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

Background – Reference Counting

- Reference Counting (RC): technique for automatic memory management
- Strategy: maintain per-object counters to determine each object's liveness

Background – Reference Counting

- Reference Counting (RC): technique for automatic memory management
- Strategy: maintain per-object counters to determine each object's liveness
- $\text{RC}(\text{obj}) > 0 \Rightarrow$ alive
- $\text{RC}(\text{obj}) = 0 \Rightarrow$ dead: can be collected

Background – Reference Counting

- Reference Counting (RC): technique for automatic memory management
- Strategy: maintain per-object counters to determine each object's liveness
- $\text{RC}(\text{obj}) > 0 \Rightarrow$ alive
- $\text{RC}(\text{obj}) = 0 \Rightarrow$ 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
- $\text{RC}(\text{obj}) > 0 \Rightarrow$ alive
- $\text{RC}(\text{obj}) = 0 \Rightarrow$ 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 Example

Key: Actively maintain reference count for each object throughout execution

Reference Counting Example

Key: Actively maintain reference count for each object throughout execution

```
        rc(obj1) = 1
1       var a = new obj1
```

Reference Counting Example

Key: Actively maintain reference count for each object throughout execution

```
        rc(obj1) = 1
1       var a = new obj1
        rc(obj1)++
2       var b = a
        rc(obj1)--, rc(obj2) = 1
3       b = new obj2
```

Reference Counting Example

Key: Actively maintain reference count for each object throughout execution

```
        rc(obj1) = 1
1       var a = new obj1
        rc(obj1)++
2       var b = a
        rc(obj1)--, rc(obj2) = 1
3       b = new obj2
        rc(obj1)--,rc(obj3) = 1
        free(obj1)
4       a = new obj3
```

Maintaining Thread Safety

- Problem: multiple threads may try to adjust an objects RC counter concurrently

Maintaining Thread Safety

- Problem: multiple threads may try to adjust an objects RC counter concurrently

Initialization

```
rc(obj) = 1
1  var g = new obj;
```

Thread A	Thread B
rc(obj)++	*rc(obj)++*
2_A `var a = g;`	2_B `var b = g;`

Maintaining Thread Safety

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

Maintaining Thread Safety

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

```
procedure INCREMENT(obj)
    do
        old := obj.rc_counter                    ▷ read old value
        new := old
        new += 1                                 ▷ set new value
    while !CAS(&obj.rc_counter, old, new)
                                                 ▷ Atomic update of counter
        end procedure
```

Maintaining Thread Safety

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

```
procedure INCREMENT(obj)
    do
        old := obj.rc_counter                    ▷ read old value
        new := old
        new += 1                                 ▷ set new value
    while !CAS(&obj.rc_counter, old, new)
                                                 ▷ Atomic update of counter
        end procedure
```

Overheads of Reference Counting

- Problem: RC has significant performance overheads

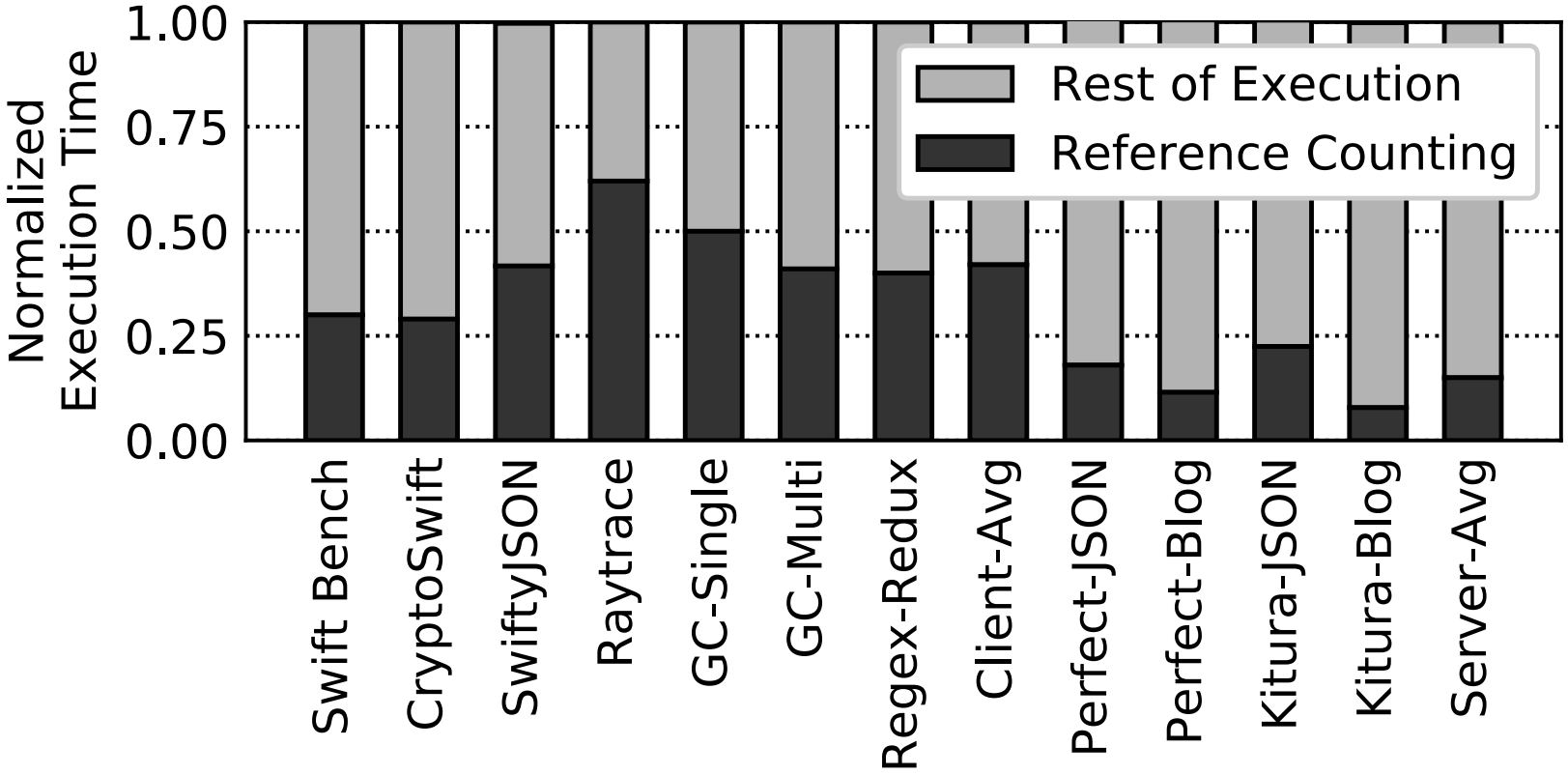
Overheads of Reference Counting

- 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 developement
  - Includes many optimizations to reduce the number of reference counting operations (RCOps) needed
    - removes up to 97% of the RCOps

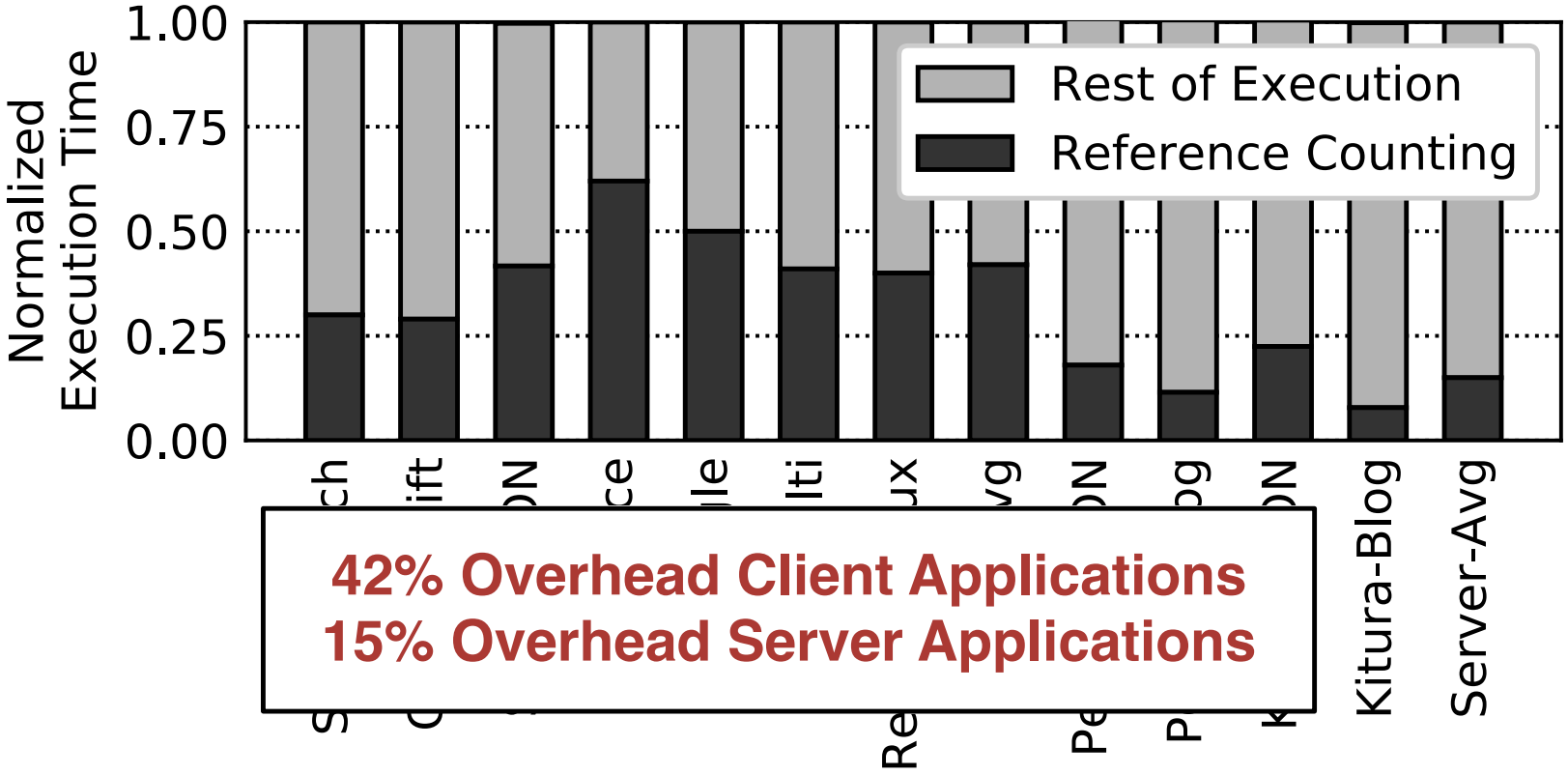
Overheads of Reference Counting

- 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 developement
  - 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



Overheads of Reference Counting



Breaking Down Overhead of RC

- Compare-And-Swap (CAS) Operations are expensive

Breaking Down Overhead of RC

- Compare-And-Swap (CAS) Operations are expensive

Configuration	Time (ns)
Non-atomic Increment	5.77
CAS Increment	13.84

Breaking Down Overhead of RC

- Compare-And-Swap (CAS) Operations are expensive
- Can reduce execution time by on average 25% by using non-atomic operations

Configuration	Time (ns)
Non-atomic Increment	5.77
CAS Increment	13.84

Breaking Down Sharing Patterns of RC

- 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

	Program Name	Objects		RC operations	
		Priv (%)	Shar (%)	Priv (%)	Shar (%)
Client	Swift Benchmark	100.00	0.00	100.00	0.00
	CryptoSwift	100.00	0.00	100.00	0.00
	SwiftyJSON	100.00	0.00	100.00	0.00
	Raytrace	100.00	0.00	100.00	0.00
	GCBench-Single	100.00	0.00	100.00	0.00
	GCBench-Multi	99.84	0.16	99.68	0.32
	Regex-Redux	99.99	0.01	51.13	48.87
	Average	99.98	0.02	92.97	7.03
Server	Perfect-JSON	94.74	5.26	83.99	16.01
	Perfect-Blog	94.58	5.42	95.33	4.67
	Kitura-JSON	91.41	8.59	84.29	15.71
	Kitura-Blog	91.59	8.41	83.32	16.68
	Average	93.08	6.92	86.73	13.27

Breaking Down Sharing Patterns of RC

	Program Name	Objects		RC operations	
		Priv (%)	Shar (%)	Priv (%)	Shar (%)
Client	Swift Benchmark	100.00	0.00	100.00	0.00
	CryptoSwift	100.00	0.00	100.00	0.00
	SwiftyJSON	100.00	0.00	100.00	0.00
	Raytrace	100.00	0.00	100.00	0.00
	GCBench-Single	100.00	0.00	100.00	0.00
	GCBench-Multi	99.84	0.16	99.68	0.32
	Regex-Redux	99.99	0.01	51.13	48.87
	Average	99.98	0.02	92.97	7.03
Server	Perfect-JSON	94.74	5.26	83.99	16.01
	Perfect-Blog	94.58	5.42	95.33	4.67
	Kitura-JSON	91.41	8.59	84.29	15.71
	Kitura-Blog	91.59	8.41	83.32	16.68
	Average	93.08	6.92	86.73	13.27

Breaking Down Sharing Patterns of RC

	Program Name	Objects		RC operations	
		Priv (%)	Shar (%)	Priv (%)	Shar (%)
Client	Swift Benchmark	100.00	0.00	100.00	0.00
	CryptoSwift	100.00	0.00	100.00	0.00
	SwiftyJSON	100.00	0.00	100.00	0.00
	Raytrace	100.00	0.00	100.00	0.00
	GCBench-Single	100.00	0.00	100.00	0.00
	GCBench-Multi	99.84	0.16	99.68	0.32
	Regex-Redux	99.99	0.01	51.13	48.87
	Average	99.98	0.02	92.97	7.03
Server	Perfect-JSON	94.74	5.26	83.99	16.01
	Perfect-Blog	94.58	5.42	95.33	4.67
	Kitura-JSON	91.41	8.59	84.29	15.71
	Kitura-Blog	91.59	8.41	83.32	16.68
	Average	93.08	6.92	86.73	13.27

Breaking Down Sharing Patterns of RC

	Program Name	Objects		RC operations	
		Priv (%)	Shar (%)	Priv (%)	Shar (%)
Client	Swift Benchmark	100.00	0.00	100.00	0.00
	CryptoSwift	100.00	0.00	100.00	0.00
	SwiftyJSON	100.00	0.00	100.00	0.00
	Raytrace	100.00	0.00	100.00	0.00
	GCBench-Single	100.00	0.00	100.00	0.00
	GCBench-Multi	99.84	0.16	99.68	0.32
	Regex-Redux	99.99	0.01	51.13	48.87
	Average	99.98	0.02	92.97	7.03
Server	Perfect-JSON	94.74	5.26	83.99	16.01
	Perfect-Blog	94.58	5.42	95.33	4.67
	Kitura-JSON	91.41	8.59	84.29	15.71
	Kitura-Blog	91.59	8.41	83.32	16.68
	Average	93.08	6.92	86.73	13.27

Breaking Down Sharing Patterns of RC

	Program Name	Objects		RC operations	
		Priv (%)	Shar (%)	Priv (%)	Shar (%)
Client	Swift Benchmark	100.00	0.00	100.00	0.00
	CryptoSwift	100.00	0.00	100.00	0.00
	SwiftyJSON	100.00	0.00	100.00	0.00
	Raytrace	100.00	0.00	100.00	0.00
	GCBench-Single	100.00	0.00	100.00	0.00
	GCBench-Multi	99.84	0.16	99.68	0.32
	Regex-Redux	99.99	0.01	51.13	48.87
	Average	99.98	0.02	92.97	7.03
Server	Perfect-JSON	94.74	5.26	83.99	16.01
	Perfect-Blog	94.58	5.42	95.33	4.67
	Kitura-JSON	91.41	8.59	84.29	15.71
	Kitura-Blog	91.59	8.41	83.32	16.68
	Average	93.08	6.92	86.73	13.27

Introducing Biased Reference Counting (BRC)

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

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 intializing 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 intializing 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

Counter	Reserved
30 bits	**34 bits**

Original RCWord.

New BRC RCWord

Counter	Reserved
30 bits	**34 bits**

Original RCWord.

Biased		Shared		
TID	Counter	Counter	Flags	Reserved
18 bits	**14 bits**	**14 bits**	**2 bits**	**16 bits**

BRC's RCWord.

BRC Increment Example

```
procedure INCREMENT(obj)

owner_tid := obj.rcword.biased.tid
    my_tid := GetThreadID()
```

BRC Increment Example

```
procedure INCREMENT(obj)
owner_tid := obj.rcword.biased.tid
    my_tid := GetThreadID()
    if owner_tid == my_tid then
        FastIncrement(obj)
    else
        SlowIncrement(obj)
    end if
end procedure
```

BRC Increment Example

```
procedure INCREMENT(obj)
owner_tid := obj.rcword.biased.tid
    my_tid := GetThreadID()
    if owner_tid == my_tid then
        FastIncrement(obj)
    else
        SlowIncrement(obj)
    end if
end procedure

procedure FASTINCREMENT(obj)
    obj.rcword.biased.counter += 1
end procedure
```

BRC Increment Example

```
procedure INCREMENT(obj)
owner_tid := obj.rcword.biased.tid
    my_tid := GetThreadID()
    if owner_tid == my_tid then
        FastIncrement(obj)
    else
        SlowIncrement(obj)
    end if
end procedure

procedure FASTINCREMENT(obj)
    obj.rcword.biased.counter += 1
end procedure
```

```
procedure SLOWINCREMENT(obj)
    do
        old := obj.rcword.shared
        new := old
        new.counter += 1
    while !CAS(&obj.rcword.shared, old, new)
        end procedure
```

BRC Increment Example

```
procedure INCREMENT(obj)
owner_tid := obj.rcword.biased.tid
    my_tid := GetThreadID()
    if owner_tid == my_tid then
        FastIncrement(obj)
    else
        SlowIncrement(obj)
    end if
end procedure

procedure FASTINCREMENT(obj)
    obj.rcword.biased.counter += 1
end procedure
```

```
procedure SLOWINCREMENT(obj)
    do
        old := obj.rcword.shared
        new := old
        new.counter += 1
    while !CAS(&obj.rcword.shared, old, new)
        end procedure
```

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

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

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

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

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

Merging Counters

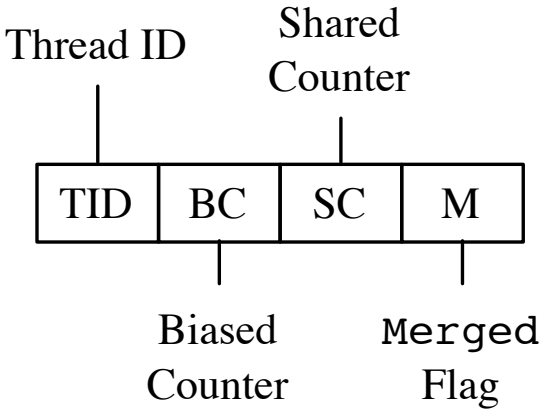
- Transfering to *merged* state involves merging the two counters
  - Merge operation must be performed by owner thread
- Can transfer to the *merged* state both implicitly and explicitly
- *Implicit Merge*: merge counters when BC = 0
  - No arithmetic necessary
  - CAS is needed to set the `merged`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

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

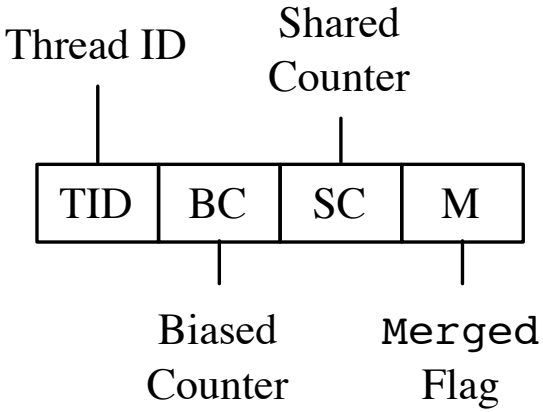
More details in paper

Example – Private Object



RCWord Structure

Example – Private Object

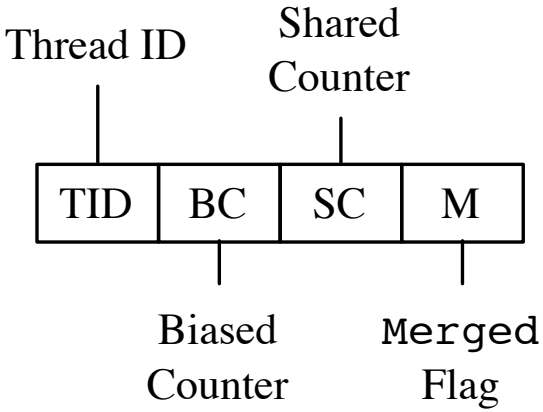


RCWord Structure

T1	**1**	**0**	**0**

T1 allocates an object

Example – Private Object



RCWord Structure

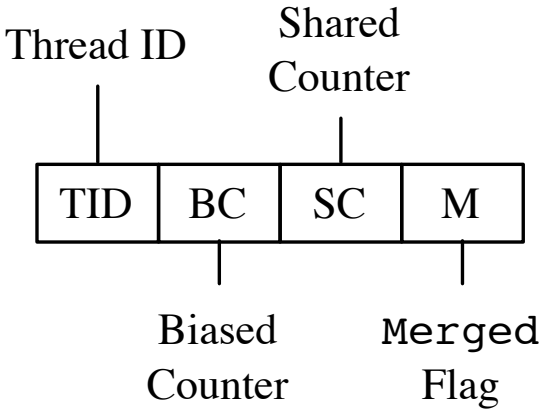
T1	1	0	0

T1 allocates an object

T1	**N**	**0**	**0**

T1 creates N references

Example – Private Object



RCWord Structure

T1	1	0	0

T1 allocates an object

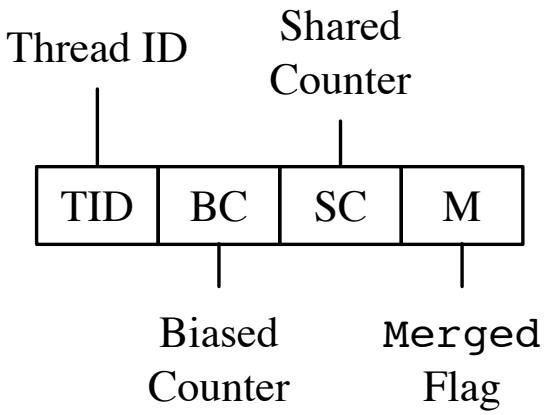
T1	N	0	0

T1 creates *N* references

T1	**0**	**0**	**0**

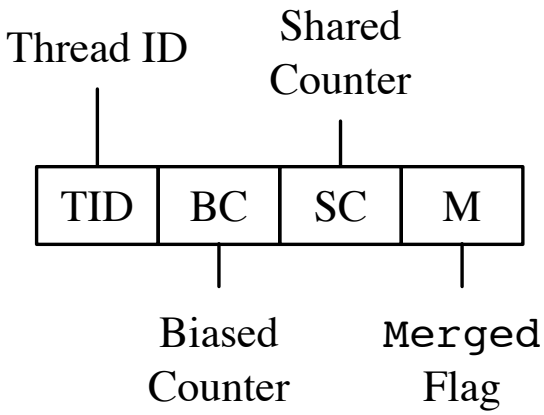
T1 removes all references
T1 deallocates object

Example – Shared Object



RCWord Structure

Example – Shared Object

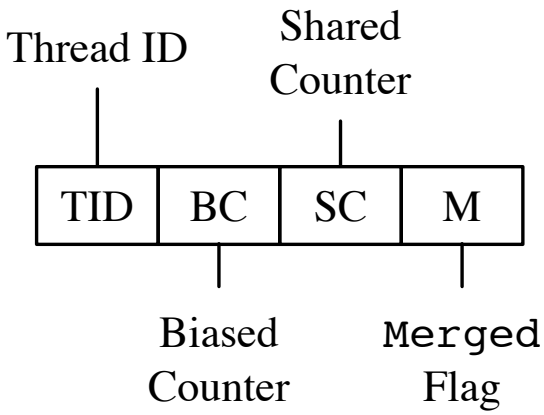


RCWord Structure

T1	**1**	**0**	**0**

T1 allocates an object

Example – Shared Object



RCWord Structure

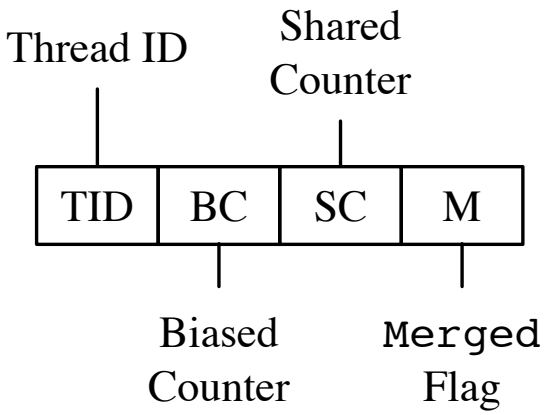
T1	1	0	0

T1 allocates an object

T1	**1**	**1**	**0**

T2 creates a reference

Example – Shared Object



RCWord Structure

T1	1	0	0

T1 allocates an object

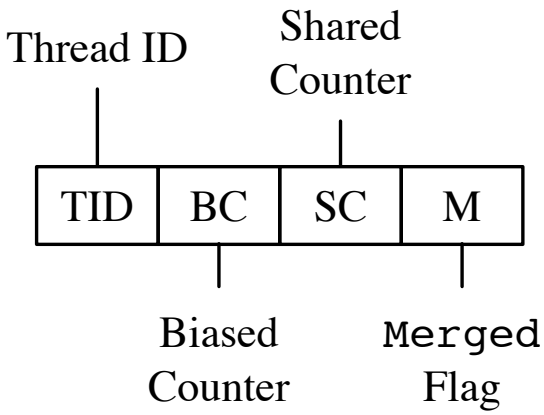
T1	1	1	0

T2 creates a reference

	0	**1**	**1**

T1 removes its reference
T1 implicitly merges

Example – Shared Object



RCWord Structure

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

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

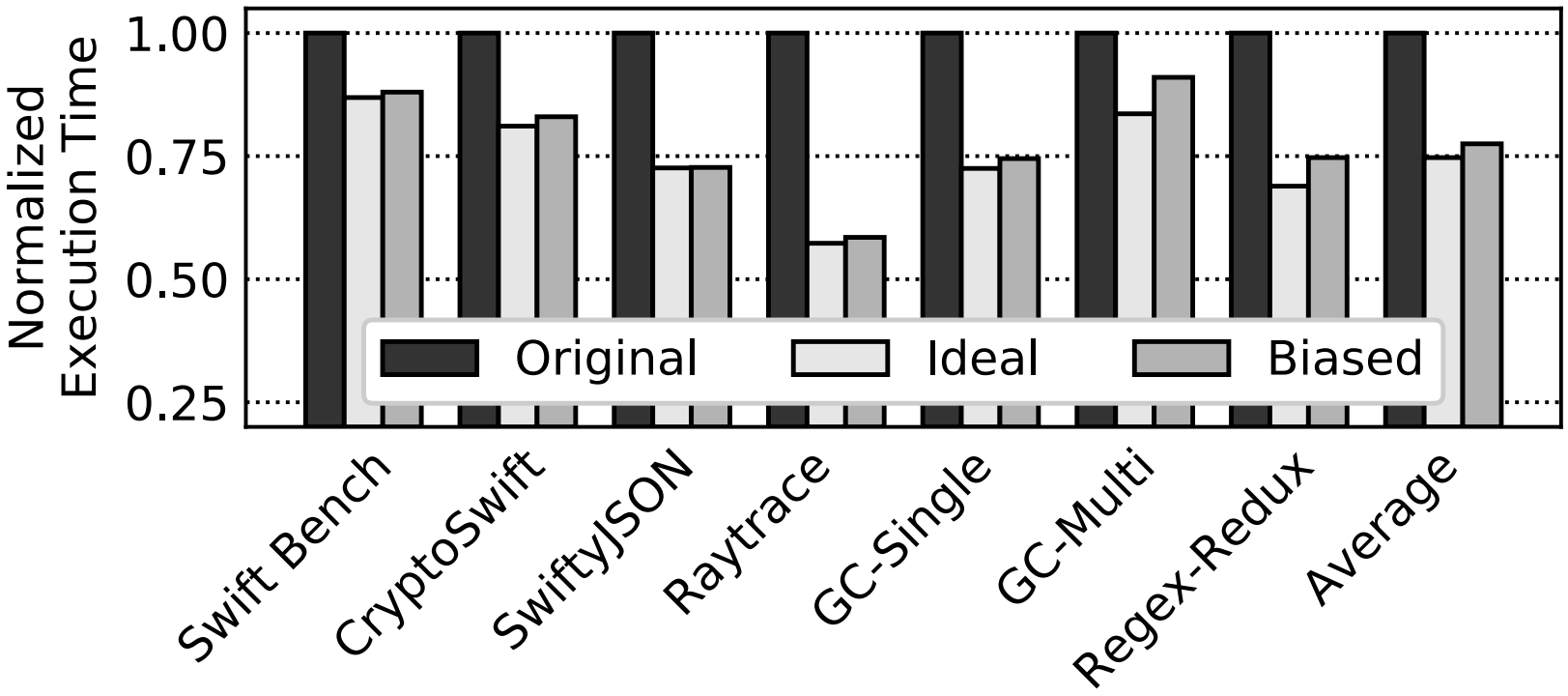
Evaluation Setup

- 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

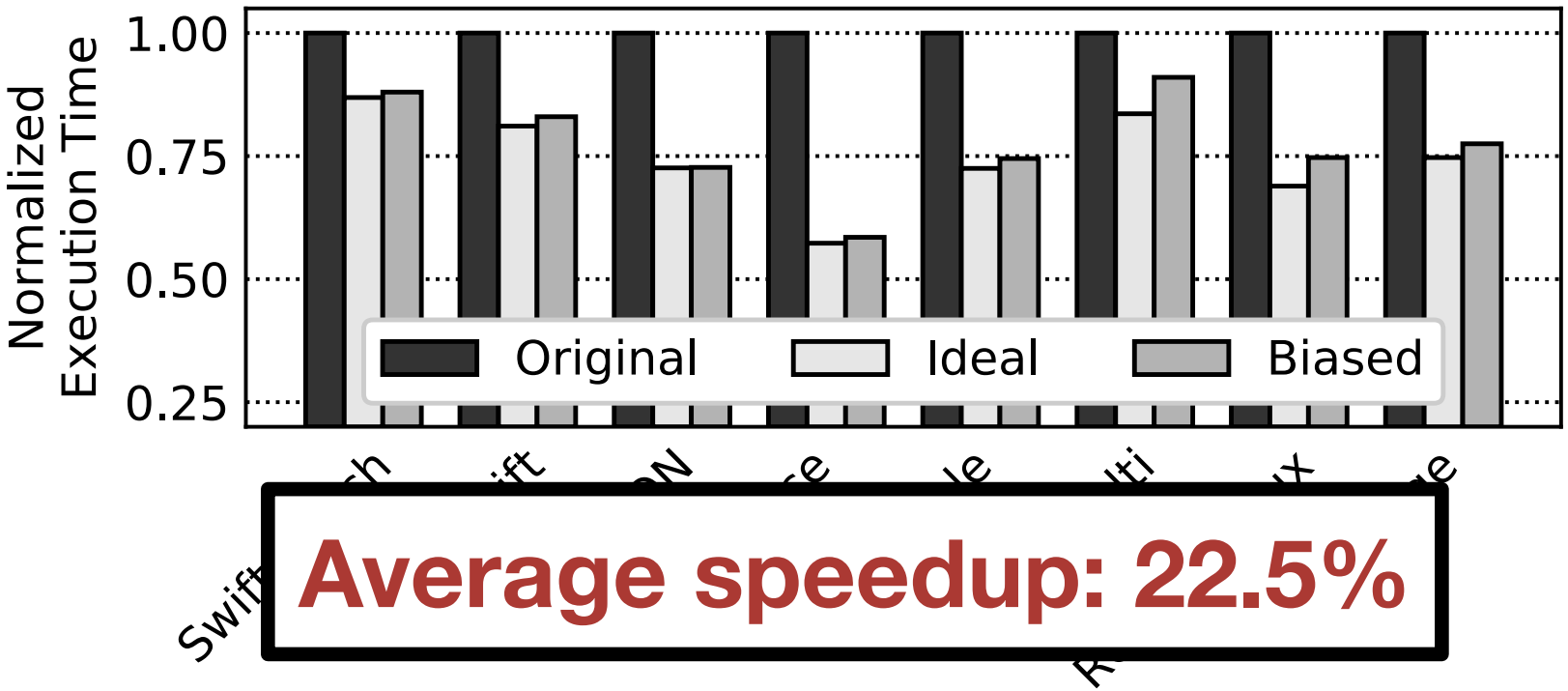
Evaluation – Operation Performance

Configuration	Time (ns)
Original Counter	13.84
Ideal Counter	5.77
Biased Counter	6.28
Shared Counter	15.57

Evaluation – Application Performance

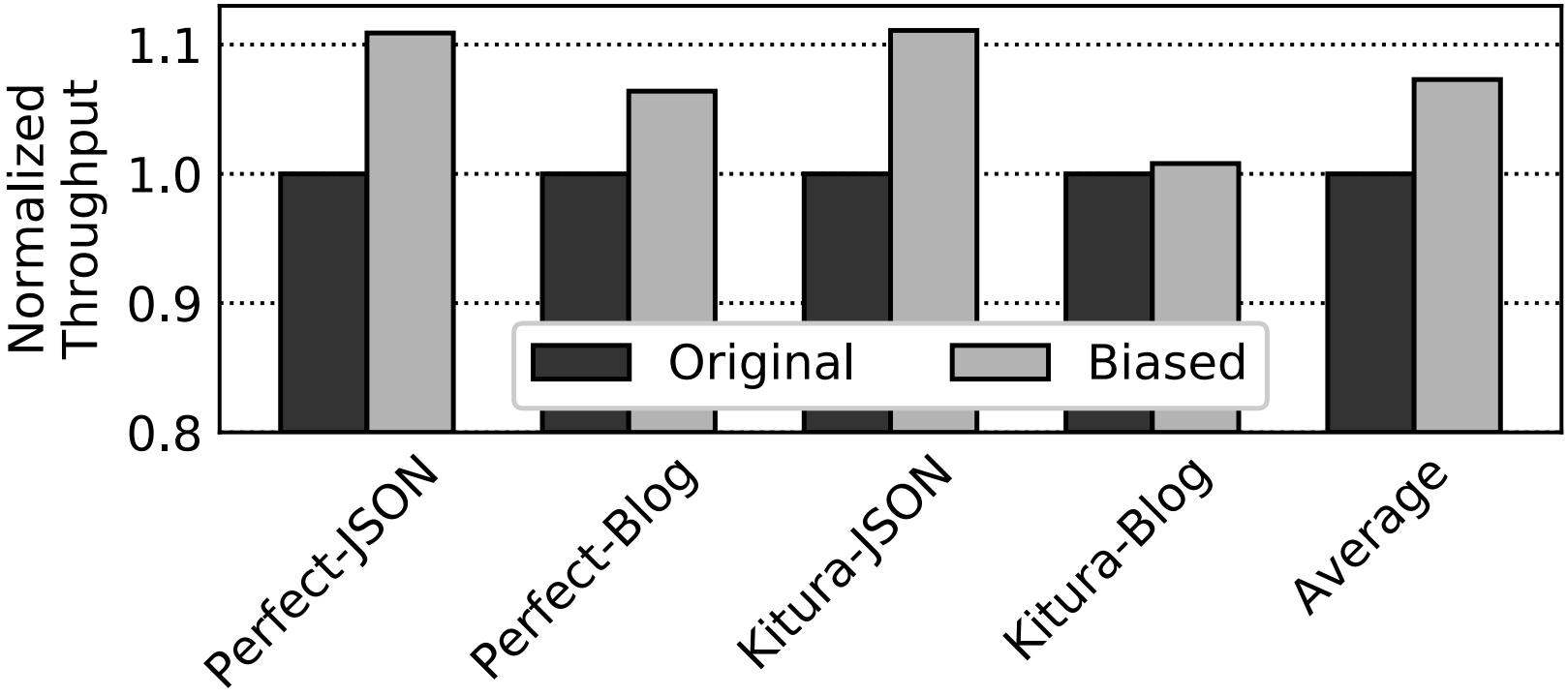


Evaluation – Application Performance

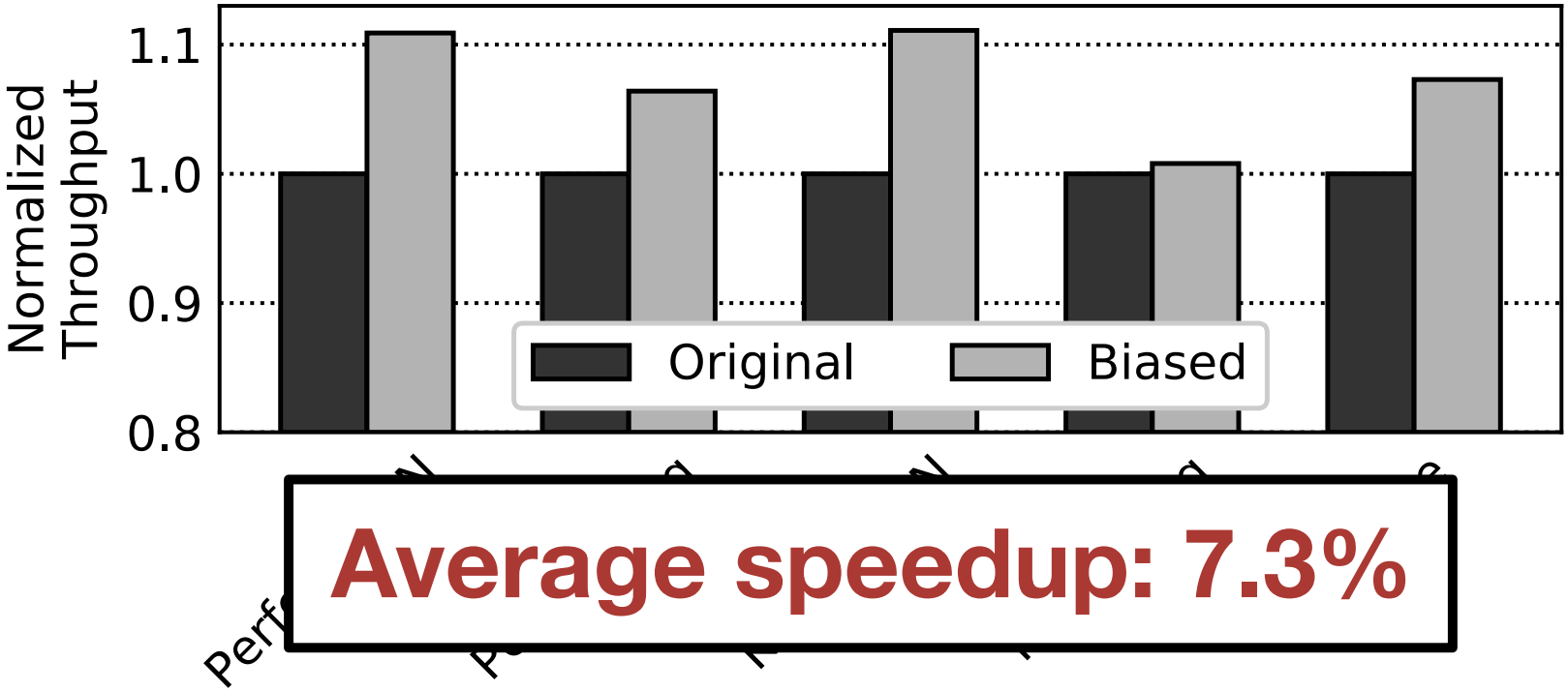


Average speedup: 22.5%

Evaluation – Application Performance

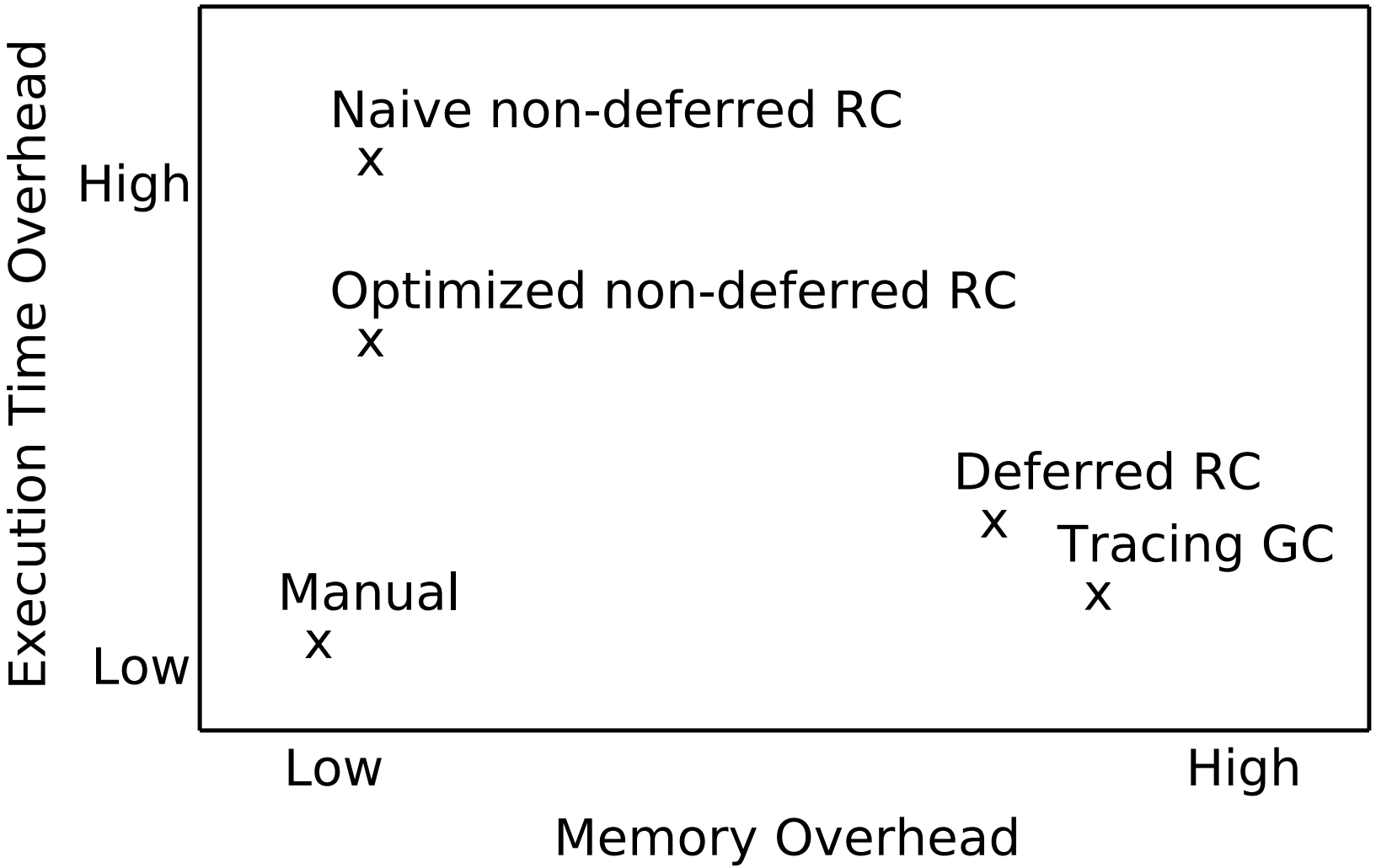


Evaluation – Application Performance

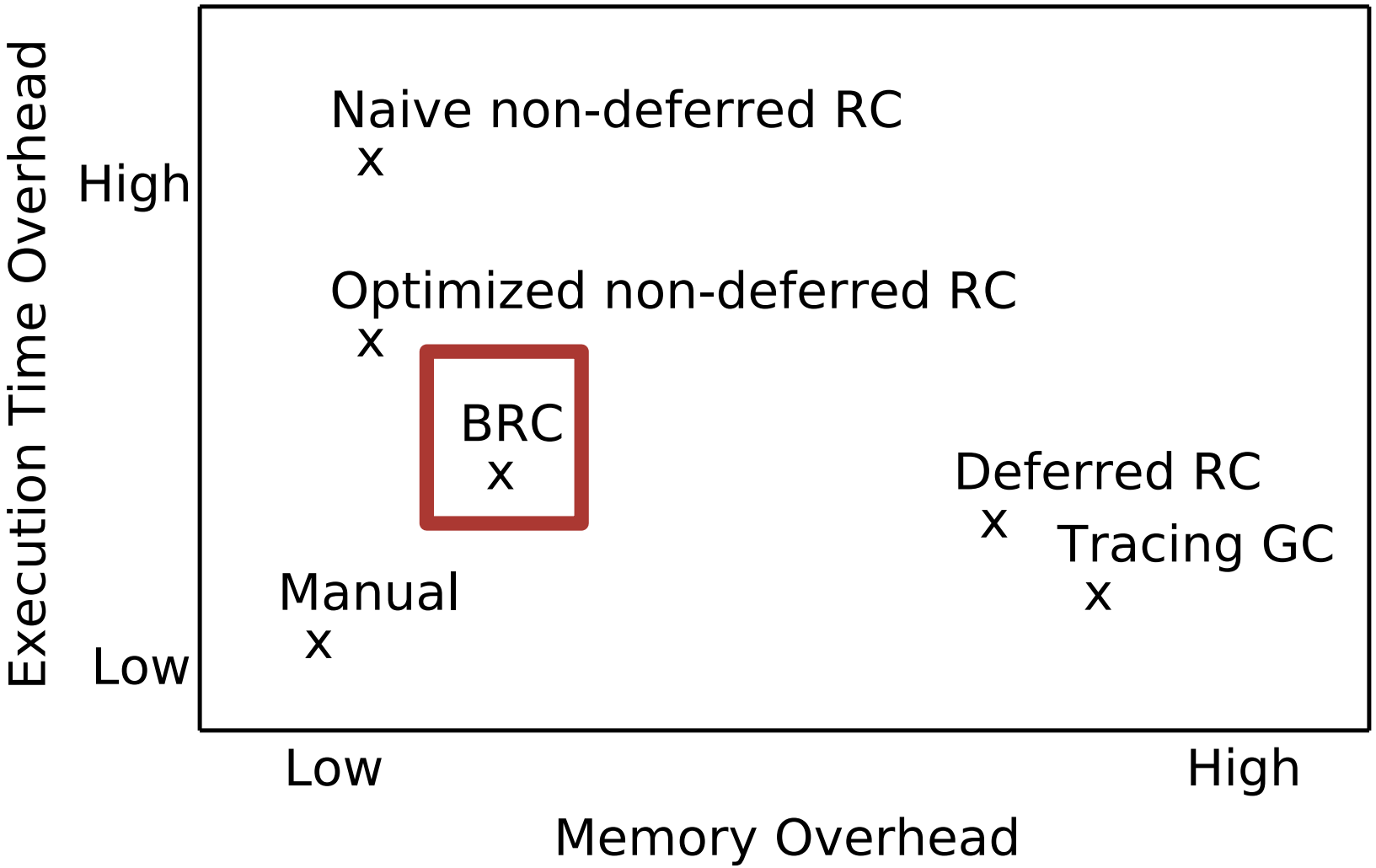


Average speedup: 7.3%

Putting BRC in Context



Putting BRC in Context



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?

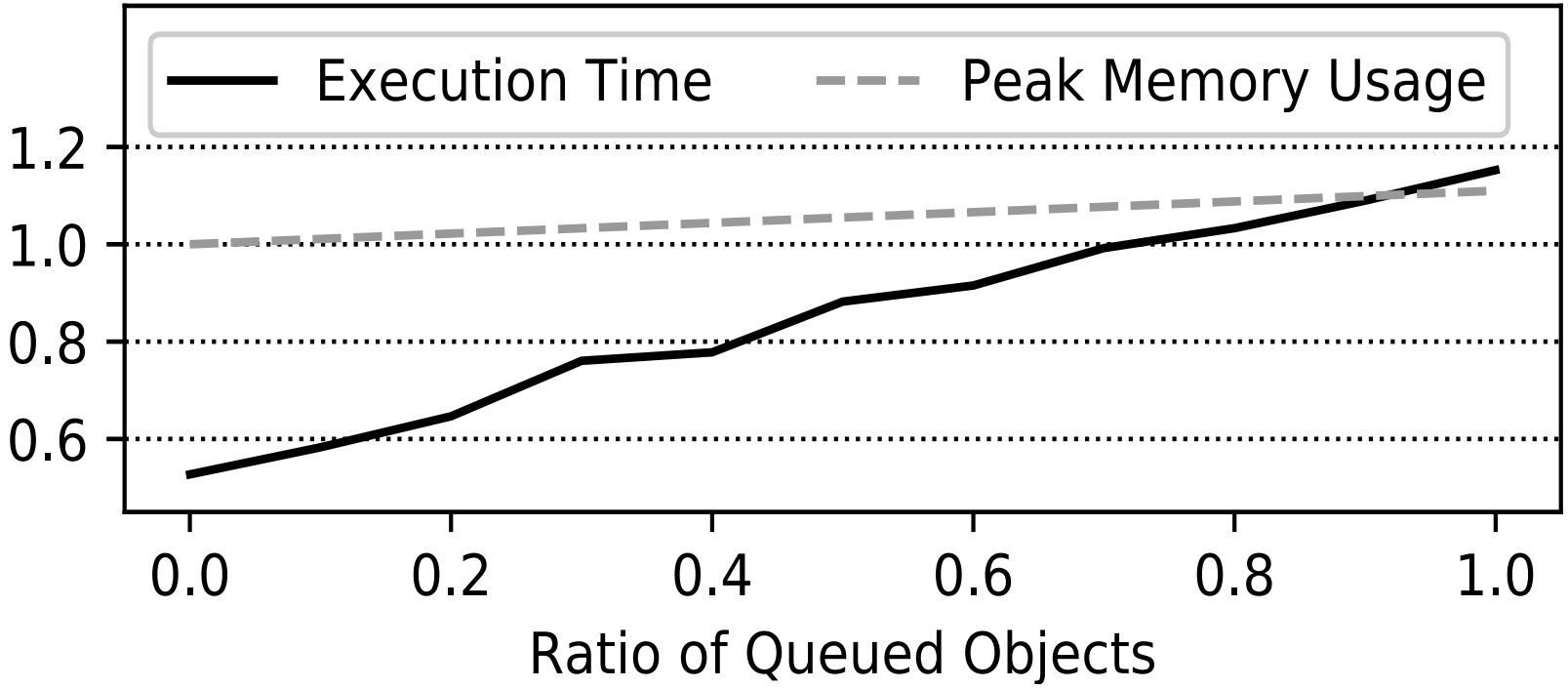
Backup Slides

Evaluation – Rebiasing

We did evaluate the performance of rebaising

- Rebiase 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

Evaluation – Behavior

	Program Name	% of Shared Obj.	Original			Biased					
			Obj. Allocs. per μs	RC Ops. per μs	RC Ops. per Obj.	RC Ops. per μs	% of RC Ops. to Shared Obj.	% of RC Ops. to Biased Counter	% of RC Ops. to Shared Counter	% of Queued Obj.	% of RC Ops. Setting Queued Flag
Client	Swift Benchmark	0.00	1.91	29.18	15.24	33.16	0.00	100.00	0.00	0.00	0.00
	CryptoSwift	0.00	2.39	56.54	23.61	68.16	0.00	100.00	0.00	0.00	0.00
	SwiftyJSON	0.00	2.60	68.19	26.24	93.77	0.00	100.00	0.00	0.00	0.00
	Raytrace	0.00	0.00	101.70	27258.42	173.94	0.00	100.00	0.00	0.00	0.00
	GCBench-Single	0.00	12.87	64.07	4.98	86.03	0.00	100.00	0.00	0.00	0.00
	GCBench-Multi	0.16	50.69	150.39	2.97	195.44	0.32	99.97	0.03	0.01	0.00
	Regex-Redux	0.01	2.58	92.61	35.91	123.99	48.87	88.02	11.98	0.00	0.00
	Average	0.02	10.44	80.38	39338.19	110.64	7.03	84.00	1.72	0.00	0.00
Server	Perfect-JSON	5.26	0.55	4.56	8.25	4.95	16.01	88.64	11.36	1.99	0.24
	Perfect-Blog	5.42	0.45	12.57	27.95	12.90	4.67	96.72	3.28	1.95	0.07
	Kitura-JSON	8.59	0.40	7.27	18.05	7.55	15.71	87.38	12.62	2.76	0.15
	Kitura-Blog	8.41	0.39	6.12	15.81	6.34	16.68	86.68	13.32	2.70	0.17
	Average	6.92	0.45	7.63	17.51	7.93	13.27	89.85	10.15	2.35	0.16

Table 7: Reference counting statistics.