Biased Reference Counting: Minimizing Atomic Operations in Garbage Collection

Jiho Choi, Thomas Shull, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

ABSTRACT

Reference counting (RC) is one of the two fundamental approaches to garbage collection. It has the desirable characteristics of low memory overhead and short pause times, which are key in today's interactive mobile platforms. However, RC has a higher execution time overhead than its counterpart, tracing garbage collection. The reason is that RC implementations maintain per-object counters, which must be continually updated. In particular, the execution time overhead is high in environments where low memory overhead is critical and, therefore, non-deferred RC is used. This is because the counter updates need to be performed atomically.

To address this problem, this paper proposes a novel algorithm called *Biased Reference Counting* (BRC), which significantly improves the performance of non-deferred RC. BRC is based on the observation that most objects are only accessed by a single thread, which allows most RC operations to be performed non-atomically. BRC leverages this by biasing each object towards a specific thread, and keeping two counters for each object — one updated by the owner thread and another updated by the other threads. This allows the owner thread to perform RC operations non-atomically, while the other threads update the second counter atomically.

We implement BRC in the Swift programming language runtime, and evaluate it with client and server programs. We find that BRC makes each RC operation more than twice faster in the common case. As a result, BRC reduces the average execution time of client programs by 22.5%, and boosts the average throughput of server programs by 7.3%.

CCS CONCEPTS

• **Software and its engineering → Garbage collection**; *Software performance*; *Runtime environments*;

KEYWORDS

Reference counting; Garbage collection; Swift

ACM Reference Format:
Jiho Choi, Thomas Shull, and Josep Torrellas. 2018. Biased Reference Counting:, Minimizing Atomic Operations in Garbage Collection. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3243176.3243195

1 INTRODUCTION

High-level programming languages are widely used today. They provide programmers intuitive abstractions that hide many of the underlying computer system details, improving both programmer productivity and portability. One of the pillars of high-level programming languages is automatic memory management. It frees programmers from the obligation to explicitly deallocate resources, by relying on the runtime to automatically handle memory management.

Garbage collection is the process of runtime monitoring the lifetime of objects and freeing them once they are no longer necessary. There are two main approaches to garbage collection: tracing [28] and reference counting (RC) [17]. Tracing garbage collection maintains a root set of live objects and finds the set of objects reachable from this root set. Objects not reachable from the root set are considered dead and their resources can be freed. RC garbage collection, on the other hand, maintains a counter for each object, which tracks the number of references currently pointing to the object. This counter is actively updated as references are added and removed. Once the counter reaches zero, the object can be collected.

Most implementations of managed languages use tracing garbage collection, as RC is believed to be slower. This belief stems from the fact that most of the tracing garbage collection can be done in the background, off the critical path, while RC is on the critical path. However, many optimization techniques exist to limit the number of RC operations and reduce the overhead on the critical path. Furthermore, RC has the desirable characteristics of low memory overhead and short pause times.

In garbage collection, memory overhead comes from two sources, namely garbage collector metadata, and objects that are dead but not yet reclaimed by the garbage collector. While RC adds, as metadata, one counter per object, RC can have low overall memory overhead because it can be designed to free up objects very soon after they become dead.

Pause times are times when the application is stopped, to allow the garbage collector to perform maintenance operations. RC can be designed to have only short pause times — when individual objects are freed up.

Overall, the combination of low memory overhead and short pause times makes RC suitable for today's interactive mobile platforms. For this reason, some languages such as Swift [8] use RC.

Unfortunately, RC can have significant execution time overhead when using algorithms that reclaim objects immediately after they become dead — i.e., *non-deferred* RC algorithms. For example, we find that the non-deferred RC algorithm used in Swift causes Swift programs to spend 32% of their execution time performing RC operations. Still, using non-deferred RC is highly desirable when it is critical to keep memory overhead to a minimum, as in many mobile platforms.

We find that a major reason for this execution time overhead of non-deferred RC is the use of atomic operations to adjust the reference counters of objects. Note that even if a Swift program has little sharing and, in fact, even if it is single-threaded, it may have to use atomic operations. This is because, like many programming languages, Swift compiles components separately to allow for maximum modularity. Separate compilation forces the compiler to be conservative. Furthermore, Swift is compiled ahead of time, so the compiler cannot leverage program information gathered throughout execution to limit the use of atomic operations.

The goal of this paper is to reduce the execution time overhead of non-deferred RC. To accomplish this goal, we propose to replace the atomic RC operations with *biased* RC operations. Similar to biased locks [25], our biased operations leverage uneven sharing to create asymmetrical execution times for RC operations based on the thread performing the operation. Each object is biased toward, or favors, a specific thread. This allows an object's favored thread to perform RC operations without atomic operations, at the cost of slowing down all the other threads' RC operations on that object.

While biased RC operations are very effective in most cases, sometimes multiple threads do try to adjust the reference counter of the same object. To handle this, we exploit the fact that, unlike locking, RC does not require strong exclusivity. While only one thread is allowed to acquire a lock at any given time, it is possible for multiple threads to perform RC operations on an object concurrently — if they use multiple counters and eventually merge the counters.

Based on these ideas, we propose a novel algorithm called *Biased Reference Counting* (BRC), which significantly improves the performance of non-deferred RC. BRC maintains two counters per object — one for the owner thread and another for the other threads. The owner thread updates its counter without atomic operations; the other threads update the other counter with atomic operations.

We implement BRC in the Swift runtime. We run various client and server Swift programs and analyze both their performance and sharing patterns. Overall, we find that, on average, BRC improves the execution time of client programs by 22.5%, and the throughput of server programs by 7.3%.

The contributions of this paper are as follows:

• Characterizes the overheads of RC in Swift programs.
• Characterizes the memory behavior and sharing patterns of Swift programs.
• Proposes BRC, a new algorithm to reduce the overhead of non-deferred RC with an efficient biasing technique.
• Implements BRC in the state-of-the-art Swift runtime.
• Provides a detailed evaluation of BRC's performance.

2 BACKGROUND

2.1 Reference Counting

The fundamental idea of RC [17] is to maintain a per-object counter denoting the current number of references to the object. These per-object counters are updated as references are created, reassigned, and deleted.

Figure 1 shows a simple program which highlights all possible RC operations. The normal program commands are on the numbered lines. The RC operations required for each command are on the lines directly above the command in gray and are not numbered. A RC operation on an object `obj` is described by `rc(`*obj*`)`. In this example, the reference counts of objects obj_1, obj_2, and obj_3 are adjusted as different assignments execute.

```
    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
```

Figure 1: Basic RC operations.

While the idea of RC is straightforward, it should be implemented carefully to ensure correctness. This is because it is possible to have data races while adjusting reference counters in otherwise correct programs. Consider the code example in Figure 2. Note that in the traditional sense there is no data race in this code. Each thread only reads shared variable g, and all writes are performed to thread-local variables. However, due to both threads adding another reference to *obj*, it is possible for the reference counter of *obj* to be updated incorrectly without synchronization. In other words, it is possible for the `rc(obj)++` corresponding to the commands on lines 2_A and 2_B to race and produce incorrect results. Hence, updates to an object's reference counter must be done in a synchronized manner.

```
          Initialization

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

     Thread A          Thread B

     rc(obj)++         rc(obj)++
  2A var a = g;     2B var b = g;
```

Figure 2: Data race due to RC.

There are several approaches to synchronizing RC operations. The most obvious approach is to add locks around the RC operations. This approach has two main drawbacks. First, the runtime must decide the number of locks to create. At one extreme, the runtime can use a single lock for all objects, but this would incur a lot of contention for the lock by threads trying to adjust the reference counters of different objects. At the other extreme, each object can have its own lock. However, this adds an extra overhead to each object header which can result in less locality and more cache misses. Another problem is that there is very little work done in between the lock's acquisition and release – i.e., one simple arithmetic operation to adjust the reference counter. This makes processor stalls likely.

An alternative approach is a lock-free approach using atomic operations as shown in Algorithm 1. In this approach, the reference counter is updated using an atomic compare-and-swap (CAS) operation. If the CAS is successful, then the operation is complete. Otherwise, the process is repeated until the operation can complete successfully. This lock-free approach addresses the problems described above for the lock-based approach. First, since there are no locks, there is no tradeoff between the contention for locks versus the memory overhead of locks. Second, this algorithm has only one synchronization operation (the CAS). Because of these benefits, modern RC implementations use lock-free algorithms.

Algorithm 1 CAS-based increment operation

```
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 !$CAS(\&obj.rc\_counter, old, new)$
7:                                                     ▷ Atomic update of counter
8: end procedure
```

2.2 RC Optimization

Previous works to optimize RC can be categorized into two groups: works that defer reclamation of objects (deferred RC) and works that do not (non-deferred RC).

2.2.1 Deferred RC. These works postpone dead object reclamation, and divide execution into distinct mutation and collection phases. During mutation phases, RC operations are simplified and there is no reclamation of dead objects; during collection phases, the dead objects are reclaimed.

There are two groups of techniques: deferral and coalescing. In deferral [12, 15, 18, 37–40], the mutation phase does not perform RC operations for local pointer variables stored in the stack or registers. Then, the collection phase scans the stack and registers, and determines the objects that have a reference count equal to zero, and therefore can be reclaimed.

In coalescing [26, 32], the mutation phase only records the modified pointer variables and their initial values. Then, the collection phase compares the initial and final values of the modified pointer variables, and performs RC operations only on the objects pointed to initially and at the end. Dead objects are found and reclaimed during the collection phase.

A hybrid approach [15] uses simple tracing garbage collection for young objects and RC for old objects.

2.2.2 Optimizations of Non-deferred RC. These techniques remove unnecessary RC operations through static compiler analysis. Some proposals [14, 21–23, 30] eliminate the RC operations for a reference R to an object when R's lifetime is completely nested in the lifetime of another reference to the same object. Figure 1 shows a simple example of a candidate for this optimization. In this figure, the lifetime of the reference to obj_1 created on line 2 is completely nested in the lifetime of the reference to obj_1 created on line 1. Because of this, it is unnecessary to adjust obj_1's reference counter to reflect `var b`'s effect, so lines two and three need not adjust `rc(`obj_1`)`.

Another optimization [21] is to look at sequential chains of RC operations on the same object, and find matching pairs of increments and decrements (potentially created by different references). These RC operations can also be removed, as they negate one another.

2.3 Swift Programming Language

The Swift Programming Language was introduced by Apple in 2014 [7] as an alternative programming language to Objective-C for development for the Apple platform. Due to its support by Apple and incorporation into the Apple software ecosystem, Swift has quickly become popular and is now the preferred programming language for development on the Apple platform.

Like most modern programming languages, Swift has automatic memory management. Because of its popularity in the mobile environment, where memory overhead is a primary concern, Apple's implementation of Swift uses non-deferred RC and uses the optimizations described in Section 2.2.2. Swift uses weak references to avoid cyclic references, an approach popular in previous literature [11, 16].

3 MOTIVATION

3.1 Overhead of Reference Counting

To assess the overhead of state-of-the-art non-deferred RC, we measure the time spent performing RC operations in Swift programs. Figure 3 shows, for a set of programs, the fraction of time spent on RC operations for each program. The programs we evaluate are explained in detail in Section 6. They include client programs (Swift Bench, CryptoSwift, SwiftyJSON, Raytrace, GCBench-Single, GCBench-Multi, and Regex-Redux) and server programs (Perfect-JSON, Perfect-Blog, Kitura-JSON, and Kitura-Blog).

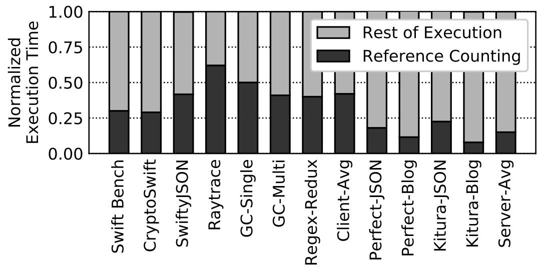

Figure 3: Overhead of RC in Swift programs.

As shown in the figure, performing RC operations takes on average 42% of the execution time in client programs, and 15% in server programs. The average across all programs can be shown to be 32%. The Swift compiler does implement optimization techniques to reduce the number of RC operations similar to those described in Section 2.2.2. Without them, the overhead would be higher. The RC overhead is lower in server programs than in client programs. This is because server programs spend relatively less time in Swift code and RC operations; they spend relatively more time in runtime functions for networking and I/O written in C++.

To estimate the contribution of using atomic operations to this overhead, we remove the CAS operation from the RC code. Hence, the Swift runtime performs RC without safeguards. As explained in Section 2.1, not using atomic operations is incorrect, and may result in either memory leaks or objects being prematurely freed. As a result, we are able to run only a subset of the programs without crash. We will show later that, on average, not using atomic operations reduces the average execution time of the programs by 25%. This means that the large majority of the RC overhead in these programs is due to the use of the CAS operations.

This large overhead is due to two reasons. First, CAS instructions are more expensive than normal updates. The reason is that there is a memory fence associated with a CAS instruction. This limits the amount of instruction overlapping performed by the hardware, preventing the out-of-order capabilities of modern cores from being effectively utilized. Second, due to contention, it may be necessary to execute a CAS instruction multiple times before completing successfully. Note that we run all of our experiments on a modern Intel Haswell processor, which has an efficient CAS implementation.

3.2 Sharing Patterns of Swift Programs

Since the use of atomic operations greatly affects performance, we evaluate how often in practice they are necessary for correctness. To do so, we modify the runtime so that, for each RC operation, we record which thread is invoking the operation and which object's reference counter is being updated. We classify objects as private or shared. An object is classified as private if all of its reference counter updates throughout its lifetime come from one single thread. Otherwise, the object is classified as shared.

Table 1 shows our results. Each row corresponds to a different program. Columns 3 and 4 show the percentage of objects that we classify as private or shared. Columns 5 and 6 show the percentage of reference counter updates that go to objects classified as private or shared. We can see that, on average, over 99% of the objects in client programs, and over 93% of those in server programs are private objects. Similarly, about 93% of the RC operations in client programs, and about 87% of those in server programs are to private objects. This means that the large majority of RC operations are to private objects, and one could skip the corresponding atomic operation. However, as argued in Section 1, the Swift compiler does not know this because Swift compiles components separately. Hence, Swift is forced to use atomic operations always for correctness.

	Program	Objects		RC operations	
	Name	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

Table 1: Sharing patterns of Swift programs.

4 BIASED REFERENCE COUNTING

4.1 Main Idea

The goal of this paper is to reduce the overhead of non-deferred RC by minimizing the use of atomic operations. We do this with a novel algorithm for RC that we call *Biased Reference Counting (BRC)*. BRC leverages the observation that many objects are only accessed by a single thread. Hence, BRC gives the ownership of, or *biases*, each object to a specific thread. BRC provides two modes of updating an object's reference count: the object's owner thread is allowed to update the reference count using non-atomic operations, while non-owner threads must use atomic operations to update the reference count.

BRC allows these two modes of execution by maintaining *separate counters* for the owner (or biased) thread and for the non-owner threads. The first counter, called the *Biased* counter, counts the number of references to the object added by the owner thread minus those removed by the owner thread. The second counter, called the *Shared* counter, maintains the active reference count for all non-owner threads combined. Since the first counter is only accessed by the owner thread, it can be accessed without atomic operations. The second counter may be accessed by multiple threads concurrently. Therefore, it requires atomic operations to prevent data races. The biasing information is maintained on a per-object basis. This allows each object to be biased toward the thread most likely to update its reference counter.

In BRC, an object can be deallocated only when the sum of its two counters is zero. Hence, the two counters first need to be merged. Since only the owner thread can read the biased counter reliably, the owner thread is responsible for merging the counters. To merge the counters, the owner thread first atomically accumulates the biased counter into the shared counter. Next, the owner sets a flag to indicate that the two counters have been merged. Once the counters are merged, if the shared counter is zero, the object may be deallocated; otherwise, the owner unbiases the object, and all subsequent reference counter updates will be performed on the shared counter. When the shared counter reaches zero, the object may be deallocated.

In the following, we describe the changes that BRC introduces to the object header, list the invariants in the BRC algorithm, show a few examples of counter transitions, and then describe the BRC algorithm in detail.

4.2 Object Header Structure

To support RC, the compiler reserves one word in each object's header, called *RCWord* (for Reference Counting Word). The Swift runtime uses a 64-bit word. Figure 4 shows the structure of the RCWord. It has a 30-bit counter to keep track of the number of references to the object. The remaining 34 bits are reserved for weak reference counting and flags to describe the state of the object. Weak references are used to prevent cycles, and are outside of the scope of this paper. To prevent race conditions, accesses to RCWord always use atomic operations.

BRC modifies RCWord as shown in Figure 5. The new RCWord is now divided into two half-words: *Biased* and *Shared.* The biased

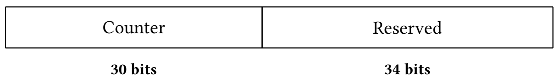

Figure 4: Original RCWord.

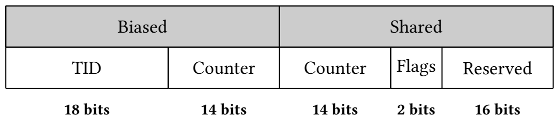

Figure 5: BRC's RCWord.

half-word contains two fields: the owner thread identifier (TID) and the biased counter. The TID indicates which thread, if any, the object is currently biased to. This thread has the exclusive right to modify the biased counter.

The shared half-word contains fields shared by all the threads. The shared counter field tracks RC activity by non-owner threads. Then, there are two flags to support the BRC algorithm: `Merged` and `Queued`. The `Merged` flag is set by the owner thread when it has merged the counters. The `Queued` flag is set by a non-owner thread to explicitly request the owner thread to merge counters. More details about the counter operations are explained later.

To prevent race conditions, BRC uses atomic operations to access the shared half-word. In some situations, the BRC algorithm requires multiple fields of the shared half-word to be updated together atomically. This is why the flags must be inside of the shared half-word.

BRC reduces the number of bits per counter from 30 bits to 14 bits, which is more than enough for RC. Many Java programs need only 7 bits [37], and we observe similar behavior in our Swift programs. BRC also reduces the number of the reserved bits used for weak reference counting and existing flags to 16 bits. Weak references occur significantly less frequently than regular RC operations, so this size is acceptable. Alternatively, we could keep the number of bits per counter unchanged to 30 by increasing the size of RCWord at the cost of adding more memory overhead. We evaluate the memory overhead of this alternative design as well in Section 7.4.

4.3 Algorithm Invariants

To understand the BRC algorithm, we start by describing its main invariants. They are described in Table 2. Recall that the value of the counter in the original RCWord (Figure 4) reflects the number of current references to the object, and must always be zero or higher. For the same reason, in BRC, invariant *I1* in Table 2 says that the sum of the biased and shared counters must always be zero or higher.

Invariant *I2* in Table 2 says that the biased counter must always be zero or higher. This is because, as we will show, as soon as the biased counter reaches zero, the owner unbiases the object. This action makes the biased counter inaccessible, and we say it *implicitly merges* the two counters into the shared counter. The owner thread also sets the `Merged` flag.

On the other hand, *I3* says that the shared counter can be negative. This is because a pair of positive and negative updates may be split between the biased and shared counters, pushing the shared counter below zero. As an example, consider two threads T1 and T2. Thread T1 creates an object and sets itself as the owner of it. It points a

Invariant Description
I1: biased + shared = total number of references to object * Must be zero or higher * If zero, object can be deallocated
I2: biased = (references added - references removed) by owner * Must be zero or higher * When it reaches 0, owner unbiases object, implicitly merging counters
I3: shared = (references added - references removed) by non-owners * Can be negative * If negative, biased must be positive, and object is placed in owner's `QueuedObjects` list so that owner can unbias it
I4: Owner only gives up ownership when it merges counters, namely: * When biased reaches zero (implicit merge) * Or when the owner finds the object in its `QueuedObjects` list (explicit merge)
I5: Object can only be placed into `QueuedObjects` list once * Placed when shared becomes negative for first time * Removed when counters are explicitly merged

Table 2: Invariants of the BRC algorithm.

global pointer to the object, setting the biased counter to one. Then, T2 overwrites the global pointer, decrementing the shared counter of the object. As a result, the shared counter becomes negative.

When the shared counter for an object becomes negative for the first time, the non-owner thread updating the counter also sets the object's `Queued` flag. In addition, it puts the object in a linked list belonging to the object's owner thread called `QueuedObjects`. Without any special action, this object would leak. This is because, even after all the references to the object are removed, the biased counter will not reach zero — since the shared counter is negative. As a result, the owner would trigger neither a counter merge nor a potential subsequent object deallocation.

To handle this case, BRC provides a path for the owner thread to explicitly merge the counters called the `ExplicitMerge` operation. Specifically, each thread has its own thread-safe `QueuedObjects` list. The thread owns the objects in the list. At regular intervals, a thread examines its list. For each queued object, the thread merges the object's counters by accumulating the biased counter into the shared counter. If the sum is zero, the thread deallocates the object. Otherwise, the thread unbiases the object, and sets the `Merged` flag. Then, when a thread sets the shared counter to zero, it will deallocate the object. Overall, as shown in invariant *I4*, an owner only gives up ownership when it merges the counters.

Invariant *I5* in Table 2 says that an object can be placed into `QueuedObjects` list only once. It is placed there when its shared counter becomes negative for the first time. After that, while its shared counter may continue to change, since the object is already marked as queued, no action is required. It will remain in the owner's `QueuedObjects` list until the owner unbiases it.

4.4 Examples of Counter Transitions

Figure 6 shows some examples of RCWord transitions in BRC. To start with, Figure 6(a) shows the RCWord structure without the Reserved field. Then, in Figure 6(b), we show the RCWord transitions for a private (i.e., thread-local) object. In this example, thread T1 allocates the object and becomes the owner thread. Next, T1 creates up to N references to the object, incrementing the biased counter up to N. Finally, T1 removes these references, decrementing the biased counter to zero, and deallocates the object.

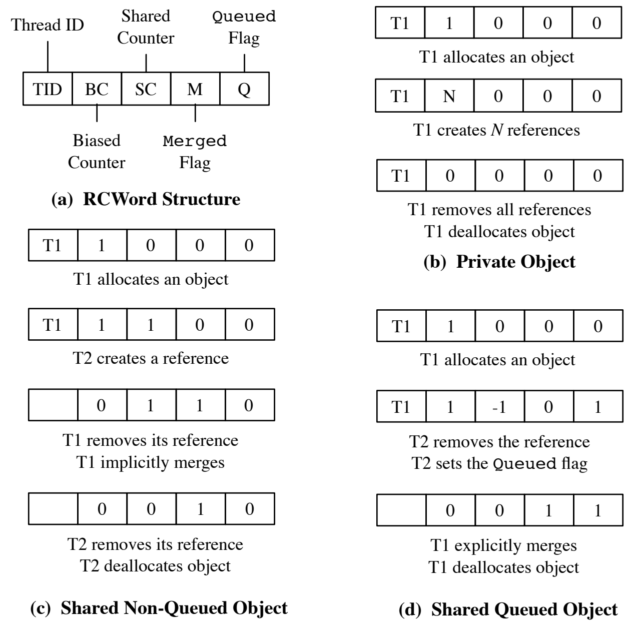

Figure 6: Examples of RCWord transitions.

In Figure 6(c), we show the RCWord transitions for a shared object that is not queued in a `QueuedObjects` list during its lifetime. Thread T1 first allocates the object and sets itself as the owner of it. Next, a second thread T2 creates a reference to the object, incrementing the shared counter. Then, T1 removes its reference to the object, decrementing the biased counter. As the biased counter becomes zero, T1 performs an implicit counter merge: it sets the `Merged` flag and unbiases the object. Later, T2 removes its reference to the object, decrementing the shared counter. Since the shared counter is zero and the `Merged` flag is set, T2 deallocates the object.

In Figure 6(d), we show the RCWord transitions for a shared object that is queued in a `QueuedObjects` list during its lifetime. Thread T1 first allocates the object and sets itself as the owner of it. Then, thread T2 overwrites the reference to the object and hence decrements the shared counter. Since the shared counter becomes negative, T2 also sets the `Queued` flag and places the object in T1's `QueuedObjects` list. Later, T1 invokes the `ExplicitMerge` operation and explicitly merges the counters, setting the `Merged` flag and unbiasing the object. Since the sum of the counters is zero, T1 deallocates the object.

4.5 BRC Algorithm

The BRC algorithm introduces several changes to a conventional RC algorithm. First, when an object is allocated, BRC saves the ID of the thread allocating the object in the RCWord, effectively biasing the object. Second, BRC modifies the RC operations (i.e., `Increment` and `Decrement`) to update one of the two RCWord counters based on which thread an object is biased to. Finally, BRC adds two new operations, `Queue` and `ExplicitMerge`, to handle a special case introduced by using two counters. In the following paragraphs, we explain these operations in detail. We use a dot notation to access the biased and shared half-words, and their fields in the algorithms. Note that the algorithms given below sacrifice performance for maximum clarity. BRC's implementation on the Swift runtime is more efficient than what is shown here.

Algorithm 2 shows BRC's `Increment` operation. It begins by checking whether the new reference is being created by the object's owner thread (line 4). If so, the owner thread continues to the `FastIncrement` procedure to increment the biased counter (line 11). Otherwise, a non-owner thread calls `SlowIncrement` and uses an atomic CAS operation to increment the shared counter (line 19).

Algorithm 2 Increment operation

```
 1: procedure INCREMENT($obj$)                    ▷ Increment the reference count of obj
 2:     $owner\_tid := obj.rcword.biased.tid$
 3:     $my\_tid := GetThreadID()$
 4:     if $owner\_tid == my\_tid$ then
 5:         $FastIncrement(obj)$                            ▷ Owner access
 6:     else
 7:         $SlowIncrement(obj)$                        ▷ Non-owner access
 8:     end if
 9: end procedure

10: procedure FASTINCREMENT($obj$)
11:     $obj.rcword.biased.counter$ += 1
12:                                         ▷ Non-atomic increment of biased counter
13: end procedure

14: procedure SLOWINCREMENT($obj$)
15:     do
16:         $old := obj.rcword.shared$                ▷ Read shared half-word
17:         $new := old$
18:         $new.counter$ += 1
19:     while $!CAS(\&obj.rcword.shared, old, new)$
20:                                         ▷ Atomic increment of shared counter
21: end procedure
```

Algorithm 3 shows BRC's `Decrement` operation. Similar to `Increment`, it first checks whether the reference is being removed by the object's owner thread (line 4). If so, the owner thread continues to the `FastDecrement` procedure to decrement the biased counter (line 11). If the resulting value of the counter is higher than zero (line 13), no further action is required. Otherwise, the biased counter is zero, and the owner thread performs an implicit merge of the counters. Specifically, it sets the `Merged` flag (line 19) by atomically updating the shared half-word (line 20). Next, the shared counter is read. If its value is zero (line 22), the object is deallocated. Otherwise, BRC unbiases the object by clearing the owner TID (line 25). Now, all future RC operations to this object will invoke either the `SlowIncrement` or the `SlowDecrement` procedures. In addition, any thread can make the decision to deallocate the object. Note that the `Deallocate` call in line 23 does not need to lock the object. This is because the last reference has been removed so no other thread can access the object.

If the `Decrement` operation is invoked by a non-owner thread, it continues to the `SlowDecrement` procedure (line 28). BRC decrements the shared counter (line 32) and, if the counter's new value is negative, BRC also sets the `Queued` flag (line 34). The shared half-word is updated atomically (line 36). If the `Queued` flag has been set for the *first* time by this invocation (line 38), BRC invokes function `Queue` to insert the object in a list to be handled later by the owner (line 40) — note that this case implies that the counters have not been merged yet, as the shared counter's value is negative. Otherwise, if the `Merged` flag is set and the shared counter is zero (line 41), BRC deallocates the object.

Algorithm 3 Decrement operation

```
 1: procedure DECREMENT($obj$)                  ▷ Decrement the reference count of obj
 2:     $owner\_tid := obj.rcword.biased.tid$
 3:     $my\_tid := GetThreadID()$
 4:     if $owner\_tid == my\_tid$ then
 5:         $FastDecrement(obj)$                  ▷ Owner access
 6:     else
 7:         $SlowDecrement(obj)$                  ▷ Non-owner access
 8:     end if
 9: end procedure

10: procedure FASTDECREMENT($obj$)
11:     $obj.rcword.biased.counter$ −= 1
12:                                   ▷ Non-atomic decrement of biased counter
13:     if $obj.rcword.biased.counter > 0$ then
14:         return
15:     end if
16:     do                                        ▷ biased counter is zero
17:         $old := obj.rcword.shared$            ▷ Read shared half-word
18:         $new := old$
19:         $new.merged$ :=True                   ▷ Set merged flag
20:     while !$CAS(\&obj.rcword.shared, old, new)$
21:                                   ▷ Atomic update of shared half-word
22:     if $new.counter == 0$ then
23:         $Deallocate(obj)$
24:     else
25:         $obj.rcword.biased.tid := 0$          ▷ Give up ownership
26:     end if
27: end procedure

28: procedure SLOWDECREMENT($obj$)
29:     do
30:         $old := obj.rcword.shared$            ▷ Read shared half-word
31:         $new := old$
32:         $new.counter$ −= 1
33:         if $new.counter < 0$ then
34:             $new.queued$ :=True               ▷ Set queued flag
35:         end if
36:     while !$CAS(\&obj.rcword.shared, old, new)$
37:                                   ▷ Atomic decrement of shared counter
38:     if $old.queued \neq new.queued$ then
39:                                   ▷ queued has been first set in this invocation
40:         $Queue(obj)$
41:     else if $new.merged$ ==True and $new.counter == 0$ then
42:                     ▷ Counters are merged and shared counter is zero
43:         $Deallocate(obj)$
44:     end if
45: end procedure
```

BRC adds two new operations, `Queue` and `ExplicitMerge` (Algorithm 4), to support a special case introduced by having two counters. Specifically, the first time that the shared counter attains a negative value, `Queue` is invoked. As indicated in Section 4.3, at this point, the biased counter has a positive value. If BRC did not take any special action, the biased counter might never be decremented to zero and, thus, the counters might never be merged, and the object might never be deallocated. This is a memory leak.

To guard against such scenarios, BRC keeps track of objects that may leak. As shown in the `Queue` procedure of Algorithm 4, the non-owner thread that first finds that the shared counter becomes negative, inserts the object in a thread-safe list belonging to the object's owner thread. The list is part of a structure called `QueuedObjects` (line 3), which is organized as per-thread lists of potentially leaked objects. Potentially leaked objects are added to the `QueuedObjects` list belonging to the object's owner thread.

At regular intervals, a thread checks its `QueuedObjects` list, to explicitly merge counters and enable object deallocation. The `ExplicitMerge` procedure of Algorithm 4 performs this operation. The procedure searches through the thread's `QueuedObjects` list and, for each object, explicitly merges its two counters. Note that this merging can only be done by the owner thread, so the procedure only accesses the `QueuedObjects` list owned by the thread invoking the procedure (line 8). For each object in the list, BRC accumulates the biased counter into the shared counter (line 12) and sets the `Merged` flag (line 14). This change is atomic (line 15). If the merged counter becomes zero, the owner deallocates the object (line 18). Otherwise, it unbiases the object (line 20) so that all future RC operations are performed on the shared counter. Once this merging is completed, it is no longer possible for the object to be leaked, and thus the owner removes the object from `QueuedObjects` in a thread-safe manner (line 22).

Algorithm 4 Extra operations

```
 1: procedure QUEUE($obj$)
 2:     $owner\_tid := obj.rcword.biased.tid$
 3:     $QueuedObjects[owner\_tid].append(obj)$
 4:                           ▷ Adds object to list belonging to owner_tid
 5: end procedure

 6: procedure EXPLICITMERGE
 7:     $my\_tid := GetThreadID()$
 8:     for all $obj \in QueuedObjects[my\_tid]$ do
 9:         do
10:             $old := obj.rcword.shared$        ▷ Read shared half-word
11:             $new := old$
12:             $new.counter$ += $obj.rcword.biased.counter$
13:                                               ▷ Merge counters
14:             $new.merged$ :=True
15:         while !$CAS(\&obj.rcword.shared, old, new)$
16:                                   ▷ Atomic update of shared half-word
17:         if $new.counter == 0$ then
18:             $Deallocate(obj)$
19:         else
20:             $obj.rcword.biased.tid := 0$      ▷ Give up ownership
21:         end if
22:         $QueuedObjects[my\_tid].remove(obj)$
23:     end for
24: end procedure
```

A given object can only be put in the `QueuedObjects` list once. This is because, before an object is taken out of the list, its counters are merged. Such merging eliminates the possibility that the shared counter become negative anymore.

Lastly, when a thread terminates, it processes the objects remaining in its `QueuedObjects` list, and de-registers itself from the `QueuedObjects` structure. Theoretically, an object can outlive its owner thread if its biased counter is positive, and has not been queued in the `QueuedObjects` list when the owner thread terminates. We handle this case as follows. When a non-owner thread makes the shared counter of an object negative, it first checks whether the object's owner thread is alive by looking-up the `QueuedObjects` structure — which implicitly records the live threads. If the owner thread is not alive, the non-owner thread merges the counters instead of queuing the object, and either deallocates the object or unbiases it.

5 PUTTING BRC IN CONTEXT

In this section, we qualitatively compare BRC to other RC algorithms. Table 4 examines the space-time trade-off of various RC implementations. Each row corresponds to a different RC implementation. The table ranks the RC implementations from 1 (lowest) to 4 (highest) in terms of performance overhead and memory overhead.

	Program Name	Multi-threaded?	Description
Client	Swift Benchmark	No	A set of 212 benchmarks covering a number of important Swift workloads designed to track Swift performance and catch performance regressions
	CryptoSwift	No	Performance tests of a Swift package for cryptography algorithms
	SwiftyJSON	No	Performance tests of a Swift package for JSON handling
	Raytrace	No	Ray tracing application
	GCBench-Single	No	Single-threaded implementation of an artificial garbage collection benchmark that creates perfect binary trees
	GCBench-Multi	Yes	Multi-threaded implementation of GCBench
	Regex-Redux	Yes	Benchmark that uses regular expressions to match and replace DNA 8-mers
Server	Perfect-JSON	Yes	JSON generator running on the Perfect framework
	Perfect-Blog	Yes	Blog engine running on the Perfect framework
	Kitura-JSON	Yes	JSON generator running on the Kitura framework
	Kitura-Blog	Yes	Blog engine running on the Kitura framework

Table 3: Client and server programs used.

Algorithm	Performance Overhead	Memory Overhead
Basic non-deferred RC	4	1 (tie)
Non-deferred RC w/ optimization	3	1 (tie)
Deferred RC (DRC)	1	3
BRC	2	2

Table 4: Ranking performance and memory overheads of RC implementations from 1 (lowest) to 4 (highest).

The first row corresponds to the basic non-deferred RC described in Section 2.1. It suffers from a high execution time overhead due to frequent atomic RC operations. However, it has a minimal memory overhead thanks to immediate reclamation.

The second row corresponds to the non-deferred RC with the optimization described in Section 2.2.2. This is Swift's RC implementation. Compared to basic non-deferred RC, the execution time overhead is dramatically reduced. This is because many unnecessary RC operations are removed at compile time. Specifically, we found that Swift removes up to 97% of RC operations in our programs. This implementation is very effective at removing RC operations for local variables. At the same time, it maintains immediate reclamation, and hence has the same minimal memory overhead as the basic non-deferred RC.

The third row corresponds to deferred RC (DRC) implementations, as described in Section 2.2.1. The performance overhead is lower, as deferral and coalescing avoid atomic RC operations during the mutation phase. However, since DRC does not perform immediate reclamation for all objects, the memory overhead is higher than the basic non-deferred RC.

The last row corresponds to BRC. While Swift's non-deferred RC with optimization is fast, it is still slower than DRC (about 20% [21]). BRC narrows this performance gap by replacing atomic RC operations with non-atomic ones in most cases. It also retains immediate reclamation for most objects in our programs. Hence, it increases the memory overhead *very little* compared to the basic non-deferred RC. We discuss BRC's impact on performance and memory in Section 7.3 and 7.4 in detail.

Overall, we believe that BRC enables a new space-time trade-off in the RC design space, different from what has been proposed thus far. Further, we believe that BRC aligns well with Swift's philosophy that emphasizes speed and low memory consumption.

6 EXPERIMENTAL SETUP

To evaluate BRC, we implement it in the Swift version 3.1.1 runtime. We evaluate the three configurations shown in Table 5. The *Original* configuration (*O*) is the unmodified Swift runtime, which implements RC with lock-free atomic operations. The *Ideal* configuration (*I*) takes *O* and eliminates all the atomic operations. In this configuration, due to data races, counters may have incorrect values. In particular, an object may be accessed after being deallocated, which may lead to a crash. We collect data from *I* only when the program runs to completion, and its output and number RC operations are same as in *O*'s execution. This ensures that *I* did not change semantics. Lastly, the biased configuration (*B*) is *O* enhanced with BRC. As a result, all of Swift's RC optimizations (which are present in *O*) are enabled in *B* by default.

Name	Configuration
O	Original: The unmodified Swift runtime
I	Ideal: *O* with no atomic operations
B	Biased: *O* enhanced with BRC

Table 5: Configurations evaluated.

Table 3 shows the client and server programs that we evaluate. The official Swift Benchmark Suite [6] consists of a set of tests which cover important Swift workloads. The suite is designed to track Swift performance and catch performance regressions. CryptoSwift [2] and SwiftyJSON [10] are popular Swift packages for cryptography and JSON handling, respectively. We also use a Swift version of ray tracing [9]. GCBench [1] is an artificial garbage collection benchmark which creates and discards perfect binary trees to estimate the collector performance. We use single-threaded and multi-threaded implementations of GCBench. Lastly, Regex-Redux is a regular expression benchmark that uses regular expressions to match and replace DNA sequences.

Our server programs are based on two popular server-side frameworks for Swift, namely Perfect [4] and Kitura [3]. For each framework, we run a blog engine that returns random images and blog posts for each request, and a JSON generator that returns a JSON dictionary of random numbers for each request [5]. For the server programs, we measure throughput instead of execution time.

We run our experiments on a desktop machine with an Intel Core i7 processor and 16 GB of memory running Ubuntu 16.04 LTS. The processor has four cores cycling at 3.50 GHz. Each experiment is run 10 times and the average is reported.

	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 6: Reference counting statistics.

7 EVALUATION

7.1 Characterization

We start by investigating the overhead of RC in the Original (*O*) and Biased (*B*) configurations. Table 6 shows various metrics of RC behavior during execution for both configurations. For reference, Column 3 repeats the data shown in Table 1 about the percentage of shared objects in each program. Recall that we consider an object as shared if its reference counter updates come from more than one thread. Next, Columns 4-6 refer to the *O* configuration, while columns 7-12 refer to the *B* configuration.

Columns 4 and 5 show the number of object allocations per μsecond and the number of RC operations per μsecond, respectively. The latter are counter increments and decrements. Based on the data in these two columns, Column 6 shows the average number of RC operations per object. We can see that, discounting Raytrace, there are 3–36 RC operations per object in client programs, and 8–28 in server programs.

Column 7 shows the number of RC operations per μsecond in the *B* configuration. Due to the improved performance of *B*, these numbers are higher than in *O* for all the programs.

Column 8 shows the percentage of RC operations to shared objects, and Columns 9 and 10 the percentage of RC operations to the biased and shared counters, respectively. We see that only a small percentage of the RC operations are performed on shared objects (7.03% in client programs and 13.27% in server programs), and an even smaller percentage are performed on shared counters (1.72% in client programs and 10.15% in server programs). The outlier is Regex-Redux, where nearly 50% of the RC operations are on shared objects, and 12% use the shared counter. Overall, the small fraction of the RC operations that use the shared counter is the reason for the speed-ups of *B* over *O*; only such operations use atomic instructions.

Column 11 shows the percentage of the total objects that are queued. On average, this number is 0.00% in client programs, and 2.35% in server programs. This number is very small, in part because the percentage of objects that are shared (Column 3) is already small. Finally, Column 12 shows the percentage of RC operations that set the `Queued` flag and add the object to the `QueuedObjects` list. We see that this is a rare event, which occurs 0.00% of the time in client programs, and 0.16% in server programs. Overall, queuing in the `QueuedObjects` list is a negligible overhead.

7.2 Latency of RC Operations

We measure the time it takes to increment a reference counter in the different configurations. For this measurement, we create kernels that repeatedly increment the counter in a loop. Therefore, the operations have a near-perfect cache behavior. In addition, these kernels are single-threaded and, therefore, the measured times do not include contention. Overall, our experiments measure best-case timings.

Table 7 shows the time to perform a counter increment in our different configurations: *O*, *I*, and *B*. For the *B* configuration, we show the operation time for the owner thread and for non-owner threads. As shown in the table, the increment operation takes 13.84 ns in *O* and 5.77 ns *I*. Hence, the use of atomic operations slows down the operation by 2.40x. In *B*, the owner's increment takes only 6.28 ns, while the non-owner increment takes 15.57 ns. Ideally, the former should be as fast as *I*, while the latter should take as long as *O*. In practice, BRC adds some overhead to each of these operations, as the TID and various flags are checked before performing the increment. Consequently, *B* owner takes 8.8% longer than *I*, and *B* non-owner takes 12.5% longer than *O*.

Configuration	Time (ns)
Original	13.84
Ideal	5.77
Biased (operation by owner)	6.28
Biased (operation by non-owner)	15.57

Table 7: Time of counter increment operations.

7.3 Performance Improvement

In this section, we evaluate the performance improvements attained by BRC. Figure 7 shows the execution time of the client programs for the *O*, *I*, and *B* configurations, normalized to the *O* configuration. On average, *B* reduces the execution time by 22.5% over *O*. This is a substantial speed-up, which is attained inexpensively in software by improving the RC algorithm. Further, this speed-up implies that a large fraction of the RC overhead in *O* has been eliminated. Indeed, as shown in Figure 3, client programs spend on average 42% of their time in RC operations. With *B*, it can be shown that we eliminate more than half of such RC time.

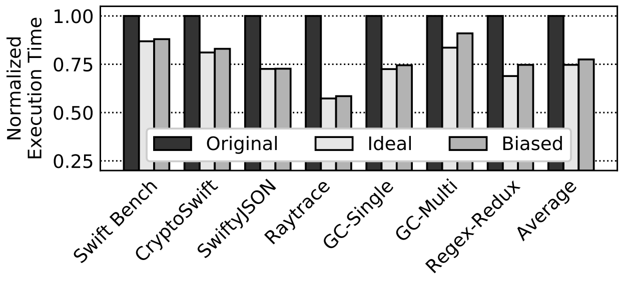

Figure 7: Execution time of client programs under the *O*, *I*, and *B* configurations.

We see that, on average, the *B* configuration is within 3.7% of the *I* configuration. This difference is smaller than the 8.8% difference observed in Table 7 between the *B* (owner) and *I* increment operations. This is due to Amdahl's law, as programs only spend a fraction of their time performing RC operations.

Figure 8 shows the throughput of the server programs under the *O* and *B* configurations, normalized to the *O* configuration. We do not show data for the *I* configuration because running these programs without atomic operations causes frequent program crashes due to premature object deallocations. The figure shows that *B* attains a substantial average throughput increase of 7.3% over *O*. This improvement is smaller than the 22.5% average reduction in the execution time of the client programs. This is expected, given that the overhead of RC in Figure 3 is higher in the client programs than in the server ones.

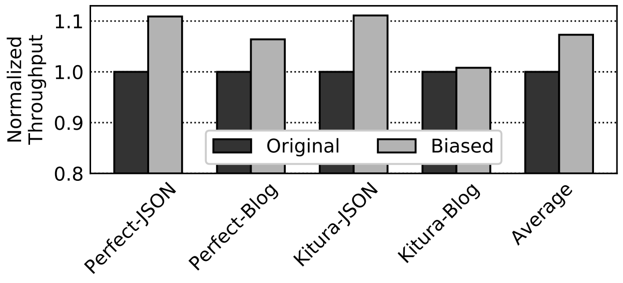

Figure 8: Throughput of the server programs under the *O* and *B* configurations.

7.4 Memory Overhead

In this section, we evaluate BRC's memory overhead by comparing the peak memory usage of the *O* and *B* configurations. Figure 9 shows the peak memory usage of these configurations normalized to the peak memory usage of *O*. Recall from Section 4.2 that our BRC design does not increase the size of the per-object RCWord. Hence, the additional memory overhead of *B* comes from the use of the `QueuedObjects` structure.

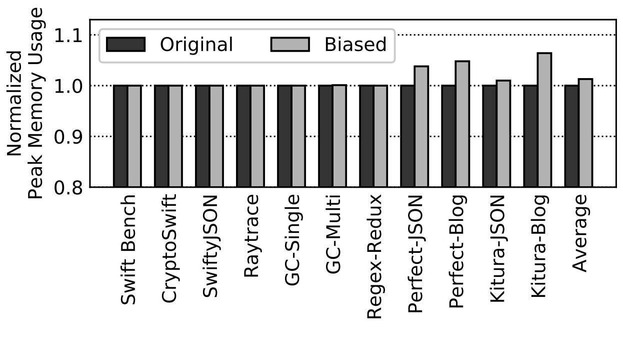

Figure 9: Peak memory usage under the *O* and *B* configurations.

We observe that, on average, *B* has only a 1.5% higher memory overhead than *O*. Single-threaded programs (i.e., the first 5 programs) have no additional memory overhead in *B* because the shared counter is not used and, consequently, there are no queued objects. While GCBench-single and Regex-Redux are multithreaded, they have negligible additional memory overhead because they have almost no queued objects (Column 11 of Table 6). The server programs have only a small fraction of queued objects and, therefore, their additional memory overhead is on average about 4%.

We also measure the memory overhead in the alternative *B* implementation described in Section 4.2, where we add an additional 64-bit word to the object header to preserve 30-bit counters. In this case, the peak memory usage in *B* can be shown to be, on average, a modest 6% higher than in *O*.

7.5 Sensitivity Study

To simulate a worst-case scenario for BRC, we create a synthetic benchmark where we can control the number of queued objects. In the benchmark, a main thread creates 1,000,000 objects, creating a reference to each object, then performs a fixed amount of dummy computation, and finally removes any remaining references to the objects. In parallel, a second thread removes the references to $1,000,000 \times R$ objects allocated by the main thread. When a shared counter becomes negative, the second thread adds the corresponding object to the main thread's `QueuedObjects` list. In our experiments, we vary *R*, which we call Ratio of Queued Objects. Figure 10 shows the execution time and peak memory usage under the *B* configuration as we vary *R*, normalized to the *O* configuration.

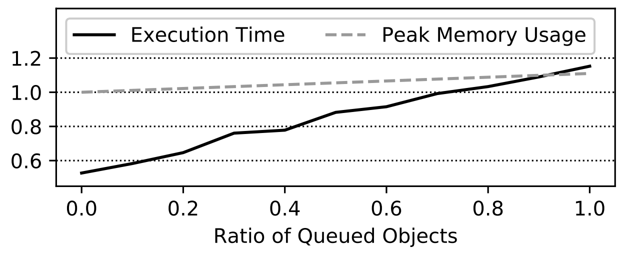

Figure 10: Normalized execution time and normalized peak memory usage of the *B* configuration as we vary the number of queued objects.

Our results show that, for *B* to perform worse than the *O* configuration, one needs 75% or more queued objects. In reality, as shown in Table 6, the percentage of queued objects in our programs is much lower than this break-even point.

We also measure the additional memory overhead of *B* for this benchmark. As we see in the figure, the additional memory overhead is kept low, under 12% over the *O* configuration. This is because the counter merging and object dequeuing happen frequently enough that reclamation of dead queued objects is not delayed too much. Note that, for our programs in Table 6, the percentage of queued objects is very small and, therefore, the additional memory overhead of BRC is small.

8 RELATED WORK

There have been many works [19, 25, 29, 34–36, 42] which try to limit the amount of overhead to acquire uncontested locks. While BRC is inspired by biased locking [25], it is not a straightforward re-application of biased locking. BRC proposes an efficient biasing technique tailored to RC by exploiting the fact that *RC does not require strong exclusivity like locking*. BRC lets multiple threads access the same object concurrently, by dividing an object's reference count into two counters. In biased locking, this is not possible. BRC also makes ownership revocation very cheap. This is because ownership is typically voluntarily revoked in BRC and only requires one CAS. On the other hand, ownership revocation is extremely expensive in biased locking. It is triggered by a non-owner thread, and requires inter-thread communication through OS signals or safepoints. This is the main drawback of biased locking.

Subsequent works on biased locking [19, 29, 34–36, 42] improve on the original work by making ownership revocation more efficient, enabling ownership transfer, or determining when it is best to bias an object. It is possible to apply such ownership transfer techniques to BRC. In future work, we plan to implement similar techniques to better support various program behaviors.

Many prior works on RC [12, 14, 15, 18, 21–23, 26, 30, 32, 37–40] focus on reducing the number of RC operations. They are are briefly summarized in Section 2.2, and compared to BRC in Section 5. Another category of works attempt to efficiently detect and remove cyclic references [13, 24, 27, 31, 33]. Swift solves this problem through weak references, an approach popular in previous literature [11, 16]. We believe that BRC can also be integrated into RC implementations with cyclic reference detection and removal algorithms.

Joao et al. [20] propose hardware support for RC. They augment the cache hierarchy to gradually merge RC operations. Due to the delay of merging in hardware, their technique does not support immediate reclamation. In contrast, BRC supports immediate reclamation in most cases.

Recently, Ungar et al. [41] propose a compiler-assisted dynamic optimization technique for RC in Swift. It is similar to BRC in that it dynamically replaces atomic RC operations with non-atomic ones. It adds checks before stores to conservatively capture escaping objects, and uses atomic RC operations for escaped objects only. Compared to BRC, their technique maintains the immediate reclamation property of non-deferred RC, while BRC relaxes this for queued objects. However, their technique uses more atomic operations than BRC due to its conservative escape detection, and its lack of the notion of biased threads. In addition, it increases the overhead of the store barrier to detect and recursively mark escaping objects. Finally, it does not fully support all of Swift's function argument passing semantics.

9 CONCLUSION

This paper proposed Biased Reference Counting (BRC), a novel approach to speed-up non-deferred RC for garbage collection. BRC is based on the observations that most objects are mostly accessed by a single thread, and that atomic operations have significant overheads. BRC biases each object toward a specific thread. Further, BRC adds a second counter to the object header, enabling the owner thread to have its own counter. These changes allow the owner thread of each object to perform RC operations without atomic operations, while the other threads atomically update the other counter. BRC correctly manages the merging of these two counters, handling all corner cases.

We implemented BRC in the Swift programming language runtime and evaluated it with various client and server programs. We found that BRC accelerated non-deferred RC. Specifically, it reduced the average execution time of client programs by 22.5%, and improved the average throughput of server programs by 7.3%.

ACKNOWLEDGMENTS

This work was supported in part by NSF under grant CCF 15-27223.

REFERENCES

[1] An Artificial Garbage Collection Benchmark. http://www.hboehm.info/gc/gc_bench.html
[2] CryptoSwift. https://github.com/krzyzanowskim/CryptoSwift
[3] Kitura: A Swift Web framework and HTTP Server. http://www.kitura.io/
[4] Perfect: Server-side Swift. http://perfect.org/
[5] Server-side swift benchmarks. https://github.com/rymcol/Server-Side-Swift-Benchmarks-Summer-2017
[6] Swift Benchmark Suite. https://github.com/apple/swift/tree/master/benchmark
[7] Swift Has Reached 1.0. https://developer.apple.com/swift/blog/?id=14
[8] Swift Programming Language. https://swift.org/.
[9] Swift Version of Ray Tracing. https://github.com/rnapier/raytrace
[10] SwiftyJSON. https://github.com/SwiftyJSON/SwiftyJSON
[11] T. H. Axford. 1990. Reference Counting of Cyclic Graphs for Functional Programs. *Comput. J.* 33, 5 (1990), 466–472.
[12] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. 2001. Java Without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. 92–103.
[13] David F. Bacon and V. T. Rajan. 2001. Concurrent Cycle Collection in Reference Counted Systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*. 207–235.
[14] Jeffrey M. Barth. 1977. Shifting Garbage Collection Overhead to Compile Time. *Commun. ACM* 20, 7 (1977), 513–518.
[15] Stephen M. Blackburn and Kathryn S. McKinley. 2003. Ulterior Reference Counting: Fast Garbage Collection Without a Long Wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA '03)*. 344–358.
[16] David R. Brownbridge. 1985. Cyclic reference counting for combinator machines. In *Conference on Functional Programming and Computer Architecture*. 273–288.
[17] George E. Collins. 1960. A Method for Overlapping and Erasure of Lists. *Commun. ACM* 3, 12 (Dec. 1960), 655–657.
[18] L. Peter Deutsch and Daniel G. Bobrow. 1976. An Efficient, Incremental, Automatic Garbage Collector. *Commun. ACM* 19, 9 (1976), 522–526.
[19] David Dice, Mark Moir, and William Scherer III. 2003. *Quickly Reacquirable Locks.* Technical Report. Sun Microsystem Laboratories.
[20] José A. Joao, Onur Mutlu, and Yale N. Patt. 2009. Flexible Reference-counting-based Hardware Acceleration for Garbage Collection. In *Proceedings of the 36th*

Annual International Symposium on Computer Architecture (ISCA '09). 418–428.

[21] Pramod G. Joisha. 2006. Compiler Optimizations for Nondeferred Reference-Counting Garbage Collection. In *Proceedings of the 5th International Symposium on Memory Management (ISMM '06)*.

[22] Pramod G. Joisha. 2007. Overlooking Roots: A Framework for Making Nondeferred Reference-counting Garbage Collection Fast. In *Proceedings of the 6th International Symposium on Memory Management (ISMM '07)*. 141–158.

[23] Pramod G. Joisha. 2008. A Principled Approach to Nondeferred Reference-counting Garbage Collection. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*. 131–140.

[24] Richard E. Jones and Rafael D. Lins. 1993. Cyclic Weighted Reference Counting Without Delay. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe (PARLE '93)*. 712–715.

[25] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. 2002. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *Proc. of the 17th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*.

[26] Yossi Levanoni and Erez Petrank. 2001. An On-the-fly Reference Counting Garbage Collector for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. 367–380.

[27] A. D. Martínez, R. Wachenchauzer, and R. D. Lins. 1990. Cyclic Reference Counting with Local Mark-scan. *Inf. Process. Lett.* 34, 1 (1990), 31–35.

[28] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195.

[29] Tamiya Onodera, Kikyokuni Kawachiya, and Akira Koseki. 2004. Lock Reservation for Java Reconsidered. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP '04)*.

[30] Young Gil Park and Benjamin Goldberg. 1991. Reference Escape Analysis: Optimizing Reference Counting Based on the Lifetime of References. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '91)*. 178–189.

[31] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. 2007. An Efficient On-the-fly Cycle Collection. *ACM Trans. Program. Lang. Syst.* 29, 4 (2007).

[32] Harel Paz and Erez Petrank. 2007. Using Prefetching to Improve Reference-counting Garbage Collectors. In *Proceedings of the 16th International Conference on Compiler Construction (CC'07)*. 48–63.

[33] Harel Paz, Erez Petrank, David F. Bacon, Elliot K. Kolodner, and V. T. Rajan. 2005. An Efficient On-the-fly Cycle Collection. In *Proceedings of the 14th International Conference on Compiler Construction (CC'05)*. 156–171.

[34] Filip Pizlo, Daniel Frampton, and Antony L. Hosking. 2011. Fine-grained Adaptive Biased Locking. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. 171–181.

[35] Ian Rogers and Balaji Iyengar. 2011. Reducing Biased Lock Revocation by Learning. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. 65–73.

[36] Kenneth Russell and David Detlefs. 2006. Eliminating Synchronization-related Atomic Operations with Biased Locking and Bulk Rebiasing. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. 263–272.

[37] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. 2012. Down for the Count? Getting Reference Counting Back in the Ring. In *Proceedings of the 2012 International Symposium on Memory Management (ISMM '12)*. 73–84.

[38] Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. 2014. Fast Conservative Garbage Collection. In *Proceedings of the 2014 ACM International Conference on Object-oriented Programming Systems Languages, and Applications (OOPSLA '14)*. 121–139.

[39] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S. McKinley. 2013. Taking off the Gloves with Reference Counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-oriented Programming Systems Languages, and Applications (OOPSLA '13)*. 93–110.

[40] David Ungar. 1984. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. 157–167.

[41] David Ungar, David Grove, and Hubertus Franke. 2017. Dynamic Atomicity: Optimizing Swift Memory Management. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '17)*. 15–26.

[42] Nalini Vasudevan, Kedar S. Namjoshi, and Stephen A. Edwards. 2010. Simple and Fast Biased Locks. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. 65–74.