[Who goes first? detecting go concurrency bugs via message reordering]

背景

现有的并发 bug 检测技术不能高效的检测 go 语言中的并发 bug，尤其是使用消息传递引起的 bug，因为现在大多数并发 bug 检测技术是为传统编程语言比如 c/c++或 java 这样使用共享内存的语言来设计的。所以需要构建一种 go 语言的检测器，用来检测 go 中因为消息传递引起的并发 bug。构建 go 的并发 bug 检测器的困难有两点：

1. 使用静态检测器来构建，那么静态检测器必须要覆盖所有的错误代码模式，否则检测器就会产生大量误报；

2. 使用动态检测器来构建，那么动态检测器只会报告当前发现的 bug，很多潜在的 bug 是无法检测到的。

```
1   // go parent()
2   func parent() {   // parent goroutine
3       ... // initialize object daemon
4       ch, errCh := daemon.discoveryWatcher.Watch()
5       select {
6       case <- Fire(1 * time.Second):
7           Log("Timeout!")
8       case e := <-ch:
9           if !reflect.DeepEqual(e, expected) {
10              Log("Unexpected!")
11          }
12      case e := <-errCh:
13          Log("Error!")
14      }
15      return
16  }
17  func (s *Discovery) Watch() (chan discovery.Entries, chan error) {
18  -   ch := make(chan discovery.Entries)
19  -   errCh := make(chan error)
20  +   ch := make(chan discovery.Entries, 1)
21  +   errCh := make(chan error, 1)
22      go func() { // child goroutine
23          entries, err := s.fetch()
24          if err != nil {
25              errCh <- err
26          } else {
27              ch <- entries
28          } ...
29      }()
30      return ch, errCh
31  }
```

这是一个使用同步通道引起的 bug，如果父 goroutine 执行了程序第 6 行，那么由第 22 行创建的子 goroutine 将会被永远的阻塞。检测到这个 bug 需要两个条件同时成立：

1. 程序第 6 行首先被执行；

2. 检测器能够分析出没有其他 goroutine 能够接受子 goroutine 中通道的数据。

静态分析的难点：静态分析无法有效推断间接调用的目标，比如第 4 行调用的目标。

动态分析的难点：在离线测试中，第 6 行总是不先被执行，那么第一个条件就不满足。

GFUZZ 的设计

以往的动态检测器不干预程序语句的执行顺序，但是这样很低效，因此 GFUZZ 故意改变并发消息的执行顺序来增加触发并发 bug 的概率。虽然改变并发消息的执行顺序听起来很简答，但是实现 GFUZZ 有三个难点：

1.如何识别并发消息?

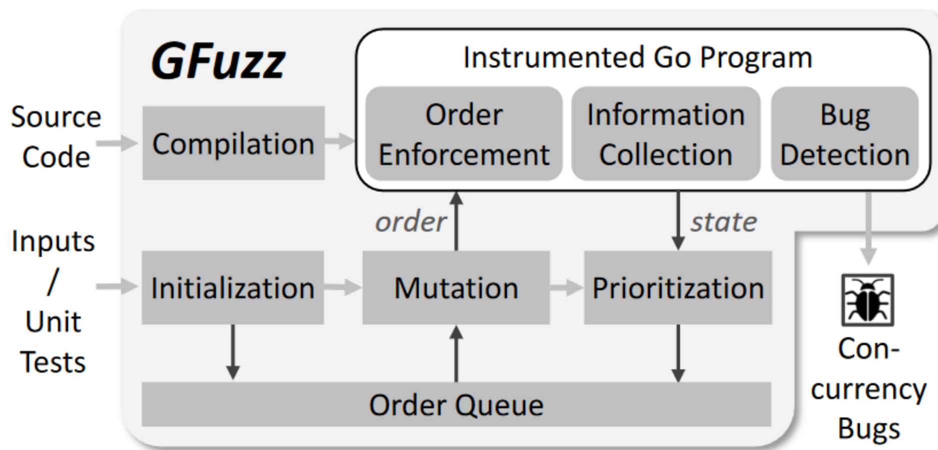
作者采取了一种简单的方法，考虑同一个 select 中的 case，认为这些 case 的消息是并发的，改变这些 case 的执行顺序并不会修改源程序的语义。

2.如何识别可疑的消息序列以及如何确定测试的优先级?

作者使用了模糊测试方法，从已经执行的消息序列来生成新的未执行的消息序列，并且设计了一种反馈机制来优先考虑更可能触发 bug 的消息序列。

3.如何识别与通道相关的 bug?

作者设计了一个运行时清理程序，可以跟踪程序中所有的通道引用的传递，并且设计了一种算法来识别一个 goroutine 是否会被其他 goroutine 解除阻塞的算法。



GFUZZ 的输入为 go 程序和一些程序的输入或者单元测试文件，输出为检测到的 bug。GFUZZ 会强制重新排序 go 程序中的并发消息，并通过反馈机制来优先运行更容易出错的并发消息顺序，最后通过内置的检测器来检测是否有 goroutine 被阻塞，从而输出检测结果。

消息重排

传统的动态检测器是运行多次程序来尽可能多的覆盖所有的并发消息执行顺序，但是由于一些原因，比如在本地测试的时候，有些消息的条件几乎不可能触发，所以这种盲目的运行程序会导致检测器的执行效率很低，GFUZZ 通过对消息顺序重排，每种并发消息顺序最多只会执行一次。但是在 go 程序中确定所有的消息之间有没有并发关系是很困难的，因此 GFUZZ 采用了一种简单直接的方法：每一个 select 语句中所有的 case 都是视为并发的消息。

GFUZZ 会识别 go 程序中所有的 select 语句，对其中的每个 case 都标号，并设置了一个三元组（si，ci，ei）来标识执行第 si 个 select 语句中的第 ei 个 case，ci 表示这个 select 语句中的 case 数。GFUZZ 使用 switch 语句来替换 select 语句，具体的规则如下图所示：

```
1 switch FetchOrder(…) {
2   case 0:
3     select {
4     case <- Fire(1 * time.Second):
5       Log("Timeout!")
6     case <- time.After(T):
7-16    ......
17    }
18  case 1:
19    select {
20    case e := <- ch:
21-23   ......
24    case <- time.After(T):
25-34   ......
35    }
36  case 2:
37    select {
38    case e := <- errCh:
39      Log("Error!")
40    case <- time.After(T):
41-50   ......
51    }
52  default:
53-62   ......
63 }
```

```
5)  select {
6)  case <- Fire(1 * time.Second):
7)    Log("Timeout!")
8)  case e := <- ch:
9-11) …
12) case e := <- errCh:
13)   Log("Error!")
14) }
```

右侧代码是未修改的 go 程序，左侧代码是经过 GFUZZ 修改后的 go 程序，假设右侧的 select 是 go 源程序中第一个 select 语句，这个 select 语句包含 3 个 case，如果执行第 6 行的 case，那么三元组表示为（0，3，0），同理如果执行第 8 行的 case，三元组表示为（0，3，1）。如果优先执行第 6 行的 case，那么只需要用左侧代码中第 2-17 行的结构来替换原来的 go 代码即可。通过这样的修改，我们就可以把并发消息的执行顺序转换为三元组的组合问题，并根据每个三元组来强制执行对应的消息。

优先级顺序反馈

由于程序中可能的消息数量很大，所以每个消息顺序组合全部运行一次的效率很低，GFUZZ 通过收集与通道相关的运行时信息来评价这个消息顺序的质量并给出一个分数，并且会根据评分的高低来考虑是否基于这个顺序来进行多次修改，如果评分很低，说明这个顺序是比较安全的，修改执行顺序产生并发 bug 的可能性不大；如果评分高，说明这个顺序很可能产生 bug，同时修改执行顺序也很可能会触发 bug。评分的依据是 GFUZZ 收集的运行时相关信息：通道的状态和通道操作的交错。

Table 1: Runtime Information as Feedback. *The interesting criteria determine whether we keep the current order for future mutations.*

Information	Semantics	Interesting Criteria	Identifier
CountChOpPair	# execs of each pair of channel operations	new pair / counter heavily changes	$(ID_{prev_op} \gg 1) \oplus ID_{cur_op}$
CreateCh	distinct channels created	new distinct channel created	ID of channel-create instruction
CloseCh	distinct channels closed	new distinct channel closed	ID of channel-create instruction
NotCloseCh	distinct channels remaining open	new distinct channel not closed	ID of channel-create instruction
MaxChBufFull	maximum fullness of each buffered channel	new maximum fullness	ID of channel-create instruction

GFUZZ 的监视粒度是每个单独的通道操作，表中第一行就是监视通道操作的交错，其他四行是创建或关闭通道，未关闭的通道个数以及通道满的个数的通道状态，评价的计算公式为

$$score = \sum \log_2 CountChOpPair + 10 * \#CreateCh + 10 * \#CloseCh + 10 * \sum MaxChBufFull$$

GFUZZ 会根据分数高的消息顺序来基于这个顺序做出更多的顺序调整，因为分数高的消息顺序更可能触发并发 bug。

运行时清理程序

作者认为 go 的运行时会捕获通道相关的非阻塞性错误，因此 GFUZZ 专注于与通道相关的阻塞性错误。GFUZZ 会主动跟踪并维护通道引用在不同 goroutine 间的传递关系，维护了三个变量

1. mapChToChan：每个 go 程序中的通道映射为 go 运行时的内部表示；
2. stGoInfo：维护有关 goroutine 的信息；
3. stPInfo：跟踪有关同步原语的信息。

并根据下面的阻塞算法来检测被阻塞的 goroutine

Algorithm 1 Blocking Bug Detection

```
Require: goroutine (g) and channel (c) /* g blocks on c */
 1: function (g, c)
 2:     VisitedPrim_set, VisitedGo_set ← {c}, {}
 3:     Go_list ← stPInfo_map[c].getGos()
 4:     while Go_list is not empty do
 5:         go ← Go_list.pop_front()
 6:         if not stGoInfo_map[go].blocking then
 7:             return False, {}
 8:         end if
 9:         VisitedGo_set.insert(go)
10:         for each primitive (p) in stGoInfo_map[go].getPrims() do
11:             if p not in VisitedPrim_set then
12:                 VisitedPrim_set.insert(p)
13:                 for each goroutine (g') in stPInfo_map[p].getGos() do
14:                     Go_list.append(g')
15:                 end for
16:             end if
17:         end for
18:     end while
19:     return True, VisitedGo_set
20: end function
```

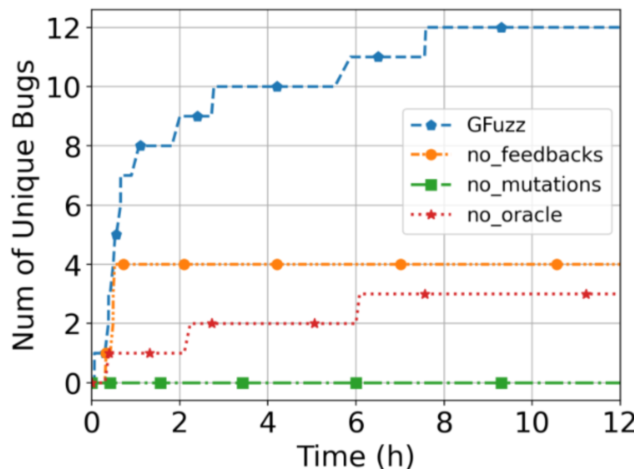
检测的原则是如果当前 goroutine 因为某个通道操作被阻塞，检查是否有其他 goroutine 持有该通道的引用，如果发现检查结果为空，说明不存在任何 goroutine 能够在程序后续执行中唤醒被阻塞的 goroutine。根据这个原则，我们就可以检测到由于通道操作引起的并发 bug。

评价

Table 2: Benchmarks and Evaluation Results. *LoC means lines of source code; and test denotes numbers of unit tests used in our experiments. In the "Detected New Bugs" columns, subscript b denotes blocking bugs; NBK represents non-blocking bugs; "-" means zero bugs; and $GFuzz_3$ represents bugs detected in the first three fuzzing hours. The "$Overhead_s$" column shows the overhead of the sanitizer and "/" means not applicable.*

	Benchmark Info.			Detected New Bugs							
App	**Star**	**LoC**	**Test**	**$chan_b$**	**$select_b$**	**$range_b$**	**NBK**	**Total**	**$GFuzz_3$**	**GCatch**	**$Overhead_s$**
Kubernetes	74K	3453K	3176	28	4	9	2	43	18	3	36.75%
Docker	60K	1105K	1227	17	2	-	-	19	5	4	44.53%
Prometheus	35K	1186K	570	14	-	1	3	18	8	-	18.08%
etcd	35K	181K	452	7	12	-	1	20	7	5	14.43%
Go-Ethereum	28K	368K	1622	11	43	6	2	62	40	5	75.18%
TiDB	27K	476K	264	-	-	-	-	-	-	-	17.65%
gRPC	13K	117K	888	15	-	1	6	22	7	8	20.00%
Total	272K	6887K	8199	92	61	17	14	184	85	25	/

1. 有效性：GFUZZ 检测到了多少新的错误
   所有的检测结果都在 table2 中，GFUZZ 检测到了 184 个以前未知的 bug，其中包括 170 个阻塞 bug 和 14 个非阻塞 bug，但是有 12 个误报。
2. 先进性：GFUZZ 是否比最先进的 gcatch 检测到更多错误
   GFUZZ 在前 3 个小时内发现了 85 个 bug，gcatch 在前 3 小时只检测到 5 个，总共发现了 25 个 bug，但是两方都发现的 bug 只有 5 个，由于 gcatch 保证精度的设计方式使得很多过程间分析的 bug 都会被 gcatch 放弃分析，并且 gcatch 不检测非阻塞型 bug，也缺少一些动态信息，比如通道的大小或指针指向哪个通道，所以 GFUZZ 检测到的 bug 比 gcatch 要多。但是有一些 bug 也会被 GFUZZ 遗漏，比如只有当特定返回值时出现的 bug，消息重排并不能改变返回值。
3. 必要性：GFUZZ 的每个组件是否有助于错误检测



   作者在 grpc 仓库上测试 GFUZZ 前 12 个小时每种设置检测到的 bug 数量。完全的 GFUZZ 可以检测到 12 个 bug；少了任何一部分都会使得 GFUZZ 的检测效率明显下降。
4. 性能：GFUZZ 的运行时开销是多少
   GFUZZ 每秒运行 0.62 的单元测试，会导致 3x 的开销，开销的原因主要是消息重排中为了防止死锁，会引入一个等待时间，和收集运行时信息导致的开销。

限制

1.GFUZZ 并发检测的规模小，只考虑了同一个 select 语句中所有的 case，有更多的并发消息没有被识别。

2.反馈机制类似模糊测试，基于贪心的思想，但不一定是全局最优解。

贡献

本文提出了一种新的检测工具 GFUZZ，通过主动改变并发消息的处理顺序来触发并发错误，检测了与通道相关的并发 bug，并且在开源社区中 7 个流行的 go 仓库中贡献了 184 个以前未知的 bug，并且 GFUZZ 的性能优于最先进的 go 并发错误检测器。