Effective Software

Lecture 10: JVM - Object Allocation, Bloom Filters, References, Effective Caching

David Šišlák
david.sislak@fel.cvut.cz
Fast Object Allocation

» based on bump-the-pointer technique
   • track previously allocated object
   • fit new object into remainder of generation end

» thread-local allocation buffers (TLABs)
   • remove concurrency bottleneck
     • no synchronization among threads (remove slower atomics)
     • remove false sharing (cache line used just by one CPU core)
   • exclusive allocation takes about few native instructions
   • each thread has small exclusive area (few % of Eden in total) aligned NUMA
   • infrequent full TLABs implies synchronization (based on lock inc)
   • thread-based adaptive resizing of TLAB
     • not working well for thread pools with varying allocation pressure

» tuning options
   • -XX:+UseTLAB ; -XX:AllocatePrefetchStyle=1; -XX:+PrintTLAB
   • -XX:AllocateInstancePrefetchLines=1 ; -XX:AllocatePrefetchLines=3
   • -XX:+ResizeTLAB ; -XX:TLABSize=10k ; -XX:MinTLABSize=2k
Fast Object Allocation - Example

C2 compiler, standard OOP, size 96 Bytes:

- read thread allocation pointer
- bump the pointer
- fits into TLAB?
- store thread allocation pointer
- prefetch 3 cache lines ahead
- prepare for object nulling
- RDI object data; ECX=10 qwords
- fill object header
- null instance

```assembly
mov  0x60(%r15),%r11
mov  %r11,%r10
add  $0x60,%r10
cmp  0x70(%r15),%r10
jae  0x00000000107895244
mov  %r10,0x60(%r15)
prefetchnta 0xc0(%r10)
mov  %r11,%rdi
add  $0x10,%rdi
mov  $0xa,%ecx
movabs $0x220558080,%r10 ; {metadata('Structure')}
mov  0xa8(%r10),%r8
mov  %r8,(%r11)
mov  %r10,0x8(%r11)
xor  %rax,%rax
shl  $0x3,%rcx
rep rex.W stos %al,%es:(%rdi) ;*new
    ; - StructureTest::allocate@4 (line 5)

8B - mark word
4B / 8B – Klass ref.
... object data
```

class Structure {
    private boolean boolean1;
    private byte byte1;
    private char char1;
    private short short1;
    private int int1;
    private long long1;
    private float float1;
    private double double1;
    private Object object1;
    private boolean boolean2;
    private byte byte2;
    private char char2;
    private short short2;
    private int int2;
    private long long2;
    private float float2;
    private double double2;
    private Object object2;
}

Structure(int value, Object r

@Override
public String toString() {
    return "Structure\n" + value + "\n" + object + "\n";
Flight Recording to Analyze TLAB

example with million of allocations of Structure

<table>
<thead>
<tr>
<th>Thread Local Allocation Buffer (TLAB) Statistics</th>
<th>Statistics for Object Allocations (Outside TLABs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLAB Count</td>
<td>Object Count</td>
</tr>
<tr>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>Maximum TLAB Size</td>
<td>Maximum Object Size</td>
</tr>
<tr>
<td>720.58 kB</td>
<td>N/A</td>
</tr>
<tr>
<td>Minimum TLAB Size</td>
<td>Minimum Object Size</td>
</tr>
<tr>
<td>615.77 kB</td>
<td>N/A</td>
</tr>
<tr>
<td>Average TLAB Size</td>
<td>Average Object Size</td>
</tr>
<tr>
<td>718.96 kB</td>
<td>N/A</td>
</tr>
<tr>
<td>Total Memory Allocated for TLABs</td>
<td>Total Memory Allocated for Objects</td>
</tr>
<tr>
<td>45.64 MB</td>
<td>N/A</td>
</tr>
<tr>
<td>Allocation Rate for TLABs</td>
<td>Allocation Rate for Objects</td>
</tr>
<tr>
<td>439.59 MB/s</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Allocation

[Graph showing allocation statistics over time]
Flight Recording to Analyze TLAB

example with million of allocations of Structure

<table>
<thead>
<tr>
<th>Class</th>
<th>Average Object Size</th>
<th>TLABs</th>
<th>Total TLAB Size</th>
<th>Pressure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>80 bytes</td>
<td>64</td>
<td>44.93 MB</td>
<td>98.46%</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>88 bytes</td>
<td>1</td>
<td>720.58 kB</td>
<td>1.54%</td>
</tr>
</tbody>
</table>

Stack Trace

- StructureTest.allocate(int, Object)

TLABs | Total TLAB Size  | Pressure |
---    |------------------|----------|
64     | 44.93 MB         | 100.00%  |
Example – Dynamic Memory Analysis

```java
public static String[] method1(String[] args) {
    return Arrays.stream(args).
        filter(t -> t.matches( regex: "[^0-9]+")) .
        sorted(Comparator.<String,String>comparing(String::toLowerCase).reversed()).
        collect(Collectors.toList()).toArray(new String[0]);
}
```
Example – Dynamic Memory Analysis

```java
public static String[] method1(String[] args) {
    return Arrays.stream(args).
        filter(t -> t.matches( regex: "[^0-9]+" )).
        sorted(Comparator.<String,String>comparing(String::toLowerCase).reversed()).
        collect(Collectors.toList()).toArray(new String[0]);
}
```

allocations when called with 40 elements (27 without digits):

<table>
<thead>
<tr>
<th>Method</th>
<th>Objects</th>
<th>Size</th>
<th>Objects (Own)</th>
<th>Size (Own)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda$_method1(String[]) Lambda.java</td>
<td>1,486</td>
<td>101,944</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

Class list for objects selected in the upper table

<table>
<thead>
<tr>
<th>Class</th>
<th>Objects</th>
<th>Shallow Size</th>
<th>Retained Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>int[]</td>
<td>167</td>
<td>12,648</td>
<td>12,648</td>
</tr>
<tr>
<td>byte[]</td>
<td>57</td>
<td>12,136</td>
<td>12,136</td>
</tr>
<tr>
<td>char[]</td>
<td>179</td>
<td>10,928</td>
<td>10,928</td>
</tr>
<tr>
<td>boolean[]</td>
<td>40</td>
<td>10,880</td>
<td>10,880</td>
</tr>
<tr>
<td>jdk.internal.org.objectweb.asm.Item</td>
<td>180</td>
<td>10,080</td>
<td>10,080</td>
</tr>
<tr>
<td>jdk.internal.org.objectweb.asm.Item[]</td>
<td>7</td>
<td>7,280</td>
<td>7,280</td>
</tr>
<tr>
<td>java.lang.Class</td>
<td>7</td>
<td>3,968</td>
<td>3,968</td>
</tr>
<tr>
<td>jdk.internal.org.objectweb.asm.MethodWriter</td>
<td>15</td>
<td>3,360</td>
<td>3,360</td>
</tr>
<tr>
<td>java.util.regex.Pattern</td>
<td>40</td>
<td>2,880</td>
<td>2,880</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>113</td>
<td>2,712</td>
<td>2,712</td>
</tr>
<tr>
<td>java.util.regex.Matcher</td>
<td>40</td>
<td>2,560</td>
<td>2,560</td>
</tr>
<tr>
<td>java.util.regex.Pattern$GroupHead[]</td>
<td>40</td>
<td>2,240</td>
<td>2,240</td>
</tr>
<tr>
<td>java.util.regex.Pattern$Curly</td>
<td>40</td>
<td>1,280</td>
<td>1,280</td>
</tr>
<tr>
<td>java.lang.invoke.MethodType</td>
<td>30</td>
<td>1,200</td>
<td>1,200</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>24</td>
<td>1,200</td>
<td>1,200</td>
</tr>
<tr>
<td>jdk.internal.org.objectweb.asm.ClassWriter</td>
<td>7</td>
<td>1,176</td>
<td>1,176</td>
</tr>
<tr>
<td>java.lang.invoke.MemberName</td>
<td>15</td>
<td>840</td>
<td>840</td>
</tr>
<tr>
<td>java.lang.invoke.MethodType$ConcurrentWeakIntSetWeakEntry</td>
<td>30</td>
<td>960</td>
<td>960</td>
</tr>
</tbody>
</table>
Example – Optimized – Dynamic Memory Analysis

```java
private static Comparator<String> reverseIgnoreCaseComparator = String.CASE_INSENSITIVE_ORDER.reversed();

public static String[] reversedAlphabeticalOnlyOrderOptimized(String[] args) {
    String[] arr = new String[args.length];
    int i = 0;
    for (String arg : args) {
        boolean filterOut = false;
        for (int k = 0; k < arg.length(); k++) {
            char c = arg.charAt(k);
            if ((c >= '0') && (c <= '9')) {
                filterOut = true;
                break;
            }
        }
        if (!filterOut) arr[i++] = arg;
    }
    Arrays.sort(arr, fromIndex: 0, i, reverseIgnoreCaseComparator);

    return Arrays.copyOf(arr, i);
}
```
Example – Optimized – Dynamic Memory Analysis

```java
private static Comparator<String> reverseIgnoreCaseComparator = String.CASE_INSENSITIVE_ORDER.reversed();

public static String[] reversedAlphabeticalOnlyOrderOptimized(String[] args) {
    String[] arr = new String[args.length];
    int i = 0;
    for (String arg : args) {
        boolean filterOut = false;
        for (int k = 0; k < arg.length(); k++) {
            char c = arg.charAt(k);
            if (((c >= '0') && (c <= '9'))) {
                filterOut = true;
                break;
            }
        }
        if (!filterOut) arr[i++] = arg;
    }
    Arrays.sort(arr, fromIndex: 0, i, reverseIgnoreCaseComparator);
    return Arrays.copyOf(arr, i);
}

allocations when called with 40 elements (27 without digits):
```

![Memory allocation table](image-url)
Know Your Application Behavior

» simple code could be very inefficient – **know what you are using**

» a lot of **small short-lived objects** still slow down your application
  • allocations in TLAB are quite fast but not as no as fast as no allocation
    – check escape analysis or change your code
  • objects in TLAB fulfill **data locality** and **NUMA** aligned
  • **no false sharing** between cores (data in cache line are just used by one CPU core)
  • increase pressure on young generation and thus minor GC
    – other objects are promoted earlier to old generation
    – increase number of major GC

» a lot of **long-lived objects** slow your application even more
  • each time all live objects have to be traversed
  • compacting GC have to copy objects
    – breaks data locality
    – can imply false sharing between cores
Escape Analysis – Not All Objects Are Allocated

» **C2 compiler** perform escape analysis of new object *after inline of hot methods*
  
  • *NoEscape* – object does not escape method in which it is created
    - all its usages are inlined
    - never assigned to static or object field, just to local variables
    - at any point must be JIT-time determinable and not depending on any unpredictable control flow
    - if the object is an *array*, indexing into it must be JIT-time constant
  
  • *ArgEscape* – object is passed as, or referenced from, an argument to a method but does not escape the current thread
  
  • *GlobalEscape* – object is accessed by different method and thread

» *NoEscape* objects are **not allocated at all** but JIT does scalar replacement
  
  • object deconstructed into its constituent fields (stack allocated)
  • disappear automatically after stack frame pop (return from the method)
  • no GC impact at all + do not need track references (write comp. barrier)

» *ArgEscape* objects are allocated on the heap but all monitors are eliminated
public static class Vector {
    private final int a1, a2;

    public Vector(int a1, int a2) {
        this.a1 = a1;
        this.a2 = a2;
    }

    public Vector add(Vector v) {
        return new Vector(a1+v.getA1(), a2+v.getA2());
    }

    public int mul(Vector v) {
        return v.getA1() * a1 + v.getA2() * a2;
    }

    public int getA1() {
        return a1;
    }

    public int getA2() {
        return a2;
    }
}

public int compute(int val) {
    Vector v = new Vector(val+1, val*2);
    synchronized (v) {
        return v.add(v).mul(v);
    }
}
Escape Analysis Example

C2 compilation with inline:

```assembly
sub $0x18,%rsp
mov %rbp,0x10(%rsp)
mov %edx,%r11d
add %edx,%r11d
mov %edx,%r10d
shl %r10d
mov %r10d,%r8d
add %r10d,%r8d
imul %r10d,%r8d
add $0x2,%r11d
inc %edx
imul %r11d,%edx
mov %edx,%eax
add %r8d,%eax
add $0x10,%rsp
pop %rbp
test %eax,-0x21742ec(%rip)
retq
```

```java
public int compute(int val) {
    Vector v = new Vector(val+1, val*2);
    synchronized (v) {
        return v.add(v).mul(v);
    }
}
```

no allocation at all, no synchronization all done out of stack in registers only
Bloom Filter

- **bloom filter** function
  - add a new object to the set
  - test whether a given object is a member of the set
  - **no deletion** is possible

- **strong memory reduction** (~9.6 bits per element) compared to other collections
  - compensated by **small false positive** rate (~1%)
  - guaranteed **no false negative**
  - not storing object itself (where *all standard collections must store objects*)
  - always **constant complexity** (even for collisions)

- **constant** add and test/query **complexity**

- very useful in **big data** processing and other applications
  - used to test that the **object is certainly not present**
  - e.g. reduce a lot of I/O operations reading full collections in a particular file where bloom filters are kept in RAM or read quickly from disk
Bloom Filter

» use bit array with a $m$ bits
» use $k$ independent hash functions
» **add** operation – $O(k)$

query operation
Bloom Filter

» number of bits in the filter

\[ \text{ceil} \left( \frac{n \cdot \ln(p)}{\ln\left( \frac{1}{2^{\ln(2)}} \right)} \right) \]

» number of hash functions

\[ \text{round} \left( \frac{\ln(2) \cdot m}{n} \right) \]

» example – store 1 million of Strings with total size 25 MB

• Set<String> requires more than 25 MB

• Bloom Filter with FP rate 1% requires 1.13 MB and 7 hash functions
  – more than 22 times smaller and in 99% cases query is TP
Extensions of Bloom Filter

» counting bloom filter
  • support delete operation

  • each position in filter is buckets (e.g. 3 bits) working as counter
    - add – increment
    - delete – decrement
    - query – test non-zero

  • bucket overflow problem
    - no more increments
    - increasing FN errors

» bitwise bloom filter
  • multiple counting filters to address issues above
Reference Objects

- mortem hooks more **flexible** than finalization
- **reference types** (ordered from strongest one):
  - {strong reference}
  - soft reference – optional reference queue
  - weak reference – optional reference queue
  - {final reference} – mandatory reference queue
  - phantom references – mandatory reference queue
- can enqueue the reference object on a designated **reference queue** when GC finds its referent to be less reachable, referent is released
- references are enqueued **only if you have strong reference to REFERENCE**
- GC has to run to pass them to **Reference Handler** to enqueue them into **reference queue**
- Reference is **another instance** on the heap – 48 Bytes for standard OOP, 64-bit JVM

```
ref = new WeakReference(foo, rq);
```
» **strongly reachable** – from GC roots without any Reference
» **softly reachable** – not strongly, but can be reached via soft reference
» **weakly reachable** – not strongly, not softly, but can be reached via weak reference; clear referent link and become eligible for finalization
» **eligible for finalization** – not strongly, not softly, not weakly and have non-trivial finalize method
» **phantom reachable** – not strongly, not softly, not weakly, already finalized or no finalize method, but can be reached via phantom reference
» **unreachable** – none of above; eligible for reclamation
Weak Reference

» pre-finalization processing

» usage:
  • do not retain this object because of this reference
  • don't own target, e.g. listeners
  • canonicalizing map – e.g. ObjectOutputStream
  • implement flexible version of finalization:
    – prioritize
    – decide when to run finalization

» get() returns
  • referent if not cleared
  • null, otherwise

» referent is cleared by GC (cleared when passed to Reference Handler) and can be reclaimed

» need copy referent to strong reference and check that it is not null before using it

» WeakHashMap<K,V> - uses weak keys; cleanup during all standard operations
Weak Reference – External Resource Clean-up

» clean-up approach for ReferenceQueue\(<T>\)
  • own dedicated thread

```java
ReferenceQueue<Image3> refQueue =
NativeImage3.referenceQueue();
while (true) {
    NativeImage3 nativeImg =
        (NativeImage3) refQueue.remove();
    nativeImg.dispose();
}
```

• clean-up before creation of new objects
  – limited clean-up processing to mitigate long processing
  – use poll() – non-blocking fetch of first
Custom Finalizer Example

```java
public abstract class CustomFinalizer extends WeakReference<Object> {
    private static final ReferenceQueue<Object> referenceQueue = new ReferenceQueue<>();
    private static final CustomFinalizer circularEnd = new CustomFinalizer() {...};

    private CustomFinalizer next, prev;

    public CustomFinalizer(Object referent) {...}

    private CustomFinalizer() {...}

    private void executeCustomFinalize() {...}

    public abstract void customFinalize();

    static {
        Thread cleanupThread = new Thread() -> {
            for (; ; ) {
                try {
                    CustomFinalizer toCleanup = (CustomFinalizer) referenceQueue.remove();
                    toCleanup.executeCustomFinalize();
                } catch (InterruptedException e) {
                }
            }, name: "Custom finalizer";
        cleanupThread.setDaemon(true);
        cleanupThread.start();
    }
```
Custom Finalizer Example

```java
public CustomFinalizer(Object referent) {
    super(referent, referenceQueue);
    synchronized (circularEnd) {
        next = circularEnd.next;
        circularEnd.next.prev = this;
        prev = circularEnd;
        circularEnd.next = this;
    }
}

private void executeCustomFinalize() {
    if (next == null) return;
    synchronized (circularEnd) {
        prev.next = next;
        next.prev = prev;
    }
    next = prev = null;
    customFinalize();
}
```

» usage example, beware of implicit this strong reference in instance context

```java
new CustomFinalizer(monitoredObjectForFinalization) {
    @Override
    public void customFinalize() {
        // custom finalization
    }
};
```
Soft Reference

» pre-finalization processing
» usage:
  • would like to keep referent, but can loose it
  • suitable for caches – create strong reference to data to keep them
    – objects with long initialization
    – frequently used information
» get() returns:
  • referent if not cleared; null, otherwise
  • updates timestamp of usage (can keep recently used longer)
  • all are cleared before OutOfMemoryError
  • reclaim only if there is “memory pressure” based on heap usage
    now – timestamp > (SoftRefLRUPolicyMSPerMB * amountOfFreeMemoryInMB)
    -XX:SoftRefLRUPolicyMSPerMB=N (default 1000)

» referent is cleared by GC (cleared when passed to Reference Handler) and can be reclaimed
Efficient Cache Example

efficient LRU tracking in combination with memory pressure for older ValueHolder

- "hard" refs.

External References

WeakHashMap

<table>
<thead>
<tr>
<th>key_1</th>
<th>key_2</th>
<th>key_3</th>
<th>key_4</th>
<th>key_5</th>
<th>key_6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value_2</td>
<td>Value_3</td>
<td>Value_4</td>
<td>Value_5</td>
<td>Value_6</td>
</tr>
</tbody>
</table>

SoftCache

ValueHolder_1  ValueHolder_2  ValueHolder_3  ValueHolder_4  ValueHolder_5  ValueHolder_6  ValueHolder_7  ValueHolder_8
Final Reference – Object with Non-Trivial Finalize

» finalize hook cannot be used directly (package limited)
» instance allocation of object with **non-trivial finalize method**
  • slower allocation than standard objects
  • run time call of `Finalizer.register` with possible global safe point
    – not inlined, all references saved in stack with OopMap
  • allocates `FinalReference` instance and do synchronized tracking

» referent is not cleared and reclaimed before finalization
  • **all referenced objects** cannot be reclaimed as well

» one `Finalizer` thread only for all Final references
  • call `finalize` method and `clear` referent
    – **issue** when finalize creates **strong reference again**
    – no priority control between multiple finalize methods
    – long running finalize delays all other finalization
  • daemon thread and JVM can terminate before finalization of all

» finalized objects can be reclaimed during **subsequent GC cycle**
Phantom Reference

» post-finalization processing

» usage:
  • notifies that the object is not used – before its reclamation
  • possible replacement for finalization (but prefer WeakReference)
  • can keep some data after the object becomes finalized

» get() returns:
  • null always
  • referent can be read using reflection
    • avoid making strong reference again

» have to specify reference queue for constructor

» referent is not cleared and reclaimed until all phantom references are not become unreachable or manually cleared using method clear()
  » all referenced objects cannot be reclaimed as well
Reference Object

» only **one GC cycle** needed to reclaim *referent object* if there is only soft references or weak references to the same object

» **multiple GC cycles** needed for *referent objects* with multiple reference types or have at least one final or phantom reference

**Reference Handler** thread enqueue respective Reference(s) to their ReferenceQueue(s) if there are defined some

- SoftReferences
- WeakReferences
- FinalReferences

**Finalizer** thread executed non-trivial finalize if referent object was weakly and/or softly reachable and/or has finalize method

(reclaimed)

**PhantomReferences**

- referent object was phantomly reachable

**custom thread** called clear

**GC** object reclaimed if not in the first GC

**Time**
Performance Cost for References

» **creation** cost
  • allocation instance
  • synchronization with tracking of Reference (strong references)

» **garbage collection** cost (-XX:+PrintReferenceGC –XX:+PrintGCDetails)
  • tracking live not follow referents
  • construct list of live References each GC cycle
    - discovered field in Reference
  • per-reference overhead regardless referent is collected or not
    - softly, weakly + finalizable, phantomly
  • Reference Object itself are subject for garbage collection

» **enqueue** cost
  • reference handler enqueue with synchronization

» **reference queue processing** cost
  • synchronized queue consumption
Reachability of Object
Reachability of Object