Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services

Mingyu Wu¹, Ziming Zhao¹, Yanfei Yang¹, Haoyu Li¹, Haibo Chen¹, Binyu Zang¹, Haibing Guan¹, Sanhong Li², Chuansheng Lu², Tongbao Zhang²
1 Shanghai Jiao Tong University
2 Alibaba Group
A page is multiple services

- A user request relies on multiple services
  - Small, single-purposed, and interactive
A page is multiple services

- A user request relies on multiple services
  - Small, single-purposed, and interactive

- Most online services in Alibaba are written in Java
A page is multiple services

- A user request relies on multiple services
  - Small, single-purposed, and interactive

- Most online services in Alibaba are written in Java
  - Services might be paused by the modules in Java runtime
Garbage Collection (GC)

- Language runtimes (like JVM) leverages GC for automatic memory management

- Prior GC in JVM is Stop-The-World (STW) GC
  - Pause application threads (mutators) for memory reclamation
  - Pros: High GC throughput, satisfying CPU efficiency
  - Cons: High application latency
GC Pauses Introduce The Long-Tail Problem

- Running the Coupon service from Alibaba for 30 seconds
  - Each GC cycle follows stragglers
  - GC is the killer factor for the tail latency
Alternative Design: Concurrent GC

- Allowing mutators to execute even during collection
  - Effective on reducing GC pauses
  - Two categories: partially-concurrent / mostly-concurrent

- Partially-concurrent collectors (example: G1)
  - Mutators can execute in some GC phases (other phases are still STW)
  - Proposing running arguments to reduce the duration of STW phases
Alternative Design: Concurrent GC

• Allowing mutators to execute even during collection
  – Effective on reducing GC pauses
  – Two categories: partially-concurrent / mostly-concurrent

• Mostly-concurrent collectors (example: Shenandoah)
  – Mutators can execute in nearly all GC phases
  – Only introducing short pauses in some synchronization points
Concurrent GC Is Not Always Helpful

- Partially concurrent GC: Tuning may lead to worse tail latency
  - Evaluation on G1: tuning the MaxGCPauseMillis argument to restrict pause time
  - Decreasing MaxGCPauseMillis can reduce per-GC pauses

<table>
<thead>
<tr>
<th>MaxGCPauseMillis</th>
<th>30ms</th>
<th>40ms</th>
<th>60ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. GC pause (ms)</td>
<td>21.815</td>
<td>21.459</td>
<td>39.856</td>
</tr>
<tr>
<td>Avg. GC pause (ms)</td>
<td>34.441</td>
<td>40.724</td>
<td>48.491</td>
</tr>
<tr>
<td>The number of GC</td>
<td>550</td>
<td>392</td>
<td>111</td>
</tr>
<tr>
<td>Avg. CPU util.</td>
<td>51.45%</td>
<td>50.81%</td>
<td>36.17%</td>
</tr>
<tr>
<td>p99 latency (ms)</td>
<td>1942.09</td>
<td>1389.99</td>
<td>148.85</td>
</tr>
</tbody>
</table>
Concurrent GC Is Not Always Helpful

- Partially concurrent GC: Tuning may lead to worse tail latency
  - Evaluation on G1: tuning the `MaxGCPauseMillis` argument to restrict pause time
  - Decreasing `MaxGCPauseMillis` can reduce per-GC pauses
  - But the GC frequency increase and consume more CPU resource

<table>
<thead>
<tr>
<th>MaxGCPauseMillis</th>
<th>30ms</th>
<th>40ms</th>
<th>60ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. GC pause (ms)</td>
<td>21.815</td>
<td>21.459</td>
<td>39.856</td>
</tr>
<tr>
<td>Avg. GC pause (ms)</td>
<td>34.441</td>
<td>40.724</td>
<td>48.491</td>
</tr>
<tr>
<td>The number of GC</td>
<td>550</td>
<td>392</td>
<td>111</td>
</tr>
<tr>
<td>Avg. CPU util.</td>
<td>51.45%</td>
<td>50.81%</td>
<td>36.17%</td>
</tr>
<tr>
<td>p99 latency (ms)</td>
<td>1942.09</td>
<td>1389.99</td>
<td>148.85</td>
</tr>
</tbody>
</table>
Concurrent GC Is Not Always Helpful

- Partially concurrent GC: Tuning may lead to worse tail latency
  - Evaluation on G1: tuning the `MaxGCPauseMillis` argument to restrict pause time
  - Decreasing `MaxGCPauseMillis` can reduce per-GC pauses
  - But the GC frequency increase and consume more CPU resource
  - **Result: worse application tail latency**

<table>
<thead>
<tr>
<th>MaxGCPauseMillis</th>
<th>30ms</th>
<th>40ms</th>
<th>60ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. GC pause (ms)</td>
<td>21.815</td>
<td>21.459</td>
<td>39.856</td>
</tr>
<tr>
<td>Avg. GC pause (ms)</td>
<td>34.441</td>
<td>40.724</td>
<td>48.491</td>
</tr>
<tr>
<td>The number of GC</td>
<td>550</td>
<td>392</td>
<td>111</td>
</tr>
<tr>
<td>Avg. CPU util.</td>
<td>51.45%</td>
<td>50.81%</td>
<td>36.17%</td>
</tr>
<tr>
<td>p99 latency (ms)</td>
<td>1942.09</td>
<td>1389.99</td>
<td>148.85</td>
</tr>
</tbody>
</table>
Concurrent GC Is Not Always Helpful

- Mostly-concurrent GC consumes even more CPU resource
  - Evaluation on Shenandoah: the same application throughput as G1
  - The GC pauses in Shenandoah become quite short

<table>
<thead>
<tr>
<th>GC Type</th>
<th>G1-30ms</th>
<th>Shenandoah</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. GC pause (ms)</td>
<td>21.815</td>
<td>5.860</td>
</tr>
<tr>
<td>Avg. GC pause (ms)</td>
<td>34.441</td>
<td>18.764</td>
</tr>
<tr>
<td>GC Duration (s)</td>
<td>18.94</td>
<td>53.05</td>
</tr>
<tr>
<td>Avg. CPU util.</td>
<td>51.45%</td>
<td>83.05%</td>
</tr>
<tr>
<td>p99 latency (ms)</td>
<td>1942.09</td>
<td>3614.58</td>
</tr>
</tbody>
</table>
Concurrent GC Is Not Always Helpful

- Mostly-concurrent GC consumes even more CPU resource
  - Evaluation on Shenandoah: the same application throughput as G1
  - The GC pauses in Shenandoah become quite short
  - But: the overall GC time is longer and consumes more CPU resource

<table>
<thead>
<tr>
<th>GC Type</th>
<th>G1-30ms</th>
<th>Shenandoah</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. GC pause (ms)</td>
<td>21.815</td>
<td>5.860</td>
</tr>
<tr>
<td>Avg. GC pause (ms)</td>
<td>34.441</td>
<td>18.764</td>
</tr>
<tr>
<td>GC Duration (s)</td>
<td>18.94</td>
<td>53.05</td>
</tr>
<tr>
<td>Avg. CPU util.</td>
<td>51.45%</td>
<td>83.05%</td>
</tr>
<tr>
<td>p99 latency (ms)</td>
<td>1942.09</td>
<td>3614.58</td>
</tr>
</tbody>
</table>
Concurrent GC Is Not Always Helpful

- Mostly-concurrent GC consumes even more CPU resource
  - Evaluation on Shenandoah: the same application throughput as G1
  - The GC pauses in Shenandoah become quite short
  - But: the overall GC time is longer and consumes more CPU resource
  - Result: worse application tail latency

<table>
<thead>
<tr>
<th>GC Type</th>
<th>G1-30ms</th>
<th>Shenandoah</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. GC pause (ms)</td>
<td>21.815</td>
<td>5.860</td>
</tr>
<tr>
<td>Avg. GC pause (ms)</td>
<td>34.441</td>
<td>18.764</td>
</tr>
<tr>
<td>GC Duration (s)</td>
<td>18.94</td>
<td>53.05</td>
</tr>
<tr>
<td>Avg. CPU util.</td>
<td>51.45%</td>
<td>83.05%</td>
</tr>
<tr>
<td>p99 latency (ms)</td>
<td>1942.09</td>
<td>3614.58</td>
</tr>
</tbody>
</table>
Why Are Concurrent Collectors Inefficient?

- **Contentions on CPU resources**
  - GC threads and mutators may run on the same CPU

- **Synchronizations**
  - GC threads have to synchronize with mutators when accessing the same object

- **Software barrier code**
  - Mutators should check invariants before *every* read/write operations

```java
if (is_being_collected(y)) {
    slow_path(y.x, z);
} else {
    y.x = z;
}
```
Why Concurrent Collectors are Inefficient?

• CONTENTIONS ON CPU RESOURCES
  – GC threads and mutators may run on the same CPU

• SYNCHRONIZATIONS
  – GC threads and mutators may synchronize with each other when accessing the same object

• SOFTWARE BARRIER CODE
  – Mutators should check invariants before every read/write operations

Can we design a collector with both low **latency** and high **CPU-efficiency**?

```java
if (is_being_collected(y)) {
    slow_path(y.x, z);
} else {
    y.x = z;
}
```
Opportunities: Idle Cores in GC

- It is hard to reach perfect scalability for GC
  - Workload for each GC threads is unknown before processing -> imbalance

- Evaluation on Specjbb2015 with 80 logic cores
  - Even for STW GC, performance remains stable when reaching 30 cores
Opportunities: Idle Cores in GC

• It is hard to reach perfect scalability for GC
  – Workload for each GC threads is unknown before processing -> imbalance

• Evaluation on Specjbb2015 with 80 logic cores
  – Even for STW GC, performance remains stable when reaching 30 cores

• The default setting in OpenJDK: using less GC threads
  – The default GC thread number with 80 cores: 53
Opportunities: Skewed Write Behavior

- Interactive services process requests in sessions
  - A service creates a session for each request

- Skewed write behaviors: most writes fall in the newly-created objects (named working-set)
Opportunities: Skewed Write Behavior

- **Interactive services process requests in sessions**
  - A service creates a session for each request

- **Skewed write behaviors: most writes fall in the newly-created objects (named working-set)**
  - Different services show similar behaviors
  - Other applications (Spark) are different
Opportunities: MPK

- MPK (Memory Protection Keys) allows intra-process page permission configuration
  - Each page in a process belongs to a domain
  - Each thread can gain different permissions on different domains

- MPK is mainly exploited for security consideration
  - Is it possible to use MPK for performance improvement?
Our Design: Platinum

• A mostly-concurrent collector with moderate CPU consumption
  – Low latency: allowing mutators to execute with GC threads
  – CPU-efficient: removing synchronizations and barriers

• Execution flow: a three-phase collection algorithm
  – Init pause: initializing the collection (<1ms)
  – Concurrent scavenge: mutators can execute (but with restrictions)
  – Final pause: updating stale references (~10ms)
Design Overview

- Idle core collection
  - Collecting idle cores (not used by GC threads) and give them to mutators

- Heap partition

- Barrier elimination
Design Overview

- **Idle core collection**
- **Heap partition**
  - Isolating the memory between threads
  - Minimizing inter-thread synchronizations
- **Barrier elimination**
Design Overview

• Idle core collection

• Heap partition

• Barrier elimination
  – Using MPK to divide heap into domains
  – Removing the needs for barriers
Idle Core Collection

• Binding GC threads to separated cores (with `sched_setaffinity`)
  – Each GC thread can monopoly its core during GC
  – Other unused cores are identified as *idle cores*

• Changing the affinity of mutators with/without GC
  – During GC: mutators can only run on idle cores
  – Out of GC: mutators can run on all cores

• Avoiding CPU contention between GC threads and mutators
Heap Partition

- Partition the heap into three areas according to the skewed write behavior
  - Collection area: containing objects under collection
  - Pinned area: containing active objects being used by mutators (estimated)
  - Allocation area: reserved for memory allocation during GC
Heap Partition

- **Partition the heap into three areas according to the skewed write behavior**
  - Collection area: containing objects under collection
  - Pinned area: containing active objects being used by mutators (estimated)
  - Allocation area: reserved for memory allocation during GC

- **Enforcing isolation for GC threads and mutators**
  - GC threads only collect objects in the collection area
  - Mutators will mostly access the other two areas
It is still possible for mutators to access the collection area
  – Although it is rare (due to the skewed write behavior)

Traditional solution: write barriers
  – Checking the address before every write -> runtime overhead

```java
if (is_in_collection_area(y)) {
    slow_path(y.x, z);
} else {
    y.x = z;
}
```
It is still possible for mutators to access the collection area
  - Although it is rare (due to the skewed write behavior)

Our solution: MPK-based hardware protection
  - Mutators only has read-only permission on the collection area
  - Mutator-writes to the collection area will trigger page faults and thus be corrected
Evaluation Setup

• **Hardware:** Intel Gold 6138 CPU (80 cores)

• **Software:** OpenJDK 8u141, 16GB Java heap

• **Baseline:**
  – Concurrent-Mark-Sweep (CMS): a classic partially concurrent collector
  – Garbage-First (G1): a tunable partially concurrent collector (default in OpenJDK 9)
  – Shenandoah: a mostly-concurrent collector

• **Applications:**
  – Specjbb2015: A simulated online supermarket (web services)
  – Cassandra: A key-value store (storage service)
  – Coupon: A real service in Alibaba
Specjbb2015

• Evaluating its performance with different throughput settings
  – Better than CMS (up to 79.3%), comparable with G1
  – The best maximum throughput among all
Cassandra

- **Evaluation with two YCSB workload settings:**
  - Read-intensive (76000 reads and 4000 updates per second)
  - Write-intensive (40000 reads and 40000 updates per second)
  - Result:
    - Comparable with Shenandoah in read-intensive
    - Less improvement in write-intensive but still better than CMS
• **Evaluation with production traces in Alibaba**
  – 96-core machine, 16GB Java heap
  – 66.8% and 23.5% improvement on p99 latency compared with CMS and G1
CPU Utilization Under Stressful Workload

- Collecting the average CPU utilization with Linux `sar`
  - CMS induces moderate CPU consumption (but poor latency)
  - Platinum has better CPU utilization compared with G1 and Shenandoah

<table>
<thead>
<tr>
<th>Application</th>
<th>CMS</th>
<th>G1</th>
<th>Shenandoah</th>
<th>Platinum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specjbb2015</td>
<td>48.79%</td>
<td>77.66%</td>
<td>77.80%</td>
<td>50.56%</td>
</tr>
<tr>
<td>Cassandra</td>
<td>12.10%</td>
<td>15.97%</td>
<td>14.93%</td>
<td>13.79%</td>
</tr>
<tr>
<td>Coupon</td>
<td>38.47%</td>
<td>36.17%</td>
<td>83.05%</td>
<td>34.50%</td>
</tr>
</tbody>
</table>
Conclusion

- Prior GC makes a trade-off between latency and CPU efficiency

- **Platinum**: a mostly-concurrent GC with satisfying CPU efficiency
  - Idle core collection -> mitigate CPU contention between threads
  - Heap partition -> minimize synchronizations
  - MPK-based barrier elimination -> reduce runtime overhead

- Achieving both low latency and moderate CPU consumption for interactive services

Thanks!

Q&A: mingywu93@gmail.com