Biased Reference Counting: Minimizing Atomic Operations in Garbage Collection

Jiho Choi, Thomas Shull, Josep Torrellas
University of Illinois at Urbana-Champaign
November 3, 2018
Overview

- Paper Goal: reduce the overhead of reference counting
- Introduces Biased Reference Counting (BRC), a new algorithm which biases reference counting operations to a specific thread
  - No longer need to perform expensive atomic operations in the common case
- BRC improves performance by on average 22.5% for client applications
  - Improves throughput by on average 7.3% for server applications
Reference Counting (RC): technique for automatic memory management

Strategy: maintain per-object counters to determine each object’s liveness

RC(obj) > 0 \Rightarrow \text{alive}

RC(obj) = 0 \Rightarrow \text{dead: can be collected}

Allows for immediate reclamation of dead objects

Good for environments which require low memory overheads

Mobile

Embedded

Clouds with many concurrent VMs
• Reference Counting (RC): technique for automatic memory management
• Strategy: maintain per-object counters to determine each object’s liveness
• $\text{RC(obj)} > 0 \Rightarrow \text{alive}$
• $\text{RC(obj)} = 0 \Rightarrow \text{dead}: \text{can be collected}$
• Reference Counting (RC): technique for automatic memory management
• Strategy: maintain per-object counters to determine each object’s liveness
• $\text{RC}(\text{obj}) > 0 \Rightarrow \text{alive}$
• $\text{RC}(\text{obj}) = 0 \Rightarrow \text{dead: can be collected}$
• Allows for immediate reclamation of dead objects
Background – Reference Counting

- Reference Counting (RC): technique for automatic memory management
- Strategy: maintain per-object counters to determine each object’s liveness
  - $\text{RC}(\text{obj}) > 0 \Rightarrow \text{alive}$
  - $\text{RC}(\text{obj}) = 0 \Rightarrow \text{dead}: \text{can be collected}$
- Allows for immediate reclamation of dead objects
  - Good for environments which require low memory overheads
  - Mobile
  - Embedded
  - Clouds with many concurrent VMs
Key: Actively maintain reference count for each object throughout execution
Reference Counting Example

**Key:** Actively maintain reference count for each object throughout execution

\[ rc(obj_1) = 1 \]

\[ 1 \quad \text{var} \ a = \text{new} \ obj_1 \]
Reference Counting Example

Key: Actively maintain reference count for each object throughout execution

\[ rc(obj_1) = 1 \]

1. \( \text{var } a = \text{new } obj_1 \)
   \( rc(obj_1)++ \)

2. \( \text{var } b = a \)
   \( rc(obj_1)--, \ rc(obj_2) = 1 \)

3. \( b = \text{new } obj_2 \)
Reference Counting Example

**Key:** Actively maintain reference count for each object throughout execution

\[
rc(obj_1) = 1
\]

```javascript
1 var a = new obj_1
rc(obj_1)++
2 var b = a
rc(obj_1)--, rc(obj_2) = 1
3 b = new obj_2
rc(obj_1)--, rc(obj_3) = 1
free(obj_1)
4 a = new obj_3
```
• Problem: multiple threads may try to adjust an objects RC counter concurrently

procedure Increment(obj)
    old := obj.rc.read old value
    new := old += 1.set new value
    while !CAS(&obj.rc, old, new)
        Atomic update of counter
end procedure
Maintaining Thread Safety

- Problem: multiple threads may try to adjust an object's RC counter concurrently

**Initialization**

\[
rc(obj) = 1
\]

1. \( \text{var } g = \text{new } \text{obj}; \)

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( rc(obj)++ )</td>
<td>( rc(obj)++ )</td>
</tr>
<tr>
<td>( 2_A \text{ var } a = g; )</td>
<td>( 2_B \text{ var } b = g; )</td>
</tr>
</tbody>
</table>
Maintaining Thread Safety

- Problem: multiple threads may try to adjust an object's RC counter concurrently
- Solution: Use atomic Compare-And-Swap (CAS) Operations
Maintaining Thread Safety

- Problem: multiple threads may try to adjust an object's RC counter concurrently
- Solution: Use atomic Compare-And-Swap (CAS) Operations

```
1: procedureINCREMENT(obj)
2:   do
3:     old := obj.rc_counter  ▶ read old value
4:     new := old
5:     new += 1                 ▶ set new value
6:     while !CAS(&obj.rc_counter, old, new)  ▶ Atomic update of counter
7:       end procedure
```
Maintaining Thread Safety

- Problem: multiple threads may try to adjust an object's RC counter concurrently
- Solution: Use atomic Compare-And-Swap (CAS) Operations

1: procedure INCREMENT(obj)
2:   do
3:     old := obj.rc_counter  ▷ read old value
4:     new := old
5:     new += 1  ▷ set new value
6:   while !CAS(&obj.rc_counter, old, new)  ▷ Atomic update of counter
7:   end procedure
Problem: RC has significant performance overheads
Overheads of Reference Counting

- Problem: RC has significant performance overheads
- We evaluate Apple’s Swift implementation on client and server applications
  - Swift is the main language used for iOS development
  - Includes many optimizations to reduce the number of reference counting operations (RCOps) needed
    - Removes up to 97% of the RCOps
Problem: RC has significant performance overheads

We evaluate Apple’s Swift implementation on client and server applications

- Swift is the main language used for iOS development
- Includes many optimizations to reduce the number of reference counting operations (RCOps) needed
  - Removes up to 97% of the RCOps
- Separate RCOps time from rest of execution
Overheads of Reference Counting

The diagram shows the normalized execution time for various benchmarks and libraries. The y-axis represents the normalized execution time, ranging from 0.00 to 1.00. The x-axis lists different benchmarks and libraries, including Swift Bench, CryptoSwift, SwiftyJSON, Raytrace, GC-Single, GC-Multi, Regex-Redux, Client-Avg, Perfect-JSON, Perfect-JSON, Kitura-JSON, Kitura-Blog, and Server-Avg.

The bars are divided into two sections: Rest of Execution (light gray) and Reference Counting (dark gray). The chart illustrates the proportion of time spent in each category for each benchmark.

Choi et al. Biased Reference Counting
Overheads of Reference Counting

42% Overhead Client Applications
15% Overhead Server Applications
Compare-And-Swap (CAS) Operations are expensive

Choi et al. Biased Reference Counting
• Compare-And-Swap (CAS) Operations are expensive

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-atomic Increment</td>
<td>5.77</td>
</tr>
<tr>
<td>CAS Increment</td>
<td>13.84</td>
</tr>
</tbody>
</table>
• Compare-And-Swap (CAS) Operations are expensive
• Can reduce execution time by on average 25% by using non-atomic operations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-atomic Increment</td>
<td>5.77</td>
</tr>
<tr>
<td>CAS Increment</td>
<td>13.84</td>
</tr>
</tbody>
</table>
• Profiling: log each RCOp
• For each RCOp log entry, record
  • Which thread performed operation
  • Which object’s reference count was updated
• Log to use determine
  • % of objects whose RC counter is updated by only object’s initializing thread
  • % of RCOps performed by the object’s initializing thread
### Breaking Down Sharing Patterns of RC

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Objects</th>
<th>RC operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Priv (%)</td>
<td>Shar (%)</td>
</tr>
<tr>
<td>Swift Benchmark</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>CryptoSwift</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>SwiftyJSON</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Raytrace</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>GCBench-Single</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>GCBench-Multi</td>
<td>99.84</td>
<td>0.16</td>
</tr>
<tr>
<td>Regex-Redux</td>
<td>99.99</td>
<td>0.01</td>
</tr>
<tr>
<td>Average</td>
<td>99.98</td>
<td>0.02</td>
</tr>
<tr>
<td>Server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perfect-JSON</td>
<td>94.74</td>
<td>5.26</td>
</tr>
<tr>
<td>Perfect-Blog</td>
<td>94.58</td>
<td>5.42</td>
</tr>
<tr>
<td>Kitura-JSON</td>
<td>91.41</td>
<td>8.59</td>
</tr>
<tr>
<td>Kitura-Blog</td>
<td>91.59</td>
<td>8.41</td>
</tr>
<tr>
<td>Average</td>
<td>93.08</td>
<td>6.92</td>
</tr>
</tbody>
</table>

Choi et al. Biased Reference Counting
## Breaking Down Sharing Patterns of RC

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Objects</th>
<th>RC operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Priv (%)</td>
<td>Shar (%)</td>
</tr>
<tr>
<td></td>
<td>Priv (%)</td>
<td>Shar (%)</td>
</tr>
<tr>
<td>Client</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Benchmark</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>CryptoSwift</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>SwiftyJSON</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Raytrace</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>GCBench-Single</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>GCBench-Multi</td>
<td>99.84</td>
<td>0.16</td>
</tr>
<tr>
<td>Regex-Redux</td>
<td>99.99</td>
<td>0.01</td>
</tr>
<tr>
<td>Average</td>
<td>99.98</td>
<td>0.02</td>
</tr>
<tr>
<td>Server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perfect-JSON</td>
<td>94.74</td>
<td>5.26</td>
</tr>
<tr>
<td>Perfect-Blog</td>
<td>94.58</td>
<td>5.42</td>
</tr>
<tr>
<td>Kitura-JSON</td>
<td>91.41</td>
<td>8.59</td>
</tr>
<tr>
<td>Kitura-Blog</td>
<td>91.59</td>
<td>8.41</td>
</tr>
<tr>
<td>Average</td>
<td>93.08</td>
<td>6.92</td>
</tr>
</tbody>
</table>

Choi et al.  Biased Reference Counting
## Breaking Down Sharing Patterns of RC

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Client</th>
<th>Objects</th>
<th>RC operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Priv (%)</td>
<td>Shar (%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Priv (%)</td>
<td>Shar (%)</td>
</tr>
<tr>
<td>Swift Benchmark</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>CryptoSwift</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>SwiftyJSON</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>Raytrace</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>GCBench-Single</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>GCBench-Multi</td>
<td>99.84</td>
<td>0.16</td>
<td>99.68</td>
</tr>
<tr>
<td>Regex-Redux</td>
<td>99.99</td>
<td>0.01</td>
<td>51.13</td>
</tr>
<tr>
<td>Average</td>
<td>99.98</td>
<td>0.02</td>
<td>92.97</td>
</tr>
<tr>
<td>Server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perfect-JSON</td>
<td>94.74</td>
<td>5.26</td>
<td>83.99</td>
</tr>
<tr>
<td>Perfect-Blog</td>
<td>94.58</td>
<td>5.42</td>
<td>95.33</td>
</tr>
<tr>
<td>Kitura-JSON</td>
<td>91.41</td>
<td>8.59</td>
<td>84.29</td>
</tr>
<tr>
<td>Kitura-Blog</td>
<td>91.59</td>
<td>8.41</td>
<td>83.32</td>
</tr>
<tr>
<td>Average</td>
<td>93.08</td>
<td>6.92</td>
<td>86.73</td>
</tr>
</tbody>
</table>

Choi et al. Biased Reference Counting
## Breaking Down Sharing Patterns of RC

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Objects</th>
<th>RC operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Priv (%)</td>
<td>Shar (%)</td>
</tr>
<tr>
<td><strong>Client</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift Benchmark</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>CryptoSwift</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>SwiftyJSON</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Raytrace</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>GCBench-Single</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>GCBench-Multi</td>
<td>99.84</td>
<td>0.16</td>
</tr>
<tr>
<td>Regex-Redux</td>
<td>99.99</td>
<td>0.01</td>
</tr>
<tr>
<td>Average</td>
<td>99.98</td>
<td>0.02</td>
</tr>
<tr>
<td><strong>Server</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perfect-JSON</td>
<td>94.74</td>
<td>5.26</td>
</tr>
<tr>
<td>Perfect-Blog</td>
<td>94.58</td>
<td>5.42</td>
</tr>
<tr>
<td>Kitura-JSON</td>
<td>91.41</td>
<td>8.59</td>
</tr>
<tr>
<td>Kitura-Blog</td>
<td>91.59</td>
<td>8.41</td>
</tr>
<tr>
<td>Average</td>
<td>93.08</td>
<td>6.92</td>
</tr>
</tbody>
</table>

Choi et al. Biased Reference Counting
## Breaking Down Sharing Patterns of RC

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Objects</th>
<th>RC operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Priv (%)</td>
<td>Shar (%)</td>
</tr>
<tr>
<td></td>
<td>Priv (%)</td>
<td>Shar (%)</td>
</tr>
<tr>
<td><strong>Client</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift Benchmark</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>CryptoSwift</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>SwiftyJSON</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Raytrace</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>GCBench-Single</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>GCBench-Multi</td>
<td>99.84</td>
<td>0.16</td>
</tr>
<tr>
<td>Regex-Redux</td>
<td>99.99</td>
<td>0.01</td>
</tr>
<tr>
<td>Average</td>
<td>99.98</td>
<td>0.02</td>
</tr>
</tbody>
</table>

| **Server**       |         |               |
| Perfect-JSON     | 94.74   | 5.26          |
| Perfect-Blog     | 94.58   | 5.42          |
| Kitura-JSON      | 91.41   | 8.59          |
| Kitura-Blog      | 91.59   | 8.41          |
| Average          | 93.08   | 6.92          |

The table above shows the distribution of privatized and shared objects across different programs. The **Average** row provides the overall statistics for each category.
Introducing Biased Reference Counting (BRC)

- Observations: CASes are expensive and most reference count updates come from the object’s initializing thread.

- Solution: Bias reference counting operations.
  - Split original RC counter into two counters.
    - Biased Counter (BC): used exclusively by object’s initializing thread (Owner thread).
      - Does not use atomic operations.
    - Shared Counter (SC): used by all other threads.
      - Uses CASes like before.

We call our technique Biased Reference Counting (BRC).
Introducing Biased Reference Counting (BRC)

- Observations: CASes are expensive and most reference count updates come from the object’s initializing thread
- Solution: *Bias* reference counting operations
Introducing Biased Reference Counting (BRC)

- Observations: CASes are expensive and most reference count updates come from the object’s initializing thread
- Solution: *Bias* reference counting operations
- Split original RC counter into *two* counters
  - Biased Counter (BC): used exclusively by object’s initializing thread (Owner thread)
    - Does not use atomic operations
  - Shared Counter (SC): used by all other threads
    - Uses CASes like before
Introducing Biased Reference Counting (BRC)

• Observations: CASes are expensive and most reference count updates come from the object’s initializing thread
• Solution: *Bias* reference counting operations
• Split original RC counter into *two* counters
  • Biased Counter (BC): used exclusively by object’s initializing thread (Owner thread)
    • Does not use atomic operations
  • Shared Counter (SC): used by all other threads
    • Uses CASes like before
• We call our technique *Biased Reference Counting (BRC)*
New BRC RCWord

<table>
<thead>
<tr>
<th>Counter</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 bits</td>
<td>34 bits</td>
</tr>
</tbody>
</table>

Original RCWord.
New BRC RCWord

<table>
<thead>
<tr>
<th>Counter</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 bits</td>
<td>34 bits</td>
</tr>
</tbody>
</table>

Original RCWord.

<table>
<thead>
<tr>
<th>Biased</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>TID</td>
<td>Counter</td>
</tr>
<tr>
<td>18 bits</td>
<td>14 bits</td>
</tr>
</tbody>
</table>

BRC’s RCWord.
BRC Increment Example

1: procedure INCREMENT(obj)
2:  
   owner_tid := obj.rcword.biased.tid
3:   my_tid := GetThreadID()

4: if owner_tid == my_tid then
5:   FastIncrement(obj)
6: else
7:   SlowIncrement(obj)
8: end if
9: end procedure

10: procedure FastIncrement(obj)
11:  obj.rcword.biased.counter += 1
12: end procedure

13: procedure SlowIncrement(obj)
14:  do
15:     old := obj.rcword.shared
16:     new := old
17:     new.counter += 1
18:    while !CAS(&obj.rcword.shared, old, new)
19:  end procedure
1: `procedure Increment(obj)`
2: 
3:   `owner_tid := obj.rcword.biased.tid`
4: 
5:   `my_tid := GetThreadID()`
6: 
7:   if `owner_tid == my_tid` then
8:     `FastIncrement(obj)`
9:   else
10:     `SlowIncrement(obj)`
11: end if
12: end procedure
BRC Increment Example

1: `procedure Increment(obj)`
2:
   `owner_tid := obj.rcword.biased.tid`
3:   `my_tid := GetThreadID()`
4:   if `owner_tid == my_tid` then
5:       `FastIncrement(obj)`
6:   else
7:       `SlowIncrement(obj)`
8:   end if
9: `end procedure`

10: `procedure FastIncrement(obj)`
11:   `obj.rcword.biased.counter += 1`
12: `end procedure`
BRC Increment Example

```plaintext
1: procedure Increment(obj)  
2:   owner_tid := obj.rcword.biased.tid  
3:   my_tid := GetThreadID()  
4:   if owner_tid == my_tid then  
5:     FastIncrement(obj)  
6:   else  
7:     SlowIncrement(obj)  
8:   end if  
9: end procedure  

10: procedure FastIncrement(obj)  
11:   obj.rcword.biased.counter += 1  
12: end procedure  

1: procedure SlowIncrement(obj)  
2:   do  
3:     old := obj.rcword.shared  
4:     new := old  
5:     new.counter += 1  
6:     while !CAS(&obj.rcword.shared, old, new)  
7:   end procedure
```

Choi et al. Biased Reference Counting
BRC Increment Example

1: `procedure Increment(obj)`
2: 
   `owner_tid := obj.rcword.biased.tid`
3: `my_tid := GetThreadID()`
4: `if owner_tid == my_tid then`
5:     `FastIncrement(obj)`
6: `else`
7:     `SlowIncrement(obj)`
8: `end if`
9: `end procedure`

10: `procedure FastIncrement(obj)`
11:     `obj.rcword.biased.counter += 1`
12: `end procedure`

1: `procedure SlowIncrement(obj)`
2:     `do`
3:         `old := obj.rcword.shared`
4:         `new := old`
5:         `new.counter += 1`
6:     `while !CAS(&obj.rcword.shared, old, new)`
7:     `end do`
8:     `end procedure`
• Complication: with two counters, extra steps are needed to determine if an object is alive or not
  • Must add together BC and SC
  • Only owner thread can accurately read BC
Determining an Object’s Liveness

- Complication: with two counters, extra steps are needed to determine if an object is alive or not
  - Must add together BC and SC
  - Only owner thread can accurately read BC
- Solution: BRC introduces two RC counter states
Determining an Object’s Liveness

- Complication: with two counters, extra steps are needed to determine if an object is alive or not
  - Must add together BC and SC
  - Only owner thread can accurately read BC
- Solution: BRC introduces two RC counter states
  - *Biased*: both counters are being used
  - *Merged*: only the shared counter is being used
Determining an Object’s Liveness

• Complication: with two counters, extra steps are needed to determine if an object is alive or not
  • Must add together BC and SC
  • Only owner thread can accurately read BC
• Solution: BRC introduces two RC counter states
  • Biased: both counters are being used
  • Merged: only the shared counter is being used
• Lifetime Process: start out in biased state, later move to merged state
• Once in the merged state, maining the RC counter is the same as before
Merging Counters

- Transferring to *merged* state involves merging the two counters
  - Merge operation must be performed by owner thread

More details in paper
Choi et al. Biased Reference Counting
Merging Counters

- Transferring to *merged* state involves merging the two counters
  - Merge operation must be performed by owner thread
- Can transfer to the *merged* state both implicitly and explicitly

More details in paper Choi et al. Biased Reference Counting
Merging Counters

- Transfering to *merged* state involves merging the two counters
  - Merge operation must be performed by owner thread
- Can transfer to the *merged* state both implicitly and explicitly
  - *Implicit Merge*: merge counters when BC = 0
    - No arithmetic necessary
    - CAS is needed to set the *merged* flag

Choi et al. Biased Reference Counting
Merging Counters

- Transfering to *merged* state involves merging the two counters
  - Merge operation must be performed by owner thread
- Can transfer to the *merged* state both implicitly and explicitly
  - *Implicit Merge*: merge counters when BC = 0
    - No arithmetic necessary
    - CAS is needed to set the *merged* flag
  - *Explicit Merge*: non-owner thread requests the counters to be merged
    - Occurs when shared counter become negative
    - Can happen with the use of global shared variables
    - Counters are merged by the owner thread at a later point

More details in paper Choi et al. Biased Reference Counting
Merging Counters

- Transferring to *merged* state involves merging the two counters
  - Merge operation must be performed by owner thread
- Can transfer to the *merged* state both implicitly and explicitly
- *Implicit Merge*: merge counters when $BC = 0$
  - No arithmetic necessary
  - CAS is needed to set the *merged* flag
- *Explicit Merge*: non-owner thread requests the counters to be merged
  - Occurs when shared counter become negative
  - Can happen with the use of global shared variables
  - Counters are merged by the owner thread at a later point

More details in paper
Example – Private Object

RCWord Structure
Example – Private Object

Thread ID | Shared Counter
---|---
TID | BC | SC | M

Biased Counter | Merged Flag

RCWord Structure

T1 allocates an object
Example – Private Object

RCWord Structure

T1 allocates an object

T1 creates N references
Example – Private Object

<table>
<thead>
<tr>
<th>TID</th>
<th>BC</th>
<th>SC</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

T1 allocates an object

<table>
<thead>
<tr>
<th>T1</th>
<th>N</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

T1 creates N references

<table>
<thead>
<tr>
<th>T1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

T1 removes all references

T1 deallocates object
Example – Shared Object

RCWord Structure

Thread ID
Shared Counter

TID | BC | SC | M

Biased Counter
Merged Flag
Example – Shared Object

RCWord Structure

<table>
<thead>
<tr>
<th>TID</th>
<th>BC</th>
<th>SC</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

T1 allocates an object
Example – Shared Object

RCWord Structure

Thread ID | Shared Counter
---|---
TID | BC | SC | M

T1 allocates an object:

| T1 | 1 | 0 | 0 |

T2 creates a reference:

| T1 | 1 | 1 | 0 |
Example – Shared Object

RCWord Structure

<table>
<thead>
<tr>
<th>TID</th>
<th>BC</th>
<th>SC</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

T1 allocates an object

| T1  | 1  | 1  | 0  |

T2 creates a reference

| 0   | 1  | 1  |

T1 removes its reference
T1 implicitly merges
Example – Shared Object

Thread ID | Shared Counter | Biased Counter | Merged Flag
--- | --- | --- | ---
T1 | 1 | 0 | 0

T1 allocates an object

T1 | 1 | 1 | 0

T2 creates a reference

0 | 1 | 1

T1 removes its reference
T1 implicitly merges

0 | 0 | 1

T2 removes its reference
T2 deallocates object

RCWord Structure
Evaluation Setup

• Add BRC to Apple’s Swift implementation
• Configurations
  • Original (O): unmodified Swift runtime
  • Ideal (I): O with no atomic operations
    • Is not thread-safe
  • Biased (B): O enhanced with BRC

Choi et al. Biased Reference Counting
Evaluation Setup

- Add BRC to Apple’s Swift implementation
- Configurations
  - Original \((O)\): unmodified Swift runtime
  - Ideal \((I)\): \(O\) with no atomic operations
    - Is not thread-safe
  - Biased \((B)\): \(O\) enhanced with BRC
- Benchmarks
  - Client-Side: measure performance time improvements for both single and multi-threaded applications
  - Server-Side: use the Perfect and Kitura frameworks
## Evaluation – Operation Performance

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Counter</td>
<td>13.84</td>
</tr>
<tr>
<td>Ideal Counter</td>
<td>5.77</td>
</tr>
<tr>
<td>Biased Counter</td>
<td>6.28</td>
</tr>
<tr>
<td>Shared Counter</td>
<td>15.57</td>
</tr>
</tbody>
</table>
Evaluation – Application Performance

The graph shows normalized execution time for various applications under different conditions:
- **Original**
- **Ideal**
- **Biased**

Applications include:
- Swift Bench
- CryptoSwift
- SwiftyJSON
- Raytrace
- GC-Single
- GC-Multi
- Regex-Redux
- Average

The x-axis represents different applications, while the y-axis shows normalized execution time.
Evaluation – Application Performance

Average speedup: 22.5%
Evaluation – Application Performance

Average speedup: 7.3%
Putting BRC in Context

<table>
<thead>
<tr>
<th>Execution Time Overhead</th>
<th>Memory Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

- Naive non-deferred RC
- Optimized non-deferred RC
- Manual
- Deferred RC
- Tracing GC
Putting BRC in Context

<table>
<thead>
<tr>
<th>Execution Time Overhead</th>
<th>Memory Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Naive non-deferred RC</td>
<td></td>
</tr>
<tr>
<td>Optimized non-deferred RC</td>
<td></td>
</tr>
<tr>
<td>BRC</td>
<td></td>
</tr>
<tr>
<td>Deferred RC</td>
<td></td>
</tr>
<tr>
<td>Tracing GC</td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td></td>
</tr>
</tbody>
</table>

Choi et al.  Biased Reference Counting
Summary

Paper makes the following contributions:

- Characterizes the overhead of reference counting in Swift programs
- Characterizes the memory behavior and sharing patterns of Swift programs
- Identifies that most RC operations are performed by the object’s initializing thread
- Proposes Biased Reference Counting (BRC), a new algorithm for RC
  - Removes the need for CAS operations in the common case
- Implements BRC in Apple’s Swift Runtime
- Finds BRC improves client application performance by 22.5%
  - Improves server application throughput by 7.3%
Thank You

Questions?
Backup Slides
We did evaluate the performance of rebaising

- Rebiasing to next access after merge
- Minimal effect on performance
  - $< 1\%$ Slowdown
  - Original owner tends to not give up ownership
  - Overhead in attaining ownership
Evaluation – Sensitivity Study

Until 75% of objects are queued
BRC is better

Until 75% of objects are queued
BRC is better

Choi et al. Biased Reference Counting
### Evaluation – Behavior

<table>
<thead>
<tr>
<th>Program Name</th>
<th>% of Shared Obj.</th>
<th>Obj. Allocs. per μs</th>
<th>RC Ops. per μs</th>
<th>RC Ops. per Obj.</th>
<th>% of RC Ops. to Shared Obj.</th>
<th>% of RC Ops. to Biased Counter</th>
<th>% of RC Ops. to Shared Counter</th>
<th>% of Queued Obj.</th>
<th>% of Queued Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift</td>
<td>0.00</td>
<td>1.91</td>
<td>29.18</td>
<td>15.24</td>
<td>33.16</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Benchmark</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CryptoSwift</td>
<td>0.00</td>
<td>2.39</td>
<td>56.54</td>
<td>23.61</td>
<td>68.16</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>SwiftyJSON</td>
<td>0.00</td>
<td>2.60</td>
<td>68.19</td>
<td>26.24</td>
<td>93.77</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Raytrace</td>
<td>0.00</td>
<td>0.00</td>
<td>101.70</td>
<td>27258.42</td>
<td>173.94</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>GCBench-Single</td>
<td>0.00</td>
<td>12.87</td>
<td>64.07</td>
<td>4.98</td>
<td>86.03</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>GCBench-Multi</td>
<td>0.16</td>
<td>50.69</td>
<td>150.39</td>
<td>2.97</td>
<td>195.44</td>
<td>0.32</td>
<td>99.97</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>Regex-Redux</td>
<td>0.01</td>
<td>2.58</td>
<td>92.61</td>
<td>35.91</td>
<td>123.99</td>
<td>48.87</td>
<td>88.02</td>
<td>11.98</td>
<td>0.00</td>
</tr>
<tr>
<td>Average</td>
<td>0.02</td>
<td>10.44</td>
<td>80.38</td>
<td>39338.19</td>
<td>110.64</td>
<td>7.03</td>
<td>84.00</td>
<td>1.72</td>
<td>0.00</td>
</tr>
<tr>
<td><strong>Server</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perfect-JSON</td>
<td>5.26</td>
<td>0.55</td>
<td>4.56</td>
<td>8.25</td>
<td>4.95</td>
<td>16.01</td>
<td>88.64</td>
<td>11.36</td>
<td>1.99</td>
</tr>
<tr>
<td>Perfect-Blog</td>
<td>5.42</td>
<td>0.45</td>
<td>12.45</td>
<td>27.95</td>
<td>12.90</td>
<td>4.67</td>
<td>96.72</td>
<td>3.28</td>
<td>1.95</td>
</tr>
<tr>
<td>Kitura-JSON</td>
<td>8.59</td>
<td>0.40</td>
<td>7.27</td>
<td>18.05</td>
<td>7.55</td>
<td>15.71</td>
<td>87.38</td>
<td>12.62</td>
<td>2.76</td>
</tr>
<tr>
<td>Kitura-Blog</td>
<td>8.41</td>
<td>0.39</td>
<td>6.12</td>
<td>15.81</td>
<td>6.34</td>
<td>16.68</td>
<td>86.68</td>
<td>13.32</td>
<td>2.70</td>
</tr>
<tr>
<td>Average</td>
<td>6.92</td>
<td>0.45</td>
<td>7.63</td>
<td>17.51</td>
<td>7.93</td>
<td>13.27</td>
<td>89.85</td>
<td>10.15</td>
<td>2.35</td>
</tr>
</tbody>
</table>

**Table 7: Reference counting statistics.**