Efficient Runtime Tracking of Allocation Sites in Java

Rei Odaira, Kazunori Ogata, Kiyokuni Kawachiya, Tamiya Onodera, Toshio Nakatani

IBM Research - Tokyo
Why Do You Need Allocation Site Information?

How to fix a memory leak after two weeks of execution?

>` java com.example.Server started
      ......running for one week
      ......running for two weeks
      server crashed due to java.lang.OutOfMemoryError!
>`

![Java Memory Profiler - Main](image)

- **Heap**: 96.75921 MB
- **Used**: 60.95346 MB
- **Filtered**: 60.95346 MB

Garbage collection completed: 0 objects moved, 160277 objects freed in 1.478211 seconds
Allocation Site Tracker Helps!

Tracker tells you where each object was allocated.

Bytes       Class  Allocation site
           (Method name and bytecode index)
900,278,800  String  com.example.Property.putProperty()#191
98,148,020   LinkedList  com.example.DataTable.putInteger()#187
20,352,384   String  com.example.Property.prepare()#35
......       .....                ......

void putProperty(Element elem) {
    ......  
    String attribute = new String(elem.getString());
    ......  
}

Allocation site is a good starting point for fixing the leak.

Also, optimizations in JVM can benefit from the tracker.
Tracker Should Be Always Enabled.

- For fixing memory leaks ...

```java
> java com.example.Server started
......running for one week
......running for two weeks
server crashed due to java.lang.OutOfMemoryError!
```

```java
> java -enableTracker com.example.Server started
......running for one week
......running for two weeks
```

😢 should have always enabled the tracker.

- And also for JVM optimizations ....
Challenge: Performance Overhead

- Adding allocation site information to each object hits performance [Hauswirth et al., 2004].
  - Reducing effective CPU cache size, increasing GC frequency and overhead, etc.

IBM Java 6 SR3 64-bit compressed pointer / Linux 2.6.18 / 8x x86-64 Intel Xeon 1.86GHz

Not good for production environments
Minimal-Overhead Allocation Site Trackers

Never increase per-object space.

- **Allocation-Site-as-a-Hash-code (ASH) Tracker**
  - Performance overhead: ~0% on average, 1.4% at maximum.
  - Some JVMs do not always have a hash code field.

- **Allocation-Site-via-a-Class-pointer (ASC) Tracker**
  - Performance overhead: 1% on average, 2% at maximum.
  - Almost all JVMs have a class pointer field.
Outline

- Introduction
- ASH Tracker
- ASC Tracker
- Experiments
- Applications of ASH/ASC Trackers
- Conclusions
Allocation-Site-as-a-Hash-code (ASH) Tracker

- Embed an allocation-site ID into a hash code field.
  - Embed at allocation time.
- ID is a unique integer of the site.
  - Assigned by an interpreter or a JIT compiler.

**Original layout**

<table>
<thead>
<tr>
<th>Class pointer</th>
<th>Hash code</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Instance fields |

**w/ ASH Tracker**

<table>
<thead>
<tr>
<th>Class pointer</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Instance fields |

Object-specific random value, used as a hash table index, for example.

Allocation site ID
How to Deal with Hash Code Collisions?

- Hash code values should be as distinct as possible from all others [Java API Spec].
  - Collisions slow down hash-table access, for example.
- How about appending a site ID field when hash code is first referred to?
  😞 Some programs often refer to hash code.

![Diagram](image.png)

Allocation time: Site ID

First time hash code is referred to: Site ID, Hash code
Collision Avoidance without Increasing Space

- **Our solution:**
  - Embed a shorter *dynamic* ID and a random value.

- **Hash code value** = dynamic ID + random value
  - Random value helps avoid collisions.

### Diagram:
- **Allocation time**
  - Object X
    - Static ID: 1111010111
    - Dynamic ID
    - Hash code value

- **First time hash code is referred to.**
  - Object X
    - Dynamic ID: 0001100101
    - Object-specific random value

- **Other objects are not affected.**
  - Object Y
    - Static ID: 1111010111
    - Dynamic ID
    - Hash code value
How to Deal with Hash Code Collisions? (2\textsuperscript{nd} Round)

- All objects allocated at the same site have the same high-order bits in their hash code values.

Need a long random-value field to avoid collisions.

Hash code field: \textbf{Dynamic ID} Random

Need a long ID field to track allocation sites accurately.
Variable-Length Dynamic ID

- Shorter IDs for hot allocation sites.
  - Allocate many objects.
    → Need a long random value.
  - Not so many hot allocation sites in a program.
    → Short site IDs suffice.

- Longer IDs for cold allocation sites.
  - Allocate few objects.
    → Short random values suffice.
  - Many cold allocations sites in a program.
    → Need long site IDs.

```java
for (i = 0; i < BIG_NUMBER; i++) {
    obj = new Object();
    use(obj.hashCode());
}
```

```java
if (error1) {
    obj = new Object();
    use(obj.hashCode());
}
```

```java
if (error2) {
    obj = new Object();
    use(obj.hashCode());
}
```
Dynamic Shrinking of ID Field

- It is not known in advance ...
  - How many objects will be allocated at each site; or
  - How many of their hash code values will be referred to.
- Our solution: Make the IDs of a site shorter and shorter ...
  - As more and more hash code values are referred to.

Hash code of objects allocated at putProperty()#191

1) Assign a long dynamic ID at first.

2) Occasionally, a random value becomes all zero.

3) Assign a new shorter ID. Maintain a mapping table.
Outline

- Introduction
- ASH Tracker
- ASC Tracker
- Experiments
- Applications of ASH/ASC Trackers
- Conclusions
What If There Is No Hash Code Field?

- Some JVMs do not always have a hash code field.
- Almost all JVMs have a class pointer field.
  - To access class meta data.

```
Class pointer
  Flags
  Instance fields
```

```
Objects allocated at putProperty()#191
```

```
Class pointer
  Flags
  Instance fields
```

```
Virtual function table
Instance size
Other class meta data
```

```
Class pointer
  Flags
  Instance fields
```

```
Objects allocated at prepare()#35
```

```
Class pointer
  Flags
  Instance fields
```

```
Class pointer
  Flags
  Instance fields
```

```
Class pointer
  Flags
  Instance fields
```

```
Class pointer
  Flags
  Instance fields
```

Allocation-Site-via-a-Class-pointer (ASC) Tracker

- Replace the class pointer with a pointer to its allocation site structure.
  - This is possible because each allocation site always allocates objects of the same class.

```
Alloc site ptr
Alloc site ptr
Alloc site ptr
Alloc site ptr

Class pointer
putProperty() #191

Virtual function table
Instance size
Other class meta data

Class pointer
prepare() #35

Alloc site ptr
Alloc site ptr
Alloc site ptr
Objects allocated at
putProperty() #191
Objects allocated at
prepare() #35
```
Mitigating Indirection Overhead (1)

- Duplicate frequently-accessed constant class fields.
- Need to choose carefully which fields to duplicate.
  - Not to increase cache misses.
  - Not to increase space overhead.

```
<table>
<thead>
<tr>
<th>Alloc site ptr</th>
<th>Class pointer</th>
<th>Class pointer</th>
<th>Alloc site ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instance size</td>
<td>Instance size</td>
<td></td>
</tr>
<tr>
<td>putProperty()</td>
<td></td>
<td>prepare()</td>
<td></td>
</tr>
<tr>
<td>#191</td>
<td></td>
<td>#35</td>
<td></td>
</tr>
<tr>
<td>Virtual</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>function table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instance size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other class</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>meta data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alloc site ptr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Profiling-based devirtualization by a JIT compiler.

```c
if (object->class_ptr != HOT_CLASS)
goto SlowPath;
/* Inlined method, etc. */
```

Another load needed by ASC Tracker.

```c
if (object->alloc_site->class_ptr != HOT_CLASS)
goto SlowPath;
/* Inlined method, etc. */
```

Emit an allocation-site equality check where possible.

```c
if (object->alloc_site != HOT_ALLOCATION_SITE)
goto SlowPath;
/* Inlined method, etc. */
```
Outline

- Introduction
- ASH Tracker
- ASC Tracker
- Experiments
- Applications of ASH/ASC Trackers
- Conclusions
Evaluation

- Environment
  - 64-bit IBM J9/TR JVM 1.6.0 SR3
    - 3-word header (1 word = 32 bits)
    - Generational GC (copying young + mark-and-sweep old GC)
  - 4x minimum Java heap
  - 8-core 1.8GHz Intel Xeon
  - Linux 2.6.18

- Benchmarks
  - SPECjvm2008, DaCapo, and SPECjbb2005

- ASH Tracker
  - Uses a 15-bit hash code field.
  - Shrinks a dynamic ID field from 11 bits to 2 bits.

- ASC Tracker
  - Duplicates an instance-size field and an array-element-class field.
Performance Overhead

- **ASH Tracker**: on average ~0% and at most 1.4% overhead.
  - Up to 1.73x hash code collisions compared with the baseline.
- **ASC Tracker**: on average 1.0% and at most 2.0% overhead.
Performance Overhead

- **ASH Tracker**: on average \(\sim0\%\) and at most 1.4\% overhead.
  - Up to 1.73x hash code collisions compared with the baseline.
- **ASC Tracker**: on average 1.0\% and at most 2.0\% overhead.

Serial and pmd refer to the hash code of:
- more than 20\% of allocated objects, and
- 5-11\% of average live objects.
Outline

- Introduction
- ASH Tracker
- ASC Tracker
- Experiments
- Applications of ASH/ASC Trackers
- Conclusions
Minimal-Overhead Memory Leak Detector

- Scan the entire Java heap at each global GC time.
  - Count the numbers of live objects for each allocation site.
  - Save the numbers in per-allocation-site histories.
- (Inform users of possible memory leaks.)

### Java heap

```
obj  obj  obj  obj  obj  obj  obj
```

### Per-allocation-site counters

- **Property.putProperty()**
  - history: 6, 5, 4, ...
  - 191
- **DataTable.putInteger()**
  - history: 3, 3, 3, ...
  - 187
- **Property.prepare()**
  - history: 2, 1, 1, ...
  - 35
Performance Overhead of the Leak Detector

- ASH Tracker: on average 0.3% and at most 1.7% overhead
- ASC Tracker: on average 1.1% and at most 2.4% overhead
Allocation-Site-Based Object Pretenuring for Generational GC

Refer to our paper for more details.
- 4x min Java heap: at maximum 11% speed-up
- 2x min Java heap: at maximum 15% speed-up
Related Work

- **Bit-Encoding Leak Location (Bell)**
  [Bond, et al., ASPLOS 2006]
  - Probabilistic allocation site tracker
  - Requires a sufficient number of samples.
  - Cannot identify the allocation site of each object.
    → Not suitable for JVM optimizations.

- **Techniques for object header compression**
  [Bacon, et al., ECOOP 2002]
  - Remove the hash code field.
    → Use ASC Tracker.
  - Steal several bits of the class pointer.
    → Steal those bits of the allocation site pointer.
Conclusion

Minimal-overhead allocation site trackers

- **ASH Tracker**
  - Embeds an allocation site ID into the hash code field.
  - Performance overhead: ~0% on average, 1.4% at maximum.

- **ASC Tracker**
  - Makes the class pointer field point to an allocation site structure.
  - Performance overhead: 1% on average, 2% at maximum.

- Useful for both reliability and optimization.
  - Reliability: minimal-overhead memory leak detector.
  - Optimization: allocation-site-based object pretenuring.
Thank you!

- Questions?
Back-up
Space Overhead Compared with Physical Memory Usage

- ASH Tracker: on average 0.4% overhead
- ASC Tracker: on average 0.2% overhead
Using ASH Tracker for Object Pretenuring

- Copy likely-to-be-long-lived objects directly to a “tenured” space.
  - Not copying such objects multiple times within a young space.

1. Online profiling
   - Compute the ratio of tenured objects ($\#t/\#s$) for each allocation site at young GC time.

2. Pretenuring
   - Enable pretenuring for an allocation site if its ratio exceeds a certain threshold.