/). It is useful not only for building the benchmarks, but also for the most important part of engineering, *analyzing* them. Let's start with a simple benchmark:

```java
@Fork(..., jvmArgsPrepend = {"-XX:-UseBiasedLocking"})
@State(Scope.Benchmark)
public class LockRoach {
    int x;

    @Benchmark
    @CompilerControl(CompilerControl.Mode.DONT_INLINE)
    public void test() {
        for (int c = 0; c < 1000; c++) {
            synchronized (this) {
                x += 0x42;
            }
        }
    }
}
```

(full source here)

There are a few important tricks here:

1. Disabling biased locking with `-XX:-UseBiasedLocking` avoids longer warmups, because biased locking is not started up immediately, but instead waits 5 seconds through the initialization phase. (See `BiasedLockingStartupDelay` option).
2. Disabling inlining for `@Benchmark` method helps to separate it in the disassembly.
3. Adding up a magic number, `0x42` helps to quickly find the increment in the disassembly.

Running at i7 4790K, Linux x86_64, JDK EA 9b156:

```
Benchmark            Mode  Cnt      Score    Error  Units
LockRoach.test       avgt    5   5331.617 ± 19.051  ns/op
```

What can you tell from this number? You can't tell anything, right? We need to look into what actually happened down below. `-prof perfasm` is very useful for this, as it shows you the hottest regions in the generated code. Running with default settings would tell that the hottest instructions are actual `lock cmpxchg` (compare-and-sets) that perform locking, and only print hot things around them. Running with `-prof perfasm:mergeMargin=1000` to coalesce these hot regions into a solid picture, one would get this scary-at-first-sight piece of output.

Stripping it further down — the cascades of jumps are the locking/unlocking — and paying attention to the code that accumulates the most cycles (first column), we can see that the hottest loop looks like this:

```asm
↗  0x00007f455cc708c1: lea    0x20(%rsp),%rbx
|          < blah-blah-blah, monitor enter >     ; <--- coarsened!
|  0x00007f455cc70918: mov    (%rsp),%r10        ; load $this
|  0x00007f455cc7091c: mov    0xc(%r10),%r11d    ; load $this.x
|  0x00007f455cc70920: mov    %r11d,%r10d        ; ...hm...
|  0x00007f455cc70923: add    $0x42,%r10d        ; ...hmmm...
|  0x00007f455cc70927: mov    (%rsp),%r8         ; ...hmmmmm!...
|  0x00007f455cc7092b: mov    %r10d,0xc(%r8)     ; LOL Hotspot, redundant store, killed two lines below
|  0x00007f455cc7092f: add    $0x108,%r11d       ; add 0x108 = 0x42 * 4 <-- unrolled by 4
|  0x00007f455cc70936: mov    %r11d,0xc(%r8)     ; store $this.x back
|          < blah-blah-blah, monitor exit >      ; <--- coarsened!
|  0x00007f455cc709c6: add    $0x4,%ebp          ; c += 4   <--- unrolled by 4
|  0x00007f455cc709c9: cmp    $0x3e5,%ebp        ; c < 1000?
└  0x00007f455cc709cf: jl     0x00007f455cc708c1
```

Huh. The loop seems to be unrolled (https://en.wikipedia.org/wiki/Loop_unrolling) by 4, and *then* locks are coarsened within those 4 iterations! Okay then, if that happens due to loop unrolling, we can quantify the performance benefits of doing this limited coarsening, but trimming down the unrolling with `-XX:LoopUnrollLimit=1`:

```
Benchmark            Mode  Cnt      Score    Error  Units

# Default
LockRoach.test       avgt    5   5331.617 ± 19.051  ns/op

# -XX:LoopUnrollLimit=1
LockRoach.test       avgt    5  20679.043 ±  3.133  ns/op
```

Whoa, 4x performance hit! That stands to reason, because we have already observed that the hottest things are `lock cmpxchg` from locking. Naturally, 4x coarsened lock means 4x better throughput. Very cool, we can claim success and move on? Not yet, we have to verify that disabling loop unrolling actually gives us what we want to compare against. perfasm seems to indicate it does the similar hot loop, but with a single stride.

```asm
↗  0x00007f964d0893d2: lea    0x20(%rsp),%rbx
│          < blah-blah-blah, monitor enter >
│  0x00007f964d089429: mov    (%rsp),%r10        ; load $this
│  0x00007f964d08942d: addl   $0x42,0xc(%r10)    ; $this.x += 0x42
│          < blah-blah-blah, monitor exit >
│  0x00007f964d0894be: inc    %ebp               ; c++
│  0x00007f964d0894c0: cmp    $0x3e8,%ebp        ; c < 1000?
└  0x00007f964d0894c6: jl     0x00007f964d0893d2 ;
```

Ah, OK, everything checks out.

Observations

While lock coarsening does not work on the entire loop, *another* loop optimization — loop unrolling — sets up the stage for the regular lock coarsening, once the intermediate representation starts to look as if there are N adjacent lock-unlock sequences. This reaps the performance benefits, and helps to limit the scope of coarsening, to avoid over-coarsening over fat loops.

JVM Anatomy Quark #2: Transparent Huge Pages

Question

What are Large Pages? What are Transparent Huge Pages? How does it help me?!

Theory

Virtual memory is taken for granted now. Only a few now remember, let alone do, some "real mode" programming, where you are exposed to the actual physical memory. Instead, every process has its own *virtual* memory space, and that space is mapped onto actual memory. That allows, for instance, for two processes have distinct data at *same* virtual address `0x42424242`, which will be backed by *different* physical memory. Now, when a program does the access at that address, something should translate that virtual address to physical one.

This is normally achieved by OS maintaining the "page table" (https://en.wikipedia.org/wiki/Page_table), and hardware doing the "page table walk" through that table to translate the address. The whole thing gets easier when translations are maintained at page granularity. But it is nevertheless not very cheap, and it needs to happen for **every** memory access! Therefore, there is also a small cache of latest translations, Translation Lookaside Buffer (TLB) (https://en.wikipedia.org/wiki/Translation_lookaside_buffer). TLB is usually very small, below 100 of entries, because it needs to be at least as fast as L1 cache, if not faster. For many workloads, TLB misses and associated page table walks take significant time.

Since we cannot do TLB larger, we can do something else: make larger **pages**! Most hardware has 4K basic pages, and 2M/4M/1G "large pages". Having larger pages to cover the same region also makes page tables themselves smaller, making the cost of page table walk lower.

In Linux world, there are at least two distinct ways to get this in applications:

- **hugetlbfs** (https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt). Cut out the part of system memory, expose it as virtual filesystem, and let applications `mmap(2)` from it. This is a peculiar interface that requires both OS configuration and application changes to use. This also "all or nothing" kind of deal: the space allocated for (the persistent part of) hugetlbfs cannot be used by regular processes.
- **Transparent Huge Pages (THP)** (https://www.kernel.org/doc/Documentation/vm/transhuge.txt). Let application allocate memory as usual, but try to provide large-pages-backed storage transparently to the application. Ideally, no application changes are needed, but we will see how applications can benefit from knowing THP is available. In practice, though, there are memory overheads (because you will allocate an entire large page for something small), or time overheads (because sometimes THP needs to defrag memory to allocate pages). The good part is that there is a middle-ground: `madvise(2)` lets application tell Linux where to use THP.

Why the nomenclature uses "large" and "huge" interchangeably is beyond me. Anyway, OpenJDK supports both modes:

```
$ java -XX:+PrintFlagsFinal 2>&1 | grep Huge
  bool UseHugeTLBFS              = false      {product} {default}
  bool UseTransparentHugePages   = false      {product} {default}
$ java -XX:+PrintFlagsFinal 2>&1 | grep LargePage
  bool UseLargePages             = false   {pd product} {default}
```

`-XX:+UseHugeTLBFS` mmaps Java heap into hugetlbfs, that should be prepared separately.

`-XX:+UseTransparentHugePages` just `madvise`-s that Java heap should use THP. This is convenient option, because we know that Java heap is large, mostly contiguous, and probably benefits from large pages the most.

`-XX:+UseLargePages` is a generic shortcut that enables anything available. On Linux, it enables hugetlbfs, not THP. I guess that is for historical reasons, because hugetlbfs came first.

Some applications do suffer (https://bugs.openjdk.java.net/browse/JDK-8024838) with large pages enabled. (It is sometimes funny to see how people do manual memory management to avoid GCs, only to hit THP defrag causing latency spikes for them!) My gut feeling is that THP regresses mostly short-lived applications where defrag costs are visible in comparison to short application time.

Experiment

Can we show what benefit large pages give us? Of course we can, let's take a workload that any systems performance engineer had run at least once by their mid-thirties. Allocate and randomly touch a `byte[]` array:

```java
public class ByteArrayTouch {

    @Param(...)
    int size;

    byte[] mem;

    @Setup
    public void setup() {
        mem = new byte[size];
    }

    @Benchmark
    public byte test() {
        return mem[ThreadLocalRandom.current().nextInt(size)];
    }
}
```

(full source here)

We know that depending on size, the performance would be dominated either by L1 cache misses, or L2 cache misses, or L3 cache misses, etc. What this picture usually omits is the TLB miss costs.

Before we run the test, we need to decide how much heap we will take. On my machine, L3 is about 8M, so 100M array would be enough to get past it. That means, pessimistically allocating 1G heap with `-Xmx1G -Xms1G` would be enough. This also gives us a guideline how much to allocate for hugetlbfs.

So, making sure these options are set:

```shell
# HugeTLBFS should allocate 1000*2M pages:
sudo sysctl -w vm.nr_hugepages=1000

# THP to "madvise" only (some distros have an opinion about defaults):
echo madvise | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
echo madvise | sudo tee /sys/kernel/mm/transparent_hugepage/defrag
```

I like to do "madvise" for THP, because it lets me to "opt-in" for particular parts of memory we know would benefit.

Running on i7 4790K, Linux x86_64, JDK 8u101:

```
Benchmark               (size)  Mode  Cnt   Score   Error  Units

# Baseline
ByteArrayTouch.test       1000  avgt   15   8.109 ± 0.018  ns/op
ByteArrayTouch.test      10000  avgt   15   8.086 ± 0.045  ns/op
ByteArrayTouch.test    1000000  avgt   15   9.831 ± 0.139  ns/op
ByteArrayTouch.test   10000000  avgt   15  19.734 ± 0.379  ns/op
ByteArrayTouch.test  100000000  avgt   15  32.538 ± 0.662  ns/op

# -XX:+UseTransparentHugePages
ByteArrayTouch.test       1000  avgt   15   8.104 ± 0.012  ns/op
ByteArrayTouch.test      10000  avgt   15   8.060 ± 0.005  ns/op
ByteArrayTouch.test    1000000  avgt   15   9.193 ± 0.086  ns/op // !
ByteArrayTouch.test   10000000  avgt   15  17.282 ± 0.405  ns/op // !!
ByteArrayTouch.test  100000000  avgt   15  28.698 ± 0.120  ns/op // !!!

# -XX:+UseHugeTLBFS
ByteArrayTouch.test       1000  avgt   15   8.104 ± 0.015  ns/op
ByteArrayTouch.test      10000  avgt   15   8.062 ± 0.011  ns/op
ByteArrayTouch.test    1000000  avgt   15   9.303 ± 0.133  ns/op // !
ByteArrayTouch.test   10000000  avgt   15  17.357 ± 0.217  ns/op // !!
ByteArrayTouch.test  100000000  avgt   15  28.697 ± 0.291  ns/op // !!!
```

A few observations here:

1. On smaller sizes, both cache and TLB are fine, and there is no difference against baseline.
2. On larger sizes, cache misses start to dominate, this is why costs grow in every configuration.
3. On larger sizes, TLB misses come to picture, and enabling large pages helps a lot!
4. Both `UseTHP` and `UseHTLBFS` help the same, because they are providing the same service to application.

To verify the TLB miss hypothesis, we can see the hardware counters. JMH `-prof perfnorm` gives them normalized by operation.

```
Benchmark                                (size)  Mode  Cnt    Score    Error  Units

# Baseline
ByteArrayTouch.test                   100000000  avgt   15   33.575 ±  2.161  ns/op
ByteArrayTouch.test:cycles            100000000  avgt    3  123.207 ± 73.725   #/op
ByteArrayTouch.test:dTLB-load-misses  100000000  avgt    3    1.017 ±  0.244   #/op  // !!!
ByteArrayTouch.test:dTLB-loads        100000000  avgt    3   17.388 ±  1.195   #/op

# -XX:+UseTransparentHugePages
ByteArrayTouch.test                   100000000  avgt   15   28.730 ±  0.124  ns/op
ByteArrayTouch.test:cycles            100000000  avgt    3  105.249 ±  6.232   #/op
ByteArrayTouch.test:dTLB-load-misses  100000000  avgt    3   ≈ 10⁻³            #/op
ByteArrayTouch.test:dTLB-loads        100000000  avgt    3   17.488 ±  1.278   #/op
```

There we go! One dTLB load miss per operation in baseline, and much less with THP enabled.

Of course, with THP defrag enabled, you will pay the upfront cost of defragmentation at allocation/access time. To shift these costs to the JVM startup that will avoid surprising latency hiccups when application is running, you may instruct JVM to touch every single page in Java heap with `-XX:+AlwaysPreTouch` during initialization. It is a good idea to enable pre-touch for larger heaps anyway.

And there comes the funny part: enabling `-XX:+UseTransparentHugePages` actually makes `-XX:+AlwaysPreTouch` faster, because OS now handles larger pages: there are less of them to handle, and there are more wins in streaming (zeroing) writes by OS. Freeing memory after process dies is also faster with THP, sometimes gruesomely so, until parallel freeing patch (https://lwn.net/Articles/715501/) trickles down to distro kernels.

Case in point, using 4 TB (terabyte, with a T) heap:

```shell
$ time java -Xms4T -Xmx4T -XX:-UseTransparentHugePages -XX:+AlwaysPreTouch
real    13m58.167s  # About 5 GB/sec
user    43m37.519s
sys     1011m25.740s

$ time java -Xms4T -Xmx4T -XX:+UseTransparentHugePages -XX:+AlwaysPreTouch
real    2m14.758s   # About 31 GB/sec
user    1m56.488s
sys     73m59.046s
```

Committing and freeing 4 TB sure takes a while!

Observations

Large pages are easy trick to boost application performance. Transparent Huge Pages in Linux kernel makes it more accessible. Transparent Huge Pages support in JVM makes it easy to opt-in. It is always a good idea to try large pages, especially if your application has lots of data and large heaps.

JVM Anatomy Quark #3: GC Design and Pauses

Do these play together at all?

Question

Garbage Collection is the enemy. But I must not fear. Fear is the mind-killer. Fear is the little-death that brings total obliteration... wait, what was the question again? The actual question was about to discuss the claim that **"Allocating 100M objects in array list is enough to show how Java hiccups for seconds"**. Is that true?

Landscape

It is easy to scapegoat generic GC as performance hog, while the problem lies in the GC *implementations* that do not perform up to your expectations on your workloads. In many cases, those workloads are themselves problematic, but in many cases workloads are running with unsuitable GCs! Let's see what is there in OpenJDK landscape GC-wise:

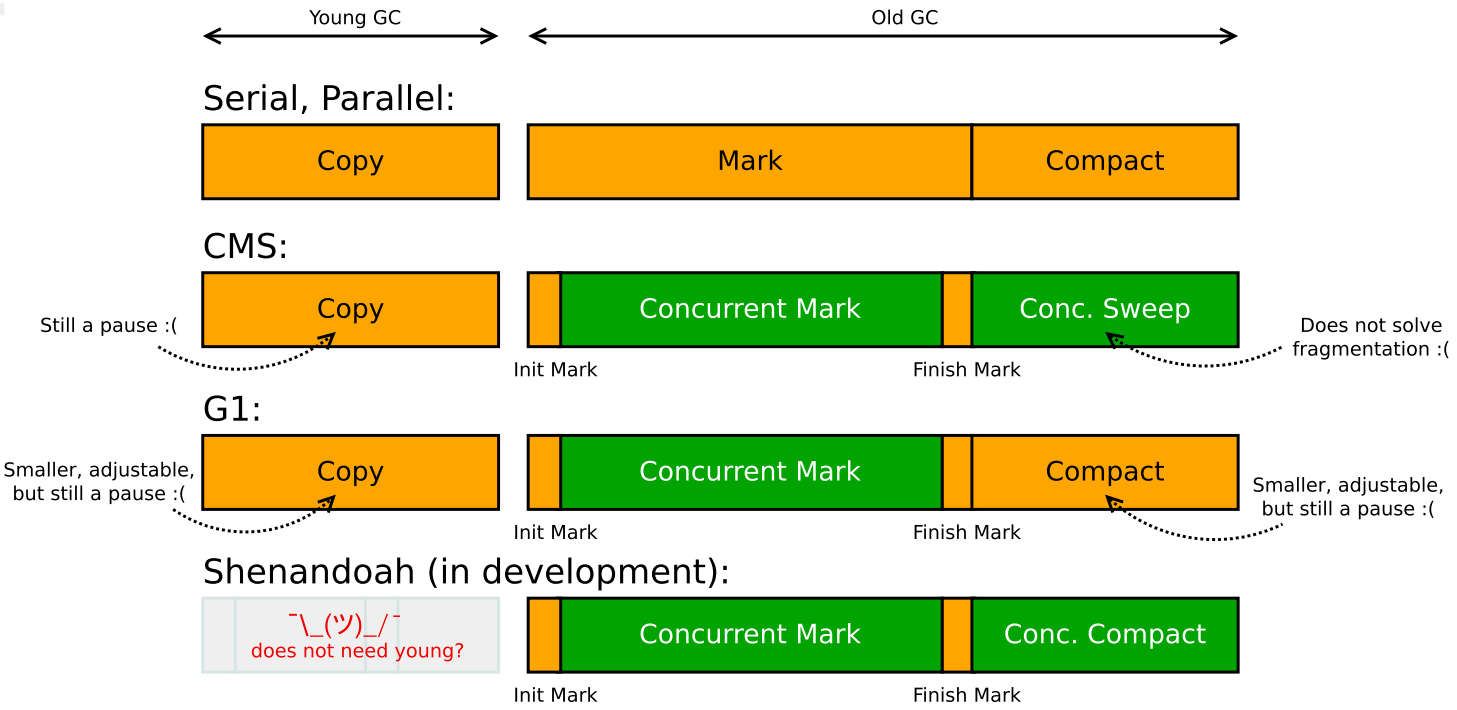


Figure 1. OpenJDK GC landscape. Yellow are stop-the-world phases, green are concurrent phases.

Notice how most collectors have pauses in their regular GC cycles.

Experiment

While rejecting the "100M Objects into ArrayList" test as unrealistic is fun, we can still run it and see how it performs. Quickly hacking it together:

```java
import java.util.*;

public class AL {
    static List<Object> l;
    public static void main(String... args) {
        l = new ArrayList<>();
        for (int c = 0; c < 100_000_000; c++) {
            l.add(new Object());
        }
    }
}
```

A little bit of cow wisdom about this:

```
$ cowsay ...
 _____________________________________
/ This is a crappy GC benchmark, by the  \
| way. I am a cow, and even I understand |
\ this.                                  /
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Still, even a crappy benchmark tells you something about the system under test. You have to only be careful to understand *what* it tells you. Turns out, the workload like above highlights the GC design choices of different collectors in OpenJDK.

Let's run with latest JDK 9 + Shenandoah forest to get all the latest GC implementation improvements. For a change, do this on a low-level 1.7 GHz i5 ultrabook with Linux x86_64. Since we are about to allocate 100M 16-byte objects, setting up the heap to static 4 GB seems fine, and eliminates some degrees of freedom between collectors.

G1 (default in JDK 9)

```
$ time java -Xms4G -Xmx4G -Xlog:gc AL
[0.030s][info][gc] Using G1
[1.525s][info][gc] GC(0) Pause Young (G1 Evacuation Pause) 370M->367M(4096M) 991.610ms
[2.808s][info][gc] GC(1) Pause Young (G1 Evacuation Pause) 745M->747M(4096M) 928.510ms
[3.918s][info][gc] GC(2) Pause Young (G1 Evacuation Pause) 1105M->1107M(4096M) 764.967ms
[5.061s][info][gc] GC(3) Pause Young (G1 Evacuation Pause) 1553M->1555M(4096M) 601.680ms
[5.835s][info][gc] GC(4) Pause Young (G1 Evacuation Pause) 1733M->1735M(4096M) 465.216ms
[6.459s][info][gc] GC(5) Pause Initial Mark (G1 Humongous Allocation) 1894M->1897M(4096M) 398.453ms
[6.459s][info][gc] GC(6) Concurrent Cycle
[7.790s][info][gc] GC(7) Pause Young (G1 Evacuation Pause) 2477M->2478M(4096M) 472.079ms
[8.524s][info][gc] GC(8) Pause Young (G1 Evacuation Pause) 2656M->2659M(4096M) 434.435ms
[11.104s][info][gc] GC(6) Pause Remark 2761M->2761M(4096M) 1.020ms
[11.979s][info][gc] GC(6) Pause Cleanup 2761M->2215M(4096M) 2.446ms
[11.988s][info][gc] GC(6) Concurrent Cycle 5529.427ms

real  0m12.016s
user  0m34.588s
sys   0m0.964s
```

What do we see with G1? Young pauses that are in in 500..1000 ms range. These pauses are likely to be less once we reach steady state and heuristics figure out how much to collect to keep pause times on target. After a while, concurrent GC cycle starts, and continues until almost to the end. (Notice how Young collections overlap with concurrent phases too). It should have been followed by "mixed" collection pauses, but VM had exited before that. These non-steady-state pauses really contributed to the long run times for this one-off job.

Also, notice how "user" time is larger than "real" (wallclock) time. This is because GC work is *parallel*, and so while the application is running in a single thread, GCs are using all available parallelism to make the collections faster wallclock-wise.

Parallel

```
$ time java -XX:+UseParallelOldGC -Xms4G -Xmx4G -Xlog:gc AL
[0.023s][info][gc] Using Parallel
[1.579s][info][gc] GC(0) Pause Young (Allocation Failure) 878M->714M(3925M) 1144.518ms
[3.619s][info][gc] GC(1) Pause Young (Allocation Failure) 1738M->1442M(3925M) 1739.009ms

real  0m3.882s
user  0m11.032s
sys   0m1.516s
```

With Parallel, we see similar Young pauses, which also probably resized the Eden/Survivors just enough to accept more temporary allocations. Therefore, we have only two large pauses, and the workload finishes quickly. In steady state, this collector would probably keep the same large pauses. "user" >> "real" as well, so some overhead is hiding in concurrent GC work here.

Concurrent Mark Sweep

```
$ time java -XX:+UseConcMarkSweepGC -Xms4G -Xmx4G -Xlog:gc AL
[0.012s][info][gc] Using Concurrent Mark Sweep
[1.984s][info][gc] GC(0) Pause Young (Allocation Failure) 259M->231M(4062M) 1788.983ms
[2.938s][info][gc] GC(1) Pause Young (Allocation Failure) 497M->511M(4062M) 871.435ms
[3.970s][info][gc] GC(2) Pause Young (Allocation Failure) 777M->850M(4062M) 949.590ms
[4.779s][info][gc] GC(3) Pause Young (Allocation Failure) 1117M->1161M(4062M) 732.888ms
[6.604s][info][gc] GC(4) Pause Young (Allocation Failure) 1694M->1964M(4062M) 1662.255ms
[6.619s][info][gc] GC(5) Pause Initial Mark 1969M->1969M(4062M) 14.831ms
[6.619s][info][gc] GC(5) Concurrent Mark
[8.373s][info][gc] GC(6) Pause Young (Allocation Failure) 2230M->2365M(4062M) 1656.866ms
[10.397s][info][gc] GC(7) Pause Young (Allocation Failure) 3032M->3167M(4062M) 1761.868ms
[16.323s][info][gc] GC(5) Concurrent Mark 9704.075ms
[16.323s][info][gc] GC(5) Concurrent Preclean
[16.365s][info][gc] GC(5) Concurrent Preclean 41.998ms
[16.365s][info][gc] GC(5) Concurrent Abortable Preclean
[16.365s][info][gc] GC(5) Concurrent Abortable Preclean 0.022ms
[16.478s][info][gc] GC(5) Pause Remark 3390M->3390M(4062M) 113.598ms
[16.479s][info][gc] GC(5) Concurrent Sweep
[17.696s][info][gc] GC(5) Concurrent Sweep 1217.415ms
[17.696s][info][gc] GC(5) Concurrent Reset
[17.701s][info][gc] GC(5) Concurrent Reset 5.439ms

real  0m17.719s
user  0m45.692s
sys   0m0.588s
```

Contrary to popular belief, in CMS, "Concurrent" means the concurrent collections in old generation. The young collections are still stopping the world, as we can see here. From the GC log standpoint, the phasing looks like G1: young pauses, concurrent cycle. The difference is that "Concurrent Sweep" cleans up old without stopping the application, in contrast to G1 Mixed pauses. Anyhow, the longer Young GC pauses without heuristics able to catch up with them yet defines the performance on this quick job.

Shenandoah

```
$ time java -XX:+UseShenandoahGC -Xms4G -Xmx4G -Xlog:gc AL
[0.026s][info][gc] Using Shenandoah
[0.808s][info][gc] GC(0) Pause Init Mark 0.839ms
[1.883s][info][gc] GC(0) Concurrent marking 2076M->3326M(4096M) 1074.924ms
[1.893s][info][gc] GC(0) Pause Final Mark 3326M->2784M(4096M) 10.240ms
[1.894s][info][gc] GC(0) Concurrent evacuation  2786M->2792M(4096M) 0.759ms
[1.894s][info][gc] GC(0) Concurrent reset bitmaps 0.153ms
[1.895s][info][gc] GC(1) Pause Init Mark 0.920ms
[1.998s][info][gc] Cancelling concurrent GC: Stopping VM
[2.000s][info][gc] GC(1) Concurrent marking 2794M->2982M(4096M) 104.697ms

real  0m2.021s
user  0m5.172s
sys   0m0.420s
```

In Shenandoah (https://wiki.openjdk.java.net/display/shenandoah/Main), there are no young collections. (At least today. There are some ideas how to get quick partial collections without introducing generations — but they are unlikely to be stop-the-world). The concurrent GC cycle starts and runs along with application, stopping it with two minor pauses to initiate and finish the concurrent mark. Concurrent copying takes nothing, because everything is alive and not yet fragmented. The second GC cycle terminates early due to VM shutdown. The absence of pauses like in other collectors explains why the workload can finish quickly.

Epsilon

```
$ time java -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC -Xms4G -Xmx4G  -Xlog:gc AL
[0.031s][info][gc] Initialized with 4096M non-resizable heap.
[0.031s][info][gc] Using Epsilon GC
[1.361s][info][gc] Total allocated: 2834042 KB.
[1.361s][info][gc] Average allocation rate: 2081990 KB/sec

real  0m1.415s
user  0m1.240s
sys   0m0.304s
```

Running with experimental "no-op" Epsilon GC (http://openjdk.java.net/jeps/318) can help to estimate GC overheads when no collector is running at all. Here, we can fit exactly in 4 GB pre-sized heap, and application runs with no pauses whatsoever. It would not survive anything more actively mutating the heap, though. Notice that "real" and "user"+"sys" times are almost equal, which corroborates the theory there was a single application thread only.

Observations

Different GCs have different tradeoffs in their implementations. Brushing the GC off as "bad idea" is a stretch. Choose a collector for your kind of workload, by understanding your workload, available GC implementations, and your performance requirements. Even if you choose to target platforms without GCs, you would still need to know (and choose!) your native memory allocators. When running experimental workloads, try to understand what they tell you, and learn from that. Peace.

JVM Anatomy Quark #4: TLAB allocation

Do these play together at all?

Question

What is TLAB allocation? Pointer-bump allocation? Who is responsible for allocating objects anyway?

Theory

Most of the time we do `new MyClass()`, the runtime environment has to allocate storage for the instance in question. The textbook GC (memory manager) interface for allocation is very simple:

```
ref Allocate(T type);
ref AllocateArray(T type, int size);
```

Of course, since memory managers are usually written in the language different from the language runtime is targeted by (e.g. Java targets JVM, yet HotSpot JVM is written in C++), the interface gets murkier. For example, such a call from Java program needs to transit into native VM code. Does it cost much? Probably. Does the memory manager have to cope with multiple threads begging for memory? For sure.

So to optimize this, we may instead allow threads to allocate the entire *blocks* of memory for their needs, and only transit to VM to get a new block. In Hotspot, these blocks are called Thread Local Allocation Buffers (TLABs), and there is a sophisticated machinery built to support them. Notice that TLABs are thread-local in the temporal sense, meaning they act like the buffers to accept current allocations. They still are parts of Java heap, the thread can still write the reference to a newly allocated object into the field outside of TLAB, etc.

All known OpenJDK GCs support TLAB allocation. This part of VM code is pretty well shared among them. All Hotspot compilers support TLAB allocation, so you would usually see the generated code for object allocation like this:

ASM

```
0x00007f3e6bb617cc: mov    0x60(%r15),%rax        ; TLAB "current"
0x00007f3e6bb617d0: mov    %rax,%r10              ; tmp = current
0x00007f3e6bb617d3: add    $0x10,%r10             ; tmp += 16 (object size)
0x00007f3e6bb617d7: cmp    0x70(%r15),%r10        ; tmp > tlab_size?
0x00007f3e6bb617db: jae    0x00007f3e6bb61807     ; TLAB is done, jump and request another one
0x00007f3e6bb617dd: mov    %r10,0x60(%r15)        ; current = tmp (TLAB is fine, alloc!)
0x00007f3e6bb617e1: prefetchnta 0xc0(%r10)        ; ...
0x00007f3e6bb617e9: movq   $0x1,(%rax)            ; store header to (obj+0)
0x00007f3e6bb617f0: movl   $0xf80001dd,0x8(%rax)  ; store klass to (obj+8)
0x00007f3e6bb617f7: mov    %r12d,0xc(%rax)        ; zero out the rest of the object
```

The allocation path is inlined in the generated code, and as such does not require calling into GC to allocate the object. If we are requesting to allocate the object that depletes the TLAB, or the objects is large enough to never fit into the TLAB, then we take a "slow path", and either satisfy the allocation there, or come back with a fresh TLAB. Notice how the most frequent "normal" path is *just* adding the object size to TLAB current cursor, and then moving on.

This is why this allocation mechanism is sometimes called *"pointer bump allocation"*. Pointer bump requires a contiguous chunk of memory to allocate to, though — which brings back the need for heap compaction. Notice how CMS does free-list allocation in "old" generation, thus enabling concurrent sweep, but it has compacting stop-the-world "young" collections, that benefit from pointer bump allocation! A much lower quantity of objects that survived the young collection would pay the cost of free list allocation.

For the sake of experiment, we can turn TLAB machinery off with `-XX:-UseTLAB`. Then, all allocations would take into the native method, like this:

SHELL

```
-   17.12%     0.00%  org.openjdk.All  perf-31615.map
   - 0x7faaa3b2d125
      - 16.59% OptoRuntime::new_instance_C
         - 11.49% InstanceKlass::allocate_instance
              2.33% BlahBlahBlahCollectedHeap::mem_allocate  <---- entry point to GC
              0.35% AllocTracer::send_allocation_outside_tlab_event
```

...but, as you would see below, it is usually a bad idea.

Experiment

As usual, let us try to construct an experiment to see TLAB allocation in action. Since the machinery is shared by all GC implementations there, in makes sense to minimize the impact of other parts of runtime by using experimental Epsilon GC (http://openjdk.java.net/jeps/8174901). Indeed, it implements allocation *only*, and thus provides a good research vessel for this work.

Quickly drafting a workload: allocating 50M objects (why not?), and running with JMH with SingleShot mode to get statistics and profiling for free. You could do this with a standalone test too, but SingleShot is just too convenient here.

JAVA

```
@Warmup(iterations = 3)
@Measurement(iterations = 3)
@Fork(3)
@BenchmarkMode(Mode.SingleShotTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class AllocArray {
    @Benchmark
    public Object test() {
        final int size = 50_000_000;
        Object[] objects = new Object[size];
        for (int c = 0; c < size; c++) {
            objects[c] = new Object();
        }
        return objects;
    }
}
```

This test allocates 50M objects in a single thread. This is empirically selected to make 20 GB heap last for at least 6 iterations, as we would see next. The experimental `-XX:EpsilonTLABSize` option is used to control the TLAB sizing exactly. Other OpenJDK GCs share the *adaptive* TLAB sizing (https://blogs.oracle.com/daviddetlefs/entry/tlab_sizing_an_annoying_little) policy that selects the sizes based on allocation pressure and other concerns. For our performance tests it is easier to nail the TLAB size.

Without further ado, these are the results:

SHELL

```
Benchmark                     Mode  Cnt     Score    Error   Units

# Times, lower is better                                    # TLAB size
AllocArray.test                 ss    9   548.462 ±  6.989   ms/op  #      1 KB
AllocArray.test                 ss    9   268.037 ± 10.966   ms/op  #      4 KB
AllocArray.test                 ss    9   230.726 ±  4.119   ms/op  #     16 KB
AllocArray.test                 ss    9   223.075 ±  2.267   ms/op  #    256 KB
AllocArray.test                 ss    9   225.404 ± 17.080   ms/op  #   1024 KB

# Allocation rates, higher is better
AllocArray.test:·gc.alloc.rate  ss    9  1816.094 ± 13.681  MB/sec  #      1 KB
AllocArray.test:·gc.alloc.rate  ss    9  2481.909 ± 35.566  MB/sec  #      4 KB
AllocArray.test:·gc.alloc.rate  ss    9  2608.336 ± 14.693  MB/sec  #     16 KB
AllocArray.test:·gc.alloc.rate  ss    9  2635.857 ±  8.229  MB/sec  #    256 KB
AllocArray.test:·gc.alloc.rate  ss    9  2627.845 ± 60.514  MB/sec  #   1024 KB
```

Notice how we are able to pull off 2.5 GB/sec allocation rate in a single thread. With 16 byte objects, this means 160 million objects per second. In multi-threaded workloads, the allocation rates may reach tens of gigabytes per second. Of course, once TLAB size gets smaller, both the allocation costs go up, and allocation rate goes down. Unfortunately, we cannot make TLABs lower than 1 KB, because Hotspot mechanics needs some space wasted there, but we can turn off TLAB machinery completely, to see the performance impact:

SHELL

```
Benchmark                     Mode  Cnt     Score    Error    Units

# -XX:-UseTLAB
AllocArray.test                 ss    9  2784.988 ± 18.925   ms/op
AllocArray.test:·gc.alloc.rate  ss    9   580.533 ±  3.342  MB/sec
```

Whoa, the allocation rate goes down at least 5x, and time to execute goes up 10x! And this is not even starting to touch what a collector has to do when multiple threads are asking for memory (probably contended atomics), or if it needs to look up where to allocate the memory from (try to allocate fast from free lists!). For Epsilon, the allocation path in GC is a single compare-and-set

— because it issues the memory blocks by pointer-bumps itself. If you add one additional thread — totaling just two running threads — without TLABs, everything goes downhill:

```
SHELL
Benchmark                              Mode  Cnt           Score       Error   Units

# TLAB = 4M (default for Epsilon)
AllocArray.test                          ss    9         407.729 ±     7.672   ms/op
AllocArray.test:·gc.alloc.rate           ss    9        4190.670 ±    45.909  MB/sec

# -XX:-UseTLAB
AllocArray.test                          ss    9        8490.585 ±   410.518   ms/op
AllocArray.test:·gc.alloc.rate           ss    9         422.960 ±    19.320  MB/sec
```

This is 20x performance hit now. Project the slowness you would experience with larger thread counts!

Observations

TLABs are the workhorses of allocation machinery: they unload the concurrency bottlenecks of the allocators, provide a cheap allocation path, and improve performance all around. It is funny to consider that having TLABs is the way to experience more frequent GC pauses, just because the allocation is so damn cheap! In reverse, having no fast allocation path in any memory manager implementation for sure hides the memory reclamation performance problems. When comparing the memory managers, do understand both parts of the story, and how they relate to each other.

JVM Anatomy Quark #5: TLABs and Heap Parsability

What the heck are those int[] arrays in my heap dumps?!

Question

Have you ever encountered the large `int[]` arrays that cannot be accounted for? Those that are seemingly allocated nowhere, but still consuming heap? Those that have some garbage-looking data in them?

Theory

In GC theory, there is an important property that good collectors try to maintain, *heap parsability*, that is, shaping the heap in such a way it could be parsed for objects, fields, etc. without complicated metadata supporting it. In OpenJDK, for example, many introspection tasks walk the heap with a simple loop like this:

```
C
HeapWord* cur = heap_start;
while (cur < heap_used) {
  object o = (object)cur;
  do_object(o);
  cur = cur + o->size();
}
```

That's it! If heap is parsable, then we can assume there is a contiguous stream of objects from the start to the allocated end. This is not, strictly speaking, a required property, but it makes GC implementation, testing and debugging much easier.

Enter Thread Local Allocation Buffer (TLAB) machinery: now, each thread has its own TLAB it can currently allocate to. From the GC perspective, this means the *entire* TLAB is claimed. GC cannot easily know what threads are up to there: are they in the middle of bumping the TLAB cursor? What is the value for TLAB cursor anyway? It is possible that a thread just keeps it somewhere in the register (in OpenJDK, it is not) and never shows it to external observers. So, there is a problem: outsiders do **not** know what exactly happens in TLABs.

We might want to stop the threads to avoid their TLAB mutation, and then traverse the heap accurately, checking if what we are walking right now is the part of some TLAB. But there is a more convenient *trick*: why don't we make heap parsable by inserting *filler objects*? That is, if we have:

```
...........|====================           ]...........
           ^                    ^           ^
       TLAB start         TLAB used    TLAB end
```

...we can stop the threads, and *ask them* to allocate a dummy object in the rest of the TLAB to make their part of heap parsable:

```
...........|====================!!!!!!!!!!]...........
           ^                    ^           ^
       TLAB start         TLAB used    TLAB end
```

What is a good candidate for a dummy object? Of course, something that has variable length. Why not `int[]` array? Note that "putting" the object like this only amounts to putting out the array header, and letting heap mechanics to work out the rest, jumping over its contents. Once thread resumes allocating in TLAB, it can just overwrite whatever filler we allocated, like nothing happened.

The same thing, by the way, simplifies *sweeping* the heap. If we remove (sweep out) the object, it is convenient to place a filler in its place to keep heap walking routines happy.

Experiment

Can we see it in action? Of course we can. What we want is to start lots of threads that would claim some TLABs of their own, and one loner thread what will exhaust the Java heap, crashing with OutOfMemoryException, which we will use as the trigger for a heap dump.

Workload like this is fine:

```java
import java.util.*;
import java.util.concurrent.*;

public class Fillers {
  public static void main(String... args) throws Exception {
    final int TRAKTORISTOV = 300;
    CountDownLatch cdl = new CountDownLatch(TRAKTORISTOV);
    for (int t = 0 ; t < TRAKTORISTOV; t++) {
      new Thread(() -> allocateAndWait(cdl)).start();
    }
    cdl.await();
    List<Object> l = new ArrayList<>();
    new Thread(() -> allocateAndDie(l)).start();
  }

  public static void allocateAndWait(CountDownLatch cdl) {
    Object o = new Object();  // Request a TLAB
    cdl.countDown();
    while (true) {
      try {
        Thread.sleep(1000);
      } catch (Exception e) {
        break;
      }
    }
    System.out.println(o); // Use the object
  }

  public static void allocateAndDie(Collection<Object> c) {
    while (true) {
      c.add(new Object());
    }
  }
}
```

Now, in order to get the predictable TLAB sizes, we can again use Epsilon GC (http://openjdk.java.net/jeps/8174901). Running with `-Xmx1G -Xms1G -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC -XX:+HeapDumpOnOutOfMemoryError` quickly fails and produces the heap dump for us.

Opening this heap dump in Eclipse Memory Analyzer (MAT) (http://www.eclipse.org/mat/) — I like that tool a lot — we can see this class histogram:

```
Class Name                                 |   Objects | Shallow Heap |
-------------------------------------------------------------------------
                                           |           |              |
int[]                                      |     1,099 |  814,643,272 |
java.lang.Object                           | 9,181,912 |  146,910,592 |
java.lang.Object[]                         |     1,521 |  110,855,376 |
byte[]                                     |     6,928 |      348,896 |
java.lang.String                           |     5,840 |      140,160 |
java.util.HashMap$Node                     |     1,696 |       54,272 |
java.util.concurrent.ConcurrentHashMap$Node|     1,331 |       42,592 |
java.util.HashMap$Node[]                   |       413 |       42,032 |
char[]                                     |        50 |       37,432 |
-------------------------------------------------------------------------
```

See how `int[]` is the dominating heap consumer! These are our filler objects. Granted, this experiment has a few caveats.

First, we configured Epsilon to have static TLAB sizes. A high-performance collector would instead make the *adaptive* TLAB sizing decisions, which would minimize the heap slack when a thread had allocated a few objects, but still sits on troves of TLAB memory. This is one of the reasons why you don't want to issue large TLABs without thinking twice. Still, it is possible to observe filler objects when an actively allocating thread has the large TLAB issued to it, and it is only half way there in filling it up with real data.

Second, we have configured MAT to show us unreachable objects. These filler objects are, by definition, unreachable. Their presence in the heap dumps is just a side effect of heap dumping using heap parsability property to walk the heap. These objects do not *really* exist, and a good heap dump analyzer tool will happily filter them out for you — this might be one of the reasons why a crashing 1G heap dump has only, say, 900 MB worth of objects in it.

Observations

Having TLABs is fun. Having heap parsability is fun too. Combining both is even funnier, and sometimes leaks out internal trickery. If you see a surprising behavior from any runtime, you might be looking at some clever trick!

JVM Anatomy Quark #6: New Object Stages

What the heck are those int[] arrays in my heap dumps?!

Question

So I've heard allocation is not initialization. But Java has constructors! Are they allocating? Or initializing?

Theory

If you open the GC Handbook (http://gchandbook.org/), it would tell you that creating a new object usually entails three phases:

1. **Allocation**. That is, figuring out what part of process space to get for instance data.
2. **System initialization**. That is, the initialization required by the language. In C, no initialization is required for `new`-ly allocated objects. In Java, system initialization is required for all objects: it is expected to see only default values for a newly created object, it is expected to see all headers intact, etc.
3. **Secondary (user) initialization**. That is, running whatever instance initializers and constructors associated with that object type.

We have covered (1) in previous note, "TLAB Allocation". It is time to see what initialization is doing. If you are familiar with Java bytecode, then you will know that "new" code path takes *several* bytecode instructions. For example, this:

```
JAVA
public Object t() {
  return new Object();
}
```

...compiles into:

```
JAVA
  public java.lang.Object t();
    descriptor: ()Ljava/lang/Object;
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: new           #4                  // class java/lang/Object
         3: dup
         4: invokespecial #1                  // Method java/lang/Object."<init>":()V
         7: areturn
```

It *feels* like `new` is doing allocation and system initialization, while invoking the constructor (`<init>`) does user init. But, smart runtimes can coalesce initialization when they know nobody will call the bluff — for example, by observing the object before the constructor finished. Can we show if this initialization subsuming works for Hotspot?

Experiment

Sure we can. To do this, we just want to take a test that initialized two variants of single- `int`-field class:

```java
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(value = 3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class UserInit {

    @Benchmark
    public Object init() {
        return new Init(42);
    }

    @Benchmark
    public Object initLeaky() {
        return new InitLeaky(42);
    }

    static class Init {
        private int x;
        public Init(int x) {
            this.x = x;
        }
    }

    static class InitLeaky {
        private int x;
        public InitLeaky(int x) {
            doSomething();
            this.x = x;
        }

        @CompilerControl(CompilerControl.Mode.DONT_INLINE)
        void doSomething() {
            // intentionally left blank
        }
    }
}
```

This test is specially crafted to forbid inlining of empty `doSomething()`, forcing optimizers to assume that something accesses `x` downstream. In other words, it would effectively *leak* the object to some external code — because we cannot say if code in `doSomething()` actually leaks it.

It is better to run with `-XX:+UseParallelGC -XX:-TieredCompilation -XX:-UseBiasedLocking` to make generated code more understandable — this is an educational exercise anyway. JMH's `-prof perfasm` is perfect to dump the generated code for these tests.

This is the `Init` case:

```asm
                                                  ; ------- allocation ----------
0x00007efdc466d4cc: mov    0x60(%r15),%rax          ; TLAB allocation below
0x00007efdc466d4d0: mov    %rax,%r10
0x00007efdc466d4d3: add    $0x10,%r10
0x00007efdc466d4d7: cmp    0x70(%r15),%r10
0x00007efdc466d4db: jae    0x00007efdc466d50a
0x00007efdc466d4dd: mov    %r10,0x60(%r15)
0x00007efdc466d4e1: prefetchnta 0xc0(%r10)
                                                  ; ------- /allocation ---------
                                                  ; ------- system init ---------
0x00007efdc466d4e9: movq   $0x1,(%rax)              ; put mark word header
0x00007efdc466d4f0: movl   $0xf8021bc4,0x8(%rax)    ; put class word header
                                                  ; ...... system/user init .....
0x00007efdc466d4f7: movl   $0x2a,0xc(%rax)          ; x = 42.
                                                  ; -------- /user init ---------
```

You can see TLAB allocation, initialization of object metadata, and then coalesced system+user initialization of the field. This changes quite a bit for `InitLeaky` case:

```asm
                                                ; ------- allocation ----------
0x00007fc69571bf4c: mov    0x60(%r15),%rax
0x00007fc69571bf50: mov    %rax,%r10
0x00007fc69571bf53: add    $0x10,%r10
0x00007fc69571bf57: cmp    0x70(%r15),%r10
0x00007fc69571bf5b: jae    0x00007fc69571bf9e
0x00007fc69571bf5d: mov    %r10,0x60(%r15)
0x00007fc69571bf61: prefetchnta 0xc0(%r10)
                                                ; ------- /allocation ---------
                                                ; ------- system init ---------
0x00007fc69571bf69: movq   $0x1,(%rax)            ; put mark word header
0x00007fc69571bf70: movl   $0xf8021bc4,0x8(%rax)  ; put class word header
0x00007fc69571bf77: mov    %r12d,0xc(%rax)        ; x = 0 (%r12 happens to hold 0)
                                                ; ------- /system init --------
                                                ; -------- user init ----------
0x00007fc69571bf7b: mov    %rax,%rbp
0x00007fc69571bf7e: mov    %rbp,%rsi
0x00007fc69571bf81: xchg   %ax,%ax
0x00007fc69571bf83: callq  0x00007fc68e269be0     ; call doSomething()
0x00007fc69571bf88: movl   $0x2a,0xc(%rbp)        ; x = 42
                                                ; ------ /user init ------
```

Here, since optimizers cannot figure if the value of `x` is needed, they have to assume the worst, and perform system initialization first, and only then finish up user init.

Observations

While textbook definition is sound, and bytecode reflects the same definition, the optimizers may do magic undercover to optimize performance, as long as it would not yield surprising behaviors. From compiler perspective, this is a trivial optimization, but from the conceptual point of view it operates over the theoretical "staging" boundaries.

JVM Anatomy Quark #7: Initialization Costs

What time is it? Payback time.

Question

What is so expensive about creating new objects? What defines the object instantiation performance?

Theory

If you look closely at object instantiation paths on larger objects, you will inevitably wonder how the different parts scale, and what is the actual bottleneck in real world cases. We have seen that TLAB allocation seems very efficient, and that system initialization can be coalesced with user initialization. But in the end, we still have to do writes to memory — can we tell how much it costs?

Experiment

One primitive that can tell us the initialization part of story is plain old Java array. It needs to be initialized too, and its length is variable, which lets us see what is going on at different sizes. With that in mind, let us construct this benchmark:

```java
import org.openjdk.jmh.annotations.*;

@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(value = 3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class UA {
    @Param({"1", "10", "100", "1000", "10000", "100000"})
    int size;

    @Benchmark
    public byte[] java() {
        return new byte[size];
    }
}
```

It makes sense to run with the latest JDK 9 EA (the follow up test below requires it), with `-XX:+UseParallelOldGC` to minimize GC overhead, and `-Xmx20g -Xms20g -Xmn18g` to keep most of the heap for these new allocations. Without further delay, this is what you will get on a good development desktop (like mine i7-4790K, 4.0 GHz, Linux x86_64), when all 8 hardware threads are busy:

```shell
Benchmark                  (size)  Mode  Cnt       Score       Error   Units

# Time to allocate
UA.java                         1  avgt   15      20.307 ±     4.532   ns/op
UA.java                        10  avgt   15      26.657 ±     6.072   ns/op
UA.java                       100  avgt   15     106.632 ±    34.742   ns/op
UA.java                      1000  avgt   15     681.176 ±   124.980   ns/op
UA.java                     10000  avgt   15    4576.433 ±   909.956   ns/op
UA.java                    100000  avgt   15   44881.095 ± 13765.440   ns/op

# Allocation rate
UA.java:·gc.alloc.rate          1  avgt   15    6228.153 ±  1059.385  MB/sec
UA.java:·gc.alloc.rate         10  avgt   15    6335.809 ±   986.395  MB/sec
UA.java:·gc.alloc.rate        100  avgt   15    6126.333 ±  1354.964  MB/sec
UA.java:·gc.alloc.rate       1000  avgt   15    7772.263 ±  1263.453  MB/sec
UA.java:·gc.alloc.rate      10000  avgt   15   11518.422 ±  2155.516  MB/sec
UA.java:·gc.alloc.rate     100000  avgt   15   12039.594 ±  2724.242  MB/sec
```

One can see that the allocation takes around 20 ns (single-threaded is much lower, but this average is skewed by hyper-threads), which goes gradually up to 40 us for 100K array. If you look at the allocation rate, then you will see it saturates at around 12 GB/sec. These experiments form the basis for other performance tests, by the way: it is important to understand the order of memory bandwidth / allocation rate you can pull off on a particular machine.

Can we see what code dominates the execution time? Of course we can, again, with JMH's `-prof perfasm`. For `-p size=100000`, it will print out the hottest code like this:

```
ASM
             0x00007f1f094f650b: movq   $0x1,(%rdx)              ; store mark word
  0.00%      0x00007f1f094f6512: prefetchnta 0xc0(%r9)
  0.64%      0x00007f1f094f651a: movl   $0xf80000f5,0x8(%rdx)    ; store klass word
  0.02%      0x00007f1f094f6521: mov    %r11d,0xc(%rdx)          ; store array length
             0x00007f1f094f6525: prefetchnta 0x100(%r9)
  0.05%      0x00007f1f094f652d: prefetchnta 0x140(%r9)
  0.07%      0x00007f1f094f6535: prefetchnta 0x180(%r9)
  0.09%      0x00007f1f094f653d: shr    $0x3,%rcx
  0.00%      0x00007f1f094f6541: add    $0xfffffffffffffffe,%rcx
             0x00007f1f094f6545: xor    %rax,%rax
             0x00007f1f094f6548: cmp    $0x8,%rcx
       ╭     0x00007f1f094f654c: jg     0x00007f1f094f655e       ; large enough? jump
       │     0x00007f1f094f654e: dec    %rcx
       │╭    0x00007f1f094f6551: js     0x00007f1f094f6565       ; zero length? jump
       ││↗   0x00007f1f094f6553: mov    %rax,(%rdi,%rcx,8)       ; small loop init
       │││   0x00007f1f094f6557: dec    %rcx
       ││╰   0x00007f1f094f655a: jge    0x00007f1f094f6553
       ││ ╭  0x00007f1f094f655c: jmp    0x00007f1f094f6565
       ↘│ │  0x00007f1f094f655e: shl    $0x3,%rcx
  89.12%  │ │  0x00007f1f094f6562: rep rex.W stos %al,%es:(%rdi)   ; large loop init
   0.20%  ↘ ↘  0x00007f1f094f6565: mov    %r8,(%rsp)
```

You may find this code shape familiar from "TLAB Allocation" and "New Object Stages" issues in the series. What is interesting here, is that we have to initialize much larger chunk of array data. For this reason, we see the inlined `rep stos` sequence on x86 — which repeatedly stores the given number of given byte — and seems most effective on recent x86-s. One can spy with a little eye that there is also an initializing loop for small arrays (smaller or equal of 8 elements) — `rep stos` has upfront startup costs, so smaller loop is beneficial there.

This example shows that for large objects/arrays initialization costs would dominate the performance. If objects/arrays are small, then writes of metadata (headers, array lengths) would dominate. Small array case would not be significantly different from small object case.

Can we guesstimate what would the performance be, if we somehow avoid initialization? The compilers are routinely coalesce system and user initialization, but can we get *no initialization at all*? Granted, it would serve no practical purpose to have uninitialized object, because we will still fill it with user data later — but synthetic tests are fun, am I right?

Turns out, there is an `Unsafe` method that allocates uninitialized arrays, so we may use it to see the benefits. `Unsafe` is not Java, it does not play by Java rules, and it sometimes even defies the JVM rules. It is not the API for the public use, and it is there for internal JDK use and JDK-VM interoperability. You can't have it, it is not guaranteed to work for you, and it can go away, crash and burn without telling you.

Still, we can use it in synthetics, like this:

```java
import jdk.internal.misc.Unsafe;
import org.openjdk.jmh.annotations.*;

@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(value = 3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class UA {
    static Unsafe U;

    static {
        try {
            Field field = Unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            U = (Unsafe) field.get(null);
        } catch (Exception e) {
            throw new IllegalStateException(e);
        }
    }

    @Param({"1", "10", "100", "1000", "10000", "100000"})
    int size;

    @Benchmark
    public byte[] unsafe() {
        return (byte[]) U.allocateUninitializedArray(byte.class, size);
    }
}
```

Why don't we run it? Okay:

```shell
Benchmark                  (size)  Mode  Cnt        Score       Error   Units
UA.unsafe                       1  avgt   15       19.766 ±     4.002   ns/op
UA.unsafe                      10  avgt   15       27.486 ±     7.005   ns/op
UA.unsafe                     100  avgt   15       80.040 ±    15.754   ns/op
UA.unsafe                    1000  avgt   15      156.041 ±     0.552   ns/op
UA.unsafe                   10000  avgt   15      162.384 ±     1.448   ns/op
UA.unsafe                  100000  avgt   15      309.769 ±     2.819   ns/op

UA.unsafe:·gc.alloc.rate        1  avgt   15     6359.987 ±   928.472  MB/sec
UA.unsafe:·gc.alloc.rate       10  avgt   15     6193.103 ±  1160.353  MB/sec
UA.unsafe:·gc.alloc.rate      100  avgt   15     7855.147 ±  1313.314  MB/sec
UA.unsafe:·gc.alloc.rate     1000  avgt   15    33171.384 ±   153.645  MB/sec
UA.unsafe:·gc.alloc.rate    10000  avgt   15   315740.299 ±  3678.459  MB/sec
UA.unsafe:·gc.alloc.rate   100000  avgt   15  1650860.763 ± 14498.920  MB/sec
```

Holy crap! 100K arrays are allocated with 1.6 **terabytes per second** rate. Can we see where we spend time *now*?

```asm
          0x00007f65fd722c74: prefetchnta 0xc0(%r11)
 66.06%   0x00007f65fd722c7c: movq   $0x1,(%rax)           ; store mark word
  0.40%   0x00007f65fd722c83: prefetchnta 0x100(%r11)
  4.43%   0x00007f65fd722c8b: movl   $0xf80000f5,0x8(%rax) ; store class word
  0.01%   0x00007f65fd722c92: mov    %edx,0xc(%rax)        ; store array length
          0x00007f65fd722c95: prefetchnta 0x140(%r11)
  5.18%   0x00007f65fd722c9d: prefetchnta 0x180(%r11)
  4.99%   0x00007f65fd722ca5: mov    %r8,0x40(%rsp)
          0x00007f65fd722caa: mov    %rax,%rdx
```

Oh yeah, touching the memory to push the metadata out. The cycles accounting is skewed towards the prefetches, because they are now paying the most of the cost for pre-accessing memory for upcoming writes.

One might wonder what toll that exerts on GC, and the answer is: not that much! The objects are almost all dead, and so any mark-(copy|sweep|compact) GC would breeze through the workload like this. When objects start *surviving* at TB/sec rate, the picture gets interesting. Some GC guys I know call these things "impossible workloads" — because they both impossible in reality, and impossible to deal with. Try to drink from a firehose to learn this in practice.

Anyhow, we can see that with pure allocations, GCs are surviving fine. With the same workload, we can see what are the apparent application pauses, by using JMH's `-prof pauses` profiler. It runs a high-priority thread and records the perceived pauses:

```
SHELL
Benchmark                  (size)  Mode  Cnt    Score   Error  Units
UA.unsafe                  100000  avgt    5  315.732 ± 5.133  ns/op
UA.unsafe:·pauses          100000  avgt   84  537.018             ms
UA.unsafe:·pauses.avg      100000  avgt          6.393             ms
UA.unsafe:·pauses.count    100000  avgt         84.000              #
UA.unsafe:·pauses.p0.00    100000  avgt          2.560             ms
UA.unsafe:·pauses.p0.50    100000  avgt          6.148             ms
UA.unsafe:·pauses.p0.90    100000  avgt          9.642             ms
UA.unsafe:·pauses.p0.95    100000  avgt          9.802             ms
UA.unsafe:·pauses.p0.99    100000  avgt         14.418             ms
UA.unsafe:·pauses.p0.999   100000  avgt         14.418             ms
UA.unsafe:·pauses.p0.9999  100000  avgt         14.418             ms
UA.unsafe:·pauses.p1.00    100000  avgt         14.418             ms
```

So, it had detected around 84 pauses, the longest is 14 ms, while the average is about 6 ms. Profilers like these are inherently imprecise, because they contend on CPUs with workload threads, they get into the mercy of OS scheduler, etc.

In many cases, it is better to enable the JVM to tell when it stops the application threads. This can be done with JMH's `-prof safepoints` profiler, which records the "safe point", "stop the world" events when all application threads are stopped, and VM does its work. GC pauses are naturally the subset of safepoint events.

```
SHELL
Benchmark                            (size)  Mode  Cnt     Score    Error  Units
UA.unsafe                            100000  avgt    5   328.247 ± 34.450  ns/op
UA.unsafe:·safepoints.interval       100000  avgt       5043.000              ms
UA.unsafe:·safepoints.pause          100000  avgt  639   617.796              ms
UA.unsafe:·safepoints.pause.avg      100000  avgt          0.967              ms
UA.unsafe:·safepoints.pause.count    100000  avgt        639.000               #
UA.unsafe:·safepoints.pause.p0.00    100000  avgt          0.433              ms
UA.unsafe:·safepoints.pause.p0.50    100000  avgt          0.627              ms
UA.unsafe:·safepoints.pause.p0.90    100000  avgt          2.150              ms
UA.unsafe:·safepoints.pause.p0.95    100000  avgt          2.241              ms
UA.unsafe:·safepoints.pause.p0.99    100000  avgt          2.979              ms
UA.unsafe:·safepoints.pause.p0.999   100000  avgt         12.599              ms
UA.unsafe:·safepoints.pause.p0.9999  100000  avgt         12.599              ms
UA.unsafe:·safepoints.pause.p1.00    100000  avgt         12.599              ms
```

See, this profiler records 639 safepoints, with average time of less than 1 ms, and the largest time of 12 ms. Not bad, taking into account 1.6 TB/sec allocation rate!

Observations

The initialization costs are the significant part of object/array instantiation. With TLAB allocation, the object/array creation speed is largely dominated by either metadata writes (for smaller things), or the content initialization (for larger things). The allocation rate alone is not *always* a good predictor of performance, as you can manage huge allocation rates if you runtime pulls weird tricks on you.

JVM Anatomy Quark #8: Local Variable Reachability

Question

The references stored in local variables are collected when they go out of scope. Right?

Theory

This has deep roots in C/C++ experience of many programmers, because it is said in the scripture:

> 1. *Local objects explicitly declared auto or register or not explicitly declared static or extern have automatic storage duration. The storage for these objects lasts until the block in which they are created exits.*
> 2. *[Note: these objects are initialized and destroyed as described in 6.7.]*
> 3. *If a named automatic object has initialization or a destructor with side effects, it shall not be destroyed before the end of its block, nor shall it be eliminated as an optimization even if it appears to be unused, except that a class object or its copy may be eliminated as specified in 12.8.*
>
> — *C++98 Standard*
> *3.7.2 "Automatic storage duration"*

This is a very useful language property, because it allows to bind the object lifetime to the syntactic code block. Which allows doing, for example, this:

```C++
void method() {
  ...something...

  {
     MutexLocker ml(mutex);
     ...something under the lock...
  } // ~MutexLocker unlocks

  ...something else...
}
```

Coming from a C++ land, you would naturally expect the same property to hold in Java. There are no destructors, but there are ways to detect if object is deemed unreachable, and act accordingly, e.g. via soft/weak/phantom references or finalizers. *However*, the syntactic code blocks in Java do not act that way. See for example:

> *"Optimizing transformations of a program can be designed that reduce the number of objects that are reachable to be less than those which would naively be considered reachable. For example, a Java compiler or code generator may choose to set a variable or parameter that will no longer be used to null to cause the storage for such an object to be potentially reclaimable sooner."*
>
> — *Java Language Specification 8*
> *12.6.1 "Implementing Finalization"*

Does this really matter?

Experiment

This difference is fairly easy to demonstrate with the experiment. Take this class as example:

```java
public class LocalFinalize {
    ...
    private static volatile boolean flag;

    public static void pass() {
        MyHook h1 = new MyHook();
        MyHook h2 = new MyHook();

        while (flag) {
            // spin
        }

        h1.log();
    }

    public static class MyHook {
       public MyHook() {
           System.out.println("Created " + this);
       }

       public void log() {
           System.out.println("Alive " + this);
       }

       @Override
       protected void finalize() throws Throwable {
           System.out.println("Finalized " + this);
       }
    }
}
```

Naively, one could presume that the lifetime of `h2` extends to the end of the `pass` method. And since there is a waiting loop in the middle that might not terminate with `flag` set to `true`, the object would never be considered for finalization.

The caveat is that we want the method to be compiled to see the interesting behavior. To force this, we can do two passes: first pass will enter the method, spin for a while, and then exit. This will compile the method fine, because the loop body would be executed many times, and that will trigger compilation. Then we can enter the second time, but never leave the loop again.

Something like this will do:

```java
public static void arm() {
    new Thread(() -> {
        try {
             Thread.sleep(5000);
             flag = false;
        } catch (Throwable t) {}
    }).start();
}

public static void main(String... args) throws InterruptedException {
    System.out.println("Pass 1");
    arm();
    flag = true;
    pass();

    System.out.println("Wait for pass 1 finalization");
    Thread.sleep(10000);

    System.out.println("Pass 2");
    flag = true;
    pass();
}
```

We would also like a background thread forcing GC repeatedly, to cause finalization. Okay, the setup is done (full source here), let's run:

```
$ java -version
java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)

$ java LocalFinalize
Pass 1
Created LocalFinalize$MyHook@816f27d     # h1 created
Created LocalFinalize$MyHook@87aac27     # h2 created
Alive LocalFinalize$MyHook@816f27d       # h1.log called

Wait for pass 1 finalization
Finalized LocalFinalize$MyHook@87aac27   # h1 finalized
Finalized LocalFinalize$MyHook@816f27d   # h2 finalized

Pass 2
Created LocalFinalize$MyHook@3e3abc88    # h1 created
Created LocalFinalize$MyHook@6ce253f1    # h2 created
Finalized LocalFinalize$MyHook@6ce253f1  # h2 finalized (!)
```

Oops. That happened because the optimizing compiler knew the last use of `h2` was right after the allocation. Therefore, when communicating what live variables are present — later during the loop execution — to the garbage collector, it does not consider `h2` live anymore. Therefore, garbage collector treats that `MyHook` instance as dead and runs its finalization. Since the `h1` use is later after the loop, it is considered reachable, and finalization is silent.

This is actually a great feature, because it lets GC can reclaim huge buffers allocated locally without requiring to exit the method, e.g.:

JAVA

```
void processAndWait() {
  byte[] buf = new byte[1024 * 1024];
  writeToBuf(buf);
  processBuf(buf); // last use!
  waitForTheDeathOfUniverse(); // oops
}
```

Going Deeper

In fact, you can see the technicalities of this in various disassemblies. First, the bytecode disassembly does not even mention the local variable data at all, and the slot 1 where the `h2` instance is stored is left alone until the end of the method:

```
$ javap -c -v -p LocalFinalize.class

  public static void pass();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=0
         0: new           #17                 // class LocalFinalize$MyHook
         3: dup
         4: invokespecial #18                 // Method LocalFinalize$MyHook."<init>":()V
         7: astore_0
         8: new           #17                 // class LocalFinalize$MyHook
        11: dup
        12: invokespecial #18                 // Method LocalFinalize$MyHook."<init>":()V
        15: astore_1
        16: getstatic     #10                 // Field flag:Z
        19: ifeq          25
        22: goto          16
        25: aload_0
        26: invokevirtual #19                 // Method LocalFinalize$MyHook.log:()V
        29: return
```

Compiling with debug data (`javac -g`) yields Local Variable Table (LVT), where the lifetime of local variable "seems" to extend by the end of the method:

```
  public static void pass();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=0
         0: new           #17                 // class LocalFinalize$MyHook
         3: dup
         4: invokespecial #18                 // Method LocalFinalize$MyHook."<init>":()V
         7: astore_0
         8: new           #17                 // class LocalFinalize$MyHook
        11: dup
        12: invokespecial #18                 // Method LocalFinalize$MyHook."<init>":()V
        15: astore_1
        16: getstatic     #10                 // Field flag:Z
        19: ifeq          25
        22: goto          16
        25: aload_0
        26: invokevirtual #19                 // Method LocalFinalize$MyHook.log:()V
        29: return
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            8      22     0    h1   LLocalFinalize$MyHook; //  8 + 22 = 30
           16      14     1    h2   LLocalFinalize$MyHook; // 16 + 14 = 30
```

This might confuse people into believing the reachability is actually extended to the end of the method, because "scope" is thought to be defined by LVT. But it is not, because optimizers could actually figure out that local is not used further, and optimize accordingly. In our current test, this happens (in pseudocode):

JAVA

```java
public static void pass() {
    MyHook h1 = new MyHook();
    MyHook h2 = new MyHook();

    while (flag) {
        // spin
        // <gc safe point here>
        // Here, compiled code knows what references are present in machine
        // registers and on stack. By then, the "h2" is already past its last use,
        // and this map has no evidence of "h2". Therefore, GC treats it as dead.
    }

    h1.log();
}
```

This is somewhat visible in `-XX:+PrintAssembly` output:

ASM

```asm
data16 data16 xchg %ax,%ax      # loop alignment
                                # output would also say:
                                # ImmutableOopMap{r10=Oop rbp=Oop}
LOOP:
  test   %eax,0x15ae2bca(%rip)  # safepoint poll, switch to GC can happen here
  movzbl 0x70(%r10),%r8d        # get this.flag
  test   %r8d,%r8d              # check flag and loop back
  jne    LOOP

...
```

`ImmutableOopMap{r10=Oop rbp=Oop}` basically says that `%r10` and `%rbp` hold the "ordinary object pointers". `%r10` holds `this` — see how we read `flag` off it, and `%rbp` holds the reference to `h1` that would be used later. Reference to `h2` is missing here. If GC happens during the loop, the thread would block when doing the safepoint poll, and at that time the runtime would know exactly what registers to care for, with the help of this map.

Alternatives

Extending the reachability of the object stored in local variable to the given program point can be done by using that local variable later. However, that is seldom easy to do without observable side effects. For example, "just" calling the method and passing that local variable is not enough, because the method might get inlined, and the same optimization kicks in. Since Java 9,

there is `java.lang.ref.Reference::reachabilityFence` (http://download.java.net/java/jdk9/docs/api/java/lang/ref/Reference.html#reachabilityFence-java.lang.Object-) method that provides required semantics.

If you "just" want to have C++ like "release on block exit" construct — to do something when leaving the block — then `try-finally` is your friend in Java.

Observations

Reachability for Java local variables is not defined by syntactic blocks, it is *at least* to the last use, and may be *exactly* to the last use. Using the mechanisms that notify when some object becomes unreachable (finalizers, weak/soft/phantom references) may fall victim of "early" detection while execution had not yet left the method/block it was reachable from first.

JVM Anatomy Quark #9: JNI Critical and GC Locker

Question

How does JNI `Get*Critical` cooperate with GC? What is GC Locker?

Theory

If you are familiar with JNI, you know there are two sets of methods that can get you the array contents. There is `Get<PrimitiveType>Array*` family of methods, and then there are these fellas (http://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html#GetPrimitiveArrayCritical_ReleasePrimitiveArrayCritical):

```
void * GetPrimitiveArrayCritical(JNIEnv *env, jarray array, jboolean *isCopy);
void ReleasePrimitiveArrayCritical(JNIEnv *env, jarray array, void *carray, jint mode);
```

> *The semantics of these two functions are very similar to the existing Get/Release*ArrayElements functions. If possible, the VM returns a pointer to the primitive array; otherwise, a copy is made. However, there are significant restrictions on how these functions can be used.*
>
> — *JNI Guide*
> *Chapter 4: JNI Functions*

The benefit for these are obvious: instead of providing you with the copy of the Java array, VM may choose to return a direct pointer, thus improving performance. That obviously comes with caveats, that are listed further down:

> *After calling GetPrimitiveArrayCritical, the native code should not run for an extended period of time before it calls ReleasePrimitiveArrayCritical. We must treat the code inside this pair of functions as running in a "critical region." Inside a critical region, native code must not call other JNI functions, or any system call that may cause the current thread to block and wait for another Java thread. (For example, the current thread must not call read on a stream being written by another Java thread.)*
>
> *These restrictions make it more likely that the native code will obtain an uncopied version of the array, even if the VM does not support pinning. For example, a VM may temporarily disable garbage collection when the native code is holding a pointer to an array obtained via GetPrimitiveArrayCritical.*
>
> — *JNI Guide*
> *Chapter 4: JNI Functions*

These paragraphs are read by some as if VM is stopping GC when critical region is running.

Actually, the only strong invariant for VM to maintain is that the object that is "critically" acquired is not moved. There are different strategies the implementation can try:

1. **Disable the GC completely** while any critical object is acquired. This is by far the simplest coping strategy, because it does not affect the rest of GC. The downside is that you have to block GC for an indefinite time (basically commiting to the mercy of users "release"-ing quickly enough), which might get problematic.
2. **Pin the object**, and work around it during the collection. This is tricky to get right if collectors expect contiguous spaces to allocate in, and/or expect the collection to process the entire heap subspace. For example, if you pin the object in young generation in simple generational GC, you cannot now "ignore" what is left in young after the collection. You cannot move the object from there either, because it breaks the very invariant you want to enforce.
3. **Pin the subspace** in heap that contains the object. Again, if GC is granular to entire generations, this is getting nowhere. But if you have regionalized heap, then you can pin a single region, and avoid GC for that region alone, keeping everyone happy.

We have seen people relying on JNI Critical to disable GC temporarily, but that only works for option "a", and not every collector employs the simplistic behavior like that.

Can we see this in practice?

Experiment

As always, we can look into it by constructing the experiment that acquires the `int[]` array with JNI Critical, and then **deliberately ignores** the suggestion to release the array after we are done with it. Instead, it would allocate and retain lots of objects between the acquire and release:

JAVA

```
public class CriticalGC {

  static final int ITERS = Integer.getInteger("iters", 100);
  static final int ARR_SIZE = Integer.getInteger("arrSize", 10_000);
  static final int WINDOW = Integer.getInteger("window", 10_000_000);

  static native void acquire(int[] arr);
  static native void release(int[] arr);

  static final Object[] window = new Object[WINDOW];

  public static void main(String... args) throws Throwable {
    System.loadLibrary("CriticalGC");

    int[] arr = new int[ARR_SIZE];

    for (int i = 0; i < ITERS; i++) {
      acquire(arr);
      System.out.println("Acquired");
      try {
        for (int c = 0; c < WINDOW; c++) {
          window[c] = new Object();
        }
      } catch (Throwable t) {
        // omit
      } finally {
        System.out.println("Releasing");
        release(arr);
      }
    }
  }
}
```

...and the native parts:

C

```
#include <jni.h>
#include <CriticalGC.h>

static jbyte* sink;

JNIEXPORT void JNICALL Java_CriticalGC_acquire
(JNIEnv* env, jclass klass, jintArray arr) {
   sink = (*env)->GetPrimitiveArrayCritical(env, arr, 0);
}

JNIEXPORT void JNICALL Java_CriticalGC_release
(JNIEnv* env, jclass klass, jintArray arr) {
   (*env)->ReleasePrimitiveArrayCritical(env, arr, sink, 0);
}
```

We need to generate the appropriate headers, compile the native parts into a library, and then make sure JVM know where to find that library. Everything is encapsulated here.

Parallel/CMS

First, obvious thing, Parallel:

```
$ make run-parallel
java -Djava.library.path=. -Xms4g -Xmx4g -verbose:gc -XX:+UseParallelGC CriticalGC
[0.745s][info][gc] Using Parallel
...
[29.098s][info][gc] GC(13) Pause Young (GCLocker Initiated GC) 1860M->1405M(3381M) 1651.290ms
Acquired
Releasing
[30.771s][info][gc] GC(14) Pause Young (GCLocker Initiated GC) 1863M->1408M(3381M) 1589.162ms
Acquired
Releasing
[32.567s][info][gc] GC(15) Pause Young (GCLocker Initiated GC) 1866M->1411M(3381M) 1710.092ms
Acquired
Releasing
...
1119.29user 3.71system 2:45.07elapsed 680%CPU (0avgtext+0avgdata 4782396maxresident)k
0inputs+224outputs (0major+1481912minor)pagefaults 0swaps
```

Notice how GC is not happening in-between "Acquired" and "Released", this the implementation detail leaking out to us. But the smoking gun is "GCLocker Initiated GC" message. GCLocker (http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/f36e864e66a7/src/share/vm/gc/shared/gcLocker.hpp) is a **lock** that prevents GC from running when JNI critical is acquired. See the relevant block (http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/f36e864e66a7/src/share/vm/prims/jni.cpp#l3173) in OpenJDK codebase:

```c
JNI_ENTRY(void*, jni_GetPrimitiveArrayCritical(JNIEnv *env, jarray array, jboolean *isCopy))
  JNIWrapper("GetPrimitiveArrayCritical");
  GCLocker::lock_critical(thread);    // <--- acquire GCLocker!
  if (isCopy != NULL) {
    *isCopy = JNI_FALSE;
  }
  oop a = JNIHandles::resolve_non_null(array);
  ...
  void* ret = arrayOop(a)->base(type);
  return ret;
JNI_END

JNI_ENTRY(void, jni_ReleasePrimitiveArrayCritical(JNIEnv *env, jarray array, void *carray, jint mode))
  JNIWrapper("ReleasePrimitiveArrayCritical");
  ...
  // The array, carray and mode arguments are ignored
  GCLocker::unlock_critical(thread); // <--- release GCLocker!
  ...
JNI_END
```

If GC was attempted, JVM should see if anybody holds that lock. If anybody does, then at least for Parallel, CMS, and G1, we cannot continue with GC. When the last critical JNI operation ends with "release", then VM checks if there are pending GC blocked by GCLocker, and if there are, then it triggers GC (http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/f36e864e66a7/src/share/vm/gc/shared/gcLocker.cpp#l138). This yields "GCLocker Initiated GC" collection.

G1

Of course, since we are playing with fire — doing weird things in JNI critical region — it can spectacularly blow up. This is reproducible with G1:

```
$ make run-g1
java -Djava.library.path=. -Xms4g -Xmx4g -verbose:gc -XX:+UseG1GC CriticalGC
[0.012s][info][gc] Using G1
<HANGS>
```

Oops! It hangs all right. `jstack` will even say we are `RUNNABLE`, but waiting on some weird condition:

```
"main" #1 prio=5 os_prio=0 tid=0x00007fdeb4013800 nid=0x4fd9 waiting on condition [0x00007fdebd5e0000]
   java.lang.Thread.State: RUNNABLE
  at CriticalGC.main(CriticalGC.java:22)
```

The easiest way to have a clue about this to run with "fastdebug" build, which will then fail on this interesting assert:

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  Internal Error (/home/shade/trunks/jdk9-dev/hotspot/src/share/vm/gc/shared/gcLocker.cpp:96), pid=17842, tid=17843
#  assert(!JavaThread::current()->in_critical()) failed: Would deadlock
#
Native frames: (J=compiled Java code, A=aot compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [libjvm.so+0x15b5934]  VMError::report_and_die(...)+0x4c4
V  [libjvm.so+0x15b644f]  VMError::report_and_die(...)+0x2f
V  [libjvm.so+0xa2d262]  report_vm_error(...)+0x112
V  [libjvm.so+0xc51ac5]  GCLocker::stall_until_clear()+0xa5
V  [libjvm.so+0xb8b6ee]  G1CollectedHeap::attempt_allocation_slow(...)+0x92e
V  [libjvm.so+0xba423d]  G1CollectedHeap::attempt_allocation(...)+0x27d
V  [libjvm.so+0xb93cef]  G1CollectedHeap::allocate_new_tlab(...)+0x6f
V  [libjvm.so+0x94bdba]  CollectedHeap::allocate_from_tlab_slow(...)+0x1fa
V  [libjvm.so+0xd47cd7]  InstanceKlass::allocate_instance(Thread*)+0xc77
V  [libjvm.so+0x13cfef0]  OptoRuntime::new_instance_C(Klass*, JavaThread*)+0x830
v  ~RuntimeStub::_new_instance_Java
J 87% c2 CriticalGC.main([Ljava/lang/String;)V (82 bytes) ...
v  ~StubRoutines::call_stub
V  [libjvm.so+0xd99938]  JavaCalls::call_helper(...)+0x858
V  [libjvm.so+0xdbe7ab]  jni_invoke_static(...) ...
V  [libjvm.so+0xdde621]  jni_CallStaticVoidMethod+0x241
C  [libjli.so+0x463c]  JavaMain+0xa8c
C  [libpthread.so.0+0x76ba]  start_thread+0xca
```

Looking closely at this stack trace, we can reconstruct what had happened: we tried to allocate new object, there were no TLABs (https://shipilev.net/jvm/anatomy-quarks/4-tlab-allocation/) to satisfy the allocations from, so we jumped to slowpath allocation trying to get new TLAB. Then we discovered no TLABs are available, tried to allocate, failed, and discovered we need to wait for GCLocker to initiate GC. Enter `stall_until_clear` to wait for this… but since we are the thread who holds the GCLocker, waiting here leads to deadlock. Boom (http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/f36e864e66a7/src/share/vm/gc/shared/gcLocker.cpp#l95).

This is within the specfication, because the test had tried to allocate things within the acquire-release block. Leaving the JNI method without the paired `release` was a mistake that exposed us to this. If we haven't left, we could not allocate in acquire-release without calling JNI, thus violating the "thou shalt not call JNI functions" principle.

You can tune up the test for collectors to not to fail this way, but then you will discover that GCLocker delaying the collection means we can start the GC when there is already too low space left in the heap, which will force us into Full GC. Oops.

Shenandoah

As described in theoreticals, the regionalized collector can pin the particular region holding the object, and leave that object alone without collection until JNI Critical is released. This is what Shenandoah (https://wiki.openjdk.java.net/display/shenandoah/Main) is doing in its current implementation.

```
$ make run-shenandoah
java -Djava.library.path=. -Xms4g -Xmx4g -verbose:gc -XX:+UseShenandoahGC CriticalGC
...
Releasing
Acquired
[3.325s][info][gc] GC(6) Pause Init Mark 0.287ms
[3.502s][info][gc] GC(6) Concurrent marking 3607M->3879M(4096M) 176.534ms
[3.503s][info][gc] GC(6) Pause Final Mark 3879M->1089M(4096M) 0.546ms
[3.503s][info][gc] GC(6) Concurrent evacuation  1089M->1095M(4096M) 0.390ms
[3.504s][info][gc] GC(6) Concurrent reset bitmaps 0.715ms
Releasing
Acquired
....
41.79user 0.86system 0:12.37elapsed 344%CPU (0avgtext+0avgdata 4314256maxresident)k
0inputs+1024outputs (0major+1085785minor)pagefaults 0swaps
```

Notice how the GC cycle started and finished while JNI Critical was acquired. Shenandoah just pinned the region holding the array, and proceeded collecting other regions like nothing happened. It can even perform the JNI Critical on object that is in the collected region, by evacuating it first, and then pinning the target region (that is obviously not in the collection set). This allows to implement JNI Critical *without GCLocker*, and therefore without GC stalls.

Observations

Handling JNI Critical requires assistance from VM to either disable GC with GCLocker-like mechanism, or pin the subspace containing the object, or pin the object alone. Different GCs employ different strategies to deal with JNI Critical, and side-effects visible when running with one collector — like delaying the GC cycle — may not be visible with another.

Please note that specification says: *"Inside a critical region, native code must not call other JNI functions"*, and this is the minimal requirement. The example above underlines the fact that within the bounds of allowed specification, quality of implementation defines how bad it would be to break the specification. Some GCs may let more things slide, others may be more restrictive. If you want to be portable, adhere to the specification requirements, not implementation details.

Or, if you rely on implementation details (which **is** a bad idea), and you run into these problems using JNI, understand what collectors are doing, and choose the approriate GC.

JVM Anatomy Quark #10: String.intern()

Question

How exactly `String.intern()` works? Should I avoid it?

Theory

If you have ever studied `String` Javadocs, you would know there is an interesting method in the public API:

> ```
> public String intern()
> ```
>
> *Returns a canonical representation for the string object. A pool of strings, initially empty, is maintained privately by the class String.*
>
> *When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.*
>
> *— JDK Javadoc*
> *java.lang.String*

This reads as if String provides the user-accessible entry to String pool, and we can use it to optimize for memory, right? However, that comes with a drawback: in OpenJDK, `String.intern()` is native, and it actually calls into JVM, to intern the String in the native JVM String pool. This is due the to the fact that String interning is a part of JDK-VM interface when both VM native and JDK code have to agree on identity of particular String objects.

There are implications for having the implementation like that:

1. You need to cross the JDK-JVM interface on every `intern()`, which wastes cycles.
2. The performance is at the mercy of the *native* HashTable implementation, which may lag behind what is available in high-performance Java world, especially under concurrent access.
3. Since Java Strings are references from the native VM structures, they become the part of GC rootset. In many cases, that requires additional work during the GC pauses to process.

Does this matter?

Experiment: Throughput

Once again, we can construct the simple experiment. Both deduplication and interning and trivally implementable with `HashMap` and `ConcurrentHashMap`, which gives us a very nice JMH (http://openjdk.java.net/projects/code-tools/jmh/) benchmark:

```java
@State(Scope.Benchmark)
public class StringIntern {

    @Param({"1", "100", "10000", "1000000"})
    private int size;

    private StringInterner str;
    private CHMInterner chm;
    private HMInterner hm;

    @Setup
    public void setup() {
        str = new StringInterner();
        chm = new CHMInterner();
        hm = new HMInterner();
    }

    public static class StringInterner {
        public String intern(String s) {
            return s.intern();
        }
    }

    @Benchmark
    public void intern(Blackhole bh) {
        for (int c = 0; c < size; c++) {
            bh.consume(str.intern("String" + c));
        }
    }

    public static class CHMInterner {
        private final Map<String, String> map;

        public CHMInterner() {
            map = new ConcurrentHashMap<>();
        }

        public String intern(String s) {
            String exist = map.putIfAbsent(s, s);
            return (exist == null) ? s : exist;
        }
    }

    @Benchmark
    public void chm(Blackhole bh) {
        for (int c = 0; c < size; c++) {
            bh.consume(chm.intern("String" + c));
        }
    }

    public static class HMInterner {
        private final Map<String, String> map;

        public HMInterner() {
            map = new HashMap<>();
        }

        public String intern(String s) {
            String exist = map.putIfAbsent(s, s);
            return (exist == null) ? s : exist;
        }
    }

    @Benchmark
    public void hm(Blackhole bh) {
        for (int c = 0; c < size; c++) {
            bh.consume(hm.intern("String" + c));
        }
    }
}
```

The test tries to intern lots of Strings, but the actual interning happens only for the first walk through the loop, and then we only checking the String after the existing mappings. `size` parameter controls the number of Strings we intern, thus limiting the String table size we are dealing with. This is the usual case with interners like that.

Running this with JDK 8u131:

```
Benchmark              (size)  Mode  Cnt       Score       Error  Units

StringIntern.chm            1  avgt   25       0.038 ±     0.001  us/op
StringIntern.chm          100  avgt   25       4.030 ±     0.013  us/op
StringIntern.chm        10000  avgt   25     516.483 ±     3.638  us/op
StringIntern.chm      1000000  avgt   25   93588.623 ±  4838.265  us/op

StringIntern.hm             1  avgt   25       0.028 ±     0.001  us/op
StringIntern.hm           100  avgt   25       2.982 ±     0.073  us/op
StringIntern.hm         10000  avgt   25     422.782 ±     1.960  us/op
StringIntern.hm       1000000  avgt   25   81194.779 ±  4905.934  us/op

StringIntern.intern         1  avgt   25       0.089 ±     0.001  us/op
StringIntern.intern       100  avgt   25       9.324 ±     0.096  us/op
StringIntern.intern     10000  avgt   25    1196.700 ±   141.915  us/op
StringIntern.intern   1000000  avgt   25  650243.474 ± 36680.057  us/op
```

Oops, what gives? `String.intern()` is significantly slower! The answer lies somewhere in the native implementation ("native" does not equal "better", folks), which is clearly visible in with `perf record -g`:

```
-    6.63%     0.00%  java     [unknown]           [k] 0x00000006f8000041
   - 0x6f8000041
      - 6.41% 0x7faedd1ee354
         - 6.41% 0x7faedd170426
            - JVM_InternString
               - 5.82% StringTable::intern
                  - 4.85% StringTable::intern
                       0.39% java_lang_String::equals
                       0.19% Monitor::lock
                     + 0.00% StringTable::basic_add
                  - 0.97% java_lang_String::as_unicode_string
                       resource_allocate_bytes
                 0.19% JNIHandleBlock::allocate_handle
                 0.19% JNIHandles::make_local
```

While the JNI transition costs quite a bit on itself, we seem to spend quite some time in StringTable (http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/910e24afc502/src/share/vm/classfile/stringTable.cpp) implementation. Poking around it, you will eventually discover `-XX:+PrintStringTableStatistics` (http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/910e24afc502/src/share/vm/runtime/globals.hpp#l2578), which will print something like:

```
StringTable statistics:
Number of buckets       :     60013 =    480104 bytes, avg   8.000
Number of entries       :   1002714 =  24065136 bytes, avg  24.000
Number of literals      :   1002714 =  64192616 bytes, avg  64.019
Total footprint         :           =  88737856 bytes
Average bucket size     :    16.708  ; <---- !!!!!!
```

16 elements per bucket in a chained hash table speaks "overload, overload, overload". What is worse, that string table is *not resizeable* — although there was experimental work to make them resizable, that was shot down for "reasons" . It might be alleviated with setting larger `-XX:StringTableSize`, for example to 10M:

```
Benchmark              (size)  Mode  Cnt       Score       Error  Units

# Default, copied from above
StringIntern.chm            1  avgt   25       0.038 ±     0.001  us/op
StringIntern.chm          100  avgt   25       4.030 ±     0.013  us/op
StringIntern.chm        10000  avgt   25     516.483 ±     3.638  us/op
StringIntern.chm      1000000  avgt   25   93588.623 ±  4838.265  us/op

# Default, copied from above
StringIntern.intern         1  avgt   25       0.089 ±     0.001  us/op
StringIntern.intern       100  avgt   25       9.324 ±     0.096  us/op
StringIntern.intern     10000  avgt   25    1196.700 ±   141.915  us/op
StringIntern.intern   1000000  avgt   25  650243.474 ± 36680.057  us/op

# StringTableSize = 10M
StringIntern.intern         1  avgt    5       0.097 ±     0.041  us/op
StringIntern.intern       100  avgt    5      10.174 ±     5.026  us/op
StringIntern.intern     10000  avgt    5    1152.387 ±   558.044  us/op
StringIntern.intern   1000000  avgt    5  130862.190 ± 61200.783  us/op
```

...but this is only a palliative measure, because you have to plan this in advance. You will waste memory if you blindly set String table size to large value, and do not use it. Even with large StringTable that you fully use, the native call costs are still eating away cycles.

Experiment: GC pauses

But what would trigger the most dramatic consequence of native String table is that it is the part of GC roots! Which means, it should be scanned/updated by the garbage collector specially. In OpenJDK, that means doing hard work during the pause. Indeed, for Shenandoah (https://wiki.openjdk.java.net/display/shenandoah/Main) where pauses depend mostly on GC root set size, having just 1M records in String table yields this:

```
$ ... StringIntern -p size=1000000 --jvmArgs "-XX:+UseShenandoahGC -Xlog:gc+stats -Xmx1g -Xms1g"
...
Initial Mark Pauses (G)    = 0.03 s (a = 15667 us) (n = 2) (lvls, us = 15039, 15039, 15039, 15039, 16260)
Initial Mark Pauses (N)    = 0.03 s (a = 15516 us) (n = 2) (lvls, us = 14844, 14844, 14844, 14844, 16088)
  Scan Roots               = 0.03 s (a = 15448 us) (n = 2) (lvls, us = 14844, 14844, 14844, 14844, 16018)
    S: Thread Roots        = 0.00 s (a =    64 us) (n = 2) (lvls, us =    41,    41,    41,    41,    87)
    S: String Table Roots  = 0.03 s (a = 13210 us) (n = 2) (lvls, us = 12695, 12695, 12695, 12695, 13544)
    S: Universe Roots      = 0.00 s (a =     2 us) (n = 2) (lvls, us =     2,     2,     2,     2,     2)
    S: JNI Roots           = 0.00 s (a =     3 us) (n = 2) (lvls, us =     2,     2,     2,     2,     4)
    S: JNI Weak Roots      = 0.00 s (a =    35 us) (n = 2) (lvls, us =    29,    29,    29,    29,    42)
    S: Synchronizer Roots  = 0.00 s (a =     0 us) (n = 2) (lvls, us =     0,     0,     0,     0,     0)
    S: Flat Profiler Roots = 0.00 s (a =     0 us) (n = 2) (lvls, us =     0,     0,     0,     0,     0)
    S: Management Roots    = 0.00 s (a =     1 us) (n = 2) (lvls, us =     1,     1,     1,     1,     1)
    S: System Dict Roots   = 0.00 s (a =     9 us) (n = 2) (lvls, us =     8,     8,     8,     8,    11)
    S: CLDG Roots          = 0.00 s (a =    75 us) (n = 2) (lvls, us =    68,    68,    68,    68,    81)
    S: JVMTI Roots         = 0.00 s (a =     0 us) (n = 2) (lvls, us =     0,     0,     0,     0,     1)
```

So, you have +13 ms per pause *just because* we decided to put more stuff in the root set.

This prompts some GC implementations to only do the String table cleanups when something heavy is also done. For example, it makes little sense from JVM perspective to clean String table if classes were not unloaded — because loaded classes are the major sources of interned Strings. So, this workload would exhibit interesting behaviors at least in G1 and CMS:

JAVA
```
public class InternMuch {
  public static void main(String... args) {
    for (int c = 0; c < 1_000_000_000; c++) {
      String s = "" + c + "root";
      s.intern();
    }
  }
}
```

Running with CMS:

```
$ java -XX:+UseConcMarkSweepGC -Xmx2g -Xms2g -verbose:gc -XX:StringTableSize=6661443 InternMuch

GC(7) Pause Young (Allocation Failure) 349M->349M(989M) 357.485ms
GC(8) Pause Initial Mark 354M->354M(989M) 3.605ms
GC(8) Concurrent Mark
GC(8) Concurrent Mark 1.711ms
GC(8) Concurrent Preclean
GC(8) Concurrent Preclean 0.523ms
GC(8) Concurrent Abortable Preclean
GC(8) Concurrent Abortable Preclean 935.176ms
GC(8) Pause Remark 512M->512M(989M) 512.290ms
GC(8) Concurrent Sweep
GC(8) Concurrent Sweep 310.167ms
GC(8) Concurrent Reset
GC(8) Concurrent Reset 0.404ms
GC(9) Pause Young (Allocation Failure) 349M->349M(989M) 369.925ms
```

SHELL

So far so relatively good. Walking the overloaded String table takes a while. But the most damning thing would be to disable class unloading with `-XX:-ClassUnloading`. This effectively (http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/385668275400/src/share/vm/gc/cms/concurrentMarkSweepGeneration.cpp#l2559) disables (http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/385668275400/src/share/vm/gc/cms/concurrentMarkSweepGeneration.cpp#l5239) String table cleanup in regular GC cycles! You can guess what happens next:

```
$ java -XX:+UseConcMarkSweepGC -Xmx2g -Xms2g -verbose:gc -XX:-ClassUnloading -XX:StringTableSize=6661443 InternMuch

GC(11) Pause Young (Allocation Failure) 273M->308M(989M) 338.999ms
GC(12) Pause Initial Mark 314M->314M(989M) 66.586ms
GC(12) Concurrent Mark
GC(12) Concurrent Mark 175.625ms
GC(12) Concurrent Preclean
GC(12) Concurrent Preclean 0.539ms
GC(12) Concurrent Abortable Preclean
GC(12) Concurrent Abortable Preclean 2549.523ms
GC(12) Pause Remark 696M->696M(989M) 133.920ms
GC(12) Concurrent Sweep
GC(12) Concurrent Sweep 175.949ms
GC(12) Concurrent Reset
GC(12) Concurrent Reset 0.463ms
GC(14) Pause Full (Allocation Failure) 859M->0M(989M) 1541.465ms  <---- !!!
GC(13) Pause Young (Allocation Failure) 859M->0M(989M) 1541.515ms
```

SHELL

Full STW GC, my old friend. For CMS, there is `ExplicitGCInvokesConcurrentAndUnloadsClasses` that kinda alleviates that, assuming user will call `System.gc()` sometimes.

Observations

> We are only discussing the ways one can achieve interning/deduplication, under the presumption it is needed for either memory footprint improvements, or low-level `==` optimization, or some other obscure need. Those needs can be accepted or challenged separately. For more details about Java Strings, I'd plug my own talk, "java.lang.String Catechism" (https://shipilev.net/#string-catechism).

For OpenJDK, `String.intern()` is the gateway to native JVM String table, and it comes with caveats: throughput, memory footprint, pause time problems will await the users. It is very easy to underestimate the impact of these caveats. Hand-rolled deduplicators/interners are working much more reliably, because they are working on Java side, are just the regular Java objects, generally better sized/resized, and also can be thrown away completely when not needed anymore. GC-assisted String deduplication (http://openjdk.java.net/jeps/192) does alleviate things even more.

In almost every project we were taking care of, removing `String.intern()` from the hotpaths, or optionally replacing it with a handrolled deduplicator, was the very profitable performance optimization. Do not use `String.intern()` without thinking very hard about it, okay?

JVM Anatomy Quark #11: Moving GC and Locality

Question

So I have heard non-moving garbage collectors are okay, because $reasons. Slower allocations and fragmentation do not concern me. Are there other implications?

Theory

If you open GC Handbook (http://gchandbook.org/), there is an a small section on "Is compaction necessary?". The last point they make is:

> *Mark-compact collectors may preserve the allocation order of objects in the heap or they may rearrange them arbitrarily. Although arbitrary order collectors may be faster than other mark-compact collectors and impose no space overheads, the mutator's locality likely to suffer from an arbitrary scrambling of object order.*
>
> — *GC Handbook*
> *3.5. Issues to consider. Locality*

Does that really matter?

Experiment

Once again, we can construct the simple experiment to see how that works. The simplest test case we can come up with in the array of references, that is either shuffled or not, and walk its contents. In JMH (http://openjdk.java.net/projects/code-tools/jmh/) speak, something like this:

```java
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.*;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(value = 1, jvmArgsAppend = {"-Xms8g", "-Xmx8g", "-Xmn7g" })
public class ArrayWalkBench {

    @Param({"16", "256", "4096", "65536", "1048576", "16777216"})
    int size;

    @Param({"false", "true"})
    boolean shuffle;

    @Param({"false", "true"})
    boolean gc;

    String[] arr;

    @Setup
    public void setup() throws IOException, InterruptedException {
        arr = new String[size];
        for (int c = 0; c < size; c++) {
            arr[c] = "Value" + c;
        }
        if (shuffle) {
            Collections.shuffle(Arrays.asList(arr), new Random(0xBAD_BEE));
        }
        if (gc) {
            for (int c = 0; c < 5; c++) {
                System.gc();
                TimeUnit.SECONDS.sleep(1);
            }
        }
    }

    @Benchmark
    public void walk(Blackhole bh) {
        for (String val : arr) {
            bh.consume(val.hashCode());
        }
    }
}
```

Note the experiment has three degrees of freedom:

1. `size` of the array in question — it is always a good idea to try several sizes when you want to implicate different caches in the memory hierarchy, and afraid to put the benchmark into accidental sweet spot.
2. `shuffle`, that tells if we shuffle the array before walking it. Enabling shuffling simulates the case when insertions are not in order and/or happen over time to different indexes.
3. `gc`, that tells to force GC after preparing the data set. Since workload is not allocating in the payload code, we need to force GC explicitly, otherwise it would never run.

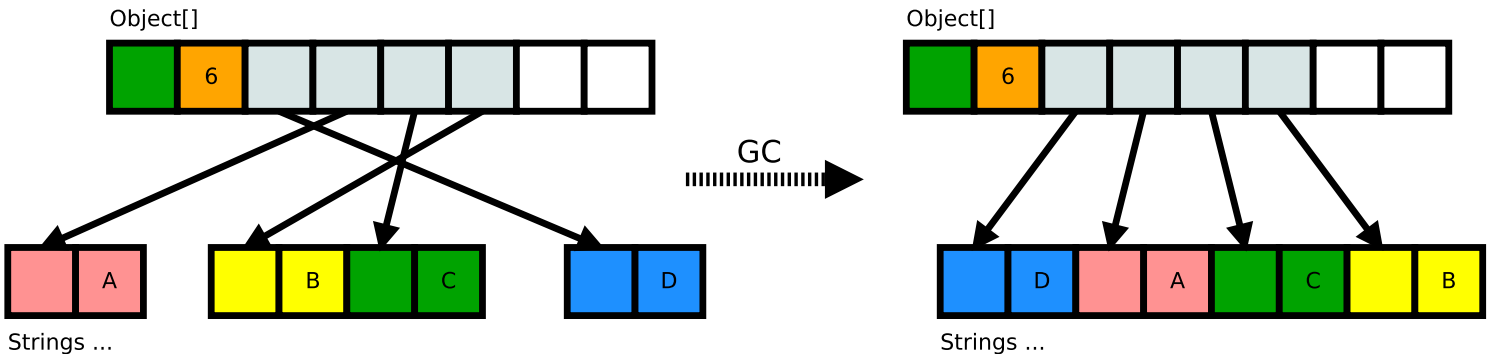
What would change with different settings? Let us take `-XX:+UseParallelGC` and see:

```
Benchmark              (gc)  (shuffle)    (size)  Mode  Cnt       Score      Error  Units

ArrayWalkBench.walk   false      false        16  avgt    5       0.051 ±    0.001  us/op
ArrayWalkBench.walk   false      false       256  avgt    5       0.821 ±    0.001  us/op
ArrayWalkBench.walk   false      false      4096  avgt    5      14.516 ±    0.026  us/op
ArrayWalkBench.walk   false      false     65536  avgt    5     307.210 ±    4.789  us/op
ArrayWalkBench.walk   false      false   1048576  avgt    5    4306.660 ±    7.950  us/op
ArrayWalkBench.walk   false      false  16777216  avgt    5   60561.476 ±  925.685  us/op

ArrayWalkBench.walk   false       true        16  avgt    5       0.051 ±    0.001  us/op
ArrayWalkBench.walk   false       true       256  avgt    5       0.823 ±    0.003  us/op
ArrayWalkBench.walk   false       true      4096  avgt    5      18.646 ±    0.044  us/op
ArrayWalkBench.walk   false       true     65536  avgt    5     461.187 ±   31.183  us/op
ArrayWalkBench.walk   false       true   1048576  avgt    5   16350.706 ±   75.757  us/op
ArrayWalkBench.walk   false       true  16777216  avgt    5  312296.960 ±  632.552  us/op

ArrayWalkBench.walk    true      false        16  avgt    5       0.051 ±    0.001  us/op
ArrayWalkBench.walk    true      false       256  avgt    5       0.820 ±    0.004  us/op
ArrayWalkBench.walk    true      false      4096  avgt    5      13.639 ±    0.063  us/op
ArrayWalkBench.walk    true      false     65536  avgt    5     174.475 ±    0.771  us/op
ArrayWalkBench.walk    true      false   1048576  avgt    5    4345.980 ±   15.230  us/op
ArrayWalkBench.walk    true      false  16777216  avgt    5   68687.192 ± 1375.171  us/op

ArrayWalkBench.walk    true       true        16  avgt    5       0.051 ±    0.001  us/op
ArrayWalkBench.walk    true       true       256  avgt    5       0.828 ±    0.010  us/op
ArrayWalkBench.walk    true       true      4096  avgt    5      13.749 ±    0.088  us/op
ArrayWalkBench.walk    true       true     65536  avgt    5     174.230 ±    0.655  us/op
ArrayWalkBench.walk    true       true   1048576  avgt    5    4365.162 ±   88.927  us/op
ArrayWalkBench.walk    true       true  16777216  avgt    5   70631.288 ± 1144.980  us/op
```

What do we see here?

We see that walking the shuffled array is indeed much, much slower than the initial in-order array — around 4x times slower! So, here is the hint: **memory layout of object graph matters!** You can control this in the code in some way during the initial load, but not when external clients put something at (possibly random) indexes.

We also see that after GC happened, both cases improved, because GC had compacted the space taken by the array out of temporary objects, if any, plus it moved the objects in memory so that they laid out in memory in *array order*. The difference between shuffled and non-shuffled version is basically gone. Therefore, here is another hint: not only GCs introduce overheads in your application, but they also **pay back by helping to re-arrange objects to benefit locality**, like this:



If you only have a non-moving collector, you pay the price of GC without one of its major benefits! Indeed, this is one of the reasons why no-op GC like Epsilon (http://openjdk.java.net/jeps/8174901) may run application slower than a compacting GC. This is Epsilon running the same workload (`gc = true` is not applicable to it):

```
Benchmark                (gc)  (shuffle)    (size)  Mode  Cnt       Score      Error  Units

ArrayWalkBench.walk     false      false        16  avgt    5       0.051 ±    0.001  us/op
ArrayWalkBench.walk     false      false       256  avgt    5       0.826 ±    0.006  us/op
ArrayWalkBench.walk     false      false      4096  avgt    5      14.556 ±    0.049  us/op
ArrayWalkBench.walk     false      false     65536  avgt    5     274.073 ±   37.163  us/op
ArrayWalkBench.walk     false      false   1048576  avgt    5    4383.223 ±  997.953  us/op
ArrayWalkBench.walk     false      false  16777216  avgt    5   60322.171 ±  266.683  us/op

ArrayWalkBench.walk     false       true        16  avgt    5       0.051 ±    0.001  us/op
ArrayWalkBench.walk     false       true       256  avgt    5       0.826 ±    0.007  us/op
ArrayWalkBench.walk     false       true      4096  avgt    5      18.169 ±    0.165  us/op
ArrayWalkBench.walk     false       true     65536  avgt    5     312.345 ±   26.149  us/op
ArrayWalkBench.walk     false       true   1048576  avgt    5   16445.739 ±   54.241  us/op
ArrayWalkBench.walk     false       true  16777216  avgt    5  311573.643 ± 3650.280  us/op
```

Yes, you read that right, Epsilon runs slower than Parallel. Being a no-op GC, it does not incur GC overheads, but it also does not bring any benefits.

The cause for performance difference is very simple, and visible with `-prof perfnorm` (we also use `-opi 1048576` to divide by number of elements):

```
Benchmark                      (gc)  (shuffle)   (size)  Mode  Cnt   Score    Error  Units

walk                           true       true  1048576  avgt   25   4.247 ±  0.090  ns/op
walk:CPI                       true       true  1048576  avgt    5   0.498 ±  0.050   #/op
walk:L1-dcache-load-misses     true       true  1048576  avgt    5   0.955 ±  0.025   #/op
walk:L1-dcache-loads           true       true  1048576  avgt    5  12.149 ±  0.432   #/op
walk:L1-dcache-stores          true       true  1048576  avgt    5   7.027 ±  0.176   #/op
walk:LLC-load-misses           true       true  1048576  avgt    5   0.156 ±  0.029   #/op
walk:LLC-loads                 true       true  1048576  avgt    5   0.514 ±  0.014   #/op
walk:cycles                    true       true  1048576  avgt    5  17.056 ±  1.673   #/op
walk:instructions              true       true  1048576  avgt    5  34.223 ±  0.860   #/op

walk                          false       true  1048576  avgt   25  16.155 ±  0.101  ns/op
walk:CPI                      false       true  1048576  avgt    5   1.885 ±  0.069   #/op
walk:L1-dcache-load-misses    false       true  1048576  avgt    5   1.922 ±  0.076   #/op
walk:L1-dcache-loads          false       true  1048576  avgt    5  12.128 ±  0.326   #/op
walk:L1-dcache-stores         false       true  1048576  avgt    5   7.032 ±  0.212   #/op
walk:LLC-load-misses          false       true  1048576  avgt    5   1.031 ±  0.032   #/op
walk:LLC-loads                false       true  1048576  avgt    5   1.365 ±  0.101   #/op
walk:cycles                   false       true  1048576  avgt    5  64.700 ±  2.613   #/op
walk:instructions             false       true  1048576  avgt    5  34.335 ±  1.564   #/op
```

With shuffled version, you have around 2 clocks per instruction, and almost every last-level cache load is a miss. No wonder it runs slower: random walks over memory would cost you a lot.

There are also interesting visualizations for moving GCs in JOL (http://openjdk.java.net/projects/code-tools/jol/) Samples, under JOLSample_22_Compaction (http://hg.openjdk.java.net/code-tools/jol/file/018c0e12f70f/jol-samples/src/main/java/org/openjdk/jol/samples/JOLSample_22_Compaction.java), JOLSample_23_Defragmentation (http://hg.openjdk.java.net/code-tools/jol/file/018c0e12f70f/jol-samples/src/main/java/org/openjdk/jol/samples/JOLSample_23_Defragmentation.java), and JOLSample_24_Colocation (http://hg.openjdk.java.net/code-tools/jol/file/018c0e12f70f/jol-samples/src/main/java/org/openjdk/jol/samples/JOLSample_24_Colocation.java).

Observations

The irony of all GC discussions is that **having GC is sometimes painful, but having no GC is sometimes painful too!**

It is very easy to underestimate the locality implications of having the non-moving GC.

I think one of the reasons why CMS works fine, while not relocating the objects in "tenured" generation, is that it has the copying "young" collection that at least makes attempt to compact before committing to particular object order in "tenured". STW collectors like Serial and Parallel(Old) reap the benefits of this for almost every collection. Regionalized collectors like G1 and Shenandoah can, should, and will exploit this too — although substantially more work is needed there because heap traversals are decoupled from evacuation. It would be audacious to claim locality does not matter. Enter NUMA, where locality penalties skyrocket, and be prepared to get royally screwed.

Note this locality property is about the object *graphs*, not the object layout itself. Even if a language provides the capabilities for controlling the memory layout of objects, that in all cases that I am aware of, cares about the object *interiors* (or, at most array of structures -like ensembles of objects), but not the arbitrary object graphs. Once you have put the regular objects in the particular places in memory — for example, not the dense array, but linked list, linked queue, concurrent skiplist, chained hashtable, what have you — you are stuck with the object graph linearized in memory in that particular way, unless you have a moving memory manager.

Also note that this locality property is *dynamic* — that is, it is dependent on what is actually going on in a particular application session, because applications *change* the object graph when running. You can teach your application to react to this appropriately by cleverly relocating its own data structures, but then you will find yourself implementing the moving automatic memory manager — or, a moving GC.

It also has nothing to do with the allocation rates — notice that the workload in almost purely static in the example above — and this is usually the case for many real life collections, especially large ones, representing the in-memory chunks of moderately frequently changed data. In this overwhelmingly frequent case it makes sense to let GC adapt to new layout, and then run with it for a while, until it changes again. Hoping that application would put them in proper order to benefit locality by itself is a wishful thinking, unless carefully designed in such a way.

Be ever so slightly wary when someone sells you the non-moving GC or no-GC solutions, and telling everything is going to be fine. Because they probably shift the locality problems (if not all other memory management problems) to your code to handle. There is no solution without (not so) hidden costs. Maybe it would be fine, the benefits would outweigh these known costs? Or maybe you just oblivious of the costs, and sellers would diplomatically avoid the topic?

JVM Anatomy Quark #12: Native Memory Tracking

Question

I have 512 MB of available memory, so I set `-Xms512m -Xmx512m`, and my VM fails with "not enough memory to proceed". Why?

Theory

JVM is the native application, and it also needs memory to maintain its internal data structures that represent application code, generated machine code, heap metadata, class metadata, internal profiling, etc. This is not accounted in Java heap, because most of those things are native, allocated in C heap, or mmap-ed to memory. JVM also prepares lots of things expecting the active long-running application with decent number of classes loaded, enough generated code created at runtime, etc. The defaults may be too high for short-lived applications in memory-constrained scenarios.

OpenJDK 8 onwards has a nice internal VM feature, called "Native Memory Tracking" (NMT) (https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr007.html): it instruments all internal VM allocations and lets categorize them, get the idea where they are coming from, etc. This feature is invaluable for understanding what VM uses memory for.

NMT can be enabled with `-XX:NativeMemoryTracking=summary`. You can get jcmd to dump the current NMT data, or you may request the data dump at JVM termination with `-XX:+PrintNMTStatistics`. Saying `-XX:NativeMemoryTracking=detail` would get the memory map for mmaps and callstacks for mallocs.

Most of the time, "summary" suffices for the overview. But we can also read the "detail"-ed log, see where allocations are coming from or for what, read the VM source code, and/or play with VM options to see what affects what. For example, take a simple "Hello World" application, like this:

```java
public class Hello {
  public static void main(String... args) {
    System.out.println("Hello");
  }
}
```

It is obvious that Java heap takes a significant part of allocated memory, let's trim `-Xmx16m -Xms16m`, and see our baseline:

```
Native Memory Tracking:

Total: reserved=1373921KB, committed=74953KB
-                 Java Heap (reserved=16384KB, committed=16384KB)
                            (mmap: reserved=16384KB, committed=16384KB)

-                     Class (reserved=1066093KB, committed=14189KB)
                            (classes #391)
                            (malloc=9325KB #148)
                            (mmap: reserved=1056768KB, committed=4864KB)

-                    Thread (reserved=19614KB, committed=19614KB)
                            (thread #19)
                            (stack: reserved=19532KB, committed=19532KB)
                            (malloc=59KB #105)
                            (arena=22KB #38)

-                      Code (reserved=249632KB, committed=2568KB)
                            (malloc=32KB #297)
                            (mmap: reserved=249600KB, committed=2536KB)

-                        GC (reserved=10991KB, committed=10991KB)
                            (malloc=10383KB #129)
                            (mmap: reserved=608KB, committed=608KB)

-                  Compiler (reserved=132KB, committed=132KB)
                            (malloc=2KB #23)
                            (arena=131KB #3)

-                  Internal (reserved=9444KB, committed=9444KB)
                            (malloc=9412KB #1373)
                            (mmap: reserved=32KB, committed=32KB)

-                    Symbol (reserved=1356KB, committed=1356KB)
                            (malloc=900KB #65)
                            (arena=456KB #1)

-    Native Memory Tracking (reserved=38KB, committed=38KB)
                            (malloc=3KB #41)
                            (tracking overhead=35KB)

-               Arena Chunk (reserved=237KB, committed=237KB)
                            (malloc=237KB)
```

Okay. 75 MB for 16 MB Java heap is certainly unexpected.

Slimdown: Sane Parts

Let's roll over different parts of that NMT output to see if those parts are tunable.

Start with something familiar:

```
-                        GC (reserved=10991KB, committed=10991KB)
                            (malloc=10383KB #129)
                            (mmap: reserved=608KB, committed=608KB)
```

This accounts for GC native structures. The log says GC malloc-ed around 10 MB and mmap-ed around 0.6 MB. One should expect this to grow with increasing heap size, if those structures describe something about the heap — for example, marking bitmaps, card tables, remembered sets, etc. Indeed it does so:

```
# Xms/Xmx = 512 MB
-                        GC (reserved=29543KB, committed=29543KB)
                            (malloc=10383KB #129)
                            (mmap: reserved=19160KB, committed=19160KB)

# Xms/Xmx = 4 GB
-                        GC (reserved=163627KB, committed=163627KB)
                            (malloc=10383KB #129)
                            (mmap: reserved=153244KB, committed=153244KB)

# Xms/Xmx = 16 GB
-                        GC (reserved=623339KB, committed=623339KB)
                            (malloc=10383KB #129)
                            (mmap: reserved=612956KB, committed=612956KB)
```

Quite probably malloc-ed parts are the C heap allocations of task queues for parallel GC, and mmap-ed regions are the bitmaps. Not surprisingly, they grow with heap size, and take around 3-4% from the configured heap size. This raises deployment questions, like in the original question: **configuring the heap size to take all available physical memory will blow the memory limits**, possibly swapping, possibly invoking OOM killer.

But that overhead is **also** dependent on the GC in use, because different GCs choose to represent Java heap differently. For example, switching back to the most lightweight GC in OpenJDK, `-XX:+UseSerialGC`, yields this dramatic change in our test case:

```
-Total: reserved=1374184KB, committed=75216KB
+Total: reserved=1336541KB, committed=37573KB

--                     Class (reserved=1066093KB, committed=14189KB)
+-                     Class (reserved=1056877KB, committed=4973KB)
                             (classes #391)
-                            (malloc=9325KB #148)
+                            (malloc=109KB #127)
                             (mmap: reserved=1056768KB, committed=4864KB)

--                    Thread (reserved=19614KB, committed=19614KB)
-                            (thread #19)
-                            (stack: reserved=19532KB, committed=19532KB)
-                            (malloc=59KB #105)
-                            (arena=22KB #38)
+-                    Thread (reserved=11357KB, committed=11357KB)
+                            (thread #11)
+                            (stack: reserved=11308KB, committed=11308KB)
+                            (malloc=36KB #57)
+                            (arena=13KB #22)

--                        GC (reserved=10991KB, committed=10991KB)
-                            (malloc=10383KB #129)
-                            (mmap: reserved=608KB, committed=608KB)
+-                        GC (reserved=67KB, committed=67KB)
+                            (malloc=7KB #79)
+                            (mmap: reserved=60KB, committed=60KB)

--                  Internal (reserved=9444KB, committed=9444KB)
-                            (malloc=9412KB #1373)
+-                  Internal (reserved=204KB, committed=204KB)
+                            (malloc=172KB #1229)
                             (mmap: reserved=32KB, committed=32KB)
```

Note this improved both "GC" parts, because less metadata is allocated, **and** "Thread" part, because there are less GC threads needed when switching from Parallel (default) to Serial GC. This means we can get partial improvement by tuning down the number of GC threads for Parallel, G1, CMS, Shenandoah, etc. We'll see about the thread stacks later. **Note that changing the GC or the number of GC threads will have performance implications** — by changing this, you are selecting another point in space-time tradeoffs.

It also improved "Class" parts, because metadata representation is slightly different. Can we squeeze out something from "Class"? Let us try Class Data Sharing (CDS) (https://docs.oracle.com/javase/8/docs/technotes/guides/vm/class-data-sharing.html), enabled with `-Xshare:on`:

```
-Total: reserved=1336279KB, committed=37311KB
+Total: reserved=1372715KB, committed=36763KB

--                     Symbol (reserved=1356KB, committed=1356KB)
-                            (malloc=900KB #65)
-                            (arena=456KB #1)
-
+-                    Symbol (reserved=503KB, committed=503KB)
+                            (malloc=502KB #12)
+                            (arena=1KB #1)
```
DIFF

There we go, saved another 0.5 MB in internal symbol tables by loading the pre-parsed representation from the shared archive.

Now let's focus on threads. The log would say:

```
-                    Thread (reserved=11357KB, committed=11357KB)
                            (thread #11)
                            (stack: reserved=11308KB, committed=11308KB)
                            (malloc=36KB #57)
                            (arena=13KB #22)
```
BASH

Looking into this, you can see that most of the space taken by threads are the thread stacks. You can try to trim the stack size down from the default (which appears to be 1M in this example) to something less with `-Xss`. **Note would yield a greater risk of `StackOverflowException`-s**, so if you do change this option, be sure to test all possible configurations of your software to look out for ill effects. Adventurously setting this to 256 KB with `-Xss256k` yields:

```
-Total: reserved=1372715KB, committed=36763KB
+Total: reserved=1368842KB, committed=32890KB

--                     Thread (reserved=11357KB, committed=11357KB)
+-                    Thread (reserved=7517KB, committed=7517KB)
                            (thread #11)
-                           (stack: reserved=11308KB, committed=11308KB)
+                           (stack: reserved=7468KB, committed=7468KB)
                            (malloc=36KB #57)
                            (arena=13KB #22)
```
DIFF

Not bad, another 4 MB is gone. Of course, the improvement would be more drastic with more application threads, and it will quite probably be the second largest consumer of memory after Java heap.

Continuing on threading, JIT compiler itself also has threads. This partially explains why we set stack size to 256 KB, but the data above says the average stack size is still `7517 / 11 = 683 KB`. Trimming the number of compiler threads down with `-XX:CICompilerCount=1` **and** setting `-XX:-TieredCompilation` to enable only the latest compilation tier yields:

```
-Total: reserved=1368612KB, committed=32660KB
+Total: reserved=1165843KB, committed=29571KB

--                     Thread (reserved=7517KB, committed=7517KB)
-                            (thread #11)
-                            (stack: reserved=7468KB, committed=7468KB)
-                            (malloc=36KB #57)
-                            (arena=13KB #22)
+-                    Thread (reserved=4419KB, committed=4419KB)
+                            (thread #8)
+                            (stack: reserved=4384KB, committed=4384KB)
+                            (malloc=26KB #42)
+                            (arena=9KB #16)
```
DIFF

Not bad, three threads are gone, and their stacks gone too! **Again, this has performance implications: less compiler threads means slower warmup**.

Trimming down Java heap size, selecting appropriate GC, trimming down the number of VM threads, trimming down the Java stack thread sizes and thread counts are the general techniques for reducing VM footprint in memory-constrained scenarios. With these, we have trimmed down our 16 MB Java heap test case to:

```
-Total: reserved=1373922KB, committed=74954KB
+Total: reserved=1165843KB, committed=29571KB
```
DIFF

Slimdown: Insane Parts

What is suggested in this section is insane. Use this at your own risk. Do not try this at home.

Moving to insane parts, which involve tuning down internal VM settings. This is not guaranteed to work, and may crash and burn unexpectedly. For example, we can control the stack sizes required for our Java application by coding it carefully. But we don't know what is going on inside the JVM itself, so trimming down the stack size for VM threads is dangerous. Still, hilarious to try with `-XX:VMThreadStackSize=256`:

```
-Total: reserved=1165843KB, committed=29571KB
+Total: reserved=1163539KB, committed=27267KB

--                    Thread (reserved=4419KB, committed=4419KB)
+-                    Thread (reserved=2115KB, committed=2115KB)
                             (thread #8)
-                            (stack: reserved=4384KB, committed=4384KB)
+                            (stack: reserved=2080KB, committed=2080KB)
                             (malloc=26KB #42)
                             (arena=9KB #16)
```
DIFF

Ah yes, another 2 MB are gone along with compiler and GC thread stacks.

Let's continue abusing the compiler code: why don't we trim down the initial code cache size — the size of area for generated code? Enter `-XX:InitialCodeCacheSize=4096` (bytes!):

```
-Total: reserved=1163539KB, committed=27267KB
+Total: reserved=1163506KB, committed=25226KB

--                      Code (reserved=49941KB, committed=2557KB)
+-                      Code (reserved=49941KB, committed=549KB)
                             (malloc=21KB #257)
-                            (mmap: reserved=49920KB, committed=2536KB)
+                            (mmap: reserved=49920KB, committed=528KB)

 -                        GC (reserved=67KB, committed=67KB)
                             (malloc=7KB #78)
```
DIFF

Ho-ho! This will balloon up once we hit heavy compilation, but so far so good.

Looking closer to "Class" again, we can see that most of the 4 MB committed for our Hello World is the initial metadata storage size. We can trim it down with `-XX:InitialBootClassLoaderMetaspaceSize=4096` (bytes!):

```
-Total: reserved=1163506KB, committed=25226KB
+Total: reserved=1157404KB, committed=21172KB

--                     Class (reserved=1056890KB, committed=4986KB)
+-                     Class (reserved=1050754KB, committed=898KB)
                             (classes #4)
-                            (malloc=122KB #83)
-                            (mmap: reserved=1056768KB, committed=4864KB)
+                            (malloc=122KB #84)
+                            (mmap: reserved=1050632KB, committed=776KB)

 -                    Thread (reserved=2115KB, committed=2115KB)
                             (thread #8)
```
DIFF

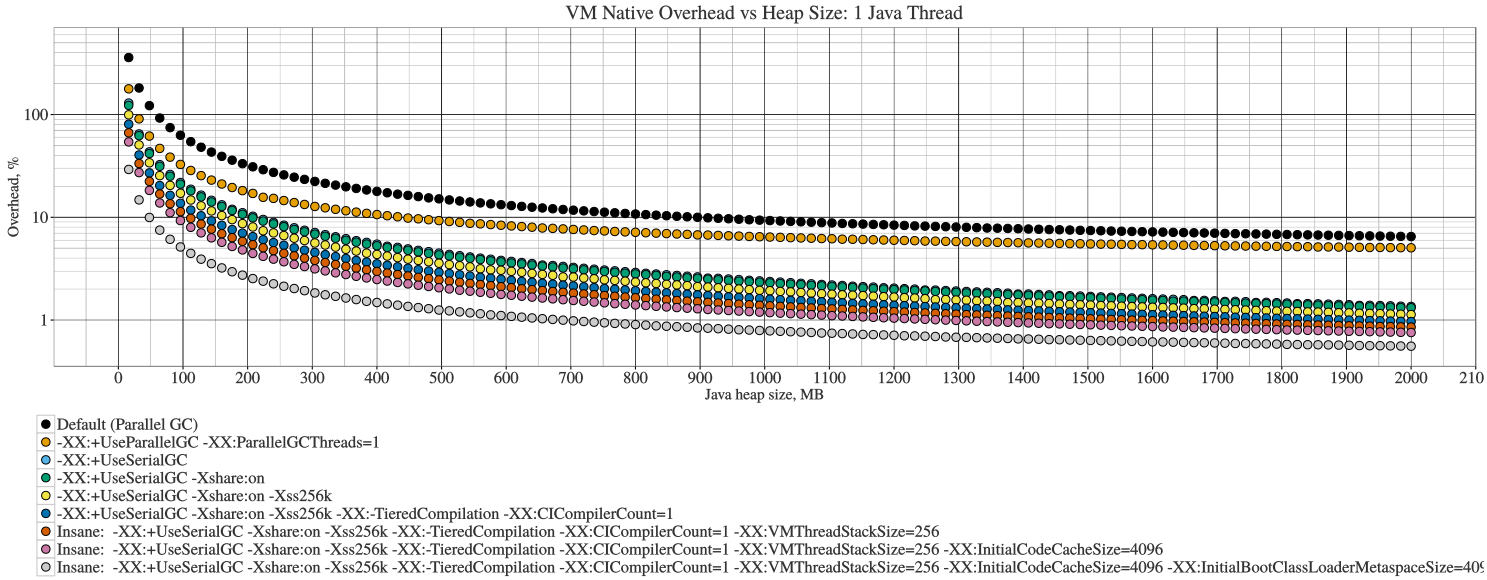
Overall, after all this madness, we came even closer to 16 MB of Java heap size, wasting only 8.5 MB on top of that:

```
-Total: reserved=1165843KB, committed=29571KB
+Total: reserved=1157404KB, committed=21172KB
```
DIFF

We can probably come even closer if we start yanking parts of JVM in our custom build.

Putting All Together

For fun, we can see how the native overhead changes with heap size on our test workload:



This confirms our gained intuition that GC overheads are the constant factor of Java heap size, and native VM overheads starts to matter only on lower heap sizes when the absolute values for VM overhead start to become a factor in overall footprint. **This picture omits the second most important thing though: thread stacks.**

Observations

Default JVM configuration is usually tailored to long-running server-class applications, and so its initial guesses about the GCs, the initial sizes for internal data structures, stack sizes, etc. may be not appropriate for short-running memory-constrained applications. Understanding what are the major memory hogs in current JVM configuration helps cramming more JVMs on the host.

Using NMT to discover where VM spends memory is usually an enlightening exercise. It almost immediately leads to insights where to get memory footprint improvements for the particular application. Hooking up online NMT monitor to performance management systems would help to adjust the JVM parameters when running actual production applications. This is much, much, much easier than trying to figure out what JVM is doing by parsing the opaque memory maps from e.g. `/proc/$pid/maps`.

Also see "OpenJDK and Containers" (https://developers.redhat.com/blog/2017/04/04/openjdk-and-containers/) by Christine Flood.

JVM Anatomy Quark #13: Intergenerational Barriers

Question

Epsilon GC JEP (http://openjdk.java.net/jeps/8174901) mentions in its "Motivation" section, among other things:

> *"Last-drop performance improvements: ... Even for non-allocating workloads, the choice of GC means choosing the set of GC barriers that the workload has to use, even if no GC cycle actually happens. Most OpenJDK GCs are generational, and they emit at least one reference write barrier. Avoiding this barrier brings the last bit of performance improvement."*

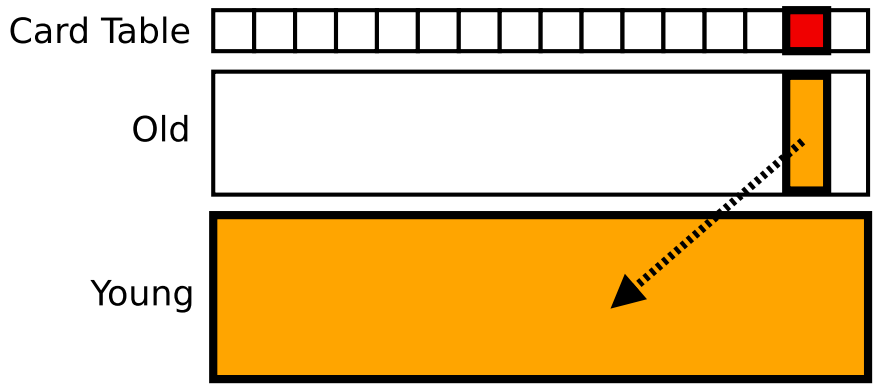
What gives?

Theory

If any garbage collector wants to collect parts of managed heap without touching the entire heap, then it has to understand what references are pointing into that collected part. Otherwise, it cannot reliably tell what is reachable in that collected part, thus having to conservatively assume everything is reachable... and that puts it in position where nothing can be treated as garbage, burning the idea to the ground. Second, it wants to understand which locations to update, if it ever moves any objects in that collected part — this concerns mostly "outside" pointers that would not be visited when processing the collected part.

The simplest (and very effective) way to split the heap in parts is to segregate objects *by age*, that is, introduce *generations* (https://en.wikipedia.org/wiki/Tracing_garbage_collection#Generational_GC_.28ephemeral_GC.29). The key idea here is *weak generational hypothesis* (http://www.memorymanagement.org/glossary/g.html#term-generational-hypothesis) that claims "new objects die young". The practical thing that lets capitalize on weak generational hypothesis is that in *marking* collectors, the performance is dependent on the number of surviving objects. Which means, we can have a *young generation*, where everything is mostly dead, process it quickly and maybe more frequently, while *old generation* is kept aside.

However, there could be references from *old* to *young* generations that you need to take care of when collecting young generation alone. Old generation is usually collected along with the entire heap, young→old references can be omitted from tracking. Ditto for young→young and old→old references, because their referees would get visited in the same collection.



Taking Parallel GC from OpenJDK as the simplest example, it records the old→young references with the help of *Card Table*: the coarse bitmap that spans the old generation. When store happens, it needs to flip a bit in that card table. That bit would mean that the part of old generation "under the card" can potentially have pointers to young gen, and it needs to be scanned when young gen is collected. For all this to work, reference stores have to be augmented with *write barriers*, little pieces of code that do card table management.

Practice

Does that really matter in practice? It sometimes does! Take this workload as example:

```java
@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(value = 3, jvmArgsAppend = {"-Xms4g", "-Xmx4g", "-Xmn2g"})
@Threads(Threads.MAX)
public class HashWalkBench {

    @Param({"16", "256", "4096", "65536", "1048576", "16777216"})
    int size;

    Map<String, String> map;

    @Setup
    public void setup() throws Exception {
        create();
        System.gc();
    }

    private void create() {
        String[] keys = new String[size];
        String[] values = new String[size];
        for (int c = 0; c < size; c++) {
            keys[c] = "Key" + c;
            values[c] = "Value" + c;
        }
        map = new HashMap<>(size);
        for (int c = 0; c < size; c++) {
            String key = keys[c];
            String val = values[c];
            map.put(key, val);
        }
    }

    @Benchmark
    public void walk(Blackhole bh) {
        for (Map.Entry<String, String> e : map.entrySet()) {
            bh.consume(e.getKey());
            bh.consume(e.getValue());
        }
    }
}
```

It creates a `HashMap`, performs a few rounds of GC, and then walks it. Running it with Epsilon vs Parallel will yield:

```shell
Benchmark             (size)  Mode  Cnt       Score      Error  Units

# Epsilon
HashWalkBench.walk        16  avgt   15       0.238 ±    0.005  us/op
HashWalkBench.walk       256  avgt   15       3.638 ±    0.072  us/op
HashWalkBench.walk      4096  avgt   15      59.222 ±    1.974  us/op
HashWalkBench.walk     65536  avgt   15    1102.590 ±    4.331  us/op
HashWalkBench.walk   1048576  avgt   15   19683.680 ±  195.086  us/op
HashWalkBench.walk  16777216  avgt   15  328319.596 ± 7137.066  us/op

# Parallel
HashWalkBench.walk        16  avgt   15       0.240 ±    0.001  us/op
HashWalkBench.walk       256  avgt   15       3.679 ±    0.078  us/op
HashWalkBench.walk      4096  avgt   15      64.778 ±    0.275  us/op
HashWalkBench.walk     65536  avgt   15    1377.634 ±   28.132  us/op
HashWalkBench.walk   1048576  avgt   15   25223.994 ±  853.601  us/op
HashWalkBench.walk  16777216  avgt   15  400679.042 ± 8155.414  us/op
```

Wait, but haven't we talked about the *stores*? Where is the store here? Apart from the infrastructure stores handled by JMH itself, the interesting thing happens behind the for-each loop cover. It implicitly instantiates `HashMap.EntrySetIterator`, which *saves* the current entry in its field. (It would have been scalarized, if escape analysis was not this fragile).

We can clearly see that store and the associated barrier in the generated code:

```asm
1.58%    0.91%    ...a4e2c: mov    %edi,0x18(%r9)       ; %r9 = iterator, 0x18(%r9) = field
0.27%    0.33%    ...a4e30: mov    %r9,%r11             ; r11 = &iterator
0.26%    0.38%    ...a4e33: shr    $0x9,%r11            ; r11 = r11 >> 9
0.13%    0.20%    ...a4e37: movabs $0x7f2c535aa000,%rbx ; rbx = card table base
0.58%    0.57%    ...a4e41: mov    %r12b,(%rbx,%r11,1)  ; put 0 to (rbx+r11)
```

There are a few observations here:

1. The card mark sets the entire byte, not just a bit. This avoids synchronization: most hardware can perform the byte store atomically, without touching the data around it. This makes card table beefier than it could theoretically be, but still quite dense: 1 byte of card table per 512 bytes of heap, notice the shift by 9.
2. The barrier happens unconditionally, while we only need to record old→young references. This seems practical: we don't have excess branches on hot path, and we trade card table that spans the entire heap, not only the old. Given how dense the card table is, we would introduce only a tiny fraction of additional footprint. This also helps to move the ephemeral boundary between young and old generations in collectors that can tune themselves, without having to patch the code.
3. The card table address is encoded in the generated code, which is practical again, because it is a native unmovable structure. That saves memory loads, because we would have to poll the card table address from somewhere otherwise.
4. The card mark "set" is actually encoded as "0". This is again practical, because we ca then reuse the zero registers — especially on architectures that have them explicitly — to get the source operand. It does not matter what value to use for card table initialization, 0 or 1, later in native GC code.

This performance picture is further corroborated by hardware counters (normalized to operation, and then divided by 1M):

```shell
Benchmark                                (size)  Mode  Cnt    Score    Error  Units

# Epsilon
HashWalkBench.walk                       1048576  avgt   15    0.019 ±  0.001  us/op
HashWalkBench.walk:L1-dcache-load-misses 1048576  avgt    3    0.389 ±  0.495   #/op
HashWalkBench.walk:L1-dcache-loads       1048576  avgt    3   25.439 ±  2.411   #/op
HashWalkBench.walk:L1-dcache-stores      1048576  avgt    3   20.090 ±  1.184   #/op
HashWalkBench.walk:cycles                1048576  avgt    3   75.230 ± 11.333   #/op
HashWalkBench.walk:instructions          1048576  avgt    3   90.075 ± 10.484   #/op

# Parallel
HashWalkBench.walk                       1048576  avgt   15    0.024 ±  0.001  us/op
HashWalkBench.walk:L1-dcache-load-misses 1048576  avgt    3    1.156 ±  0.360   #/op
HashWalkBench.walk:L1-dcache-loads       1048576  avgt    3   25.417 ±  1.711   #/op
HashWalkBench.walk:L1-dcache-stores      1048576  avgt    3   23.265 ±  3.552   #/op
HashWalkBench.walk:cycles                1048576  avgt    3   97.435 ± 69.688   #/op
HashWalkBench.walk:instructions          1048576  avgt    3  102.477 ± 12.689   #/op
```

So, parallel does the same amount of loads (thanks to encoded card table pointer), and it makes 3 additional stores, that cost around 22 additional cycles and 12 instructions. This addition seems to correspond to three write barriers in this particular workload. Notice that L1 cache misses are ever so slightly greater too: because card mark stores pollute it, reducing the effective cache capacity for the application.

Observations

GCs are usually coming with the set of barriers that affect application performance even when no actual GC work is happening. Even in the case of very basic generational collector like Serial/Parallel, you have at least one reference store barrier that has to record inter-generational barriers. More advanced collectors like G1 have even more sophisticated barriers that track references between the regions. In some cases, that cost is painful to warrant tricks to avoid it, including the no-op GC, like Epsilon.

JVM Anatomy Quark #14: Constant Variables

Question

Are `final` instance fields ever trivially treated as constants?

Theory

If you read the Java Language Specification chapters that concerns themselves with describing the base semantics of `final` variables (https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.12.4), then you will discover a spooky paragraph:

> *A constant variable is a final variable of primitive type or type String that is initialized with a constant expression (§15.28). Whether a variable is a constant variable or not may have implications with respect to class initialization (§12.4.1), binary compatibility (§13.1, §13.4.9), and definite assignment (§16 (Definite Assignment)).*
>
> — *Java Language Specification 4.12.4*

Brilliant! Is this observable in practice?

Practice

Consider this code. What does it print?

```java
import java.lang.reflect.Field;

public class ConstantValues {

    final int fieldInit = 42;
    final int instanceInit;
    final int constructor;

    {
        instanceInit = 42;
    }

    public ConstantValues() {
        constructor = 42;
    }

    static void set(ConstantValues p, String field) throws Exception {
        Field f = ConstantValues.class.getDeclaredField(field);
        f.setAccessible(true);
        f.setInt(p, 9000);
    }

    public static void main(String... args) throws Exception {
        ConstantValues p = new ConstantValues();

        set(p, "fieldInit");
        set(p, "instanceInit");
        set(p, "constructor");

        System.out.println(p.fieldInit + " " + p.instanceInit + " " + p.constructor);
    }

}
```

On my machine, it prints:

```
42 9000 9000
```

In other words, even though we had overwritten the "fieldInt" field, we don't observe its new value. More confusingly, other two variables seem to be happily updated. The answer is that two other fields are *blank final fields*, and the first field is *constant variable*. If you look into the generated bytecode for the class above, then:

```
$ javap -c -v -p ConstantValues.class
...

final int fieldInit;
  descriptor: I
  flags: ACC_FINAL
  ConstantValue: int 42  <---- oh...

final int instanceInit;
  descriptor: I
  flags: ACC_FINAL

final int constructor;
  descriptor: I
  flags: ACC_FINAL

...
public static void main(java.lang.String...) throws java.lang.Exception;
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC, ACC_VARARGS
  Code:
     ...
     41: bipush        42   // <--- Oh wow, inlined fieldInit field
     43: invokevirtual #18  // StringBuilder.append
     46: ldc           #19  // String " "
     48: invokevirtual #20  // StringBuilder.append
     51: aload_1
     52: getfield      #3   // Field instanceInit:I
     55: invokevirtual #18  // StringBuilder.append
     58: ldc           #19  // String ""
     60: invokevirtual #20  // StringBuilder.append
     63: aload_1
     64: getfield      #4   // Field constructor:I
     67: invokevirtual #18  // StringBuilder.append
     70: invokevirtual #21  // StringBuilder.toString
     73: invokevirtual #22  // System.out.println
```

No wonder we do not see the update to "fieldInit" field: the *javac* itself had inlined its value at use, and there is no chance the JVM would double-back and rewrite the bytecode to reflect something else.

This optimization is handled by the bytecode compiler itself. This has obvious performance benefits: no need for complicated analysis in JIT compiler to make use of constness of constant variables. But, as always, that comes with a cost. Besides implications for binary compatibility (for example, what happens if we recompile the class with new value?), which is briefly discussed in relevant chapters of JLS (https://docs.oracle.com/javase/specs/jls/se8/html/jls-13.html#jls-13.4.9), this has interesting implications on low-level benchmarking. For example, blindly trying to quantify if `final` modifier on instance field gives the performance improvement for real classes, we might want to measure the most trivial thing:

```
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class FinalInitBench {
    // Too lazy to actually build the example class with constructor that initializes
    // final fields, like we have in production code. No worries, we shall just model
    // this with naked fields. Right?

    final int fx = 42;  // Compiler complains about initialization? Okay, put 42 right here!
          int x  = 42;

    @Benchmark
    public int testFinal() {
        return fx;
    }

    @Benchmark
    public int test() {
        return x;
    }
}
```

Initializing the final field with its own initializer silently introduces the effect we are not probably after! Running this example benchmark with "perfnorm" profiler right away to see the low-level performance counters, you get a spooky result: `final` field access is slightly better, **and** it produces less loads![1]

```
SHELL
Benchmark                                  Mode  Cnt   Score    Error  Units
FinalInitBench.test                        avgt    9   1.920 ±  0.002  ns/op
FinalInitBench.test:CPI                    avgt    3   0.291 ±  0.039   #/op
FinalInitBench.test:L1-dcache-loads        avgt    3  11.136 ±  1.447   #/op
FinalInitBench.test:L1-dcache-stores       avgt    3   3.042 ±  0.327   #/op
FinalInitBench.test:cycles                 avgt    3   7.316 ±  1.272   #/op
FinalInitBench.test:instructions           avgt    3  25.178 ±  2.242   #/op

FinalInitBench.testFinal                   avgt    9   1.901 ±  0.001  ns/op
FinalInitBench.testFinal:CPI               avgt    3   0.285 ±  0.004   #/op
FinalInitBench.testFinal:L1-dcache-loads   avgt    3   9.077 ±  0.085   #/op  <--- !
FinalInitBench.testFinal:L1-dcache-stores  avgt    3   4.077 ±  0.752   #/op
FinalInitBench.testFinal:cycles            avgt    3   7.142 ±  0.071   #/op
FinalInitBench.testFinal:instructions      avgt    3  25.102 ±  0.422   #/op
```

This is because there is no field load in the generated code at all, and all we do is use the inlined constant from the incoming bytecode:

```
SHELL
# test
...
1.02%    1.02%  mov    0x10(%r10),%edx ; <--- get field x
2.50%    1.79%  nop
1.79%    1.60%  callq  CONSUME
...

# testFinal
...
8.25%    8.21%  mov    $0x2a,%edx      ; <--- just use inlined "42"
1.79%    0.56%  nop
1.35%    1.19%  callq  CONSUME
...
```

Not a problem in itself, but that result would be different for *blank* final fields, which would be closer aligned with real-world usages. So, a less lazier version:

```
JAVA
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class FinalInitCnstrBench {
    final int fx;
    int x;

    public FinalInitCnstrBench() {
        this.fx = 42;
        this.x = 42;
    }

    @Benchmark
    public int testFinal() {
        return fx;
    }

    @Benchmark
    public int test() {
        return x;
    }
}
```

...produces more sensible results, where both tests produce equal performance:[2]

```
Benchmark                                        Mode  Cnt   Score    Error  Units
FinalInitCnstrBench.test                         avgt    9   1.922 ±  0.003  ns/op
FinalInitCnstrBench.test:CPI                     avgt    3   0.289 ±  0.049   #/op
FinalInitCnstrBench.test:L1-dcache-loads         avgt    3  11.171 ±  1.429   #/op
FinalInitCnstrBench.test:L1-dcache-stores        avgt    3   3.042 ±  0.031   #/op
FinalInitCnstrBench.test:cycles                  avgt    3   7.301 ±  0.445   #/op
FinalInitCnstrBench.test:instructions            avgt    3  25.235 ±  1.732   #/op

FinalInitCnstrBench.testFinal                    avgt    9   1.919 ±  0.002  ns/op
FinalInitCnstrBench.testFinal:CPI                avgt    3   0.287 ±  0.014   #/op
FinalInitCnstrBench.testFinal:L1-dcache-loads    avgt    3  11.170 ±  1.104   #/op
FinalInitCnstrBench.testFinal:L1-dcache-stores   avgt    3   3.039 ±  0.864   #/op
FinalInitCnstrBench.testFinal:cycles             avgt    3   7.278 ±  0.394   #/op
FinalInitCnstrBench.testFinal:instructions       avgt    3  25.314 ±  0.588   #/op
```

Observations

The constant propagation story in Java is complicated, and there are some interesting corner cases. Constant variables that are treated specially by the *bytecode compiler* is one of those corner cases. It is most likely you will blow yourself up on this in low-level benchmarking, not dealing with production code that initializes fields in constructors anyway. The need for capturing and quantifying these corner cases is one of the reasons why JMH has "perfasm" and "perfnorm" profilers are there to make sense of the results.

JVM Anatomy Quark #15: Just-In-Time Constants

Question

Surely there are constant values in the program that optimizers can exploit. Does JVM do any tricks there?

Theory

Of course, constant-based optimizations are among the most profitable ones around. Nothing beats not doing the work at run time, when it can be done at compile time. But what is the constant? It seems that plain fields are not constants: they change all the time. What about `final`-s? They should stay the same. But, since instance fields are the part of the object state, `final` instance fields values also depend on the *identity* of the object in question:

```
JAVA
class M {
  final int x;
  M(int x) { this.x = x; }
}

M m1 = new M(1337);
M m2 = new M(8080);

void work(M m) {
  return m.x; // what to compile in here, 1337 or 8080?
}
```

Therefore, it stands to reason that if we compile method `work` above without knowing anything about the identity of the object coming as the argument [3], the only thing we can trust is **static final** fields: they are unchangeable because of `final`, and we know exactly the identity of "holding object", because it is held by the class, not by the every individual object.

Can we observe this in practice?

Practice

Consider this JMH benchmark:

```
JAVA
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class JustInTimeConstants {

    static final long x_static_final = Long.getLong("divisor", 1000);
    static       long x_static       = Long.getLong("divisor", 1000);
           final long x_inst_final   = Long.getLong("divisor", 1000);
                 long x_inst         = Long.getLong("divisor", 1000);

    @Benchmark public long _static_final() { return 1000 / x_static_final; }
    @Benchmark public long _static()       { return 1000 / x_static;       }
    @Benchmark public long _inst_final()   { return 1000 / x_inst_final;   }
    @Benchmark public long _inst()         { return 1000 / x_inst;         }

}
```

It is carefully constructed so that compilers can use the fact that divisor is constant and optimize the division out. If we run this test, this is what we shall see this:

```
SHELL
Benchmark                          Mode  Cnt  Score   Error  Units
JustInTimeConstants._inst          avgt   15  9.670 ± 0.014  ns/op
JustInTimeConstants._inst_final    avgt   15  9.690 ± 0.036  ns/op
JustInTimeConstants._static        avgt   15  9.705 ± 0.015  ns/op
JustInTimeConstants._static_final  avgt   15  1.899 ± 0.001  ns/op
```

Briefly studying the hottest loop in this benchmark with `-prof perfasm` reveals a few implementation details and the reason why some tests are faster.

`_inst` and `_inst_final` are not surprising: they read the field and use it as divisor. The bulk of cycles is spent doing the actual integer division:

```
ASM
# JustInTimeConstants._inst / _inst_final hottest loop
0.21%            ↗  mov    0x40(%rsp),%r10
0.02%            |  mov    0x18(%r10),%r10    ; get field x_inst / x_inst_final
                 |  ...
0.13%            |  idiv   %r10               ; ldiv
76.59%   95.38%  |  mov    0x38(%rsp),%rsi    ; prepare and consume the value (JMH infra)
0.40%            |  mov    %rax,%rdx
0.10%            |  callq  CONSUME
                 |  ...
1.51%            |  test   %r11d,%r11d        ; call @Benchmark again
                 ↳  je     BACK
```

`_static` is a bit more interesting: it reads the static field off the native class mirror, where static fields reside. Since runtime knows what class we are dealing with (static field accesses are statically resolved!), we inline the constant pointer to mirror, and access the field by its predefined offset. But, since we don't know what is the value of the field — indeed someone could have changed it after the code was generated — we still do the same integer division:

```
ASM
# JustInTimeConstants._static hottest loop
0.04%            ↗  movabs $0x7826385f0,%r10  ; native mirror for JustInTimeConstants.class
0.02%            |  mov    0x70(%r10),%r10    ; get static x_static
                 |  ...
0.02%            |  idiv   %r10               ;*ldiv
72.78%   95.51%  |  mov    0x38(%rsp),%rsi    ; prepare and consume the value (JMH infra)
0.38%            |  mov    %rax,%rdx
0.04%    0.06%   |  data16 xchg %ax,%ax
         0.02%   |  callq  CONSUME
                 |  ...
0.13%            |  test   %r11d,%r11d        ; call @Benchmark again
                 ↳  je     BACK
```

`_static_final` is the most interesting of them all. JIT compiler knows exactly the value it is dealing with, and so it can aggressively optimize for it. Here, the loop computation just reuses the slot which holds the precomputed value of "1000 / 1000", which is "1" [4]:

```
ASM
# JustInTimeConstants._static_final hottest loop
1.36%    1.40%   ↗  mov    %r8,(%rsp)
7.73%    7.40%   |  mov    0x8(%rsp),%rdx       ; <--- slot holding the "long" constant "1"
0.45%    0.51%   |  mov    0x38(%rsp),%rsi      ; prepare and consume the value (JMH infra)
3.59%    3.24%   |  nop
1.44%    0.54%   |  callq  CONSUME
                 | ...
3.46%    2.37%   |  test   %r10d,%r10d          ; call @Benchmark again
                 ↳  je     BACK
```

So the performance is explained by compiler's ability to constant fold through `static final`.

Observations

Note that in this example, the *bytecode compiler* (e.g. javac) has no idea what is the value of `static final` field is, because that field is initialized with a *runtime value*. When JIT compilation happens, the class had succeeded initialization, and the value is there, and can be used! This is really the *just-in-time constant*. This allows to develop the very efficient, yet runtime-adjustable code: indeed the whole thing was thought up as the replacement for preprocessor-based asserts. [5] I frequently miss this kind of trick in C++ land, where compilation is fully ahead-of-time, and thus you have to be creative if you want to have critical code depend on runtime options.[6]

A significant part of the story is the interpreter / tiered compilation. Class initializers are usually cold code, because they are executed once. But the more important thing is handling the *lazy* part of class initialization, when we want to load and initialize class the very first time on the very first access to field. Interpreter or baseline JIT compiler (e.g. C1 in Hotspot) runs it for us. By the time optimizing JIT compiler (e.g. C2 in Hotspot) runs for the same method, the classes that recompiled method needs are usually fully initialized, and their `static final`-s are fully known.

JVM Anatomy Quark #16: Megamorphic Virtual Calls

Question

I have heard megamorphic virtual calls are so bad, they are getting called by interpreter, not optimizing compiler. Is that true?

Theory

If you have read numerous articles about virtual call optimization (https://shipilev.net/blog/2015/black-magic-method-dispatch/) in Hotspot, you may have left with the belief that megamorphic calls are pure evil, because they invoke the slowpath handling, and do not enjoy compiler optimizations. If you try to comprehend what OpenJDK does when it fails to devirtualize the call, you might wonder that it crashes and burns performance-wise. But, consider that JVMs work decently well even with baseline compilers, and in some cases even the interpreter performance is okay (and it matters for time-to-performance).

So, it would be premature to conclude that runtime just gives up?

Practice

Let us try and see how does the virtual call slowpath looks. For that, we make the artificial megamorphic call site in a JMH benchmark: make the three subclasses visiting the same call site:

JAVA

```
import org.openjdk.jmh.annotations.*;

@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(1)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class VirtualCall {

    static abstract class A {
        int c1, c2, c3;
        public abstract void m();
    }

    static class C1 extends A {
        public void m() { c1++; }
    }
    static class C2 extends A {
        public void m() { c2++; }
    }
    static class C3 extends A {
        public void m() { c3++; }
    }

    A[] as;

    @Param({"mono", "mega"})
    private String mode;

    @Setup
    public void setup() {
        as = new A[300];
        boolean mega = mode.equals("mega");
        for (int c = 0; c < 300; c += 3) {
            as[c]   = new C1();
            as[c+1] = mega ? new C2() : new C1();
            as[c+2] = mega ? new C3() : new C1();
        }
    }

    @Benchmark
    public void test() {
        for (A a : as) {
            a.m();
        }
    }
}
```

To simplify things for analysis, we invoke this with `-XX:LoopUnrollLimit=1 -XX:-TieredCompilation`: this will block loop unrolling from complicating the disssembly, and disabling tiered compilation would guarantee compilation with the final optimizing compiler. We don't care about performance numbers all that much, but let's have them to frame the discussion:

```
SHELL
Benchmark         (mode)  Mode  Cnt     Score    Error  Units
VirtualCall.test    mono  avgt    5   325.478 ± 18.156  ns/op
VirtualCall.test    mega  avgt    5  1070.304 ± 53.910  ns/op
```

To give you some taste of what would happen if we **do not** use the optimizing compiler on test, run with `-XX:CompileCommand=exclude,org.openjdk.VirtualCall::test`

```
SHELL
Benchmark         (mode)  Mode  Cnt      Score     Error  Units
VirtualCall.test    mono  avgt    5  11598.390 ± 535.593  ns/op
VirtualCall.test    mega  avgt    5  11787.686 ± 884.384  ns/op
```

So, the megamorphic call does indeed cost something, but it is definitely not interpeter-bad performance. The difference between "mono" and "mega" in optimized case is basically the call overhead: we spend 3ns per element for "mega" case, while spending only 1ns per element in "mono" case.

How does "mega" case look like in `perfasm`? Like this, with many things pruned for brevity:

```
ASM
....[Hottest Region 1]..............................................................................
C2, org.openjdk.generated.VirtualCall_test_jmhTest::test_avgt_jmhStub, version 88 (143 bytes)

  6.93%    5.40%  ↗  0x...5c450: mov    0x40(%rsp),%r9
                  |  ...
  3.65%    4.31%  |  0x...5c47b: callq  0x...0bf60 ;*invokevirtual m
                  |                               ; - org.openjdk.VirtualCall::test@22 (line 76)
                  |                               ;   {virtual_call}
  3.12%    2.34%  |  0x...5c480: inc    %ebp
  3.33%    0.02%  |  0x...5c482: cmp    0x10(%rsp),%ebp
                  ╰  0x...5c486: jl     0x...5c450
                     ...
.....................................................................................................
 31.26%   21.77%  <total for region 1>

....[Hottest Region 2].............................................................................
C2, org.openjdk.VirtualCall$C1::m, version 84 (14 bytes) <--- mis-attributed :(

                     ...
                   Decoding VtableStub vtbl[5]@12
  3.95%    1.57%     0x...59bf0: mov    0x8(%rsi),%eax
  3.73%    3.34%     0x...59bf3: shl    $0x3,%rax
  3.73%    5.04%     0x...59bf7: mov    0x1d0(%rax),%rbx
 16.45%   22.42%     0x...59bfe: jmpq   *0x40(%rbx)        ; jump to target
                     0x...59c01: add    %al,(%rax)
                     0x...59c03: add    %al,(%rax)
                     ...
.....................................................................................................
 27.87%   32.37%  <total for region 2>

....[Hottest Region 3].............................................................................
C2, org.openjdk.VirtualCall$C3::m, version 86 (26 bytes)

# {method} {0x00007f75aaf4dd50} 'm' '()V' in 'org/openjdk/VirtualCall$C3'

                     ...
                   [Verified Entry Point]
 17.82%   26.04%     0x...595c0: sub    $0x18,%rsp
  0.06%    0.04%     0x...595c7: mov    %rbp,0x10(%rsp)
                     0x...595cc: incl   0x14(%rsi)       ; c3++
  3.53%    5.14%     0x...595cf: add    $0x10,%rsp
                     0x...595d3: pop    %rbp
  3.29%    5.10%     0x...595d4: test   %eax,0x9f01a26(%rip)
  0.02%    0.02%     0x...595da: retq
                     ...
.....................................................................................................
 24.73%   36.35%  <total for region 3>
```

So the benchmarking loop calls into something, which we can assume is the virtual call handler, then it ends up with VirtualStub, that is supposedly does what every other runtime does with virtual calls: jumps the the actual method with the help of Virtual Method Table (VMT) (https://en.wikipedia.org/wiki/Virtual_method_table).[7]

But wait a minute, this does not compute! The disassembly says we are actually calling to `0x…0bf60`, not into `VirtualStub` that is at `0x…59bf0`?! And that call is hot, so the call target should also be hot, right? This is where runtime itself plays tricks on us. Even if the compiler bails to optimize the virtual call, the runtime can handle "pessimistic" cases on its own. To diagnose this better, we need to get the fastdebug OpenJDK build (https://builds.shipilev.net/openjdk-jdkX/), and supply a tracing option for Inline Caches (IC) (https://en.wikipedia.org/wiki/Inline_caching): `-XX:+TraceIC`. Additionally, we want to save the Hotspot log to file with `-prof perfasm:saveLog=true`

Lo and behold!

```
$ grep IC org.openjdk.VirtualCall.test-AverageTime.log
    IC@0x00007fac4fcb428b: to megamorphic {method} {0x00007fabefa81880} 'm' ()V';
                                in 'org/openjdk/VirtualCall$C2'; entry: 0x00007fac4fcb2ab0
```
SHELL

Okay, it says inline cache had acted for the call-site at `0x00007fac4fcb428b`. Who is it? This is our Java call!

```
$ grep -A 4 0x00007fac4fcb428b: org.openjdk.VirtualCall.test-AverageTime.log
   0.02%    0x00007fac4fcb428b: callq  0x00007fac4fb7dda0
                                ;*invokevirtual m {reexecute=0 rethrow=0 return_oop=0}
                                ; - org.openjdk.VirtualCall::test@22 (line 76)
                                ;   {virtual_call}
```
SHELL

But what was the address in that Java call? This is the *resolving* runtime stub:

```
$ grep -C 2  0x00007fac4fb7dda0 org.openjdk.VirtualCall.test-AverageTime.log
                  0x00007fac4fb7dcdf: hlt
                Decoding RuntimeStub - resolve_virtual_call 0x00007fac4fb7dd10
                  0x00007fac4fb7dda0: push   %rbp
                  0x00007fac4fb7dda1: mov    %rsp,%rbp
                  0x00007fac4fb7dda4: pushfq
```
SHELL

This guy basically called to runtime, figured out what method we want to call, and then asked IC to **patch** the call to point to new resolved address! Since that is the one-time action, no wonder we do not see it as the hot code. IC action line mentions changing the entry to another address, which is, by the way, our actual VtableStub:

```
$ grep -C 4 0x00007fac4fcb2ab0: org.openjdk.VirtualCall.test-AverageTime.log
                Decoding VtableStub vtbl[5]@12
  8.94%    6.49%    0x00007fac4fcb2ab0: mov    0x8(%rsi),%eax
  0.16%    0.06%    0x00007fac4fcb2ab3: shl    $0x3,%rax
  0.20%    0.10%    0x00007fac4fcb2ab7: mov    0x1e0(%rax),%rbx
  2.34%    1.90%    0x00007fac4fcb2abe: jmpq   *0x40(%rbx)
                    0x00007fac4fcb2ac1: int3
```
SHELL

In the end, no runtime/compiler calls were needed to dispatch over resolved call: the call-site just calls the `VtableStub` that does the VMT dispatch — never leaving the generated machine code. This IC machinery would handle virtual monomorphic and static calls in the similar way, pointing to the stub/address that does not do VMT dispatch.

What we see in initial JMH perfasm output is the generated code as it was looking **after** the compilation, but **before** the execution and potential runtime patching.[8]

Observations

Just because compiler had failed to optimize for the best case, it does not mean the worst case is abysmally worse. True, you will give up some optimizations, but the overhead would not be so devastating that you would need to avoid virtual calls altogether. This rhymes with the "Black Magic of (Java) Method Dispatch" conclusion (https://shipilev.net/blog/2015/black-magic-method-dispatch/#_conclusion): unless you care very much, you don't have to worry about call performance.

JVM Anatomy Quark #17: Trust Nonstatic Final Fields

Question

Is there **any** case when JVM can trust non-static `final` fields?

Theory

As we have seen in "#15: Just In Time Constants" (https://shipilev.net/jvm/anatomy-quarks/15-just-in-time-constants/), compilers routinely trust `static final` fields, because the value is known not to depend on particular object, and it is known not to change. But what if we know the object identity well, e.g. the *reference itself* is in `static final`, can we then trust its `final` instance fields? For example:

```
JAVA
class M {
  final int x;
  M(int x) { this.x = x; }
}

static final M KNOWN_M = new M(1337);

void work() {
  // We know exactly the slot that holds the variable, can we just
  // inline the value 1337 here?
  return KNOWN_M.x;
}
```

The tricky question is, what happens if someone **changes** that field? Java Language Specification *allows* not seeing the update like this, because the fields is `final`. Unfortunately, real frameworks manage to depend on stronger behavior: the field update would be seen. The ongoing experiments with aggressively optimizing these cases and deoptimizing when the actual write happens were tried (https://bugs.openjdk.java.net/browse/JDK-8058164). The current state is that some internal classes are implicitly trusted (http://hg.openjdk.java.net/jdk/jdk/file/2c1af559e922/src/hotspot/share/ci/ciField.cpp#l203):

```
CPP
static bool trust_final_non_static_fields(ciInstanceKlass* holder) {
  if (holder == NULL)
    return false;
  if (holder->name() == ciSymbol::java_lang_System())
    // Never trust strangely unstable finals:  System.out, etc.
    return false;
  // Even if general trusting is disabled, trust system-built closures in these packages.
  if (holder->is_in_package("java/lang/invoke") || holder->is_in_package("sun/invoke"))
    return true;
  // Trust VM anonymous classes. They are private API (sun.misc.Unsafe) and can't be serialized,
  // so there is no hacking of finals going on with them.
  if (holder->is_anonymous())
    return true;
  // Trust final fields in all boxed classes
  if (holder->is_box_klass())
    return true;
  // Trust final fields in String
  if (holder->name() == ciSymbol::java_lang_String())
    return true;
  // Trust Atomic*FieldUpdaters: they are very important for performance, and make up one
  // more reason not to use Unsafe, if their final fields are trusted. See more in JDK-8140483.
  if (holder->name() == ciSymbol::java_util_concurrent_atomic_AtomicIntegerFieldUpdater_Impl() ||
      holder->name() == ciSymbol::java_util_concurrent_atomic_AtomicLongFieldUpdater_CASUpdater() ||
      holder->name() == ciSymbol::java_util_concurrent_atomic_AtomicLongFieldUpdater_LockedUpdater() ||
      holder->name() == ciSymbol::java_util_concurrent_atomic_AtomicReferenceFieldUpdater_Impl()) {
    return true;
  }
  return TrustFinalNonStaticFields;
}
```

...and regular `final` fields are only trusted when the experimental `-XX:+TrustFinalNonStaticFields` is provided.

Practice

Can we see this in practice? Using the modified JMH benchmark from "#15: Just In Time Constants" (https://shipilev.net/jvm/anatomy-quarks/15-just-in-time-constants/), but this time we use the `final` field from the object, not the object itself:

```java
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class TrustFinalFields {

    static final T t_static_final;
    static       T t_static;
           final T t_inst_final;
                 T t_inst;

    static {
        t_static_final = new T(1000);
        t_static = new T(1000);
    }

    {
        t_inst_final = new T(1000);
        t_inst = new T(1000);
    }

    static class T {
        final int x;

        public T(int x) {
            this.x = x;
        }
    }

    @Benchmark public int _static_final() { return 1000 / t_static_final.x; }
    @Benchmark public int _static()       { return 1000 / t_static.x;       }
    @Benchmark public int _inst_final()   { return 1000 / t_inst_final.x;   }
    @Benchmark public int _inst()         { return 1000 / t_inst.x;         }

}
```

In my machine, it yields:

```shell
Benchmark                       Mode  Cnt  Score   Error  Units
TrustFinalFields._inst          avgt   15  4.316 ± 0.003  ns/op
TrustFinalFields._inst_final    avgt   15  4.317 ± 0.002  ns/op
TrustFinalFields._static        avgt   15  4.282 ± 0.011  ns/op
TrustFinalFields._static_final  avgt   15  4.202 ± 0.002  ns/op
```

So it seems as if `static final` did not help much. Indeed, if you look into the generated code:

```asm
0.02%   ↗  movabs $0x782b67520,%r10   ; {oop(a 'org/openjdk/TrustFinalFields$T';)}
        |  mov    0x10(%r10),%r10d    ; get field $x
        |  ...
0.19%   |  cltd
0.02%   |  idiv   %r10d               ; idiv
        |  ...
0.16%   |  test   %r11d,%r11d         ; check and run @Benchmark again
        ↳  je     BACK
```

The object itself is trusted to be at given place in the heap (`$0x782b67520`), but we did not trust the field! Running the same with `-XX:+TrustFinalNonStaticFields` yields:

```shell
Benchmark                       Mode  Cnt  Score    Error  Units
TrustFinalFields._inst          avgt   15  4.318 ±  0.001  ns/op
TrustFinalFields._inst_final    avgt   15  4.317 ±  0.003  ns/op
TrustFinalFields._static        avgt   15  4.290 ±  0.002  ns/op
TrustFinalFields._static_final  avgt   15  1.901 ±  0.001  ns/op  # <--- !!!
```

...and here the `final` field is folded, as can be seen in perfasm output:

```asm
3.04%   ↗  mov    %r10,(%rsp)
        │  mov    0x38(%rsp),%rsi
8.26%   │  mov    $0x1,%edx           ; <--- constant folded to 1
        │  ...
0.04%   │  test   %r11d,%r11d         ; check and run @Benchmark again
        └  je     BACK
```

Observations

Trusting instance final fields requires knowing the object we are operating with. But even then, we may pragmatically do it when we are sure it does not break applications — so, minimally, for known system classes. Constant folding through these final fields is the corner-stone for performance story for `MethodHandle`-s, `VarHandle`-s`, `Atomic*FieldUpdaters` and other high-performance implementations from the core library. Applications may try to use the experimental VM options, but the potential breakage from misbehaving applications may severely dampen the benefits.

JVM Anatomy Quark #18: Scalar Replacement

Question

I have heard Hotspot can do stack allocation. Called Escape Analysis, and it is magical. Right?

Theory

This gets a fair bit of confusion. In "stack allocation", "allocation" seems to assume that the entire object is allocated on the stack instead of the heap. But what really happens is that *the compiler* performs the so called Escape Analysis (EA) (https://en.wikipedia.org/wiki/Escape_analysis), which can identify which newly created objects are not escaping into the heap, and then it can do a few interesting optimizations. Note that EA itself is *not* the optimization, it is the analysis phase that gives important pieces of data for the optimizer.[9]

One of the things that optimizer can do for non-escaping objects is to remap the accesses to the object fields to accesses to synthetic local operands:[10] perform *Scalar Replacement*. Since those operands are then handled by register allocator, some of them may claim stack slots (get "spilled") in current method activation, and it might *look* like the object field block is allocated on stack. But this is a false symmetry: operands may not even materialize at all, or may reside in registers, object header is not created at all, etc. The operands that get mapped from object field accesses might not even be contiguous on stack! This is different from stack allocation.

If stack allocation was really done, it would allocate the entire object storage on the stack, including the header and the fields, and reference it in the generated code. The caveat in this scheme is that once the object is *escaping*, we would need to copy the entire object block from the stack to the heap, because we cannot be sure current thread stays in the method and keeps this part of the stack holding the object alive. Which means we have to intercept *stores* to the heap, in case we ever store stack-allocated object — that is, do the GC write barrier.

Hotspot does not do stack allocations *per se*, but it does approximate that with Scalar Replacement.

Can we observe this in practice?

Practice

Consider this JMH benchmark. We create the object with a single field that is initialized off our input, and it reads the field right away, discarding the object:

JAVA

```
import org.openjdk.jmh.annotations.*;

@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class ScalarReplacement {

    int x;

    @Benchmark
    public int single() {
        MyObject o = new MyObject(x);
        return o.x;
    }

    static class MyObject {
        final int x;
        public MyObject(int x) {
            this.x = x;
        }
    }

}
```

If you run the test with `-prof gc`, you would notice it does not allocate anything:

```
Benchmark                                     Mode  Cnt     Score    Error   Units

ScalarReplacement.single                      avgt   15     1.919 ±  0.002   ns/op
ScalarReplacement.single:·gc.alloc.rate       avgt   15    ≈ 10⁻⁴            MB/sec
ScalarReplacement.single:·gc.alloc.rate.norm  avgt   15    ≈ 10⁻⁶              B/op
ScalarReplacement.single:·gc.count            avgt   15       ≈ 0            counts
```

`-prof perfasm` shows there is only a single access to field `x` left.

```
....[Hottest Region 1].............................................................
C2, level 4, org.openjdk.ScalarReplacement::single, version 459 (26 bytes)

                  [Verified Entry Point]
  6.05%    2.82%    0x00007f79e1202900: sub    $0x18,%rsp          ; prolog
  0.95%    0.78%    0x00007f79e1202907: mov    %rbp,0x10(%rsp)
  0.04%    0.21%    0x00007f79e120290c: mov    0xc(%rsi),%eax      ; get field $x
  5.80%    7.43%    0x00007f79e120290f: add    $0x10,%rsp          ; epilog
                    0x00007f79e1202913: pop    %rbp
 23.91%   33.34%    0x00007f79e1202914: test   %eax,0x17f0b6e6(%rip)
  0.21%    0.02%    0x00007f79e120291a: retq
.......................................................................................
```

Notice the magic of it: the compiler was able to detect that `MyObject` instance is not escaping, remapped its fields to local operands, and then (drum-roll) identified that successive store to that operand follows the load, and eliminated that store-load pair altogether — as it would do with local variables! Then, pruned the allocation, because it is not needed anymore, and any reminiscent of the object had evaporated.

Of course, that requires a sophisticated EA implementation to identify non-escaping candidates. When EA breaks, Scalar Replacement also breaks. The most trivial breakage in current Hotspot EA is when control flow merges before the access. For example, if we have two different objects (yet with the same content), under the branch that selects either of them, EA breaks, even though both objects are evidently (for us, humans) non-escaping:

```java
public class ScalarReplacement {

    int x;
    boolean flag;

    @Setup(Level.Iteration)
    public void shake() {
        flag = ThreadLocalRandom.current().nextBoolean();
    }

    @Benchmark
    public int split() {
        MyObject o;
        if (flag) {
            o = new MyObject(x);
        } else {
            o = new MyObject(x);
        }
        return o.x;
    }

   // ...
}
```

Here, the code allocates:

```
Benchmark                                      Mode  Cnt     Score    Error   Units

ScalarReplacement.single                       avgt   15     1.919 ±  0.002   ns/op
ScalarReplacement.single:·gc.alloc.rate        avgt   15    ≈ 10⁻⁴           MB/sec
ScalarReplacement.single:·gc.alloc.rate.norm   avgt   15    ≈ 10⁻⁶             B/op
ScalarReplacement.single:·gc.count             avgt   15       ≈ 0           counts

ScalarReplacement.split                        avgt   15     3.781 ±  0.116   ns/op
ScalarReplacement.split:·gc.alloc.rate         avgt   15  2691.543 ± 81.183  MB/sec
ScalarReplacement.split:·gc.alloc.rate.norm    avgt   15    16.000 ±  0.001    B/op
ScalarReplacement.split:·gc.count              avgt   15  1460.000           counts
ScalarReplacement.split:·gc.time               avgt   15   929.000               ms
```

If that was a "true" stack allocation, it would trivially handle this case: it'd extend the stack at runtime for either allocation, do the accesses, then scratch off the stack contents before leaving the method, and stack allocations would get retracted. The complication with write barriers that should guard object escapes still stands.

Observations

Escape analysis is an interesting compiler technique that enables interesting optimizations. Scalar Replacement is one of them, and it is not about putting the object storage on stack. Instead, it is about exploding the object and rewriting the code into local accesses, and optimizing them further, sometimes spilling these accesses on stack when register pressure is high. In many cases on critical hotpaths it can be successfully and profitably done.

But, EA is not ideal: if we cannot statically determine the object is not escaping, we have to assume it does. Complicated control flow may bail earlier. Calling non-inlined — and thus opaque for current analysis — instance method bails. Doing some things that rely on object identity bail, although trivial things like reference comparison with non-escaping objects gets folded efficiently.

This is not an ideal optimization, but when it works, it works magnificently well. Further improvements in compiler technology might widen the number of cases where EA works well.[11]

JVM Anatomy Quark #19: Lock Elision

Question

I have heard that JVM bails out any compiler optimization with locks, so if I write `synchronized`, this is what JVM has to do! Right?

Theory

With current Java Memory Model, unobserved locks are not guaranteed to have any memory effects (https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/#wishful-unobserved-sync). Among other things, this means that synchronization on non-shared objects is futile, and thus runtime does not have to do anything there. It still might, but not really required, and this opens up optimization opportunities.

Therefore, if escape analysis figures out the object is non-escaping, compiler is free to eliminate synchronization. Is that observable in practice?

Practice

Consider this simple JMH benchmark. We increment something with and without synchronization on new object:

```java
import org.openjdk.jmh.annotations.*;

@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class LockElision {

    int x;

    @Benchmark
    public void baseline() {
        x++;
    }

    @Benchmark
    public void locked() {
        synchronized (new Object()) {
            x++;
        }
    }
}
```

If we run this test, and enable `-prof perfnorm` profiler right away, this is what we shall see:

```shell
Benchmark                                  Mode  Cnt   Score    Error  Units

LockElision.baseline                       avgt   15   0.268 ±  0.001  ns/op
LockElision.baseline:CPI                   avgt    3   0.200 ±  0.009   #/op
LockElision.baseline:L1-dcache-loads       avgt    3   2.035 ±  0.101   #/op
LockElision.baseline:L1-dcache-stores      avgt    3  ≈ 10⁻³            #/op
LockElision.baseline:branches              avgt    3   1.016 ±  0.046   #/op
LockElision.baseline:cycles                avgt    3   1.017 ±  0.024   #/op
LockElision.baseline:instructions          avgt    3   5.076 ±  0.346   #/op

LockElision.locked                         avgt   15   0.268 ±  0.001  ns/op
LockElision.locked:CPI                     avgt    3   0.200 ±  0.005   #/op
LockElision.locked:L1-dcache-loads         avgt    3   2.024 ±  0.237   #/op
LockElision.locked:L1-dcache-stores        avgt    3  ≈ 10⁻³            #/op
LockElision.locked:branches                avgt    3   1.014 ±  0.047   #/op
LockElision.locked:cycles                  avgt    3   1.015 ±  0.012   #/op
LockElision.locked:instructions            avgt    3   5.062 ±  0.154   #/op
```

Whoa, the tests perform **exactly** the same: timing is the same, the number of loads, stores, cycles, instructions are the same. With high probability, this means that the generated code is the same. Indeed it is, and looks like this:

```asm
 14.50%   16.97%  ↗  incl   0xc(%r8)              ; increment field
 76.82%   76.05%  │  movzbl 0x94(%r9),%r10d       ; JMH infra: do another @Benchmark
  0.83%    0.10%  │  add    $0x1,%rbp
  0.47%    0.78%  │  test   %eax,0x15ec6bba(%rip)
  0.47%    0.36%  │  test   %r10d,%r10d
                  ╰  je     BACK
```

The lock is completely *elided*, there is nothing left out of allocation, out of synchronization, nothing. If we supply JVM flag `-XX:-EliminateLocks`, or we disable EA with `-XX:-DoEscapeAnalysis` (that breaks every optimization that depends on EA, including lock elision), then `locked` counters would balloon up:

```shell
Benchmark                                  Mode  Cnt   Score    Error  Units

LockElision.baseline                       avgt   15   0.268 ±  0.001  ns/op
LockElision.baseline:CPI                   avgt    3   0.200 ±  0.001   #/op
LockElision.baseline:L1-dcache-loads       avgt    3   2.029 ±  0.082   #/op
LockElision.baseline:L1-dcache-stores      avgt    3   0.001 ±  0.001   #/op
LockElision.baseline:branches              avgt    3   1.016 ±  0.028   #/op
LockElision.baseline:cycles                avgt    3   1.015 ±  0.014   #/op
LockElision.baseline:instructions          avgt    3   5.078 ±  0.097   #/op

LockElision.locked                         avgt   15  11.590 ±  0.009  ns/op
LockElision.locked:CPI                     avgt    3   0.998 ±  0.208   #/op
LockElision.locked:L1-dcache-loads         avgt    3  11.872 ±  0.686   #/op
LockElision.locked:L1-dcache-stores        avgt    3   5.024 ±  1.019   #/op
LockElision.locked:branches                avgt    3   9.027 ±  1.840   #/op
LockElision.locked:cycles                  avgt    3  44.236 ±  3.364   #/op
LockElision.locked:instructions            avgt    3  44.307 ±  9.954   #/op
```

...and show the cost of allocation and trivial synchronization.

Observations

Lock elision is another optimization that is enabled by escape analysis, and it removes some superfluous synchronization. This is especially profitable when internally synchronized implementations are not escaping into the wild: then, we can dispense with synchronization completely! This is a Zen of compiler optimizations — if no one ever sees the synchronized lock, does it make a sound?

JVM Anatomy Quark #20: FPU Spills

Question

I look into JVM-generated machine code, and see weird XMM register usages on x86, when my code has no floating-point or vector operations at all. What is up with that?

Theory

FPU and vector units are ubiquitous in modern CPUs, and in many cases they provide the alternative sets of registers for FPU-specific operations. For example, SSE and AVX extensions in Intel x86_64 have additional set of wide XMM, YMM and ZMM registers that can be used in conjunction with wider instructions.

While the non-vector instruction set is not usually orthogonal with vector and non-vector registers (for example, we cannot use general-purpose IMUL with XMM register on x86_64), these registers still provide an interesting **storage** option: we can temporarily store data there, even if that data is not used for vector operations. [12]

Enter register allocation. The register allocator duty is to take the program representation with all the operands the program needs in a particular compilation unit (method, for example), and map these virtual operands to actual machine registers — *allocate registers* for them. In many real programs, the number of live virtual operands at given program location is greater than the number of machine registers available. At that point, register allocator has to put some operands out of the registers somewhere else — e.g. on stack — that is, *spill* the operands.

Now, we have 16 general purpose registers on x86_64 (not all of them are usable), and 16 more AVX registers on most modern machines. Can we spill to XMM registers instead of the stack? Yes, we can! Does it bring any benefit?

Practice

Consider this simple JMH benchmark. We construct that benchmark in a very special way (assume Java has pre-processing capabilities, for simplicity):

```java
import org.openjdk.jmh.annotations.*;

import java.util.concurrent.TimeUnit;

@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class FPUSpills {

    int s00, s01, s02, s03, s04, s05, s06, s07, s08, s09;
    int s10, s11, s12, s13, s14, s15, s16, s17, s18, s19;
    int s20, s21, s22, s23, s24;

    int d00, d01, d02, d03, d04, d05, d06, d07, d08, d09;
    int d10, d11, d12, d13, d14, d15, d16, d17, d18, d19;
    int d20, d21, d22, d23, d24;

    int sg;
    volatile int vsg;

    int dg;

    @Benchmark
#ifdef ORDERED
    public void ordered() {
#else
    public void unordered() {
#endif
        int v00 = s00; int v01 = s01; int v02 = s02; int v03 = s03; int v04 = s04;
        int v05 = s05; int v06 = s06; int v07 = s07; int v08 = s08; int v09 = s09;
        int v10 = s10; int v11 = s11; int v12 = s12; int v13 = s13; int v14 = s14;
        int v15 = s15; int v16 = s16; int v17 = s17; int v18 = s18; int v19 = s19;
        int v20 = s20; int v21 = s21; int v22 = s22; int v23 = s23; int v24 = s24;
#ifdef ORDERED
        dg = vsg; // Confuse optimizer a little
#else
        dg = sg;  // Just a plain store...
#endif
        d00 = v00; d01 = v01; d02 = v02; d03 = v03; d04 = v04;
        d05 = v05; d06 = v06; d07 = v07; d08 = v08; d09 = v09;
        d10 = v10; d11 = v11; d12 = v12; d13 = v13; d14 = v14;
        d15 = v15; d16 = v16; d17 = v17; d18 = v18; d19 = v19;
        d20 = v20; d21 = v21; d22 = v22; d23 = v23; d24 = v24;
    }
}
```

It reads and writes multiple pairs of fields at once. Optimizers are not actually tied up (https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/#wishful-hb-actual) to the particular program order. Indeed, that is what we would observe in `unordered` test:

```shell
Benchmark                                Mode  Cnt   Score    Error  Units

FPUSpills.unordered                      avgt   15   6.961 ±  0.002  ns/op
FPUSpills.unordered:CPI                  avgt    3   0.458 ±  0.024   #/op
FPUSpills.unordered:L1-dcache-loads      avgt    3  28.057 ±  0.730   #/op
FPUSpills.unordered:L1-dcache-stores     avgt    3  26.082 ±  1.235   #/op
FPUSpills.unordered:cycles               avgt    3  26.165 ±  1.575   #/op
FPUSpills.unordered:instructions         avgt    3  57.099 ±  0.971   #/op
```

This gives us around 26 load-store pairs, which corresponds roughly to 25 pairs we have in the test. But we don't have 25 general purpose registers! Perfasm reveals that optimizer had merged load-store pairs close to each other, so that register pressure is much lower:

```asm
  0.38%    0.28%  ↗  movzbl 0x94(%rcx),%r9d
                  |  ...
  0.25%    0.20%  |  mov    0xc(%r11),%r10d    ; getfield s00
  0.04%    0.02%  |  mov    %r10d,0x70(%r8)    ; putfield d00
                  |  ...
                  |  ... (transfer repeats for multiple vars) ...
                  |  ...
                  └  je     BACK
```

At this point, we want to cheat the optimizer a little, and make a point of confusion so that all loads are performed well before the stores. This is what `ordered` test does, and there, we can see the loads and stores are happening in bulk: first all the loads, then all the stores. The register pressure is highest at the point where all the loads have completed, but none of the stores have started yet. Even then, we have no significant difference against `unordered`:

```shell
Benchmark                                  Mode  Cnt   Score    Error  Units

FPUSpills.unordered                        avgt   15   6.961 ±  0.002  ns/op
FPUSpills.unordered:CPI                    avgt    3   0.458 ±  0.024   #/op
FPUSpills.unordered:L1-dcache-loads        avgt    3  28.057 ±  0.730   #/op
FPUSpills.unordered:L1-dcache-stores       avgt    3  26.082 ±  1.235   #/op
FPUSpills.unordered:cycles                 avgt    3  26.165 ±  1.575   #/op
FPUSpills.unordered:instructions           avgt    3  57.099 ±  0.971   #/op

FPUSpills.ordered                          avgt   15   7.961 ±  0.008  ns/op
FPUSpills.ordered:CPI                      avgt    3   0.329 ±  0.026   #/op
FPUSpills.ordered:L1-dcache-loads          avgt    3  29.070 ±  1.361   #/op
FPUSpills.ordered:L1-dcache-stores         avgt    3  26.131 ±  2.243   #/op
FPUSpills.ordered:cycles                   avgt    3  30.065 ±  0.821   #/op
FPUSpills.ordered:instructions             avgt    3  91.449 ±  4.839   #/op
```

...and that is **because** we have managed to spill operands to *XMM registers*, not on stack:

```asm
  3.08%    3.79%  ↗  vmovq  %xmm0,%r11
                  |  ...
  0.25%    0.20%  |  mov    0xc(%r11),%r10d    ; getfield s00
  0.02%           |  vmovd  %r10d,%xmm4        ; <--- FPU SPILL
  0.25%    0.20%  |  mov    0x10(%r11),%r10d   ; getfield s01
  0.02%           |  vmovd  %r10d,%xmm5        ; <--- FPU SPILL
                  |  ...
                  |  ... (more reads and spills to XMM registers) ...
                  |  ...
  0.12%    0.02%  |  mov    0x60(%r10),%r13d   ; getfield s21
                  |  ...
                  |  ... (more reads into registers) ...
                  |  ...
                  |  ------- READS ARE FINISHED, WRITES START ------
  0.18%    0.16%  |  mov    %r13d,0xc4(%rdi)   ; putfield d21
                  |  ...
                  |  ... (more reads from registers and putfileds)
                  |  ...
  2.77%    3.10%  |  vmovd  %xmm5,%r11d        : <--- FPU UNSPILL
  0.02%           |  mov    %r11d,0x78(%rdi)   ; putfield d01
  2.13%    2.34%  |  vmovd  %xmm4,%r11d        ; <--- FPU UNSPILL
  0.02%           |  mov    %r11d,0x70(%rdi)   ; putfield d00
                  |  ...
                  |  ... (more unspills and putfields)
                  |  ...
                  └  je     BACK
```

Notice that we do use general-purpose registers (GPRs) for some operands, but when they are depleted, we spill. "Then" is ill-defined here, because we appear to *first* spill, and then use GPRs, but this is a false appearance, because register allocators may operate on the complete graph.[13].

The latency of XMM spills seems minimal: even though we do claim more instructions for spills, they execute very efficiently and fill the pipelining gaps: with 34 additional instructions, which means around 17 spill pairs, we have claimed only 4 additional cycles. Note that it would be incorrect to compute the CPI as 4/34 = ~0.11 clk/insn, which would be larger than current CPUs are capable of. But the improvement is real, because we use execution blocks we weren't using before.

The claims of efficiency mean nothing, if we don't have anything to compare with. But here, we do! We can instruct Hotspot to avoid using FPU spills with `-XX:-UseFPUForSpilling`, which gives us the idea how much do we win with XMM spills:

```
SHELL
Benchmark                                  Mode  Cnt   Score    Error  Units

# Default
FPUSpills.ordered                          avgt   15   7.961 ±  0.008  ns/op
FPUSpills.ordered:CPI                      avgt    3   0.329 ±  0.026   #/op
FPUSpills.ordered:L1-dcache-loads          avgt    3  29.070 ±  1.361   #/op
FPUSpills.ordered:L1-dcache-stores         avgt    3  26.131 ±  2.243   #/op
FPUSpills.ordered:cycles                   avgt    3  30.065 ±  0.821   #/op
FPUSpills.ordered:instructions             avgt    3  91.449 ±  4.839   #/op

# -XX:-UseFPUForSpilling
FPUSpills.ordered                          avgt   15  10.976 ±  0.003  ns/op
FPUSpills.ordered:CPI                      avgt    3   0.455 ±  0.053   #/op
FPUSpills.ordered:L1-dcache-loads          avgt    3  47.327 ±  5.113   #/op
FPUSpills.ordered:L1-dcache-stores         avgt    3  41.078 ±  1.887   #/op
FPUSpills.ordered:cycles                   avgt    3  41.553 ±  2.641   #/op
FPUSpills.ordered:instructions             avgt    3  91.264 ±  7.312   #/op
```

Oh, see the increased load/store counters per operation? These are stack spills: the stack itself, while fast, still resides in memory, and thus accesses to stack land in L1 cache. It is roughly the same 17 additional spill pairs, but now they take ~11 cycles. The throughput of L1 cache is the limiting factor here.

Finally, we can eyeball the perfasm output for `-XX:-UseFPUForSpilling`:

```
ASM
  2.45%    1.21%  ↗  mov    0x70(%rsp),%r11
                  |  ...
  0.50%    0.31%  |  mov    0xc(%r11),%r10d    ; getfield s00
  0.02%           |  mov    %r10d,0x10(%rsp)   ; <--- stack spill!
  2.04%    1.29%  |  mov    0x10(%r11),%r10d   ; getfield s01
                  |  mov    %r10d,0x14(%rsp)   ; <--- stack spill!
                  |  ...
                  |  ... (more reads and spills to stack) ...
                  |  ...
  0.12%    0.19%  |  mov    0x64(%r10),%ebp    ; getfield s22
                  |  ...
                  |  ... (more reads into registers) ...
                  |  ...
                  |  ------- READS ARE FINISHED, WRITES START ------
  3.47%    4.45%  |  mov    %ebp,0xc8(%rdi)    ; putfield d22
                  |  ...
                  |  ... (more reads from registers and putfields)
                  |  ...
  1.81%    2.68%  |  mov    0x14(%rsp),%r10d   ; <--- stack unspill
  0.29%    0.13%  |  mov    %r10d,0x78(%rdi)   ; putfield d01
  2.10%    2.12%  |  mov    0x10(%rsp),%r10d   ; <--- stack unspill
                  |  mov    %r10d,0x70(%rdi)   ; putfield d00
                  |  ...
                  |  ... (more unspills and putfields)
                  |  ...
                  ↳  je     BACK
```

Yup, the stack spills are at the similar places where we had XMM spills.

Observations

FPU spills are the nice trick to alleviate register pressure problems. While it does not increase the number of registers available for general operations, it does provide a faster temporary storage for spills: so when we need just a few additional spill slots, we can avoid tripping to L1 cache-backed stack for this.

This is sometimes the cause of interesting performance deviations: if FPU spills are not used on some critical path, we may see diminished performance. For example, introducing a slow-path GC barrier call that is assumed to trash the FPU registers may tell compiler to get back to usual stack-based spills, without trying anything fancy.

In Hotspot, `-XX:+UseFPUForSpilling` is enabled by default for SSE-capable x86 platforms, ARMv7, and AArch64. So, this works with most of your programs, whether you know about this trick or not.

JVM Anatomy Quark #21: Heap Uncommit

Question

I want my memory back. That was not a question.

Theory

JVM uses memory for different reasons, to store its internal VM state in native memory, as well as providing the storage for Java objects ("Java heap"). We have seen the native memory part of the story in "Native Memory Tracking" (https://shipilev.net/jvm/anatomy-quarks/12-native-memory-tracking/), but the major contender in many applications is the Java heap itself.

Java heap is normally managed by automatic memory manager, sometimes called *garbage collector*. [14] Naive GCs would allocate the large block of memory from the underlying OS memory manager, and slice it themselves for accepting allocations. This immediately means that even if there are only a few Java objects in the heap, from the perspective of OS the JVM process had acquired all the possible memory for the Java heap.[15]

So, if we want to have unused parts of Java heap returned back to OS, we need cooperation from the GC.

There are two ways to achieve this cooperation: do more frequent GCs instead of "expanding" the Java heap to `-Xmx`; or explicit uncommit of unused parts of Java heap, even after Java heap is inflated to `-Xmx`. First way helps only so much, and usually in earlier phases of application lifetime — eventually, applications would like to allocate a lot. In this piece, we would concentrate on the second part, what to do when heap is already inflated.

What do modern GCs do on this front?

Experimental Setup

Footprint measurement is tricky, because we have to define what footprint actually is. Since we are talking about the footprint from the perspective of OS, it makes most sense to measure the RSS of the entire JVM process, which would include both native VM memory and Java heap.

The other significant question is when to measure the footprint. It stands to reason that the amount of application data in different phases of application lifecycle is different. That is *especially* true when application deliberately optimizes for footprint, with lazy/delayed operations that only happen when the actual work comes along. The easiest mistake to make while capacity planning for footprint is to start such application, snapshot its footprint, and then blow all estimates when the actual work comes in.

Automatic memory managers usually react on what is happening to application: they trigger GCs based on allocation pressure, free space availability, idleness, etc. Measuring footprint only in active phase is probably not very telling either. This is further exarcebated by observation that most applications in the world (outside the high load servers) are idle most of the time, or run on low duty cycle.

All this means we need to have the application going through different lifecycle phases to see the faces of the memory footprint story. Let us take simple spring-boot-petclinic project (https://github.com/spring-projects/spring-petclinic) and run it with different GCs. These are the configurations we use:

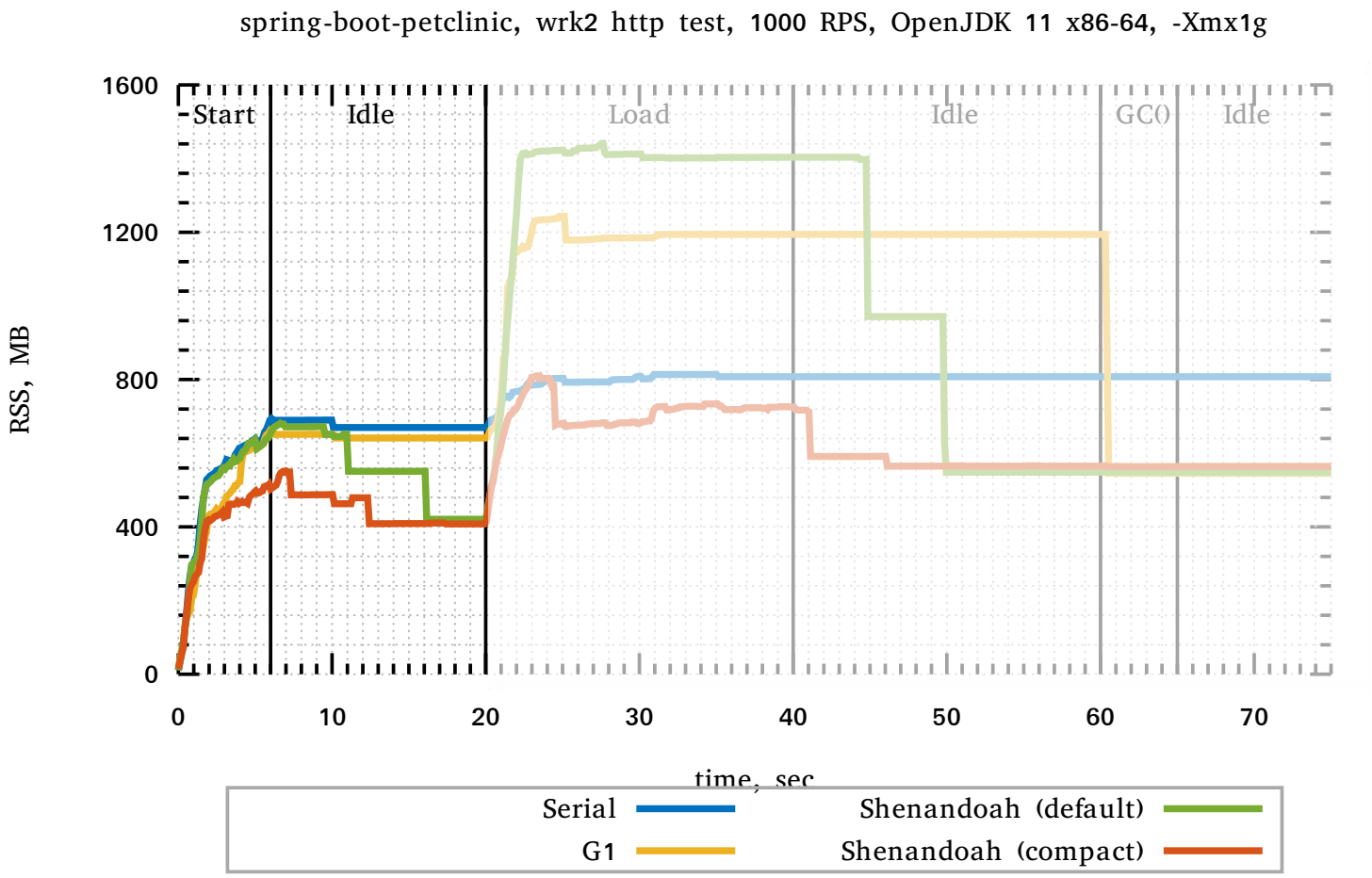
- Serial GC: the go-to GC for small-heap applications. It has low native overhead, a bit more aggressive GC policies, etc;
- G1 GC: the workhorse of OpenJDK, default since JDK 9;
- Shenandoah GC (https://wiki.openjdk.java.net/display/shenandoah/Main): the concurrent GC from Red Hat. We include it here to show some behaviors footprint-savvy GC would have. [16] For the purpose of this experiment, Shenandoah runs in two modes: **default** mode, and **compact** mode that tunes collector for lowest footprint. [17]

The experiment is driven by this simple script. We use OpenJDK 11, as decently recent JDK, but the same can be demonstrated with OpenJDK 8, as GC behaviors are not significantly different between 8 and 11 in this test.

Results and Discussion

Start+Idle

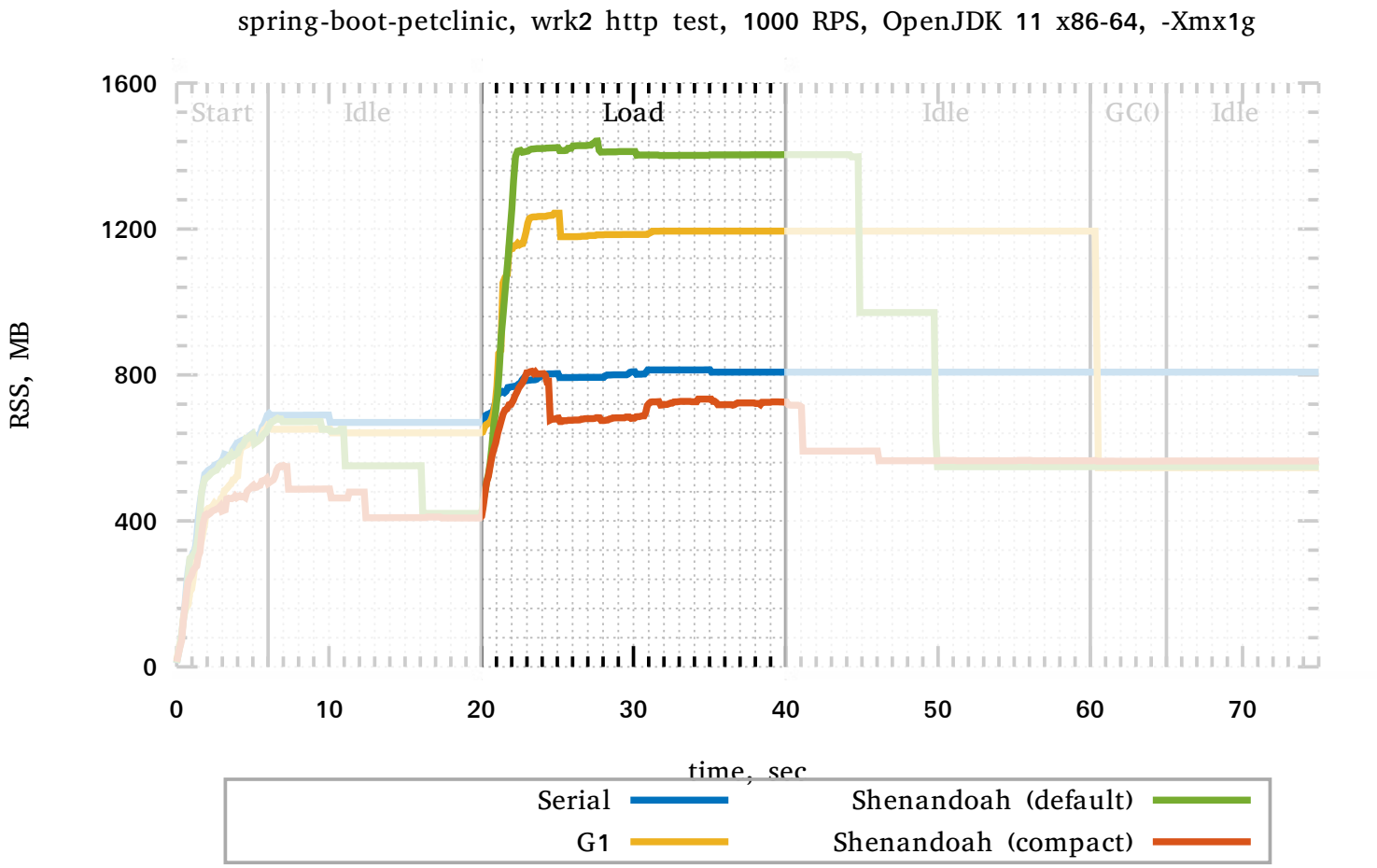
Let us digest the RSS charts. What can we see here?



During startup, all GCs try to cope with small initial heap, and many do frequent GCs. This keeps them from inflating the heap too much. After initial active phase is done, workloads stabilize on some particular footprint level. In absence of any GC triggers, **this level would be largerly defined by heuristics used for triggering the GC** during startup, even if the amount of data stored in heap is the same. This gets especially quirky when heuristics has to guess what user wanted from the acupuncture of 100+ GC options.

Load

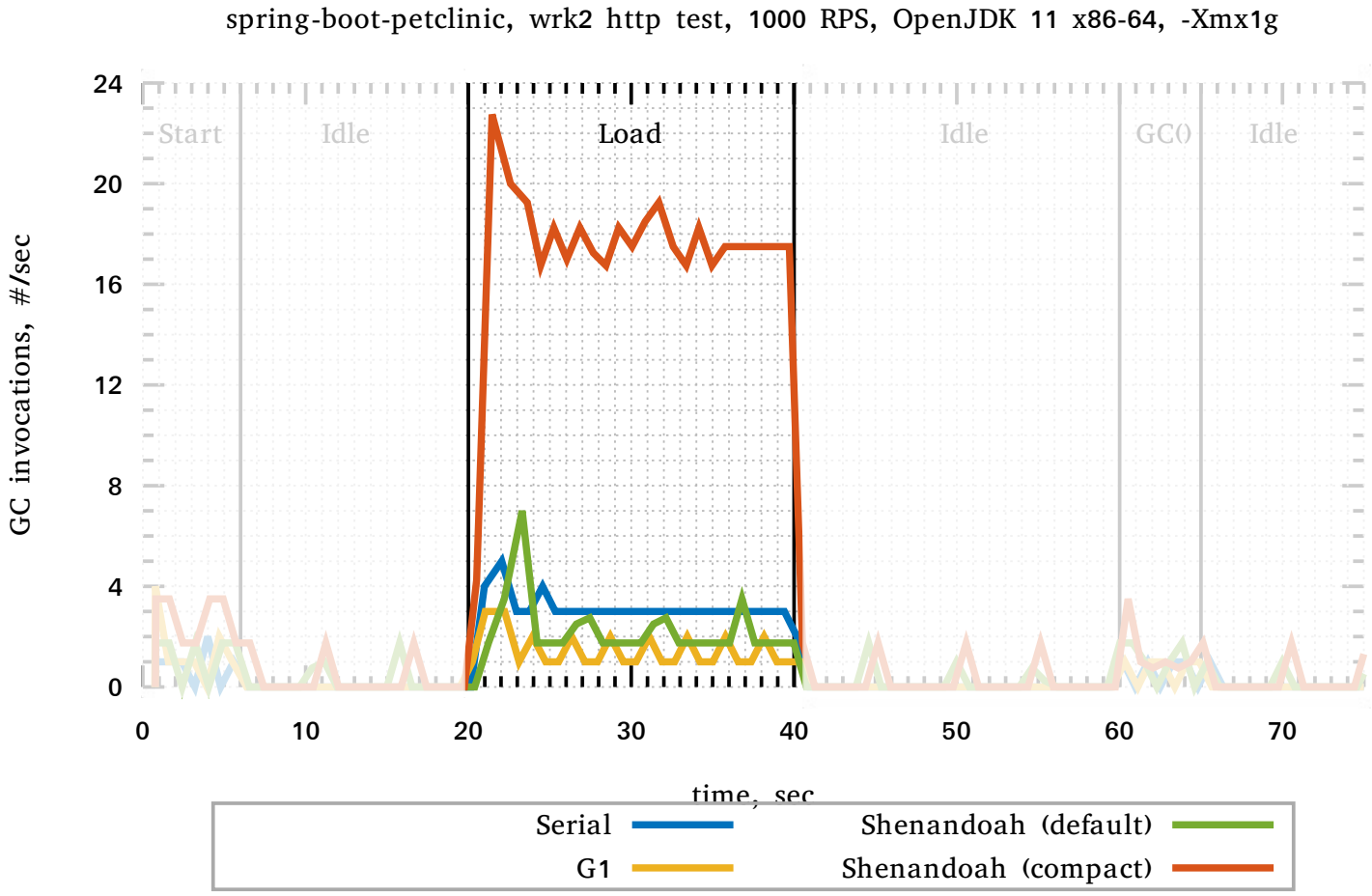
Same RSS chart as above, repeated for convenience:



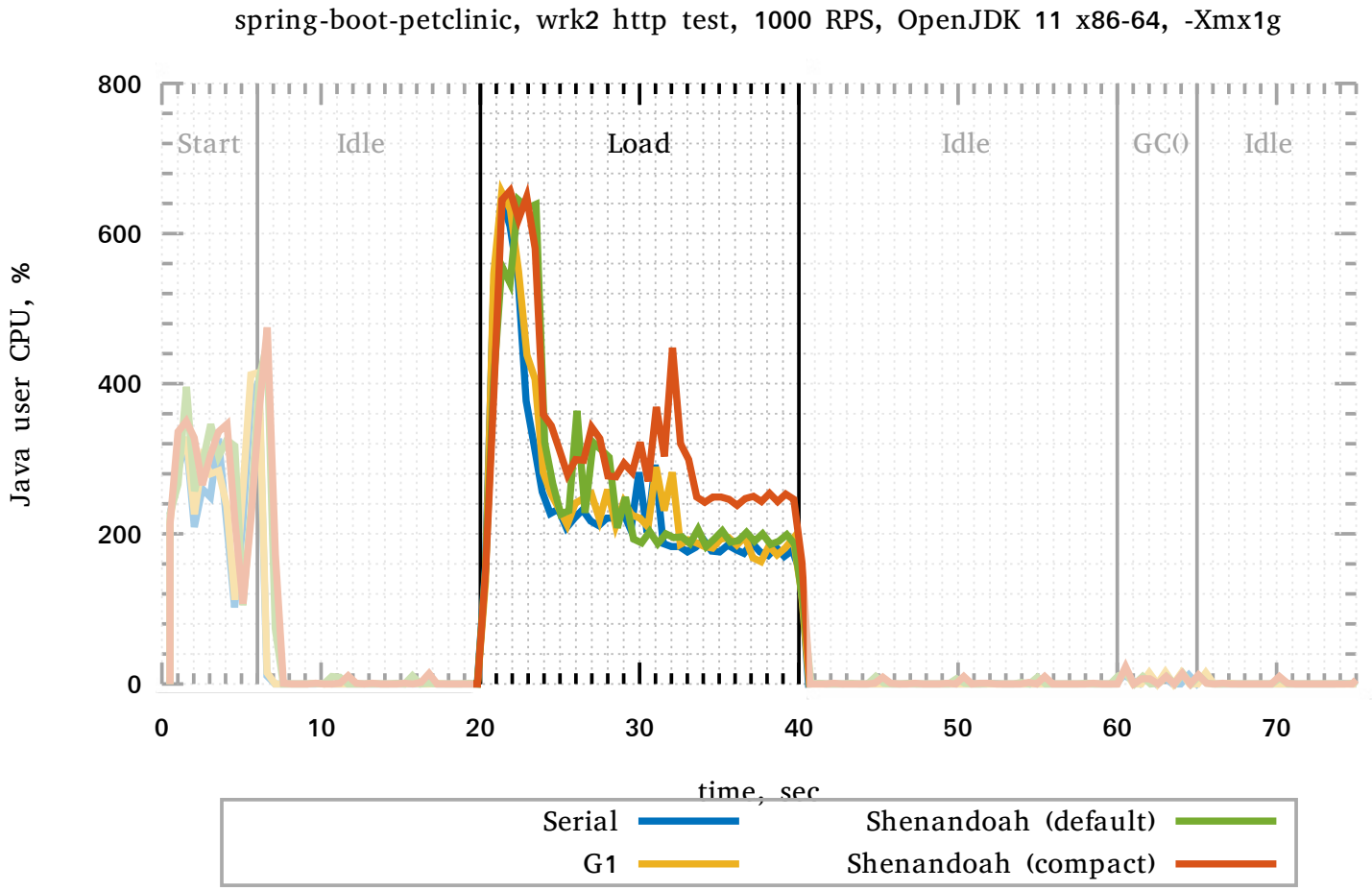
When load comes, GC heuristics again have to decide a few things. Depending on GC, its implementation and configuration, **it has to decide whether to expand the heap, or do more aggressive GC cycles**.

Here, Serial GC decided to perform more cycles. G1 inflated to around 3/4 of the max heap, and started doing moderately frequent cycles to cope with allocation pressure. Shenandoah in default mode, being a concurrent GC running in dense heap, opted to inflate the heap as much as possible to maintain application concurrency without too frequent cycles. Shenandoah in compact mode, being instructed to maintain low footprint, opted to make much more aggressive cycles.

This is corroborated by the actual GC frequency logs:



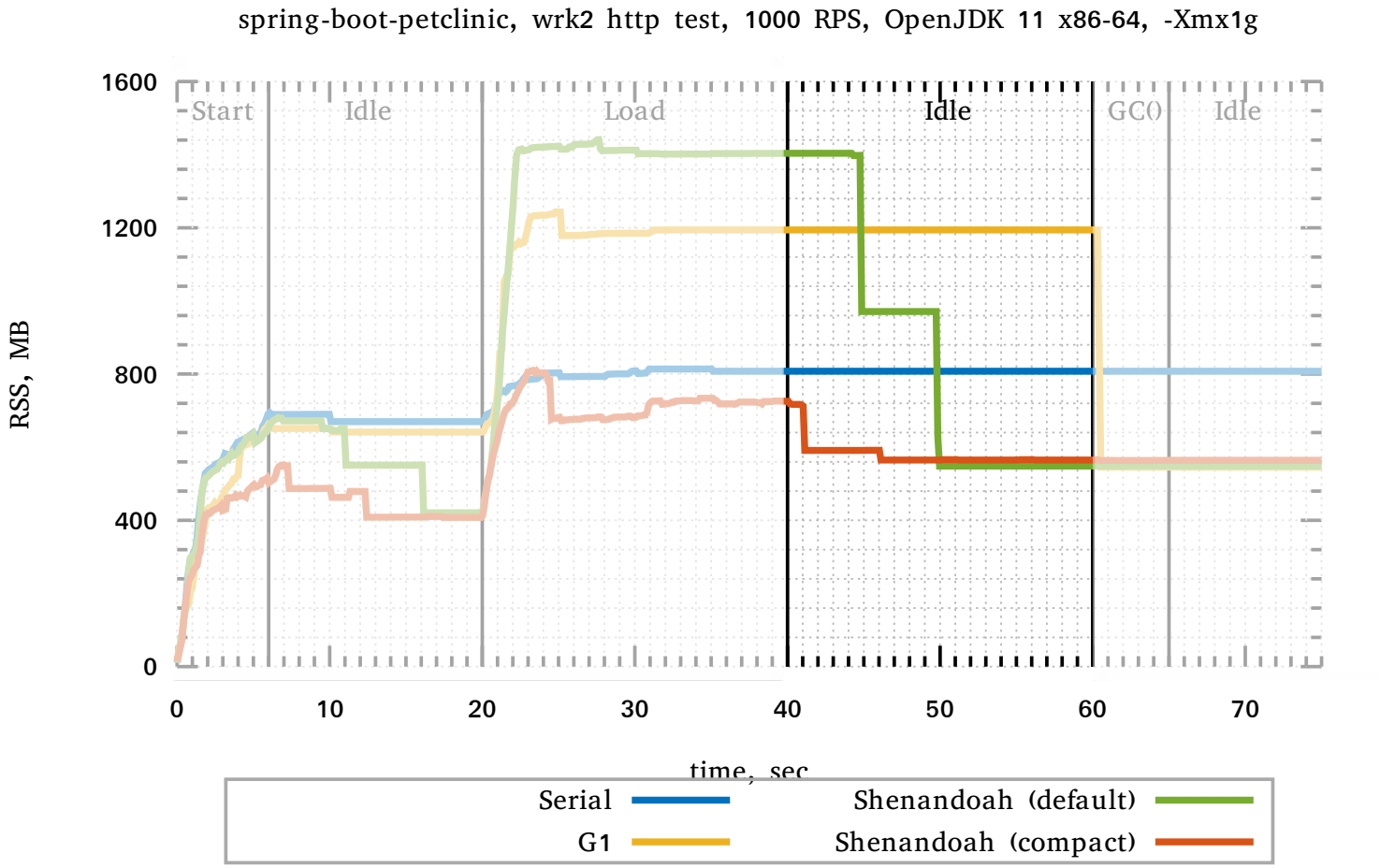
More frequent GC cycles also mean more CPU needed to deal with GC work:



While most of the lines are noisy here, we can clearly see "Shenandoah (compact)" taking quite a some additional time to work. That is the price we have to pay to have the denser footprint. Or, in other words, **this is the manifestation of throughput-latency-footprint** tradeoff. There are, of course, tunable settings to say how much we want to trade, and this experiment only shows the difference between two rather polar defaults: prefer throughput and prefer footprint. Since Shenandoah is concurrent GC, even performing effectively back-to-back GCs does not stall application all that much.

Idle

Same RSS chart as above, repeated for convenience:



When application comes idle, GCs may decide to return some resources. **The obvious thing to do would be *uncommitting* parts of the empty heap.** This is rather simple to do if heap is already sliced in independent chunks, for example when you have a regionalized collector like G1 or Shenandoah. Still, the GC has to decide if/when to do it.

Many OpenJDK GCs perform GC-related actions only in conjunction with the actual GC cycles. But an interesting thing happens. Most OpenJDK GCs are *allocation-triggered*, which means they start the cycle when a particular heap occupancy had been reached. If application went into idle state abruptly, it means it also stopped allocating, so whatever occupancy level it is at right now, would linger until something happens. It makes some sense for *stop-the-world GCs*, because we do not really want to start long-ish GC pause just because we feel like it.

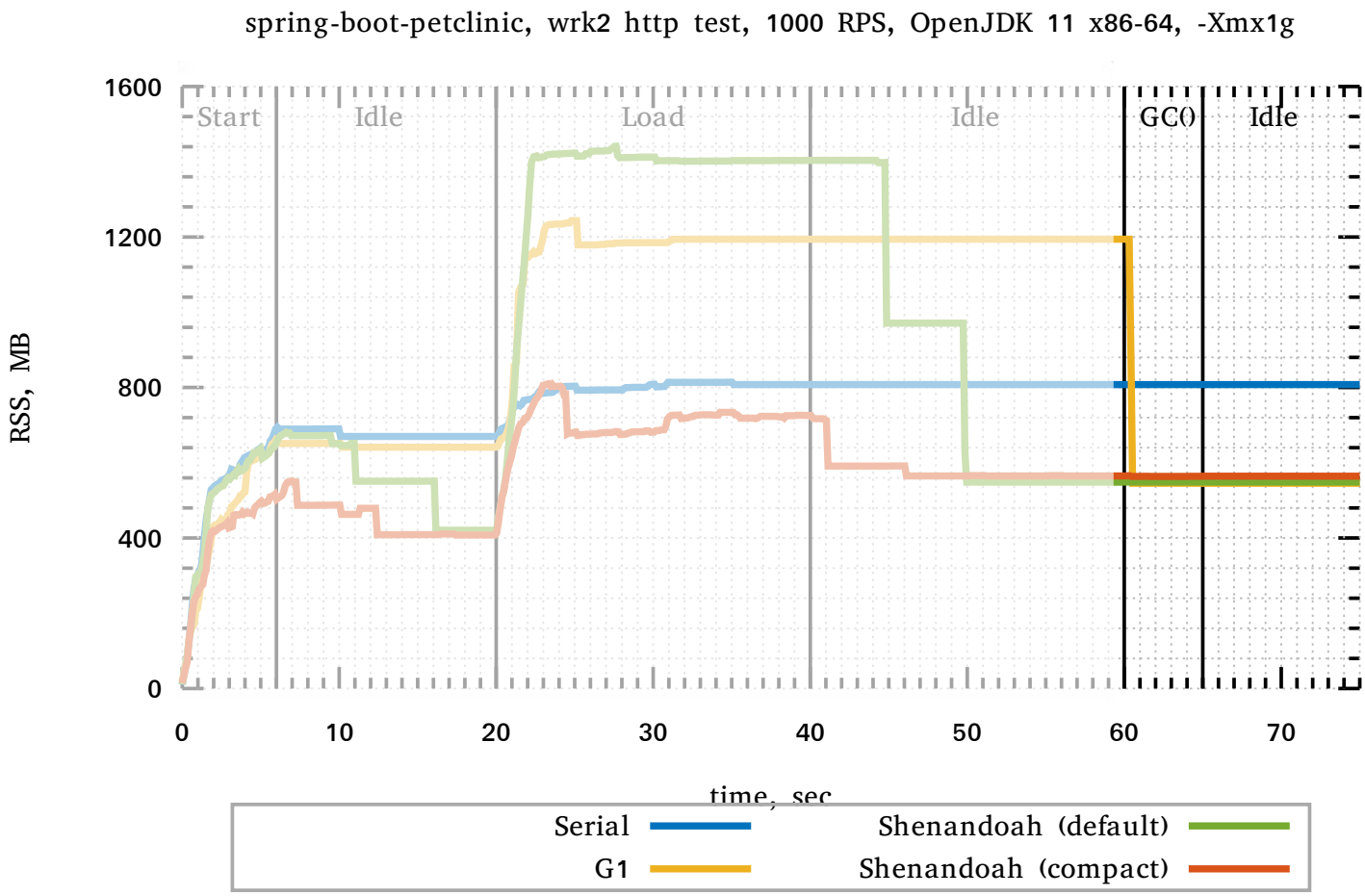
There is no particular need to hook up uncommit to the GC cycle to begin with. In the case of Shenandoah, there is an **asynchronous periodic uncommit**, and we can see it in action as the first large drop in idle phase. For this experiment, the uncommit delay was deliberately set at 5 seconds, and we can see it indeed happened after a few seconds in idle. This performed uncommit on regions that were emptied the last GC cycle, and have not been allocated yet.

But, there is another significant part of the story: since application went to idle abruptly, there is some floating garbage that we would like to collect. **This provides the motivation for having a periodic GC that should knock out the lingering garbage out.** Periodic GC is responsible for the second big step down in idle phase. It frees up new regions for periodic uncommit to deal with later.

If GC cycles were frequent enough already (see "Shenandoah (compact)"), the effect of all this is largerly irrelevant, as footprint is already quite low, and nothing excessive had been committed on top.

Full GC

Same RSS chart as above, repeated for convenience:



Again, doing periodic GCs with concurrent GC implementation is less intrusive to do: if load is back up when we are mid-GC-cycle, nothing bad is going to happen. That is in contrast to STW GC, that would have to guess if performing a major GC cycle is a good idea or not. In worst case, we would have to explicitly tell JVM to perform it, and at least G1 reacts to this request reliably. Note how the footprint for most collectors is down to the same level after Full GC, and how periodic GC and uncommit got there much earlier without user intervention.

Conclusion

Periodic GCs. Perfoming periodic GC cycles help to knock out lingering garbage. Concurrent GCs routinely perform periodic GC cycles: Shenandoah and ZGC are known to do it. G1 is supposed to gain this feature in JDK 12 with JEP 346 (http://openjdk.java.net/jeps/346). Otherwise, one can employ the external or internal agent to call for GC periodically when time is right, with the hard part of defining what is the right time. See, for example, Jelastic GC Agent (https://docs.jelastic.com/garbage-collector-agent).

Heap uncommit. Many GCs already do heap uncommits when they think it is a good idea: Shenandoah does it asynchronously even without the GC requests, G1 sure does it on explicit GC requests, pretty sure Serial and Parallel also do it in some conditions. ZGC is going to do it (http://mail.openjdk.java.net/pipermail/zgc-dev/2018-October/000489.html) soon as well, let's hope JDK 12. G1 is supposed to deal with synchronicity by performing periodic GC cycles with JEP 346 (http://openjdk.java.net/jeps/346) in JDK 12. Of course, there is a trade-off: committing memory back may take a while (http://mail.openjdk.java.net/pipermail/hotspot-gc-dev/2018-June/022206.html), so practical implementations would impose some timeouts before uncommits.

Footprint-targeted GCs. Many GCs provide flexible options to make GC cycles more frequent to optimize for footprint. Even something like increasing the frequency of periodic GCs would help to knock the garbage out earlier. Some GCs may give you the pre-canned configuration packages that instruct the implementation to make footprint-savvy choices, including configuring more frequent/periodic GC cycles and uncommits, like Shenandoah's "compact" mode.

Every time you see switching to some GC implementation made the footprint happy, do understand why and how it did so. This would help you to clearly understand what you paid for it, and also whether you can achieve the same without any migration.

JVM Anatomy Quark #22: Safepoint Polls

Questions

- How does JVM stop the Java threads for stop-the-world?
- What are those weird `test` instructions in my hot loops? (https://stackoverflow.com/a/54055300/2613885)
- Why Java 11 suddenly makes empty methods slower? (https://stackoverflow.com/a/54010845/2613885)

All these questions have the same answer.

Theory

Suppose you have the managed runtime like JVM, and you need to stop the Java threads occasionally to run some runtime code. For example, you want to do the stop-the-world GC. You can wait for all threads to eventually call into JVM, for example, ask for allocation (usually, a TLAB (https://shipilev.net/jvm/anatomy-quarks/4-tlab-allocation/) refill), or enter some native method (where transition to native would capture it), or do something else. But that is not guaranteed to happen! What if the thread is currently running in busy-loop of some kind, never doing anything special?

Well, on most machines, stopping the running thread is actually simple: you can send it a signal, force processor interrupt, etc. to make it stop what the thread is doing and transfer control to somewhere else. However, it usually not enough for the Java thread to stop at arbitrary points, especially if you want the precise garbage collection. There, you want to know what is in the registers and stack, in case those values are actually object references you need to deal with. Or, if you want to unbias the lock, you *want* to have precise information about the state of the thread and acquired locks. Or, if you deoptimize the method, you really want to do it from the safe location without losing already executed part of the code and/or temporary values.

Therefore, modern JVMs, like Hotspot, implement the cooperative scheme: threads ask every so often if they should transfer the control to VM, at some known points in their lifetime, when their state is known. When all threads stop at those known points, the VM is said to reach the *safepoint*. The pieces of code that check for safepoint requests are therefore known as *safepoint polls*.

The implementation needs to satisfy the interesting tradeoff: safepoint polls almost never fire, so they should be very efficient when not triggered. Can we glimpse it in the experiments?

Practice

Consider this simple JMH benchmark:

```java
import org.openjdk.jmh.annotations.*;

import java.util.concurrent.TimeUnit;

@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class EmptyBench {
    @Benchmark
    public void emptyMethod() {
        // This method is intentionally left blank.
    }
}
```

You might *think* this benchmark measures the empty method, but in reality it measures the minimal infrastructure code that services the benchmark: counts the iterations and waits for the iteration time to be over. Fortunately, that piece of code is rather fast, and so it can be dissected in full with the help of `-prof perfasm`.

This is out-of-the-box OpenJDK 8u191:

```asm
3.60%  ↗  ...a2: movzbl 0x94(%r8),%r10d       ; load "isDone" field
0.63%  |  ...aa: add    $0x1,%rbp             ; iterations++;
32.82% |  ...ae: test   %eax,0x1765654c(%rip) ; global safepoint poll
58.14% |  ...b4: test   %r10d,%r10d           ; if !isDone, do the cycle again
       ↳  ...b7: je     ...a2
```

The empty method got inlined, and everything evaporated out of it, only the infrastructure remains.

See that "global safepoint poll"? When safepoint is needed, JVM would arm the "polling page",[18] so any attempt to read that page would trigger the segmentation fault (SEGV) (https://en.wikipedia.org/wiki/Segmentation_fault). When SEGV finally fires from this safepoint poll, the control would be passed to any existing SEGV handlers first, and JVM has one ready! See, for example, how `JVM_handle_linux_signal` does (http://hg.openjdk.java.net/jdk/jdk/file/af7afdababd3/src/hotspot/os_cpu/linux_x86/os_linux_x86.cpp#l430) it (http://hg.openjdk.java.net/jdk/jdk/file/af7afdababd3/src/hotspot/os_cpu/linux_x86/os_linux_x86.cpp#l577).

The goal of all those tricks is to make the safepoint polls as cheap as possible, because they need to happen in many places, and they almost always **do not** fire. For this reason, the `test %eax, (addr)` is used: it has no effects when safepoint poll is not triggered.[19] It is also has very compact encoding, "only" 6 bytes on x86_64. The polling page address is fixed for a given JVM process, so the code generated by JIT in that process can use RIP-relative addressing (https://en.wikipedia.org/wiki/Addressing_mode#PC-relative_2): it says that the page is at given offset from the current instruction pointer, saving the need to spend precious bytes encoding the absolute 8-byte address.

There is also normally a single polling page that handles all threads at once, so generated code does not have to disambiguate which thread is currently running. But what if VM wants to stop individual threads? That is the question answered by JEP-312: "Thread-Local Handshakes" (https://openjdk.java.net/jeps/312). It provides the VM capability to trigger the *handshake* poll for the individual thread, which is currently implemented by assigning the individual polling page for *each thread*, and poll instruction reading that page address from thread-local storage.[20][21]

This is out-of-the-box OpenJDK 11.0.1:

```asm
0.31%  ↗  ...70: movzbl 0x94(%r9),%r10d   ; load "isDone" field
0.19%  |  ...78: mov    0x108(%r15),%r11  ; reading the thread-local poll page addr
25.62% |  ...7f: add    $0x1,%rbp         ; iterations++;
35.10% |  ...83: test   %eax,(%r11)       ; thread-local handshake poll
34.91% |  ...86: test   %r10d,%r10d       ; if !isDone, do the cycle again
       ↳  ...89: je     ...70
```

This is purely a runtime consideration, so this can be disabled with `-XX:-ThreadLocalHandshakes`, and the generated code would then be the same as in 8u191. This explains why this benchmark performs differently on 8 and 11 (let us run it under `-prof perfnorm` right away):

```
Benchmark                              Mode  Cnt  Score   Error  Units

# 8u191
EmptyBench.test                        avgt   15   0.383 ±  0.007  ns/op
EmptyBench.test:CPI                    avgt    3   0.203 ±  0.014   #/op
EmptyBench.test:L1-dcache-load-misses  avgt    3  ≈ 10⁻⁴            #/op
EmptyBench.test:L1-dcache-loads        avgt    3   2.009 ±  0.291   #/op
EmptyBench.test:cycles                 avgt    3   1.021 ±  0.193   #/op
EmptyBench.test:instructions           avgt    3   5.024 ±  0.229   #/op

# 11.0.1
EmptyBench.test                        avgt   15   0.590 ±  0.023  ns/op ; +0.2 ns
EmptyBench.test:CPI                    avgt    3   0.260 ±  0.173   #/op
EmptyBench.test:L1-dcache-loads        avgt    3   3.015 ±  0.120   #/op ; +1 load
EmptyBench.test:L1-dcache-load-misses  avgt    3  ≈ 10⁻⁴            #/op
EmptyBench.test:cycles                 avgt    3   1.570 ±  0.248   #/op ; +0.5 cycles
EmptyBench.test:instructions           avgt    3   6.032 ±  0.197   #/op ; +1 instruction

# 11.0.1, -XX:-ThreadLocalHandshakes
EmptyBench.test                        avgt   15   0.385 ±  0.007  ns/op
EmptyBench.test:CPI                    avgt    3   0.205 ±  0.027   #/op
EmptyBench.test:L1-dcache-loads        avgt    3   2.012 ±  0.122   #/op
EmptyBench.test:L1-dcache-load-misses  avgt    3  ≈ 10⁻⁴            #/op
EmptyBench.test:cycles                 avgt    3   1.030 ±  0.079   #/op
EmptyBench.test:instructions           avgt    3   5.031 ±  0.299   #/op
```

So, thread-local handshakes add another L1-hitting load, which costs around half a cycle. This also gives us some ground to estimate the cost of the safepoint poll itself: it is the L1-hitting load itself, and it probably takes another half a cycle.

Observations

Safepoint and handshake polls are interesting bits of trivia in managed runtime implementations. They are frequently visible on hotpath in the generated code, and they sometimes affect the performance, especially in the tight loops. Yet, their existence is necessary for runtime to implement important features like precise garbage collection, locking optimizations, deoptimization, etc.

There are lots of safepoint-related optimizations which we shall discuss separately.

JVM Anatomy Quark #23: Compressed References

Questions

- What is the size of Java reference anyway?
- What are compressed oops/references?
- What are the problems around compressed references?

Naive Approach

Java specification is silent on the storage size for the data types. Even for primitives, it only mandates the ranges the primitive types should definitely support (https://docs.oracle.com/javase/specs/jls/se11/html/jls-4.html#jls-4.2.1) and their behavior of operations, but not the actual storage size. This, for example, allows `boolean` fields to take 1, 2, 4 bytes in some implementations.

The question of Java references size is murkier, because specification is also silent about what the Java reference *is*, leaving this decision to the JVM implementation. Most JVM implementations translate Java references to machine pointers, without additional indirections, which simplifies the performance story.

For example, for the simple JMH benchmark like this:

JAVA
```
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class CompressedRefs {

    static class MyClass {
        int x;
        public MyClass(int x) { this.x = x; }
        public int x() { return x; }
    }

    private MyClass o = new MyClass(42);

    @Benchmark
    @CompilerControl(CompilerControl.Mode.DONT_INLINE)
    public int access() {
        return o.x();
    }

}
```

...the access to the field would look like this: [22]

ASM
```
....[Hottest Region 3]..............................................
c2, level 4, org.openjdk.CompressedRefs::access, version 712 (35 bytes)
         [Verified Entry Point]
  1.10%    ...b0: mov    %eax,-0x14000(%rsp) ; prolog
  6.82%    ...b7: push   %rbp                ;
  0.33%    ...b8: sub    $0x10,%rsp          ;
  1.20%    ...bc: mov    0x10(%rsi),%r10     ; get field "o" to %r10
  5.60%    ...c0: mov    0x10(%r10),%eax     ; get field "o.x" to %eax
  7.21%    ...c4: add    $0x10,%rsp          ; epilog
  0.50%    ...c8: pop    %rbp
  0.54%    ...c9: mov    0x108(%r15),%r10    ; thread-local handshake
  0.60%    ...d0: test   %eax,(%r10)
  6.63%    ...d3: retq                       ; return %eax
```

Notice the accesses to fields, both reading the reference field `CompressedRefs.o` and the primitive field `MyClass.x` are just dereferencing the regular machine pointer. The field is at offset 16 from the beginning of the object, this is why we read at `0x10`. This can be verified by looking into the memory representation of the `CompressedRefs` instance. We would see the reference field takes 8 bytes on 64-bit VM, and it is indeed at offset 16: [23]

```
$ java ... -jar ~/utils/jol-cli.jar internals -cp target/bench.jar org.openjdk.CompressedRefs
...
# Running 64-bit HotSpot VM.
# Objects are 8 bytes aligned.
# Field sizes by type: 8, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 8, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

Instantiated the sample instance via default constructor.

org.openjdk.CompressedRefs object internals:
 OFFSET  SIZE     TYPE DESCRIPTION        VALUE
      0     4          (object header)    01 00 00 00
      4     4          (object header)    00 00 00 00
      8     4          (object header)    f0 e8 1f 57
     12     4          (object header)    34 7f 00 00
     16     8  MyClass CompressedRefs.o   (object)
Instance size: 24 bytes
```
SHELL

Compressed References

But does that mean the size of Java reference is the same as the machine pointer width? Not necessarily. Java objects are usually quite reference-heavy, and there is pressure for runtimes to employ the optimizations that make the references smaller. The most ubiquitous trick is to *compress the references*: make their representation smaller than the machine pointer width. In fact, the example above was executed with that optimization explicitly disabled.

Since Java runtime environment is in full control of internal representation, this can be done without changing any user programs. It is possible to do in other environments, but you would need to handle the leakage through ABIs, etc, see for example X32 ABI (https://en.wikipedia.org/wiki/X32_ABI).

In Hotspot, due to a historical accident, the internal names had leaked to the VM options list that control this optimization. In Hotspot, the references to Java objects are called *"ordinary object pointers"*, or *"oops"*, which is why Hotspot VM options have these weird names: `-XX:+UseCompressedOops`, `-XX:+PrintCompressedOopsMode`, `-Xlog:gc+heap+coops`. In this post we would try to use the proper nomenclature, where possible.

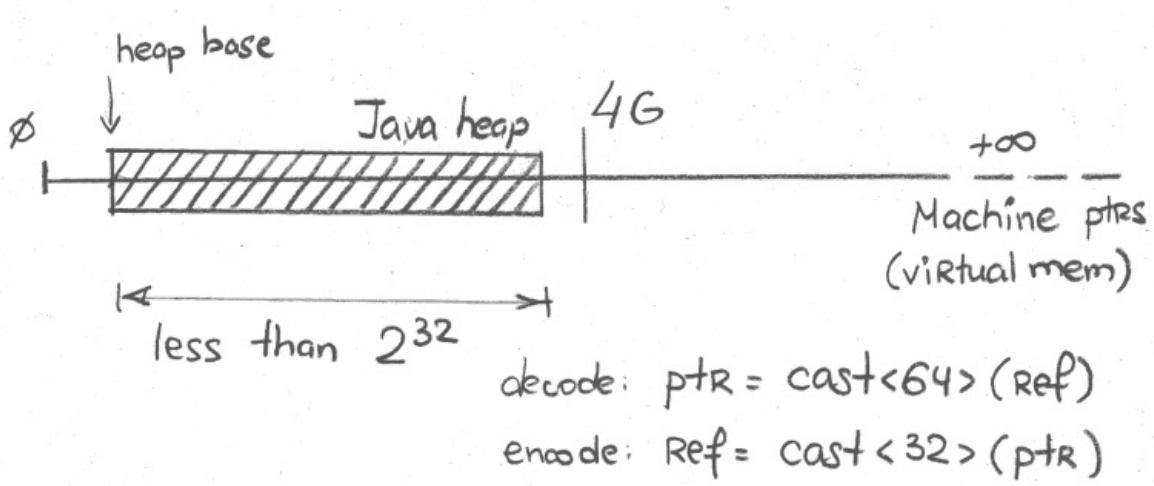
"32-bit" Mode

On most heap sizes, the higher bits of 64-bit machine pointer are usually zero. On the heap that can be mapped over the first 4 GB of virtual memory, higher 32 bits are definitely zero. In that case, we can just use the lower 32-bit to store the reference in 32-bit machine pointer. In Hotspot, this is called "32-bit" mode, as can be seen with logging:

```
$ java -Xmx2g -Xlog:gc+heap+coops ...
[0.016s][info][gc,heap,coops] Heap address: 0x0000000080000000, size: 2048 MB, Compressed Oops mode: 32-bit
```
SHELL

This whole shebang is obviously possible when heap size is less than 4 GB (or, 2^{32} bytes). Technically, the heap start address might be far away from zero address, and so the actual limit is lower than 4 GB. See the "Heap Address" in logging above. It says that heap starts at 0x0000000080000000 mark, closer to 2 GB.

Graphically, it can be sketched like this:



Now, the reference field only takes 4 bytes *and* the instance size is down to 16 bytes:[24]

```
SHELL
$ java -Xmx1g -jar ~/utils/jol-cli.jar internals -cp target/bench.jar org.openjdk.CompressedRefs
# Running 64-bit HotSpot VM.
# Using compressed oop with 0-bit shift.
# Using compressed klass with 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

Instantiated the sample instance via default constructor.

org.openjdk.CompressedRefs object internals:
 OFFSET  SIZE      TYPE DESCRIPTION        VALUE
      0     4           (object header)    01 00 00 00
      4     4           (object header)    00 00 00 00
      8     4           (object header)    85 fd 01 f8
     12     4   MyClass CompressedRefs.o   (object)
Instance size: 16 bytes
```

In generated code, the access looks like this:

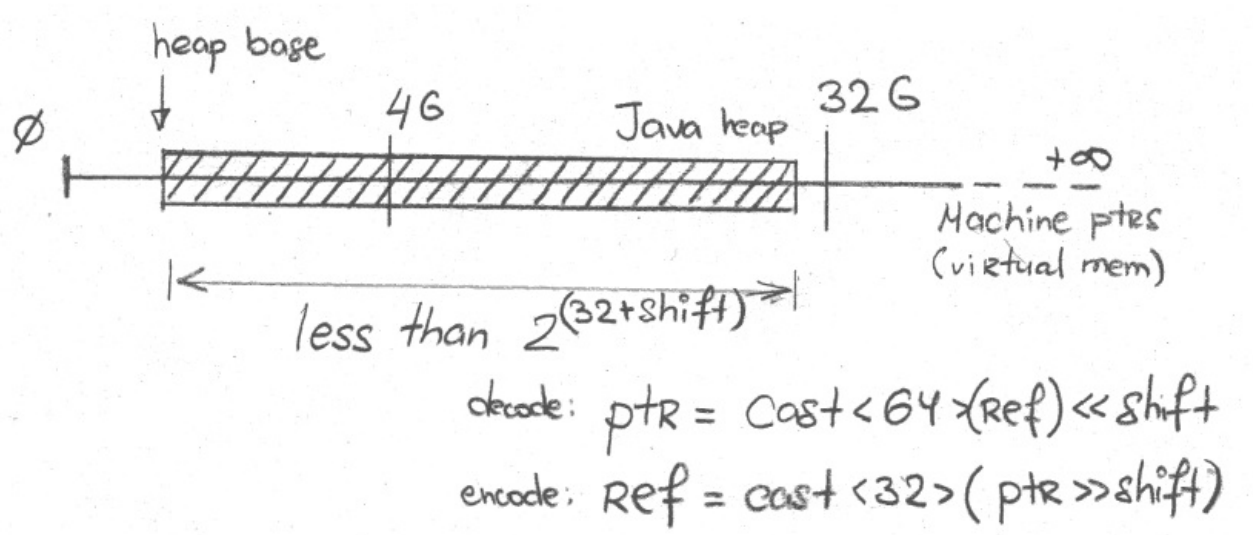
```
ASM
....[Hottest Region 2]..............................................
c2, level 4, org.openjdk.CompressedRefs::access, version 714 (35 bytes)
         [Verified Entry Point]
  0.87%    ...c0: mov    %eax,-0x14000(%rsp)  ; prolog
  6.90%    ...c7: push   %rbp
  0.35%    ...c8: sub    $0x10,%rsp
  1.74%    ...cc: mov    0xc(%rsi),%r11d      ; get field "o" to %r11
  5.86%    ...d0: mov    0xc(%r11),%eax       ; get field "o.x" to %eax
  7.43%    ...d4: add    $0x10,%rsp           ; epilog
  0.08%    ...d8: pop    %rbp
  0.54%    ...d9: mov    0x108(%r15),%r10     ; thread-local handshake
  0.98%    ...e0: test   %eax,(%r10)
  6.79%    ...e3: retq                        ; return %eax
```

See, the access is still in the same form, that is because the hardware itself just accepts the 32-bit pointer and extends it to 64 bits when doing the access. We have got this optimization for almost free.

"Zero-Based" Mode

But what if we cannot fit the untreated reference into 32 bits? There is a way out as well, and it exploits the fact that objects are *aligned*: objects always start at some multiple of alignment. So, the lowest bits of untreated reference representation are always zero. This opens up the way to use those bits for storing *significant* bits that did not fit into 32 bits. The easiest way to do that is to *bit-shift-right* the reference bits, and this gives us $2^{(32+\text{shift})}$ bytes of heap encodeable into 32 bits.

Graphically, it can be sketched like this:



With default object alignment of 8 bytes, shift is 3 ($2^3 = 8$), therefore we can represent the references to 2^{35} = 32 GB heap. Again, the same problem with base heap address surfaces here and makes the actual limit a bit lower.

In Hotspot, this mode is called "zero based compressed oops", see for example:

```
$ java -Xmx20g -Xlog:gc+heap+coops ...
[0.010s][info][gc,heap,coops] Heap address: 0x0000000300000000, size: 20480 MB, Compressed Oops mode: Zero based, Oop
shift amount: 3
```

SHELL

The access via the reference is now a bit more complicated:

```
....[Hottest Region 3]..............................................
c2, level 4, org.openjdk.CompressedRefs::access, version 715 (36 bytes)
         [Verified Entry Point]
  0.94%    ...40: mov    %eax,-0x14000(%rsp)    ; prolog
  7.43%    ...47: push   %rbp
  0.52%    ...48: sub    $0x10,%rsp
  1.26%    ...4c: mov    0xc(%rsi),%r11d        ; get field "o"
  6.08%    ...50: mov    0xc(%r12,%r11,8),%eax  ; get field "o.x"
  6.94%    ...55: add    $0x10,%rsp             ; epilog
  0.54%    ...59: pop    %rbp
  0.27%    ...5a: mov    0x108(%r15),%r10       ; thread-local handshake
  0.57%    ...61: test   %eax,(%r10)
  6.50%    ...64: retq
```

ASM

Getting the field `o.x` involves executing `mov 0xc(%r12,%r11,8),%eax`: "Taketh the ref'rence from %r11, multiplyeth the ref'rence by 8, addeth the heapeth base from %r12, and that wouldst be the objecteth that you can now readeth at offset `0xc`; putteth that value into `%eax`, please". In other words, this instruction combines the decoding of the compressed reference with the access through it, and it is done in one sway. In zero-based mode, `%r12` is zero, but it is easier on code generator to emit the access involving `%r12` nevertheless. The fact that `%r12` is zero in this mode can be used by code generator in other places too.

To simplify the internal implementation, Hotspot usually carries only uncompressed references in registers, and that is why the access to field `o` is just the plain access from `this` (that is in `%rsi`) at offset `0xc`.

"Non-Zero Based" Mode

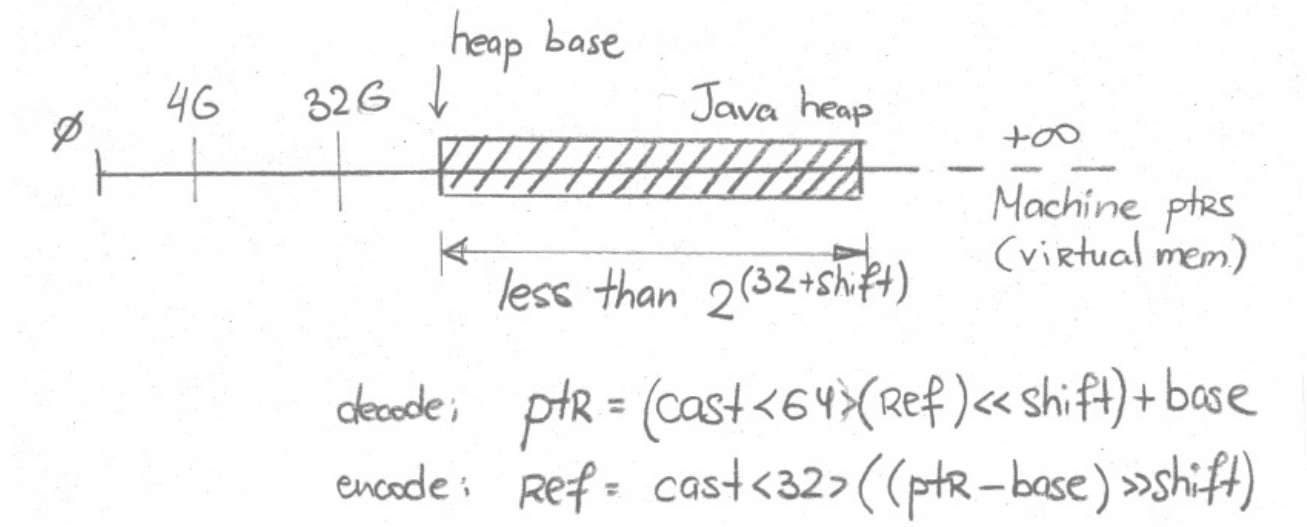
But zero-based compressed references still rely on assumption that heap is mapped at lower addresses. If it is not, we can just make heap base address non-zero for decoding. This would basically do the same thing as zero-based mode, but now heap base would mean more and participate in actual encoding/decoding.

In Hotspot, this mode is called "Non-zero base" mode, and you can see it in logs like this: [25]

```
$ java -Xmx20g -XX:HeapBaseMinAddress=100G -Xlog:gc+heap+coops
[0.015s][info][gc,heap,coops] Heap address: 0x0000001900400000, size: 20480 MB, Compressed Oops mode: Non-zero based:
0x0000001900000000, Oop shift amount: 3
```

SHELL

Graphically, it can be sketched like this:



As we suspected earlier, the access would look the same as in zero-based mode:

```
....[Hottest Region 1]..............................................
c2, level 4, org.openjdk.CompressedRefs::access, version 706 (36 bytes)
         [Verified Entry Point]
  0.08%    ...50: mov    %eax,-0x14000(%rsp)    ; prolog
  5.99%    ...57: push   %rbp
  0.02%    ...58: sub    $0x10,%rsp
  0.82%    ...5c: mov    0xc(%rsi),%r11d        ; get field "o"
  5.14%    ...60: mov    0xc(%r12,%r11,8),%eax  ; get field "o.x"
 28.05%    ...65: add    $0x10,%rsp             ; epilog
            ...69: pop    %rbp
  0.02%    ...6a: mov    0x108(%r15),%r10       ; thread-local handshake
  0.63%    ...71: test   %eax,(%r10)
  5.91%    ...74: retq                          ; return %eax
```

ASM

See, the same thing. Why wouldn't it be. The only hidden difference here is that `%r12` is now carrying the non-zero heap base value.

Limitations

The obvious limitation is the heap size. Once the heap size gets larger than the threshold under which compressed references are working, a surprising thing happens: references suddenly become uncompressed and take twice as much memory. Depending on how many references you have in the heap, you can have a significant increase in the perceived heap occupancy.

To illustrate that, let's estimate how much heap is actually taken by allocating some objects, with the toy example like this:

```
import java.util.stream.IntStream;

public class RandomAllocate {
    static Object[] arr;

    public static void main(String... args) {
        int size = Integer.parseInt(args[0]);
        arr = new Object[size];
        IntStream.range(0, size).parallel().forEach(x -> arr[x] = new byte[(x % 20) + 1]);
        System.out.println("All done.");
    }
}
```

JAVA

It is much more convenient to run with Epsilon GC (https://openjdk.java.net/jeps/318), which would fail on heap exhaustion, rather than trying to GC its way out. There is no point in GC-ing this example, because all objects are reachable. Epsilon would also print heap occupancy stats for our convenience.[26]

Let's take some reasonable amount of small objects. 800M objects sounds enough? Run:

```
SHELL
$ java -XX:+UseEpsilonGC -Xlog:gc -Xlog:gc+heap+coops -Xmx31g RandomAllocate 800000000
[0.004s][info][gc] Using Epsilon
[0.004s][info][gc,heap,coops] Heap address: 0x0000001000001000, size: 31744 MB, Compressed Oops mode: Non-zero disjoint
base: 0x0000001000000000, Oop shift amount: 3
All done.
[2.380s][info][gc] Heap: 31744M reserved, 26322M (82.92%) committed, 26277M (82.78%) used
```

There, we took 26 GB to store those objects, good. Compressed references got enabled, so the references to those `byte[]` arrays are smaller now. But let's suppose our friends who admin the servers said to themselves: "Hey, we have a gigabyte or two we can spare for our Java installation", and have bumped the old `-Xmx31g` to `-Xmx33g`. Then this happens:

```
SHELL
$ java -XX:+UseEpsilonGC -Xlog:gc -Xlog:gc+heap+coops -Xmx33g RandomAllocate 800000000
[0.004s][info][gc] Using Epsilon
Terminating due to java.lang.OutOfMemoryError: Java heap space
```

Oopsies. Compressed references got disabled, because heap size is too large. References became larger, and the dataset does not fit anymore. I would say this again: *the same dataset* does not fit anymore *just because* we requested the excessively large heap size, even though *we don't even use it*.

If we try to figure out what is the minimum heap size required to fit the dataset after 32 GB, this would be the minimum:

```
SHELL
$ java -XX:+UseEpsilonGC -Xlog:gc -Xlog:gc+heap+coops -Xmx36g RandomAllocate 800000000
[0.004s][info][gc] Using Epsilon
All done.
[3.527s][info][gc] Heap: 36864M reserved, 35515M (96.34%) committed, 35439M (96.13%) used
```

See, we used to take ~26 GB for the dataset, now we are taking ~35 GB, almost 40% increase!

Conclusions

Compressed references is a nice optimization that keeps memory footprint at bay for reference-heavy workloads. The improvements provided by this optimization can be very impressive. But so can be the surprises when this enabled-by-default optimization stops working due to heap size and/or other environmental problems.

Knowing how this optimization works, when it breaks, and how to deal with breakages is important as heap sizes reach the interesting thresholds of 4 GB and 32 GB. There are ways to alleviate this breakage by fiddling with object alignment, which we would take on in "Object Alignment" quark (https://shipilev.net/jvm/anatomy-quarks/24-object-alignment/).

But one lesson is clear: it is sometimes good to over-provision the heap for the application (makes GC life easier, for example), but at the same time this over-provisioning should be done with care, and *smaller heap may mean more free space available*.

JVM Anatomy Quark #24: Object Alignment

Questions

- Are there alignment constraints for Java objects?
- I have heard Java objects are 8-byte aligned, is that true?
- Can we fiddle with alignment to improve our compressed references story?

Theory

Many hardware implementations require the accesses to data to be *aligned*, that is make sure that all accesses of N byte width are done on addresses that are integer multiples of N. Even when this is not specifically required for the plain accesses to data, special operations (notably, atomic operations), usually have alignment constraints too.

For example, x86 is generally receptive to misaligned reads and writes, and misaligned CAS that spans two cache lines at once still works, but it tanks the throughput performance. Other architectures would just plainly refuse to do such atomic operation, yielding a `SIGBUS` or another hardware exception. x86 also does not guarantee access atomicity for values that span multiple cache lines, which is a possibility when access is misaligned. Java specification, on the other hand, requires access atomicity for most types, and definitely for all `volatile` accesses.

So, if we have the `long` field in Java object, and it takes 8 bytes in memory, we have to make sure it is aligned by 8 bytes for performance reasons. Or even for correctness reasons, if that field is `volatile`. In a simple approach,[27] two things need to happen for this to hold true: the field offset inside the object should be aligned by 8 bytes, **and** the object itself should be aligned by 8 bytes. That is indeed what we shall see if we peek into `java.lang.Long` instance:[28]

SHELL

```
$ java -jar jol-cli.jar internals java.lang.Long
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

java.lang.Long object internals:
 OFFSET  SIZE   TYPE DESCRIPTION               VALUE
      0     4        (object header)           01 00 00 00
      4     4        (object header)           00 00 00 00
      8     4        (object header)           ce 21 00 f8
     12     4        (alignment/padding gap)
     16     8   long Long.value                0
Instance size: 24 bytes
Space losses: 4 bytes internal + 0 bytes external = 4 bytes total
```

Here, the `value` field itself is at offset 16 (it is multiple of 8), and the object is aligned by 8.

Even if there are no fields that require special treatment, there are still object headers that also need to be accessed atomically. It is technically possible to align the majority of Java objects by 4 bytes, rather by 8 bytes, however the runtime work required to pull that off is quite immense (https://bugs.openjdk.java.net/browse/JDK-8025677).

So, in Hotspot, the minimum object alignment is 8 bytes. Can it be larger, though? Sure it can, there is the VM option for that: `-XX:ObjectAlignmentInBytes`. And it comes with two consequences, one negative and one positive.

Instance Sizes Get Large

Of course, once the alignment gets larger, it means that the average space wasted per-object would also increase. See, for example, the object alignment increased to 16 and 128 bytes:

```
$ java -XX:ObjectAlignmentInBytes=16 -jar jol-cli.jar internals java.lang.Long

java.lang.Long object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           01 00 00 00
      4     4        (object header)                           00 00 00 00
      8     4        (object header)                           c8 10 01 00
     12     4        (alignment/padding gap)
     16     8   long Long.value                                0
     24     8        (loss due to the next object alignment)
Instance size: 32 bytes
Space losses: 4 bytes internal + 8 bytes external = 12 bytes total
```

SHELL

```
$ java -XX:ObjectAlignmentInBytes=128 -jar jol-cli.jar internals java.lang.Long

java.lang.Long object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           01 00 00 00
      4     4        (object header)                           00 00 00 00
      8     4        (object header)                           a8 24 01 00
     12     4        (alignment/padding gap)
     16     8   long Long.value                                0
     24   104        (loss due to the next object alignment)
Instance size: 128 bytes
Space losses: 4 bytes internal + 104 bytes external = 108 bytes total
```

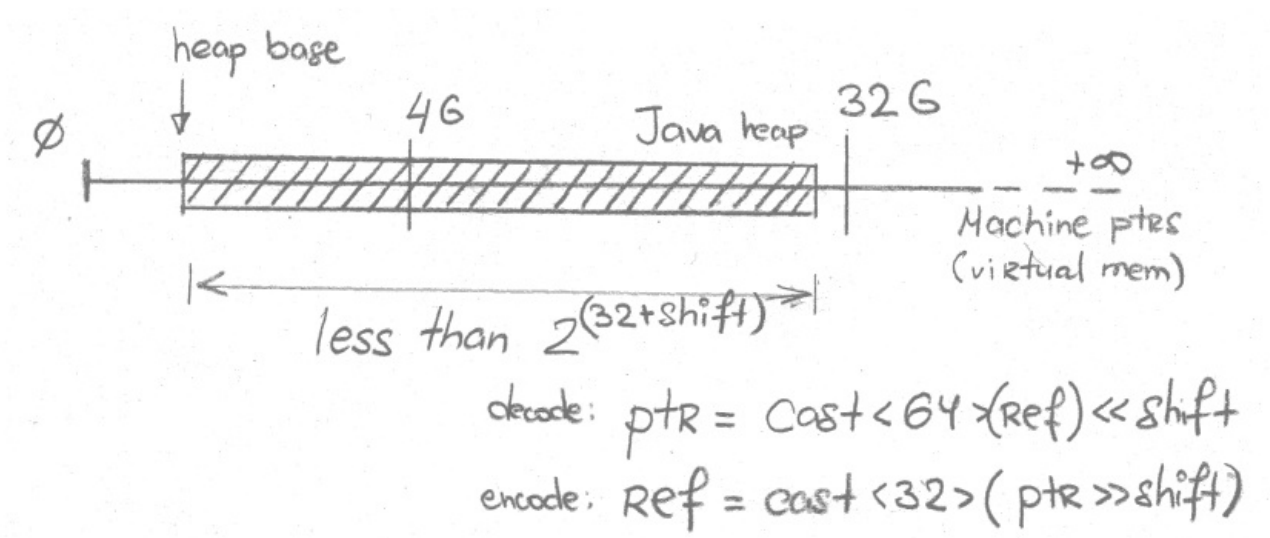
SHELL

Hell, 128 bytes per instance that only has 8 bytes of useful data seems excessive. Why would anyone do that?

Compressed References Threshold Get Shifted

(pun intended)

Remember this picture from the "Compressed References" quark (https://shipilev.net/jvm/anatomy-quarks/23-compressed-references/)?



It says that we can have compressed references enabled on heaps larger than 4 GB by shifting the reference by a few bits. The length of that shift depends on how many lower bits in reference are zero. That is, how objects are aligned! With 8-byte alignment by default, 3 lower bits are zero, we shift by 3, and we get $2^{(32+3)}$ bytes = 32 GB addressable with compressed references. And with 16-byte alignment, we have $2^{(32+4)}$ bytes = 64 GB heap *with compressed references*!

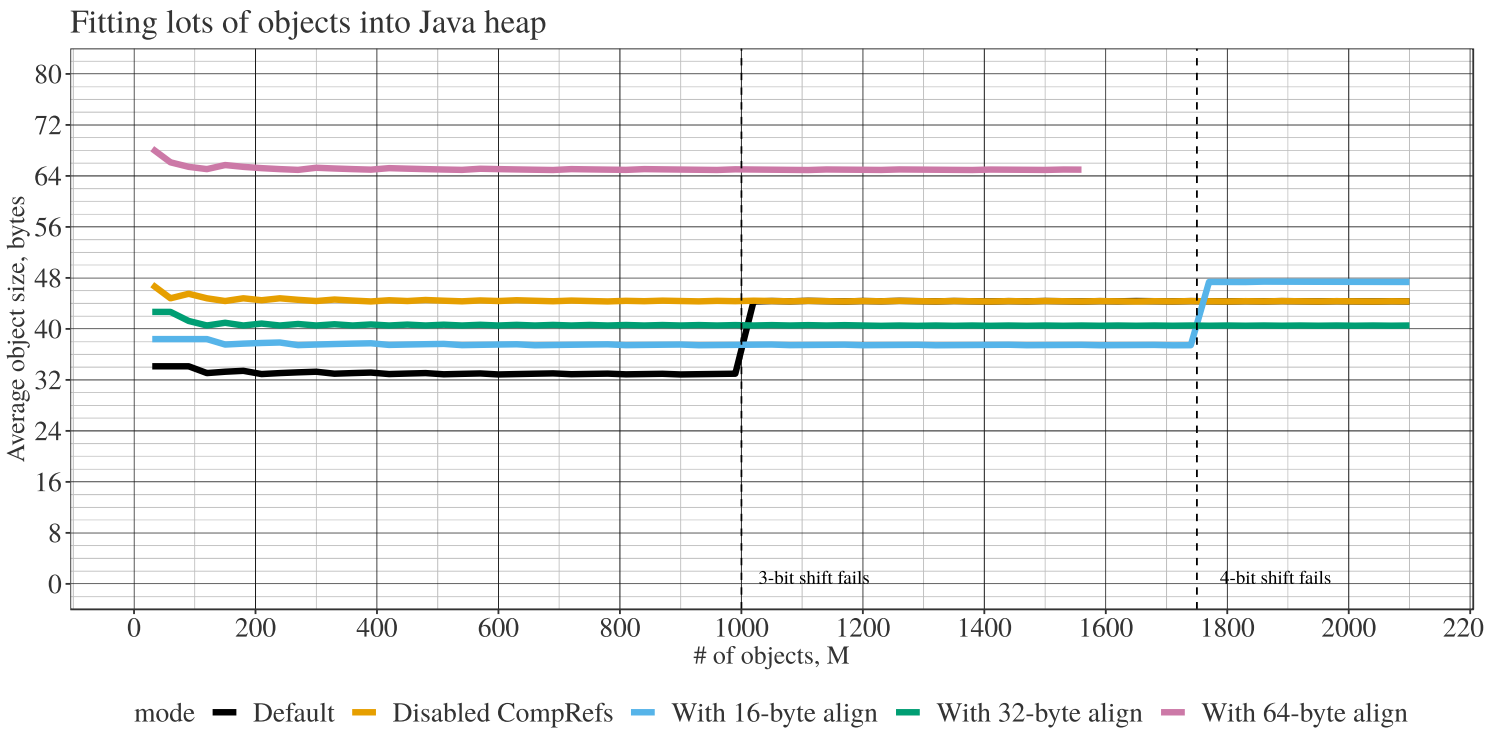
Experiment

So, object alignment blows up instance sizes, which increases heap occupancy, but allows compressed references on larger heaps, which decreases heap occupancy! Do these things cancel each other? Depends on the structure of the heap. We could use the same test we had before, but let's automate it a little.

Make the little test that tries to identify the minimum heap to accommodate the given number of objects, like this:

```java
import java.io.*;
import java.util.*;

public class CompressedOopsAllocate {

  static final int MIN_HEAP = 0 * 1024;
  static final int MAX_HEAP = 100 * 1024;
  static final int HEAP_INCREMENT = 128;

  static Object[] arr;

  public static void main(String... args) throws Exception {
    if (args.length >= 1) {
      int size = Integer.parseInt(args[0]);
      arr = new Object[size];
      IntStream.range(0, size).parallel().forEach(x -> arr[x] = new byte[(x % 20) + 1]);
      return;
    }

    String[] opts = new String[]{
      "",
      "-XX:-UseCompressedOops",
      "-XX:ObjectAlignmentInBytes=16",
      "-XX:ObjectAlignmentInBytes=32",
      "-XX:ObjectAlignmentInBytes=64",
    };

    int[] lastPasses = new int[opts.length];
    int[] passes = new int[opts.length];
    Arrays.fill(lastPasses, MIN_HEAP);

    for (int size = 0; size < 3000; size += 30) {
      for (int o = 0; o < opts.length; o++) {
        passes[o] = 0;
        for (int heap = lastPasses[o]; heap < MAX_HEAP; heap += HEAP_INCREMENT) {
          if (tryWith(size * 1000 * 1000, heap, opts[o])) {
            passes[o] = heap;
            lastPasses[o] = heap;
            break;
          }
        }
      }

      System.out.println(size + ", " + Arrays.toString(passes).replaceAll("[\\[\\]]",""));
    }
  }

  private static boolean tryWith(int size, int heap, String... opts) throws Exception {
    List<String> command = new ArrayList<>();
    command.add("java");
    command.add("-XX:+UnlockExperimentalVMOptions");
    command.add("-XX:+UseEpsilonGC");
    command.add("-XX:+UseTransparentHugePages"); // faster this way
    command.add("-XX:+AlwaysPreTouch");          // even faster this way
    command.add("-Xmx" + heap + "m");
    Arrays.stream(opts).filter(x -> !x.isEmpty()).forEach(command::add);
    command.add(CompressedOopsAllocate.class.getName());
    command.add(Integer.toString(size));

    Process p = new ProcessBuilder().command(command).start();
    return p.waitFor() == 0;
  }
}
```

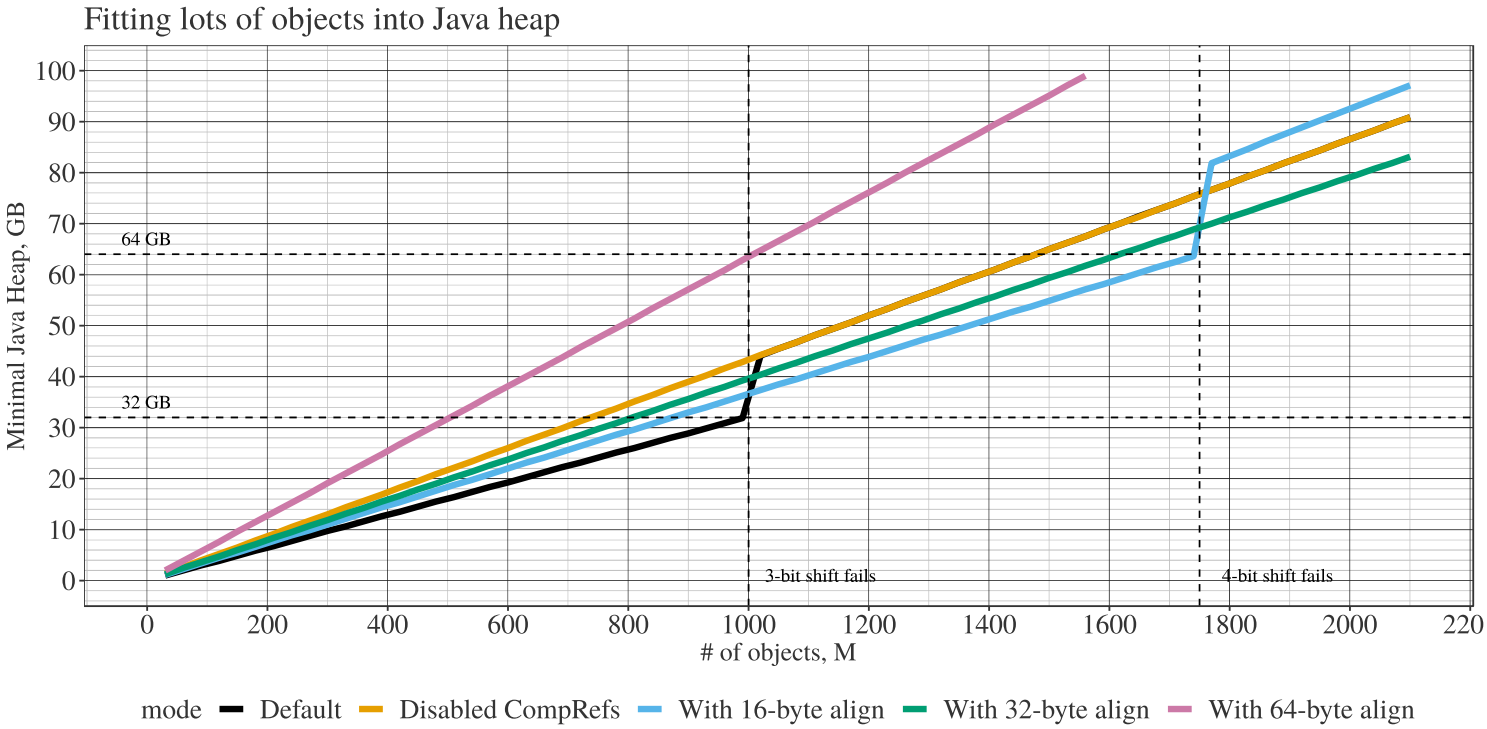
Running this test on large machine that can go up to 100+ GB heap, would yield predictable results. Let us start with average object sizes to set the narrative. Note these are average object sizes *in that particular test*, that allocates lots of small `byte[]` arrays. Here:



Not surprisingly, increasing alignment does inflate the average object sizes: 16-byte and 32-byte alignments have managed to increase the object size "just a little", while 64-byte alignment exploded the average considerably. Note that object alignment basically tells *the minimum* object size, and once that minimum goes up, average also goes up.

As we seen in "Compressed References" quark (https://shipilev.net/jvm/anatomy-quarks/23-compressed-references/), compressed references would normally fail around 32 GB. But notice that **higher alignments prolong this, and the higher the alignment, the longer it takes to fail**. For example, 16-byte alignment would have 4-bit shift in compressed references, and fail around 64 GB. 32-byte alignment would have 5-bit shift and fail around 128 GB.[29] In this particular test, on some object counts, the object size inflation due to higher alignment is balanced by lower footprint due to compressed references are active. Of course, when compressed references get finally disabled, the alignment costs catch up.

It can be more aptly seen at "minimal heap size" graph:



Here, we clearly see the 32 GB and 64 GB failure thresholds. Notice how 16-byte and 32-byte alignment took *less heap* in some configurations, piggybacking on more efficient reference encoding. That improvement is not universal: when 8-byte alignment is enough or when compressed references fail, higher alignments waste memory.

Conclusions

Object alignment is a funny thing to tinker with. While it inflates the object sizes considerably, it can also make the overall footprint lower, once compressed references come into picture. Sometimes it makes sense to bump the alignment a little bit, to reap the footprint benefits [sic!]. In many cases, however, this would degrade overall footprint. Careful study on the given application and given dataset is required to figure out if bumping alignment pays off or not. **It is a sharp tool, use it with care.**

JVM Anatomy Quark #25: Implicit Null Checks

Question

Java specification says that `NullPointerException` would be thrown when we access the `null` object fields. Does this mean the JVM has to always employ runtime checks for nullity?

Theory

In theory, (JIT) compiler can know that the object is not `null` and elide the runtime null check, for example when something is constant (https://shipilev.net/jvm/anatomy-quarks/15-just-in-time-constants/):

```java
static class Holder { int x; }
static final Holder H = new Holder();

int m() {
  return H.x; // H is known to be not null at JIT compilation time
}
```

If that does not work, for example when the nullity cannot be inferred automatically, compilers can also employ dataflow analysis to remove the successive null checks after first null check for the object was done. For example:

```java
int m(Holder h) {
  int x1 = h.x; // null-check here
  int x2 = h.x; // no need to null-check here again
  return x1 + x2;
}
```

Those optimizations are very useful, but quite boring, and they don't solve the need for null checks in all other cases.

Fortunately, there is even a smarter way to do this: **let the user code access the object without the explicit check**! Most of the time, nothing bad is going to happen, as most object accesses **do not ever see** the null object. But we still need to handle the corner case when the `null` access does happen. When it does, the JVM can intercept (http://hg.openjdk.java.net/jdk/jdk/file/b9d1ce20dd4b/src/hotspot/os_cpu/linux_x86/os_linux_x86.cpp#l486) the resulting SIGSEGV ("Signal: Segmentation Fault"), look at the return address for that signal, and figure out where that access was made in the generated code. Once it figures that bit out, it can then know where to dispatch the control to handle this case — in most cases, throwing `NullPointerException` or branching somewhere.

This mechanism is known in Hotspot under the name *"implicit null checks"* (http://hg.openjdk.java.net/jdk/jdk/file/b9d1ce20dd4b/src/hotspot/share/runtime/globals.hpp#l1029). It was recently added to LLVM under the similar name (https://llvm.org/docs/FaultMaps.html), to cater for the same use case.

Can we see how it works in practice?

Practice

Consider this cunningly simple JMH benchmark:

```java
import org.openjdk.jmh.annotations.*;

import java.util.concurrent.TimeUnit;

@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(value = 3, jvmArgsAppend = {"-XX:LoopUnrollLimit=1"})
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class ImplicitNP {

    @Param({"false", "true"})
    boolean blowup;

    volatile Holder h;

    int itCnt;

    @Setup
    public void setup() {
        h = null;
        if (blowup && ++itCnt == 3) { // blow it up on 3-rd iteration
            for (int c = 0; c < 10000; c++) {
                try {
                    test();
                } catch (NullPointerException npe) {
                    // swallow
                }
            }
            System.out.print("Boom! ");
        }
        h = new Holder();
    }

    @CompilerControl(CompilerControl.Mode.DONT_INLINE)
    @Benchmark
    public int test() {
        int sum = 0;
        for (int c = 0; c < 100; c++) {
            sum += h.x;
        }
        return sum;
    }

    static class Holder {
        int x;
    }
}
```

On the surface, this benchmark is simple: it performs the 100x integer addition.

Methodology-wise, this benchmark is cunning in several ways:

1. It is parametrized by `blowup` flag that would expose `null` object to `test()` method at the 3-rd iteration when `blowup = true`, and leave it alone otherwise.
2. It uses the looping in benchmark-unsafe manner. That is mitigated by asking Hotspot to not to unroll the loops with `LoopUnrollLimit`.
3. It accesses the same object over and over again. A smart optimizer would be able to hoist the load of `h` outside the loop, and then aggressively optimize. This is mitigated by declaring `h` as `volatile`: unless we are dealing with a God-like-smart optimizer, this is enough to break hoisting.
4. It uses compiler hints to break inlining for `test`. This is not, strictly speaking, needed for this benchmark, but it is safety measure. The reasoning goes as follows: the test relies on profiling information for `test`, and smarter compilers can use caller-callee profiles to split the profile between the version called from `setup()`, and from the benchmark loop itself.

Out of the curiosity, with recent 8u232,[30] it yields the following result:

```
Benchmark        (blowup)  Mode  Cnt   Score   Error  Units
ImplicitNP.test     false  avgt   15  40.417 ± 0.030  ns/op
ImplicitNP.test      true  avgt   15  63.187 ± 0.156  ns/op
```

Absolute numbers do not matter much here, the important bit is that one of the cases is much faster than the other one. The `blowup = false` case is significantly faster here. If we drill down into why, we would probably start with characterizing it with the help of `-prof perfnorm`, which can show the low-level machine counters for both tests:

```
Benchmark                        (blowup)  Mode  Cnt    Score    Error  Units

ImplicitNP.test                     false  avgt   15   40.484 ±  0.090  ns/op
ImplicitNP.test:L1-dcache-loads     false  avgt    3  206.606 ± 24.336   #/op
ImplicitNP.test:L1-dcache-stores    false  avgt    3    5.861 ±  0.426   #/op
ImplicitNP.test:branches            false  avgt    3  102.972 ± 13.679   #/op
ImplicitNP.test:cycles              false  avgt    3  141.252 ± 22.330   #/op
ImplicitNP.test:instructions        false  avgt    3  521.998 ± 87.292   #/op

ImplicitNP.test                      true  avgt   15   63.254 ±  0.047  ns/op
ImplicitNP.test:L1-dcache-loads      true  avgt    3  206.154 ± 15.231   #/op
ImplicitNP.test:L1-dcache-stores     true  avgt    3    4.971 ±  0.677   #/op
ImplicitNP.test:branches             true  avgt    3  199.993 ± 20.805   #/op ; +100 branches
ImplicitNP.test:cycles               true  avgt    3  221.388 ± 13.126   #/op ;  +80 cycles
ImplicitNP.test:instructions         true  avgt    3  714.439 ± 64.476   #/op ; +190 insns
```

So, we are hunting some excess branches. Note we had the loop with 100 iterations, so there must be the excess branch per iteration? Also, we have about 200 excess instructions, which makes sense as "branch" is really the `test` and `jcc` on x86_64.

Now that we have that hypothesis, let's see the actual hot code for both cases, with the help of `-prof perfasm`. The highly edited snippets are below.

First, `blowup = false` case:

```
          ...
  1.71%  ↗  0x...020: mov    0x10(%rsi),%r11d       ; get field "h"
  9.19%  │  0x...024: add    0xc(%r12,%r11,8),%eax  ; sum += h.x
         │                                          ; implicit exception:
         │                                          ; dispatches to 0x...03e
 59.60%  │  0x...029: inc    %r10d                  ; increment "c" and loop
  0.02%  │  0x...02c: cmp    $0x64,%r10d
         └  0x...030: jl     0x...d204020
  4.57%     0x...032: add    $0x10,%rsp
  3.16%     0x...036: pop    %rbp
  3.37%     0x...037: test   %eax,0x16a18fc3(%rip)
            0x...03d: retq
            0x...03e: mov    $0xfffffff6,%esi
            0x...043: callq  0x00007f8aed0453e0     ; <uncommon trap>
            ...
```

Here, we can see a very tight loop, and the instruction at `0x…024` combines the compressed reference (https://shipilev.net/jvm/anatomy-quarks/23-compressed-references) decoding of `h`, the access to `h.x`, and **the implicit null check**. We do not pay with any additional instructions to check `h` for nullity.[31]

The `implicit exception: dispatches to 0x…03e` line is the part of VM output that says VM knows SEGV exception coming from that instruction had actually failed null check. The JVM signal handler would then do its bidding and dispatch the control to `0x…03e`, which would then go on to throwing the exception.[32]

Of course, if `null`-s are frequent on that path, going via the signal handler every time is rather slow. For our current case, we could have said that throwing the exception would still be heavy, but it runs into two logistical problems. First, even though exceptions are sometimes slow (https://shipilev.net/blog/2014/exceptional-performance/), there is no reason to make them even slower if we can avoid it. Second, we would like to deal with user-written null-checks using the same machinery, and users would not like their simple `if (h == null) { … } else { … }` branches run dramatically worse depending on the nullity of `h`. Therefore, we would like to use implicit null-checks only when the frequency of actual `null`-s is very low.

Luckily, the JVM can compile the code **knowing the runtime profile**. That is, when JIT compiler decides whether to emit the implicit null check, it can look into profile and see if the object was ever `null`. Moreover, even if it does emit the implicit null check, it can **recompile** the code later when that optimistic assumption about `null` frequency is violated. `blowup = true` case specifically violates that assumption by feeding `null` to our code. As the result, the JVM recompiles the whole thing into: [33]

```asm
            ...
 11.36%  ↗  0x...bd1: mov    0x10(%rsi),%r11d       ; get field "h"
 12.81%  │  0x...bd5: test   %r11d,%r11d            ; EXPLICIT NULL CHECK
  0.02%  ┌  0x...bd8: je     0x...bf4
 17.23%  ||  0x...bda: add    0xc(%r12,%r11,8),%eax  ; sum += h.x
 25.07%  ||  0x...bdf: inc    %r10d                  ; increment "c" and loop
  8.70%  ||  0x...be2: cmp    $0x64,%r10d
  0.02%  |└  0x...be6: jl     0x...bd1
  3.31%  |   0x...be8: add    $0x10,%rsp
  2.49%  |   0x...bec: pop    %rbp
  2.72%  |   0x...bed: test   %eax,0x160e640d(%rip)
         |   0x...bf3: retq
         ↘   0x...bf4: movabs $0x7821044f8,%rsi      ; <preallocated NullPointerException>
             0x...bfe: mov    %r12d,0x10(%rsi)       ; WTF
             0x...c02: add    $0x10,%rsp
             0x...c06: pop    %rbp
             0x...c07: jmpq   0x00007f887d1053a0     ; throw_exception
             ...
```

Bam! There is the explicit null check in the generated code now! [34] Implicit null check turned itself into explicit one, without user intervention.

You can see that in flight when looking into the full benchmark log:

```shell
# JMH version: 1.22
# VM version: JDK 1.8.0_232, OpenJDK 64-Bit Server VM, 25.232-b09
# VM options: -XX:LoopUnrollLimit=1
# Warmup: 5 iterations, 1 s each
# Measurement: 5 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark: org.openjdk.ImplicitNP.test
# Parameters: (blowup = true)

# Run progress: 50.00% complete, ETA 00:00:30
# Fork: 1 of 3
Warmup Iteration   1: 40.900 ns/op
Warmup Iteration   2: 40.698 ns/op
Warmup Iteration   3: Boom! 63.157 ns/op  // <--- recompilation happened here
Warmup Iteration   4: 63.158 ns/op
Warmup Iteration   5: 63.130 ns/op
Iteration   1: 63.188 ns/op
Iteration   2: 63.208 ns/op
Iteration   3: 63.128 ns/op
Iteration   4: 63.137 ns/op
Iteration   5: 63.143 ns/op
```

See, everything was fine the first two iterations, then third iteration exposed `null` to the code, the JVM noticed that and recompiled.[35] This gives us more or less flat performance model for null checks.

Other Trivia: Shenandoah GC

Overall, this is quite a useful technique, and so it is used even outside handling the original Java accesses to the heap. For example, Shenandoah GC (https://wiki.openjdk.java.net/display/Shenandoah)'s load-reference-barrier (https://developers.redhat.com/blog/2019/06/27/shenandoah-gc-in-jdk-13-part-1-load-reference-barriers/) needs to check if the object is in collection set. If it is not, the barrier can shortcut, as the current object does not move.

In x86_64 code:

```asm
................. LRB fastpath.........................
     0x...067: testb  $0x1,0x20(%r15)
  ┌  0x...06c: jne    0x...086
..│.............. actual heap access ....................
  │↗ 0x...06e: movl   $0x2a,0xc(%r9)
  ||  ...
..||.............. LRB mid path ..........................
..||.............. checking in-cset ......................
  ↘| 0x...086: mov    %r9,%r10
   | 0x...089: shr    $0x17,%r10           ; %r10 is biased region idx
   | 0x...08d: movabs $0x7f60d00919f0,%r8  ; %r8 is biased cset bitmap
   | 0x...097: cmpb   $0x0,(%r8,%r10,1)    ; <--- implicit check for null here!
   └ 0x...09c: je     0x...06e
     ...
```

The "collection set" bit is the property of the region, so there is a global "cset bitmap" that tells which regions are in collection set. To figure out whether the *object* in in collection set, the code divides the object address by the region size, and then checks against the region bitmap. The caveat here is that heap does not necessarily start at zero address. So, that division does not give you the actual region index. Instead, it gives you the *biased* region index: something that has the constant offset, depending on the actual heap base. To compensate for it, we can access the cset bitmap *itself* at its *biased offset*!

This makes us hit the region bitmap for every legitimate object address, **except** `null`, which would access something outside the bitmap. But then we know which address `null` would hit, and so we can allocate and commit the zero page there (http://hg.openjdk.java.net/jdk/jdk/rev/24eb7720919c), then this check can **pretend** the answer for `null` is `0`, or "false". And it would do so without handling `null`-s with separate runtime checks, or involving any signal handling machinery.

Conclusion

Virtual memory provides some nifty tricks when dealing with memory accesses. *Implicit null checks* profitably exploit the fact that most null checks never actually fire, and let the virtual memory subsystem notify us in case they do. Managed runtimes with recompilation provide us with the way to exploit profile to make the correct guess about the shape of the check, or even dynamically reshape the code when the assumption about null-check frequency was violated. In the end, the whole thing becomes more or less invisible to the user, while providing substantial performance benefits.

JVM Anatomy Quark #26: Identity Hash Code

Questions

What happens when we call `Object.hashCode` without the user-provided hash code? How does `System.identityHashCode` work? Does it take the object address?

Theory

In Java, every object has `equals` and `hashCode`, even if users do not provide one. If user does not provide the override for `equals`, then `==` (identity) comparison is used. If user does not provide the override for `hashCode`, then `System.identityHashCode` is used to perform the hashcode computation.

The Javadoc (https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--) for `Object.hashCode` says:

> *The general contract of hashCode is:*
>
> - *Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.*
> - *If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.*
> - *It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.*
>
> *As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)*

Hash codes are supposed to have two properties: a) *good distribution*, meaning the hash codes for distinct objects are as distinct as practically possible; b) *idempotence*, meaning having the same hash code for the objects that have the same key object components. Note the latter implies that if object had not changed those key object components, its hash code should not change as well.

It is a frequent source of bugs to change the object in such a way that its `hashCode` changes after it was used. For example, adding the object to a `HashMap` as key, then changing its fields so that hashCode mutates as well would lead to surprising behaviors: the object might not be found in the map at all, because internal implementation would look in the "wrong" bucket. Likewise, it is a frequent source of performance anomalies to have badly distributed hash codes, for example returning a constant value.

For user-specified hash code, both properties are achieved by computing it over the set of user-selected fields. With enough variety of fields and field values, it would be well distributed, and by computing it over the unchanged (for example, final) fields we get idempotence. In this case, we don't need to store the hash code anywhere. Some hash code implementations may choose to cache it in another field, but that is not required.

For identity hash code, there is no guarantee there are fields to compute the hash code from, and even if we have some, then it is unknown how stable those fields actually are. Consider `java.lang.Object` that does not have fields: what's its hash code? Two allocated `Object`-s are pretty much the mirrors of each other: they have the same metadata, they have the same (that is, empty) contents. The only distinct thing about them is their allocated address, but even then there are two troubles. First, addresses have very low entropy, especially coming from a bump-ptr allocator like most Java GCs employ, so it is not well distributed. Second, GC moves the objects, so address is not idempotent.[36] Returning a constant value is a no-go from performance standpoint.

So, current implementations compute the identity hash code from the internal PRNG ("good distribution"), and store it for every object ("idempotence").

To achieve this, Hotspot JVM has a few different styles of identity hashcode generators and it stores the computed identity hashcode in the object header for stability. The choice of the identity hashcode generator has direct impact on the the performance of `hashCode` itself and the `hashCode` users, notably `java.util.HashMap`. The implementation choice to store the computed identity hashcode in the object header directly impacts the hashcode accuracy (how many bits we can store), and the complex interaction with other users of the object headers.

There is a place in Hotspot codebase where the hashcode is generated (https://github.com/openjdk/jdk/blob/4927ee426aedbeea0f4119bac0a342c6d3576762/src/hotspot/share/runtime/synchronizer.cpp#L760-L798):

```cpp
static inline intptr_t get_next_hash(Thread* current, oop obj) {
  ...
  if (hashCode == 0) {
    // Use os::random();
  } else if (hashCode == 1) {
    // Use address with some mangling
  } else if (hashCode == 2) {
    // Use constant 1
  } else if (hashCode == 3) {
    // Use global counter
  } else if (hashCode == 4) {
    // Use raw address
  } else {
    // Use thread-local PRNG
  }
  ...
}
```

This setting it accessible as `-XX:hashCode` VM option.

The generated hashcode would be installed into object header in `ObjectSynchronizer::FastHashCode` later (https://github.com/openjdk/jdk/blob/4927ee426aedbeea0f4119bac0a342c6d3576762/src/hotspot/share/runtime/synchronizer.cpp#L800-L907), and reused on next hash code requests.

Can we see how it works in practice?

Hashcode Storage

We can look into the identity hash code storage with JOL (https://github.com/openjdk/jol). In fact, there is a JOLSample_15_IdentityHashCode (https://github.com/openjdk/jol/blob/5d72f262b215a05cd66d71c06a0e38ac437490a0/jol-samples/src/main/java/org/openjdk/jol/samples/JOLSample_15_IdentityHashCode.java) sample that already captures what we want:

```shell
$ java -cp jol-samples/target/jol-samples.jar org.openjdk.jol.samples.JOLSample_15_IdentityHashCode
...

**** Fresh object
org.openjdk.jol.samples.JOLSample_15_IdentityHashCode$A object internals:
OFF  SZ  DESCRIPTION             VALUE
  0   8  (object header: mark)   0x0000000000000001 (non-biasable; age: 0)
  8   4  (object header: class)  0x00cc4000
 12   4  (object alignment gap)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

hashCode: 4e9ba398

**** After identityHashCode()
org.openjdk.jol.samples.JOLSample_15_IdentityHashCode$A object internals:
OFF  SZ  DESCRIPTION             VALUE
  0   8  (object header: mark)   0x0000004e9ba39801 (hash: 0x4e9ba398; age: 0)
  8   4  (object header: class)  0x00cc4000
 12   4  (object alignment gap)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

Here, the internal generator figured out the hashcode for this object is `4e9ba398`, and it recorded it as such in the object header. Every subsequent call for identity hashcode would now reuse this value.

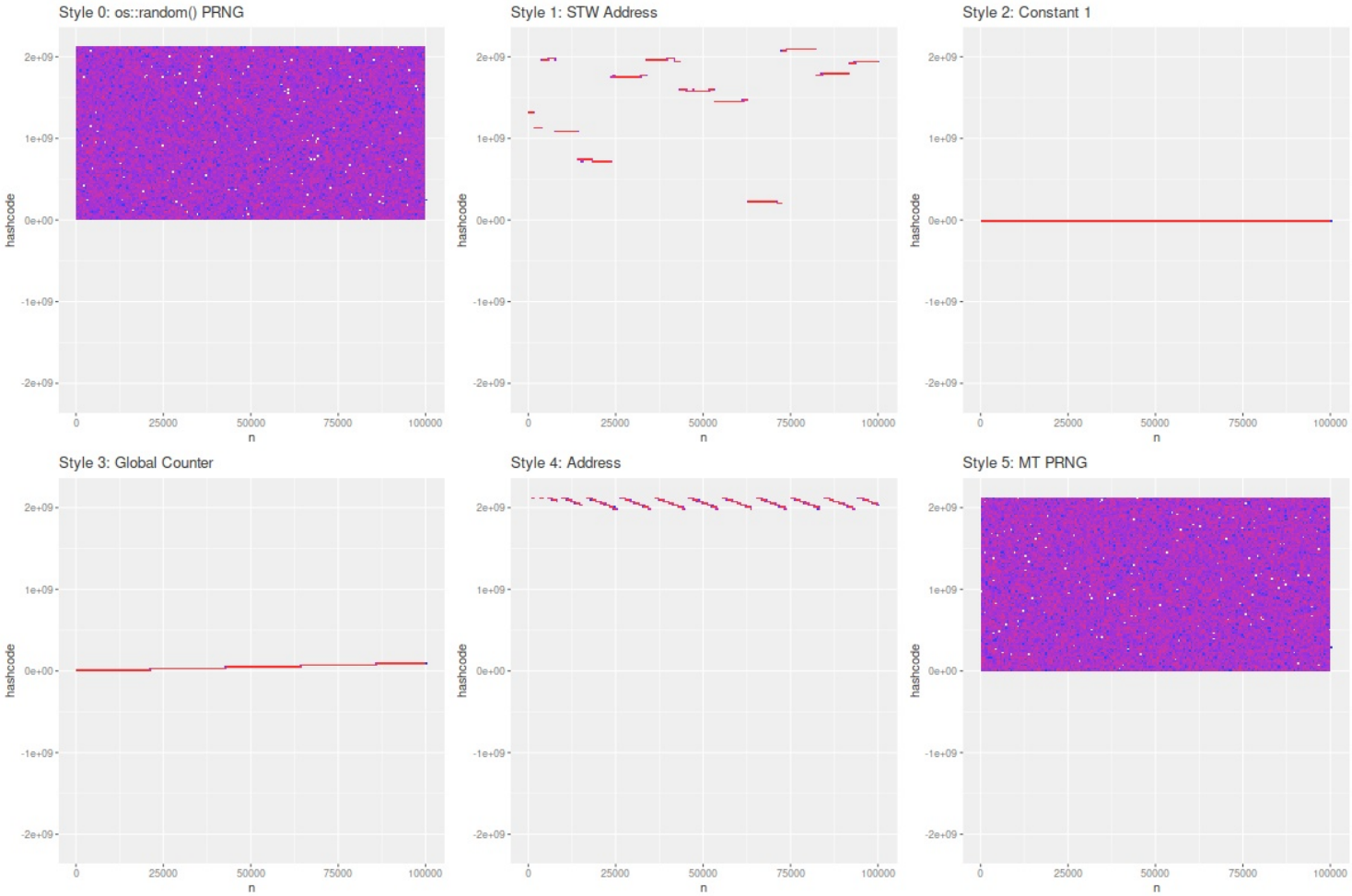
Hashcode Generators Randomness

In order to estimate the randomness of identity hash code generators, we can use the test like this:

```java
public class HashCodeValues {
  static long sink;
  public static void main(String... args) {
    for (int t = 0; t < 100000; t++) {
      for (int c = 0; c < 1000; c++) {
         sink = new Object().hashCode();
      }
      System.out.println(new Object().hashCode());
    }
  }
}
```

The goal for this test is to print the identity hash codes for consequtive objects. It comes with the demonstration problem: some generators are distributed in appalingly bad way, so on large graph scales they would be indistinguishable from each other. Therefore, the test skips printing the hashcode for the majority of intermediate objects, while still (awkwardly) making sure the hashcode is computed.

The heatmap of hash code values would be something like this:



Note a few things here:

1. Both PRNGs have the apparent value domain of nearly half of all possible hashcode values. Only the "upper" half is present in values, because only the first 31 bits (https://github.com/openjdk/jdk/blob/edff55607b9bc47bc1a5d9de7ad1a5d622be9736/src/hotspot/share/oops/markWord.hpp#L103) of identity hash code is stored in the header in 64-bit JVMs.[37]

2. The entropy of object addresses is very low. This is due to linear nature of (T)LAB allocations (https://shipilev.net/jvm/anatomy-quarks/4-tlab-allocation/): the temporaly-adjacent objects would have the very similar addresses. Indeed, this is why generating the hashcode from the object address is a bad idea!
3. Global counters are inconveniently distributed. Thir value domain for global counter is only the number of objects we have ever computed the hashcode for.
4. Not surprisingly, constant hashcodes exhibit apallingly bad distribution.

The frequently overlooked bit for both address-based and global counter hashcodes is that — while they can be more distinct than PRNGs (which additionally suffer from birthday paradoxes) — they are very well correlated bit-wise, which runs into the risk of getting a sub-hash collision once you select the non-lower-bit subruns from the hash code. Additionally, regular hashcodes like the global counter ones perform oddly when we deal with elements in a regular pattern, for example, retaining every *second* object in the hash table quickly leads to having the elements with only odd/even hashcodes, which underutilizes the hash tables doing e.g. `hashcode % size` bucket placement.

Hashcode Generators Performance

It might be fun to see what performance you can get from these generators. In a simple JMH benchmark like this, you have more or less predictable results:

JAVA

```
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@Threads(Threads.MAX)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@BenchmarkMode(Mode.AverageTime)
@State(Scope.Benchmark)
public class IdentityHashCode {
    Object o = new Object();

    @Benchmark
    public void cold(Blackhole bh) {
        Object lo = new Object();
        bh.consume(lo);
        bh.consume(lo.hashCode());
    }

    @Benchmark
    public void warm(Blackhole bh) {
        Object lo = o;
        bh.consume(lo); // for symmetry
        bh.consume(lo.hashCode());
    }
}
```

On a small ultrabook with latest JDK 17 EA, it would yield:

```
Benchmark                Mode  Cnt    Score   Error  Units

# Style 0: os::random() PRNG
IdentityHashCode.cold    avgt   15  400.703 ± 12.470  ns/op
IdentityHashCode.warm    avgt   15    5.051 ±  0.064  ns/op

# Style 1: STW Address
IdentityHashCode.cold    avgt   15   86.180 ±  1.854  ns/op
IdentityHashCode.warm    avgt   15    5.109 ±  0.074  ns/op

# Style 2: Constant 1
IdentityHashCode.cold    avgt   15   83.195 ±  2.034  ns/op
IdentityHashCode.warm    avgt   15    5.045 ±  0.060  ns/op

# Style 3: Global Counter
IdentityHashCode.cold    avgt   15  124.748 ±  0.946  ns/op
IdentityHashCode.warm    avgt   15    5.069 ±  0.079  ns/op

# Style 4: Address
IdentityHashCode.cold    avgt   15   86.232 ±  2.984  ns/op
IdentityHashCode.warm    avgt   15    5.066 ±  0.058  ns/op

# Style 5: MT PRNG
IdentityHashCode.cold    avgt   15   90.809 ±  0.792  ns/op
IdentityHashCode.warm    avgt   15    5.087 ±  0.077  ns/op
```

Note a few things:

1. The `warm` variants perform the same, regardless of the generator used. This makes sense, as that path only picks up already stored identity hash code.
2. The majority of the `cold` cost is going to VM for hash code calculation. Even the most basic generator that returns a constant 1 costs quite a lot.
3. Other generators snowball with their own effects. Notably, os::random() PRNG does the atomic updated to the PRNG state, and thus suffers from the major scalability problem.

Conclusion

The choice of identity hash code generator is heavily implementation specific. The generator should be both well-distributed and highly scalable. That is why modern Hotspot VMs default to `hashCode=5`, the multi-threaded PRNGs.[38]

There is no address computation involved in identity hashcode calculation at all. That is one of the reasons why the confusing mention of address computation was finally removed from the Javadoc.[39]

JVM Anatomy Quark #27: Compiler Blackholes

Questions

How does JMH avoids dead-code elimination for nano-benchmarks? Is there an implicit or explicit compiler support?

Theory

Optimizing compilers are good at optimizing simple stuff. For example, if there is a computation that is not observable by anyone, it can be deemed "dead code" and eliminated.

It is usually a good thing, until you run benchmarks. There, you want the computation, but you don't need the result. In essence, you observe the "resources" taken by the benchmark, but there is no easy way to argue this with a compiler.

So a benchmark like this:

```
JAVA
int x, y;

@Benchmark
public void test_dead() {
  int r = x + y;
}
```

...would be routinely compiled like this:[40]

```
ASM
  1.72%  ↗  ...370: movzbl 0x94(%r9),%r10d  ; load $isDone
  2.06%  │  ...378: mov    0x348(%r15),%r11 ; safepoint poll, part 1
 27.91%  │  ...37f: add    $0x1,%rbp        ; ops++;
 28.56%  │  ...383: test   %eax,(%r11)      ; safepoint poll, part 2
 33.43%  │  ...386: test   %r10d,%r10d      ; are we done? spin back if not.
         └  ...389: je     ...370
```

That is, only the benchmark infrastructure remains, with no actual `x + y` in sight. That code was dead, and it was eliminated.

Pure Java Blackholes

Since forever, JMH provides the way to avoid dead-code elimination by accepting the result (https://github.com/openjdk/jmh/blob/6696c744003fd3920c4848d450fb3ed1c83c2239/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_08_DeadCode.java) from the benchmark. Under the hood, it is done by feeding that result into a *Blackhole*, that can also be used directly (https://github.com/openjdk/jmh/blob/6696c744003fd3920c4848d450fb3ed1c83c2239/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_09_Blackholes.java) in some cases.

In short, the Blackhole has to achieve a single side effect on the incoming argument: pretend it is used. The Blackhole implementation notes (https://github.com/openjdk/jmh/blob/6696c744003fd3920c4848d450fb3ed1c83c2239/jmh-core/src/main/java/org/openjdk/jmh/infra/Blackhole.java#L153-L252) describe what the Blackhole implementor has to deal with when trying to cooperate with compiler. Implementing it efficiently is an fine exercise in near-JVM engineering.

Anyhow, all that mess is hidden from JMH users, so they can just do:

```
JAVA
int x, y;

@Benchmark
public int test_return() {
  return x + y;
}
```

If you look at the generated code, though, you would see that both the computation and the `Blackhole` code is there:

```
ASM
main loop:
  2.09% ↗ ...e32: mov    0x40(%rsp),%r10   ; load $this
  7.46% │ ...e37: mov    0x10(%r10),%edx   ; load $this.x
  0.64% │ ...e3b: add    0xc(%r10),%edx    ; add $this.y
  2.11% │ ...e3f: mov    0x38(%rsp),%rsi   ; call Blackhole.consume
  1.74% │ ...e44: data16 xchg %ax,%ax
  6.52% │ ...e47: callq  ...a80
 18.37% │ ...e4c: mov    (%rsp),%r10
  1.50% │ ...e50: movzbl 0x94(%r10),%r11d  ; load $isDone
  2.85% │ ...e58: mov    0x348(%r15),%r10  ; safepoint poll, part 1
  6.74% │ ...e5f: add    $0x1,%rbp         ; ops++
  0.62% │ ...e63: test   %eax,(%r10)       ; safepoint poll, part 2
  0.66% │ ...e66: test   %r11d,%r11d       ; are we done? spin back if not.
        ╰ ...e69: je     ...e32

Blackhole.consume:
  2.34%   ...040: mov    %eax,-0x14000(%rsp) ; too
  9.14%   ...047: push   %rbp                ; lazy
  0.64%   ...048: sub    $0x20,%rsp          ; to
  3.38%   ...04c: mov    %edx,%r11d          ; cross-reference
  6.66%   ...04f: xor    0xb0(%rsi),%r11d    ; this
  0.68%   ...056: mov    %edx,%r8d           ; with
  1.76%   ...059: xor    0xb8(%rsi),%r8d     ; the
  1.62%   ...060: cmp    %r8d,%r11d          ; actual
        ┌ ...063: je     ...078              ; Blackhole
  7.22% │ ...065: add    $0x20,%rsp          ; code
  0.35% │ ...069: pop    %rbp
  2.01% │ ...06a: cmp    0x340(%r15),%rsp
        │ ...071: ja     ...094
  8.53% │ ...077: retq
        ↘ ...078: mov    %rsi,%rbp
```

Not surprisingly, the `Blackhole` costs dominate such a tiny benchmark. With `-prof perfnorm`, we can see how bad it is:

```
SHELL
Benchmark                              Mode  Cnt   Score    Error  Units
XplusY.test_return                     avgt   25   3.288 ±  0.032  ns/op
XplusY.test_return:L1-dcache-loads     avgt    5  13.092 ±  0.487   #/op
XplusY.test_return:L1-dcache-stores    avgt    5   3.031 ±  0.076   #/op
XplusY.test_return:branches            avgt    5   5.031 ±  0.089   #/op
XplusY.test_return:cycles              avgt    5   8.781 ±  0.351   #/op
XplusY.test_return:instructions        avgt    5  27.162 ±  0.489   #/op
```

That is, our "payload" is only 2 instructions, yet the whole benchmark takes another 25 instructions on top of them! Yes, modern CPUs can execute that whole bunch of instructions in about 9 cycles here, but it is still too much work. To add insult to injury, the calling code and related stack management introduced *stores*.

The benchmark itself takes about 3.2 ns/op, which puts a lower limit on the effects we can reliably measure.

Compiler Blackholes

Luckily, we can ask a more direct cooperation from the compiler, with the use of *compiler blackholes*. Those are implemented in OpenJDK 17 with JDK-8259316 (https://bugs.openjdk.java.net/browse/JDK-8259316), with the plan (https://mail.openjdk.java.net/pipermail/jdk-dev/2021-March/005239.html) to backport it to 11u as well. Compiler blackholes are instructing the compilers to carry all arguments through the optimization phases, and then finally drop them when emitting the generated code. Then, as long as hardware itself does not provide surprises to us, we should be good. [41]

They are supposed to work transparently for JMH users, but since the whole thing is experimental, at this time JMH users are required to opt-in to compiler blackholes with `-Djmh.blackhole.mode=COMPILER` and then check the generated code for correctness.[42] Indeed, using compiler blackholes with our benchmark, we can see that the computation is still there, and there is no `Blackhole` call anymore!

```
SHELL
  8.95% ↗ ...c00: mov    0x10(%r11),%r10d  ; load $this.x
  0.36% │ ...c04: add    0xc(%r11),%r10d   ; add $this.y
        │                                  ; (AND COMPILER BLACKHOLE IT)
  0.94% │ ...c08: movzbl 0x94(%r14),%r8d   ; load $isDone
 26.76% │ ...c10: mov    0x348(%r15),%r10  ; safepoint poll, part 1
  8.42% │ ...c17: add    $0x1,%rbp         ; ops++
  0.43% │ ...c1b: test   %eax,(%r10)       ; safepoint poll, part 2
 46.96% │ ...c1e: test   %r8d,%r8d         ; are we done? spin back if not.
  0.02% └ ...c21: je     ...c00
```

You cannot even see the blackhole code anywhere, except in extended disassembly annotation, but its effect is there: the computation is preserved. `-prof perfnorm` is also happier:

```
SHELL
Benchmark                            Mode  Cnt   Score    Error  Units
XplusY.test_return                   avgt   25   0.963 ±  0.042  ns/op
XplusY.test_return:L1-dcache-loads   avgt    5   5.029 ±  0.170   #/op
XplusY.test_return:L1-dcache-stores  avgt    5   0.001 ±  0.002   #/op
XplusY.test_return:branches          avgt    5   1.006 ±  0.019   #/op
XplusY.test_return:cycles            avgt    5   2.569 ±  0.108   #/op
XplusY.test_return:instructions      avgt    5   8.043 ±  0.182   #/op
```

No stores anymore, there are only 6 additional instructions that carry the infrastructure. The whole benchmark is able to succeed in less than 3 cycles and less than 1 ns, and that involves 5 L1 accesses, 3 of which are infrastructural ones. [43]

This makes explicit `Blackhole` uses more convenient too, for example when doing loops:

```
JAVA
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(1)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class CPI_Floor {

    @Param({"1000000"})
    private int count;

    @Benchmark
    public void test(Blackhole bh) {
        for (int c = 0; c < count; c += 10000) {
            for (int k = 0; k < 10000; k++) {
                int v = k + k + k;
                bh.consume(v);
            }
        }
    }
}
```

On TR 3970X, this hits the CPI floor or ~0.16 clks/insn or IPC ceiling of ~6 insn/clk! In fact, it appears that the whole inner loop over "k" executes in exactly one cycle!

```
SHELL
Benchmark                    (count)  Mode        Score   Error         Units
CPI_Floor.test               1000000  avgt   273422.337 ± 12722.427     ns/op
CPI_Floor.test:CPI           1000000  avgt        0.169              clks/insn
CPI_Floor.test:IPC           1000000  avgt        5.907              insns/clk
CPI_Floor.test:branches      1000000  avgt  1003135.103                   #/op
CPI_Floor.test:cycles        1000000  avgt  1022821.963                   #/op
CPI_Floor.test:instructions  1000000  avgt  6042142.469                   #/op
```

Conclusion

The compiler blackholes are great for low-level performance investigations. Try to use them, check they do what you want, show the success and failure stories, and hope all this would culminate with a new default Blackhole modes in JMH.

JVM Anatomy Quark #28: Frequency-Based Code Layout

Questions

Do JVMs perform profile-guided optimizations? Do JVMs profile branch frequency? Do JVMs use branch frequency information for anything?

Theory

Among other things that Java VMs bring to the ecosystem, one of the good features is *always on* profile-guided optimization. To understand how that works, consider that in a modern JVM, Java code is executed with several engines: the interpreter, the baseline compiler (in Hotspot, C1), the optimized compiler (in Hotspot, C2). As Java code becomes more and more hot in Hotspot, it is moving gradually from interpreter to C1, then to C2. That means the prior stages can perform the online profiling of the code, and present that data to the higher stages in compilation.

A notable piece of the useful profiling data is the branch frequency counters. The branches are relatively non-frequent in the code, and the counter itself usually needs to just record the number of times the branch was taken/skipped. This means the profiling data is not overwhelming to record.

Then, a smarter compiler can use the branch frequency data to do many things, most notably, lay out the generated code so that most often taken path is straight-forward. Yes, CPU branch predictors can alleviate some of the pain without frequency-based layouts, but straight-forward code is still better.

Can we see this in practice?

Practice

Consider this JMH benchmark:[44]

JAVA

```
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(1)
@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class BranchFrequency {
    @Benchmark
    public void mostlyFalse() {
        for (int c = 0; c < 9; c++) {
            doCall(false);
        }
        doCall(true);
    }

    @Benchmark
    public void mostlyTrue() {
        for (int c = 0; c < 9; c++) {
            doCall(true);
        }
        doCall(false);
    }

    @CompilerControl(CompilerControl.Mode.DONT_INLINE)
    public int doCall(boolean condition) {
        if (condition) {
            return 1;
        } else {
            return 2;
        }
    }
}
```

In either test, we take either one or another branch 90% of the time.

C1 compiler does not do frequency-based code layout, so we can expect it perform a bit differently. Indeed, it does! Look at `-prof perfnorm` data with `-XX:TieredStopAtLevel=1` on my i5-4210U:

```
Benchmark                                    Mode  Cnt    Score  Error   Units

BranchFrequency.mostlyFalse                  avgt   15   30.510 ±  0.212 ns/op
BranchFrequency.mostlyFalse:L1-dcache-loads  avgt    3   74.572 ± 13.337  #/op
BranchFrequency.mostlyFalse:L1-dcache-stores avgt    3   40.366 ±  1.994  #/op
BranchFrequency.mostlyFalse:branch-misses    avgt    3    0.004 ±  0.042  #/op
BranchFrequency.mostlyFalse:branches         avgt    3   52.509 ±  7.670  #/op
BranchFrequency.mostlyFalse:cycles           avgt    3   82.643 ± 14.422  #/op
BranchFrequency.mostlyFalse:instructions     avgt    3  240.178 ± 32.612  #/op

BranchFrequency.mostlyTrue                   avgt   15   27.426 ±  0.066 ns/op
BranchFrequency.mostlyTrue:L1-dcache-loads   avgt    3   74.701 ±  6.478  #/op
BranchFrequency.mostlyTrue:L1-dcache-stores  avgt    3   40.105 ±  3.923  #/op
BranchFrequency.mostlyTrue:branch-misses     avgt    3    0.001 ±  0.002  #/op
BranchFrequency.mostlyTrue:branches          avgt    3   52.411 ±  4.591  #/op
BranchFrequency.mostlyTrue:cycles            avgt    3   73.837 ±  7.356  #/op
BranchFrequency.mostlyTrue:instructions      avgt    3  239.375 ± 27.798  #/op
```

We are running roughly the same amount of instructions, L1 accesses, branches, etc. Yet the `mostlyFalse` case is slower for about 3 ns. If we look into disasembly, then sure enough the code layout is static for `doCall` method, with `true` branch laid out first:

```
# C1, mostlyTrue
  2.57%   ...44c: cmp    $0x0,%edx          ; test $condition
        ╭ ...44f: je     ...46d
  2.71% │ ...455: mov    $0x1,%eax          ; if true, return 0x1
  5.40% │ ...45a: add    $0x30,%rsp
  0.84% │ ...45e: pop    %rbp
  1.73% │ ...45f: cmp    0x340(%r15),%rsp
        │ ...466: ja     ...485
  8.85% │ ...46c: retq
  0.31% ↘ ...46d: mov    $0x2,%eax          ; if false, return 0x2
  0.29%   ...472: add    $0x30,%rsp
  0.72%   ...476: pop    %rbp
  0.06%   ...477: cmp    0x340(%r15),%rsp
          ...47e: ja     ...49b
  0.45%   ...484: retq

# C1, mostlyFalse
  2.76%   ...74c: cmp    $0x0,%edx          ; test $condition
        ╭ ...74f: je     ...76d
  0.22% │ ...755: mov    $0x1,%eax          ; if true, return 0x1
  0.06% │ ...75a: add    $0x30,%rsp
  0.96% │ ...75e: pop    %rbp
  0.06% │ ...75f: cmp    0x340(%r15),%rsp
        │ ...766: ja     ...785
  0.20% │ ...76c: retq
  2.46% ↘ ...76d: mov    $0x2,%eax          ; if false, return 0x2
  7.01%   ...772: add    $0x30,%rsp
  0.43%   ...776: pop    %rbp
  0.98%   ...777: cmp    0x340(%r15),%rsp
          ...77e: ja     ...79b
  8.86%   ...784: retq
```

So in `mostlyTrue` case, we fall-through the branch and exit, while on `mostlyFalse` case we have to take a jump and execute from another place.

If we run with C2 (either default mode, or specifically `-XX:-TieredCompilation`), which does frequency-based layouts, then we would see the performance and counters are roughly the same in both cases:

```
Benchmark                                    Mode  Cnt    Score    Error Units

BranchFrequency.mostlyFalse                  avgt   15   24.840 ±  0.027 ns/op
BranchFrequency.mostlyFalse:L1-dcache-loads  avgt    3   61.040 ±  2.702  #/op
BranchFrequency.mostlyFalse:L1-dcache-stores avgt    3   38.022 ±  0.276  #/op
BranchFrequency.mostlyFalse:branch-misses    avgt    3    0.002 ±  0.036  #/op
BranchFrequency.mostlyFalse:branches         avgt    3   52.012 ±  4.265  #/op
BranchFrequency.mostlyFalse:cycles           avgt    3   66.616 ±  1.398  #/op
BranchFrequency.mostlyFalse:instructions     avgt    3  212.290 ± 13.588  #/op

BranchFrequency.mostlyTrue                   avgt   15   24.829 ±  0.043 ns/op
BranchFrequency.mostlyTrue:L1-dcache-loads   avgt    3   61.127 ±  4.135  #/op
BranchFrequency.mostlyTrue:L1-dcache-stores  avgt    3   38.072 ±  3.190  #/op
BranchFrequency.mostlyTrue:branch-misses     avgt    3    0.002 ±  0.027  #/op
BranchFrequency.mostlyTrue:branches          avgt    3   52.153 ±  4.914  #/op
BranchFrequency.mostlyTrue:cycles            avgt    3   66.664 ±  3.982  #/op
BranchFrequency.mostlyTrue:instructions      avgt    3  212.572 ± 10.962  #/op
```

The disassembly would reveal an interesting fact: in either case, the most frequent branch is laid out first!

```
# C2, mostlyTrue
  1.49%    ...cac: test   %edx,%edx         ; check $condition
        ┌  ...cae: je     ...cc8
  1.05% │  ...cb0: mov    $0x1,%eax        ; if true, return 0x1
  1.45% │↗ ...cb5: add    $0x10,%rsp
 11.09% ││ ...cb9: pop    %rbp
 12.00% ││ ...cba: cmp    0x340(%r15),%rsp
        ││ ...cc1: ja     ...ccf
  1.55% ││ ...cc7: retq
  0.04% ↘│ ...cc8: mov    $0x2,%eax        ; if false, return 0x2
         └ ...ccd: jmp    ...cb5

# C2, mostlyFalse
  4.85%    ...32c: test   %edx,%edx         ; check $condition
        ┌  ...32e: jne    ...348
  1.10% │  ...330: mov    $0x2,%eax        ; if false, return 0x2
  1.42% │↗ ...335: add    $0x10,%rsp
  8.95% ││ ...339: pop    %rbp
 11.77% ││ ...33a: cmp    0x340(%r15),%rsp
  0.02% ││ ...341: ja     ...34f
  2.01% ││ ...347: retq
  0.08% ↘│ ...348: mov    $0x1,%eax        ; if true, return 0x1
  0.12%  └ ...34d: jmp    ...335
```

Here, the compiler capitalized handsomely on available branch frequency data to get the uniform result.

Conclusion

Always-on profiling in multi-tiered compilation schemes allows doing interesting profile-guided optimizations transparently to users.

JVM Anatomy Quark #29: Uncommon Traps

Questions

What is the finest unit for JIT compilation? If JIT decides to compile the method, does it compile everything in it? Should I warm up the methods using the real data? What tricks do JIT compilers have to optimize their compilation time?

Theory

It is a common wisdom that JIT compilers work on *methods*: once a method is deemed hot, the runtime system asks JIT compiler to produce an optimized version of it. It follows, naively, that JIT compiles the entirety of the method and hands it over to the runtime system.

But the fact is, the runtime system that allows speculative compilation/deoptimization allows JIT to compile methods with the sets of assumptions about its behavior. We have seen it before in Implicit Null Checks (https://shipilev.net/jvm/anatomy-quarks/25-implicit-null-checks/). This time, we would look at a more general thing about the cold code.

Consider this method that is effectively called *only with* `flag = true`:

```
JAVA
void m(boolean flag) {
  if (flag) {
     // do stuff A
  } else {
     // do stuff B
  }
}
```

Even if `flag` is not known from the analysis, the smart JIT compiler can use the branch profiling (https://shipilev.net/jvm/anatomy-quarks/28-frequency-based-code-layout/) to figure out that "B" branch is never taken, and compile it to:

```
JAVA
void m() {
  if (condition) {
     // do stuff A
  } else {
     // Assume this branch is never taken.
     <trap to runtime system: uncommon branch is taken>
  }
}
```

Thus, never actually compiling the *actual* code in branch B. This saves compilation time, usually improves code density, by avoiding dealing with code that would never be needed.

Note this is different from the code layout based on branch frequency. In this case, when one of the branches frequencies is exactly zero, we can skip compiling its body completely. If and only if the branch is taken, the generated code *traps* to runtime system saying that the compilation pre-condition was violated, and JIT would regenerate the method body in the new conditions, this time compiling now-not-uncommon branch.

Can we see this in practice?

Test

Consider this JMH benchmark:

```java
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(1)
@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class ColdCodeBench {

    @Param({"onlyA", "onlyB", "swap"})
    String test;

    boolean condition;

    @Setup(Level.Iteration)
    public void setup() {
        switch (test) {
            case "onlyA":
                condition = true;
                break;
            case "onlyB":
                condition = false;
                break;
            case "swap":
                condition = !condition;
                break;
        }
    }

    int v = 1;
    int a = 1;
    int b = 1;

    @Benchmark
    @CompilerControl(CompilerControl.Mode.DONT_INLINE)
    public void test() {
        if (condition) {
            v *= a;
        } else {
            v *= b;
        }
    }
}
```

In this test, we either taking the branch A only, or taking the branch B only, or we flip-flop between them on every iteration.

The point of this test is to demonstrate generated code in a simple manner. The performance for all versions would be roughly the same *in this trivial test*. In reality, the cold branches can take a lot of code, especially after inlining, and the performance impact on both compilation times and generated code density would be substantial.

Not surprisingly and in line with "Frequency-based Code Layout" (https://shipilev.net/jvm/anatomy-quarks/28-frequency-based-code-layout/), we can see that both "onlyA" and "onlyB" tests lay out the first branch right away. But then, a curious thing happens: there is no second branch code at all! Instead, there is a call to so called "uncommon trap"! That one is the notification to runtime that we have failed the compilation condition, and this "uncommon" branch is now taken.

```
# "onlyA"
  9.54%   ...3cc: movzbl 0x18(%rsi),%r10d  ; load and test $condition
  0.21%   ...3d1: test   %r10d,%r10d
        ┌ ...3d4: je     ...3f6
        │                                  ; if true, then...
  0.90% │ ...3d6: mov    0x10(%rsi),%r10d  ; ...load $a...
  7.81% │ ...3da: imul   0xc(%rsi),%r10d   ; ...multiply by $v...
 17.33% │ ...3df: mov    %r10d,0xc(%rsi)   ; ...store to $v...
  8.16% │ ...3e3: add    $0x20,%rsp        ; ...and return.
  0.60% │ ...3e7: pop    %rbp
  0.18% │ ...3e8: cmp    0x340(%r15),%rsp
  0.02% │ ...3ef: ja     ...408
 10.51% │ ...3f5: retq
        │                                  ; if false, then...
        ↘ ...3f6: mov    %rsi,%rbp
          ...3f9: mov    %r10d,(%rsp)
          ...3fd: mov    $0xffffff45,%esi
          ...402: nop
          ...403: callq  <runtime>         ; - (reexecute) o.o.CCB::test@4 (line 73)
                                           ;   {runtime_call UncommonTrapBlob}

# "onlyB"
 10.21%   ...acc: movzbl 0x18(%rsi),%r10d  ; load and test $condition
  0.25%   ...ad1: test   %r10d,%r10d
        ┌ ...ad4: jne    ...af6
        │                                  ; if false, then...
  0.29% │ ...ad6: mov    0x14(%rsi),%r10d  ; ...load $b...
  8.78% │ ...ada: imul   0xc(%rsi),%r10d   ; ...multiply by $v...
 18.87% │ ...adf: mov    %r10d,0xc(%rsi)   ; ...store $v...
  9.74% │ ...ae3: add    $0x20,%rsp        ; ...and return.
  0.24% │ ...ae7: pop    %rbp
  0.27% │ ...ae8: cmp    0x340(%r15),%rsp
        │ ...aef: ja     ...b08
  9.76% │ ...af5: retq
        │                                  ; if true, then...
        ↘ ...af6: mov    %rsi,%rbp
          ...af9: mov    %r10d,(%rsp)
          ...afd: mov    $0xffffff45,%esi
          ...b02: nop
          ...b03: callq  <runtime>         ; - (reexecute) o.o.CCB::test@4 (line 73)
                                           ;   {runtime_call UncommonTrapBlob}
```

When that "cold" branch is finally taken, then JVM would recompile the method. It would be visible in `-XX:+PrintCompilation` log like this:

```
# Warmup Iteration   1: ...

// Profiled version is compiled with C1 (+MDO)
    351  476       3       org.openjdk.ColdCodeBench::test (37 bytes)

// C2 version is installed
    352  477       4       org.openjdk.ColdCodeBench::test (37 bytes)

// Profiled version is declared dead
    352  476       3       org.openjdk.ColdCodeBench::test (37 bytes)   made not entrant

# Warmup Iteration   2: ...

// Deopt! C2 version is declared dead
   1361  477       4       org.openjdk.ColdCodeBench::test (37 bytes)   made not entrant

// Re-profiling version is compiled with C1 (+counters)
   1363  498       2       org.openjdk.ColdCodeBench::test (37 bytes)

// New C2 version is installed
   1364  499       4       org.openjdk.ColdCodeBench::test (37 bytes)

// Re-profiling version is declared dead
   1364  498       2       org.openjdk.ColdCodeBench::test (37 bytes)   made not entrant
```

The final result is clearly visible in "swap" case. There, both branches are compiled:

```
 4.25%    ...f2c: mov    0xc(%rsi),%r11d  ; load $v
 6.23%    ...f30: movzbl 0x18(%rsi),%r10d ; load and test $condition
 0.04%    ...f35: test   %r10d,%r10d
       ┌  ...f38: je     ...f45
       │                                  ; if false, then
 0.02% │  ...f3a: imul   0x10(%rsi),%r11d ; ...multiply by $a...
13.33% │  ...f3f: mov    %r11d,0xc(%rsi)  ; ...store $v
 3.82% │┌ ...f43: jmp    ...f4e
       ||                                 ; if true, then
 0.02% ↘| ...f45: imul   0x14(%rsi),%r11d ; ...multiply by $b...
18.70%  | ...f4a: mov    %r11d,0xc(%rsi)  ; ...store $v
 6.12%  ↘ ...f4e: add    $0x10,%rsp
          ...f52: pop    %rbp
 0.08%    ...f53: cmp    0x340(%r15),%rsp
          ...f5a: ja     ...f61
10.81%    ...f60: retq
```

SHELL

Conclusion

Advanced JIT compilers can compile only the actually active parts of the method. This simplifies generated code and JIT compiler overheads. On the other hand, this complicates the warmup: to avoid sudden recompilation, you need to warm up with the similar profile as you would run later, so that all paths are compiled.

JVM Anatomy Quark #30: Conditional Moves

Questions

Are there any interesting tricks that can be done when we have branch frequency data? What is a conditional move?

Theory

If you need to choose between two results coming from a branch, there are two distinct things that you can do at ISA level.

First, you can do a branch:

```ASM
    # %r = (%rCond == 1) ? $v1 : $v2
    cmp %rCond, $1
    jne A
    mov %r, $v1
    jmp E
A:  mov %r, $v2
E:
```

Second, you can perform a *predicated instruction* (https://en.wikipedia.org/wiki/Predication_(computer_architecture)) that is dependent on the result of the comparison. In x86, this takes the form of a *conditional move* (CMOV) that performs an action when a selected condition holds:

```ASM
# %r = (%rCond == 1) ? $v1 : $v2
mov %r, $v1       ; put $v1 to %r
cmp %rCond, ...
cmovne %r, $v2    ; put $v2 to %r if condition is false
```

The upside for doing a conditional move is that it sometimes generates more compact code, like in this example, and it does not suffer from possible branch misprediction penalty. The disadvantage is that it requires computing both sides at before choosing which one would be returned, which can spend excess CPU cycles, increase register pressure, etc. In branch case, we can choose not to compute stuff after checking the condition. A well-predicted branch would then outperform the conditional move.

Therefore, the choice whether to do or not to do a conditional move highly depends on its cost prediction. This is where branch profiling helps us: it can say which branches are probably not perfectly predicted, and thus amenable for CMOV replacement. Of course, the actual cost model (https://github.com/openjdk/jdk/blob/master/src/hotspot/share/opto/loopopts.cpp#L569-L764) also includes the types of the arguments we are dealing with, the relative depth of both computation branches, etc.

Can we see how this behaves in practice?

Practice 1

Consider this JMH benchmark:

JAVA

```java
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(1)
@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class BranchFrequency {
    @Benchmark
    public void fair() {
        doCall(true);
        doCall(false);
    }

    @CompilerControl(CompilerControl.Mode.DONT_INLINE)
    public int doCall(boolean condition) {
        if (condition) {
            return 1;
        } else {
            return 2;
        }
    }
}
```

We flip-flop between the branches on every call, which means its runtime profile is roughly 50%-50% between them. If we limit the conditional move replacement by supplying `-XX:ConditionalMoveLimit=0`, then we can clearly see the replacement happenning.

SHELL

```
# doCall, out of box variant
  4.36%  ...4ac: mov    $0x1,%r11d         ; move $1 -> %r11
  3.24%  ...4b2: mov    $0x2,%eax          ; move $2 -> %res
  8.46%  ...4b7: test   %edx,%edx          ; test boolean
  0.02%  ...4b9: cmovne %r11d,%eax         ; if false, move %r11 -> %res
  7.88%  ...4bd: add    $0x10,%rsp         ; exit the method
  8.12%  ...4c1: pop    %rbp
 18.60%  ...4c2: cmp    0x340(%r15),%rsp
         ...4c9: ja     ...4d0
  0.14%  ...4cf: retq

# doCall, CMOV conversion inhibited
  6.48%    ...cac: test   %edx,%edx         ; test boolean
         ┌  ...cae: je     ...cc8
         │                                  ; if true...
         │  ...cb0: mov    $0x1,%eax         ; move $1 -> %res
  7.41% │↗ ...cb5: add    $0x10,%rsp        ; exit the method
  0.02% ││ ...cb9: pop    %rbp
 27.43% ││ ...cba: cmp    0x340(%r15),%rsp
        ││ ...cc1: ja     ...ccf
  3.28% ││ ...cc7: retq
        ||                                  ; if false...
  7.04% ↘| ...cc8: mov    $0x2,%eax         ; move $2 -> %res
  0.02%  └ ...ccd: jmp    ...cb5            ; jump back
```

The CMOV version performs a little better in this example:

```
Benchmark                                Mode  Cnt   Score    Error  Units

# Branches
BranchFrequency.fair                     avgt   25   5.422 ±  0.026  ns/op
BranchFrequency.fair:L1-dcache-loads     avgt    5  12.078 ±  0.226   #/op
BranchFrequency.fair:L1-dcache-stores    avgt    5   5.037 ±  0.120   #/op
BranchFrequency.fair:branch-misses       avgt    5   0.001 ±  0.003   #/op
BranchFrequency.fair:branches            avgt    5  10.037 ±  0.216   #/op
BranchFrequency.fair:cycles              avgt    5  14.659 ±  0.285   #/op
BranchFrequency.fair:instructions        avgt    5  35.184 ±  0.559   #/op

# CMOVs
BranchFrequency.fair                     avgt   25   4.799 ±  0.094  ns/op
BranchFrequency.fair:L1-dcache-loads     avgt    5  12.014 ±  0.329   #/op
BranchFrequency.fair:L1-dcache-stores    avgt    5   5.005 ±  0.167   #/op
BranchFrequency.fair:branch-misses       avgt    5  ≈ 10⁻⁴            #/op
BranchFrequency.fair:branches            avgt    5   7.054 ±  0.118   #/op
BranchFrequency.fair:cycles              avgt    5  12.964 ±  1.451   #/op
BranchFrequency.fair:instructions        avgt    5  36.285 ±  0.713   #/op
```

SHELL

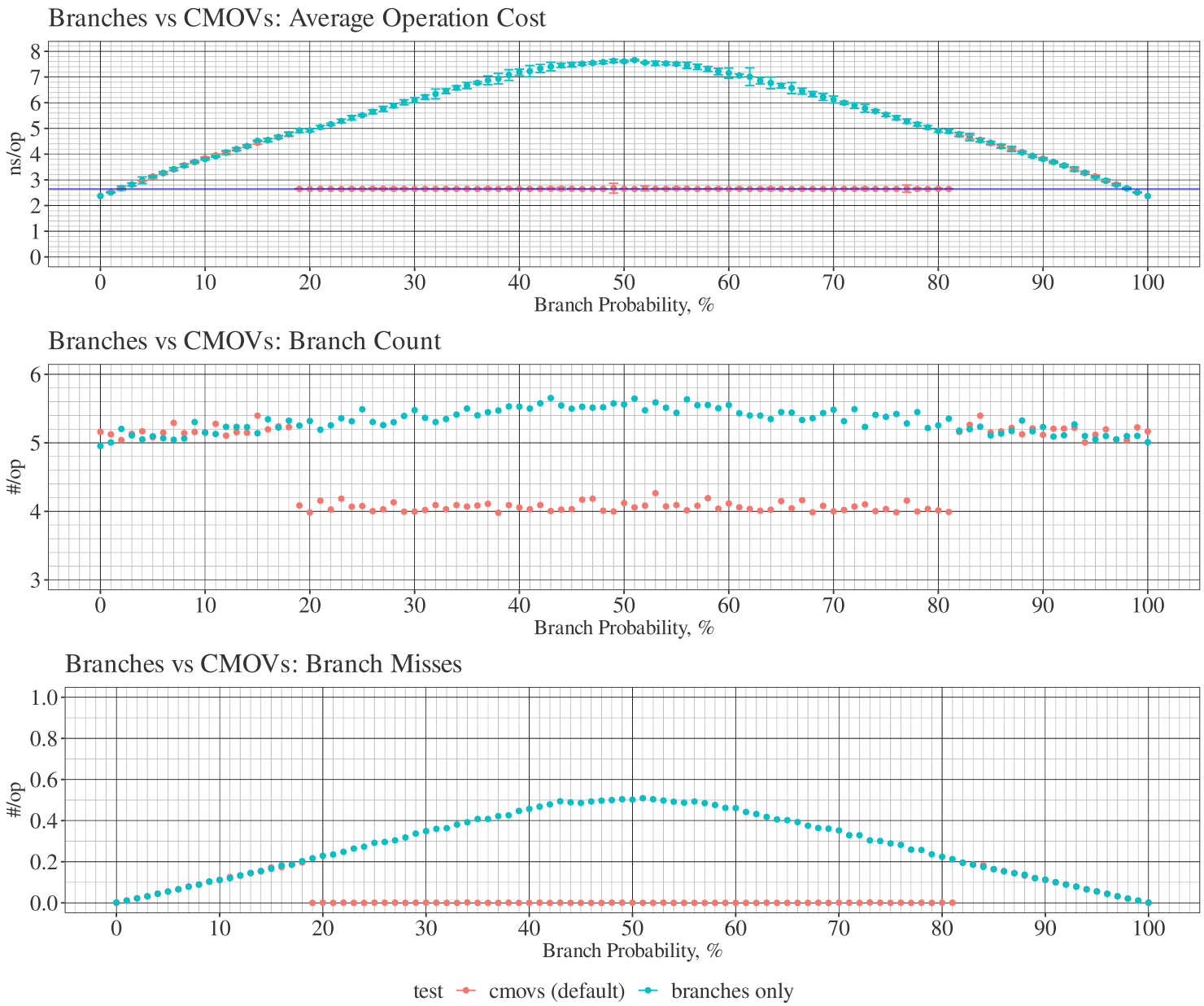
You might think that was because the branch misprediction penalty is not there for CMOV, but that explanation is at odds with counters. Note that "branch-misses" is nearly zero in both cases. This is because hardware branch predictors can actually remember a short branching history, and this flip-flopping branch poses no problems to them. The actual cause for performance difference is jumping in the branchy case: we have an additional control-flow instruction on the critical path.

Practice 2

To see the effects of branch misprediction penalties, we need to do a more advanced test, like this:

```java
@Warmup(iterations = 5, time = 500, timeUnit = TimeUnit.MILLISECONDS)
@Measurement(iterations = 5, time = 500, timeUnit = TimeUnit.MILLISECONDS)
@Fork(1)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Thread)
public class AdjustableBranchFreq {

    @Param("50")
    int percent;

    boolean[] arr;

    @Setup(Level.Iteration)
    public void setup() {
        final int SIZE = 100_000;
        final int Q = 1_000_000;
        final int THRESH = percent * Q / 100;
        arr = new boolean[SIZE];
        ThreadLocalRandom current = ThreadLocalRandom.current();
        for (int c = 0; c < SIZE; c++) {
            arr[c] = current.nextInt(Q) < THRESH;
        }

        // Avoid uncommon traps on both branches.
        doCall(true);
        doCall(false);
    }

    @Benchmark
    public void test() {
        for (boolean cond : arr) {
            doCall(cond);
        }
    }

    @CompilerControl(CompilerControl.Mode.DONT_INLINE)
    public int doCall(boolean condition) {
        if (condition) {
            return 1;
        } else {
            return 2;
        }
    }
}
```

Running it with different `percent` values and `-prof perfnorm` JMH profiler would yield this:



You can clearly see a few things:

1. The number of branches per test is about 5, and CMOV conversion drops it to 4. This correlates with the disassembly dumps before: we have one of the branches in the test converted to CMOV. Another 4 branches are from the test infrastructure itself.
2. Without CMOVs, the branch test performance suffers and gets the worst at 50% branch probability. This peak reflects the nearly-absolute confusion of hardware branch predictor, as it experiences about 0.5 branch misses per operation. This means the branch predictor does not guess wrong all the time (that would be luducrous!), but just half of the time. I speculate that history-based predictor just gives up and lets the static predictor choose the closest branch, which we take half of the time.
3. With CMOVs we can see nearly-flat operation times *once it kicks in*. This graph says that CMOV cost model was probably too conservative for this test, and it switched a bit too late. It does not necessarily mean it has a bug, because other cases would quite probably perform differently. Still, the improvements against branchy case are massive when CMOV conversion takes place.
4. You might notice that branchy variant dips under CMOV middle average when branches are predicted with >97% accuracy. Of course, this is again test, HW, VM-specific thing.

Conclusion

Branch profiling allows making more or less informed choices about doing the probability-sensitive instruction selection. Conditional moves replacement routinely uses branch frequency information to drive the substitution. Once again, this underlines the need to warm up the JIT-compiled code with the data that resemble real data, so that compilers can optimize efficiently for the particular case. Unresolved directive in complete.adoc - include::../31-profile-pollution/index.adoc[]

1. Actually it produces one less load-store pair too, which is the side effect of better register allocation.
2. Really, more sensible in the way just-in-time compilers should work, which is the theme for the next post I was writing before discovering my experiments are toasted because of this pitfall
3. This does not preclude flow-based optimizations, like calling inlined `work(new M(4242))`
4. Doing the same test with `int`-s instead of `long`-s would yield actual `mov $0x1, %edx`, but I am too lazy to reformat all the assembly listings for this case.
5. And this had not fully worked, because default inlining heuristics still counts the method size by the bytecode length, regardless of how much dead code is there. This gradually changes with e.g. incremental inlining.
6. This almost inevitably devolves into template and/or metaprogramming mess we love to write, but hate to debug.
7. Interface calls would be handled similarly, but with a twist during resolution and invocation in the stub.
8. Yet another instance of "Profilers Are Lying Hobbitses (and we hate them!)" (https://www.infoq.com/presentations/profilers-hotspots-bottlenecks)
9. I am mildly irritated when people claim EA does something: it's not, further optimizations do!
10. Like the ones the intermediate representation has for local variables and other temporary operands compiler wants to have
11. For example, Graal is known to have Partial Escape Analysis, that is supposed to be more resilient in complex data flows
12. The extreme case of this technique is using vector registers as line buffer!
13. Some register allocators do perform *linear* allocation — bringing up the speed of regalloc, trading generated code efficiency
14. "Garbage collector" is misnomer, because GC also normally takes care of allocating the memory in the heap structured as GC wants it. See e.g. Epsilon GC (https://openjdk.java.net/jeps/318)
15. There are some intricacies in this. For example, Linux would not commit actual memory until the very first use, even if application thinks the memory is available and owned by the process.
16. Full disclosure: I have implemented most of heap uncommit handling in Shenandoah, and this post is basically the re-run of our early experiments with it. If this post feels like the advert for Shenandoah, that's because it is.
17. Enabled with `-XX:ShenandoahGCHeuristics=compact`
18. In Linux/POSIX case, calling (http://hg.openjdk.java.net/jdk/jdk/file/af7afdababd3/src/hotspot/os/linux/os_linux.cpp#l3336) `mprotect(PROT_NONE)` is enough (http://hg.openjdk.java.net/jdk/jdk/file/af7afdababd3/src/hotspot/os/linux/os_linux.cpp#l5074).
19. Well, almost. On x86, it changes flags, but next instructions would overwrite them anyway, and we only need to take care of never emitting the safepoint poll between real `test` and the associated `jCC`.
20. Thread-local storage is the piece of native data that is accessible per thread. On many platforms, where register pressure is not very high, generated code always have it in a register. On x86_64, it is usually (http://hg.openjdk.java.net/jdk/jdk/file/af7afdababd3/src/hotspot/cpu/x86/x86_64.ad#l12878) `%r15`.
21. Technically, stopping a subset of threads does not get us to a "safepoint" anymore. But, when thread-local handshakes are enabled, the safepoint can be reached by handshaking with all threads. This covers both "safepoint" and "handshake" cases wholesale.
22. This, and other examples of disassembled code are generated with the help of JMH (https://openjdk.java.net/projects/code-tools/jmh/) `-prof perfasm`.
23. This, and further memory representation dumps are done with the help of JOL (https://openjdk.java.net/projects/code-tools/jol/).
24. Technically, the instance size is also down because class word in object header got compressed as well. Digging into that is outside the scope for this post.
25. There is also another interesting "disjoint" variation of this mode, which activates when heap base has a useful alignment, but this is beyond the scope for this post. You can dig into source, starting from narrow oop mode (http://hg.openjdk.java.net/jdk/jdk/file/f3fd73c3a077/src/hotspot/share/memory/universe.hpp#l374).
26. The Epsilon output form you would see in this post is available in newer versions that ship in upcoming 11.0.3, 12.0.2, and head JDK.
27. The non-simple approach would be teaching memory manager to allocate objects at different starting addresses. This requires examining objects on allocation path, which is *interesting* from performance standpoint.
28. This, and further memory representation dumps are done with the help of JOL (https://openjdk.java.net/projects/code-tools/jol/).
29. Back-envelope calculation: since `-XX:ObjectAlignmentInBytes` in Hotspot accepts alignments up to 256, this means max 8-bit shift, and 1024 GB max.
30. We use 8u — instead of whatever the bleeding edge JDK release these days is — to show this optimization is not very new ;)
31. In more complicated cases, the simplified control flow and free register/flags not used by the explicit null check give the nice code quality improvements.
32. In this code, it actually feeds into so called "uncommon trap", the topic we would cover eventually. Briefly, this is the notification to the runtime that some never-taken branch is actually taken, and asking JVM to recompile the method using that fact.
33. While this benchmark shows the dynamic recompilation, it can be shown the same effect would be achieved if we fed `null`-s, and thus updated the initial profile, before the benchmark code was executed.
34. That `0x…bfe: mov %r12d,0x10(%rsi)` is a nice low-level WTF (https://twitter.com/shipilev/status/1213203153598001154).
35. `-prof perfasm` filters everything that happens during warmup iterations, and this is why we don't see the disassembly from the previous test.
36. Note this is not a problem for the *reference comparisons* with `==`. In the overwhelming majority of VM implementations, reference equality compares addresses. This is mostly because even if GC moves the objects, the Java/runtime code sees the consistent addresses for a given object. If object A had moved, then all references to A are updated to new address "at the same time", either due to stop-the-world GC, or with the help of GC barriers.
37. It is even worse for 32-bit VMs, that store only the first 25 bits, see more discussion about it here (https://shipilev.net/jvm/objects-inside-out/#_identity_hash_code).
38. Somewhat amuzingly, I have accidentally changed it from 0 to 5 after doing the performance experiments for JDK-8006176 (https://bugs.openjdk.java.net/browse/JDK-8006176). This process mistake still haunts me.
39. See JDK-8199394 (https://bugs.openjdk.java.net/browse/JDK-8199394).
40. Here and later, all disassembly code is produced with the help of JMH `-prof perfasm`.
41. I always wondered if it is possible for hardware to skip executing/retiring instructions that are obviously non-observable. As of today, I have never seen it in practice. If such a case manifests, then compiler blackholes would break.
42. JMH can technically detect the compiler blackhole support better. This UX improvement is left for future work.
43. It would be just 2 accesses, not 3, if not the Thread-Local Handshakes (https://shipilev.net/jvm/anatomy-quarks/22-safepoint-polls/).
44. Note how the benchmark forbids the inlining of the method. This both improves the generated code fidelity, as we can see the small method compiled in full, and it also fits a rather unconventional practice

(https://github.com/openjdk/jmh/blob/0d2a82094d91a782c5d2f2d594987be5c88e956f/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_34_SafeLooping.java#L151-L174) of forbidding the inlining of the method, instead of Blackhole-ing its result.

Last updated 2021-07-31 13:30:32 +0300