NG2C: Pretenuring Garbage Collection with Dynamic Generations for HotSpot Big Data Applications

Rodrigo Bruno
Luis Picciochi Oliveira
Paulo Ferreira

03-160447

Tomokazu HIGUCHI
Overview

• Introduction
• Related Work
• Pretenuring GC with Dynamic Generations
• Implementation
• Evaluation
• Conclusions

Additional commentaries are added to explain unfamiliar words.
Introduction

Applications become..

- More latency-sensitive
- Larger scale

Developers resort to Cashing (keeping data in memory)

Cashing helps developers to avoid costly operations.

But..

The inevitable GC pause has become a serious problem.
Introduction

To solve the GC pause problem..

I. A new GC algorithm, **NG2C**.
   I. Combines an arbitrary number of dynamic generations with pretenuring.
   II. Implemented for OpenJDK 8 HotSpot JVM as an extension of next OpenJDK by-default GC, Garbage First (G1).

II. Object Lifetime Recorder (**OLR**) profiler tool on a production JVM.
Related Work

Generation Collector

- New objects are allocated in the Eden realm.
- Survivor objects are moved to “Older” realm gradually.
Related Work

Pretenuring

Objects which are known to live for a long time are allocated directly in the Old Generation realm.
Related Work

The hypothesis: Objects allocated in the same scope have similar lifetimes.

Example

```r
{o
  Family osomatsu;
  osomatsu.born();
  osomatsu.graduate();
  osomatsu.work();
  osomatsu.marry();
  ...
}
```

If Mr. Osomatsu can be pretenured, all his family can be pretenured. That’s because they are in the same scope.

Lifetime Profiling Optimization
Related Work

Off-heap based Solutions

Problems
- Developers should manage memory allocations without the GC.
- Serialization and deserialization should be done before using the data.
Additional Commentary

Garbage First Garbage Collection (G1 GC)

- Using the heap as several large data structures is inefficient.
- G1 algorithm does not adopt this heap structure.

- In the algorithm of G1, large heap is equally divided into small regions.
- Region’s size is depended on the heap size.
  - It can be changed by `-XX:G1HeapRegionSize` from 1MB to 32MB.
Additional Commentary

Parallel と Concurrent

アプリケーション停止 = Stop The World (STW)

Application Thread

GC Thread

Parallel (Serial = Threadが1つ)

Concurrent (実際は部分的に停止するMostly)

URL: https://www.slideshare.net/YujiKubota/garbage-first-garbage-collection
Additional Commentary

- The amount of unnecessary objects is different from each region.
- GC takes place in regions which have relatively many garbages.
- A Humongous Object is allocated in a dedicated region.
- Allocating it may cause the Full GC. So G1 avoid that as much as possible.

The marking takes place in the concurrent cycle without STW.
Pretenuring GC with Dynamic Generations

Existing GC (G1)

- **Heap Layout:**
  The heap is created with two static generations: Young and Old.

- **Allocation:**
  By default, all objects are allocated in the Young generation.

NG2C

- **Heap Layout:**
  An arbitrary number of dynamic generations can be added to the heap at runtime.

- **Allocation:**
  All objects can be allocated in any generations.

NG2C can reduce the amount of a survivor object promoting.
Pretenuring GC with Dynamic Generations

NG2C API

• When a thread is created, its generation is Old.

• If the thread decides to create a new generation, the thread’s current generation will change.

• To allocate an object in the current generation, the new operator must be annotated with @Gen.

• All allocation sites with no @Gen will allocate objects into the Young generation.

---

Listing 1. NG2C API

```java
// Methods added in class java.lang.System:
public static Generation newGeneration();
public static Generation getGeneration();
public static Generation setGeneration(Generation);
```

Listing 2. Job Processing Code Sample

```java
public void runTask() {
    Generation gen = System.newGeneration();
    while (running) {
        DataChunk data = new @Gen DataChunk();
        loadData(data);
        doComplexProcessing(data);}
```
Pretenuring GC with Dynamic Generations

Supplementary explanation

```java
public void runTask() {
    Generation gen = System.newGeneration();
    while (running) {
        DataChunk data = new @Gen DataChunk();
        Hoge foo = new Hoge();
        Young
        loadData(data);
        doComplexProcessing(data);
    }
}
```
Pretenuring GC with Dynamic Generations

Figure 1. Allocation of Objects in Different Generations
Pretenuring GC with Dynamic Generations

Memory Allocation Algorithms

Algorithm 1 Memory Allocation - Object Allocation

1: procedure OBJECT ALLOCATION
2: size ← size of object to allocate
3: klass ← class of object to allocate
4: gen ← current thread generation
5: isGen ← new operator annotated with @Gen?
6: if isGen then
7: tlab ← TLAB used for generation gen
8: else
9: tlab ← TLAB used for Young
10: if end(tlab) − top(tlab) >= size then
11: object ← init(klass, top(tlab))
12: bumpTop(tlab, size)
13: return object
14: slow path:
15: if size >= size(tlab)/8 then
16: returnALLOC IN REGION(klass, size)
17: else
18: returnALLOC IN TLAB(klass, size)

Algorithm 2 Memory Allocation - Allocation in Region

1: procedure ALLOC IN REGION(klass, size)
2: gen ← current thread generation
3: isGen ← new operator annotated with @Gen?
4: if isGen then
5: region ← gen alloc region
6: else
7: region ← Young alloc region
8: if end(region) − top(region) >= size then
9: object ← init(klass, top(region))
10: bumpTop(region, size)
11: return object
12: if isGen then
13: region ← new gen alloc region
14: else
15: region ← new Young alloc region
16: if region not null then
17: object ← init(klass, top(region))
18: bumpTop(region, size)
19: return object
20: else
21: trigger GC and retry allocation
Pretenuring GC with Dynamic Generations

TLAB (Thread Local Application Buffer)
Pretenuring GC with Dynamic Generations

- Minor collection:
  Collects the Young generation and promote objects that survived a number of collections to the Old generation.

- Mixed collection:
  Collects the Young generation plus other regions from multiple generations whose amount of live data is low.

- Full collection:
  Collects the whole heap.

Notes:
Concurrent marking cycles are triggered when the heap usage exceeds a configure threshold.
Pretenuring GC with Dynamic Generations

Figure 2. Types of collections (red represents unreachable data)
Pretenuring GC with Dynamic Generations

Object Lifetime Recorder

- OLR is composed by three components
  - Allocation Recorder: Record memory allocation sites.
  - JVM Dumper: Creates incremental heap dump (a memory snapshot of the Java heap).
  - Object Graph Analyzer: Process the allocation sites and heap dumps and generates an object graph.

- The result graph shows which objects should belong in the same generation, and where these objects are allocated.
Pretenuring GC with Dynamic Generations

Figure 3. Object Lifetime Recorder Profiler Architecture
Implementation

• Authors modify both the interpreter and the JIT (Just In Time) compiler to add the notion of generations.

• Before JIT, a map of bytecode index with annotation data is stored along the method meta data.
  – Using this map, it is possible to know in constant time if a particular byte code index is annotated with @Gen or not.

• The annotation data is hardcoded into the compiled code.
  – This frees the compiled code from accessing the annotation map.
Evaluation

- Cassandra: NoSQL middle-ware
- Lucene: Full-featured text search engine library
- GraphChi: Disk-based large-scale graph computation system

<table>
<thead>
<tr>
<th>Platform</th>
<th>Workload</th>
<th>CPU</th>
<th>RAM</th>
<th>OS</th>
<th>Heap Size</th>
<th>Young Gen Size</th>
<th>LOC Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>Feedzai</td>
<td>Intel Xeon E5-2680</td>
<td>64 GB</td>
<td>CentOS 6.7</td>
<td>30 GB</td>
<td>4 GB</td>
<td>22</td>
</tr>
<tr>
<td>Cassandra</td>
<td>WI,RW,RI</td>
<td>Intel Xeon E5505</td>
<td>16 GB</td>
<td>Linux 3.13</td>
<td>12 GB</td>
<td>2 GB</td>
<td>22</td>
</tr>
<tr>
<td>Lucene</td>
<td>RW</td>
<td>AMD Opteron 6168</td>
<td>128 GB</td>
<td>Linux 3.16</td>
<td>120 GB</td>
<td>2 GB</td>
<td>8</td>
</tr>
<tr>
<td>GraphChi</td>
<td>PR,CC</td>
<td>AMD Opteron 6168</td>
<td>128 GB</td>
<td>Linux 3.16</td>
<td>120 GB</td>
<td>6 GB</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1. Evaluation Environment Summary
Evaluation

Figure 8. Throughput vs Pause Time

CMS: Concurrent Mark-Sweep
Conclusion

- A new HotSpot GC algorithm, NG2C that avoids copying objects within the heap by aggregating objects with similar lifetime profiles in separate generations.

- Comparison with current collectors (G1 and CMS)
  - Object copying: up to 89.2%
  - Worst GC pause time:
    - 94.8% in Cassandra
    - 85% in Lucene
    - 96.45% in GraphChi