Older-First Garbage Collection in Practice: Evaluation in a Java Virtual Machine

Darko Stefanovic (Univ. of New Mexico)
Matthew Hertz (Univ. of Massachusetts)
Stephen M. Blackburn (Australian National Univ.)
Kathryn S. McKinley (Univ. of Texas Austin)
J. Eliot B. Moss (Univ. of Massachusetts)

ACM SIGPLAN Workshop on Memory System Performance
Berlin, 16 June 2002
New findings in copying garbage collection

- established a trade-off between copying cost and pointer-tracking cost
- the trade-off can be exploited effectively
- generational collectors: very low pointer-tracking cost
- older-first collector: very low copying cost
- total GC cost lower for older-first collector
- pause times also lower for older-first collector
Region-based copying garbage collection

Compute reachability of heap objects in a chosen region:

... then copy reachable objects over into a fresh to-space.
Actions in region-based copying garbage collection

- identifying global roots (stack scanning)
- identifying pointers across region boundaries
  - write barriers at pointer stores
  - recording pointers in remembered sets
  - processing remembered sets at GC time
- Cheney heap scanning — interleaves:
  - copying object bytes
  - identifying pointers in copied objects
  - transitively computing reachability
Age-based garbage collection

Choose regions according to *age*:

Generational collectors choose a *youngest* region. Older-first collector chooses a *middle-aged* region, processing the heap from oldest to youngest end.
When should older-first policy work well?

When the collected region stays in the middle:

Collection 1

Collection 2

Collection 3

Collection 4

Collection 5

Collection 6

Collection 7

Collection 8
Experimental setup

- first implementation of older-first collector
- JikesRVM virtual machine for Java bytecode [IBM]
- highly optimizing compiler
- GC Toolkit for JikesRVM [UMass]
- SPEC benchmarks for Java (SPECjvm98, SPEC JBB)
Implementation details

- all collectors share the same GC Toolkit infrastructure, except for write barriers
- original design of older-first collector called for large address space to use fast write barrier
- JikesRVM runs on 32-bit PowerPC
- emulate effect of large address space by indirection through age table (slower)
Write barriers (Java pseudocode)

Direct write barrier:
if (source < ((target >>> 28) << 28))
{
    ...remember pointer...
}

Indirect write barrier:
s = (source >>> 26);
t = (target >>> 26);
if (s != t
    && t >= HeapBoundary
    && agetable [s] > agetable [t])
{
    ...remember pointer...
}
Direct write barrier, used in generational collector

;; clear low-order 28 bits of pointer source:
rlwinm Rtemp, Rsource, 0x0, 0x0, 0x3
;; compare with pointer target:
cmp cr1, Rtarget, Rtemp
;; if comparison is favorable, skip remembering:
bge 1 label:do-not-remember-pointer
;; fall-through: remember pointer
Indirect write barrier, used in older-first collector

;; calculate frame numbers for source and target:
rlwinm Rtemp1, Rsource, 0x6, 0x1a, 0x1f
extsb Rtemp1, Rtemp1
rlwinm Rtemp2, Rtarget, 0x6, 0x1a, 0x1f
extsb Rtemp2, Rtemp2

;; intraframe pointers test:
cmp cr1, Rtemp1, Rtemp2
beq 1 label:do-not-remember-pointer

;; heap boundary test:
cmpi cr1, Rtemp2, 0xf
blt 1 label:do-not-remember-pointer
Indirect write barrier (cont’d)

;; load base of age table:
lwz Rtemp3, a-static-offset (JTOC)

;; look up age of source and target:
sli Rtemp1, Rtemp1, 0x2
lwzx Rtemp1, Rtemp3, Rtemp1
sli Rtemp2, Rtemp2, 0x2
lwzx Rtemp2, Rtemp3, Rtemp2

;; age comparison test:
cmp cr1, Rtemp1, Rtemp2
ble 1 label:do-not-remember-pointer

;; fall-through: remember pointer
Results: time spent in garbage collection

pseudojbb (Auto Best Config)

GC time relative to Appel

Heap size relative to minimum heap size (log)

ss

gen

of
Results: total execution time

pseudojbb (Auto Best Config)

Heap size relative to minimum heap size (log)

Total time relative to Appel

- ss: solid red line
- gen: dashed green line
- of: dotted blue line
# Copying vs. pointer-tracking

Sample run — pseudojbb, heap size 1.25 times minimum

<table>
<thead>
<tr>
<th>collector</th>
<th>Appel-style Gen.</th>
<th>Older-First 10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes allocated</td>
<td>667 million</td>
<td></td>
</tr>
<tr>
<td>bytes copied</td>
<td>221 million</td>
<td>117 million</td>
</tr>
<tr>
<td>mark/cons ratio</td>
<td>0.33</td>
<td>0.17</td>
</tr>
<tr>
<td>write barriers</td>
<td>98.2 million</td>
<td></td>
</tr>
<tr>
<td>pointers remembered</td>
<td>2.59 million</td>
<td>6.24 million</td>
</tr>
<tr>
<td>pointers processed</td>
<td>2.59 million</td>
<td>10.32 million</td>
</tr>
<tr>
<td>GC time</td>
<td>9.38s</td>
<td>5.15s</td>
</tr>
<tr>
<td>total execution time</td>
<td>45.15s</td>
<td>42.04s</td>
</tr>
</tbody>
</table>
What we haven’t done yet...

- write barrier (32-bit address space)
  - improve code sequence — estimate further
    2-3% reduction possible in total execution time
  - need to inject code lower than compiler’s HIR

- write barrier (64-bit address space)
  - porting JikesRVM to 64-bit PowerPC architecture
What we haven’t done yet... (cont’d)

- properly investigate locality effects
  - building tools: robust JikesRVM - SimpleScalar - PowerPC simulator, must support dynamic code generation, signals, system calls

- flexible policies for collection region
  - Appel collector (flexible) better than fixed generational
  - flexible older-first: need good policies
  - Beltway collector provides mechanism [PLDI]
Achievements

- shown trade-off between copying and pointer-tracking
- fulfilled promise of older-first GC
- always improve over fixed generational GC
- often improve over Appel-style generational GC
- low end-to-end times and pause times