Lilliput

The Journey

Roman Kennke (@rkennke)

Principal Engineer
Amazon

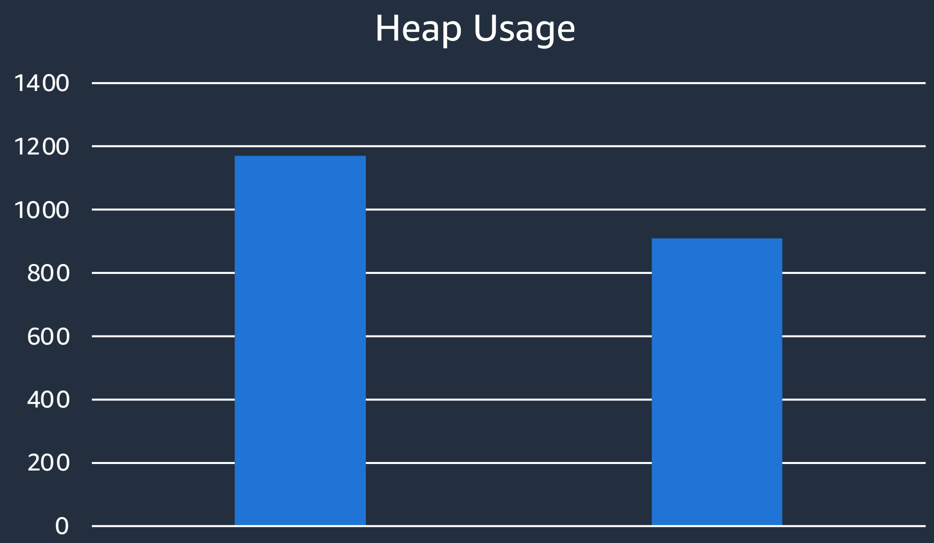
What is Lilliput about?

Reduce memory footprint of Java applications

Reduce size of object headers from 12 to 4 bytes

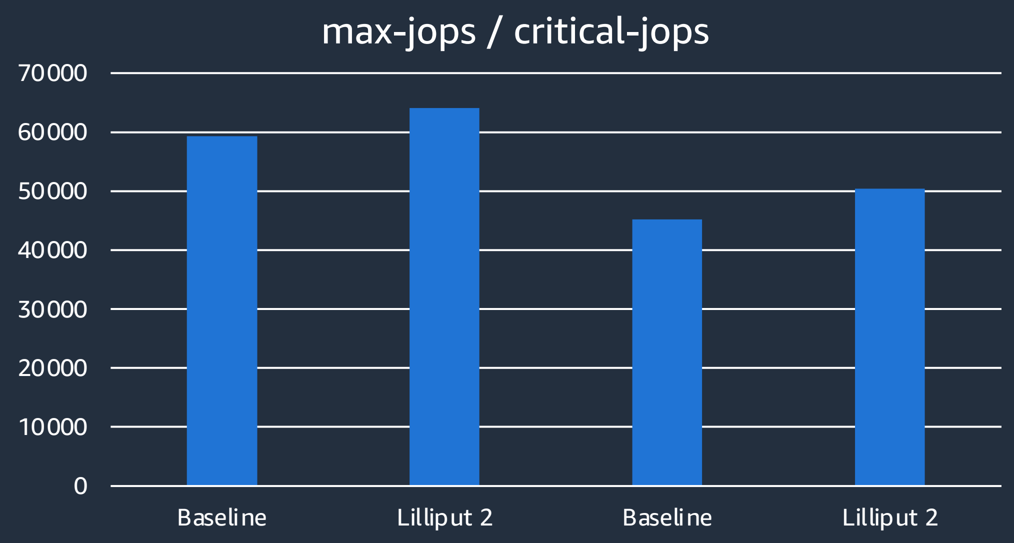
Typically adds up to 20-30% heap reductions

What is Lilliput about?



-22%

What is Lilliput about?



+8%

+11.4%

2020

FOSDEM 2020

JDK 14 around the corner

Shenandoah 2.0

How hard can it be?

2021

First commit in March 2021

Lots of prototyping

Class-Pointer improvements

2022

April: First attempts to replace stack-locking

-> Finished in May 2023 (JDK 21)

-> Default since March 2024 (JDK 23)

-> Recursive locking support in April 2024 (JDK 23)

-> Object monitor table in August 2024

2023

March: Finished Lightweight locking (JDK 21)

May: First proposed version of JEP 450 implementation

2024

April:	Recursive lightweight locking
August:	Object monitor table
November:	Integrated JEP 450 into JDK 24

2025

January: Prototype Lilliput 2 aka 4-byte-headers

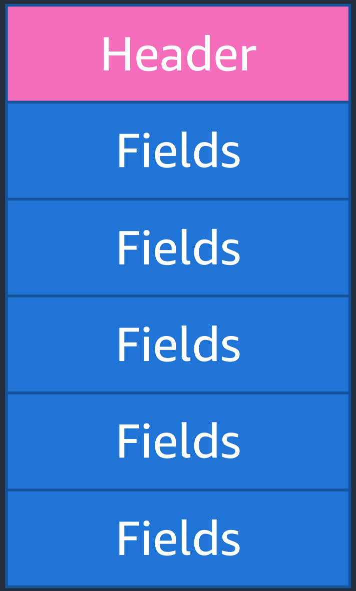
March: EA of JDK 24

What’s in it?

Current state

What’s in it?

Lilliput 1



What's in it?

Lilliput 2

What’s in it?

```
Mark Word (normal):

 64                     39                              8    3  0
  [.......................HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH.AAAA.TT]
           (Unused)                (Hash Code)         (GC Age)(Tag)
```

```
Class Word (compressed):
 32                               0
  [CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC]
      (Compressed Class Pointer)
```

Insight:
- Most objects never get i-hashed
- Most objects never get locked

What’s in it?

```
Mark Word (overwritten):

 64                                                            2 0
  [ppppppppppppppppppppppppppppppppppppppppppppppppppppppppppppppTT]
                   (Native Pointer)                        (Tag)
```

-> Pointer into stack (for stack-locking) (tag = 00)

-> Pointer to ObjectMonitor (for monitor-locking) (tag = 10)

-> Pointer to forwarded object (for GC forwarding) (tag = 11)

Lilliput 1

```
Header (compact):
 64                    42                                  7    3 0
  [CCCCCCCCCCCCCCCCCCCCCCHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHVVVVAAAASTT]
   (Compressed Class Pointer)               (Hash Code) (GC Age)^(Tag)
                                                 (Self Forwarded Tag)
```

- 22 Klass bits
- 31 Hash bits
- 4 Valhalla reserved bits
- 4 GC age bits
- 2+1 Tag bits

Lilliput 2 – The Plan

```
Header (Lilliput 2):
 32                  11   7   3  0
  [CCCCCCCCCCCCCCCCCCCHHVVVVAAAASTT]
       (Class Pointer)    ^(Age)^(Tag)
                 (Hash-Code) (Self Forwarded Tag)
```

- 19 Klass bits
- 2 Hash control bits
- 4 Valhalla reserved bits
- 4 GC age bits
- 2+1 Tag bits

Compact Identity Hash-Code

- Recap:
  - x == y   => System.identityHashCode(x) == System.identityHashCode(x)
  - ‘==‘ is object identity, not equality
  - Inverse not necessarily true
  - Id-hash must remain stable throughout object lifetime (idempotence)
  - Shoot for great entropy
  - System.identityHashCode() is default impl for Object.hashCode()

Compact Identity Hash-Code

- Current situation:
  - Allocate 31 bits slot for every object
  - First call to System.identityHashCode() installs good hash (RNG)
  - Subsequent calls read and return that hash
  - Only small fraction of objects ever use it

Compact Identity Hash-Code

- Naïve Idea:
  - Calculate I-hash based on object address
- Problem:
  - GCs can move objects around
  - -> not idempotent

Compact Identity Hash-Code

- Idea:
  - Allocate slot for id-hash on-demand
- Problem:
  - Heap is filled contiguously
  - There is no space available

Compact Identity Hash-Code

- Header (Lilliput 2):

```
 32                   11   7   3  0
  [CCCCCCCCCCCCCCCCCCCHHVVVVAAAASTT]
       (Class Pointer)    ^(Age)^(Tag)
                 (Hash-Code) (Self Forwarded Tag)
```

Hash state represented in 2 hash bits:

- Bit 1: not-hashed (0) vs hashed (1)
- Bit 2: not-expanded (0) vs expanded (1)

Compact Identity Hash-Code

- 00: Not-hashed/not-expanded (initial state)
- First call to System.identityHashCode() on an object:
  - Transitions state to 01: hashed/not-expanded
  - Calculates hash-code based on object address

Compact Identity Hash-Code

- 01: Hashed/Not-expanded
- Subsequent calls to System.identityHashCode():
  - Re-calculate hash-code based on object address

Compact Identity Hash-Code

- First GC after object transitioned to 01: hashed/not-expanded:
  - Calculate hash-code based on original object address
  - Copy object, maybe extend to make space for i-hash-slot
    - No need to expand if alignment gap is available
  - Store i-hash in i-hash-slot
  - Transition to state 11: hashed/expanded

Compact Identity Hash-Code

- 11: Hashed/Expanded
- Subsequent calls to System.identityHashCode():
  - Read i-hash from i-hash-slot
- I-hash slot precomputed by field layouter
  - Can often use gaps

Compact Identity Hash-Code

```
Header (Lilliput 2):
 32                   11   7   3  0
  [CCCCCCCCCCCCCCCCCCCHHVVVVAAAASTT]
       (Class Pointer)    ^(Age)^(Tag)
                 (Hash-Code) (Self Forwarded Tag)
```

- Requires only 2 bits per object
- Allocate space for i-hash only when needed
- Majority of objects never need i-hash
- ~50% of objects have gaps available

GC Forwarding

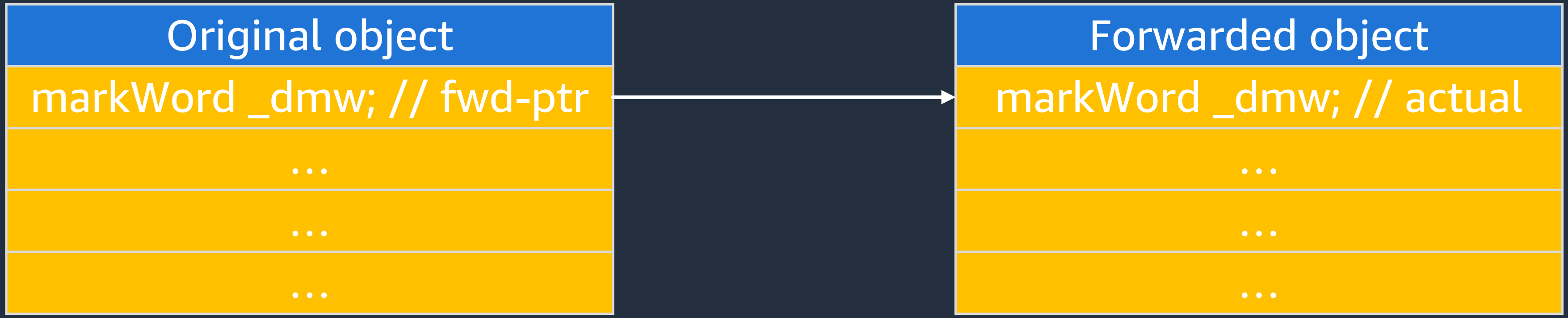
- Current situation:

```
Mark Word (normal):
 64                     39                              8    3  0
  [.......................HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH.AAAA.TT]
           (Unused)               (Hash Code)         (GC Age)(Tag)


Mark Word (GC forwarded):

 64                                                            2 0
  [pppppppppppppppppppppppppppppppppppppppppppppppppppppppppppp11]
                (Forwarding Pointer)                          (Tag)
```

GC Forwarding

- 2 cases:
  - Copying compaction (non-destructive)
  - Sliding compaction (destructive)

GC Forwarding – Copying Compaction



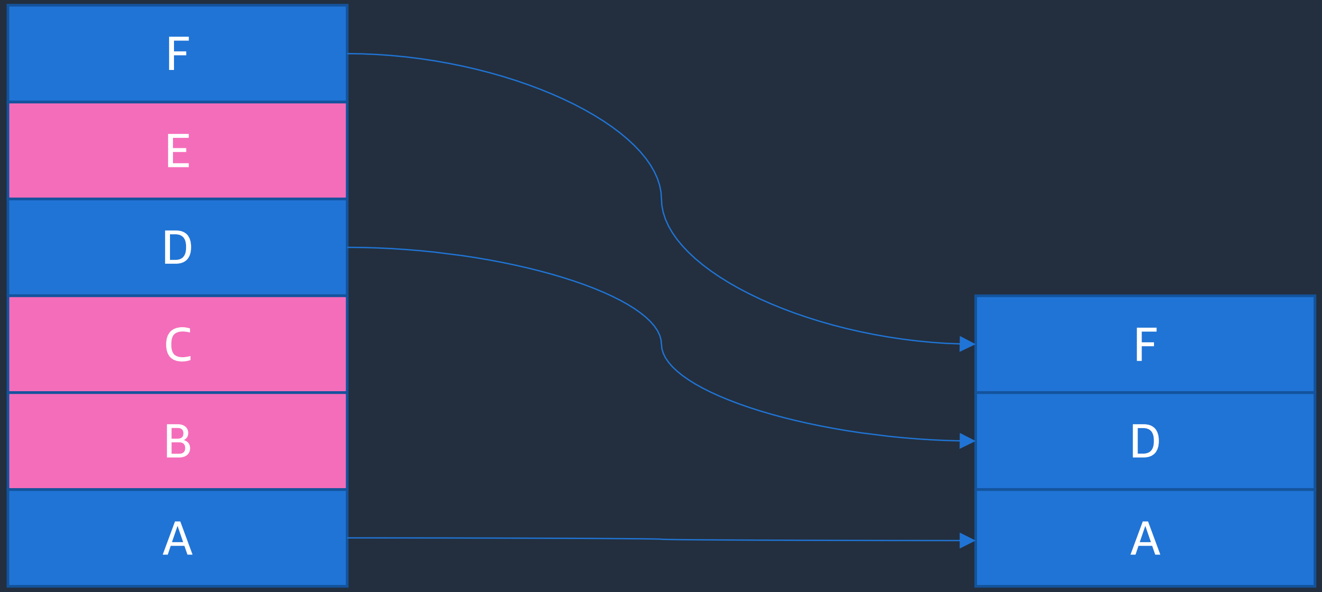
GC Forwarding – Copying Collection

- 3 phases:
  1. Marking (find live objects)
  2. Copy objects to empty space (write forwarding pointer into old object)
  3. Update all references

GC forwarding - Copying Compaction

- Easy: keep using first word as forwarding pointer
- Overwrites first field(s) of object (ok, we have a copy)
- Overwrites length of arrays (ok, but we need to be careful)

GC Forwarding - Sliding Compaction



GC Forwarding - Sliding Compaction

- 4 phases:
  1. Marking (find live objects)
  2. Calculate forwarding addresses (write them into object headers)
  3. Update all references
  4. Copy objects to new locations

GC Forwarding - Sliding Compaction

- Problem:
  - Forwarding overwrites Klass* (and array-length, hash-bits)
  - … but we don't have a copy
  - … and we only have 9 bits

```
Header (Lilliput 2):
 32                  11   7   3  0
  [CCCCCCCCCCCCCCCCCCCCHHFFFFFFFFFTT]
       (Class Pointer)    ^(Age)^(Tag)
                 (Hash-Code) (Self
Forwarded Tag)
```

GC Forwarding – Sliding Compaction

- Idea:
  - 9 bits are enough to address $2^9 = 512$ words range
  - Divide heap into 512 words (=4KB) blocks
  - Use side-table with one entry per block (= 1/512th of heap size)
  - Table stores forwarding base for each block
    - It's the forwarding address of the first live object in the block
  - Use 9 header bits to find forwarding address of object

```
fwd_addr = fwd_base[block] + fwd_bits << 3
```

GC Forwarding – Sliding Forwarding

- Fixed-overhead side table (1/512th of heap size)
- No performance loss
- Potential to reduce to 1/2048th of heap size, by using tag bits
- Eliminates 8TB limit of Lilliput 1

Putting it all together

- Tiny Class-Pointers
  - Allow 19 bit class-pointers
- Compact Identity Hash-Code
  - Need only 2 header bits for i-hash
- Improved full-GC forwarding table
- 4 Valhalla bits

```
• Header (Lilliput 2):
 32                  11   7   3  0
  [CCCCCCCCCCCCCCCCCCCHHVVVVAAAASTT]
      (Class Pointer)    ^(Age)^(Tag)
               (Hash-Code) (Self Forwarded Tag)
```

Putting it all together

- `-XX:+UseCompactObjectHeaders`