Automated Reference-Counted Object Recycling for Real-Time Java

Morgan Deters
mdeters@cs.wustl.edu
Matthew P. Hampton
mph2@cs.wustl.edu

Nicholas A. Leidenfrost
n.leidenfrost@lancaster.ac.uk
James C. Brodman
jcb1@cs.wustl.edu

Ron K. Cytron
cytron@cs.wustl.edu

Washington University in St. Louis

www.cs.wustl.edu/~doc/

10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)

28 May 2004

Sponsored by DARPA under contract F33615–00–C–1697
Introduction

- Java can be nice
  - automatic memory management
  - increasing complexity and componentization...
Introduction

- Java can be nice
  - automatic memory management
  - increasing complexity and componentization...

- Java vs. real-time

  ```java
  new Foo();
  ```

  - how long does this take?
Introduction

- Java can be nice
  - automatic memory management
  - increasing complexity and componentization...

- Java vs. real-time
  
  ```java
  nuclearReactor.on();
  new Foo();
  nuclearReactor.off();
  ```
  - how long does this take?
Introduction

- Java can be nice
  - automatic memory management
    - increasing complexity and componentization...
- Java vs. real-time
  - `nuclearReactor.on();`
  - `new Foo();`
  - `nuclearReactor.off();`
    - how long does this take?
- Real-Time Specification for Java (RTSJ)
  - scoped memory regions
    - no garbage collection in regions
  - real-time threads can’t touch the garbage-collected heap
Context

- RTSJ memory model eschews garbage collection
  - but still no \texttt{free/delete}
- User-directed garbage detection
  - application-specific determination of dead objects
  - error-prone
- Aspect-enforced garbage detection
  - application-specific determination of dead objects
  - automatically-generated
  - \textit{not} error-prone
- Object recycling
  - per type/scope
Contributions

1. Aspect reference-counting
   - heap reference counting
   - stack reference counting approximation

2. Determination (static) of reference-countable classes

3. Aspect object recycling

4. Optimization of this aspect-generated code
Reference-Counted Object Recycling: Overview

P

Compile

RC–Suitability Analysis

RC Logic Stub Weaving

Type–Aware RC Weaving

(AspectJ)

(P')

(Clazzer)
Aspect-Oriented Programming (AOP)

- AspectJ

- classes and aspects
  - join points
  - advice

- classes provide services

- aspects enforce behaviors

\[ \forall j \in S \rightarrow A(j) \]
Reference-Counting

head

a

next

"a"

1

b

"simple"

1

c

"list"

1

null
Reference-Counting

```
head.next = null;
```

```
head
  ^
  |  next
  v
  "a"
  1

b
  ^
  |  next
  v
  "simple"
  1

c
  ^
  |  next
  v
  "list"
  1

RTAS 2004
```
Reference-Counting

```
ListItem temp = head.next;
head.next = null;
if(--temp.rc == 0)
    --temp.next.rc;
```
Reference-Counting

head

a

next

"a"

1

head.next = assign(head.next, null);
Reference-Counting

Error-prone
Type-specialized
Doesn’t use language features

head

a

"a"

1

head.next = assign(head.next, null);
Aspect Reference-Counting (Heap)

head

a

"a"

1

next

b

"simple"

1

c

"list"

1

null
Aspect Reference-Counting (Heap)

head

a

"a"

next

1

advice

trigger

b

"simple"

1

c

"list"

1

null

head.next = null;
Aspect Reference-Counting (Heap)

head

a

"a"

next

b

"simple"

1

update rc

0

head.next = null;

c

"list"

null
Aspect Reference-Counting (Heap)

```
head.next = null;
```

Update `rc`
Limitations of Heap-Only Reference-Counting

- Unsound for Java
  - if all references from the heap point away, the object is collected
- But a heap-only approach requires cheaper instrumentation
  - useful when referencing behavior is known for a class
Stack Reference-Counting

- Java stack heavily used
- Full accounting incredibly expensive

- Proposed: use an approximation
  - Associate each object to the *last-to-be-popped* stack frame that might reference it
  - When that stack frame pops (on a method return), the object is known to not have any stack references
Stack-Counting Approach

- When an object is created
  - associate to current frame

- When an unassociated object is pushed onto stack (e.g. getfield)
  - associate to current frame

- When an object is returned/thrown
  - associate to previous frame

- When a stack frame pops, consider all associated objects
  - if the heap reference-count is also zero, it’s dead
  - otherwise, dissociate from stack

- When a heap reference points away (e.g. putfield)
  - if object is unassociated to a stack frame, it’s dead

- Safe: a live object is never determined dead
Aspect Reference-Counting (Stack)
Aspect Reference-Counting (Stack)

```
a.next = null;
```
Aspect Reference-Counting (Stack)
Aspect Reference-Counting (Stack)

```
return;
```
Aspect Reference-Counting (Stack)

Reference-Counted Object Recycling: Stack Reference-Counting
Deters, Leidenfrost, Hampton, Brodman, & Cytron

RTAS 2004
28 May 2004
Aspect Reference-Counting (Stack)
Aspect Reference-Counting (Stack)

return;
Aspect Reference-Counting (Stack)
Instrumentation Details

heap

- putfield
- putstatic
- aastore

stack

All of the above, plus...

- new
- getfield
- getstatic
- aaload
- areturn (and other returns)
- athrow/propagate
P

Compile

RC–Suitability Analysis

RC Logic Stub Weaving

Type–Aware RC Weaving

P'

(AspectJ)

(Clazzer)
Recycling

- Can’t explicitly return object’s bytes to allocation engine
- So place on a per-type* list of free objects
- Advise new to check free object list

*if using RTSJ memory model, must satisfy scope constraints
Recycling in Action

new Foo();

- First, check list of available Foo objects
  - if one available, re-initialize, re-construct, and return it
  - otherwise, use standard Java allocation mechanism

- Behavior enforced via aspects – no source changes

- reset()
  - re-initializes and re-constructs object
  - automatically-generated from constructor
Reference-Counted Object Recycling: Overview

Deters, Leidenfrost, Hampton, Brodman, & Cytron

16

28 May 2004
Reference-Countable Types

- Empty marker interface `ReferenceCountable`
- Programmer may mark any type as `ReferenceCountable`
  - not useful if objects are involved in cycles
- We also provide an analysis to determine RC-suitability
What is Reference-Countable?

- Conservative, static analysis
- Let $CouldBe(x)$ be the set of types that a field $x$ may be
- Let $HasA(c)$ be the set of fields of class $c$ (and supers)

foreach class $c$

foreach type $t \in HasA(c)$

foreach type $u \in CouldBe(t)$

place edge in graph from $c$ to $u$

Any nodes not in a cycle are ReferenceCountable

Diagram:

- A
  - B
  - C
  - Δ
- D
  - b
  - Δ
  - E
    - a
    - c
  - F
    - b
    - a

BEF-RTAS 2004

19

28 May 2004
Reference-Counted Object Recycling: Overview

Deters, Leidenfrost, Hampton, Brodman, & Cytron

Compile

RC–Suitability Analysis

RC Logic Stub Weaving

Type–Aware RC Weaving

P

P'

(AspectJ)

(Clazzer)
Results (unoptimized)

Unoptimized heap- and stack-counting on simple benchmark

Execution time (ms)

Benchmark size

stack counting
heap-counting
GC
Optimizing the advice

- Unnecessary Java reflection
  - `reset()` (recycling construction)
  - process fields of dead objects

- A per-type specialized weaver doesn’t need the reflection

- Our implementation uses Clazzer

- AspectJ weaves in stubs
  - Code injector inserts targetted implementations

- Hints at aspect compiler optimizations
Reference-Counted Object Recycling: All Together

Deters, Leidenfrost, Hampton, Brodman, & Cytron

P

Compile

RC–Suitability Analysis

RC Logic Stub Weaving

Type–Aware RC Weaving

P'

AspectJ

Clazzer
Experimental Setup

- Linux desktop (kernel 2.6.6, 2.4GHz Xeon, JVM 1.4.1)
- paging disabled, SCHED_FIFO
- Compared performance to
  - standard Java garbage collector alone
  - incremental collector (\texttt{-Xincgc}) alone
- Benchmarks
  - simple linked-list benchmark
    - lots of garbage objects
  - SPEC jvm98 (compress, jess, raytrace, db)
GC vs. RC+GC \((10^5)\)
**GC vs. RC+GC** $(10^6)$

![Graph showing execution time vs. Java heap size for standard JVM+collector and RC+recycling. The graph indicates that RC+recycling has a lower execution time compared to the standard JVM+collector for all Java heap sizes tested.](image-url)
GC vs. RC+GC (# GC attempts)

![Graph showing GC vs. RC+GC](image)

- **standard 10^6**
- **RC/recycle 10^6**
- **standard 10^5**
- **RC/recycle 10^5**

Number of GC attempts vs. Java heap size (MB)

**Reference-Counted Object Recycling: Results: GC vs. aspect RC+GC**

Deters, Leidenfrost, Hampton, Brodman, & Cytron
GC vs. RC+GC (GC time spent)

![Chart showing GC time spent vs. Java heap size](chart.png)
Reference-Counted Object Recycling: Results: jess

Deters, Leidenfrost, Hampton, Brodman, & Cytron

jess

Mean execution time in seconds (log scale)

- Partial Evaluation
- Reflective
- Java GC

SPEC test size

1 10 100

0.15 15.70 0.09 1.58 0.62 221.12 231.59 45599.25 52.89

Washington University in St Louis
RTAS 2004

28 May 2004
Reference-Counted Object Recycling: Results: db

Deters, Leidenfrost, Hampton, Brodman, & Cytron

db

Mean execution time in seconds (log scale)

SPEC test size

<table>
<thead>
<tr>
<th>SPEC test size</th>
<th>Partial Evaluation</th>
<th>Reflective</th>
<th>Java GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.14</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>10</td>
<td>0.88</td>
<td>0.50</td>
<td>43.63</td>
</tr>
<tr>
<td>100</td>
<td>83.55</td>
<td>43.63</td>
<td>25.84</td>
</tr>
<tr>
<td>1000</td>
<td>4042.10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RTAS 2004
Heap vs. Stack RC

![Graph showing execution time (μsec) vs. benchmark size (N) for Stack reference counting and Heap-only reference counting. The graph indicates that Stack reference counting has a lower execution time compared to Heap-only reference counting.](image)
Other Observations

- GC attempts/cumulative time decreased dramatically
  - GC pause times unaffected
- Footprint
Future Work

- Further experimentation, experimentation platforms
  - utility in absence of GC
  - GC tuning
  - MMU
  - memory use experimentation
    - investigate where to collect/recycle
      - we recycle early (reference-counting)...
      - ...but we don’t return memory to JVM
    - if we had a delete mechanism...
- incremental recycling
Conclusions

- Reference-counting at user-level without manual instrumentation
- Automated
  - detection of reference-countable classes
  - instrumentation/optimization of the code
  - recycling of objects
- No source-level changes necessary
- Can be used in Java environments where GC is unavailable or undesirable
  - Real-Time Java
- Can be used to reduce reliance on JVM’s GC
- Performance comparable to garbage collection
Questions

Morgan Deters
mdeters@cs.wustl.edu

Washington University in St. Louis

10th IEEE Real-Time and Embedded Technology and Applications Symposium

28 May 2004