Efficient and Viable Handling of Large Object Traces

Philipp Lengauer
Verena Bitto
Hanspeter Mössenböck

2016-03-15
Recap - AntTracks

... we would know all there is to know about every object?

```java
public void foo() {
    String[] object = { "Tracing", "rocks" };
}
```

... then we could reproduce the entire heap for every point in time and do offline analysis!
Allocations

Addresses of objects that are allocated into a TLAB are computable offline!

\[
addr(o_n) = \begin{cases} 
  addr(TLAB(o_n)) & \text{if } n = 1 \\
  addr(o_{n-1}) + \text{size}(o_{n-1}) & \text{else}
\end{cases}
\]
Minor GCs

New addressed of moved objects are computable offline!

Move by GC-Thread 1

Move by GC-Thread 2
Major GCs

Claim: objects live and die in groups due to their sequential allocation
Optimized Events

**Optimized allocation event**

- **address** → previous events + TLAB information
- → 4 bytes per allocation
- → **computable at compile-time (JIT)**

**Optimized move event**

- ~ 312 objects per event
- (3.65Kb -> 12b)

**Region move event**

- event type
- from address
- to address
- object count
Digging Our Own Grave

![Bar chart showing benchmarks for different applications.]
Trace Size vs Disk Limit

![Graph showing trace size vs time with a linear relationship and a disk limit threshold.]
Compression

+ Trace reduced to **21.6%**

- Overhead increased by **21.9%**

~ Trace reduced to **89.7%**

+ Overhead increased by **2.3%**
Compression

Same problem, just a little bit later ...
Similar Problem: Objects vs Heap Size

![Graph showing RAM usage over time with GC events](image)

- RAM
- Heap limit
- Time
- GC events
Rethink Trace Size vs Disk Limit

What parts of trace to remove?

divide trace into chunks

delete oldest chunk

trace size

Disk limit

time
Split trace into n files, overwrite oldest file first.

Every trace file may be eventually be the oldest.

What is the state of the heap at the beginning of the oldest file?

addresses

sizes

Need-to-know
Synchronization Points

Use GCs as synchronization points

What if no GC occurs at the right point?
Trace Size Deviation

Trigger "**Emergency GCs**" after max deviation is reached.

MaxSize=16GB  Deviation=25%

file count = 100% / Deviation

target file size = MaxSize * Deviation

max file size = file size + file size * Deviation
Synchronization GC

Optimized move event

- event type
- from address

Optimized move sync event

- event type
- allocation site
- type
- from address

Replace all move events with move sync events.
Overhead
Quality
Restoring Allocation Sites

- Must generate hash code eagerly for every (!) object.
- Reduces entropy of the hash to **0.0015%**.
void main() {
    Set<Object> set = new HashSet<>();
    for(int i = 0; i < 1_000_000_000; i++) {
        set.put(create());
    }
    for(int i = 0; i < 1_000_000_000; i++) {
        set.contains(create());
    }
}

Object create() {
    // all objects have same allocation site
    return new Object();
}

→ run time +2191%
Overhead with Saving Allocation Sites

![Bar chart showing overhead with saving allocation sites for various benchmarks.](chart.png)
Reducing Hash Code Generation

Assumption: classes overwriting `hashCode()` will *most likely* never use the identity hash!
Summary

On-the-fly compression

Trace rotation

Efficient and Viable Handling of Large Object Traces

Philipp Lengauer\textsuperscript{1} Verena Bitto\textsuperscript{2} Hanspeter Mössenböck\textsuperscript{1}

\textsuperscript{1}Institute for System Software
Johannes Kepler University Linz, Austria
philipp.lengauer@jku.at

\textsuperscript{2}Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria
verena.bitto@jku.at

ABSTRACT
Understanding and tracking down memory-related performance problems is a tedious task, especially when it involves automatically managed memory, i.e., garbage collection. A multitude of monitoring tools show the substantial need of developers to deal with these problems efficiently. Unfortunately, state-of-the-art tools either compute an infeasible number of memory objects or are too slow to provide timely feedback.

1. INTRODUCTION
The widespread use of programming languages with automatic memory management has stressed the need for memory profiling tools. Although managed memory relieves programmers from the error-prone task of freeing memory manually, it comes at the cost of performance problems that are hard to track down. When an allocation fails due to a full