

Building low latency networking channel

Maxim Vorobjov

Volition Technologies

About me

- 5+ years with Rust
- 3 years in HFT
- JavaScript, Python, C, C++ ... in background
- gh: dunnock
- t: maxsparr0w

Teaser

- What is low latency? 🚗
- What problems are we trying to solve? 🐉
- Deep dive to microseconds level 🐟
- Discover CPU cache and system scheduler impact 🔬
- 🦀

Low latency refers to the short amount of time it takes for a signal or data to travel from one point to another in a system

Low latency

High performance

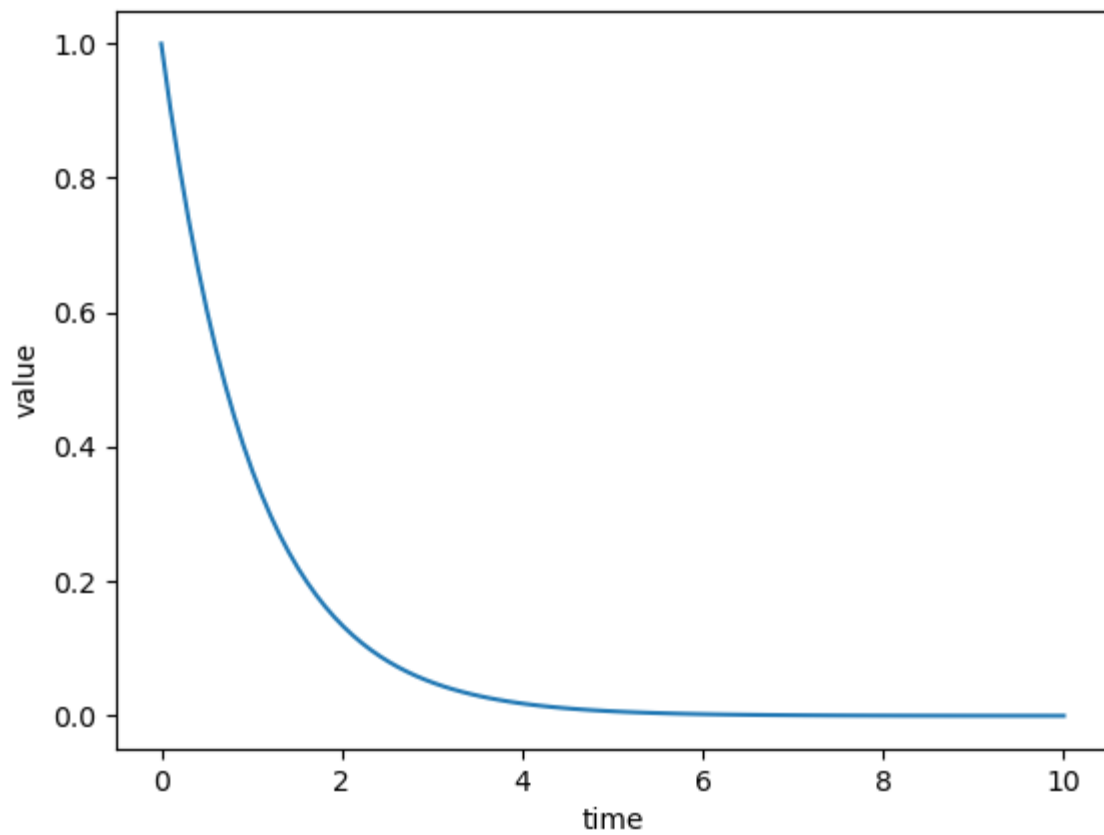
≠

↓ time ↓ dev

↑ throughput

Rust is ideal choice for low latency, it handles technical risks without sacrificing performance and provides access to low level control over execution.

When low latency matters?



Applications

- HFT
- RTB
- IoT
- Other realtime applications

Nothing comes for free

except zero cost abstractions

Assumptions

- Sequence is less important than time
- Allowance for losing messages
- Maximum control over execution

Network protocol

Choosing network protocol

- UDP is convenient protocol with minimum required guarantees
- TCP guarantees sequence and delivery which might cost $\sim 100\mu\text{s}$ - 1ms

UDP constraints

- Event must fit MTU = 1500b
- UDP multicast requires specialized hardware, not free in the cloud

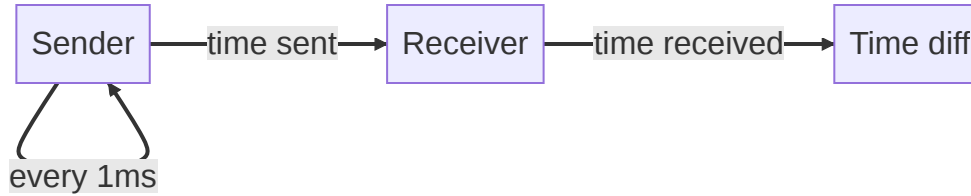
Implementation

Traditional event-loop

```
1  let channel: std::net::UdpSocket;  
2  loop {  
3      match channel.recv(&mut buf) /* .await */ {  
4          Ok(len) => handle_message(&buf[..len]),  
5          Err(err) => handle_error(err),  
6      }  
7  }
```

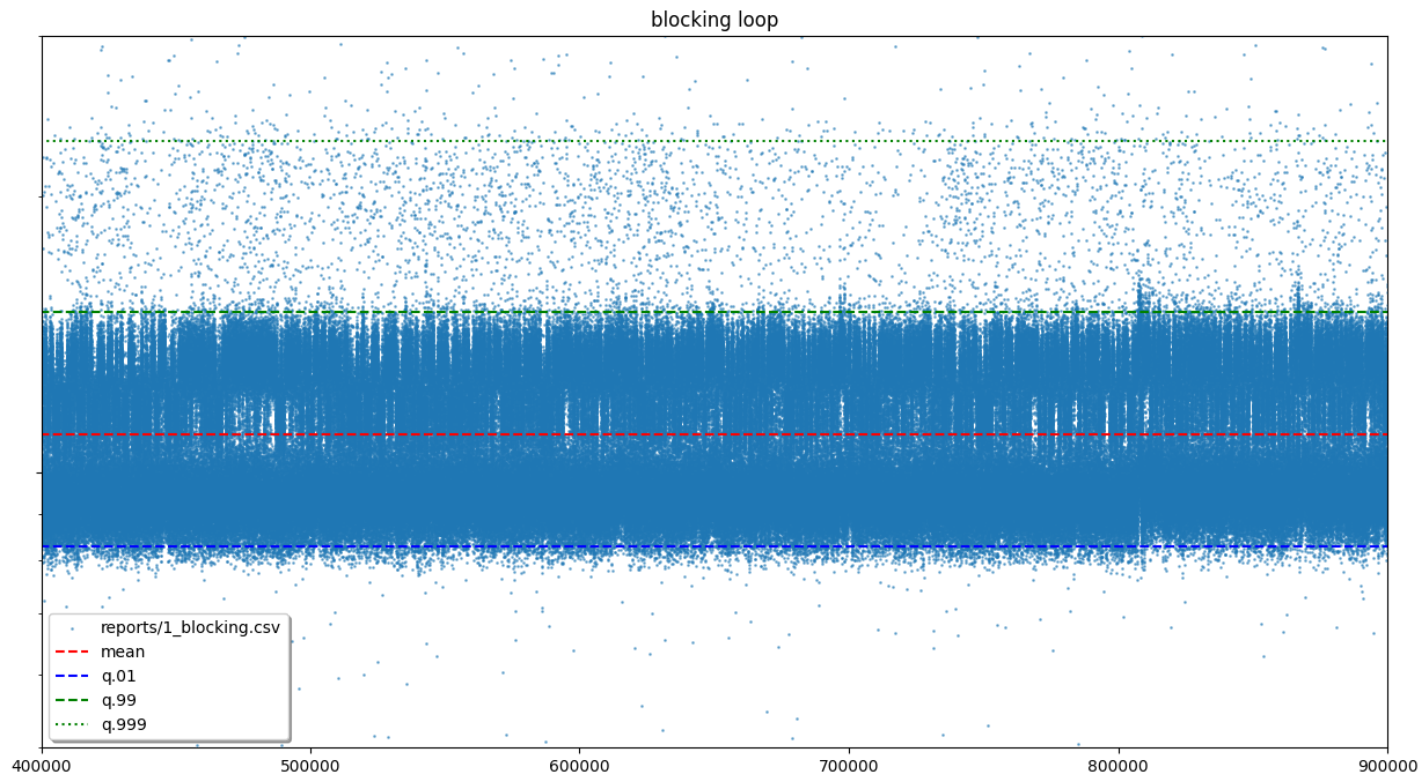
bincode

Measurement method



```
1 $ bin/receive -c ${PRIVATE_IP}:3000 -n 100000 > 1_blocking.csv &  
2  
3 $ bin/send -c ${PRIVATE_IP}:3000 -s ${PRIVATE_IP}:3001 -t 1000 -n 1000000
```

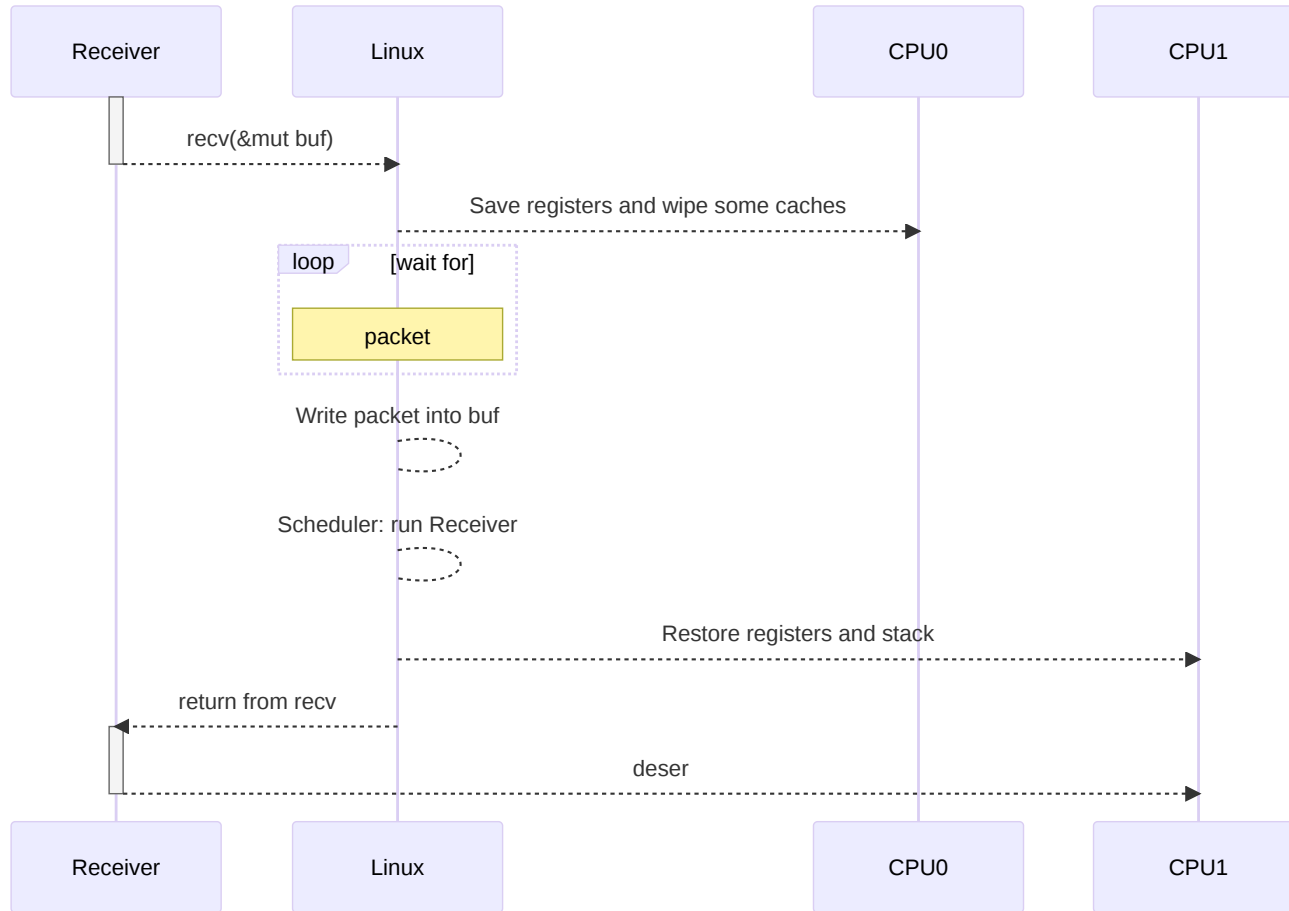

Measurement results



Profile

```
1  $ perf stat bin/receive -c ${PRIVATE_IP}:3000 -n 100000
2
3      339.31 msec task-clock          #    0.003 CPUs utilized
4      100329      context-switches   #   295.683 K/sec
5           2      cpu-migrations     #    5.894 /sec
6       787        page-faults        #    2.319 K/sec
```

Context switching



CS impact

- ~8-14 μ s - In our case
- ~8 μ s - Linux perf event Features and Overhead | Vincent Weaver | University of Maine
- ~3 μ s - How long does it take to make a context switch? | Tsuna's blog

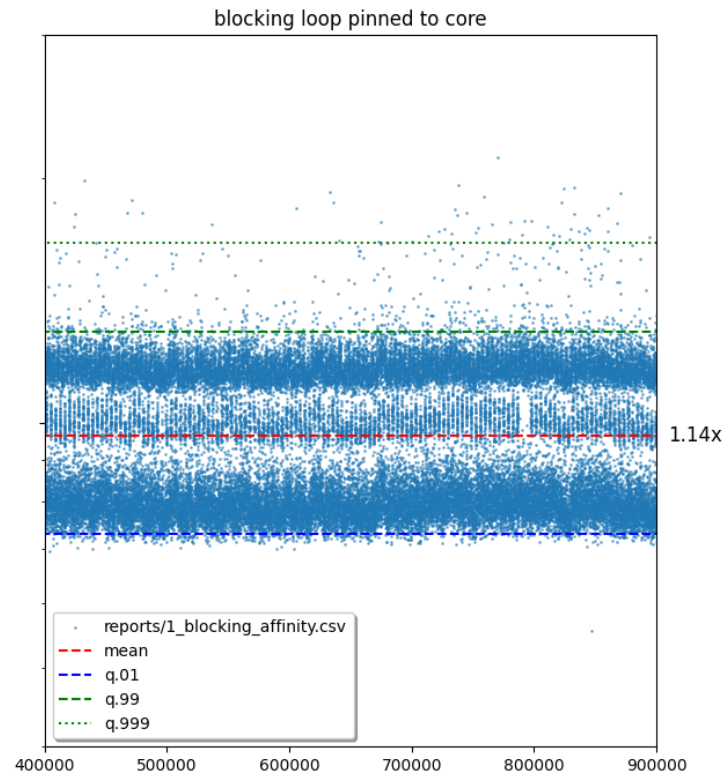
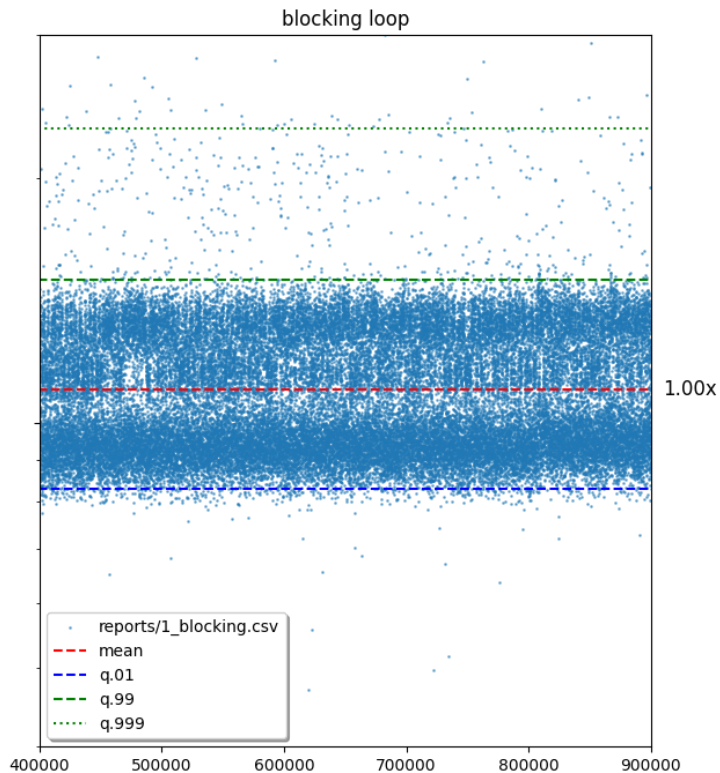
Let's pin our receiver to CPU Core

```
1  let channel: std::net::UdpSocket;  
2  core_affinity::set_for_current(core_id);  
3  loop {  
4      match channel.recv(&mut buf) /* .await */ {  
5          Ok(len) => handle_message(&buf[..len]),  
6          Err(err) => handle_error(err),  
7      }  
8  }
```

core_affinity

Use isolated cores via `isolcpus=1-7` kernel setting

Measure and compare



crossbeam_channel::array

```
1 pub(crate) fn recv(&self, deadline: Option<Instant>) -> Result<T, RecvTimeoutError> {
2     ...
3     loop {
4         if self.start_recv(token) {
5             return unsafe { self.read(token) };
6         }
7         if backoff.is_completed() {
8             break;
9         } else {
10             backoff.snooze();
11         }
12     }
13     ...
14     // Block the current thread.
15     let sel = cx.wait_until(deadline);
16     ...
17 }
```

crossbeam_utils::backoff

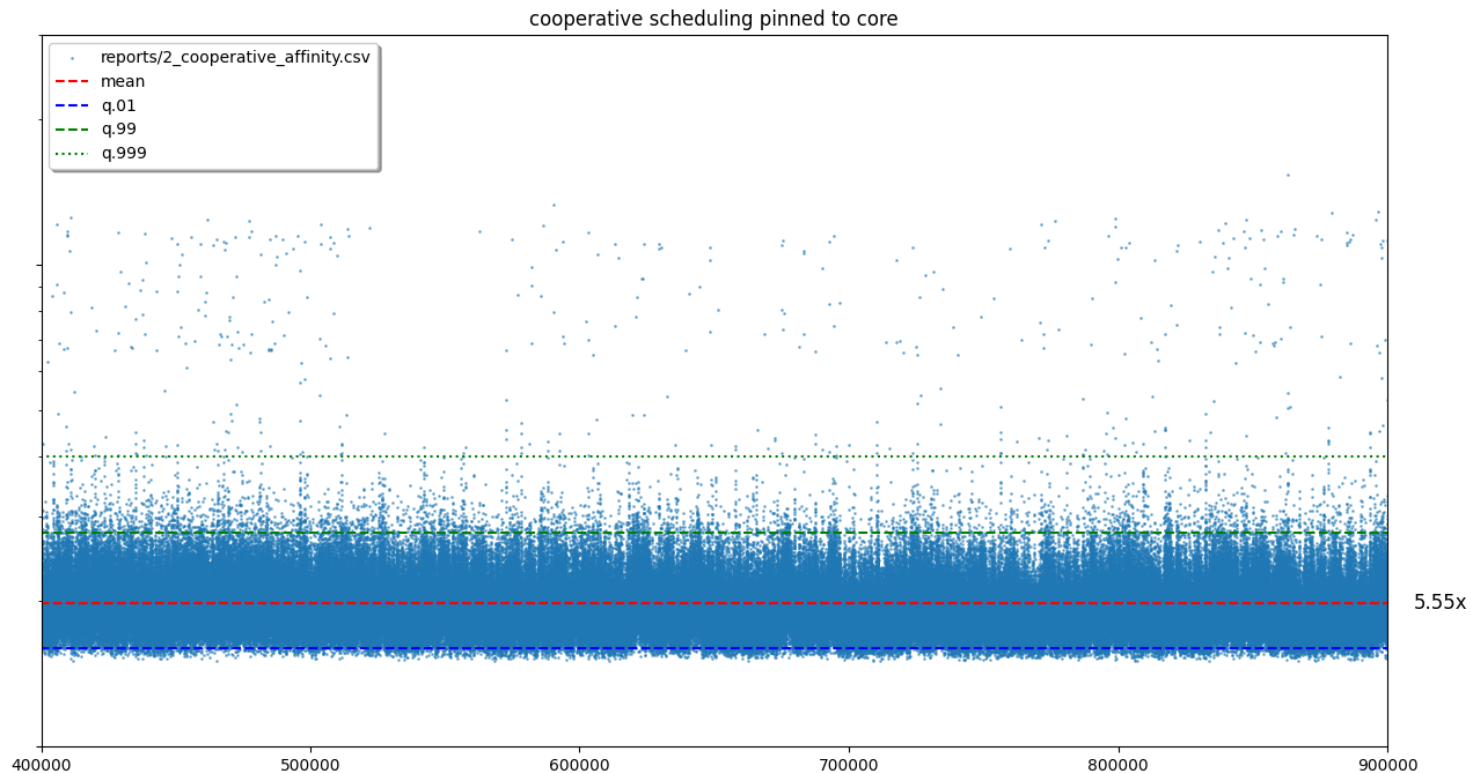
```
1  pub fn snooze(&self) {
2      if self.step.get() <= 6 {
3          for _ in 0..1 << self.step.get() {
4              ::std::hint::spin_loop(); // Busy loop
5          }
6      } else {
7          ::std::thread::yield_now(); // Cooperative scheduling
8      }
9      if self.step.get() <= 10 {
10         self.step.set(self.step.get() + 1); // => backoff.is_completed()
11     }
12 }
```

sched_yield

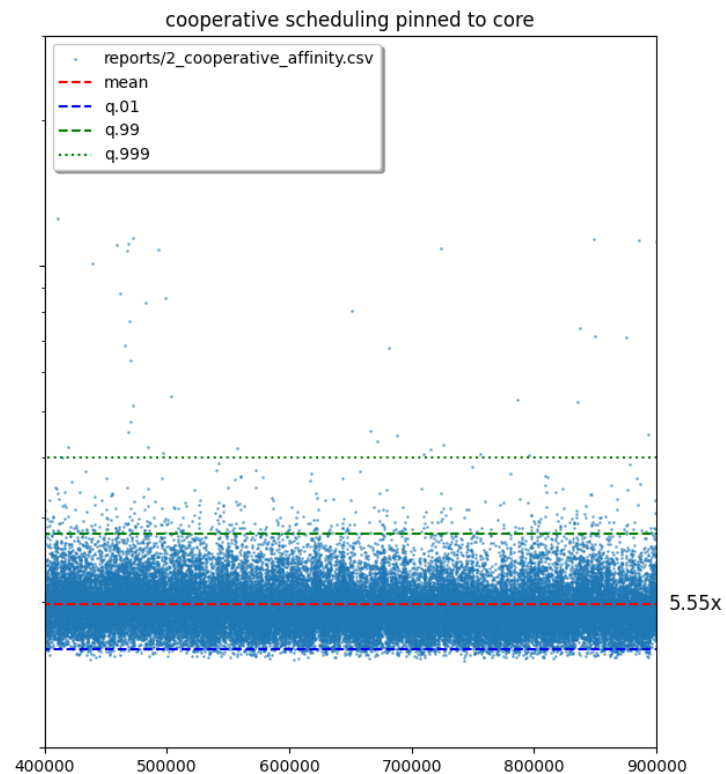
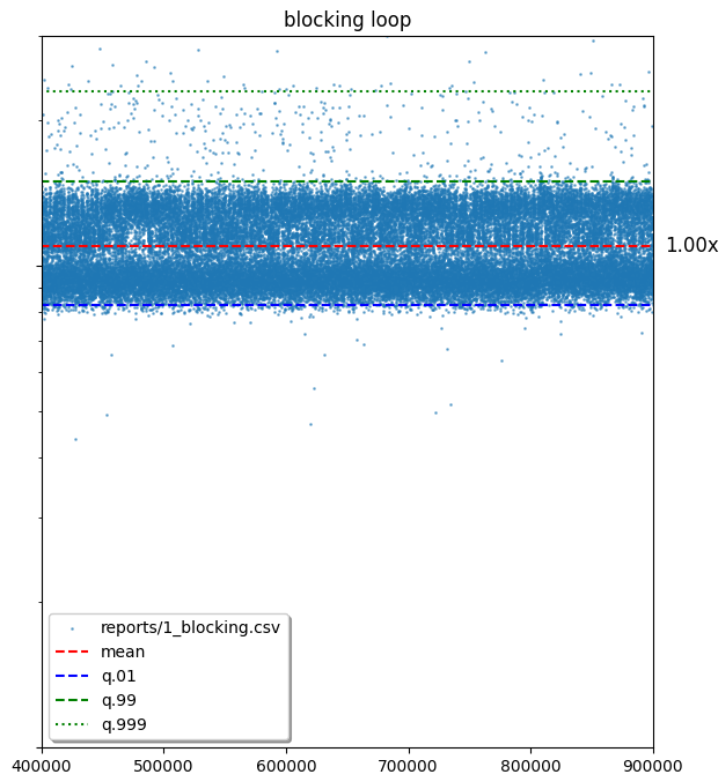
Use cooperative scheduling

```
1  let sock: std::net::UdpSocket;  
2  core_affinity::set_for_current(core_id);  
3  sock.set_nonblocking(true);  
4  loop {  
5      match sock.recv(&mut buf) {  
6          Ok(len) => handle_message(&buf[..len]),  
7          Err(err) if err.kind() == ErrorKind::WouldBlock ||  
8              err.kind() == ErrorKind::TimedOut => { }  
9          Err(err) => handle_error(err),  
10     }  
11     std::thread::yield_now();  
12 }
```

Measurement results



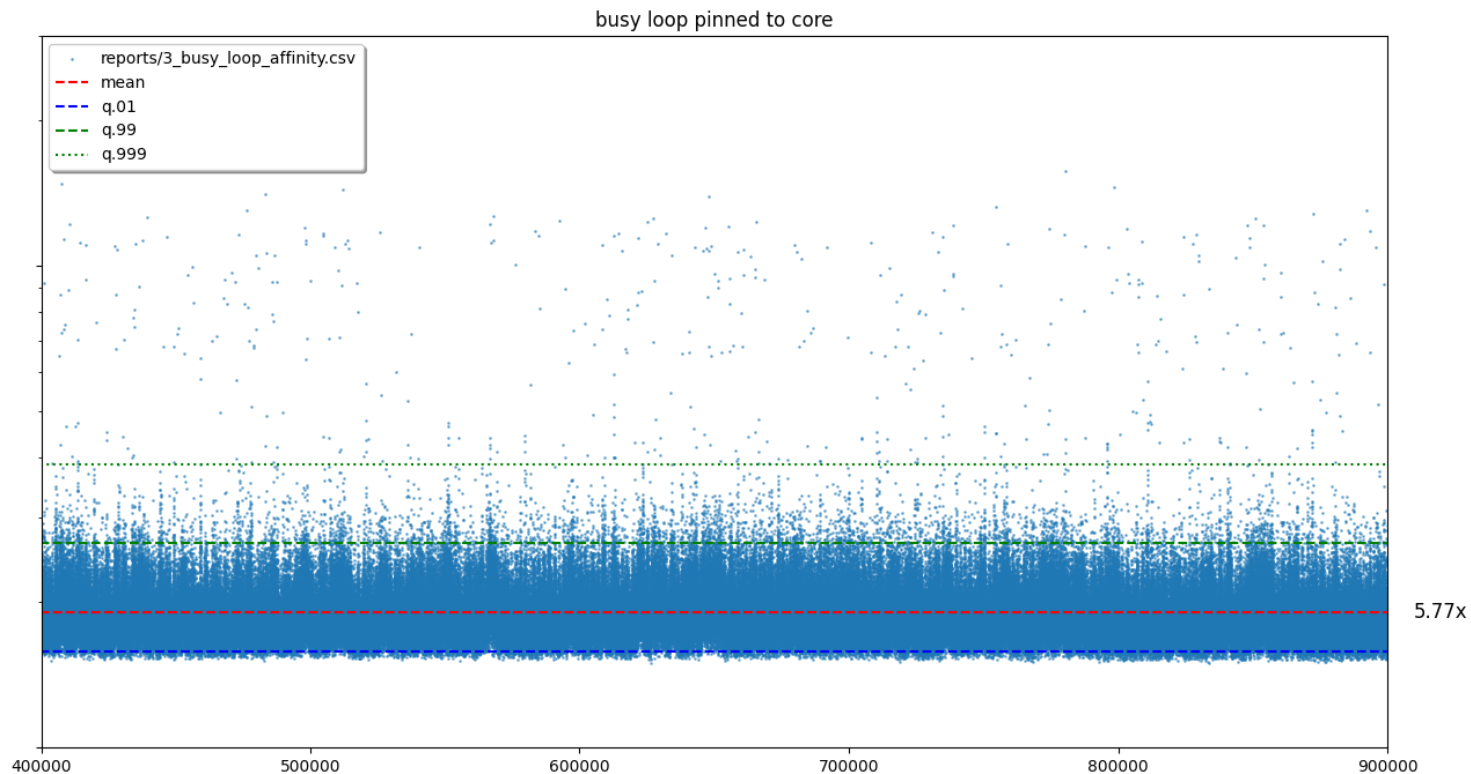
Compare with blocking



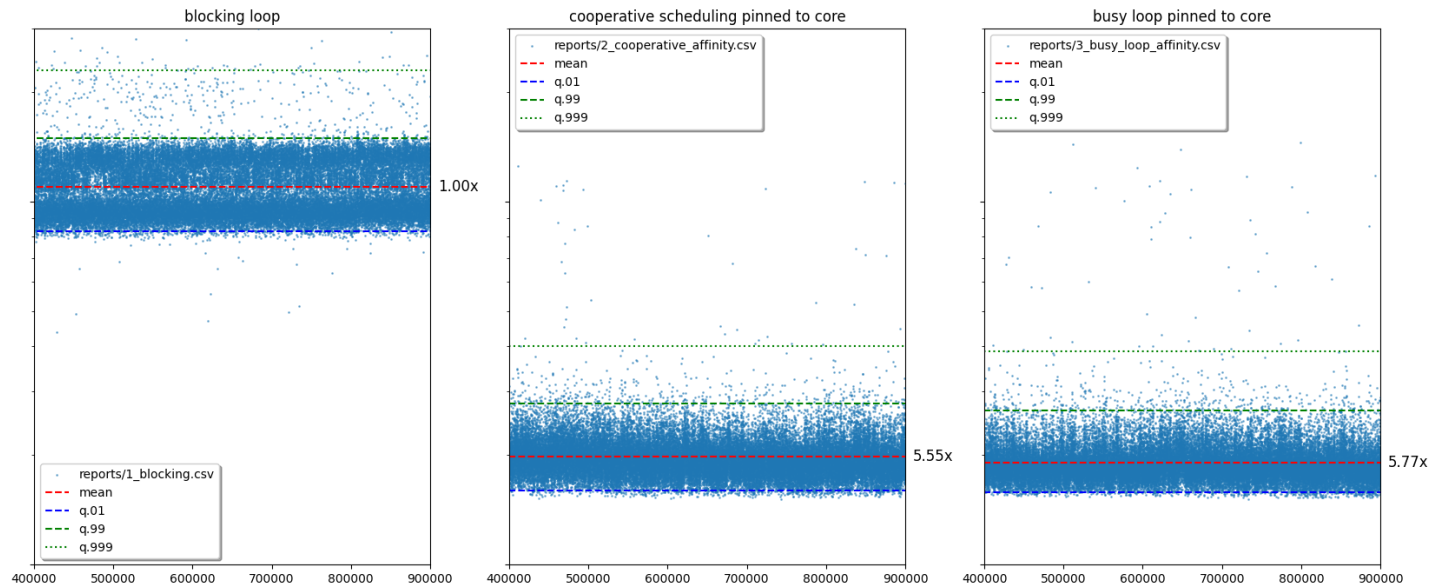
Busy loop

```
1  let sock: std::net::UdpSocket;  
2  sock.set_nonblocking(true);  
3  loop {  
4      match channel.recv(&mut buf) {  
5          Ok(len) => handle_message(&buf[..len]),  
6          Err(err) if err.kind() == ErrorKind::WouldBlock ||  
7              err.kind() == ErrorKind::TimedOut => { }  
8          Err(err) => handle_error(err),  
9      }  
10     std::hint::spin_loop();  
11 }
```

Measurement results



Compare measurement results

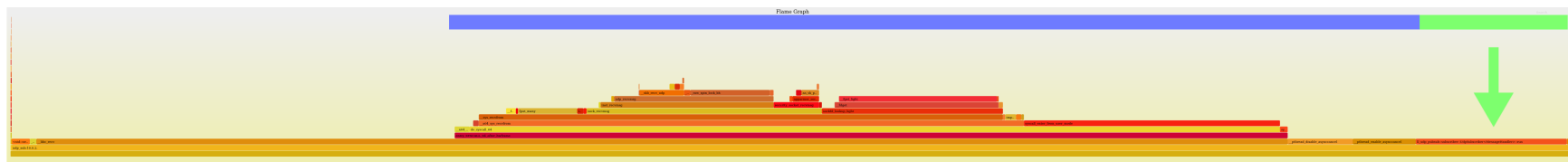


Profiling summary

```
1  $ perf stat taskset -c 1-4 bin/receive -c ${PRIVATE_IP}:3000 -n 100000 --non-blocking --core 1
2
3      106103.94 msec task-clock          #    1.000 CPUs utilized
4              253      context-switches      #    2.384 /sec
5              2        cpu-migrations        #    0.019 /sec
6              789      page-faults           #    7.436 /sec
```

PREVIOUS RESULT:

```
1      339.31 msec task-clock          #    0.003 CPUs utilized
2     100329      context-switches      #   295.683 K/sec
3              2        cpu-migrations        #    5.894 /sec
4              787      page-faults           #    2.319 K/sec
```



ll-udp-pubsub

- Generic statically linked message type via serde
- Publisher maintains list of subscriptions
- Subscriptions expire after 60 seconds
- Subscriber actively maintains subscriptions

Want to know more?

- [What Every Programmer Should Know About Memory | Ulrich Drepper | Red Hat Inc](#)



- [How GPU Computing works | Stephen Jones | nVidia](#)



- Q & A