Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications

Rodrigo Bruno(1)*, Duarte Patrício(2), José Simão(2), Luís Veiga(2), Paulo Ferreira(2,3)

EuroSys 2019 @ Dresden, 25-28 March

* Work done while at (2)
Big Data Application Stack

Big Data Application

Big Data Platform
- Cassandra
- Hadoop
- Apache Spark
- MongoDB

Managed Runtime
- OpenJDK
- IBM
- Azul Systems
Examples of Latency Sensitive apps
- Banking applications and services
- Context-Aware Ad services
- Games
- …
Big Data Application Stack

Examples of Latency Sensitive apps
- Banking applications and services
- Context-Aware Ad services
- Games
- ...

SLAs can be easily compromised
Examples of Latency Sensitive apps
- Banking applications and services
- Context-Aware Ad services
- Games
- ...

SLAs can be easily compromised

Slow Platform Response Time
Examples of Latency Sensitive apps
- Banking applications and services
- Context-Aware Ad services
- Games
- ...

SLAs can be easily compromised

Slow Platform Response Time

Application Threads are being hold
GC is known to have difficulties scaling to high number of cores and memory, mainly w.r.t. Latencies:

- [ACM CSUR 2018]
- [DSN 2018]
- [ISMM 2017]
- [ISMM 2015]
- [ASPLOS 2013]
- ...

**GC-induced Application Latency**
OpenJDK HotSpot Generational GCs (PS, CMS, G1)

- Two generations:
  - Young and Old
- Surviving objects are copied to
  - Survivor spaces and then to
  - the Old generation.
OpenJDK HotSpot Generational GCs (PS, CMS, G1)

- Two generations:
  - Young and Old
- Surviving objects are copied to
  - Survivor spaces and then to
  - the Old generation.
OpenJDK HotSpot Generational GCs (PS, CMS, G1)

- Two generations:
  - Young and Old
- Surviving objects are copied to
  - Survivor spaces and then to the Old generation.
OpenJDK HotSpot Generational GCs (PS, CMS, G1)

- Two generations: 
  - Young and Old
- Surviving objects are copied to 
  - Survivor spaces and then to 
  - the Old generation.
OpenJDK HotSpot Generational GCs (PS, CMS, G1)

- Two generations:
  - Young and Old
- Surviving objects are copied to
  - Survivor spaces and then to
  - the Old generation.
OpenJDK HotSpot Generational GCs (PS, CMS, G1)

- Two generations: Young and Old
- Surviving objects are copied to Survivor spaces and then to the Old generation.
OpenJDK HotSpot Generational GCs (PS, CMS, G1)

- Two generations:
  - Young and Old
- Surviving objects are copied to
  - Survivor spaces and then to
  - the Old generation.
OpenJDK HotSpot Generational GCs (PS, CMS, G1)

- Two generations:
  - Young and Old
- Surviving objects are copied to
  - Survivor spaces and then to
  - the Old generation.
OpenJDK HotSpot Generational GCs (PS, CMS, G1)

- Allocated Objects: 32
  Number of copies: 9

Two generations:
- Young and Old

Surviving objects are copied to:
- Survivor spaces and then to
- the Old generation.
Big Data Application (simplification)

```java
public void runTask(enum TaskType tt) {
    // Allocates memory to hold Working Set
    WorkingItem[] buffer = new WorkingItem[WS_SIZE];

    // Loads Working Set
   DataProvider.load(tt, buffer);

    // Process Working Set
    Result r = DataProcessor.process(tt, buffer);

    // Pushes results from computation
    Output.push(r);
}
```

- 4 threads (one per core), running ‘runTask’ method in loop
- Each task consumes 500 MB of memory (Working Set size)
- Eden is 2GB in size
- Tasks can take different amounts of time to finish
Big Data Application in HotSpot GCs

Thread 1: Task A, Task B
Thread 2: Task C
Thread 3: Task B
Thread 4: Task D

GC

Time
Big Data Application in HotSpot GCs

Young generation is full and Thread 1 needs more memory to allocate WS for Task B.
Big Data Application in HotSpot GCs

Thread 1: Task A, Task B
Thread 2: Task C
Thread 3: Task B
Thread 4: Task D

WS not copied
WS copied once
WS copied twice

Copies 3 WS = 1500 MB!
Big Data Application in HotSpot GCs

Thread 1
- Task A
- Task B
- Task B

Thread 2
- Task C
- Task C

Thread 3
- Task B
- Task A
- Task A
- Task B

Thread 4
- Task D

WS not copied
WS copied once
WS copied twice

Diagram:
- Eden Space
- Survivor Spaces
- Young Generation
- Old Space
Old Generation
Big Data Application in HotSpot GCs

Copies 3 WS = 1500 MB!
Big Data Application in HotSpot GCs

Thread 1: Task A, Task B, Task B, Task A, Task B

Thread 2: Task C, Task C, Task A

Thread 3: Task B, Task A, Task A, Task B, Task C

Thread 4: Task D, Task B, Task B

WS not copied
WS copied once
WS copied twice
Big Data Application in HotSpot GCs

Copies 3 WS = 1500 MB!
Big Data Application in HotSpot GCs

Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Big Data Application in HotSpot GCs

Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 20GB/s (DDR4)
Big Data Application in HotSpot GCs

Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 20GB/s (DDR4)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~ 150 ms
Big Data Application in HotSpot GCs

Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 20GB/s (DDR4)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~ = 150 ms
8 Threads, Eden 4GB = copy 7 tasks (3500 MB) ~ = 350 ms
Big Data Application in HotSpot GCs

Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 20GB/s (DDR4)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~ = 150 ms
8 Threads, Eden 4GB = copy 7 tasks (3500 MB) ~ = 350 ms
16 Threads, Eden 8GB = copy 15 task (7500 MB) ~ = 750 ms
Big Data Application in HotSpot GCs

![Bar chart showing GC Copy Time (ms) vs Number of Threads]

- Number of Threads: 2, 4, 8, 16, 32
- GC Copy Time (ms): 50, 150, 350, 750, 1550

The chart illustrates the increase in GC Copy Time (ms) as the number of threads increases from 2 to 32.
Big Data Application in HotSpot GCs

GC bound by mem bandwidth
Big Data Application in HotSpot GCs

Goal: Reduce Application Pauses by reducing Object Copying
(no negative impact on throughput; no programmer effort)
How to Avoid en-masse Object Copying

- Attempt 1: Heap Resizing
  ✓ Increase Young generation size giving more time for objects to die;
  ! Does not solve the problem, eventually the Young gen will get full and objects will be copied.
How to Avoid en-masse Object Copying

- **Attempt 1: Heap Resizing**
  - ✓ Increase Young generation size giving more time for objects to die;
  - ! Does not solve the problem, eventually the Young gen will get full and objects will be copied.

- **Attempt 2: Reduce Task/Working Set size**
  - ✓ Reduces the amount of object copying since the WS is smaller;
  - ! Increases overhead as more tasks and coordination is necessary to process smaller tasks.
How to Avoid en-masse Object Copying

● **Attempt 1: Heap Resizing**
  ✓ Increase Young generation size giving more time for objects to die;
  ! Does not solve the problem, eventually the Young gen will get full and objects will be copied.

● **Attempt 2: Reduce Task/Working Set size**
  ✓ Reduces the amount of object copying since the WS is smaller;
  ! Increases overhead as more tasks and coordination is necessary to process smaller tasks.

● **Attempt 3: Reuse data objects (object pooling)**
  ✓ Avoids allocating new memory for future Tasks;
  ! Requires major rewriting of applications combined with very unnatural Java programming style.
How to Avoid en-masse Object Copying

- Attempt 4: Off-heap memory
  ✓ Reduces GC effort as data objects can reside in off-heap
  ! Objects describing data objects still reside in the GC-managed heap
  ! Requires manual memory management (defeats the purpose of running inside a managed heap).
How to Avoid en-masse Object Copying

● Attempt 4: Off-heap memory
  ✓ Reduces GC effort as data objects can reside in off-heap
  ! Objects describing data objects still reside in the GC-managed heap
  ! Requires manual memory management (defeats the purpose of running inside a managed heap).

● Attempt 5: Region-based/Scope-based memory allocation
  ✓ Limits object's reachability by scope/region;
  ! Does not allow objects to freely move between scopes. Bag-of-tasks only, no support for DB!
How to Avoid en-masse Object Copying

● **Attempt 4: Off-heap memory**
  ✓ Reduces GC effort as data objects can reside in off-heap
  ! Objects describing data objects still reside in the GC-managed heap
  ! Requires manual memory management (defeats the purpose of running inside a managed heap).

● **Attempt 5: Region-based/Scope-based memory allocation**
  ✓ Limits object's reachability by scope/region;
  ! Does not allow objects to freely move between scopes. Bag-of-tasks only, no support for DB!

● **Attempt 6: Completely Concurrent Collectors (C4, Shenandoah, ZGC)**
  ✓ Greatly reduced pause times
  ! High throughput overhead (~30% for Cassandra workloads)
How to Avoid en-masse Object Copying

Takeaway:

- Avoiding massive object copying is non-trivial!
- Existing solutions only alleviate the problem!
- Existing solutions might work in some scenarios but do not provide a general solution.
Proposed Solution

- **Solution:**
  - Allocate objects with similar lifetimes close to each other
    - Reducing memory fragmentation
    - Reducing object promotion
  - As a consequence, object copying is reduced!
Proposed Solution

- **Solution:**
  - Allocate objects with similar lifetimes close to each other
    - Reducing memory fragmentation
    - Reducing object promotion
  - As a consequence, object copying is reduced!

- **Hypothesis:**
  - Objects allocated through the same allocation context have similar lifetimes;
  - Allocation context is a tuple of:
    - Allocation site (line of bytecode)
    - Call graph state (stack state)
Big Data Application (simplification)

```java
public void runTask(enum TaskType tt) {
    // Allocates memory to hold Working Set
    WorkingItem[] buffer = new WorkingItem[WS_SIZE];

    // Loads Working Set
    DataProvider.load(tt, buffer);

    // Process Working Set
    Result r = DataProcessor.process(tt, buffer);

    // Pushes results from computation
    Output.push(r);
}
```
Solution - NG2C + ROLP
Solution - NG2C + ROLP

N-Generational GC (ISMM’17)

Runtime Object Lifetime Profiler
Runtime Object Lifetime Profiler (overview)

- Profiler needs to answer a single question
  - How long will objects allocated through a particular allocation context live?
Runtime Object Lifetime Profiler (workflow)

Method $M$

<table>
<thead>
<tr>
<th>BCI</th>
<th>Bytenode $bc_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$bc_i$</td>
<td>new</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$bc_N$</td>
<td>$bc_N$</td>
</tr>
</tbody>
</table>

Thread Stack $T$

| Frame $F_0$ |
| ... |
| Frame $F_N$ |
Runtime Object Lifetime Profiler (workflow)
Runtime Object Lifetime Profiler (workflow)
Runtime Object Lifetime Profiler (workflow)
Runtime Object Lifetime Profiler (workflow)
Runtime Object Lifetime Profiler (workflow)
Runtime Object Lifetime Profiler (conflicts)
Runtime Object Lifetime Profiler (conflicts)
Runtime Object Lifetime Profiler (conflicts)
Runtime Object Lifetime Profiler (context tracking)

- Method calls are wrapped with context tracking code (update thread stack state)
- Context tracking is very expensive
  - Only method calls that can resolve conflicts are profiled (next slide)
  - Jitted code can dynamically enable or disable profiling
Runtime Object Lifetime Profiler (context tracking)

- Method calls are wrapped with context tracking code (update thread stack state)
- Context tracking is very expensive
  - Only method calls that can resolve conflicts are profiled
  - Jitted code can dynamically enable or disable profiling
Runtime Object Lifetime Profiler (context tracking)

- Method calls are wrapped with context tracking code (update thread stack state)
- Context tracking is very expensive
  - Only method calls that can resolve conflicts are profiled
  - Jitted code can dynamically enable or disable profiling
Runtime Object Lifetime Profiler (implementation)

- ROLP is implemented on OpenJDK HotSpot 8
  - Industrial JVM

- ROLP is integrated with NG2C

- ROLP is meant to be running in production workloads
  - Several of implementation/performance optimizations
    - Avoid inlined methods
    - Properly Handling Exceptions
    - Properly Handling On-Stack Replacement
    - Reducing Profiling Overhead for very large applications
    - Shutdown survivor tracking code to reduce overhead
    - Improving the scalability of the Object Lifetime Distribution table
    - ...

58
Runtime Object Lifetime Profiler (evaluation)

- Evaluate ROLP’s performance compared to:
  - G1 - best solution in OpenJDK, current default GC (ISMM’04)
  - NG2C - multi-generational GC (ISMM’17) - requires programmer effort and knowledge
  - CMS - concurrent mark-sweep - throughput oriented

- Big Data Platforms & Workloads:
  - Cassandra (Key-Value Store)
    - YCSB: Write-Intensive (75% writes), Read-Write (50% writes), Read-Intensive (75% reads)
  - Lucene (In-Memory Indexing Tool)
    - Read/Write transactions on Wikipedia dump (33M documents): Write-intensive (80% writes)
  - GraphChi (Graph Processing Engine)
    - Twitter graph dump (42M vertexes, 1.5B edges): PageRank, Connected Components

- Environment:
  - Intel Xeon E5505, 16GB RAM
  - Heap/Young Size: 12/2GB
Runtime Object Lifetime Profiler (pausetime percentiles)

(a) Cassandra WI
(b) Cassandra WR
(c) Cassandra RI
(d) Lucene
(e) GraphChi CC
(f) GraphChi PR

Lower is Better
Runtime Object Lifetime Profiler (pausetime percentiles)

(a) Cassandra WI
(b) Cassandra WR
(c) Cassandra RI
(d) Lucene
(e) GraphChi CC
(f) GraphChi PR

Lower is Better
Runtime Object Lifetime Profiler (pausetime distribution)

Left and Lower is Better
Runtime Object Lifetime Profiler (throughput & memory)

Throughput normalized to G1

Max Memory normalized to G1

Lower is Better
Runtime Object Lifetime Profiler (throughput & memory)

Throughput normalized to G1
Max Memory normalized to G1

Up to 6%
Up to 2%
Lower is Better
Conclusions

- Big Data applications suffer from high long tail latencies
- Taking advantage of the proposed hypothesis leads to great reductions in pause times
  - More detailed results in the paper
- ROLP can significantly reduce application pauses with
  - Negligible throughput and memory overhead
  - No code access necessary
  - No programmer effort
- ROLP + NG2C is a JVM drop-in replacement
Conclusions

- Big Data applications suffer from high long tail latencies
- Taking advantage of the proposed hypothesis leads to great reductions in pause times
  - More detailed results in the paper
- ROLP can significantly reduce application pauses with
  - Negligible throughput and memory overhead
  - No code access necessary
  - No programmer effort
- ROLP + NG2C is a JVM drop-in replacement

Thank you for your time. Questions?

Rodrigo Bruno
email: rodrigo.bruno@inf.ethz.ch
webpage: rodrigo-bruno.github.io
code: github.com/rodrigo-bruno/rolp