Why we need Off Heap memory

and

how to cook it?

Heap Memory

- Tracing garbage collection
- Reference counting (used in pooled objects like Netty’s ByteBuff)
- Allocate until death (Epsilon GC)

Heap Memory

POJO

(Records are coming

https://openjdk.java.net/jeps/359

but immulable)

Heap Memory

JOL - Java Object Layout

https://openjdk.java.net/projects/code-tools/jol/

Heap Memory

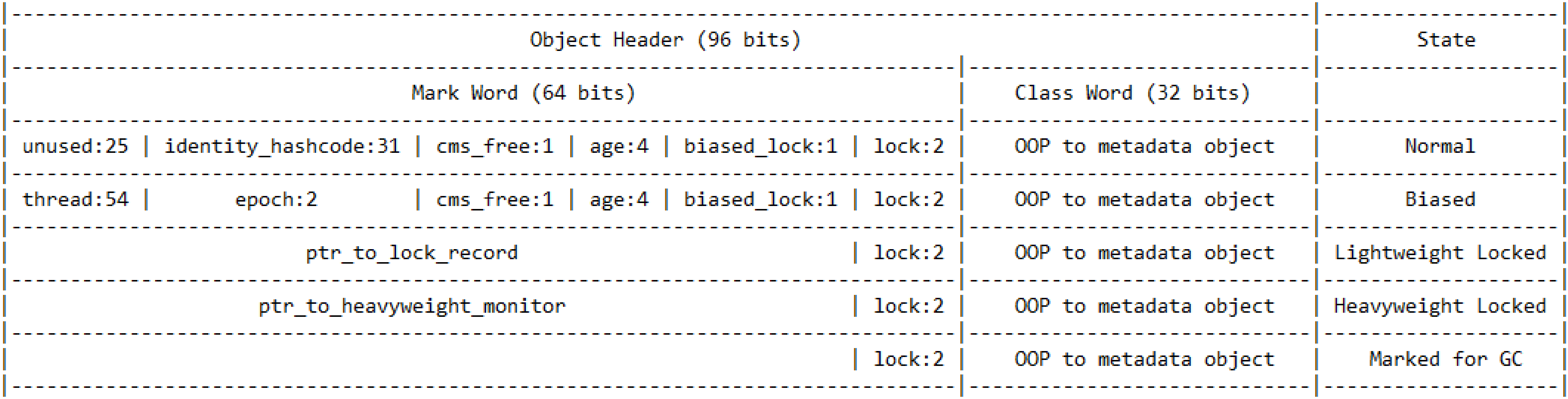
Object Layout

- Object header
- Fields aligned

Heap Memory

Object Header

64bit compressed OOPs

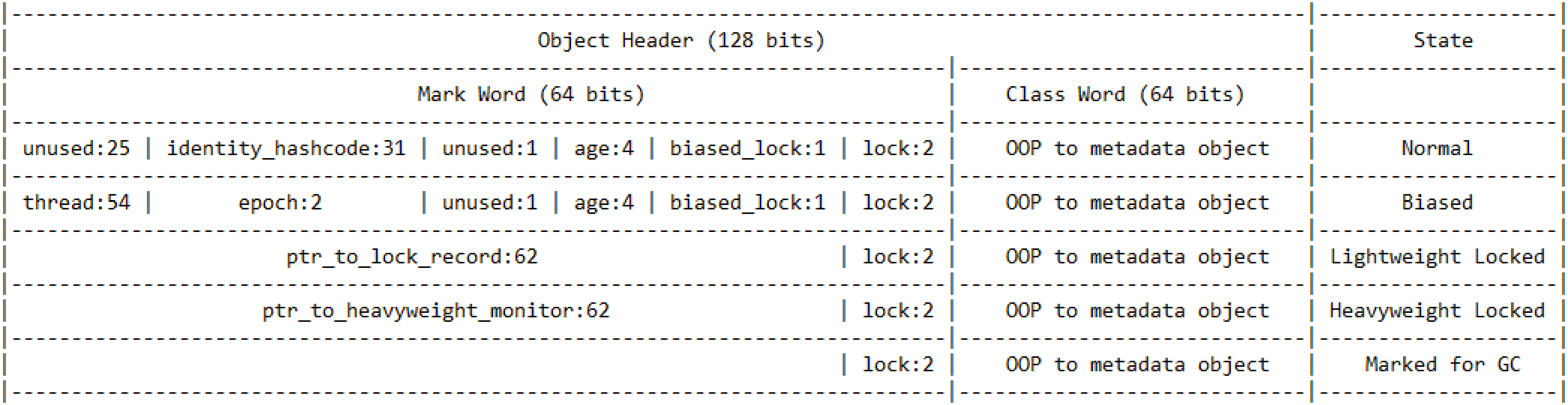


Object Header (96 bits)							State
Mark Word (64 bits)						Class Word (32 bits)	
unused:25	identity_hashcode:31	cms_free:1	age:4	biased_lock:1	lock:2	OOP to metadata object	Normal
thread:54	epoch:2	cms_free:1	age:4	biased_lock:1	lock:2	OOP to metadata object	Biased
ptr_to_lock_record					lock:2	OOP to metadata object	Lightweight Locked
ptr_to_heavyweight_monitor					lock:2	OOP to metadata object	Heavyweight Locked
					lock:2	OOP to metadata object	Marked for GC

Heap Memory

Object Header

64bit, not compressed OOPs



Object Header (128 bits)							State
Mark Word (64 bits)						Class Word (64 bits)	
unused:25	identity_hashcode:31	unused:1	age:4	biased_lock:1	lock:2	OOP to metadata object	Normal
thread:54	epoch:2	unused:1	age:4	biased_lock:1	lock:2	OOP to metadata object	Biased
ptr_to_lock_record:62					lock:2	OOP to metadata object	Lightweight Locked
ptr_to_heavyweight_monitor:62					lock:2	OOP to metadata object	Heavyweight Locked
					lock:2	OOP to metadata object	Marked for GC

Heap Memory

Field alignment

Not aligned data:

- **X86 family – Yes, We Can**
- **ARM - Oooooops**

Heap Memory

Heap-related issues:

- Memory footprint overhead (direct and indirect (GC structures))
- CPU overhead for both allocation and GC, GC pauses/latency
- Bad cache locality

Heap Memory

CompressedOops

-XX:+UseCompressedOops enabled by default

https://shipilev.net/jvm/anatomy-quarks/23-compressed-references/

Heap Memory

GC related overhead

GC	Overhead, MB	%
Serial	7	0.3%
Shenandoah	38	1.9%
CMS	76	3.7%
Parallel	90	4.4%
G1	166	8.1%
Z	238	11.6%

OpenJDK 13, Heap size = 2GB

Off-Heap Memory

- DirectByteBuffer/MappedByteBuffer
- sun.misc.Unsafe
- Custom JNI

Off-Heap Memory

DirectByteBuffer

Pros:

- Standard API for NIO

Cons:

- Not easy to free (Cleaner)

- 2GB limit

Off-Heap Memory

sun.misc.Unsafe

Pros:

- Easy (almost) to use

- No 2GB limitation

- Heap access supported

Cons:

- Internal/not public API

- Malloc’s issues like fragmentation

Off-Heap Memory

Panama Project

https://openjdk.java.net/projects/panama/

Off-Heap Memory

Own JNI

- No dependencies to Unfase
- Performance penalty

Off-Heap Memory

JNI call cost

- Create stack frame
- Converting arguments according to ABI
- Converting oops to JNI handles (jobject)
- Putting addition JNIEnv* and jclass
- Lock/release object monitor for synchronized method
- Lazy native function linking
- Tracing of enter/return into/from the method
- Changing thread state from in_Java to in_native and back
- Check a safepoint requested
- Exception handling

Off-Heap Memory

Secret weapon - JavaCritical

https://bugs.openjdk.java.net/browse/JDK-7013347

Off-Heap Memory

```
JNIEXPORT jint JNICALL
JavaCritical_compareArrays(jlong
address, jint length) {
    return compareWithSIMD(address,
length);
}
```

- **Must be static/not synchronized**
- **Primitives and primitive arrays only**
- **Must Not call JNI (no object allocations, no exceptions)**
- **Be as fast as possible (GC blocked)**

Off-Heap Memory

Common disadvantage -

no intrinsics available

https://cr.openjdk.java.net/~vlivanov/talks/2017_Vectorization_in_HotSpot_JVM.pdf

Vectorization API in Project Panama

Off-Heap Memory

Async-profiler

https://github.com/jvm-profiling-tools/async-profiler

```
profiler.sh -d 10 -f ./flamegraph.svg -e mprotect -t $JAVAPID
```

Off-Heap Memory

Performance testing?

JMH is the answer

https://openjdk.java.net/projects/code-tools/jmh/