Understanding Real-World Concurrency Bugs in Go

Tengfei Tu\textsuperscript{1}, Xiaoyu Liu\textsuperscript{2}, \textit{Linhai Song}\textsuperscript{1}, and Yiying Zhang\textsuperscript{2}

\textsuperscript{1}Pennsylvania State University
\textsuperscript{2}Purdue University
• A young but widely-used programming lang.

• Designed for efficient and *reliable* concurrency
  – Provide lightweight threads, called goroutines
  – Support both message passing and shared memory
Massage Passing vs. Shared Memory

Message Passing

Concurrency Bug

Thread 1

Thread 2

Thread 1

Thread 2

Memory

Shared Memory
Does Go Do Better?

- Message passing better than shared memory?
- How well does Go prevent concurrency bugs?
The 1st Empirical Study

• Collect 171 Go concurrency bugs from 6 apps
  – through manually inspecting GitHub commit log
• How we conduct the study?
  – Taxonomy based on two orthogonal dimensions
    • Root causes and fixing strategies
  – Evaluate two built-in concurrency bug detectors

171 Real-World Go Concurrency Bugs
Highlighted Results

- Message passing can make a lot of bugs
  - sometimes even more than shared memory
- 9 observations for developers’ references
- 8 insights to guide future research in Go
Outline

• Introduction
• A real bug example
• Go concurrency bug study
  – Taxonomy
  – Blocking Bug
  – Non-blocking Bug
• Conclusions
Outline

• Introduction
• A real bug example
• Go concurrency bug study
  – Taxonomy
  – Blocking Bug
  – Non-blocking Bug
• Conclusions
Message Passing in Go

- How to pass messages across goroutines?
  - Channel: unbuffered channel vs. buffered channel
  - Select: waiting for multiple channel operations
An Example of Go Concurrency Bug

Parent Goroutine

```go
define finishRequest(t sec) r object {
  ch := make(chan object)
  go func() {
    result := fn()
    ch <- result
  }()
  select {
    case result = <- ch:
      return result
    case <- time.timeout(t):
      return nil
  }
} //Kubernetes#5316
```
An Example of Go Concurrency Bug

Parent Goroutine

```go
func finishRequest(t sec) *object {
    ch := make(chan *object)
    go func() {
        result := fn()
        ch <- result
    }()
    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```
An Example of Go Concurrency Bug

```
func finishRequest(t sec) r object {
  ch := make(chan object)
  go func() {
    result := fn()
    ch <- result
  }()
  select {
    case result = <- ch:
      return result
    case <- time.timeout(t):
      return nil
  }
} //Kubernetes#5316
```
An Example of Go Concurrency Bug

Parent Goroutine

```go
func finishRequest(t sec) r object {
    ch := make(chan object)
    go func() {
        result := fn()
        ch <- result
    }()
    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```

Child Goroutine

```go
func() {
    result := fn()
    ch <- result
}()
```

- Blocking and goroutine leak
- Timeout signal
An Example of Go Concurrency Bug

Parent Goroutine

```go
func finishRequest(t sec) r object {
  ch := make(chan object, 1)
  go func()
    result := fn()
    ch <- result
}()
select {
  case result = <- ch:
    return result
  case <- time.timeout(t):
    return nil
}
} //Kubernetes#5316
```

Child Goroutine

```go
func() 
  result := fn()
  ch <- result 
}()
```

not blocking any more
New Concurrency Features in Go

```go
func finishRequest(t sec) r object {
    ch := make(chan object)
    go func() {
        result := fn()
        ch <- result
    }()

    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```
New Concurrency Features in Go

```go
func finishRequest(t sec) r object {
    ch := make(chan object)
    go func() {
        result := fn()
        ch <- result
    }()
    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```

- **Anonymous function**
- **Buffered channel vs. unbuffered channel**
- **Use `select` to wait for multiple channels**
• Introduction
• A real bug example
• Go concurrency bug study
  – Taxonomy
  – Blocking Bug
  – Non-blocking Bug
• Conclusions
Bug Taxonomy

- Categorize bugs based on two dimensions
  - Root cause: shared memory vs. message passing

```go
func finishRequest(t sec) r object {
    ch := make(chan object)
    go func()
        result := fn()
        ch <- result
    }()
    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```
Bug Taxonomy

• Categorize bugs based on two dimensions
  – Root cause: shared memory vs. message passing
  – Behavior: blocking vs. non-blocking

```go
func finishRequest(t sec) r object {
  ch := make(chan object)
  go func()
    result := fn()
    ch <- result
  }()
  select {
    case result = <- ch:
      return result
    case <- time.timeout(t):
      return nil
  }
} //Kubernetes#5316
```
• Categorize bugs based on two dimensions
  – Root cause: shared memory vs. message passing
  – Behavior: blocking vs. non-blocking

<table>
<thead>
<tr>
<th>Cause</th>
<th>Behavior</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>blocking</td>
<td>non-</td>
</tr>
<tr>
<td>shared</td>
<td>36</td>
<td>69</td>
</tr>
<tr>
<td>message passing</td>
<td>49</td>
<td>17</td>
</tr>
</tbody>
</table>

85 86

105 66
Observation: Share memory synchronizations are used more often in Go applications.
• Introduction
• A real bug example
• Go concurrency bug study
  – Taxonomy
  – Blocking Bug
  – Non-blocking Bug
• Conclusions
• Conducting blocking operations
  – to protect shared memory accesses
  – to pass message across goroutines
(mis)Protecting Shared Memory

**Observation:** Most blocking bugs caused by shared memory synchronizations have the same causes as traditional languages.
Misuse of Channel

Goroutine 1

\[\text{ch} \leftarrow \text{m}\]

blocking

Goroutine 2

\[\text{m} \leftarrow \text{ch}\]

<table>
<thead>
<tr>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex</td>
<td>Wait</td>
<td>RW Mutex</td>
<td>Chan</td>
<td>Chan w/ Lib</td>
<td>Lib</td>
</tr>
</tbody>
</table>

- BoltDB
- gRPC
- CockroachDB
- etcd
- Kubernetes
- Docker
Misuse of Channel with Lock

```go
func goroutine1() {
  m.Lock()
  ch <- request
  m.Unlock()
}

func goroutine2() {
  for {
    m.Lock()
    m.Unlock()
    request <- ch
  }
}
```
**Observation**: more blocking bugs in our studied Go applications are caused by wrong message passing.

20% More
**Implication**: we call for attention to the potential danger in programming with message passing.

![Chart showing mutex, wait, RW Mutex, channel, channel with permission, library usage]
Outline

• Introduction
• A real bug example
• Go concurrency bug study
  – Taxonomy
  – Blocking Bug
  – Non-blocking Bug
• Conclusions
Root Causes

• Failing to protect shared memory
• Errors during message passing
Traditional Bugs

> 50%

![Bar chart showing traditional bugs with > 50%]

- BoltDB
- gRPC
- CockroachDB
- etcd
- Kubernetes
- Docker
- misc
- chan
- lib
- waitgroup
- anon.
- traditional

Y-axis: Number of bugs
X-axis: Categories

31
Misusing Channel

Thread 1

\[ \text{close(ch)} \]

Thread 2

\[ \text{ch <- m} \]

\[ \text{panic!} \]

Thread 1

\[ \text{close(ch)} \]

Thread 2

\[ \text{close(ch)} \]

\[ \text{panic!} \]

---

### Bar Chart

<table>
<thead>
<tr>
<th>Category</th>
<th>BoltDB</th>
<th>gRPC</th>
<th>CockroachDB</th>
<th>etcd</th>
<th>Kubernetes</th>
<th>Docker</th>
</tr>
</thead>
<tbody>
<tr>
<td>traditional</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>anon.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>waitgroup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lib</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>misc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

32
**Implication:** new concurrency mechanisms Go introduced can themselves be the reasons of more concurrency bugs.
Conclusions

• 1st empirical study on go concurrency bugs
  – shared memory vs. message passing
  – blocking bugs vs. non-blocking bugs
  
  *paper contains more details (contact us for more)*

• Future works
  – Statically detecting go concurrency bugs
    • checkers built based on identified buggy patterns
    • Already found concurrency bugs in real applications
Thanks a lot!
Questions?

171 Real-World Go Concurrency Bugs

Data Set: https://github.com/system-pclub/go-concurrency-bugs