The Shape of an Object
Optimizing for space and time in IBM’s J9 Java VM

Peter Burka
peter.burka@twosigma.com
December 11, 2014
J9

- IBM’s Java virtual machine
- Cleanroom implementation
- Originally an embedded JVM
- Used by thousands of IBM customers

*Disclaimer: I’m not speaking on behalf of IBM or Two Sigma*
Objects

- Object = a value stored in memory
- Design questions:
  - Interoperability (ABI)
  - Identity and mutability (==, =)
  - Dynamic vs. static shape
  - Size / Speed
  - Subtyping / multiple inheritance
  - Reflection
  - Architectural considerations (e.g. alignment)
Some options

• Dictionaries (*Python*, *Javascript*)
  – Completely dynamic

• 2-Tuples (*Lisp*)
  – Uniformly size

• Contiguous structs (*C, C++, Fortran, Java*)
  – Space and time efficient
  – Shape known early
A conjecture...

http://xkcd.com/1310/
A conjecture...

• All objects are strings
A conjecture...

- All objects are strings
- And the remainder are arrays.
A Java string

_class (j.l.String)
flags
_hashcode
_monitor
size
offset
hashcode
(padding)

(48 bytes)

_class ([C)
flags
_hashcode
_monitor
_length
'H'
'e'
'l'
'o'
(pad)

(40 bytes)

header (24 bytes)
header (28 bytes)

= 1 byte
Can we make these smaller?
Should *not* we make these smaller?

- RAM is cheap
Should we make this smaller?

- Address space is cheap (64-bits)
Should we make these smaller?

- Cache is not cheap
  - No significant size increase in 10 years
  - (when measured per thread)

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gallatin ('04)</td>
<td>8K</td>
<td>512K</td>
<td>4.0M</td>
<td>2G</td>
</tr>
<tr>
<td>Haswell ('14)</td>
<td>32K</td>
<td>128K</td>
<td>1.5M</td>
<td>128G</td>
</tr>
</tbody>
</table>
A Java string

_class (j.l.String)
flags
_hashcode
_monitor
size
offset
hashcode
(padding)
data

(48 bytes)
Step 1: compress pointers

• Use 32-bit pointers for
  – Class pointer
  – Monitor pointer
  – Object pointers
Compress pointers

• Limited to 4GB space?
  – Classes, monitors & objects can each have own space
  – Exploit alignment

<table>
<thead>
<tr>
<th>Minimum alignment</th>
<th>Maximum heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes</td>
<td>16 GB</td>
</tr>
<tr>
<td>8 bytes</td>
<td>32 GB</td>
</tr>
<tr>
<td>16 bytes</td>
<td>64 GB</td>
</tr>
<tr>
<td>32 bytes?</td>
<td>128 GB</td>
</tr>
</tbody>
</table>
def decompress(comp_ptr) {
    if (comp_ptr == 0)
        return NULL;
    else
        return base + (comp_ptr << scale);
}
def decompress(comp_ptr) {
    #if (base == 0)
        return comp_ptr << bits;
    #else
        if (comp_ptr == 0)
            return NULL;
        else
            return base + (comp_ptr << scale);
    #endif
}

Compress pointers
Before

_class (j.l.String)
_flags
_hashcode
_monitor
size
offset
hashcode
(padding)
data

48 bytes

_class ([C]
_flags
_hashcode
_monitor
_length
‘H’
‘e’
‘l’
‘l’
‘o’

40 bytes
After

_class
_flags
_hashcode
_monitor
size
offset
hashcode
data
(32 bytes)

header

_class
_flags
_hashcode
_monitor
_length
‘H’
‘e’
‘l’
‘l’
‘o’
(32 bytes)
Step 2: Get rid of hashcode

• All objects have ‘identity’ and identity hash:
  – System.identityHashCode()

• Stored in header, because objects move

• In practice, < 2% hashed

• Can we store it lazily?
Hashcode algorithm

• First time object is hashed:
  – Generate hash based on address
  – Record ‘has been hashed’

• If object moves:
  – Store hash at end of object
  – Record ‘has been moved’

• Subsequent hashes:
  – Determine if hash is stored or not
  – Read hash, or generate hash based on address
Before

```
_class
_flags
_hashcode
_monitor
size
offset
hashcode
data

(32 bytes)
```
After

[diagram showing changes in data structure with "header" label]
Step 3: Get rid of flags slot

- What’s in the flags?

<table>
<thead>
<tr>
<th>What’s in the flags</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash state</td>
<td>2 bits</td>
</tr>
<tr>
<td>GC info (e.g. age, remembered)</td>
<td>4-12 bits</td>
</tr>
<tr>
<td>Object type (e.g. array)</td>
<td>3 bits</td>
</tr>
<tr>
<td>Misc. other stuff</td>
<td>Expands to fill available space</td>
</tr>
</tbody>
</table>
Hiding flags

• Infer some flags from class
  – One extra indirect

• Hide the rest in the class pointer
  – Classes must be 256-byte aligned
  – One extra mask instruction
Before

- `_class`
- `_flags`
- `_monitor`
- `size`
- `offset`
- `hashcode`
- `data`
- `(padding)`

(32 bytes)

`header`

- `_class`
- `_flags`
- `_monitor`
- `_length`

- `'H'`
- `'e'`
- `'l'`
- `'l'`
- `'o'`

`(padding)`

(32 bytes)
After

- _cls_flg
- _monitor
- size
- offset
- hashcode
- data

(header 24 bytes)

(24 bytes)

- _cls_flg
- _monitor
- _length
- ‘H’
- ‘e’
- ‘l’
- ‘l’
- ‘o’
- (pad)

(24 bytes)
Step 4: Get rid of monitor slot

• All objects have a monitor
  – synchronized, wait, notify

• Very few objects use the monitor
  – Strings are immutable
  – Arrays are usually wrapped in other objects

• Use monitor slot for some objects
  – “lock nursery” (i.e. hash table) for others
Can we guess where it’s needed?

• Maybe
• Static analysis can help
• But fails in some common cases
  – Object _lock = new Object();
• J9 was conservative
  – Removed monitor from a small set of classes
    • String, Number, Boolean, ...
Experimental solution

• Grow monitors on demand
  – Moving objects is expensive
    • Must update all incoming pointers (usually)
  – Combine with a small nursery to amortize cost
Before

header

<table>
<thead>
<tr>
<th>_cls_flg</th>
<th>_monitor</th>
<th>size</th>
<th>offset</th>
<th>hashcode</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(24 bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

header

<table>
<thead>
<tr>
<th>_cls_flg</th>
<th>_monitor</th>
<th>_length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>'H'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'e'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'l'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'l'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'o'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(pad)</td>
</tr>
</tbody>
</table>

(24 bytes)
After

<table>
<thead>
<tr>
<th>_cls_flg</th>
<th>header</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td></td>
</tr>
<tr>
<td>hashcode</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td></td>
</tr>
<tr>
<td>(padding)</td>
<td>(24 bytes)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>_cls_flg</th>
<th>header</th>
</tr>
</thead>
<tbody>
<tr>
<td>_length</td>
<td></td>
</tr>
<tr>
<td>‘H’</td>
<td></td>
</tr>
<tr>
<td>‘e’</td>
<td></td>
</tr>
<tr>
<td>‘l’</td>
<td></td>
</tr>
<tr>
<td>‘l’</td>
<td></td>
</tr>
<tr>
<td>‘o’</td>
<td></td>
</tr>
<tr>
<td>(padding)</td>
<td>(24 bytes)</td>
</tr>
</tbody>
</table>
Some more ideas

Sad truth: Most "mad scientists" are actually just mad engineers

http://www.neatorama.com/2009/01/01/mad-scientists-are-actually-just-mad-engineers/
Idea 1: Hot and cold fields

• Large objects may have rarely used fields
• Use runtime profiling to identify hot fields
• Split hot from cold for cache efficiency
• Caveat:
  – changing object layout at runtime is costly
Hot & cold layouts

- Sort hot fields to front

- Bidirectional objects

- Linked objects
Hot & cold layouts

- Sort hot fields to front

- Bidirectional objects

- Linked objects
Idea 2: Headerless objects?

- Can we delete the class pointer?
- Infer class from pointer
  - \( \text{class} = \text{object} \& \ 0xFFFFFFFFF000000 \)
Headerless objects (cont.)

• Wastes memory
  – But RAM and address space are cheap

• Organizes objects by class
  – Splits up related objects
  – Could be bad for cache
Lessons

- Object shape affects performance
- Language affects object shape
  - Dynamic vs. static
  - Hash codes
  - Synchronization
  - Compatibility
Further reading

• Bacon, Fink and Grove. “Space- and Time-Efficient Implementation of the Java Object Model”, 2002

• Adl-Tabatabai, et al. “Improving 64-Bit Java IPF Performance by Compressing Heap References”, 2004

Idea 3: Objlets

• Break large objects up into trees of smaller objects
• Simplifies allocation
• Avoids defragmentation
• Enables realtime allocation guarantees