Dynamic Partial Deadlock Detection and Recovery via Garbage Collection

Georgian-Vlad Saioc[*]
Aarhus University
Aarhus, Denmark
gvsaioc@cs.au.dk

I-Ting Angelina Lee[†]
Washington University in St. Louis
St. Louis, Missouri, United States
angelee@wustl.edu

Anders Møller
Aarhus University
Aarhus, Denmark
amoeller@cs.au.dk

Milind Chabbi
Programming Systems Group
Uber Technologies, Inc.
Sunnyvale, California, United States
milind@uber.com

Abstract

A challenge of writing concurrent message-passing programs is ensuring the absence of partial deadlocks, which can cause severe memory leaks in long-running systems. The Go programming language is particularly susceptible to this problem due to its support of message passing and ease of lightweight concurrency creation.

We propose a novel dynamic technique to detect partial deadlocks by soundly approximating liveness using the garbage collector's marking phase. The approach allows systems to not only detect, but also automatically redress partial deadlocks and alleviate their impact on memory.

We implement the approach in the tool Golf, as an extension to the garbage collector of the Go runtime system and evaluate its effectiveness in a series of experiments. Preliminary results show that the approach is effective at detecting 94% and 50% of partial deadlocks in a series of microbenchmarks and the test suites of a large-scale industrial codebase, respectively. Furthermore, we deployed Golf on a real service used by Uber, and over a period of 24 hours, effectively detected 252 partial deadlocks caused by three programming errors.

CCS Concepts: **• Software and its engineering → Garbage collection**; **Message passing**; **Deadlocks**; **• Theory of computation → Concurrency**.

Keywords: Go, message passing concurrency, channels, blocking errors, dynamic analysis

[*]Also with Programming Systems Group, Uber Technologies, Inc..
[†]Work done in part as a visiting Research Scientist at Uber.

ACM ISBN 979-8-4007-1079-7/2025/03
https://doi.org/10.1145/3676641.3715990

ACM Reference Format:
Georgian-Vlad Saioc, I-Ting Angelina Lee, Anders Møller, and Milind Chabbi. 2025. Dynamic Partial Deadlock Detection and Recovery via Garbage Collection. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3715990

1 Introduction

Go is a modern programming language that features a powerful concurrency model, where lightweight threads, called goroutines, are scheduled and managed by the Go runtime. Goroutines may communicate and synchronize by using either shared memory, protecting shared access of resources with locks, or through message passing [10], where channels are supported as first-class concurrency objects. Alongside scheduling, the Go runtime is also supplied with a garbage collector (GC) to facilitate automatic memory management.

Goroutines may block, e.g., when attempting to acquire a held lock or sending a message over a channel before a receiving party is available. If all goroutines are blocked, the Go runtime issues a fatal error signaling a global deadlock. In contrast, partial deadlocks, also known as goroutine leaks, occur when some but not all goroutines are permanently blocked for all possible future executions. Global deadlocks are quite rare, but partial deadlocks appear frequently in many real-world Go programs due to unexpected execution paths and thread schedules.

Partial deadlocks may not only result in expected actions failing to happen but also cause memory leaks. Previous work [22, 29, 32] has shown that, in long-running systems, partial deadlocks covertly increase CPU utilization, or lead to out-of-memory exceptions and system crashes. Figure 1 shows the number of blocked goroutines over a period of time for a Go service running in production at Uber, which is known for using thousands of micro-services [15, 33] written in Go. In this case, most of the blocked goroutines are likely

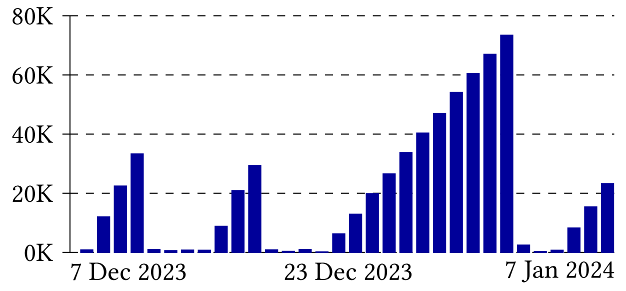

Figure 1. Number of blocked goroutines over time for a production Go service suffering from partial deadlocks. Weekday redeployments hide the leak, but the numbers spike during weekends and holidays.

due to partial deadlocks, and the drops correspond to service restarts after weekends and holidays.

The current implementation of the Go runtime does not, unfortunately, provide any built-in mechanism to detect and address partial deadlocks. Since the standard Go garbage collector behavior is unaware of partial deadlocks, it is unable to reclaim the memory of deadlocked goroutines and memory resources reachable via their stacks.

The state-of-the-art in dynamic partial deadlock detection includes Goleak [22] and LeakProf [22, 23]. Goleak is designed for finding defects based on running test suites and checking for lingering goroutines after each test run. While effective, it is not applicable in production environments with long running services. LeakProf, on the other hand, may be used for production services, but is susceptible to both false positives and false negatives. Other approaches aim to detect partial deadlocks statically, but also with mixed success for large-scale Go programs (see Section 7). We need a mechanism that detects partial deadlocks in production environments, which requires the analysis to be lightweight. Moreover, we need automatic garbage collection of the partial deadlocks and resources they hold to avoid production-time issues such as out-of-memory, which requires the analysis to be sound.

The key insight of our proposed solution is that memory reachability, as defined by the Go GC, soundly over-approximates the liveness of concurrency operations. Informally, if a goroutine is blocked at a concurrency operation, for example, on a channel send operation, but the involved channel is not reachable in memory from a live goroutine, the sender can never be unblocked, and hence is deadlocked.

The approach is implemented as the tool Golf (Goroutine Leak Fixer), an extension of the Go GC that allows it to compute reachable liveness. Using Golf, the extended Go runtime reveals several elusive partial deadlocks, and actively curbs their performance impact by forcefully shutting down affected goroutines.

In this paper we make the following contributions:

1. We present a general approach to partial deadlock detection and recovery at runtime by piggybacking on garbage collection, demonstrating that memory reachability soundly over-approximates concurrency operation liveness.
2. We show that the technique can be implemented by minimally extending Go's garbage collector, in the form of the tool Golf. This requires attention to a number of technical details in the Go runtime.
3. We evaluate the approach in a series of experiments involving three categories of programs: (1) 73 microbenchmarks with known defects from prior work [22, 32], containing a total of 121 **go** instructions that create goroutines which may deadlock, (2) a collection of 3 111 Go packages in Uber's codebase, each with a corresponding test suite, and (3) one real service that is a part of a large micro-service system at Uber. The results show that Golf successfully detects 94% of the partial deadlocks in the microbenchmarks. By running the 3 111 test suites using Golf, 180 of 357 known partial deadlocks are reported. The experiments with the real service revealed 3 programming errors that caused 252 partial deadlocks over a period of 24 hours. Furthermore, the experiments show that the implementation is robust and that the runtime overhead is negligible, demonstrating that the approach is usable in production environments.

2 Background

The Go runtime manages its own lightweight threads, termed *goroutines*, which interact either by passing messages via channels or by accessing shared memory. Goroutines may *block* when performing operations on channels, locks, and other concurrency constructs, or when performing system calls, e.g., network communication, or IO. Collectively, we refer to channels, locks, etc. as *concurrency objects* and their operations as *concurrency operations*.

A blocked goroutine is put in a waiting state until the conditions necessary for unblocking are met. A goroutine blocked at a concurrency operation may only be unblocked by another goroutine performing a (typically complementary) operation on the same concurrency object. In this paper, we focus strictly on blocking caused by concurrency operations, instead of system calls. In what follows, we provide a brief overview of the main concurrency features in Go.

Message Passing. *Channels* are bounded message queues that can be accessed by multiple goroutines. Syntactically, sending a message `m` over a channel `ch` is written as the statement `ch <- m`, whereas receiving a message from `ch` is denoted by the expression `<-ch`. Channels have the type **`chan`** `T`, where `T` is the type of messages, and are allocated by calling the **`make`** function with the desired channel type and an optional integer representing the channel *capacity*.

Listing 1. Example **select** statement.

```go
select {
case x := <-ch1: ... // receive from ch1
case ch2 <- y: ... // send to ch2
default: ... // both ch1 and ch2 are blocked
}
```

Channels with an unspecified or zero capacity are *unbuffered*. Goroutines block when sending or receiving messages over an unbuffered channel until they synchronize with another goroutine performing the complementary operation. Conversely, a channel with a positive capacity is *buffered*. Sending messages over a buffered channel only blocks when the buffer, the maximum length of which is given by the capacity, is full. Receiving messages from a buffered channel only blocks when the buffer is empty. Go developers most commonly use unbuffered channels [7, 22].

Send and receive operations over unallocated channels, i.e., with a **nil** value, block forever. A channel may be *closed* via the built-in function **close**. Receive operations on a closed channel do not block, and return the zero value of the channel payload type once the channel buffer is empty. Send operations on a closed channel cause the runtime to panic.

Channels may be iterated over in a **range** loop, where the guard is a receive operation that must be executed before the loop body for each iteration. Channel iteration blocks at the guard until messages are available, and may only be terminated by closing the channel, after emptying its buffer.

Go implements non-deterministic choice via **select** statements. Each case of a **select** statement is a statement sequence guarded by a channel operation. Executing a **select** statement blocks until at least one of its cases is unblocked, at which point it non-deterministically chooses one of the enabled cases to execute. A **select** statement is non-blocking when supplied with a **default** case, which is executed if all other cases are blocked. When declared with zero cases, a **select** statement blocks indefinitely.

Listing 1 shows an example **select** statement that simultaneously tries to receive from ch1 or send to ch2, with a **default** case if neither is possible.

Sharing Memory. Goroutines may safely access shared memory by using the synchronization features defined in the standard sync package, i.e., mutexes (write or read-write), condition variables, and wait groups. Goroutines block when attempting to acquire a held mutex, by invoking its locking methods (Lock for both standard mutex types, or RLock for read-write mutexes).

Wait groups are non-negative counters that may be incremented by invoking the Add method with an integer, or decremented by invoking Done. A goroutine invoking the Wait method blocks until the counter is zero.

Listing 2. Example wait group and locks in Go.

```go
func work(w *sync.WaitGroup, m *sync.Mutex) {
  ... // doing some work
  m.Lock()
  ... // accessing a shared resource
  m.Unlock()
  w.Done()
}
func main() {
  var w sync.WaitGroup
  var m sync.Mutex
  for i := 0; i < 100; i++ {
    w.Add(1)
    go work(&w, &m)
  }
  w.Wait()
}
```

Blocking over condition variables is similarly achieved by invoking the Wait method. When invoked, the Signal method of a condition variable randomly unblocks a waiting goroutine, if one exists, or otherwise has no effect. The Broadcast method unblocks all waiting goroutines.

In Listing 2, the main function spawns 100 work goroutines, using a wait group to determine when they are done and a mutex to avoid race conditions.

Liveness and Deadlocks. A goroutine g that can eventually make progress is *semantically live*, denoted $LIVE(g)$.[1] All unblocked (runnable) goroutines are immediately identifiable as semantically live. Blocked goroutines are only semantically live if there exists a possible future execution path where they are unblocked. If no such future exists, the goroutine is deadlocked.

The program encounters a *global deadlock* whenever all goroutines are deadlocked, which is identified by the ordinary Go runtime, and followed by prematurely terminating execution. A *partial deadlock* occurs if some but not all goroutines are deadlocked.

3 Motivating Example

Using Go's concurrency features correctly can be difficult, and subtle programming errors often result in partial deadlocks [29]. Listing 3 contains a minimal program, derived from a real-world example, that is susceptible to partial deadlocks. Line 22 defines an interface, GoFuncManager. Values implementing the interface, such as goFuncManager objects (line 30), can be created by invoking NewFuncManager (line 29). Each object embeds two channels at fields e and d

[1]To avoid confusion, we reserve the term liveness specifically for semantic liveness. Liveness in memory is referred to as memory reachability (Section 4).

Listing 3. If the condition on line 51 is met, the method `WaitForResults` is never called (line 54). Channels `e` and `d` are therefore never closed, leading the goroutines created by `NewFuncManager` (lines 35–38) to deadlock.

```
22 type GoFuncManager interface {
23   WaitForResults() ([]any, error)
24 }
25 type goFuncManager struct {
26   e chan error
27   d chan any
28 }
29 func NewFuncManager() GoFuncManager {
30   gfm := &goFuncManager{
31     e: make(chan error),
32     d: make(chan any),
33   }
34   go func() {
35     for err := range gfm.e { ... }
36   }()
37   go func() {
38     for data := range gfm.d { ... }
39   }()
40
41   return gfm
42 }
43 func (gfm *goFuncManager) WaitForResults()
44   ([]any, error) {
45   close(gfm.e)
46   close(gfm.d)
47   ...
48 }
49 func ConcurrentTask() {
50   gfm := NewFuncManager()
51   if ... {
52     return
53   }
54   gfm.WaitForResults()
55 }
```

(lines 31–32). The invocation of `NewFuncManager` also creates two goroutines, each iterating over one of the embedded channels (lines 35–38). Every `GoFuncManager` object created as such must eventually invoke the `WaitForResults` method to close the channels, (line 44), and allow the iterating goroutines to terminate. This implicit contract is not satisfied by function `ConcurrentTask` (line 49). On certain execution paths (line 51), method `WaitForResults` is never called, causing the iterating goroutines to eventually deadlock.

Diagnosing this issue is especially challenging when the faulty behavior is spread out across multiple libraries. The underlying implementation of the `GoFuncManager` interface is not exported, hiding it from library users, which may easily overlook the caller-side contract of `NewFuncManager`.

The regular Go mark-and-sweep garbage collector marks all goroutines in the system, and any memory reachable through their stacks, including the channels causing the partial deadlock. However, a close examination of the faulty code reveals that, once `ConcurrentTask` exits, the program eventually reaches a state where the channels created by `NewFuncManager` are only reachable in memory through the stacks of deadlocked goroutines. This presents an opportunity to detect the partial deadlock, by refining the garbage collection process to only mark goroutines that may still be live, allowing the runtime to observe the partial deadlocks.

Using the technique presented in the following sections, the partial deadlock is detected and reported, allowing the developers to fix the issue, and the involved memory resources are reclaimed thereby preventing the leak from draining the system resources.

4 Approach

In this section, we present an approach to dynamically detect partial deadlocks using a modified garbage collector. We first establish the necessary terminology about memory management in Go, then define a notion of reachable liveness, and finally explain how reachable liveness can be computed by a garbage collector.

Terminology. We define all memory used by a Go program as the set $\mathcal{M}$, where each element denotes an object in memory. The memory of a Go program consists of goroutines, and global or heap data. The set of all active goroutines in a program at a given point in execution is denoted as $\mathcal{G}$. For simplicity, each goroutine represents a single memory object, such that $\mathcal{G} \subseteq \mathcal{M}$.

If an object stores the address of another object, we say that there is a *reference* from one to the other. References are useful for modeling *memory reachability*, which provides an over-approximation of the set of memory objects that may be used in the remaining execution of the program. The relation $REF(a, b)$ denotes that a directly or transitively has a reference to b. If $REF(g, a)$, where $g \in \mathcal{G}$, then a is transitively referenced by the stack of g.

The GC treats certain memory objects as *roots*, i.e., intrinsically reachable, and denoted by set $R \subseteq \mathcal{M}$. Other objects are reachable if they are transitively referenced by the roots. Succintly, object a is reachable if $a \in R \vee \exists b \in R.\ REF(b, a)$. Garbage collectors compute memory reachability at runtime and reclaim any unreachable memory.

In the ordinary Go GC, the root set includes all goroutines and global data, independently of whether the goroutines are currently blocked. To simplify definitions, we define a "main goroutine," $g_0 \in \mathcal{G}$, such that $g_0 \in R$, and it references all global data, i.e., $\forall a \in \mathcal{D}.\ REF(g_0, a)$, where $\mathcal{D} \subseteq \mathcal{M}$ is the set of all global objects. We may thus define the ordinary root set used by the Go GC as simply $R = \mathcal{G}$.

4.1 A Definition of Reachable Liveness

Semantic liveness is not computable, but it can be over-approximated using memory reachability in combination with the status of each goroutine. The key observation is that a goroutine is deadlocked if it is blocked on a set of concurrency objects, all of which are unreachable in memory via any goroutine which is not deadlocked. Based on this observation, we next define a notion of reachable liveness, denoted as $LIVE^{+}(g)$, that over-approximates $LIVE(g)$.

We define goroutine states in terms of whether they are blocked by performing a concurrency operation on at least one concurrency object, as defined in Section 2. For each goroutine g, the set $B(g) \subseteq \mathcal{M}$ contains the concurrency objects of the operation that g is blocked on. If $B(g) = \emptyset$, then g is runnable (goroutines blocked at system calls are also deemed runnable). Otherwise, $B(g)$ contains the concurrency objects of the blocking operation, e.g., one channel for send and receive operations, or multiple channels, one for each case, in a blocking **select** statement. For convenience, we define $B(g) = \{\epsilon\}$ for goroutines blocked at operations over **nil** channels, or **select** statements with no cases, where ϵ is unreachable in memory.

A goroutine g is *reachably live* if it is runnable, or if at least one of its blocking concurrency objects is transitively referenced by another live goroutine, expressed as the following constraint:

$$LIVE^{+}(g) = \begin{array}{l} \Big(B(g) = \emptyset\Big) \vee \\ \Big(\exists\, o \in B(g), g' \in \mathcal{G} \colon REF(g', o) \wedge LIVE^{+}(g')\Big) \end{array}$$

Reachable liveness across all goroutines in the system is defined as the least solution satisfying the above constraints.

Any goroutine g for which $LIVE^{+}(g)$ does not hold is deemed deadlocked.

4.2 Reachable Liveness via Garbage Collection

We now describe how to adapt tricolor mark-and-sweep garbage collectors [6], such as the one implemented by Go, to compute reachable liveness. A mark-and-sweep GC computes the least fixed point of memory reachability at runtime, by marking all roots and their transitive references as reachable.

The crux of the approach is to start with a minimal root set of reachably live goroutines and use the garbage collector to progressively mark memory and expand the root set with additional reachably live goroutines until it reaches a fixed point, directly corresponding to the definition of $LIVE^{+}$:

1. The GC first constructs the *initial root set*, R'_0, by only including goroutines in a runnable state:
$$R'_0 = \{g \in \mathcal{G} \mid B(g) = \emptyset\}$$
2. The GC then performs a *mark iteration*, by marking all objects transitively referenced by the latest root set (R'_i for the i'th iteration) as reachable. Marking is performed as in the ordinary GC.
3. Once marking concludes, the GC expands the root set, by including goroutines blocked at operations over reachable concurrency objects:
$$R'_{i+1} = R'_i \cup \{g \in \mathcal{G} \mid \exists\, o \in B(g), g' \in R'_i \colon REF(g', o)\}$$

Listing 4. The child goroutine deadlocks, but the global visibility of ch prevents detection.

```
var ch = make(chan int) // Global channel.

func main() {
  go func() { ch <- 1 }()
  ...
}
```

Iterative marking (step 2) and root set expansion (step 3) repeat until a fixed point is reached, i.e., $R'_k = R'_{k+1}$ for some k. The resulting root set R'_k contains all reachably live goroutines; thus, any $g \notin R'_k$ is guaranteed to be deadlocked.

We describe the implementation as it relates to the Go runtime in more detail in Section 5.

4.3 Soundness and Completeness

The approach is sound, meaning that all semantically live goroutines are also reachably live, i.e., $LIVE(g) \implies LIVE^{+}(g)$ for all $g \in \mathcal{G}$. Soundness is essential, as false positives may lead to the deallocation of memory that is still in use and thereby cause unexpected crashes.

If $LIVE(g)$ holds, then there exists a future execution where g is eventually runnable. The proof that $LIVE^{+}(g)$ holds proceeds by induction in the number of other goroutines involved in making g runnable in that execution.

For the base case, if g is currently runnable, then $B(g) = \emptyset$, so $LIVE^{+}(g)$ trivially holds by definition.

For the inductive step, g is currently blocked but eventually unblocked by some goroutine g'' that has access to some $o \in B(g)$. Therefore, g'' must somehow eventually obtain the reference to o, possibly via some other goroutines, so $REF(g', o)$ holds for some goroutine g' (possibly g'') that is live, i.e., $LIVE(g')$. The number of goroutines involved in making g' runnable must be smaller than the number of goroutines involved in making g runnable (since neither g nor g' can contribute to make g' runnable). By the induction hypothesis we then have $LIVE^{+}(g')$. By the definition of $LIVE^{+}$, we conclude that $LIVE^{+}(g)$.

The converse property, completeness, can be expressed as $LIVE^{+}(g) \implies LIVE(g)$. This property does not hold; some deadlocked goroutines may not be detected with this approach, i.e., false negatives are possible. Listings 4 and 5 show examples of programming patterns found in real-world code that lead to false negatives.

Listing 5. The goroutine at line 72 is always live, exposing the `dispatcher.ch` channel to the heap and preventing detection of the partial deadlock at line 80.

```
62 type dispatcher struct {
63   ch          chan struct{}
64   ticks       int
65 }
66 func newDispatcher() *dispatcher {
67   d := &dispatcher{
68     ch:     make(chan struct{}),
69     ticks:  0,
70   }
71   go func() { // Heartbeat goroutine
72     for ; ; time.Sleep(time.Second) {
73       d.ticks++
74     }
75   }()
76   return d
77 }
78 func main() {
79   d := newDispatcher()
80   go func() { d.ch <- struct{}{} }()
81   if ... { return }
82   <-d.ch
83 }
```

A deadlock caused by a global channel (Listing 4) cannot be detected, as the channel is always reachable in memory. For example, the goroutine at line 59 is always considered live, even if `ch` is no longer used.

A more pernicious type of false negative involves runaway live goroutines (Listing 5). In the example, whenever a dispatcher is created, a heartbeat goroutine is created at line 72 and increments the `ticks` field of the `dispatcher` object every second. One possible execution involves a goroutine deadlocking at line 80 if the check at line 81 succeeds. However, the heartbeat goroutine is always reachably live, in turn exposing the `*dispatcher` object and its `ch` field as reachable. It is, therefore, assumed that the heartbeat goroutine may eventually unblock the sending goroutine.

5 Implementation

In this section, we present the implemention of the technique as the tool Golf, an extension to the regular Go GC (version 1.22.5). Our implementation is carefully crafted so that it incurs little overhead, in terms of both the memory footprint and time overhead. While the technique is conceptually simple, integrating Golf into the Go runtime is a delicate process which poses a number of nuances and challenges that must be accounted for. The resulting extension fits neatly into the existing Go runtime framework and consists of only ~600 effective lines of code.

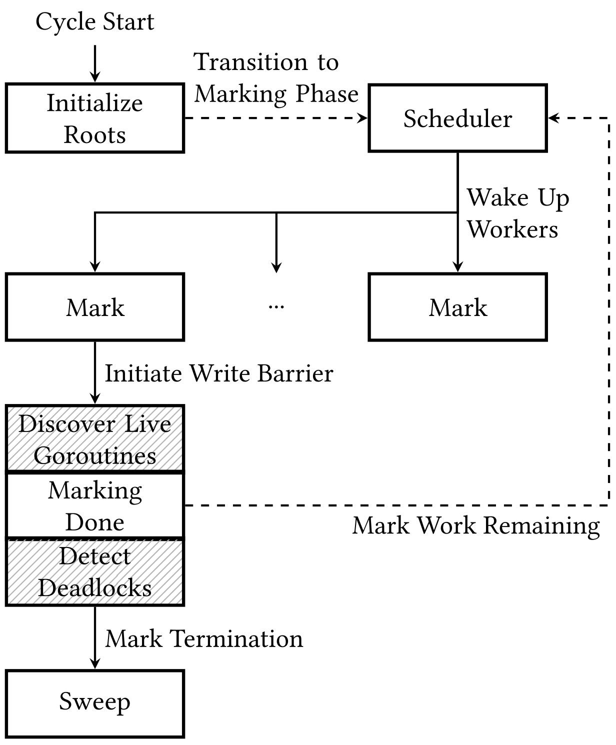

Figure 2. Garbage collection cycle. The regular cycle components and phases (white) are extended with additional phases (hatched) to enable partial deadlock detection.

Go uses a concurrent tricolor non-generational mark-and-sweep garbage collector [5, 6, 25, 34] where several phases may run concurrently with user code. Figure 2 illustrates the garbage collection cycle, with the regular phases in white, and the extensions in hatched background. We first present an overview of the regular GC behavior, and then describe the modifications required by Golf.

5.1 Regular Go Garbage Collection

The regular Go garbage collection process is organized in cycles that involve the following phases:

1. *Initialization*: The GC performs the initial setup by unmarking all objects and preparing the root set (global data and all goroutines). The initiator of the cycle waits for the completion of the previous cycle before triggering this phase.
2. *Marking*: The runtime scheduler wakes up a number of goroutines that have been designated as mark workers. Every worker performs a number of marking jobs by pulling them from a set of local and shared queues, before going back to sleep. The last worker to go to sleep transitions the GC to the next phase.
3. *Marking done*: The GC pauses execution and drains all marking queues. If reachable unmarked objects are

found, it transitions back to marking phase. Otherwise, it transitions to the next phase.

4. *Sweeping*: All unmarked objects are reclaimed.

5.2 Modifications for Partial Deadlock Detection

Partial deadlock detection follows the approach presented in Section 4.2. Initially, the runnable goroutines are set up as roots. Golf then iteratively marks reachable objects and extends the set of roots with reachably live goroutines. After the fixed point has been reached, Golf reports all unmarked goroutines as deadlocked. This process is performed between GC marking and sweeping.

As the algorithm is iterative, the Golf GC may restart the marking phase more often than the ordinary GC. Nevertheless, Golf performs exactly the same amount of marking work as the ordinary Go GC, i.e., it performs the same number of pointer traversals in order to mark objects.

In Golf, the number of mark iterations depends on the blocking pattern of goroutines. In the worst case, we have a daisy chain of n goroutines where the status of each one depends on one another. Discovering some g_i as reachable, for $i = 1 \ldots n-1$, causes the channel blocking g_{i+1} to get marked, and for g_{i+1} to be discovered in the next round. That is, each g_i is added to the root set sequentially, leading to n mark iterations. Nevertheless, this scenario is extremely unlikely and the overall marking work in aggregate remains the same.

5.3 Overhead of Golf

The two extra steps in Golf incur some overhead. A running program with N goroutines requires $O(N^2 + NS)$ time to discover live goroutines and $O(N)$ time to detect deadlocks, where S is the number of pairings between goroutines and blocking concurrency objects. We shall explain the discovery overhead in more detail, where each blocked goroutine g is checked to see if any of the concurrency objects o that g is blocked on have become reachable.

The Go runtime already maintains a linked list of channels that g is blocked on (potentially more than one channel if g is blocked at a `select` statement). Additionally, we extended the goroutine data structure to keep track of the semaphore or conditional variable that may block g. Any blocking concurrency object o is stored in the heap,[2] and can be checked for whether it is marked in constant time. If Golf cannot determine whether o is marked, it conservatively assumes that o is not stored on the heap, and therefore reachable, e.g., as a global object. Since there may be a total of S pairs of goroutines and blocking concurrency objects that need to be checked per round, in the (unlikely) worst case where the goroutines get unblocked sequentially, the overhead is $O(N^2 + NS)$.

[2]If a concurrency object is declared as a stack variable, it will be promoted to the heap by Go. [2]

An astute reader may observe that a blocking channel always stores references to the goroutines blocked by it. Thus, we could potentially reduce the overhead to $O(N^2)$ by simply checking whether the blocked goroutine was marked transitively alongside the channel, avoding the S factor in the NS term. Another approach would be adding blocked goroutines to the root set on the fly, as concurrency objects they are attached to are marked. This would reduce the overhead even further, as the marking phase would be incrementally extended with reachably live goroutines without the need to check whether a restart is necessary. However, while possible, we have not yet implemented these optimizations, as the GC does not immediately mark any related blocked goroutines, and it would overall require more invasive changes to the GC.

5.4 Implementation Details

The broad changes described above are supported by several smaller modifications, as described below.

Inspecting Goroutine States to Assess Liveness. Goroutines are deemed reachably live based on their state. Naturally, goroutines that are running or runnable are considered reachable live. However, the Go runtime may put goroutines created internally to sleep for a variety of reasons, e.g., sleeping mark workers or I/O interactions with the operating system. The regular Go runtime decorates waiting goroutines with a descriptive *wait reason*. Since Golf is only directed at partial deadlocks caused by Go users, wait reasons are useful for distinguishing between user and internal goroutines. Only goroutines decorated with wait reasons denoting operations over channels or primitives in the standard `sync` package may deadlock. All other goroutines are assumed to always be reachably live.

Address Obfuscation. To ensure that goroutines are not prematurely marked before they are reachably live, Golf applies a simple bitmask to their addresses. The regular Go runtime has two global data structures that store goroutine objects: an array maintaining all the existing allocated goroutine objects, and a treap [4] of in-use semaphores, which may have a back pointer to goroutines blocked on the given semaphore. In the regular Go GC, such global objects do not affect garbage collection, as all goroutines are part of the root set, regardless. In Golf, however, blocked goroutine objects (and their stacks) must only be marked after being shown as reachably live. This is achieved by flipping the highest-order bit of the pointers to goroutine objects stored in global data structures, and instructing the marking phase to ignore masked addresses. Once a goroutine is determined to be reachably live, the corresponding pointer is unmasked and scheduled for marking. Since primitives in the `sync` library use semaphores in their underlying implementation, Golf also applies the bitmask to semaphore addresses stored in the global treap to hide them from the GC.

Goroutine Reuse. The Go runtime represents each goroutine as an object of type *g, which captures its stack, status, and other properties that allow the scheduler to safely coordinate its internal processes and user goroutines. One optimization of the runtime includes the reuse of *g objects. When a goroutine is terminated, the runtime does not deallocate its corresponding *g object, but instead partially cleans it up and decomissions it by setting its state as *dead*. The runtime minimizes allocations of *g objects by reusing those in the dead state, only allocating new *g objects if more are needed than currently available.

Goroutines may only be terminated in the regular runtime when they reach an exit point. The regular cleanup process makes several assumptions about the state of *g properties, which do not necessarily hold when reclaiming deadlocked goroutines. For example, a goroutine may have had the fields of its corresponding *g object mutated by the blocking **select** statement causing the deadlock, depending on the underlying implementation. To ensure parity between *g objects decomissioned due to deadlocks and those decomissioned via regular termination, Golf comes with a special cleanup procedure that resets additional fields to neutral values.

Semaphores. The runtime performs additional bookkeeping of goroutines blocked over semaphores by internally storing the relation between the semaphore and the goroutine in a global treap table indexed by (masked) semaphore addresses. This must be supplemented with logic for removing deadlocked goroutine entries from the semaphore treap. All *g objects are extended with (masked) references to the addresses of the semaphore they are blocked on.

5.5 Preserving Go Semantics

It is important that the Go runtime with Golf preserves the semantics of ordinary Go modulo partial deadlocks. In most cases, reporting and then reclaiming deadlocked goroutines is semantically equivalent to the original program, and even desirable, as the performance impact of partial deadlocks is minimized. However, there are cases where reclaiming deadlocked goroutines may lead to unexpected behavior, which is where Golf instead only reports the deadlocks. This applies to programs that rely on finalizers, i.e., procedures attached to heap objects that are executed once the object is reclaimed by the GC. In the regular Go runtime, the effects of finalizers attached to objects reachable via deadlocked goroutines are never observed. However, naively reclaiming deadlocked goroutines may unassumingly trigger finalizers, leading to programmer-visible semantic differences.

Listing 6 exemplifies a finalizer (line 88) attached to the address of local variable containing a list of integers (line 87). The finalizer prints the average of all integers in the list (line 94) once the goroutine is scheduled for the GC. However, the list of integers is received over channel ch (line 97), which is created (line 85) and returned (line 100) by the function PrintAverage. If callers of PrintAverage neglect to use the channel it returns, the goroutine at line 86 may eventually deadlock. Without our extensions the finalizer will not be invoked; with our extensions, however, reclaiming the goroutine would invoke the finalizer, which in turn will lead to a division-by-zero error.

Listing 6. The goroutine at line 86 sets a finalizer at line 88 for slice vs. Reclaiming the goroutine if it deadlocks at line 97 would trigger the finalizer and lead to a runtime exception.

```
84  func PrintAverage() chan []int {
85    ch := make(chan []int)
86    go func() {
87      var vs []int
88      runtime.SetFinalizer(&vs,
89        func(vs *[]int) {
90          var vsum int
91          for _, v := range *vs {
92            vsum += v
93          }
94          fmt.Println("Avg.:", vsum / len(*vs))
95        })
96
97      vs := <-ch
98    }()
99
100   return ch
101 }
```

To avoid such issues and preserve Go semantics, deadlock detection and reclaiming is split between two GC cycles. In the first pass, deadlocked goroutines are reported and placed in a pending-to-reclaim state and scheduled for marking. While marking resources reachable only from deadlocked goroutines, the GC checks for the existence of finalizers. If any are discovered, the deadlocked goroutine is placed in a *deadlocked* state, which is treated by Golf as live for any future GC cycles. This ensures that deadlocked goroutines with finalizers are consistently reachable in memory, while also avoiding reporting the same deadlock more than once.

6 Evaluation

We evaluate Golf in a series of experiments aimed at answering the following research questions:

RQ1 To what extent can the technique unveil partial deadlocks in:
  (a) known microbenchmarks,
  (b) test suites of a large scale industrial codebase, and
  (c) a real service.

RQ2 What is the performance overhead of the technique?

6.1 Experimental Setup

RQ1 (a) Microbenchmarks. For the first experiment, Golf is deployed on 73 microbenchmarks taken from prior work [22, 32]. The microbenchmark suite contains 121 **go** instructions which may produce deadlocks, 8 and 113 for Saioc et al. [22] and Yuan et al. [32], respectively. We refer to all microbenchmarks as set $\mathcal{B}$.

Every microbenchmark $b \in \mathcal{B}$ is a code fragment that distills unique programming patterns prone to partial deadlocks. However, benchmarks may not exhibit the defect in every run due to non-deterministic execution introduced by the runtime scheduler or timing issues, a phenomenon known as *flakiness* [9]. Each microbenchmark is provided with a flakiness score from 1 (for deterministic bugs) to 10 000.

In order to reproduce bugs for each b, we construct a standalone Go program that concurrently runs n instantiations of b, where n is determined by the flakiness score. We configure the program to terminate after five seconds. We run each benchmark 100 times. We repeat the entire experiment with different number of virtual cores, configurable in the Go runtime via the `GOMAXPROCS` environment variable.

We performed the experiment for **RQ1** (a) on a 2021 M1 CPU MacBook Pro, with 10 cores and 32 GB of RAM.

RQ1 (b) Large enterprise codebase test suites. We deployed Golf on Uber's code repository of over 100 million lines of code and selected a subset of 3 111 Go packages, totalling 1.8 million lines of code. The packages were selected if they contained Go concurrency features. We built and ran the test suite of each package via continuous integration pipelines on four machines in parallel each having a 24-core Intel Cascade Lake CPU with 384 GB DRAM running Linux 6.1.53.

For comparison, we rely on the open-source Goleak [3, 22] tool, which inspects the runtime state of all goroutines when the test suite terminates and reports any unterminated goroutines. All goroutines involved in partial deadlocks are unterminated at the end of the process, but not all unterminated goroutines are involved in partial deadlocks. For example, Goleak not only flags goroutines blocked at IO operations, but even runaway live goroutines; for fairness, we exclude these categories from the comparison with Goleak.

We configure Golf to only monitor deadlocks, without reclaiming the goroutines, and compare the results of Goleak and Golf within the same test execution. We run each test only once. By design, all partial deadlocks discovered by Golf are also reported by Goleak. However, not all goroutine leaks reported by Goleak are detected by Golf.

In this context, we compare the number of *total* and *deduplicated* partial deadlock reports produced by both tools. The total number of reports counts all individual partial deadlocks detected across all test suites. Individual reports are then deduplicated by pairing the source location of the blocking operation and the source location of the **go** instruction that created the goroutine, aggregated across all packages. We perform this deduplication because the same code location (e.g., as part of a third-party library) may be involved in multiple partial deadlocks when exercised from different callers.

RQ1 (c) Real service. In this experiment, we deploy a real service built with the Golf runtime extension, and observe its behavior over a 24-hour period during which it serves real-time requests. Partial deadlocks, if found, are collected via the existing logging infrastructure of Uber. In this setting, there is no ground truth to compare against, but the number of revealed partial deadlocks indicates the practical utility of Golf in real-world settings.

6.2 Results

Table 1 aggregates the results of running Golf on the microbenchmark suite, as described in Section 6.1. Column **Benchmark line** denotes the name of the microbenchmark, and, after the colon (:), the line number of a **go** instruction that creates the deadlocked goroutines. The number of virtual cores usable by the Go runtime for a set of repeated runs is given by the **1**, **2**, **4**, and **10** columns. Each entry contains a value from 0 to 100, indicating the number of runs in which a partial deadlock was detected for the given **go** instruction. Column **Total** shows for each **go** instruction with expected partial deadlocks the percentage of runs during which a partial deadlock was detected across all configurations. For all benchmarks where Golf had 100% success rate, we show a single **Remaining benchmarks** row at the bottom. The last row **Aggregated (%)** shows the detection rate percentage across all microbenchmarks under each runtime configuration. The cell at the intersection with the **Total** column sums up the detection percentage across all microbenchmarks and runtime configurations.

RQ1 (a) Microbenchmarks. The results show that, across all runs and runtime configurations, Golf detected 100% of the partial deadlocks in the 6 microbenchmarks from Saioc et al. [22], and 94% of the partial deadlocks in the 67 microbenchmarks from Yuan et al. [32], with an overall detection rate of 94.75%. Notably, Golf was able to detect a known deadlock at each of the 121 potentially deadlocking **go** instructions in the microbenchmarks in at least one run.

RQ1 (b) Large enterprise codebase test suites. Goleak reported a total of 29 513 individual partial deadlocks in the test suites of Uber's codebase, deduplicated to 357 reports. Of those, Golf detected 17 872 (60%), which were deduplicated to 180 (50% of 357) distinct reports.

For each of the 180 deduplicated reports produced by Golf, we also measure how many individual deadlocks were detected, relative to Goleak, and present our findings in Figure 3. On the x-axis, each Golf deduplicated report is uniquely identified by a number from 1 to 180. The y-axis

Table 1. RQ1 (a) Golf results for microbenchmarks with different levels of parallelism. Each entry is the number of executions during which a partial deadlock was detected at the given source location.

Benchmark line	**Virtual cores**				**Total**
	1	**2**	**4**	**10**	
`cockroach/6181:58`	98	94	99	99	97.50%
`cockroach/6181:65`	98	97	99	99	98.25%
`cockroach/7504:170`	100	99	100	100	99.75%
`cockroach/7504:177`	100	99	100	100	99.75%
`etcd/7443:96`	0	0	0	1	0.25%
`etcd/7443:128`	0	0	0	3	0.75%
`etcd/7443:215`	0	0	0	3	0.75%
`etcd/7443:221`	0	0	0	3	0.75%
`etcd/7443:225`	0	0	0	3	0.75%
`grpc/1460:83`	100	100	97	97	98.50%
`grpc/1460:85`	100	100	97	97	98.50%
`grpc/3017:71`	0	98	100	100	74.50%
`grpc/3017:97`	0	97	100	100	74.25%
`grpc/3017:106`	0	97	100	100	74.25%
`hugo/3261:54`	100	100	100	83	95.75%
`hugo/3261:62`	100	100	100	83	95.75%
`kubernetes/1321:52`	100	100	100	99	99.75%
`kubernetes/1321:95`	100	100	100	99	99.75%
`kubernetes/10182:95`	100	100	100	99	99.75%
`kubernetes/11298:20`	100	100	99	100	99.85%
`kubernetes/11298:106`	100	100	99	100	99.85%
`kubernetes/25331:79`	100	98	99	99	99.00%
`kubernetes/62464:115`	100	100	95	95	97.50%
`kubernetes/62464:117`	100	100	95	95	97.50%
`moby/27282:65`	99	45	91	96	82.75%
`moby/27282:213`	99	45	91	96	82.75%
`moby/33781:39`	100	100	96	92	97.00%
Remaining 60 benchmarks: 94 potentially deadlocking **`go`** instructions					100.00%
Aggregated (%)	93	94	95	95	**94.75%**

shows the ratio between the number of Golf and Goleak individual reports for the given deduplicated report, where a ratio of 100% indicates that Golf was able to detect all partial deadlocks reported by Goleak. By computing the area under the curve, we infer that when faulty code exhibits partial deadlocks via one or more tests, Golf finds them in 82% of the cases where Goleak also finds them.

As expected, Goleak is more effective at detecting partial deadlocks in test suites, owing to its design. Golf is, nevertheless, able to detect a significant portion of the partial deadlocks in the test suites. Across the deduplicated reports, Golf finds all the individual partial deadlocks reported by Goleak in 103 (55%) cases.

The efficacy of Golf depends on the scheduling of GC cycles relative to the occurrence of leaks, and the termination of the test suite. While we strategically injected calls to the GC within tests, the GC is nonetheless up to the runtime

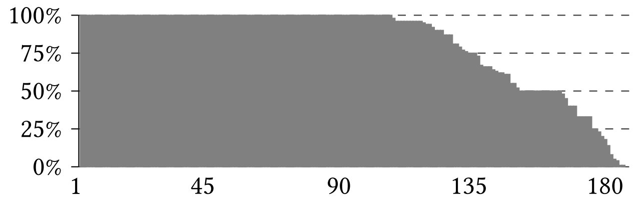

Figure 3. Ratio of individual partial deadlock reports between Golf and Goleak for each deduplicated Golf report.

Listing 7. Simplified partial deadlock from the real service.

```
102 func (s *controller) SendEmail() chan struct{} {
103   done := make(chan struct{})
104   safego.Go(func() {
105     defer func() { done <- struct{}{} }()
106     ...
107   })
108   return done
109 }
110
111 func (s *controller) HandleRequest() {
112   s.SendEmail() // Channel not used
113 }
```

scheduler. Furthermore, unlike Golf, Goleak may only be used in test suites, suggesting that the two techniques are complementary, i.e., Golf can be deployed in production to detect partial deadlocks in real systems that Goleak might have missed due to inadequate test coverage or flakiness.

RQ1 (c) Real service. For this experiment, we deployed Golf on five instances running a real service used in Uber. The deployment persisted over a period of 24 hours, during which Golf detected 252 individual partial deadlocks.

By inspecting the stack traces of deadlocked goroutines, we narrowed the errors to three source locations in the original code, all of which are represented by Listing 7. The `SendEmail` method creates a channel, done, (line 103) and a goroutine (created via the `safego.Go` wrapper function at line 104) that performs an asynchronous task. By using **`defer`**, the goroutine sets up a message to be sent over done after it completes its task (line 105), at which point it will block until a receiver for done is found. `SendEmail` returns done (line 108), and leaves it to the caller to receive a message from the channel. However, the `HandleRequest` method invokes `SendEmail` without reading the message of done, causing the task goroutine to deadlock. The error was not discovered during testing by Goleak due to insufficient coverage.

RQ2 Performance penalty. We measure the performance penalty in experiments under similar conditions as in Section 6.1.

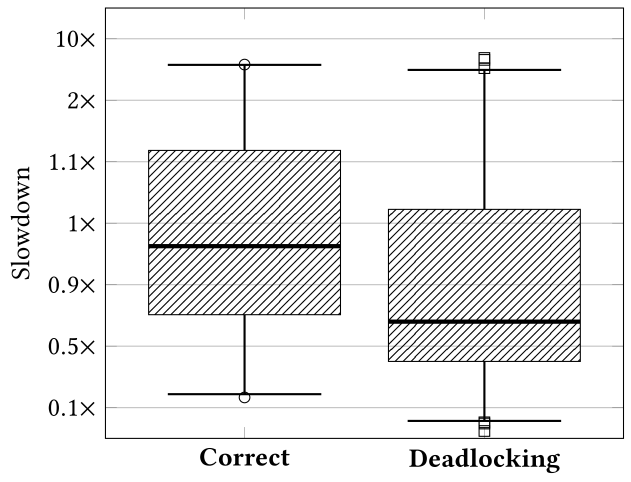

Figure 4. Slowdown of the GC marking phase for Golf compared to the baseline for microbenchmarks.

Microbenchmarks. We compare the average duration of the GC marking phase between the baseline Go and Golf across 5 repeated runs for one virtual core. Specifically, we measure the CPU time, such that the duration of the marking phase is agnostic to interleavings at the level of the Go runtime. We consider the average duration accetable, as 97% of microbenchmark executions complete the same number of GC cycles between the baseline and Golf. For completeness, we include fixed versions for 32 of the microbenchmarks, for a total of 105 programs. We omit other performance metrics, as they do not exhibit any significant deviation.

We collect aggregate metrics over the slowdown or speedup of Golf compared to the baseline, both for correct and deadlocking microbenchmarks in Figure 4. Surprisingly, the results show that Golf may be faster than the baseline in many cases, even for correct programs, where the median slowdown for correct programs is 0.96×, and the minimum slowdown is even 0.13×, from 876μs to 112μs. However, in the worst case, Golf also incurs a significant cost, with slowdowns as high as 4.8×, from 113μs to 538μs. For deadlocking programs, the median slowdown is 0.71×, with the minimum slowdown as low as 0.04×, from 3.3ms to 136μs. This is expected for programs with partial deadlocks, as the marking phase of the GC is unburdened by the need to mark a significant amount of memory. There is, nevertheless, the risk of a slowdown, the worst being 5.87×, from 136μs to 798μs.

Regardless, in terms of absolute duration, we reinforce the notion that, even with the occasionally steep slowdown, the marking phase of Golf still completes within only 10ms. For correct examples, Golf took at most 2ms to conclude the marking phase, a 3.27× slowdown over the 619μs taken by the baseline. For deadlocked examples, Golf took at most 5.5ms to conclude the marking phase, a 1.06× slowdown over the 5.2ms of the baseline.

Enterprise codebase test suites. The test suites took $\sim$ 12 hours on four machines in parallel on Uber's large repository. The running time is noisy, as machines are shared by other jobs, which is why we do not perform an exhaustive comparison of the overhead of Golf against a baseline. We observed no significant deviation from standard running times.

Services under controlled settings. We further evaluated Golf overhead by running a typical Uber service. The service comprises 1.7 million lines of code, which includes packages for handling remote procedure calls (RPCs), authentication, metrics, logging, tracing, and rate limiting, among others. The service has an endpoint and each request makes one downstream RPC and uses a parallelism package to process a directed acyclic graph of sub-tasks in parallel. Crucially, each request spawns a goroutine, and the parent communicates with the child goroutine over two channels. The parent and the child goroutines each allocate a hash map of 100K entries. The parent waits for the child using a **`select`** statement; and returns if there is a message on either of the channels. However, the child goroutine may deadlock trying to send on both channels one after another owing to a "double send" partial deadlock pattern [22]. We control the leaking frequency for our experimental purposes.

We ran experiments on a 48-core (2-socket, 24 cores each, 2-way SMT [30]) AMD EPYC at 2.4 GHz, with 396 GB of DDR memory running Linux 6.1. We allocated eight cores to the server, four to the client, and used 32 concurrent TPC connections. The client sent requests for 30 seconds following a 5-second warm-up period. The server was exercised under the following scenarios: a child goroutine leaks in 0% and 10% of requests. Table 2 presents throughput, latency, memory, and GC metrics. The client does not leak, and only runs using the baseline Go.

Without leaks, baseline and Golf performed similarly between the client and the server except for the GC pauses, which worsened in Golf. With a 10% leak, Golf resulted in a 9% higher throughput and reduced tail latency by $\sim$ 1.5× for the client. On the server side, memory usage was lowered by $\sim$ 49×.

The primary performance penalty for Golf stems from the duration of GC pause times (`PauseTotalNs`), particularly the stop-the-world (STW) phase required to complete the marking phase. On average, the GC pause time per GC cycle (`PauseTotalNs/NumGC`) for Golf is $\sim$ 2.5× higher than the baseline. Some overhead is expected, as deadlocked goroutines are reported and forcefully shut down under stop-the-world conditions. However, the penalty remains comparable across different leak scenarios (0% to 10%), indicating that memory leaks do not significantly impact the STW overhead in practice. Notably, even in the 10% leaking scenario, where the total GC pause is $1.96E + 08$ nanoseconds, this accounts for only 0.65% of the total 30-second execution time, suggesting a mild impact overall. Furthermore, the Go

Table 2. Performance impact of Golf on a service under local controlled testing. Values in bold denote the variant with a significantly better performance for each metric. Server GC metrics are collected using the standard Go runtime API for GC metrics, `MemStats` [1]; the corresponding metric field is denoted in `teletype` font. For **Throughput**, if **B/G** < 1.0, then Golf performs better than the baseline; for latency and memory metrics, if **B/G** > 1.0, then Golf performs better than the baseline.

	Metric	Leaks in **0%** requests			Leaks in **10%** requests		
		Base (B)	**Golf (G)**	**B/G**	**Base (B)**	**Golf (G)**	**B/G**
Client	Throughput (req./s)	66.45	66.56	1.00	59.15	**64.92**	0.91
	P50 latency (ms)	307	310	0.99	324	320	1.01
	P90 latency (ms)	414	423	0.98	669	**445**	1.50
	P95 latency (ms)	452	461	0.98	759	**482**	1.58
	P99 latency (ms)	542	529	1.02	929	**575**	1.62
	P99.9 latency (ms)	627	587	1.07	978	**643**	1.52
	P99.995 latency (ms)	645	593	1.09	982	**651**	1.51
	Maximum latency (ms)	659	612	1.08	982	**658**	1.49
Server	Stack spans (MB) (`StackInuse`)	1.87	1.90	0.98	2.88	**2.06**	1.40
	Heap objects allocated (MB) (`HeapAlloc`)	7.61	7.27	1.05	1,328.94	**27.35**	48.59
	Reachable heap objects (MB) (`HeapInuse`)	40.50	39.52	1.02	1,424.66	**61.90**	23.02
	No. of objects (`HeapObjects`)	4.54E+04	4.52E+04	1.00	3.98E+07	**6.48E+05**	61.47
	GC fractional CPU utilization (%) (`GCCPUFraction`)	24%	24%	1.04	30%	26%	1.13
	GC pause time (ns) (`PauseTotalNs`)	1.09E+08	2.87E+08	0.38	3.26E+07	1.96E+08	0.17
	No. of GC cycles (`NumGC`)	292.00	289.00	1.01	96.00	223.00	0.43
	Pause time per cycle (ns) (`PauseTotalNs/NumGC`)	**3.73E+05**	9.92E+05	0.38	**3.40E+05**	8.78E+05	0.39

Table 3. Performance impact of Golf on a real service.

		Latency (ms)	**CPU Usage (%)**
P50	Baseline	51 ± 136	1.46 ± 0.55
	Golf	53.65 ± 138	1.51 ± 0.62
P99	Baseline	414 ± 467	3.16 ± 2.08
	Golf	464 ± 512	3.12 ± 1.81

runtime constrains the fraction of CPU time allocated to GC (`GCCPUFraction`), trading GC speed for increased memory usage [5]. As a result, at a 10% leak rate, the baseline exhibits fewer GC cycles (`NumGC`) and lower total GC pause time (`PauseTotalNs`). However, Golf amortizes the GC's marking work, thereby reducing its required fractional utilization.

Services under production settings. Finally, to determine the overhead in practice, we measured the response latency and CPU utilization of the real service, with and without Golf over a period of 32 hours. The service emits latency and utilization metrics every three minutes. We measure the median (50th percentile) and the tail (99th percentile) in both cases, averaged over all data points along with the standard deviation, as shown in Table 3.

By analyzing the two performance metrics, which are also heavily subjected to noise and diurnal traffic patterns, it becomes apparent that Golf does not impinge on the performance of real-world systems in practice.

The empirical evaluation suggests that, while Golf incurs some overhead for correct programs, it is tolerable in practice. Furthermore, we emphasize that, throughout these experiments, the GC with the Golf extension performed deadlock detection at every cycle. If, for example, deadlock detection were only performed every 10th GC cycle, then the overhead would be reduced even further, turning it practically negligible. This would come at no cost to the efficacy of Golf, as it would still be capable of detecting the same number of partial deadlocks.

7 Related Work

Several approaches for tackling partial deadlocks have already been proposed, in the form of static, dynamic, and hybrid analysis techniques.

Static techniques include modeling Go processes as regular expressions and synchronous communication as Brzozowski derivatives [18, 26]. Other approaches propose reasoning about the liveness of programs with both synchronous and asynchronous communication via encodings to session types [13, 14, 19]. Go2Pins [12] translates Go programs to the PINS interface used by LTL model checkers. Several static approaches rely on splitting programs into fragments, i.e., sets of functions that comprise operations on concurrency objects, to improve precision and scalability. GCatch [17] leverages SMT solvers to find partial deadlocks in program fragments encoded as SMT formulas that model the order of operations and channel semantics. Goat [31] detects partial deadlocks in program fragments via abstract interpretation. Gomela [8] translates Go fragments to Promela to find partial deadlocks and safety violations via model checking [11]. Among these approaches, GCatch, Goat and Gomela have been shown to work on real-world

Go programs. Unlike Golf, they may report false positives, and they do not actively enable termination and garbage collection of deadlocked goroutines.

Several dynamic analysis technique have also been proposed. Goleak [22] examines the runtime state before a test suite terminates execution and reports any lingering goroutines. LeakProf [22, 23] extracts the goroutine profiles of running profiles and issues warnings for concurrent operations with a high concentration of blocked goroutines. GFuzz [16] uses fuzzing to reorder and prioritize **`select`** statement case branches, such that tests may explore additional execution paths. It also proposes extensions to the Go runtime that allow it to represent the ownership of channels by goroutines as a graph, which can be explored to detect partial deadlocks. In contrast to Golf, these extensions work independently of the GC, may report false positives, and incur a nontrivial overhead. It may be interesting in future work to combine the fuzzing approach of GFuzz with the GC-based deadlock detection of Golf.

GoAT [28] constructs concurrency usage models from the syntax of Go programs, and injects handlers to emit traces of concurrent events and preempt execution at key points to explore more interleavings at runtime. The traces are analyzed offline for partial deadlocks. However, only deadlocks that exist at program termination are detected, unlike with Golf, which can detect deadlocks in long-running production systems.

The idea of piggybacking on the garbage collector has also been explored for other purposes, including analysis of method purity [20], dynamic software updating [27], and dynamically checking heap properties [21].

To the best of our knowledge, none of the existing techniques soundly approximate liveness by using memory reachability, and, subsequently, directly leverage the garbage collector to detect and reclaim partial deadlocks.

8 Conclusions and Future Work

We have demonstrated that the garbage collector in Go can be leveraged to dynamically detect partial deadlocks and reclaim the memory used by the involved goroutines, thereby effectively addressing the problem with goroutine leaks, while preserving program semantics. With the tool Golf, we have shown that the approach can be implemented as a modest extension to the existing Go garbage collector, and we believe it is sufficiently robust and efficient to be used in production environments.

The effectiveness of the approach has been evaluated in a series of experiments. Golf was able to detect a significant portion of partial deadlocks in microbenchmarks with known partial deadlocks, large scale industrial codebase test suites, and a real service used in production.

By construction, all reported partial deadlocks are true positives. False negatives are possible due the conservative approximation of memory reachability computed by the garbage collector. In future work, it may be interesting to incorporate static analysis techniques to provide liveness hints to the garbage collector in order to boost the deadlock detection capability. It would also be interesting to explore the generalizability of the partial deadlock detection approach to other programming languages and garbage collection mechanisms.

Acknowledgements

This work is supported in part by the National Science Foundation under grant numbers CCF-1910568 and CCF-1943456.

References

[1] [n. d.]. Go GC stats collection at runtime. https://pkg.go.dev/runtime.

[2] [n. d.]. Go source file for escape analysis. https://tip.golang.org/src/cmd/compile/internal/escape/escape.go. Accessed on October 2024.

[3] [n. d.]. Goleak - Goroutine Leak Detector. https://github.com/uber-go/goleak.

[4] C. R. Aragon and R. G. Seidel. 1989. Randomized search trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS '89)*. IEEE Computer Society, USA, 540–545. doi:10.1109/SFCS.1989.63531

[5] Go Authors. 2024. A Guide to the Go Garbage Collector. https://go.dev/doc/gc-guide Accessed: 2025-02-09.

[6] Edsger W. Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth F. M. Steffens. 1978. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM* 21, 11 (1978), 966–975. doi:10.1145/359642.359655

[7] Nicolas Dilley and Julien Lange. 2019. An Empirical Study of Messaging Passing Concurrency in Go Projects. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. IEEE, 377–387. doi:10.1109/SANER.2019.8668036

[8] Nicolas Dilley and Julien Lange. 2021. Automated Verification of Go Programs via Bounded Model Checking. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1016–1027. doi:10.1109/ASE51524.2021.9678571

[9] Zebao Gao. 2017. *Quantifying flakiness and Minimizing its effects on Software Testing*. Ph. D. Dissertation. University of Maryland, College Park, MD, USA. doi:10.13016/M2KH0F27R

[10] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677. doi:10.1145/359576.359585

[11] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295. doi:10.1109/32.588521

[12] Alexandre Kirszenberg, Antoine Martin, Hugo Moreau, and Etienne Renault. 2021. Go2Pins: A Framework for the LTL Verification of Go Programs. In *Model Checking Software - 27th International Symposium, SPIN 2021, Virtual Event, July 12, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12864)*. Springer, 140–156. doi:10.1007/978-3-030-84629-9_8

[13] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Fencing off go: liveness and safety for channel-based programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 748–761. doi:10.1145/3009837.3009847

[14] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June*

03, 2018. ACM, 1137–1148. doi:10.1145/3180155.3180157

[15] I-Ting Angelina Lee, Zhizhou Zhang, Abhishek Parwal, and Milind Chabbi. 2024. The Tale of Errors in Microservices. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 3, Article 46 (Dec. 2024), 36 pages. doi:10.1145/3700436

[16] Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. 2022. Who Goes First? Detecting Go Concurrency Bugs via Message Reordering. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 888–902. doi:10.1145/3503222.3507753

[17] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. 2021. Automatically detecting and fixing concurrency bugs in go software systems. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 616–629. doi:10.1145/3445814.3446756

[18] Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. 2018. Process-Local Static Analysis of Synchronous Processes. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11002)*. Springer, 284–305. doi:10.1007/978-3-319-99725-4_18

[19] Nicholas Ng and Nobuko Yoshida. 2016. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. ACM, 174–184. doi:10.1145/2892208.2892232

[20] Erik Österlund and Welf Löwe. 2012. Analysis of pure methods using garbage collection. In *Proceedings of the 2012 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '12, Beijing, China, June 16, 2012*. ACM, 48–57. doi:10.1145/2247684.2247694

[21] Christoph Reichenbach, Neil Immerman, Yannis Smaragdakis, Edward Aftandilian, and Samuel Z. Guyer. 2010. What can the GC compute efficiently?: a language for heap assertions at GC time. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. ACM, 256–269. doi:10.1145/1869459.1869482

[22] Georgian-Vlad Saioc, Dmitriy Shirchenko, and Milind Chabbi. 2024. Unveiling and Vanquishing Goroutine Leaks in Enterprise Microservices: A Dynamic Analysis Approach. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*. IEEE, 411–422. doi:10.1109/CGO57630.2024.10444835

[23] Georgian-Vlad Saioc and Milind Chabbi. 2022. LeakProf: Featherlight In-Production Goroutine Leak Detection. https://www.uber.com/en-GB/blog/leakprof-featherlight-in-production-goroutine-leak-detection/.

[24] Georgian-Vlad Saioc, I-Ting Angelina Lee, Anders Møller, and Milind Chabbi. 2025. *Artifact for "Dynamic Partial Deadlock Detection and Recovery via Garbage Collection"*. doi:10.5281/zenodo.14849367

[25] Sourav Choudhary. 2023. Exploring the Inner Workings of Garbage Collection in Golang: Tricolor Mark and Sweep. https://medium.com/@souravchoudhary0306/exploring-the-inner-workings-of-garbage-collection-in-golang-tricolor-mark-and-sweep-e10eae164a12.

[26] Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann. 2016. Static Trace-Based Deadlock Analysis for Synchronous Mini-Go. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*. 116–136. doi:10.1007/978-3-319-47958-3_7

[27] Suriya Subramanian, Michael W. Hicks, and Kathryn S. McKinley. 2009. Dynamic software updates: a VM-centric approach. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. ACM, 1–12. doi:10.1145/1542476.1542478

[28] Saeed Taheri and Ganesh Gopalakrishnan. 2021. GoAT: Automated Concurrency Analysis and Debugging Tool for Go. In *IEEE International Symposium on Workload Characterization, IISWC 2021, Storrs, CT, USA, November 7-9, 2021*. IEEE, 138–150. doi:10.1109/IISWC53511.2021.00023

[29] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 865–878. doi:10.1145/3297858.3304069

[30] D.M. Tullsen, S.J. Eggers, and H.M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings 22nd Annual International Symposium on Computer Architecture*. 392–403.

[31] Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. 2022. Detecting Blocking Errors in Go Programs using Localized Abstract Interpretation. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 32:1–32:12. doi:10.1145/3551349.3561154

[32] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. 2021. GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 187–199. doi:10.1109/CGO51591.2021.9370317

[33] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 655–672. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

[34] Benjamin G. Zorn. 1990. Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. ACM, 87–98. doi:10.1145/91556.91597

A Artifact Appendix

A.1 Abstract

This artifact provides the implementation of Golf, an extension to the Go garbage collector that allows it to dynamically detect partial deadlocks in Go programs. It contains the necessary scaffolding to generate a Docker container that runs the modified Go runtime with the Golf extension on a selection of microbenchmarks containing known partial deadlocks. It supports the reproduction of **RQ1 (*a*)** and **RQ2**, but *not* **RQ1 (*b*-*c*)**.

A.2 Description

A.2.1 How to access. Download from Zenodo. [24] The expected size, when also including the virtual Docker image and container, is ~15GB.

A.2.2 Hardware dependencies. Recommended specifications: 10 cores, 32 GB RAM, 20 GB disk space. x86-64 architectures are recommended; Apple Silicon may run the Docker benchmark with `colima` (with a 15× slowdown).

A.2.3 Software dependencies. The recommended OS is Linux. Docker must be installed, and the Docker daemon must run in the background. Recommended Docker version is 26.1.4.

For MacOS with Apple Silicon. The build script `run.sh` automatically makes a best effort attempt at setting up the container with `colima` if `-apple-silicon` is the first passed argument. `colima` may also be installed via `brew`, and started manually:

```
colima start -disk 20 -cpu 10 -arch amd64
```

A.2.4 Data sets. The data set (included with the artifact) is a collection of microbenchmarks derived from real-life programs that contain known partial deadlocks. They are extracted from GoBench, and another collection of known leaky patterns.

A.3 Basic test

Run the artifact with `./run.sh`.

A.4 Experimental workflow

The experiment creates a Docker container, where it installs two versions of the Go runtime: a baseline (Go 1.22), and with Golf extensions, in the `baseline` and `golf` subdirectories, respectively. It also provides a testing harness (in `./tester`) that applies the Go runtimes to microbenchmarks. The container setup (Section A.3) runs the experiments, then starts a session within the container.

A.4.1 Go runtimes. Each version of Go is compiled by running:

```
bash ./{baseline,golf}/src/make.bash
```

The resulting binaries are found at `bin/go` in each Go directory.

The following files in `./golf/src` have been updated:

- `runtime/mgc.go`: live goroutine discovery and deadlock detection.
- `runtime/mgcmark.go`: obfuscations of goroutine addresses and deadlocked goroutine shutdown.
- `runtime/sema.go`: supports `sync` deadlocks.
- `runtime/runtime2.go`: goroutine status extensions.
- `sync/{runtime,waitgroup}.go`: enable detection of deadlocks caused by `sync.WaitGroup`.

A.4.2 Testing harness. The testing harness, in `./tester`, can be built manually with `go build .` (Go ≥1.21 recommended). When successfully built, a binary is produced at `./tester/golf-tester`.

Example programs are found in `./tester/tests`. They are further split between *deadlocking* and *correct* examples, in the `deadlock` and `correct` subdirectories, respectively. Each example is a standalone `main.go` file within its own subdirectory.

The core microbenchmark suites (Section A.2.4) are found in the subdirectories `cgo-examples` and `goker`. A subset of the microbenchmarks also feature correct versions in `tests/correct`.

The results of the testing harness include execution traces in `./tester/results-<n>` directories, where `n` is the execution run index. Each directory mimics the subdirectory structure of `tests`.

Trace files names indicate the runtime configuration, e.g., if the name contains `GOMAXPROCS-4`, then the execution was configured with `GOMAXPROCS=4` (4 logical cores). Partial deadlocks are prefixed with `partial deadlock!`.

A.5 Evaluation and expected results

Section A.3 already sets up the container, and starts a session in `/usr/app/tester`. Due to non-determinism and flakiness, results vary between executions, or compared to the paper.

A.5.1 Microbenchmark coverage (RQ1 (*a*)). The microbenchmark coverage report is found at `./results`. It marks any unexpected deadlock reports with `Unexpected DL` and runtime exceptions with `[runtime failure]`. Occasionally, `etcd/7443` fails due to `send on closed channel`. This is an issue inherent to the microbenchmark, not caused by Golf.

Afterwards, `./results` contains an aggregated report like Table 1, answering **RQ1 (*a*)**. For example:

```
Benchmark 1P 2P 4P 10P Total
goker/etcd/7443:129 0 0 0 0 0.00%
goker/etcd/7443:216 0 0 0 0 0.00%
goker/etcd/7443:222 0 0 0 0 0.00%
goker/etcd/7443:226 0 0 0 0 0.00%
goker/etcd/7443:96 0 0 0 0 0.00%
Remaining 116 go instruction (72 benchmarks) 100.00%
Aggregated 95.86% 95.86% 95.86% 95.86% 95.86%
```

```
package main

// Add more libraries, if needed
import (
  "fmt"
  "runtime"
  "time"
)

func init() {
  fmt.Println("Starting run...")
}

// Custom functions go here

func main() {
  defer func() {
    time.Sleep(...) // Wait a while
    runtime.GC() // Force a GC cycle
  }()

  // Main program goes here
}
```

Figure 5. Template for a Go microbenchmark.

It is expected for the table to include entries from `goker`, but not `cgo-examples`. The entries must add up to 121 (number of rows plus the `x` in `Remaining x go instructions`). The total detection rate value (`Aggregated/Total`) is expected to be above 90%, with a median value of ∼94% across repeated experiments.

A.5.2 GOLF overhead (RQ2). The microbenchmark performance overhead is reported at `./results-perf.csv`. It contains the performance of each individual microbenchmark execution (average marking phase duration and CPU utilization), when using both the baseline and GOLF GCs. Baseline execution metrics are denoted with `OFF`, while GOLF execution metrics are denoted with `ON`. For example, `Mark clock ON (µs)` denotes the average wall clock duration of the GOLF marking phase.

The microbenchmark performance overhead is also supplied as a LaTeX box plot for the `Mark clock` columns, in the generated `./results.tex`. To export it, follow these steps:

1. Exit the Docker container with `exit`
2. List the Docker container IDs with `docker ps -a`
3. Get the id of the latest container where the `IMAGE` value is `golf`.
4. Copy the `.tex` file from the docker container to the source system, using the container ID.
   ```
   docker cp <ID>:/usr/app/tester/results.tex ./results.tex
   ```
5. Render `./results.tex` as a PDF with the TeX compiler of your choice.
6. Remove the container when done with `docker rm <ID>`

GOLF is typically outperformed by the baseline GC for examples without partial deadlocks, but without a signficant penalty. However, it may significantly outperform the baseline GC for examples with partial deadlocks. Values vary, depending on scheduling non-determinism, but at most within 1 millisecond.

A.6 Experiment customization

Write your program at `./tester/tests`, in its own subdirectory. Place deadlocking examples in `deadlock`, and deadlock-free examples in `correct`. Each example should have its own directory, with a `main.go` following template in Figure 5

To run GOLF manually, from the example directory, execute:

```
GODEBUG=gcdetectdeadlocks=1 <artifact>/golf/bin/go main.go
```

Any partial deadlock messages have the following format:

```
partial deadlock! goroutine ... Stack size ...
runtime.gopark(...)
.../golf/src/runtime/proc.go:402
<runtime stack...>
```

Annotate the example with `deadlocks: e` comments to validate it with the testing harness. Replace `e` with an integer constant, if the number of expected deadlocks is precisely known. Otherwise, use `x > 0` to signal at least one is expected.

The annotation is paired with a goroutine name at runtime, identified syntactically. Correct annotation placement depends on the signature of the goroutine function. For functions without formal parameters, place the annotation inside the function body:

```
// For anonymous functions  |  // For named functions
go func() {                 |  func foo() {
  // deadlocks: x > 0       |     // deadlocks: x > 0
  ...                       |  }
}()                         |  go foo() // Deadlocks
```

For functions with formal parameters, or methods, place the annotation above the go instruction:

```
// deadlocks: x > 0         |  // deadlocks: x > 0
go func(x int){ ... }(10)   |  go obj.deadlockingMethod()
// deadlocks: x > 0         |
go willDeadlock(x, y, z)    |
```

The key input flags for the testing harness are:

- `-golf <path>`: Path to GOLF Go binary.
- `-match <regex>`: The harness only runs on examples with paths that match `<regex>`.
- `-repeats <n>`: Repeat microbenchmark executions `n` times.
- `-report <path>`: Outputs coverage/perf report to `path`.
- `-baseline <path>`: Baseline Go binary (only for performance).
- `-perf`: Testing harness switches to performance measurements.

Run with `-match` set to the path of your examples to run them exclusively.

A.7 Notes

For more details of each step, consult `README.md` in the artifact repository.