Concurrency Analysis of Asynchronous APIs

Anirudh Santhiar and Aditya Kanade

April 2, 2016

Computer Science and Automation, IISc

Introduction

Asynchronous Programming Model[1]

A way to organize programs to avoid blocking.

In a nutshell

waiting in line for your idly

vs

registering your order, doing other things, having store call you when ready.

We analyze the concurrency behaviours of

- Event driven asynchronous libraries with programmatic event loops to detect races (joint work with S. Kaleeswaran)
- C# asynchronous programs to find deadlocks

[1] Images courtesy `tripadvisor.in` and `commons.wikimedia.com`

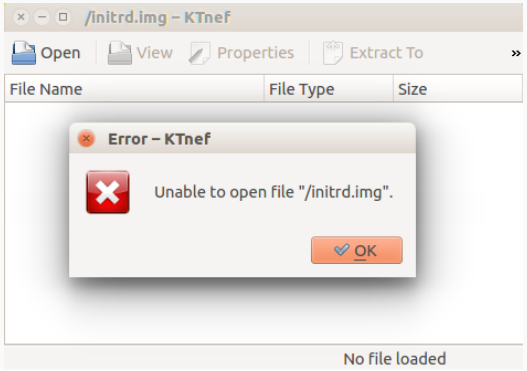
Races involving Programmatic Event Loops

- An Event Loop is the basic scheduling mechanism for programs that respond to asynchronous events

```
while(!exit) {
  e = nextEvent();
  process e;
}
```

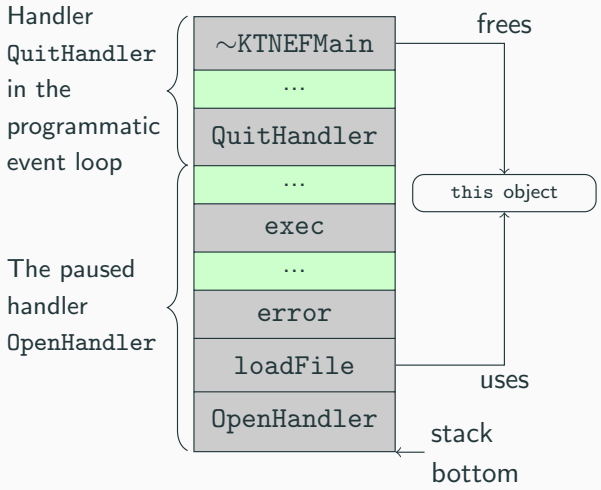
- We consider frameworks where event loops can also be spun programmatically by event handlers
- Improve responsiveness while waiting for the user or network
- Prone to interference between handler spinning event loop and handler running inside the loop

Bug in KTNef



Bug

Close the window when an error dialog is shown.



Interference between paused handler and handler running inside programmatic event loop

- The `FileOpen` event's handler spins a programmatic event loop during the time the error dialog is shown
- There is a race between `FileHandler` and `QuitHandler` that runs in the programmatic event loop

Goal

Reason about non-determinism introduced by programmatic event loops to detect such races.

Our Work

Technical Highlights

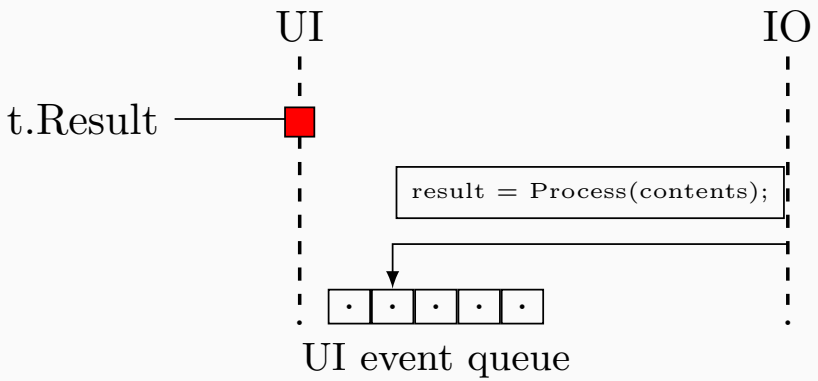
- Powerful happens-before framework to detect races beyond the state-of-the art
- Account for all general scheduling scenarios e.g., recursive and cascaded programmatic event loops
- Novel sparse happens-before relation enabling faster race detection

Contributions

- Analyzes the programmatic event loop mechanism - widely used in OS APIs, GUI libraries, Browsers
- Presents happens-before rules to detect race conditions
- Efficient computation of the happens-before relation: 5× speedup
- 13 new and harmful race conditions in 9 open-source applications including Okular, Kate and KOrganizer
- Tools: Instrumentation framework and race detector SparseRacer

Deadlocks in Asynchronous Programs

Mixing synchronous and asynchronous waiting can lead to deadlocks




```
async Task<String> GetContentsAsync(Uri uri)
{
  using (var client = new HttpClient()){
    // async wait
    var contents = await client.GetStringAsync(uri);
    result = Process(contents); // continuation
    return result;
  }
}


public void Button1_Click(...)
{
  var t = GetContentsAsync(...);
  resultBox.Text = t.Result; // sync wait
}
```

- `t.Result` is a blocking call that prevents `GetContentsAsync` from completing
- In turn, the only way to unblock `t.Result` is for `GetContentsAsync` to complete

The deadlock is observed even though there is no explicit thread creation and locking.

Our Work

- Design a static analysis to detect such deadlocks.
- Static analysis captures C# semantics for scheduling and async/await - key to determining where suspensions resume, and therefore deadlocking behaviour.
- Analyzes control flow in the presence of continuations and APIs affecting scheduling behaviour
- Preliminary results are encouraging - Prototype tool has found previously unknown deadlocks in 7 open source applications

Conclusions

- Programmers increasingly use powerful language features and APIs to structure their programs to take advantage to asynchrony
- However, this can allow non-determinism at runtime or make reasoning about the flow of control difficult, leading to bugs that are difficult to diagnose or reproduce
- We have designed static and dynamic techniques to help understand the behaviour of asynchronous programs better, and tools to automatically find some of these bugs that are beyond the state-of-the-art.

Extra Slides

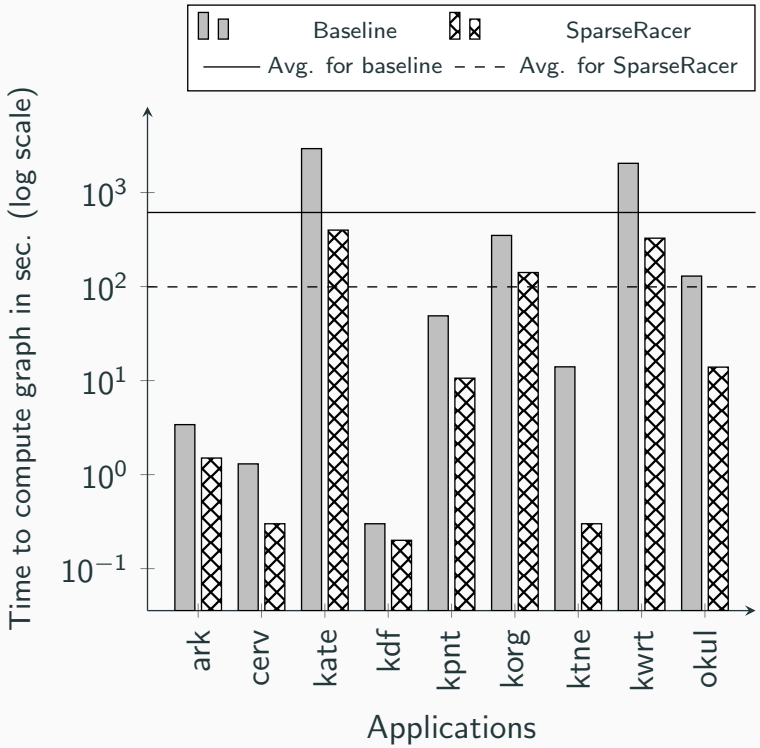
Solution Idea

Idea

- Find bugs using non-buggy executions
- Design trace language to record interesting operations
- Design *happens-before rules* to detect possible reorderings of these
- Determine if there is a re-ordering of event handlers so that *conflicting operations* such as uses and frees can be reordered to induce bugs
- Notify programmer about such re-orderings along with debug information

Results

Application	#Ops	#Evts	#Blks	#Interf. Handlers		Time in sec
				Total	True	
Ark	5008	127	148	6	2	2
Cervisia	1726	129	156	14	2	0.3
Kate	83633	194	225	4	1	480
KDF	1089	15	24	1	1	0.2
Kolourpaint	12746	67	75	2	1	12
KOrganizer	58232	273	290	12	2	179
KTnef	1158	258	275	1	1	0.3
KWrite	74105	62	75	4	1	396
Okular	16785	223	273	14	2	15
Total				58	13	



Effective!

SparseRacer found 13 harmful use after free bugs in 9 popular open source applications.

Fast!

SparseRacer was $5X$ faster than the baseline in race detection time.

Asynchronous Operations[2]

Synchronous Operations

Do not permit the caller to proceed until the operation completes

Asynchronous Operations

Decouple initiating the operation (short-lived) from waiting for it to complete

Concurrency

Asynchrony enables concurrent execution

- Postpone the waiting
- Overlap the waiting periods of multiple operations
- Avoid waiting by registering callbacks

[2] Slide inspired by Claudio Russo