The Garbage-First Garbage Collector

Tony Printezis, Sun Microsystems
Paul Ciciora, Chicago Board Options Exchange

#TS-5419
Trademarks And Abbreviations
(to get them out of the way...)

- Java™ Platform, Standard Edition (Java SE)
- Java Hotspot™ Virtual Machine (HotSpot JVM™)
- Solaris™ Operating System (Solaris OS)
- Garbage Collection (GC)
- Concurrent Mark-Sweep Garbage Collector (CMS)
- Garbage-First Garbage Collector (G1)
Who Are These Guys?

➢ **Tony Printezis**
  - Staff Engineer, HotSpot JVM GC Group
  - Sun Microsystems, Burlington, MA

➢ **Paul Ciciora**
  - “The Adventurous Customer”
  - Dept Head, Object Oriented Infrastructure
  - Chicago Board Options Exchange (CBOE), Chicago, IL
Learn how **Garbage-First**, aka **G1**, the new low-latency garbage collector in the HotSpot JVM works and what it will mean to your application in the future.
Agenda

- Garbage-First Attributes
- Garbage-First Operation
- First Impressions: CBOE
- Final Thoughts
Agenda

- Garbage-First Attributes
- Garbage-First Operation
- First Impressions: CBOE
- Final Thoughts
The Garbage-First Garbage Collector (G1)

- Future CMS Replacement
- Server “Style” Garbage Collector
- Parallel
- Concurrent
- Generational
- Good Throughput
- Compacting
- Improved ease-of-use
- Predictable (though not hard real-time)
The Garbage-First Garbage Collector (G1)

- Future CMS Replacement
- Server “Style” Garbage Collector
- Parallel
- Concurrent
- Generational
- Good Throughput
- Compacting
- Improved ease-of-use
- Predictable (though not hard real-time)

Main differences between Garbage-First and CMS
GC Work: Parallelism & Concurrency

- Application Thread
- GC Thread
Predictability

- Cannot guarantee hard real-time behavior
  - OS scheduling
  - Other processes
  - Application behavior (without analysis)

- Soft real-time
  - Achieve good level of predictable behavior
  - With high probability... but no hard guarantees

- If you want hard real-time guarantees, please consider using the Sun Microsystems' Java Real-Time System

- We are expecting G1 to be more predictable than CMS
Compaction

- Consolidates free space in large chunk(s)
- Battles fragmentation
  - Important for long-running applications
  - We can sleep better at night... and in the daytime too. :-)
- Enables fast allocation
  - Linear, no free lists
  - TLABs (thread-local allocation buffers)
  - Infrequent synchronization at allocation time
    - Only at the slow path
- No free lunch!
  - Copying is the largest contributor to pause times
Weak Generational Hypothesis

- Two observations
  - “Most newly-allocated objects will die young.”
  - “There are few old-to-young references.”

- Split the heap into “generations”
  - Usually two: young generation / old generation

- Concentrate collection effort on the young generation
  - Good payoff (a lot of space reclaimed) for your collection effort
  - Lower GC overhead
  - Most pauses are short

- Reduced allocation rate into the old generation
  - Young generation acts as a “filter”
Why Generational?

- Most Java applications
  - Conform to the weak generational hypothesis
  - Really benefit from generational GC
  - Performance-wise, generational GC is hard to beat in most cases
- All GCs in the HotSpot JVM are generational
Agenda

- Garbage-First Attributes
- Garbage-First Operation
- First Impressions: CBOE
- Final Thoughts
Color Key

- Non-Allocated Space
- Young Generation
- Old Generation
- Recently Copied in Young Generation
- Recently Copied in Old Generation
Young GCs in CMS (i)

- Young generation, split into
  - Eden
  - Survivor spaces
- Old generation
  - In-place de-allocation
  - Managed by free lists
Young GCs in CMS (ii)

During a young generation GC
- Survivors from the young generation are evacuated to
  - Other survivor space
  - Old generation

CMS
Young GCs in CMS (iii)

End of young generation GC

CMS
Young GCs in G1 (i)

- Heap split into regions
  - Currently 1MB regions
- Young generation
  - A set of regions
- Old generation
  - A set of regions
During a young generation GC

- Survivors from the young regions are evacuated to:
  - Survivor regions
  - Old regions
Young GCs in G1 (iii)

End of young generation GC
Summary: Young GCs in G1

> Single physical heap, split into regions
  - Set of contiguous regions allocated for large ("humongous") objects

> No physically separate young generation
  - A set of (non-contiguous) regions
  - Very easy to resize

> Young GCs
  - Done with "evacuation pauses"
  - Stop-the-world
  - Parallel
  - Evacuate surviving objects from one set of regions to another
Old GCs in CMS (Sweeping After Marking) (i)

Concurrent marking phase
- Two stop-the-world pauses
  - Initial mark
  - Remark
- Marks reachable (live) objects
- Unmarked objects are deduced to be unreachable (dead)
Old GCs in CMS (Sweeping After Marking) (ii)

Concurrent sweeping phase
- Sweeps over the heap
- In-place de-allocates unmarked objects
Old GCs in CMS (Sweeping After Marking) (iii)

End of concurrent sweeping phase
- All unmarked objects are deallocated
Old GCs in G1 (After Marking) (i)

Concurrent marking phase
- One stop-the-world pause
  - Remark
  - (Initial mark piggybacked on an evacuation pause)
- Calculates liveness information per region
  - Empty regions can be reclaimed immediately
Old GCs in G1 (After Marking) (ii)

- End of remark phase
Old GCs in G1 (After Marking) (iii)

- Reclaiming old regions
  - Pick regions with low live ratio
  - Collect them piggy-backed on young GCs
    - Only a few old regions collected per such GC
Old GCs in G1 (After Marking) (iv)

- We might leave some garbage objects in the heap
  - In regions with very high live ratio
  - We might collect them later
Summary: Old GCs in G1

- **Concurrent marking phase**
  - Calculates liveness information per region
  - Identifies best regions for subsequent evacuation pauses
  - No corresponding sweeping phase
  - Different marking algorithm than CMS
    - Snapshot-at-the-beginning (SATB)
    - Achieves shorter remarks

- **Old regions reclaimed by**
  - Remark (when totally empty)
  - Evacuation pauses

- **Most reclamation happens with evacuation pauses**
  - Compaction
G1: “One-And-A-Half” GCs

- **CMS**
  - Young generation GCs
  - Old generation concurrent marking and sweeping

- **G1**
  - Evacuation pauses
    - Both for young and old regions
  - Only concurrent marking
Remembered Sets

- During evacuation pauses
  - Need to identify “roots” from other regions

- We maintain “remembered sets”
  - One per region
  - Keeps track of all heap locations with references into that region

- We can pick any region to collect
  - Without sweeping the whole heap to find references into it

- Remembered set maintenance
  - Write barrier + concurrent processing

- Remembered set footprint
  - <5% of the heap
Pause Prediction Model

- **User-defined pause goal**
  - Goal, not a promise or a guarantee!

- **Stop-the-world pauses**
  - Evacuation pauses are the main “bottleneck”
    - Highly application-dependent
  - Remark pauses are short

- **Pause prediction model**
  - Keep stats on pause behavior
  - Predict the maximum amount of work that does not violate the goal
    - e.g., young generation size
  - Works best for “steady-state” applications
CMS vs. G1 Comparison
Agenda

- Garbage-First Attributes
- Garbage-First Operation
- First Impressions: CBOE
- Final Thoughts
CBOE: Leading the Options Industry for 35 Years

- Founded in 1973, CBOE is the largest U.S. options marketplace
- Original marketplace for standardized, exchange-traded options
- Industry leader in product innovation
- Powered by CBOEdirect, Hybrid Trading System integrates electronic and open outcry trading, offering unparalleled trading choice
- Launched CBOE Futures Exchange (CFE) in March 2004
CBOE Leads the Industry in Market Share

- CBOE is the number 1 U.S. options marketplace, handling one-third of total industry volume in 2007
- CBOE 2007 market share numbers:
  - Total options: 33.0%
  - Equity options: 25.7%
  - Multiply-listed index/ETF options: 36.7%
  - Cash index options: 86.1%
Profile of the Adventurous Customer
...in a hyper-competitive industry!

- Started tuning GC with 1.2 SemiSpaces
- Early adopter of Parallel Young Gen GC
- Switched to CMS with 1.4.2
- Started testing the Java platform on the Solaris OS 10 x86 in 2005
- Solaris OS 10 x86 in production since May 2006
  - Zones
  - Processor sets
  - FX scheduling
- Currently running 6u4
  - First customer in production
- Leveraging open source
Keeping Current: Effort and Reward

No Pain, No Gain

vs.

Lots of Pain, but Lots of Gain
CBOE Performance Test Environment
How to mitigate the risk of being on the bleeding edge

➢ Two full production slices, including hardware redundancy
  • Pretesting production software / hardware upgrades
➢ Two dedicated “wind tunnels”
  • Software / hardware proof-of-concept + GC tuning
  • Heavily virtualized (flexibility)
➢ ~150 servers
➢ Personnel
  • 8 full-time staff
  • Running round the clock, on a 20-hour cycle
GC Logging in Production

Turn on the headlights!

-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-XX:+PrintHeapAtGC
-XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCApplicationConcurrentTime
-verbose:gc
-XX:+PrintGCDateStamps
-XX:+PrintGCTaskTimeStamps
-XX:PrintCMSStatistics=1
-XX:+PrintGCTaskTimeStamps
-XX:+PrintTenuringDistribution
-XX:PrintFLSStatistics=1
The GC Challenge

Main objectives

• Lowering the frequency of the collections
• Lowering the time of the collections
• Avoiding a Full GC
The CBOE Problem Set

Worst-case scenario

- An average young GC promotes ~6 MBs in ~100 ms at a frequency of ~4 secs
- CMS cycles occur every ~3-4 mins
  - It usually takes ~12-13 secs for a cycle to complete
  - If it does not complete in ~45-50 secs “we lose the race”
- Tuning allows for more throughput which just brings us back to the original problem: more garbage!

- The challenge
  - Reduce young GC times to under 50 ms
  - Maintain intervals to > 5 secs
  - Never lose the race
CMS GC Pauses

GC Thread Stop Time

57 GCs in 60 secs

~1.7 GB processed ~533 MB promoted

Source: CBOE
The CMS Obstacle Course

Promotion Size / Old Generation Occupancy

Source: CBOE
The CMS Obstacle Course

Promotion Size / Old Generation Occupancy

Source: CBOE
CMS vs. G1 Results

- Homogeneous stress test at 3,000 TPS steady
  - Sun Fire™ server X4450, 16-way, 32 GB memory, 32-bit version

- Preliminary Results
  - Achieved a good measure of stability
    - We can run a lot longer than we could 3 months ago. :-)
  - Currently, pause times quite higher than CMS
    - Due to some “band-aids” to address some concurrency-related bugs
  - SATB works as advertised
    - Remarks consistently down to 12ms from 50-60ms with CMS
  - The adventurous customer very satisfied with the G1 GC output
    - Very detailed breakdown to identify tuning opportunities
  - GC team very receptive to “constructive criticism”
Agenda

- Garbage-First Attributes
- Garbage-First Operation
- First Impressions: CBOE
- Final Thoughts
G1 Summary

- Server-Style Low-Latency GC
  - Parallel
  - Concurrent
  - Compacting
  - Soft Real-Time
- Future replacement for CMS
Future Directions

➢ Reduce
  • Stop-the-world pause times
  • GC overhead

➢ Deal with mostly-static heap subsets
  • Automatically discover a mostly-static set of regions
  • Reduce marking overhead for that set
  • Target: large caches

➢ Keep taking advantage of Paul's “wind tunnels” :-}
Reading Material


THANK YOU

Tony Printezis, Sun Microsystems
Paul Ciciora, CBOE

#TS-5419