POLM2: Automatic Object Lifetime-aware Memory Management for Big Data Applications

Rodrigo Bruno and Paulo Ferreira
rodrigo.bruno@tecnico.ulisboa.pt, paulo.ferreira@inesc-id.pt
INESC-ID - Instituto Superior Técnico, University of Lisbon, Portugal

Middleware’17 @ Las Vegas
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
OpenJDK HotSpot Generational GCs

Before GC cycle 1
OpenJDK HotSpot Generational GCs

After GC cycle 1
OpenJDK HotSpot Generational GCs

Before GC cycle 2
OpenJDK HotSpot Generational GCs

After GC cycle 2
OpenJDK HotSpot Generational GCs
OpenJDK HotSpot Generational GCs

After GC cycle 3
OpenJDK HotSpot Generational GCs

After GC cycle 3

Allocated Objects: 32
Number of copies: 9
Big Data Application (simplification)

```
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
Big Data Application in HotSpot GCs

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

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
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

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 4: Task D, Task B, Task B

WS not copied
WS copied once
WS copied twice
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

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

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

Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 10GB/s (DDR3)
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 10GB/s (DDR3)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~\approx 300 \text{ 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 10GB/s (DDR3)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~\(=\) 300 ms
8 Threads, Eden 4GB = copy 7 tasks (3500 MB) ~\(=\) 700 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 10GB/s (DDR3)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~ = **300 ms**
8 Threads, Eden 4GB = copy 7 tasks (3500 MB) ~ = **700 ms**
16 Threads, Eden 8GB = copy 15 task (7500 MB) ~ = **1500 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 10GB/s (DDR3)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~ = 300 ms
8 Threads, Eden 4GB = copy 7 tasks (3500 MB) ~ = 700 ms
16 Threads, Eden 8GB = copy 15 tasks (7500 MB) ~ = 1500 ms

WS not copied
WS copied once
WS copied twice

Long Pauses!
Not Scalable!
Big Data Application in HotSpot GCs

Goal: Reduce Application Pauses caused by Object Copying
(no negative impact on throughput; no programmer effort)

Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 10GB/s (DDR3)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) \(\approx 300\) ms
8 Threads, Eden 4GB = copy 7 tasks (3500 MB) \(\approx 700\) ms
16 Threads, Eden 8GB = copy 15 tasks (7500 MB) \(\approx 1500\) ms

Long Pauses!
Not Scalable!
How to Avoid en-masse Object Copying

- **Attempt 1: Heap Resizing**
  - ✓ Increase Young generation size;
  - ✓ Gives 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 pulling)**
  - ✓ Avoids allocating new memory for future Tasks;
  - ✓ Limits GC effort;
  - ! Requires major rewriting of applications combined with very unnatural Java programming style.

- **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;
  - ✓ Limits GC effort as objects are automatically collected once the scope/region is discarded;
  - ! Requires major rewriting of existing applications;
  - ! Does not allow objects to freely move between scopes. Fits only to bag of tasks model.
How to Avoid en-masse Object Copying

● **Attempt 1: Heap Resizing**
  ✓ Increase Young generation size;
  ✓ Gives 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**
  ✓ Avoids allocating new memory for future tasks;
  ✓ Limits GC effort;
  ! Requires major rewriting of applications combined with very unnatural Java programming style.

● **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;
  ✓ Limits GC effort as objects are automatically collected once the scope/region is discarded;
  ! Requires major rewriting of existing applications;
  ! Does not allow objects to freely move between scopes. Fits only to bag of tasks model.

**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: POLM2

● Goals:
  ○ reduce long tail latencies (due to object copies)
  ○ avoid memory and/or throughput negative impact
  ○ require no programmer knowledge and effort.

● Overview:
  ○ Application execution is profiled (once, before going into production) and an application allocation profile is created (to be used in production)
  ○ Profile is used to automatically insert bytecode to
    ■ pretenure/allocate objects into different dynamic generations depending on their expected lifetime
    ■ Uses NG2C API
  ○ Profiles can then be used to improve performance in production environments
Outline

● POLM2 - Automatic Object Lifetime-aware Memory Management

● Implementation

● Evaluation

● Conclusions & Future Work
POLM2 - Architecture

- Records Allocations
- Processes memory snapshots and allocation records to produce allocation profiles
- Instruments app bytecode to apply profiling decisions
- Profiling
- Production

HotSpot JVM

Recorder

Memory Pages

Time

Memory Snapshot

Dumper

Analyzer

Allocation Profile

Instruments app bytecode to apply profiling decisions
Object Allocation Recording (profiling phase)

- Recorder:
  - Java-agent that intercepts class loading to insert recording code on object allocation
  - Recording code produces a allocation records per object allocation:
    - stack-trace and
    - object unique identifier
Object Allocation Recording (profiling phase)

- Dumper:
  - External process that creates a memory snapshot of the JVM
  - Memory snapshots are incremental (page dirty bit)
  - Memory pages with only garbage are not dumped
  - Heap dumps can be created offline
    - Reduces application interference (such as long app. pauses)
Object Allocation Recording (profiling phase)

- Analyzer:
  - Takes as input:
    - Allocation records (alloc. stack traces and object ids)
    - Heap dumps (live objects and live object ids)
  - Outputs
    - Which allocation sites should pretenure objects because produced objects live for too long
public static void shortTermFactory() { return new int[1024]; }

public static void longTermFactory() { return new int[1024]; }

public static void factory() {
    int[] arr;
    if (foo) arr = shortTermFactory();
    else arr = longTermFactory();
}

dolongTermFactory() { return new int[1024]; }

public static void doWork() {
    if (bar) {
        ...
        factory();
        ...
    } else {
        ...
        factory();
        ...
    }
}

dowork() {
    if (bar) {
        ...
        factory();
        ...
    } else {
        ...
        factory();
        ...
    }
}

public static void main(String[] args) { while(!stop) doWork(); }
Estimating Object Lifetime (profiling phase)

```java
public static void shortTermFactory() { return new int[1024]; }
public static void longTermFactory() { return new int[1024]; }

public static void factory() {
    int[] arr;
    if (foo) arr = shortTermFactory();
    else arr = longTermFactory();
}

public static void doWork() {
    if (bar) {
        ...
        factory();
        ...
    } else {
        ...
        factory();
        ...
    }
}

public static void main(String[] args) { while(!stop) doWork(); }
```
public static int[] allocator() { return new int[1024]; }

public static void shortTermFactory() { return allocator(); }
public static void longTermFactory() { return allocator(); }

public static void factory() {
    int[] arr;
    if (foo) arr = shortTermFactory();
    else arr = longTermFactory();
}

public static void doWork() {
    if (bar) {
        ...
        factory();
        ...
    } else {
        ...
        factory();
        ...
    }
    ...
}

public static void main(String[] args) { while (!stop) doWork(); }
public static int[] allocator()       { return new int[1024]; }

public static void shortTermFactory() { return allocator(); }
public static void longTermFactory()  { return allocator(); }

public static void factory() {
    int[] arr;
    if (foo) arr = shortTermFactory();
    else     arr = longTermFactory();
}

public static void doWork() {
    if (bar) {
        ...
        factory();
        ...
    } else {
        ...
        factory();
        ...
    }
    ...
}

public static void main(String[] args) { while (!stop) doWork(); }
Estimating Object Lifetime (profiling phase)

```java
public static int[] allocator() { return new int[1024]; }

public static void shortTermFactory() { return allocator(); }
public static void longTermFactory() { return allocator(); }

public static void factory() {
    int[] arr;
    if (foo) arr = shortTermFactory();
    else arr = longTermFactory();
}

public static void doWork() {
    if (bar) {
        ...
        factory();
        ...
    } else {
        ...
        factory();
        ...
    }
}

public static void main(String[] args) { while (!stop) doWork(); }
```

Conflict!
(line 1 produces objects with different lifetimes)
Estimating Object Lifetime (profiling phase)

```java
public static int[] allocator() { return new int[1024]; }

public static void shortTermFactory() { return allocator(); }
public static void longTermFactory() { return allocator(); }

public static void factory() {
    int[] arr;
    if (foo) arr = shortTermFactory();
    else arr = longTermFactory();
}

public static void doWork() {
    if (bar) {
        ...
        factory();
        ...
    } else {
        ...
        factory();
        ...
    }
}

public static void main(String[] args) { while (!stop) doWork(); }
```

STTree to resolve conflicts
Estimating Object Lifetime (profiling phase)

```java
public static int[] allocator() { return new int[1024]; }

public static void shortTermFactory() { return allocator(); }
public static void longTermFactory() { return allocator(); }

public static void factory() {
    int[] arr;
    if (foo) arr = shortTermFactory();
    else arr = longTermFactory();
}

public static void doWork() {
    if (bar) {
        ...
        factory();
        ...
    } else {
        ...
        factory();
        ...
    }
}

public static void main(String[] args) { while (!stop) doWork(); }
```

STTree to resolve conflicts
Estimating Object Lifetime (profiling phase)

```java
public static int[] allocator() { return new int[1024]; }
public static void shortTermFactory() { return allocator(); }
public static void longTermFactory() { return allocator(); }

public static void factory() {
    int[] arr;
    if (foo) arr = shortTermFactory();
    else arr = longTermFactory();
}

public static void doWork() {
    if (bar) {
        ...
        factory();
        ...
    } else {
        ...
        factory();
        ...
    }
    ...
}

public static void main(String[] args) { while (!stop) doWork(); }
```
Using Profiling Information (production phase)

- Instrumenter:
  - Takes as input an allocation profile
  - Intercepts class bytecode loading
  - Uses NG2C annotations and API to ensure that new objects are allocated in the correct generation
  - Profiling phase VS Production phase
Outline

- POLM2 - Automatic Object Lifetime-aware Memory Management
- Implementation
- Evaluation
- Conclusions & Future Work
Implementation

● POLM2 is implemented for the OpenJDK 8 HotSpot JVM
  ○ Using NG2C (ISMM’17)
  ○ You can try it with your own application

● Dumper is implemented using CRIU (checkpoint-restore for Linux)

● Recorder implemented using
  ○ Allocation-instrumenter (uses the Java Instrumentation API)

● Instrumented implemented using
  ○ ASM bytecode instrumented (low-level bytecode management API)
Outline

- POLM2 - Automatic Object Lifetime-aware Memory Management
- Implementation
- Evaluation
- Conclusions & Future Work
Evaluation

- Evaluate POLM2’s performance compared to:
  - G1 - popular OpenJDK GCs (ISMM’04)
  - NG2C - multi-generational GC (ISMM’17)
    - requires programmer effort and knowledge

- Big Data Platforms & Workloads:
  - Cassandra (Key-Value Store)
    - YCSB workloads
      - Write-Intensive (75% 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) processing
      - PageRank
      - Connected Components

- Environment:
  - Intel Xeon E5505, 16GB RAM
  - Heap/Young Size: 12/2GB
Evaluation - Pause Times for Cassandra (ms)

Read-Write

Write-Intensive

Read-Intensive

POLM2
NG2C
G1
Evaluation - Pause Times for Lucene and GraphChi (ms)

Lucene

GraphChi CC

GraphChi PR
Evaluation - Throughput (Cassandra) - 10 min sample

More results in the paper
Evaluation - Dumper vs jmap

![Graph showing Snapshot time (norm to jmap) and Snapshot size (norm to jmap)]
Outline

● POLM2 - Automatic Object Lifetime-aware Memory Management
● Implementation
● Evaluation
● Conclusions & Future Work
Conclusions

● POLM2 provides a realistic approach to improve Big Data application memory management in HotSpot
  ○ It decreases pause times by avoiding object copying
  ○ It requires no programmer effort and knowledge
  ○ It does not compromise throughput

● Results are very encouraging

● Code is available at github.com/rodrigo-bruno/polm2
Future Work

- Provide in-JVM support for dynamic generations and pretenuring
  - JVM must internally estimate the appropriate generation for each alloc. site
  - JVM must dynamically change the target generation for each alloc. site
  - Work in progress
    - Current prototype leads to up to 8% throughput degradation for Cassandra
    - There are still several performance improvements to be done.
Thank you for your time.
Questions?

Rodrigo Bruno
email: rodrigo.bruno@tecnico.ulisboa.pt
webpage: www.gsd.inesc-id.pt/~rbruno
github: github.com/rodrigo-bruno