Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector

David F. Bacon
IBM T.J. Watson Research Center

Joint work with C.R. Attanasio, Han Lee, V.T. Rajan, and Steve Smith

ACM Conference on Programming Language Design and Implementation
Snowbird, Utah
June 20, 2001
Overview

• Two types of Garbage Collection
  – Tracing [McCarthy 1960]
  – Reference Counting [Collins 1960]

• Contributions of this work:
  – Improved concurrent reference counting algorithm
  – Practical cycle collection algorithm
    • Both stop-the-world and concurrent
  – Implementation in collector w/very low pause times
Features of the *Recycler*

- Concurrent
- Multiprocessor
- Reference-counted
- Cycle collecting
  - Low-latency: under 10 ms
  - High-performance: 90% of stop-the-world
Why do it?

• RC has good locality properties
  – Will tracing collection scale to multi-GB heaps?
  – Effect of rising memory latency
  – RC is easily decoupled from mutator
  – Promises lower synchronization costs

• Java makes good GC a commercial requirement
  – Makes sense to have “genetic diversity”

• People said it couldn’t be done
Why is it hard?

- Must defer reference counting stack
  - Complicates algorithm
- Reference counts are shared variables
  - Synchronization required
- Write barriers for RC more expensive
  - Affect two objects
- RC doesn’t handle cycles
  - Other systems use backup tracing collector
Outline

✓ Introduction
✓ Motivation
➢ System Overview
  • Concurrent Reference Counting
  • Cycle Collection
  • Measurements
  • Conclusions
System Overview

• Producer/Consumer System

- Mutator Emit inc/dec Allocate
- Collector Free Memory Process inc/dec Collect Cycles
- Mutator Emit inc/dec Allocate

• Similar to Deutsch-Bobrow DRC
  – As implemented by DeTreville on SRC Firefly
Implementation

• Implemented in Jalapeño JVM at IBM TJW
• All of VM, JIT, and GC are written in Java
  – Extended with unsafe primitives
  – Multiple GC implementations
  – GC is machine-independent except for barriers
• Targets
  – IBM RS/6000 multiprocessors (optimized)
  – Linux/Intel (unoptimized; optimized coming)
Outline

✓ Introduction
✓ Motivation
✓ System Overview

➢ Concurrent Reference Counting (no cycles)
  • Cycle Collection
  • Measurements
  • Conclusions
Concurrent Reference Counting

- Time divided into epochs
  - All CPUs must participate before epoch advances
- Write barrier on heap updates
  - inc/dec operations placed in buffer
  - Objects allocated with RC=1, dec enqueued
  - Decrements processed one epoch behind increments
- Stack references are deferred
  - Snapshot stacks at epoch boundary
  - First increment; decrement at next epoch
    - Simpler invariant than Deutsch-Bobrow; no ZCT required
Ragged Barriers

CPU 1
Mutator

CPU 2
Mutator

CPU 3
Collector

Epoch 7
Epoch 8
Epoch 9

Time
Outline

✓ Introduction
✓ Motivation
✓ System Overview
✓ Concurrent Reference Counting
   ➢ Cycle Collection
     • Measurements
     • Conclusions
Synchronous Cycle Collection

- Class loader identifies acyclic classes
  - Arrays, String, and such are marked green
- Two key observations:
  - Most reference counts are 1
  - Garbage cycles created by decrement to non-0
- Use those objects as starting points
  - DFS-based algorithm subtracts internal RC’s
  - If resulting count is 0, collect cyclic garbage
- Based on algorithm by Lins, but $O(n)$ instead of $O(n^2)$
1. Process Decrement and Accumulate Roots
2. Mark Gray: Subtract Internal Reference Counts
3. Scan: Restore Live, Mark Dead White
4. Collect White
Concurrent Cycle Collection

• Based on synchronous algorithm
  – But second reference count field is used
• Relies on stability property of garbage
  – If no mutation, synchronous algorithm will work
  – Detect when mutation occurs and avoid collecting
• When cycle is found
  – Place in a buffer, wait for next epoch
  – Inc/dec stream will provide necessary information
Detecting Concurrent Mutation

• Delta test
  – Detects local changes
  – Observes recolorings of buffered cycle objects

• Sigma test
  – Detects non-local changes
  – Checks sum of RC’s = number of pointers

• For details, see ECOOP’01 paper
Measurements

• Compared to parallel mark&sweep collector
  – High-performance, throughput-oriented
  – Stop-the-world

• Benchmarks:
  – SPEC, SPECjbb, Jalapeño compiler
  – Synthetic cycle garbage generator (ggauss)

• In MP mode, 1 more CPU than threads
Pause Time vs. Parallel Mark&Sweep

Max. Pause Time (us)

compress, jess, raytrace, db, javac, mpegaudio, mrt, jack, specjbb, jalapeno, ggauss

RC Pause
M&S Pause
Cycle Collection: Buffering Roots

Potential Roots

- Roots
- Unbuffered
- Freed
- Repeat
- Acyclic
Reference Tracing vs. Mark&Sweep

The chart compares reference tracing (RC) with mark&sweep (M&S) for various benchmarks. Each benchmark is represented by a bar, with the red bar showing RC and the green bar showing M&S. The y-axis represents the number of references traced in millions. The benchmarks includecompress, jess*, raytrace, db, javac*, mpegaudio, mtrt, jack, specjbb, jalapeno, and ggauss.
Related Work

- Dijkstra et al [1976], Steele [1975,1976], Lamport [1976]
- DeTreville [1990]
- Doligez, Leroy, Gonthier [1993,1994]
- Domani, Kolodner, Petrank [2000]
- Huelsbergen et al [1993,1999]
- Martínez et al [1990], Lins [1992]
- Plakal & Fischer [2001], Levanoni & Petrank [2001]
Conclusions

• Recycler sets new benchmark for concurrent GC
  – 2.6 ms max. pause time for general-purpose programs
  – End-to-end execution times comparable

• RC can perform very well in concurrent system
  – No synchronization required in common case
  – Standard RC problems can be overcome

• Concurrent cycle collection works
  – Viable alternative to backup mark & sweep
  – More study needed to reduce memory tracing costs
More Information

• The Recycler (including this presentation)
• Cycle Collection Algorithm/Proof (ECOOP’01)
  – www.research.ibm.com/people/d/dfb
• Jalapeño
• Jalapeño BoF Thursday 7pm
Addendum

• Algorithm Animations
  – Reference counting across multiple CPUs
  – Successful acyclic cyclic collection
  – Aborted acyclic cycle collection

• Additional Measurements
  – Collection cost breakdown
  – Objects freed by cycle collection
  – Allocation and mutation rates
1. Process Decrements
2. Mark Gray
3. Scan
4. Collect White
5. Await next epoch
6. If still orange, GC
7. If changed, restore
1. Process Decrement
2. Mark Gray
3. Scan
4. Collect White
5. Calculate external in-degree

6. Await next epoch
7. Compute in-degree sum
8. If 0, GC/decrement neighbors
9. If non-0, restore
Collection Cost Breakdown

The chart represents the collection cost breakdown for various applications, showing the percentage of time spent on different collection phases:

- **Free Cycles**
- **Validate**
- **Collect**
- **Scan**
- **Mark**
- **Purge**
- **Increment**
- **Decrement**

Each bar is divided into segments representing these phases, with the x-axis indicating different applications and the y-axis showing the percentage of collection time.
Objects Freed by Cycle Collection

- Objects Freed

- Graph showing percentages for various applications.
Mutation Rate (Barrier Frequency)

![Mutation Rate Graph](image)

- **K Inc/s**
- **K Dec/s**

- compress
- jess
- raytrace
- db
- javac
- mpegaudio
- mtr
- jack
- specjbb
- jalapeno
- ggauss

**K Mutations/sec**

<table>
<thead>
<tr>
<th>K Mutations/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>150</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>250</td>
</tr>
<tr>
<td>300</td>
</tr>
<tr>
<td>350</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>450</td>
</tr>
</tbody>
</table>

**Graph Description:**
- The graph shows the mutation rate (K Mutations/sec) for various applications.
- The x-axis represents different applications.
- The y-axis represents K Mutations/sec.
- The graph includes two bars for each application, one in blue (K Inc/s) and one in red (K Dec/s).
Allocation Rate

K Allocations/sec

compress  jess  raytrace  db  javac  mpegaudio  mtrt  jack  specjbb  jalapeno  ggauss