User-level Threading: Have Your Cake and Eat It Too

Martin Karsten and Saman Barghi

David R. Cheriton School of Computer Science
University of Waterloo

June 2020

Motivation

- application programming paradigms
  - network service handling concurrent sessions

Motivation

- application programming paradigms
  - network service handling concurrent sessions
- event-based programming
  - explicit state management
  - asynchronous control flow $\rightarrow$ callback hell

Motivation

- application programming paradigms
  - network service handling concurrent sessions
- event-based programming
  - explicit state management
  - asynchronous control flow $\rightarrow$ callback hell
- thread-per-session programming
  - automatic state management
  - synchronous control flow

Motivation

- application programming paradigms
  - network service handling concurrent sessions
- event-based programming
  - explicit state management
  - asynchronous control flow $\rightarrow$ callback hell
- thread-per-session programming
  - automatic state management
  - synchronous control flow

$\Rightarrow$ *performance?*

Background

- parallel hardware → threads & synchronization

Background

- parallel hardware → threads & synchronization
- kernel thread caveats
  - limit: typically 10Ks
  - (some) execution overhead
  - complex scheduling for fairness & control

Background

- parallel hardware → threads & synchronization
- kernel thread caveats
  - limit: typically 10Ks
  - (some) execution overhead
  - complex scheduling for fairness & control

⇒ user-level threads!

- key aspect: scheduling
- requirement: user-level I/O blocking

Take Away

- user-level threads
  - similar throughput to event-based programming
  - load balancing can sometimes reduce tail latency

Take Away

- user-level threads
  - similar throughput to event-based programming
  - load balancing can sometimes reduce tail latency
- kernel threads not that bad either
  - up to a limit

Take Away

- user-level threads
  - similar throughput to event-based programming
  - load balancing can sometimes reduce tail latency
- kernel threads not that bad either
  - up to a limit
- *Fred Runtime* rules!

Table of Contents

Problem Statement

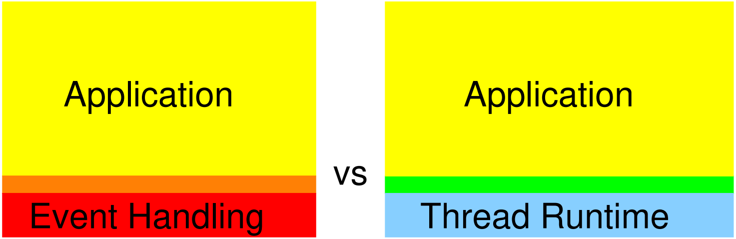
- minimum overhead of user-level threading?

Problem Statement

- minimum overhead of user-level threading?

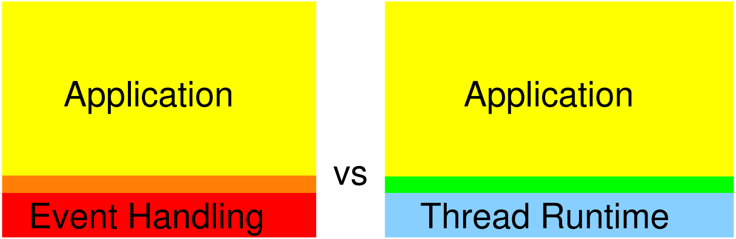
- roadmap
  - build minimum viable user-level threading runtime
  - compare to state of the art threading runtimes
  - evaluate production-grade application

Approach



UNIVERSITY OF WATERLOO

Approach



- Memcached - in-memory key/value store
  - minimum port to thread-per-session
  - fully preserved state machine
  - no structural benefits

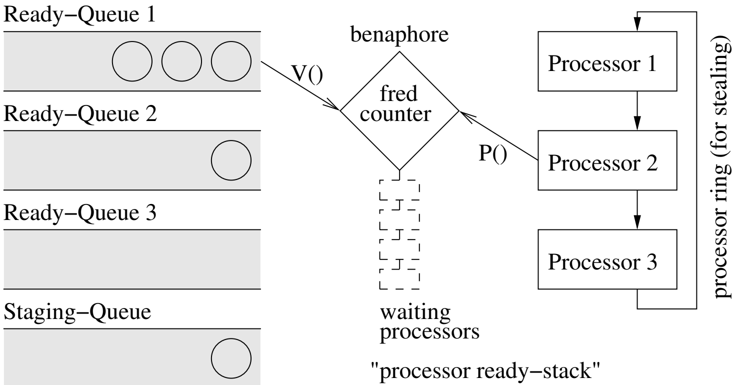
UNIVERSITY OF WATERLOO

Table of Contents

Scheduler

- performance: simple and lightweight
- scalability: local queueing
- effectiveness: load sharing
- efficiency: idle-sleep

Inverse Shared Ready Stack



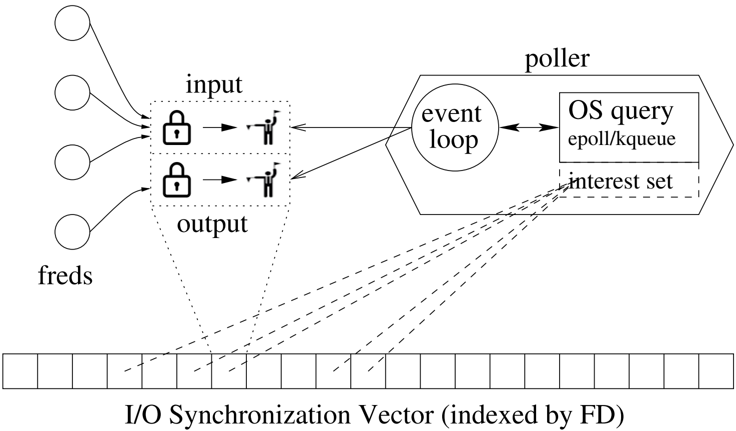
UNIVERSITY OF WATERLOO

I/O Blocking

- automatically suspend thread during I/O wait
- essential for synchronous control flow
- suspend/resume user-level thread
  - user-level synchronization primitives
  - OS-level notifications

UNIVERSITY OF WATERLOO

I/O Notifications



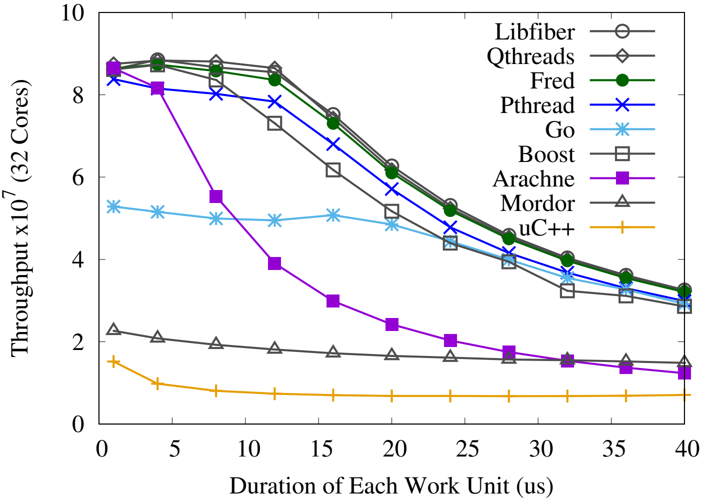
UNIVERSITY OF WATERLOO

Table of Contents

Threading Benchmarks

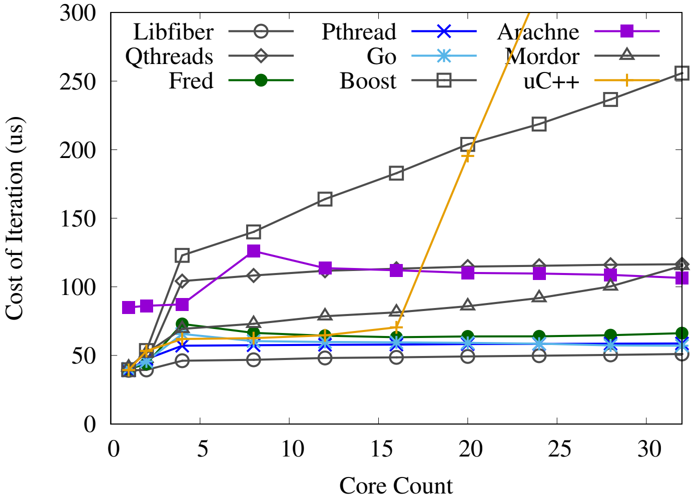
- comparison of 9 different threading runtimes
- performance & scalability problems
  - Arachne, Mordor, μC++
- efficiency problems
  - Arachne, Boost, Qthreads
  - busy-looping scheduler
- solid results
  - Fred, Libfiber, Pthreads
  - Go: higher constant scheduling overhead

Performance



UNIVERSITY OF WATERLOO

Efficiency

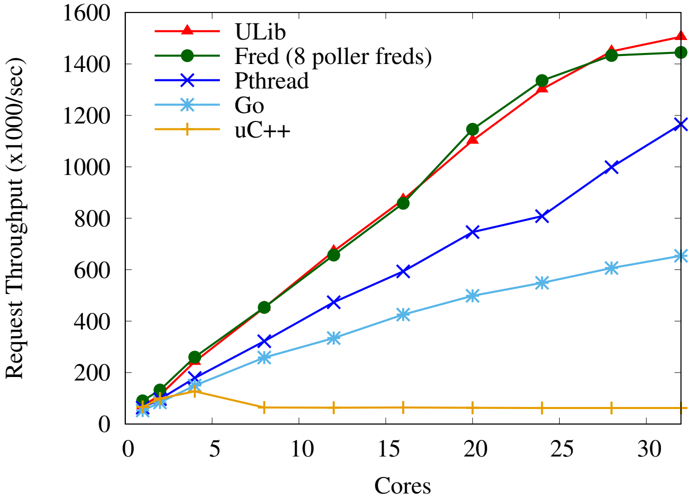


Libfiber
Qthreads
Fred
Pthread
Go
Boost
Arachne
Mordor
uC++
Cost of Iteration (us)
Core Count
300
250
200
150
100
50
0
0
5
10
15
20
25
30

I/O Benchmarks

- I/O stress test for Fred, Go, Libfiber, Pthread
- compared to best-in-class event-based server
  - Libfiber breaks
  - Go and Pthread limited
  - only Fred competitive

I/O Scalability



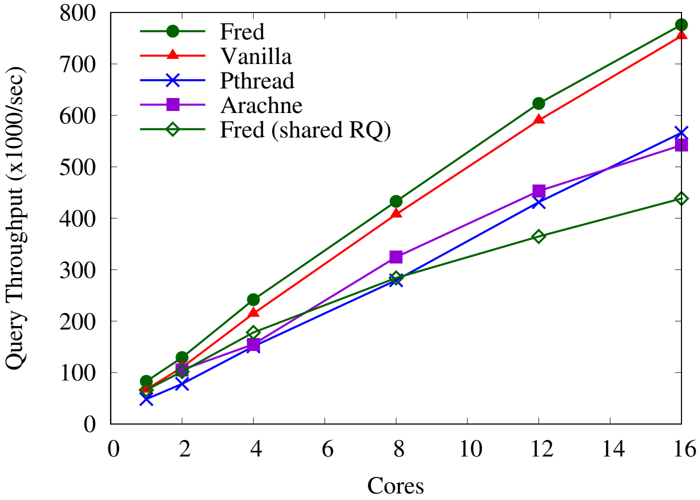
UNIVERSITY OF WATERLOO

Application Benchmarks

Application Benchmarks

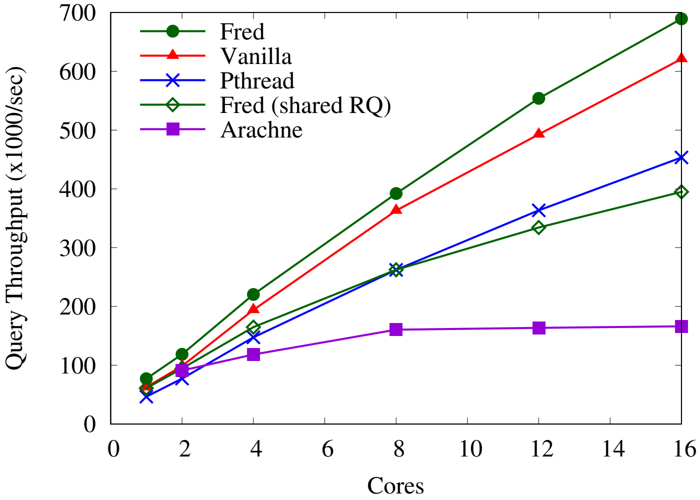
- only Fred competitive with original Memcached
- tail latency results from Arachne paper
  - only apply to special case: #RX queues $<$ #cores
  - performance of Pthread for low connection count!

Throughput



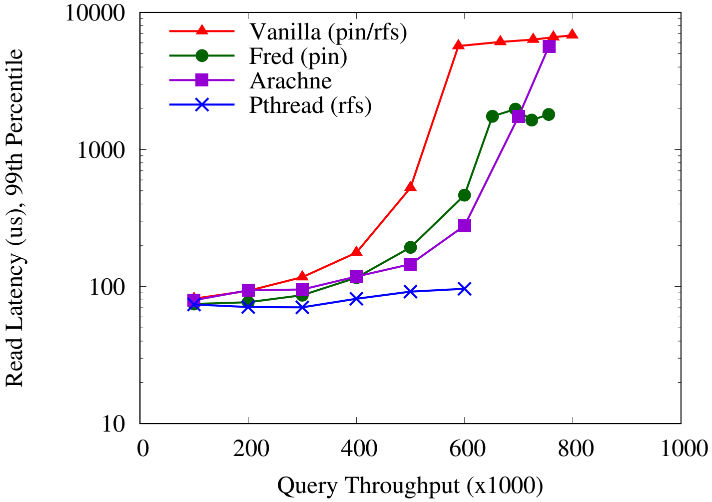
UNIVERSITY OF WATERLOO

Throughput - more connections



UNIVERSITY OF WATERLOO

Tail Latency: Arachne Results

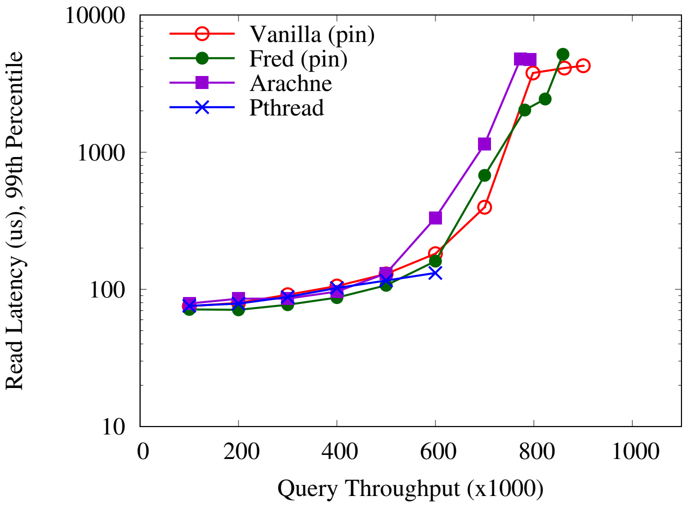


Tail Latency: Explanation

- original experiment: 8 RX queues for 12 cores
- head-of-line blocking?

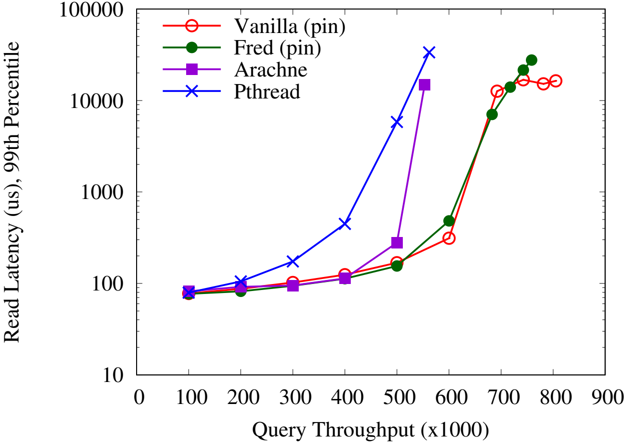
- modified setup: 16 RX queues for 12 cores
- tail latency discrepancies largely gone...

Tail Latency: Regular



Tail Latency: Higher Connection Count

- 1,536 → 7,680 connections



UNIVERSITY OF WATERLOO

Table of Contents

Wrap Up

- Fred: nimble user-level threading runtime
- comprehensive performance evaluation
- user-level threading possible at low overhead
- scenarios with improved performance?
- Fred currently the best reference platform