Profiling Request Latency with Critical Path Analysis

Felix Geisendörfer

DATADOG

Do you like fast software?


Fast
Productive
Go
Observable

Go makes Continuous Profiling Easy

- net/http/pprof exposes runtime/pprof and runtime/trace data with < 1-2% overhead

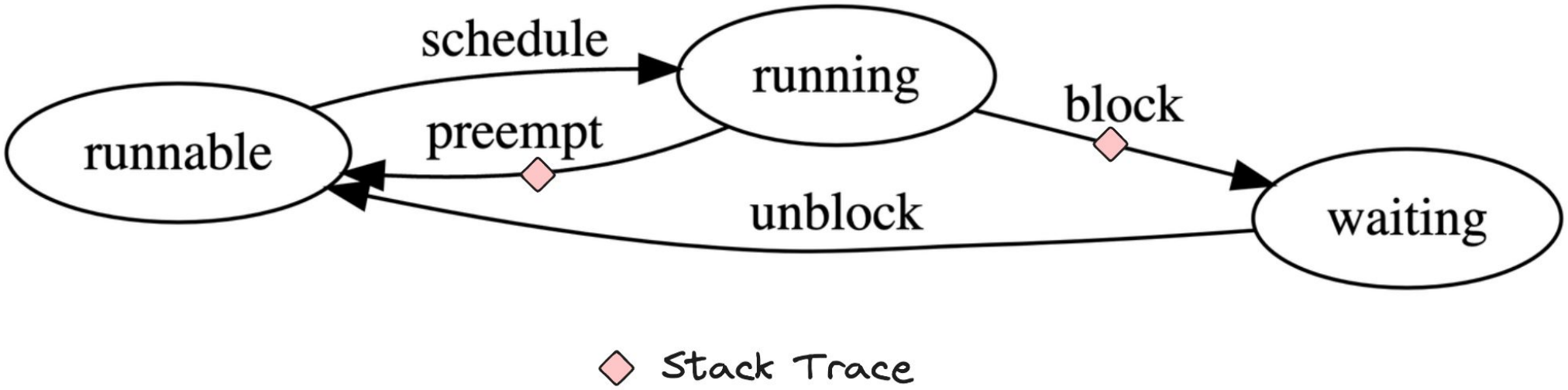

pprof
VIEW
SAMPLE
REFINE
CONFIG
DOWNLOAD
Search regexp
439.2s (100%)
root
errgroup.(*Group).add.func1
converter.(*RowConverter).parseRows-fm
converter.(*RowConverter).parseRows
log.(*traceLogger).Errorf
log.(*zapLogger).Error
zap.(*Logger).Error
zapcore.(*CheckedEntry).Write
zapcore.(*ioCore).Write
zapcore.(*lockedWriteSyncer).Write
sync.(*Mutex).Unlock

For endpoint X, how much is p99 latency caused by:

- CPU Usage
- Mutex Contention
- Channel Contention
- Scheduling Latency
- Garbage Collection (GC Assists)
- Syscalls
- Sleep
- Networking (*prefer distributed tracing for this*)

i.e. a complete breakdown of latency with nothing missing

runtime/trace has it all …



Text Analysis

```
$ go tool trace -d=parsed go.trace
…
M=1 P=1 G=2 StateTransition Time=100 GoID=2 Running->Waiting Reason="chan receive"
TransitionStack=
        runtime.chanrecv1 @ 0x1044f4873
                /code/github.com/golang/go/src/runtime/chan.go:509
        testing.(*T).Run @ 0x1045f0ee7
                /code/github.com/golang/go/src/testing/testing.go:2005
…
M=1 P=1 G=1 StateTransition Time=300 GoID=2 Waiting->Runnable Reason=""
Stack=
        runtime.chansend1 @ 0x1044f3a17
                /code/github.com/golang/go/src/runtime/chan.go:161
        testing.tRunner.func1.1 @ 0x1045f0b3f
                /code/github.com/golang/go/src/testing/testing.go:1849
M=1 P=0 G=-1 StateTransition Time=310 GoID=2 Runnable->Running Reason=""
```

UI Analysis

3 items selected. Counter Samples (3)

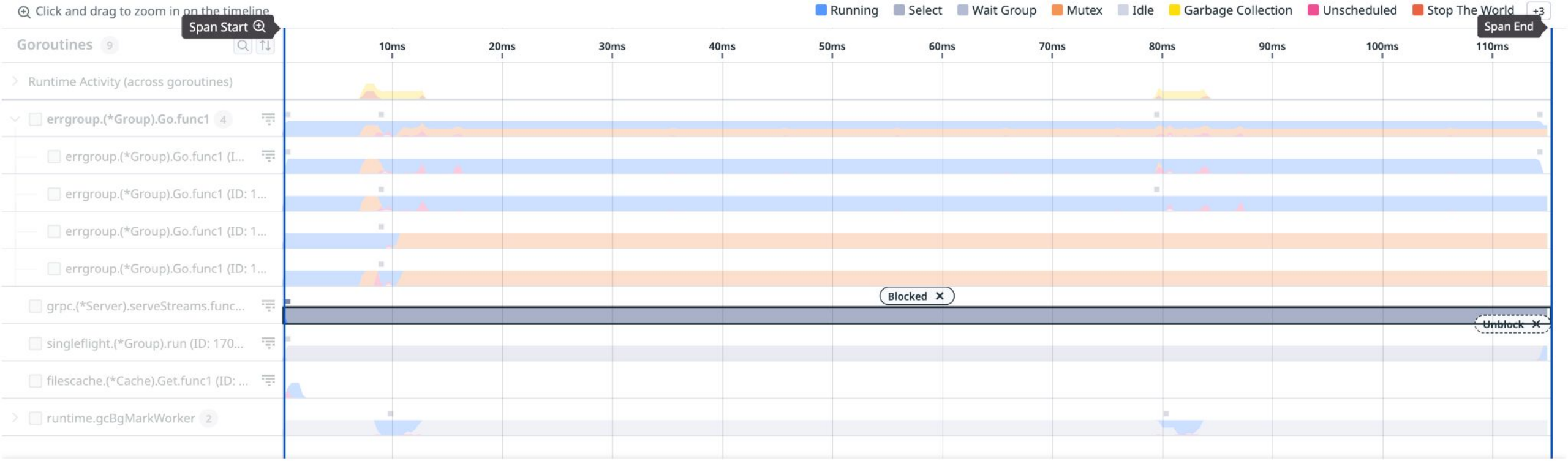
Counter	Series	Time	Value
Goroutines	GCWaiting	301.69964799999997	0
Goroutines	Runnable	301.69964799999997	175
Goroutines	Running	301.69964799999997	12

Request Annotation

```
import (
    "net/http"
    "runtime/trace"
)

func TaskMiddleware(next http.HandlerFunc) http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        ctx, task := trace.NewTask(r.Context(), "http.request")
        defer task.End()
        trace.Log(ctx, "http.request.url", r.URL.String())
        r = r.WithContext(ctx)
        next.ServeHTTP(w, r)
    })
}
```

Datadog Timeline (Goroutine Oriented Trace Viewer)



Details

Event Type — Select

Duration — 115ms

The goroutine was waiting on a channel select operation.

Stack Trace

```
runtime.selectgo (select.go:L#351)
Show all 37 lines
google.golang.org/grpc.(*Server).handleStream (server.go:L#1815)
google.golang.org/grpc.(*Server).serveStreams.func2.1 (server.go:L#1035)
```

Insight: **Critical path** *This activity was on the critical path for the span to finish*

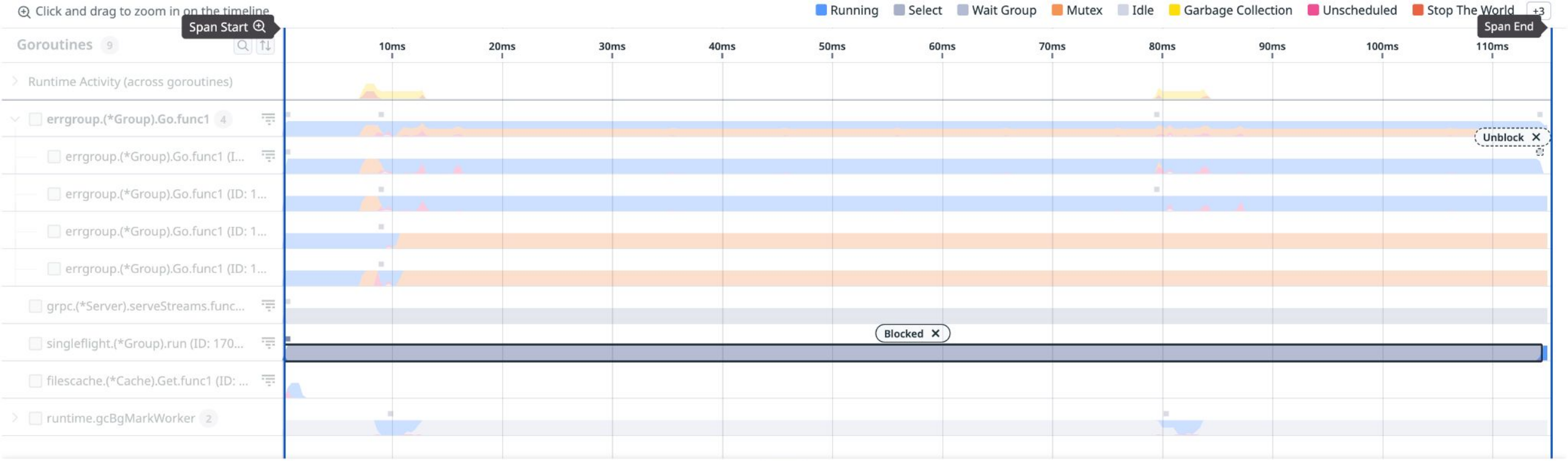
View Details →

Active spans: **singleflight.Do**

Relation with other events

Goroutine grpc.(*Server).serveStreams.func2.1 (ID: 17098411) was Blocked for

DATADOG



Details

Event Type — Wait Group

Duration — 114ms

The goroutine was waiting on a `sync.WaitGroup` call.

Stack Trace

```
sync.(*WaitGroup).Wait (waitgroup.go:L#118)
golang.org/x/sync/errgroup.(*Group).Wait (errgroup.go:L#56)
```

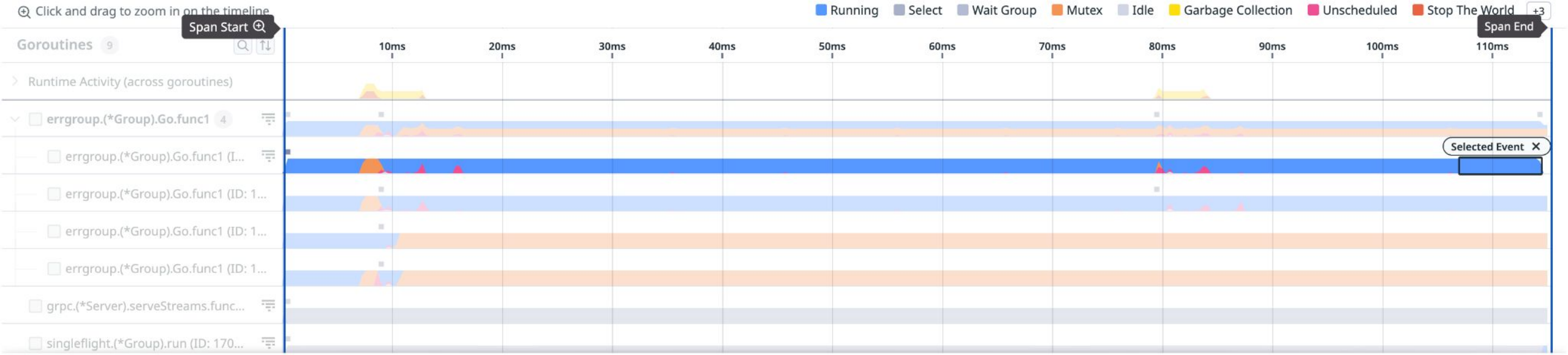
Show all 8 lines

Insight: **Critical path** *This activity was on the critical path for the span to finish*

View Details

Active spans: **singleflight.run**

Relation with other events



Details

Event Type	Running
Duration	7ms
Cpu Sample	no

The goroutine was running on an OS thread. The stack trace was captured at the end of the event.

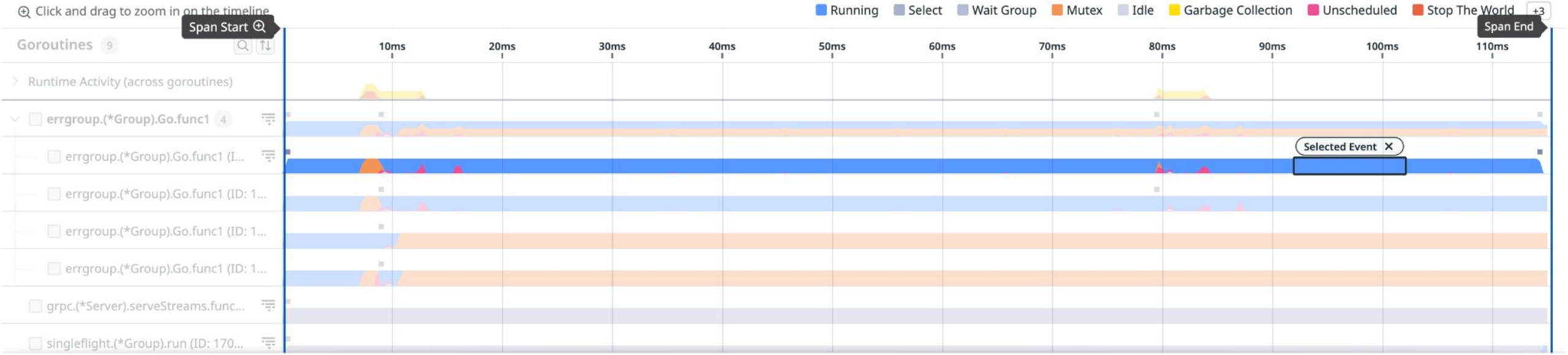
Stack Trace — No Stack Trace available

Insight: **Critical path** *This activity was on the critical path for the span to finish*

View Details →

Active spans: **singleflight.run**

singleflight.Do



Details

Event Type	Running
Duration	10ms
Cpu Sample	yes

The goroutine was running on an OS thread. The stack trace was captured at the end of the event.

Stack Trace

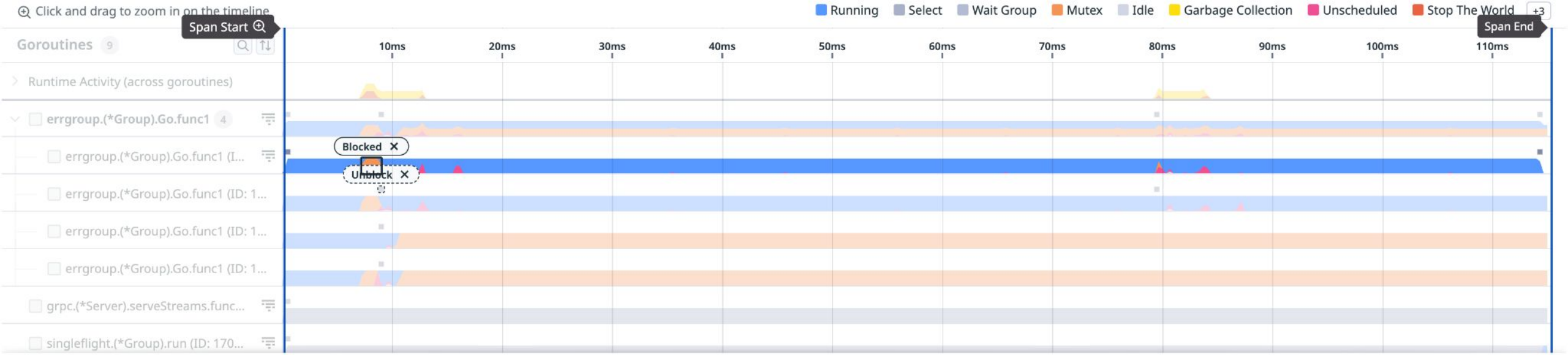
```
runtime.unlock2 (lock_spinbit.go:L#262)
runtime.unlockWithRank (lockrank_off.go:L#35)
runtime.unlock (lock_spinbit.go:L#253)
Show all 21 lines
golang.org/x/sync/errgroup.(*Group).Go.func1 (errgroup.go:L#93)
```

Insight: **Critical path** *This activity was on the critical path for the span to finish*

View Details →

Active spans: **singleflight.run**

singleflight.Do



Details

Event Type — Mutex

Duration — 2ms

The goroutine was blocked while trying to acquire a lock that was held by another goroutine.

Stack Trace

```
runtime.gcStart (mgc.go:L#668)
runtime.mallocgcSmallScanNoHeader (malloc.go:L#1425)
runtime.mallocgc (malloc.go:L#1058)
Show all 19 lines
golang.org/x/sync/errgroup.(*Group).Go.func1 (errgroup.go:L#93)
```

Insight: **Critical path** *This activity was on the critical path for the span to finish*

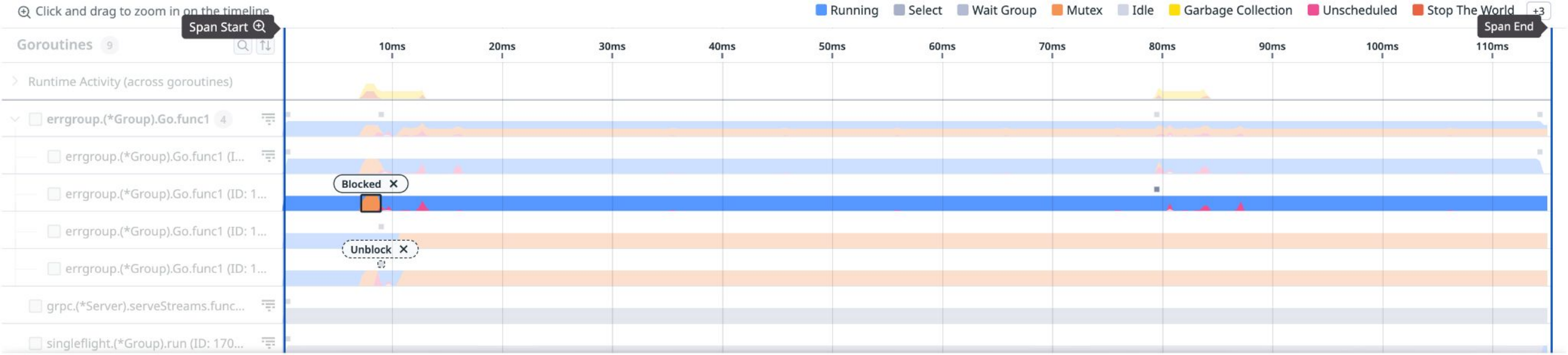
View Details →

Active spans: **singleflight.run**

singleflight.Do

Relation with other events

Goroutine `errgroup.(*Group).Go.func1 (ID: 17098413)` was **Blocked** for **1.7ms** and was then **Unblocked** by goroutine `errgroup.(*Group).Go.func1 (ID: 17099058)`



Details

Event Type: Mutex

Duration: 2ms

The goroutine was blocked while trying to acquire a lock that was held by another goroutine.

Stack Trace

```
runtime.gcStart (mgc.go:L#668)
runtime.mallocgcSmallScanNoHeader (malloc.go:L#1425)
runtime.mallocgc (malloc.go:L#1058)
Show all 19 lines
golang.org/x/sync/errgroup.(*Group).Go.func1 (errgroup.go:L#93)
```

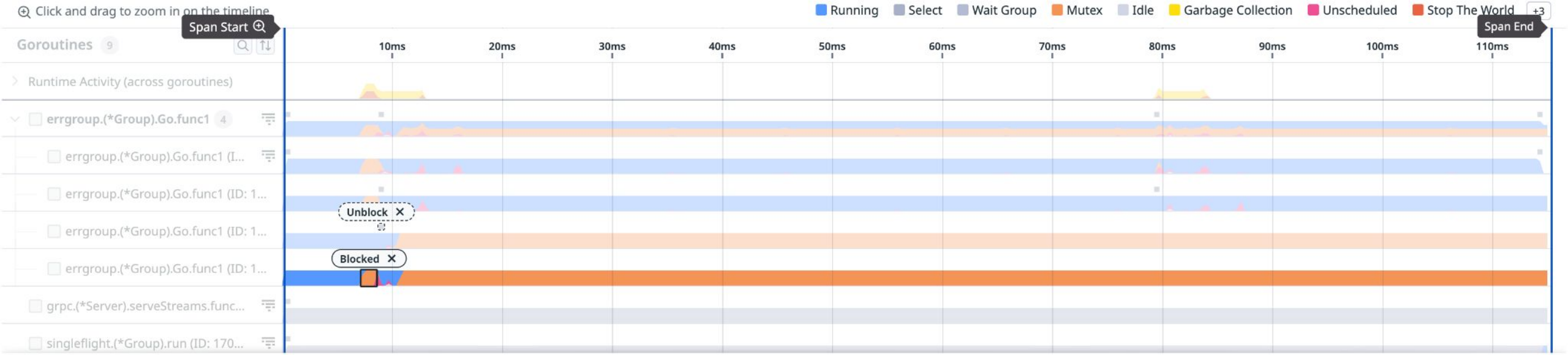
Insight: **Critical path** *This activity was on the critical path for the span to finish*

View Details →

Relation with other events

Goroutine `errgroup.(*Group).Go.func1 (ID: 17099058)` was Blocked ✕ for **1.61ms** and was then Unblocked ✕ by goroutine `errgroup.(*Group).Go.func1 (ID: 17099049)`

DATADOG



Details

Event Type: Mutex

Duration: 1ms

The goroutine was blocked while trying to acquire a lock that was held by another goroutine.

Stack Trace

```
runtime.gcStart (mgc.go:L#668)
runtime.mallocgcLarge (malloc.go:L#1591)
runtime.mallocgc (malloc.go:L#1063)
Show all 10 lines
golang.org/x/sync/errgroup.(*Group).Go.func1 (errgroup.go:L#93)
```

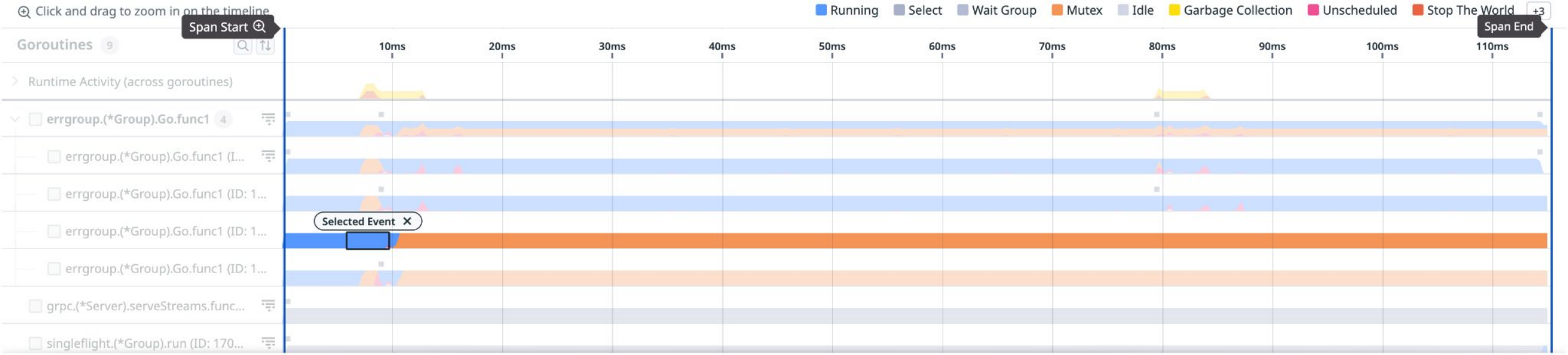
Insight: **Critical path** *This activity was on the critical path for the span to finish*

View Details →

Relation with other events

Goroutine `errgroup.(*Group).Go.func1 (ID: 17099049)` was **Blocked** ✕ for **1.31ms** and was then **Unblocked** ✕ by goroutine `errgroup.(*Group).Go.func1 (ID: 17099047)`

DATADOG



Details

Event Type	Running
Duration	4ms
Cpu Sample	no

The goroutine was running on an OS thread. The stack trace was captured at the end of the event.

Stack Trace — No Stack Trace available

Insight: **Critical path** *This activity was on the critical path for the span to finish*

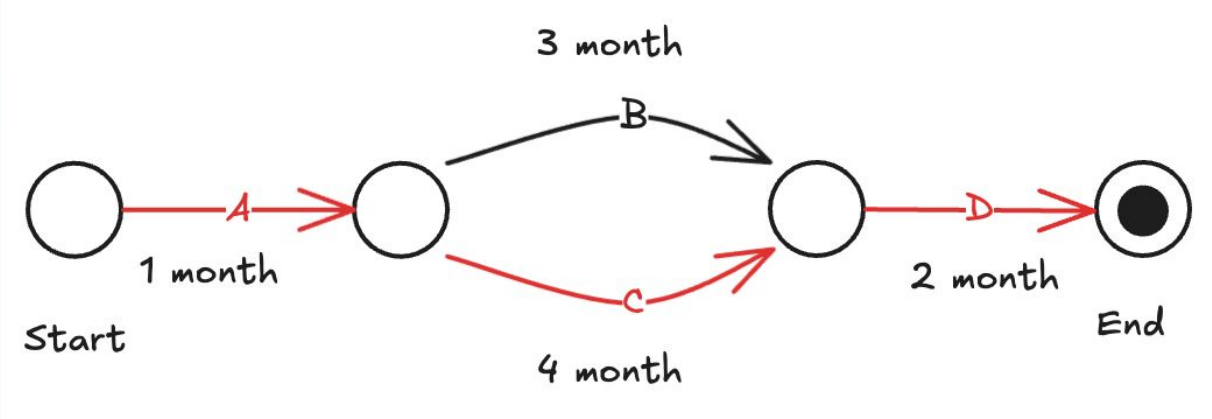
View Details →

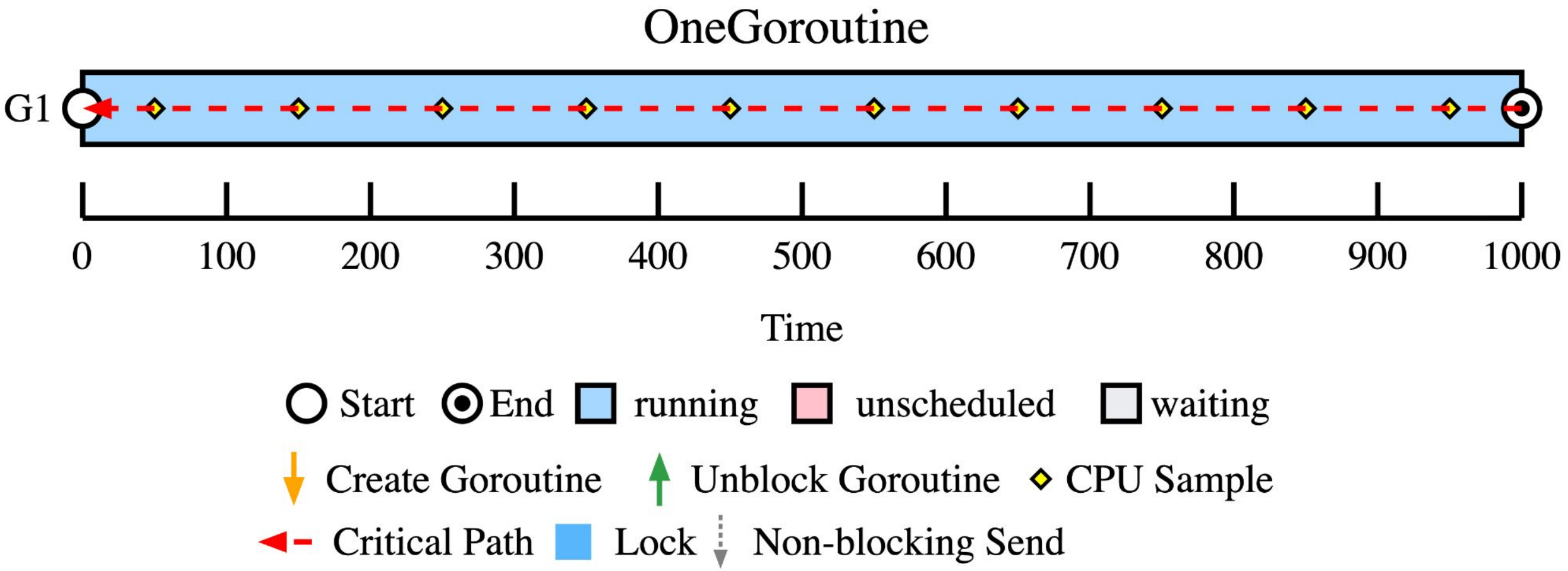
DATADOG

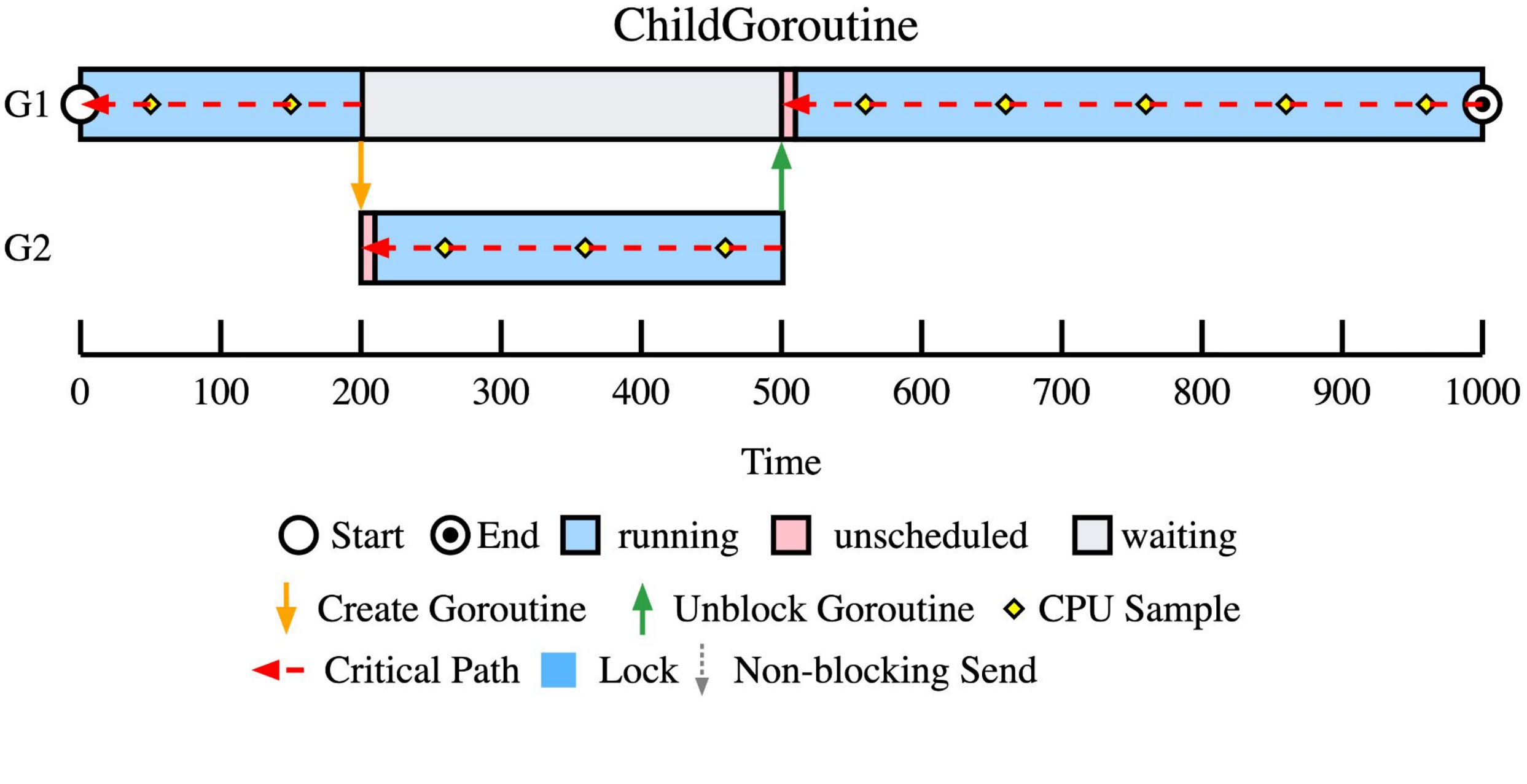
Critical Path Analysis

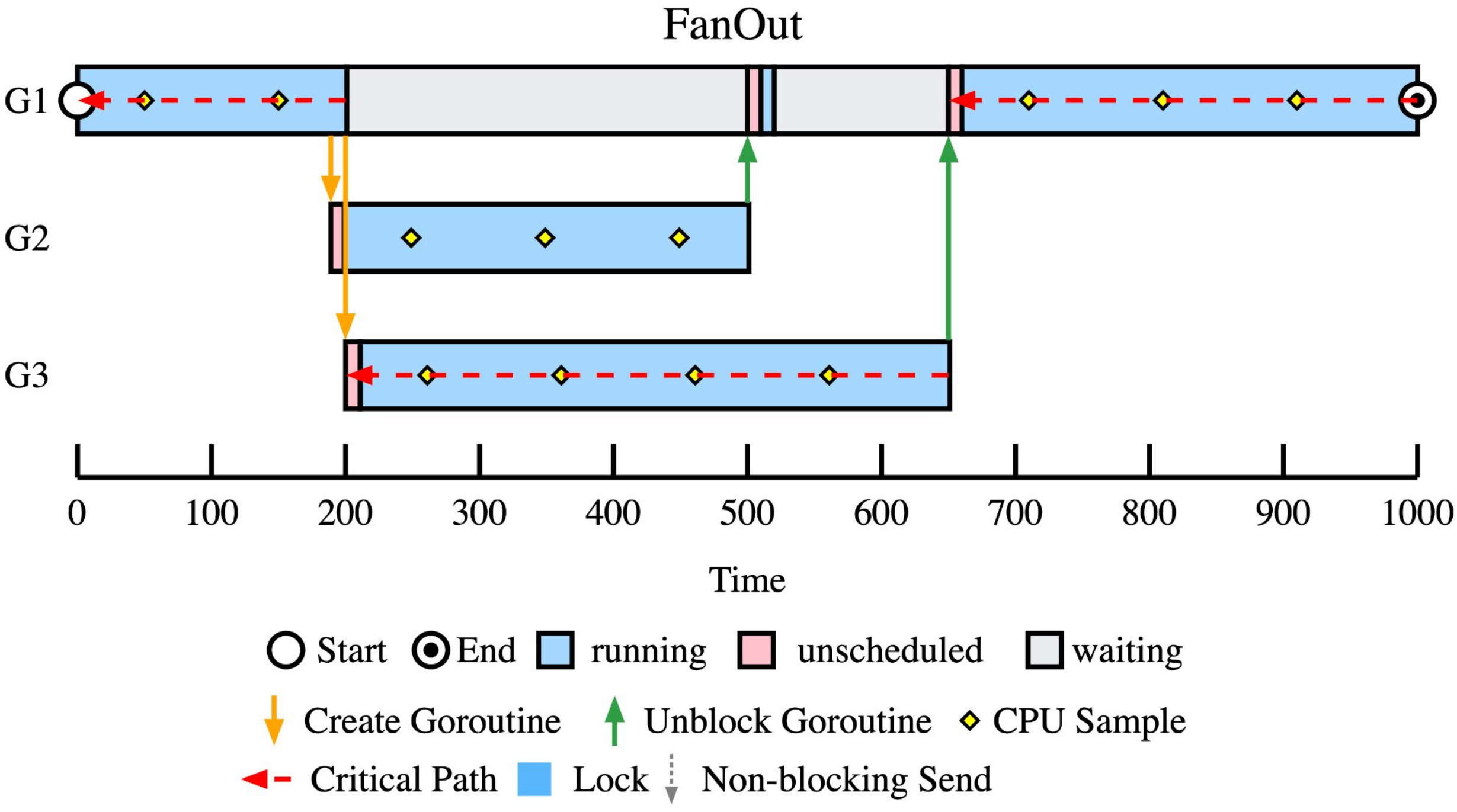
- The precursors of what came to be known as critical path were developed and put into practice by DuPont between 1940 and 1943 and contributed to the success of the Manhattan Project.

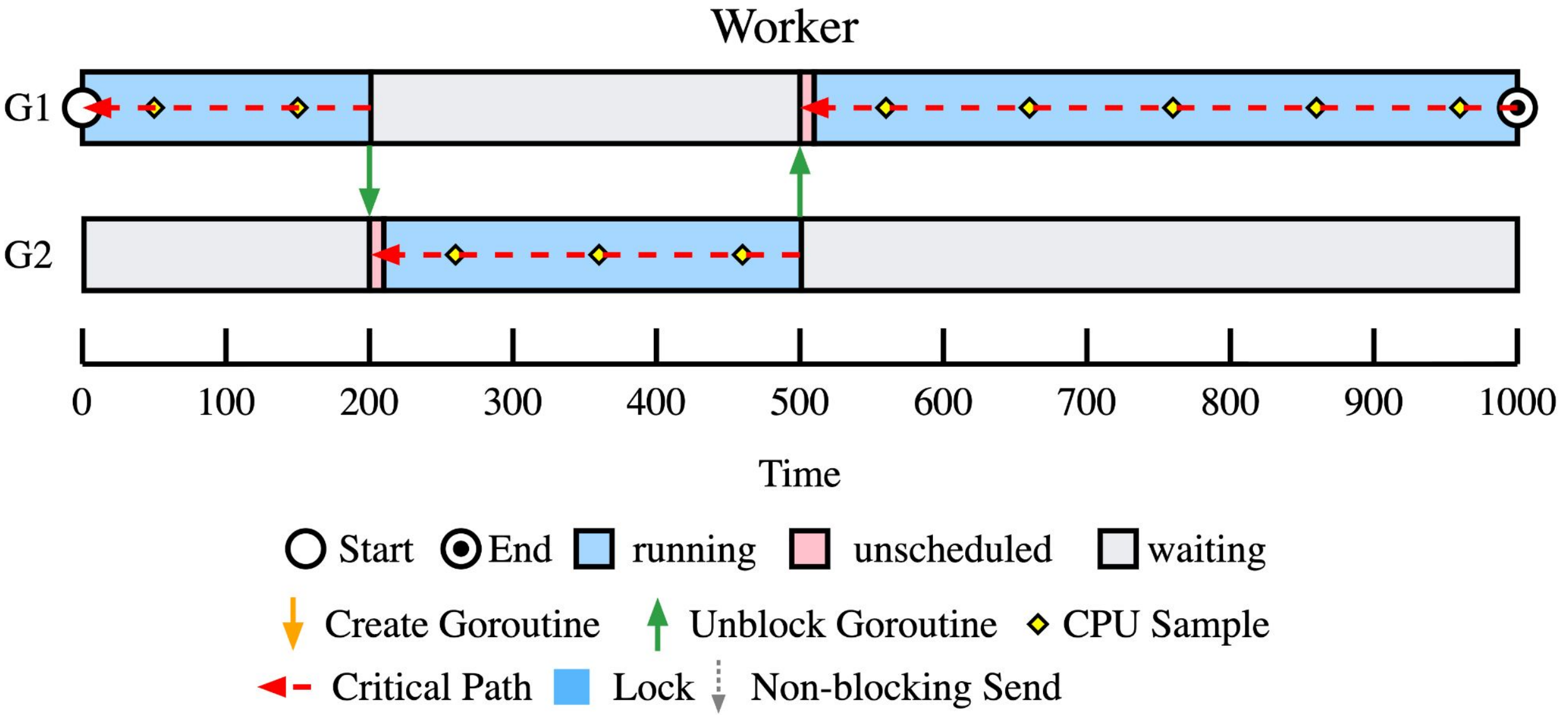
- A critical path is determined by identifying the longest stretch of dependent activities.

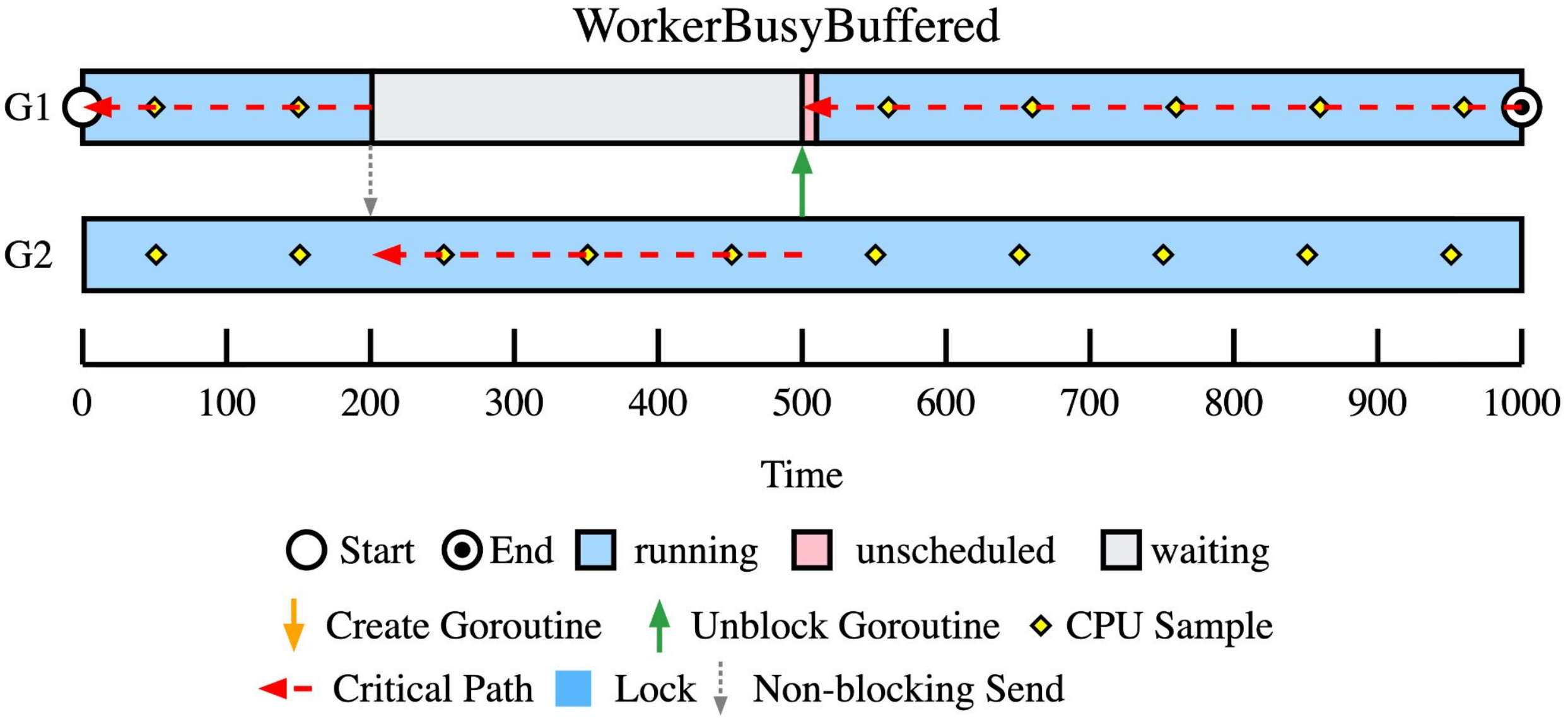


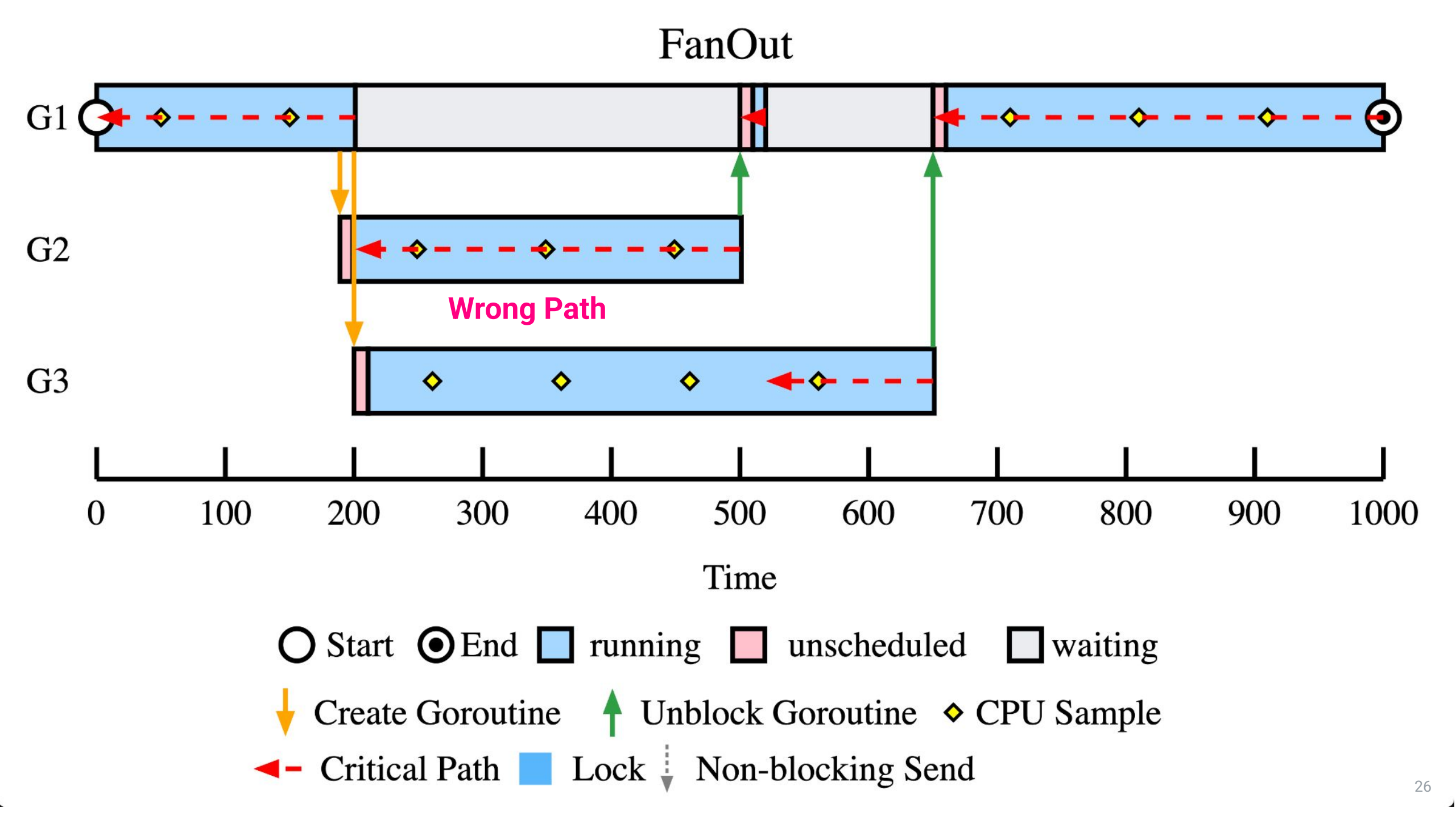

OneGoroutine
G1
0
100
200
300
400
500
600
700
800
900
1000
Time
Start
End
running
unscheduled
waiting
Create Goroutine
Unblock Goroutine
CPU Sample
Critical Path
Lock
Non-blocking Send


ChildGoroutine
G1
G2
0
100
200
300
400
500
600
700
800
900
1000
Time
Start
End
running
unscheduled
waiting
Create Goroutine
Unblock Goroutine
CPU Sample
Critical Path
Lock
Non-blocking Send


FanOut
G1
G2
G3
0
100
200
300
400
500
600
700
800
900
1000
Time
Start
End
running
unscheduled
waiting
Create Goroutine
Unblock Goroutine
CPU Sample
Critical Path
Lock
Non-blocking Send

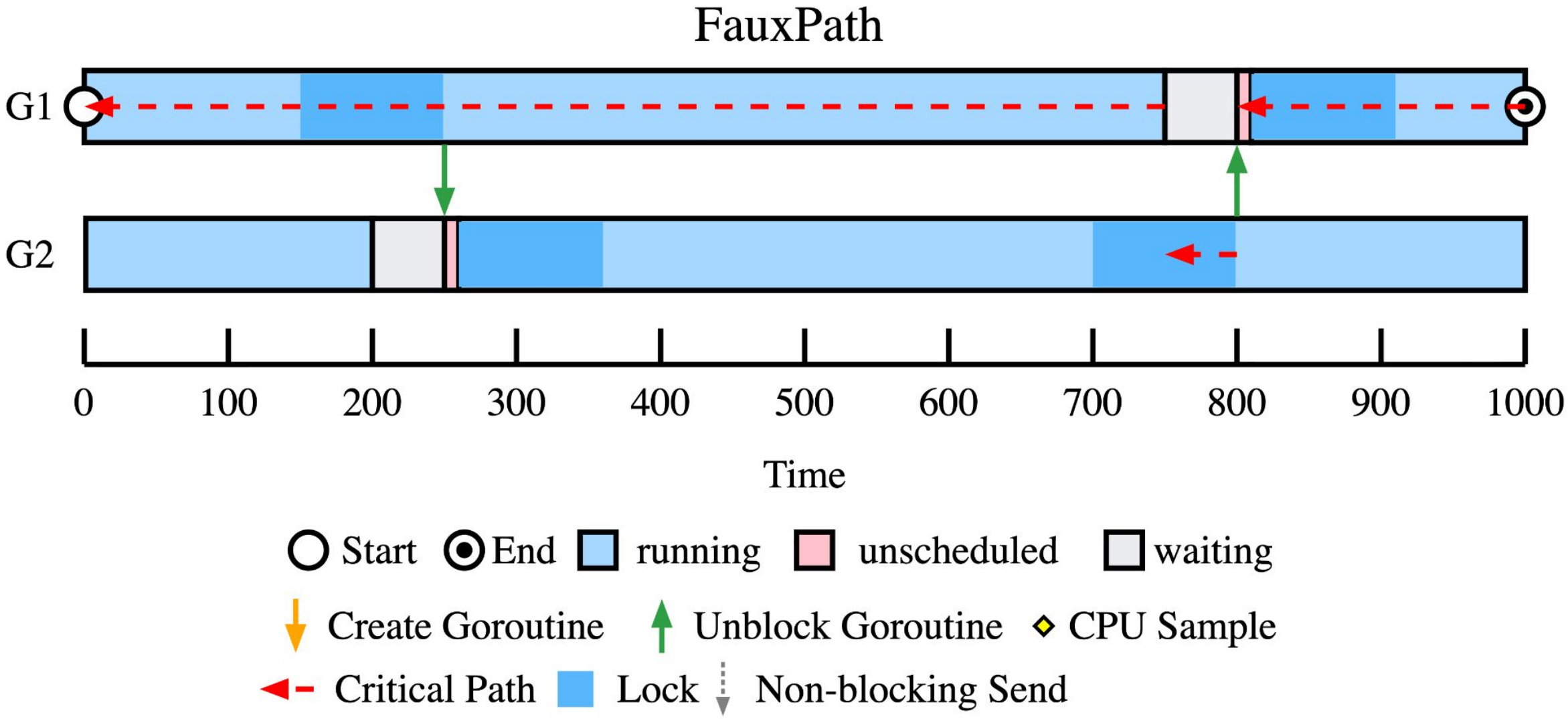

Worker
G1
G2
0
100
200
300
400
500
600
700
800
900
1000
Time
Start
End
running
unscheduled
waiting
Create Goroutine
Unblock Goroutine
CPU Sample
Critical Path
Lock
Non-blocking Send


WorkerBusyBuffered
G1
G2
0
100
200
300
400
500
600
700
800
900
1000
Time
Start
End
running
unscheduled
waiting
Create Goroutine
Unblock Goroutine
CPU Sample
Critical Path
Lock
Non-blocking Send


FanOut
G1
G2
G3
Wrong Path
0
100
200
300
400
500
600
700
800
900
1000
Time
Start
End
running
unscheduled
waiting
Create Goroutine
Unblock Goroutine
CPU Sample
Critical Path
Lock
Non-blocking Send

Critical Path Analysis Algorithm for Go Traces

1. Begin the path on the goroutine of the last event.
2. If the event is a wakeup or goroutine creation.
    a. If the event is a wakeup and the waking goroutine does not have a path to the first event or the stack trace indicates a mutex wakeup:
        i. Limit the waking goroutine to the start time of the wait preceding the wakeup.
        ii. If the current goroutine already has a limit, use the max of the two limits.
    b. Switch to the waking/creating goroutine and continue at step 2.
3. Find the next event mentioning the current goroutine.
4. If the next event is the first event: Success, return the path.
5. If there is a limit and the next event is before it, or there is no next event, return to the goroutine that set the limit.
6. If there is no next event: Failure, return without a path. (*should be impossible*)
7. Continue with the next event at step 2.

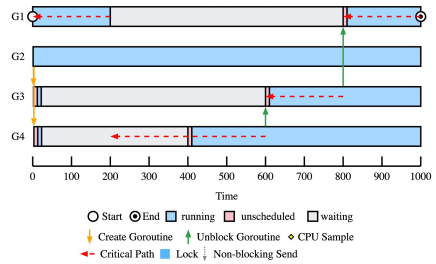

FauxPath
G1
G2
0
100
200
300
400
500
600
700
800
900
1000
Time
Start
End
running
unscheduled
waiting
Create Goroutine
Unblock Goroutine
CPU Sample
Critical Path
Lock
Non-blocking Send

Critical Path Critique

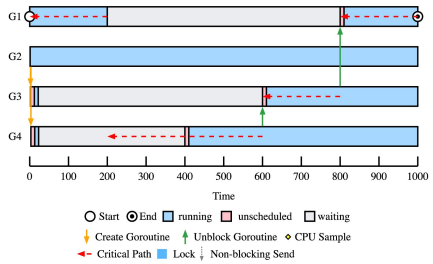
- Some fundamental issues:
  - Mutual exclusion heuristics based on stack traces are not sound.
  - Lost edges on non-blocking send operations.
  - Lost edges due to OS syscall wakeups (e.g. unix pipes).

- But … perfection is the enemy of

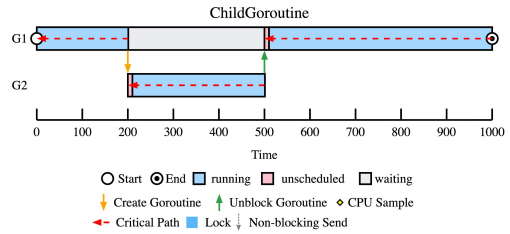
AncestorConfusion



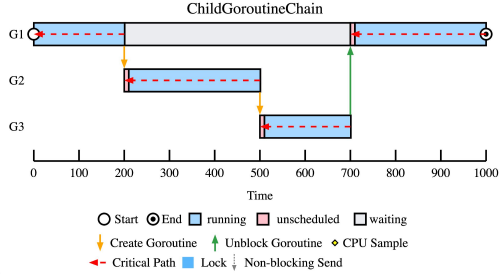
AncestorConfusion



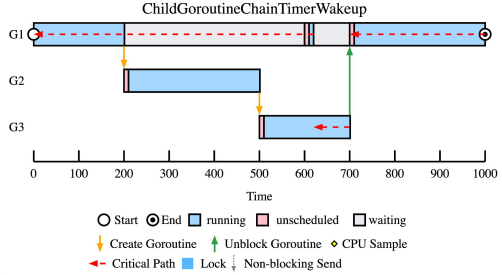
ChildGoroutine



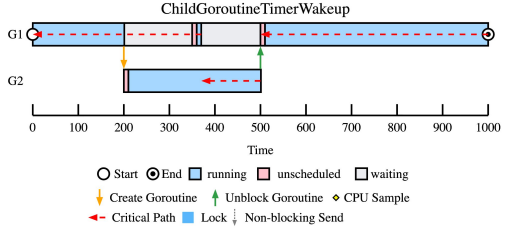
ChildGoroutineChain



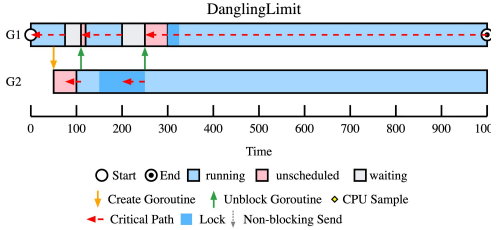
ChildGoroutineChainTimerWakeup



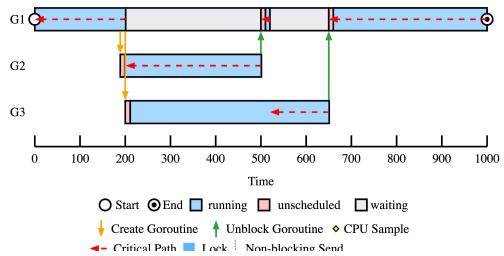
ChildGoroutineTimerWakeup



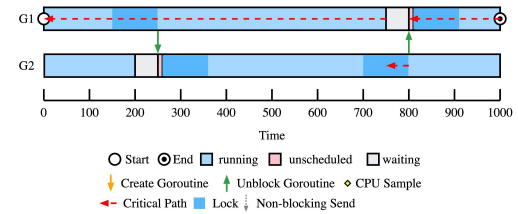
DanglingLimit



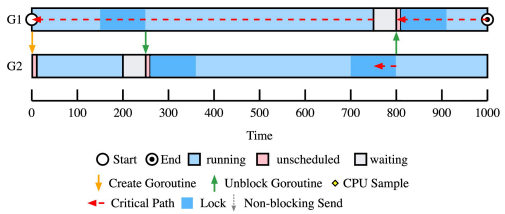
FanOut



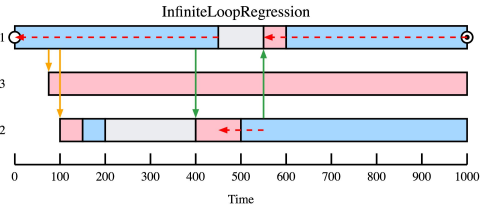
FauxPath



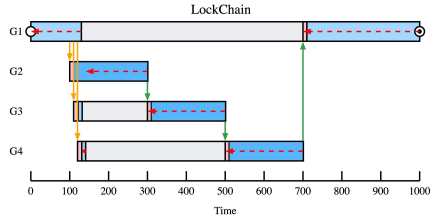
FauxPathChild



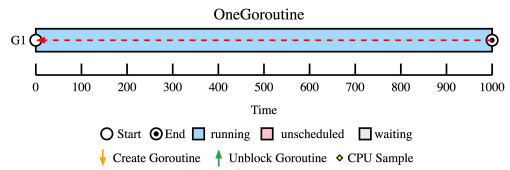
InfiniteLoopRegression



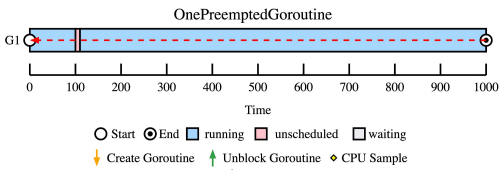
LockChain



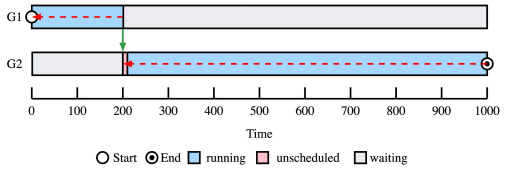
OneGoroutine



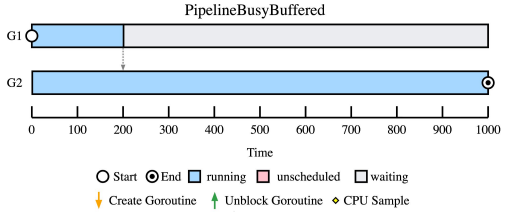
OnePreemptedGoroutine



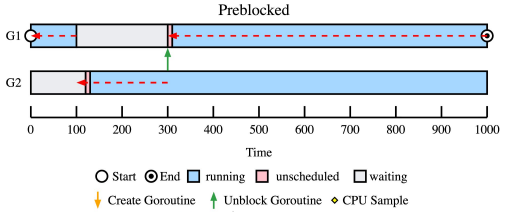
Pipeline



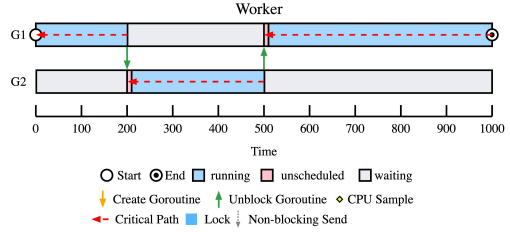
PipelineBusyBuffered



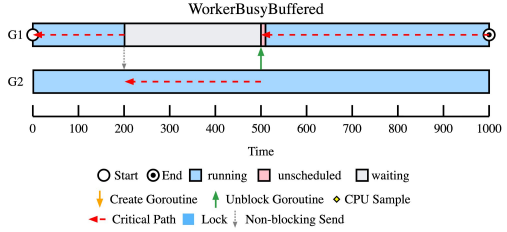
Preblocked



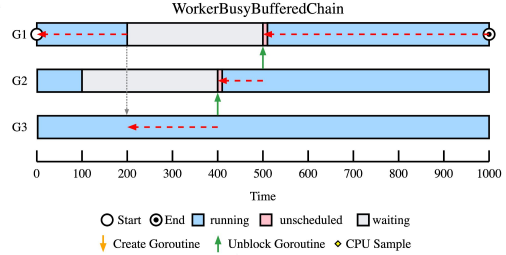
Worker



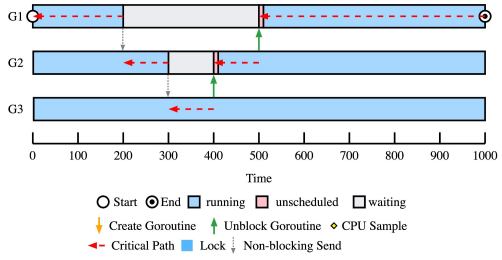
WorkerBusyBuffered



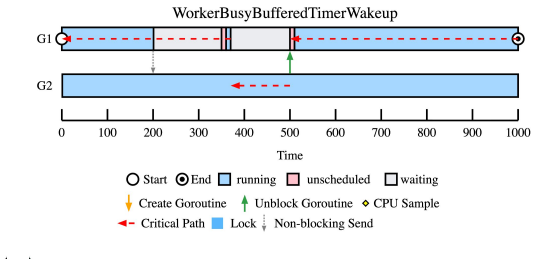
WorkerBusyBufferedChain



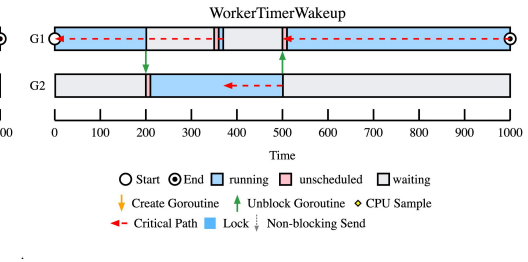
WorkerBusyBufferedChain2



WorkerBusyBufferedTimerWakeup



WorkerTimerWakeup



Summary | Call Tree | Flame Graph

Details

Events	285
Selected Duration	115.26ms
Goroutines	All

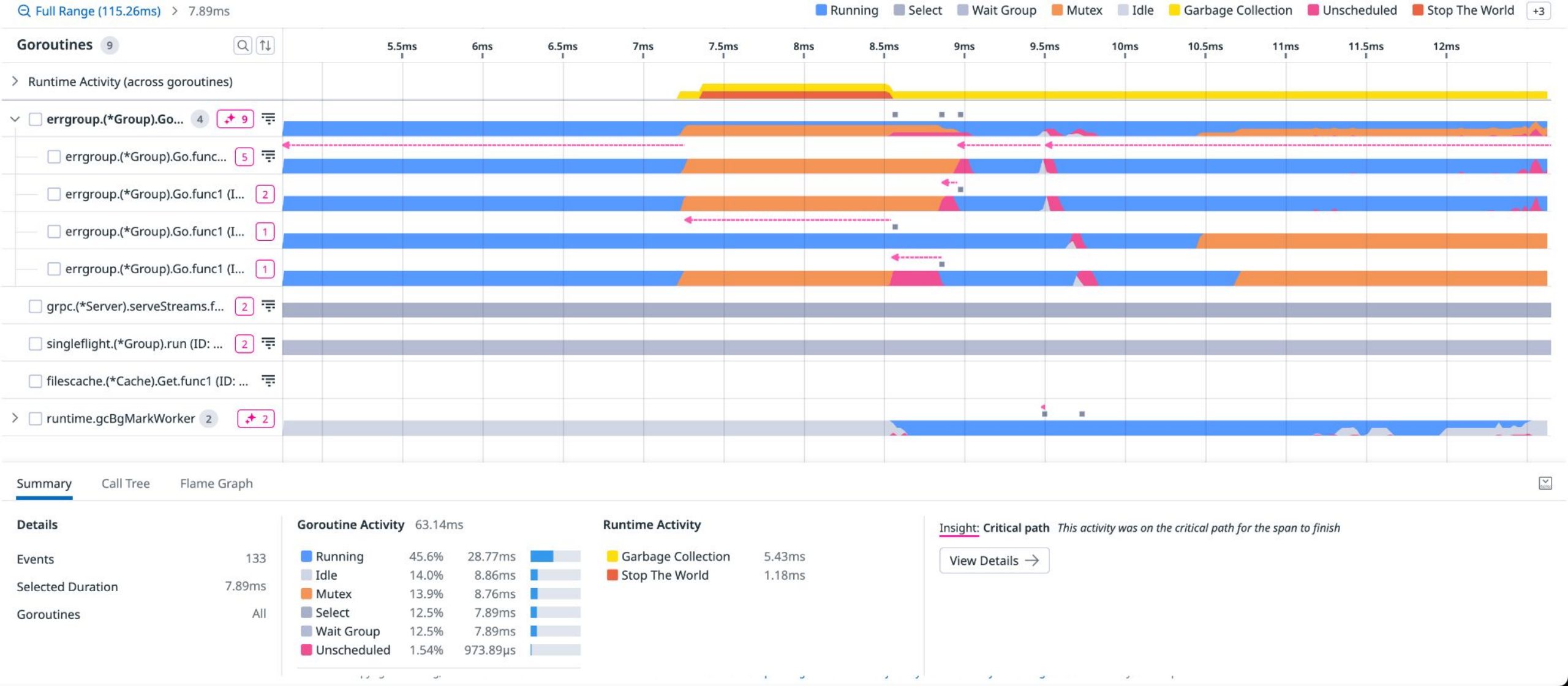
Goroutine Activity 921.83ms

Running	27.9%	257.38ms
Idle	23.7%	218.12ms
Mutex	23.2%	214.11ms
Select	12.5%	114.93ms
Wait Group	12.4%	114.06ms
Unscheduled	0.35%	3.23ms

Runtime Activity

Garbage Collection	10.08ms
Stop The World	1.61ms

DATADOG



Summary | Call Tree | Flame Graph

Details

Events	133
Selected Duration	7.89ms
Goroutines	All

Goroutine Activity 63.14ms

Running	45.6%	28.77ms
Idle	14.0%	8.86ms
Mutex	13.9%	8.76ms
Select	12.5%	7.89ms
Wait Group	12.5%	7.89ms
Unscheduled	1.54%	973.89µs

Runtime Activity

Garbage Collection	5.43ms
Stop The World	1.18ms

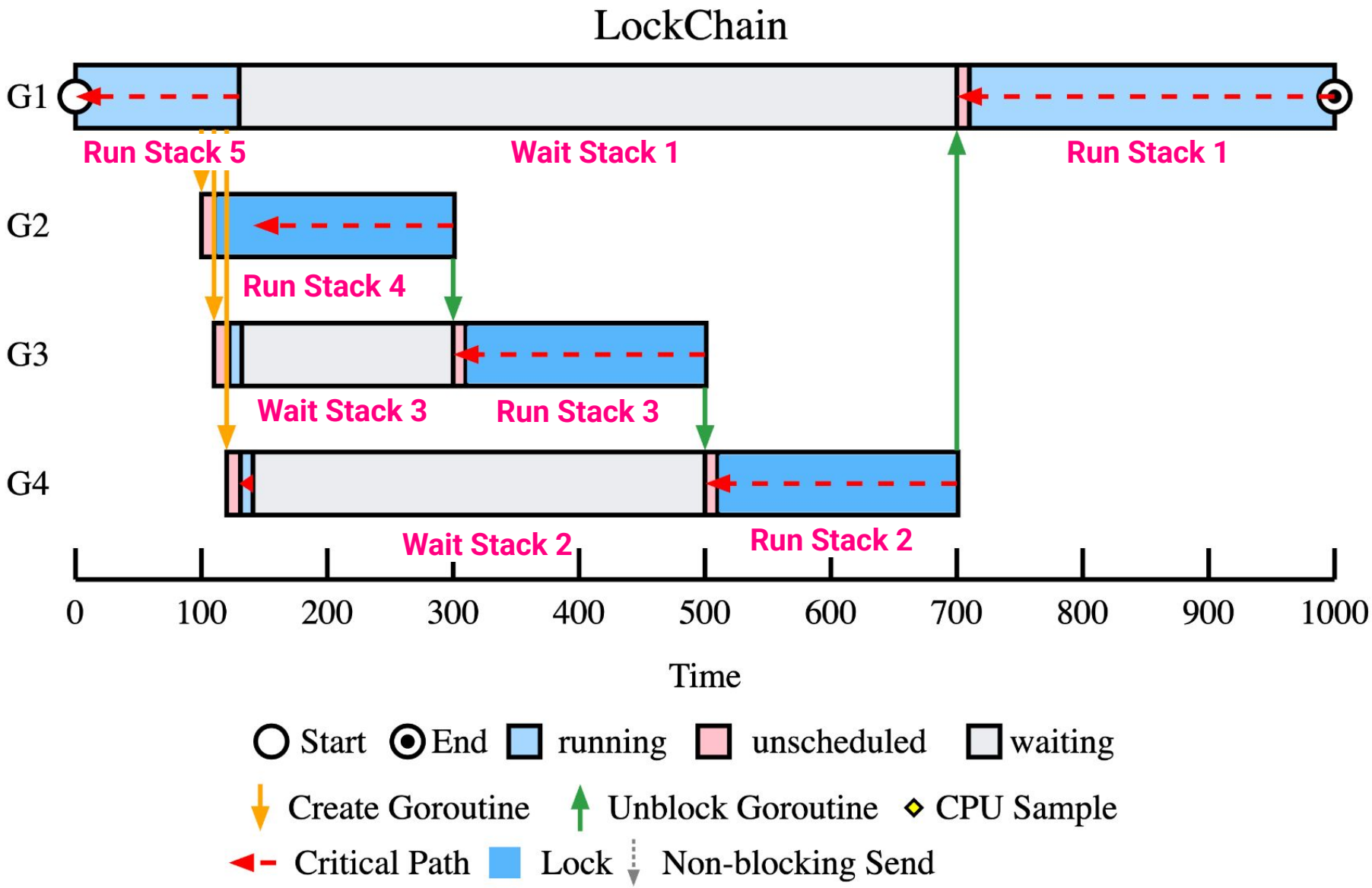
Insight: **Critical path** *This activity was on the critical path for the span to finish*

View Details →

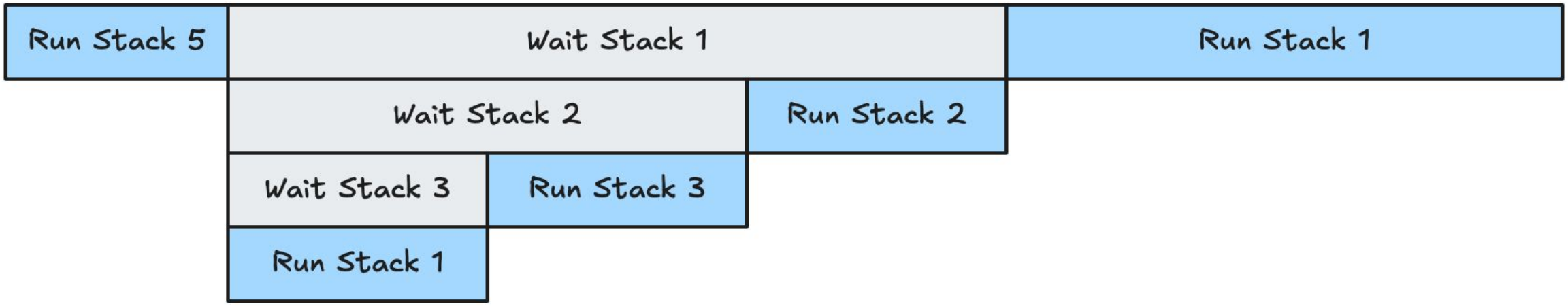
DATADOG


pprof
VIEW
SAMPLE
REFINE
CONFIG
DOWNLOAD
Search regexp
439.2s (100%)
root
errgroup.(*Group).add.func1
converter.(*RowConverter).parseRows-fm
converter.(*RowConverter).parseRows
log.(*traceLogger).Errorf
log.(*zapLogger).Error
zap.(*Logger).Error
zapcore.(*CheckedEntry).Write
zapcore.(*ioCore).Write
zapcore.(*lockedWriteSyncer).Write
sync.(*Mutex).Unlock

Wait Stacks


LockChain
G1
G2
G3
G4
Run Stack 5
Wait Stack 1
Run Stack 1
Run Stack 4
Wait Stack 3
Run Stack 3
Wait Stack 2
Run Stack 2
0
100
200
300
400
500
600
700
800
900
1000
Time
Start
End
running
unscheduled
waiting
Create Goroutine
Unblock Goroutine
CPU Sample
Critical Path
Lock
Non-blocking Send

Wait Stacks


Run Stack 5
Wait Stack 1
Run Stack 1
Wait Stack 2
Run Stack 2
Wait Stack 3
Run Stack 3
Run Stack 1

Datadog 💜 Open Source

Low-overhead tracing (landed in go1.21)

Low-overhead tracing

Prior to Go 1.21, the run-time overhead of tracing was somewhere between 10–20% CPU for many applications, which limits tracing to situational usage, rather than continuous usage like CPU profiling. It turned out that much of the cost of tracing came down to tracebacks. Many events produced by the runtime have stack traces attached, which are invaluable to actually identifying what goroutines were doing at key moments in their execution.

Thanks to work by Felix Geisendörfer and Nick Ripley on optimizing the efficiency of tracebacks, the run-time CPU overhead of execution traces has been cut dramatically, down to 1–2% for many applications. You can read more about the work done here in Felix's great blog post on the topic.

CL Pending: Create traces for testing

x/exp/trace,internal/trace: support event constructor for testing #74826

Edit | New issue

Open

felixge opened last month · edited by felixge — Edits — Contributor

As discussed in #62627, this issue proposed to extend the API of x/exp/trace to support programmatically creating trace events for testing and analysis purposes.

Note: Given that the code for this lives in `internal/trace` right now and is only exposed via `x/exp/trace`, I think this issue doesn't need to go through the official proposal process.

Overview

The proposal is to add an API using a `MakeEvent` constructor that takes a `EventConfig` struct. Below is an example for creating a goroutine state transition event:

```
details := MakeGoStateTransition(5, GoRunning, GoWaiting)
details.Stack = MakeStack([]StackFrame{
→   {PC: 1, Func: "time.Sleep", File: "runtime.go", Line: 10},
→   {PC: 2, Func: "main", File: "main.go", Line: 20},
})
details.Reason = "sleep"
ev, err := MakeEvent(EventConfig[StateTransition]{
····Time:·····timestamp,
····Goroutine: 1,
····Proc:·····2,
····Thread:····3,
····Details:···details,
})
```

A partially completed prototype of this API is available in CL 691815.

Assignees

No one assigned

Labels

FeatureRequest | LibraryProposal | NeedsDecision

Type

No type

Projects

No projects

Milestone

Unreleased

Due by December 31, 2099, 82% complete

Relationships

None yet

Development

Code with agent mode

No branches or pull requests

Thank You

Follow @felixge.de (BlueSky) or, felixge (X) for updates on Critical Path Analysis