Concurrent Reference Counting

Or Ostrovsky

13/1/15
The next 80 minutes in one slide

- Goal - Applying reference counting to multithreaded scenarios
- Try use it out of the box, fail and understand why
- Improve until we get it right
  - Lock the whole object
  - Lock a part of the object
  - Log changes locally and apply them globally
  - Snapshot the heap concurrently
  - Use single-threaded algorithms on the snapshot
# Table of contents

1. Introduction  
2. Simple Reference Counting  
3. Buffered Reference Counting  
4. Sliding View Reference Counting  
5. An Efficient On-the-Fly Cycle Collection  
6. Summery
Ground rules

The invariant
An object’s reference count is equal to the number of references to that object.

The weaker invariant
- If an object has references, it’s reference count won’t be 0.
- If an object doesn’t have references, it’s reference count will eventually be 0.

Disclaimer
We are not concerned about mutator correctness. Only the heap matters!
The old algorithm

Algorithm 1 Non-concurrent reference counting operations

1: Write(o, i, ref):
2:   addReference(ref)
3:   old ← o[i]
4:   deleteReference(old)
5:   o[i] ← ref
6: 
7: Read(o, i):
8:   ref ← o[i]
9:   addReference(ref)
10: return ref
Faces new problems

Thread 1 Write(o, i, x)  Thread 2 Write(o, i, y)

old 1       o[i]       old 2

0           1           0
x           z           y
Faces new problems

<table>
<thead>
<tr>
<th>Thread 1 Write(o,i,x)</th>
<th>Thread 2 Write(o,i,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addReference(x)</td>
<td>addReference(y)</td>
</tr>
</tbody>
</table>

\[
\text{o}[i] \leftarrow x \quad \text{o}[i] \leftarrow y
\]

Or Ostrovsky
Concurrent Reference Counting
Faces new problems

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Write(o, i, x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addReference(x)</td>
<td></td>
</tr>
<tr>
<td>old ← o[i]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 2</th>
<th>Write(o, i, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addReference(y)</td>
<td></td>
</tr>
<tr>
<td>old ← o[i]</td>
<td></td>
</tr>
</tbody>
</table>

```
old 1
   └── 1
     └── x

o[i]
  └── 1
    └── z

old 2
  └── 1
    └── y
```
**Faces new problems**

<table>
<thead>
<tr>
<th>Thread 1 Write(o,i,x)</th>
<th>Thread 2 Write(o,i,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addReference(x)</td>
<td>addReference(y)</td>
</tr>
<tr>
<td>old ← o[i]</td>
<td>old ← o[i]</td>
</tr>
<tr>
<td>deleteReference(old)</td>
<td>deleteReference(old)</td>
</tr>
</tbody>
</table>

```
old 1

1
x

old

o[i]

-1
z

old 2

1
y
```

Or Ostrovsky  
Concurrent Reference Counting  
13/1/15
Faces new problems

<table>
<thead>
<tr>
<th>Thread 1 Write(o,i,x)</th>
<th>Thread 2 Write(o,i,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addReference(x)</td>
<td>addReference(y)</td>
</tr>
<tr>
<td>old ← o[i]</td>
<td>old ← o[i]</td>
</tr>
<tr>
<td>deleteReference(old)</td>
<td>deleteReference(old)</td>
</tr>
<tr>
<td>o[i] ← x</td>
<td>o[i] ← y</td>
</tr>
</tbody>
</table>

```
old  1
  ▼
 x

old  2
  ▼
 y
```

or x - 1 and y + 1
Faces new problems

<table>
<thead>
<tr>
<th>Thread 1 Write(o,i,x)</th>
<th>Thread 2 Write(o,i,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addReference(x)</td>
<td>addReference(y)</td>
</tr>
<tr>
<td>old ← o[i]</td>
<td>old ← o[i]</td>
</tr>
<tr>
<td>deleteReference(old)</td>
<td>deleteReference(old)</td>
</tr>
<tr>
<td>o[i] ← x</td>
<td>o[i] ← y</td>
</tr>
</tbody>
</table>

```
old 1
  o[i]
    1
    x

old 2
  o[i]
    -1
    z
```

Or Ostrovsky
Concurrent Reference Counting
Let’s break it down

- Updating a reference count requires:
  - Update the Pointer
  - Decrement the old target’s ref. count
  - Increment the new target’s ref. count
  - All must be coordinated!

- Pitfalls:
  - Premature release
  - Garbage remaining indefinitely
Table of contents

1. Introduction

2. Simple Reference Counting
   - Locks
   - Atomic Memory Primitives

3. Buffered Reference Counting

4. Sliding View Reference Counting

5. An Efficient On-the-Fly Cycle Collection

6. Summary
Motivation

Problem
Multiple threads are accessing the same object at the same time.
<table>
<thead>
<tr>
<th><strong>Problem</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple threads are accessing the same object at the same time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Solution</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow only one thread to access each object at a given time.</td>
</tr>
</tbody>
</table>
Motivation

Problem
Multiple threads are accessing the same object at the same time.

Solution
Allow only one thread to access each object at a given time.

Method
Protect the object using locks
The old algorithm revisited

Algorithm 2 Eager reference counting with locks

1: Read(src, i):
2:    lock(src)
3:    tgt ← src[i]
4:    addReference(tgt)
5:    unlock(src)
6:    return ref
7:  
8: Write(src, i, ref):
9:    addReference(ref)
10:   lock(src)
11:   old ← src[i]
12:   src[i] ← ref
13:   deleteReference(old)
14:   unlock(src)
Analysis

Pro.

- Simple
- Works

Con.

- Performance - locks the whole object
Motivation

Problem

Locking the whole object impedes performance.
Motivation

Problem

Locking the whole object impedes performance.

Solution

Lock the bare minimum needed.
Motivation

Problem
Locking the whole object impedes performance.

Solution
Lock the bare minimum needed.

Method
Use atomic memory primitives.
The same algorithm, but with a twist - writing

**Algorithm 3** Eager reference counting with CompareAndSwap - write

1: Write(src, i, ref)
2:   if ref ≠ null
3:     AtomicIncrement(&rc(ref))                /*rc=reference counter*/
4:   loop
5:     old ← src[i]
6:     if CompareAndSet(&src[i], old, ref)
7:       deleteReference(old)
8:     return
9: 10: deleteReference(ref)
11:  if ref ≠ null
12:    AtomicDecrement(&rc(ref))
13:    if rc(ref) = 0
14:      for each fld in Pointers(ref)
15:        deleteReference(*fld)
16:      free(ref)
The same algorithm, but with a twist

Algorithm 4 Eager reference counting with CompareAndSwap - reading

1:   Read(src, i, ref)
2:       tgt ← src[i]
3:   AtomicIncrement(&rc(tgt))
4:   return tgt
The same algorithm, but with a twist

**Algorithm 5** Eager reference counting with CompareAndSwap - reading

1. Read(src, i, ref)
2. tgt ← src[i]
   
   free(tgt)
3. AtomicIncrement(&rc(tgt))
4. return tgt
The same algorithm, but with a twist

**Algorithm 6** Eager reference counting with CompareAndSwap2

1:  Read(src, i, ref)
2:  loop
3:    tgt ← src[i]
4:    if tgt = null
5:      return null
6:    rc ← rc(tgt)
7:    if CompareAndSet2(&src[i], &rc(tgt), tgt, 
                    rc, tgt, rc+1)
8:      return tgt
Analysis

Pro.
- Simple(ish)
- Better performance - block fields and not objects

Con.
- Not practical - no CompareAndSet2
- Performance issues:
  - Locks
  - Spin locks are bad
Motivation

Problem

Heavy write barrier
Motivation

Problem
Heavy write barrier

Solution
Let another thread do all of the accounting
Motivation

Problem
Heavy write barrier

Solution
Let another thread do all of the accounting

Method
Buffer all the ref. counting operations
Implementation

- Each mutator logs its operations using a write barrier.
- Each cycle:
  - Collect logs from each thread
  - Apply reference count updates
  - Reclaim memory
- Time is divided into epochs.
- Deferred ref. counting is used.
The new write barrier

**Algorithm 7** Concurrent buffered reference counting - write

1. shared epoch
2. shared updatesBuffer[] /*one per epoch*/
3:
4. Write(src, i, ref)
5. if src = Roots
6. src[i] ← ref
7. else
8. old ← AtomicExchange(&src[i], ref)
9. log(old, ref)
10:
11. log(old, new)
12. append(myUpdates, ⟨old, new⟩)
Collect - get local logs

Algorithm 8 Concurrent buffered reference counting - collect

1: collect()
2: myStackBuffer ← []
3: for each local ref in myStacks
4:   append(myStackBuffer, ⟨ref, ref⟩)
5: atomic
6:   append(updatesBuffer[e], myStackBuffer)
7: atomic
8:   append(updatesBuffer[e], myUpdates)
9: myUpdates ← []
10: e ← e + 1
Collect - apply changes

**Algorithm 8** Concurrent buffered reference counting - collect

```
11:   me ← myProcessorId
12:   if me < MAX_PROCESSORS
13:       schedule(collect, me+1)
14:   else
15:       for each ⟨old, new⟩ in updatesBuffer[epoch]
16:           addReference(new)
17:       for each ⟨old, new⟩ in updatesBuffer[epoch - 1]
18:           deleteReference(old)
19:       release(updatesBuffer[epoch - 1])
20:   epoch ← epoch + 1
```
Example

Or Ostrovsky
Concurrent Reference Counting
13/1/15 24/57
Example
Example

Or Ostrovsky
Concurrent Reference Counting
13/1/15 24 / 57
Example
Example

Or Ostrovsky
Concurrent Reference Counting
13/1/15 24 / 57
Example

Concurrency Reference Counting

Or Ostrovsky

Concurrent Reference Counting

13/1/15 24 / 57
Example

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e-1</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
</tr>
</tbody>
</table>
```

Or Ostrovsky  
Concurrent Reference Counting
Example

Or Ostrovsky
Concurrent Reference Counting
13/1/15 24 / 57
Example

\begin{align*}
T1 & \quad A \\
T2 & \quad B \\
\end{align*}

\begin{align*}
A & \quad A \\
B & \quad B \\
\end{align*}

\begin{align*}
e & \\
e+1 & \\
\end{align*}

Or Ostrovsky

Concurrent Reference Counting

13/1/15  24 / 57
Example

T1

T2

A 3
B 2

A A A A B
T1 T2

B B A A
T1 T2

A A
T1

Or Ostrovsky
Concurrent Reference Counting
Example

\begin{figure}
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Example diagram showing concurrent operations.}
\end{figure}
Example

Or Ostrovsky
Concurrent Reference Counting
Example
Analysis

Pro.
- Performance
  - Thin write barrier
  - On the fly
  - Allows for an independent collector thread
  - Incorporates deferred ref. counting

Con.
- Circle handling
- High memory requirements
- Log each operation
Table of contents

1 Introduction

2 Simple Reference Counting

3 Buffered Reference Counting

4 Sliding View Reference Counting
   - Coalesced Reference Counting
   - Sliding View

5 An Efficient On-the-Fly Cycle Collection

6 Summery
Motivation

Problem
Can’t use single-threaded algorithms concurrently
Motivation

Problem
Can’t use single-threaded algorithms concurrently

Solution
Create a snapshot of the heap
Motivation

**Problem**
Can’t use single-threaded algorithms concurrently

**Solution**
Create a snapshot of the heap

**Method**
Create a sliding view
An observation:

\[
\begin{align*}
src[i] & \leftarrow o_1 & rc(o_0) & --, rc(o_1) & ++ \\
src[i] & \leftarrow o_2 & rc(o_1) & --, rc(o_2) & ++ \\
& \ldots & \ldots & \ldots & \ldots \\
src[i] & \leftarrow o_n & rc(o_{n-1}) & --, rc(o_n) & ++ 
\end{align*}
\]

Redundancy?
Coalesced reference counting - solution

- Method:
  - Write barrier: log “clean” objects and mark them as “dirty”.
  - At the end of each cycle: $rc(log(ref)) --, rc(ref)++$

- Uses a snapshot of the heap - provided by the logs.

- The increments require the mutator to stop.

- Implementation
  - Add a pointer to the object header
  - The pointer points to the current log entry
  - Clean objects: $getLogPointer(obj) = \textbf{null}$
Example

Colletor’s log

A

B

C
Example

Colletor’s log

A

B
C

Or Ostrovsky
Concurrent Reference Counting
13/1/15 30 / 57
Example
Levanoni and Petrank 2001

Take an incremental, concurrent “snapshots” of the heap
  ▶ Each mutator logs writes on clean objects.
  ▶ The collector collects the logs on mutator at a time
  ▶ The collector cleans dirty objects
  ▶ When the collector needs a object current heap
    ★ If it’s clean - use it
    ★ Else - use a local log
Final Result

- Sliding view - taken at \([t_s, t_e]\)
- A spread “snapshot” - \(state_t(obj)\) s.t. \(t \in [t_s, t_e]\)
Snooping

- Problem - premature reclaims
  - $ref$ might be reclaimed

- Solution - “snoop” it
  - Keep it in a special buffer
  - Use it as a root in the current cycle
Other Problems

- Global log pointers
  - A race is possible
  - Solution: The race has no effect.

- Clean-modify conflicts
  - The mutator changes the object while it’s being cleaned
  - Solution: Reinforce dirty state if needed
Building the local logs

**Algorithm 9 Write barrier**

```plaintext
1: Write(src, i, ref)
2:   if src = Roots
3:     src[i] ← ref
4:   else
5:     if not dirty(src)
6:       log(src)
7:     src[i] ← ref
8:     snoop(ref)
9: 
10: log(ref)
11:   for each fld in Pointers(ref)
12:     if *fld ≠ null
13:       add(logs[me], *fld)
14:     if not dirty(ref)
15:       entry ← add(logs[me], ref)
16:       logPointer(ref) ← entry
17: 
18: snoop(ref)
19:   if snoopFlag[me] && ref ≠ null
20:     append(mySnoopedBuffer, ref)
```
Collect them

Algorithm 10 Sliding view collection

1: collectSlidingView()
2: for each t in threads /*handshake 1*/
3: suspend(t)
4: snoopFlag[t] ← true
5: transfer t’s buffers to updates
6: resume(t)
7: clean modified and young objects
8: for each t in threads /*handshake 2*/
9: suspend(t)
10: find modify-clean conflicts
11: resume(t)
12: reinforce dirty objects
13: for each t in threads /*handshake 3*/
14: suspend(t)
15: resume(t)
Use them

Algorithm 11 Read sliding view

1: readSlidingView(entry)
2: \hspace{1cm} obj ← objFromLog(entry)
3: \hspace{1cm} if not dirty(obj)
4: \hspace{2cm} replica ← copy(obj) \hspace{1cm} /*read the current state*/
5: \hspace{1cm} if dirty(obj)
6: \hspace{2cm} replica ← getLogPointer(obj)
7: \hspace{1cm} else
8: \hspace{2cm} replica ← getLogPointer(obj)
9: \hspace{1cm} return replica
Example

Mutators's log

Collector's log

A

B

C

Mutators's log

Snoop buffer
Example

Collector's log

Mutators's log

A

B

C

Or Ostrovsky
Concurrent Reference Counting
13/1/15 38 / 57
Example

Collector's log

Mutators's log

A

B

C

Or Ostrovsky Concurrent Reference Counting 13/1/15 38 / 57
Example

Collector’s log

A

Mutators’s log

B

C

Or Ostrovsky
Concurrent Reference Counting
13/1/15 38 / 57
Example

![Diagram showing the Collector's and Mutators' logs, with nodes A, B, and C connected by lines indicating the flow of information.](image-url)
Example
Example

Collector's log

Mutators's log

Snoop buffer
Table of contents

1. Introduction

2. Simple Reference Counting

3. Buffered Reference Counting

4. Sliding View Reference Counting

5. An Efficient On-the-Fly Cycle Collection
   - Overview
   - Concepts
   - The big picture

6. Summery
An Efficient On-the-Fly Cycle Collection

• A “Big algorithm” - incorporates many techniques
  ▶ Mark & Sweep \textit{(Lecs. 2 & 10)}
  ▶ Generational GC \textit{(Lec. 5)}
  ▶ The Recycler \textit{(Lec. 3)}
  ▶ Coalesced reference counting
  ▶ Sliding views

• Relatively new - \textit{Paz et al. 2007}

• We’ll handle it \textit{very} carefully

• No full pseudocode
Mark & Sweep

- Mark all reachable objects and reclaim the rest.
- Algorithm:
  - Trace memory from roots.
  - Mark reachable objects.
  - Release unmarked objects
- Work - proportional to the number of live objects
- Simpler version need a “fixed” heap
Generational garbage collection

- The weak generational hypothesis: most objects die young
- Young generation
  - High mutation and death rates
  - Use\(^1\) Mark & Sweep - trace only live objects
- Old generation
  - Low mutation and death rates
  - Use\(^1\) Reference counting - handle big live heaps

\(^1\)For the current scheme
The Recycler

- Proposed by Bacon & Rajan at 2001
- Reclaim unreachable circles using trial deletion

**Algorithm:**
- Build a set of candidates
- Decrement counts for each candidate (mark as gray)
- Scan objects reachable from candidates with $rc \neq 0$
  - Those items are marked black
  - The rest are marked white
- Remove white objects

- Many traversals are required - needs a “fixed” heap
Algorithm 12 Cycle collection

1: collectCycles()
2: markCandidates()
3: markLiveBlack()
4: scan()
5: collectWhite()
6: processBuffers()
The major scheme

- Objects are created young
- Log old objects
- Collect logs from each thread (one at a time)
- Clean dirty objects
- Update reference counts of old objects
- M&S young objects
- Apply the Recycler on old objects
Corner rounding & circle squaring
Corner rounding & circle squaring

- Inter-generational pointers
  - Mark & Sweep - old → young
  - The Recycler - young → old

- “Live” heaps
  - Mark & Sweep - See Lec. 10
  - The Recycler - Retracing the graph

- Aging - correct the reference count to and from a young object
Solutions

- **Inter-generational pointers**
  - old → young - mark objects when processing logs
  - young → old - young objects are candidates for the Recycler

- “Live” heaps - Use the sliding view

- **Aging**
  - References from - increment using sliding window
  - References to - are logged with the old objects
Allocating objects

Algorithm 13 Object allocation

1: New()
2: ref ← allocate()
3: add(myYoungSet, ref)
4: setDirty(ref)
5: return ref
Algorithm 14 Reference count updates

1:    incrementNew(entry)
2:        replica ← readSlidingView(entry)
3:        for each fld in Pointers(replica)
4:            child ← *fld
5:                if child ≠ null
6:                    rc(child) ← rc(child) + 1
7:                    mark(child) /*for M&S*/

8:
9:    decrementOld(entry)
10:        for each fld in Pointer(entry)
11:            child ← *fld
12:                if child ≠ null
13:                    rc(child) ← rc(child) - 1
14:                        if rc(child) = 0
15:                            add(ztc, child)

16:
17:    processReferenceCounts()
18:        for each entry in updates
19:            decrementOld(entry)
20:            incrementNew(entry)
The “main” method

**Algorithm 15 Collection**

1: collect()
2: collectSlidingView()
3: for each t in threads /*handshake 4*/
4: suspend(t)
5: scanStack(t)
6: snoopFlag[t] ← false
7: resume(t)
8: processReferenceCounts()
9: markNursery()
10: sweepNursery()
11: sweepZCT()
12: collectCycles()
Example

Collector's log

Mutators's log

Roots

Snoop buffer

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

Table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Diagram:

- Collector's log
- Mutators's log
- Roots

Nodes: A, B, C, D, E, F, G

Connections:
- A to B
- B to C
- A to D
- D to E
- E to F

Note: The diagram illustrates a simple data flow and reference counting mechanism in a concurrent reference counting system.
Example
Example

Collector’s log

Mutators’s log

Roots

Snoop buffer

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Collectors’s log

Mutators’s log

Roots

Snoop buffer
Example

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

Concurrent Reference Counting

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Example
Example

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Analysis

Pro.
- Uses single-threaded algorithms
- Performance
  - All the advantages of buffered reference counting
  - Fewer reference count changes
- Handles circles

Con.
- Very complex
- Relatively new
<table>
<thead>
<tr>
<th></th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Simple Reference Counting</td>
</tr>
<tr>
<td>3</td>
<td>Buffered Reference Counting</td>
</tr>
<tr>
<td>4</td>
<td>Sliding View Reference Counting</td>
</tr>
<tr>
<td>5</td>
<td>An Efficient On-the-Fly Cycle Collection</td>
</tr>
<tr>
<td>6</td>
<td>Summery</td>
</tr>
</tbody>
</table>
Discussion

- Sliding view limitations
- Improvements for the on the fly collector
- Combine algorithms with reference counting
- What is reference counting good for?
Overview

- Simple reference counting - both flavors
- Buffered reference counting - let another thread do the accounting
- Coalesced reference counting - know what to ignore
- Sliding view - take a snapshot of the heap while it’s moving
- On-the-Fly cycle collection - Sliding view usage example
## Any questions?

<table>
<thead>
<tr>
<th>The Simple Answers to the Questions That Get Asked About Every New Technology:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Will [ ] make us all geniuses?</td>
<td>NO</td>
</tr>
<tr>
<td>Will [ ] make us all morons?</td>
<td>NO</td>
</tr>
<tr>
<td>Will [ ] destroy whole industries?</td>
<td>YES</td>
</tr>
<tr>
<td>Will [ ] make us more empathetic?</td>
<td>NO</td>
</tr>
<tr>
<td>Will [ ] make us less caring?</td>
<td>NO</td>
</tr>
<tr>
<td>Will teens use [ ] for sex?</td>
<td>YES</td>
</tr>
<tr>
<td>Were they going to have sex anyway?</td>
<td>YES</td>
</tr>
<tr>
<td>Will [ ] destroy music?</td>
<td>NO</td>
</tr>
<tr>
<td>Will [ ] destroy art?</td>
<td>NO</td>
</tr>
<tr>
<td>But can't we go back to a time when--</td>
<td>NO</td>
</tr>
<tr>
<td>Will [ ] bring about world peace?</td>
<td>NO</td>
</tr>
<tr>
<td>Will [ ] cause widespread alienation by creating a world of empty experiences?</td>
<td>We were already alienated</td>
</tr>
</tbody>
</table>
Any comments?