Garbage-First: Low Latency, High Throughput Garbage Collection

David Detlefs
Java™ Technology Research Group
Sun Microsystems Laboratories (Burlington, MA)
(Steve Heller, Christine Flood, Tony Printezis)
Outline

- Goals.
- Mechanisms.
- Results.
- Related work.
- Future work.
Garbage-First Goals

- Scale to large heaps, large #'s of threads.
- High-probability compliance with a real-time goal (aka MMU specification):
  - “GC should consume no more than \( x \) ms in any \( y \) ms interval...”
- High throughput
  - On machines with large numbers of processors
    - Implies effective parallelism when the mutator is stopped, and...
    - Use of concurrent collection techniques when the mutator is running.
Why (some) customers care about MMU

- Typical application:

  ![Diagram](image-url)
Why (some) customers care about MMU

- Typical application:

Incoming task

queue

Application threads
Why (some) customers care about MMU

- Typical application:
Why (some) customers care about MMU

- If GC stops mutator thread, queue grows:
  - Max latency to last service last task (+ processing rate) determines real-time goal (max ms gc/ms elapsed).

![Diagram of incoming task queue and application threads]
Garbage-First Mechanisms

- Heap regions and rem sets.
- Evacuation pauses.
- Generational Garbage-First.
- Concurrent rem set maintenance.
- Concurrent marking.
- Collection set choice:
  - Garbage-First heuristic: choose best regions.
  - Pause-time modeling to meet real-time goal.
Garbage-First Heap Layout

HeapRegions

Garbage First!
Garbage-First Heap Layout
Garbage-First Heap Layout
Evacuation Pause

- Pick a *Collection Set* of regions.
- Evacuate live data elsewhere:
  - Live as indicated by roots, rem sets, previous marking information.
Evacuation Pause

- Pick a *Collection Set* of regions.
- Evacuate live data elsewhere:
  - Live as indicated by roots, rem sets, previous marking information.
Parallelism in Evac Pauses

- Based on [Flood+01]:
  - Static partitioning + fine-grained work-stealing for load-balancing.
  - (See also [Endo97].)
- Different phases combined to avoid barrier synchronizations (Flood).
Remembered Set Maintenance

- RemSet = hash tables of cards.
- Would be very expensive to update at each mutator pointer write!
- So the mutator logs updates, a concurrent remembered set thread reads the logs and updates the remembered sets.
- Mutator barrier filters for intra-region, NULL writes, writes to already-dirty cards.
- We must do RS updating concurrently, or pause times would be too long.
Generational Garbage-First

- Mutator allocation region can be declared *young*:
  - Will be included in the next collection set.
  - (Therefore) don't need to track pointer modifications.
  - But young regions are not physically segregated.

- Choose number of young regions to fit pause time (via dynamic feedback).
  - Might choose *not* to make a region young.
Concurrent Marking

- Eventually surviving objects fill up the heap (the “old gen”).
- Mark live objects concurrently.
  - We use snapshot-at-the-beginning [Yuasa90] marking (vs incremental update).
  - Some extra barrier overhead, but much shorter marking-related pause times, less total work.
    - Related work: static analysis to eliminate barriers (Detlefs&Nandivada)
- Marking also records live data in regions.
Marking, Evacuation, and Allocation

- Objects are efficiently “allocated black” by recording allocation points in all heap regions at start of marking.
- Evacuation can occur during (and independently of) a marking cycle:
  - Evacuation uses results of last completed marking cycle.
  - Evacuation preserves partial results of in-progress marking.
Using Marking Data

- Marking completes. Regions are labeled with live data (==> known garbage).
- Reclaim any completely-free regions (cleanup).

<table>
<thead>
<tr>
<th>50%</th>
<th>3%</th>
<th>0%</th>
<th>0%</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>15%</td>
<td>5%</td>
<td>25%</td>
</tr>
<tr>
<td>60%</td>
<td>80%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Using Marking Data

- Now continue allocation....

<table>
<thead>
<tr>
<th>50%</th>
<th>3%</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>15%</td>
<td>5%</td>
<td>25%</td>
</tr>
<tr>
<td>60%</td>
<td>80%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Using Marking Data

- Do collections as often as real-time spec allows.
- Pick a collection set:
  - Young + most-efficient non-young.

<table>
<thead>
<tr>
<th>50%</th>
<th>3%</th>
<th>100%</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>100%</td>
<td>25%</td>
</tr>
<tr>
<td>10%</td>
<td>15%</td>
<td>5%</td>
<td>25%</td>
</tr>
<tr>
<td>60%</td>
<td>80%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>


Using Marking Data

- (Hence the name: *Garbage-First!*)

![Using Marking Data](image)

Garbage First!

"Garbage First!"

(Hence the name: *Garbage-First!*)

<table>
<thead>
<tr>
<th>50%</th>
<th>3%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>15%</td>
<td>5%</td>
</tr>
<tr>
<td>60%</td>
<td>80%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Using Marking Data

- (Hence the name: \textit{Garbage-First!})
Using Marking Data

- (Hence the name: *Garbage-First!*)

<table>
<thead>
<tr>
<th></th>
<th>50%</th>
<th></th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15%</td>
<td></td>
<td>25%</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>80%</td>
<td>100%</td>
</tr>
</tbody>
</table>
How many non-young regions can we collect in $x$ ms?

Model predicts the cost of adding a heap region $r$ to a collection set:

- $C(r) = a \cdot r\.live\_data + b \cdot r\.rem\_set\_size + c$
- $a$, $b$, and $c$ are estimated initially, tracked dynamically.

Add regions in efficiency ($\text{garbage/cost}$) order until you 'overflow the bucket'.
A Garbage-First Execution

- Usually in fully-young mode.
- Switch to partially-young after marking
  - Harvest efficient regions.
- Switch back when efficiency drops.
Benchmarks

- "Telco"
  - A call setup application (SIP server).
  - Trying to get permission to tell you their name.
- SPECjbb
  - Java version of a TPC benchmark.
- Compare with ParNew+Concurrent MarkSweep
  - best Sun product solution for low pause time/high throughput.
MMU Specification Compliance

- **3 Metrics**
  - % of *timeslices* exceeding constraint (via sampling).
  - Avg % of excess.
  - Worst excess (as % of specified mutator time).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MMU</th>
<th>Garbage-First</th>
<th>CMS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>V%</td>
<td>AvgV%</td>
</tr>
<tr>
<td>SPECjbb</td>
<td>150/450</td>
<td>3.1%</td>
<td>2.3%</td>
</tr>
<tr>
<td>telco</td>
<td>75/225</td>
<td>0.2%</td>
<td>3.3%</td>
</tr>
</tbody>
</table>
Maximum Marking Pause

- SATB is very good at limiting marking-related pauses.

<table>
<thead>
<tr>
<th>App</th>
<th>Garbage-First</th>
<th>CMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECjbb</td>
<td>24 ms</td>
<td>935 ms</td>
</tr>
<tr>
<td>Telco</td>
<td>49 ms</td>
<td>382 ms</td>
</tr>
</tbody>
</table>
Throughput

- ~ Parity on telco.
- A little behind on SPECjbb
  - Working on it!

<table>
<thead>
<tr>
<th>App</th>
<th>Garbage-Flrst</th>
<th>CMS</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECjbb</td>
<td>27-30</td>
<td>32</td>
<td>Kops/sec</td>
</tr>
<tr>
<td>telco</td>
<td>1160-1170</td>
<td>1190</td>
<td>Calls/sec</td>
</tr>
</tbody>
</table>
Related Work

- Independent collection of heap regions:
  - [Bishop77], Train algorithm [Hudson&Moss92], MC^2 [Sachindren&Moss03]

- Concurrent Marking:
  - [Dijkstra+78], [Yuasa90], [Boehm+91], [Printezis&Detlefs00]

- Marking + Evacuation:
  - [Lang&Dupont87], [Ben-Yitzhak+02]
Related Work

- Oldest-First
  - [Stefanovic+02], [Hansen&Clinger02]

- Real-time GC
  - [Baker78], Metronome [Bacon+03], [Henriksson98]
Things I didn't have time to mention :-)

- Popular object handling
- “Pure” Garbage-First.
- Age segregation...
Future Work

- Fix remaining problems:
  - Rem set space overhead (Flood/Detlefs).
  - Make marking parallel as well as concurrent (Printezis).
  - Taking concurrent process overhead into account in MMU compliance (Printezis).

- Dynamic Pretenuring (Fabio Rojas, NU):
  - We claim it is well suited for Garbage-First:
    - Shorter, more predictable young-gen pauses.
    - Collect more via marking.
Questions?

david.detlefs@sun.com
tony.printezis@sun.com
**Snapshot-at-the-Beginning**

**Concurrent Marking**

- **Goal**: mark all objects reachable at the beginning of marking cycle.
- **Barrier for** $o.f = x$
- Logs $pre-val$ $o.f$

---

**Garbage First!**

---

**ISMM 04**
**Snapshot-at-the-Beginning**

Concurrent Marking

- **Goal:** mark all objects reachable at the beginning of marking cycle.
- **Barrier for** \( o.f = x \) logs \( \text{pre-val} \) \( o.f \)
- **Optimizations:**
  - Log only when marking in progress
  - Don't log NULL.