Building low latency networking channel

Maxim Vorobjov

Volition Technologies

About me

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

Teaser

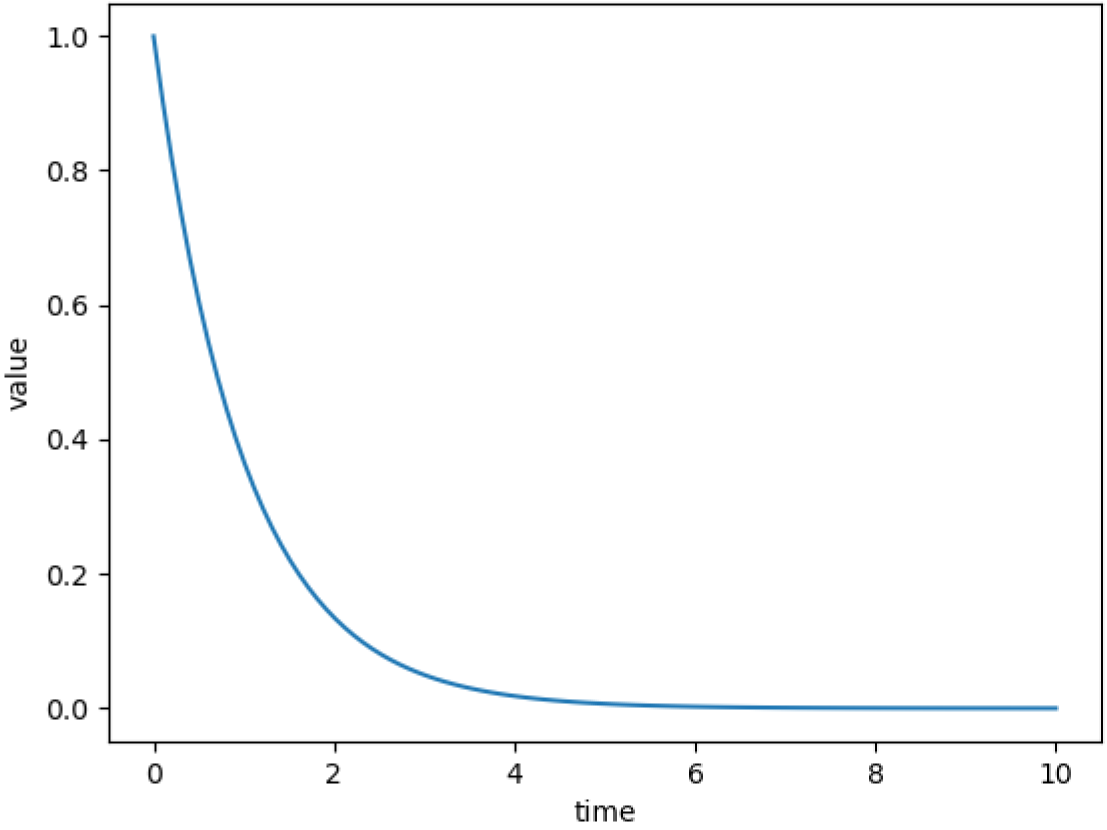
- What is low latency? 🏎️
- What problems are we trying to solve? 🐉
- Deep dive to microseconds level 🐠
- Discover CPU cache and system scheduler impact 🔬
- 🦀

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?



Low latency Applications

- HFT
- IoT
- Video Streaming
- Other realtime applications

Nothing comes for free

> except zero cost abstractions

Assumptions

- Sequence of events 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 ~100us-1ms

UDP contraints

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

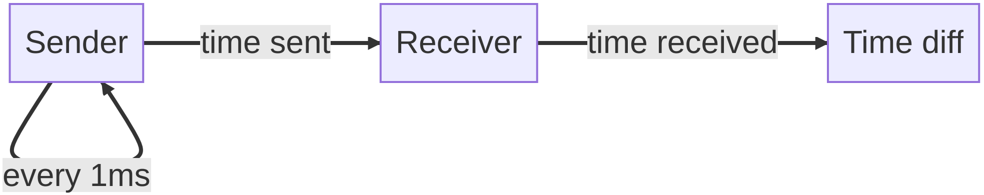
Implementation

Traditional event-loop

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

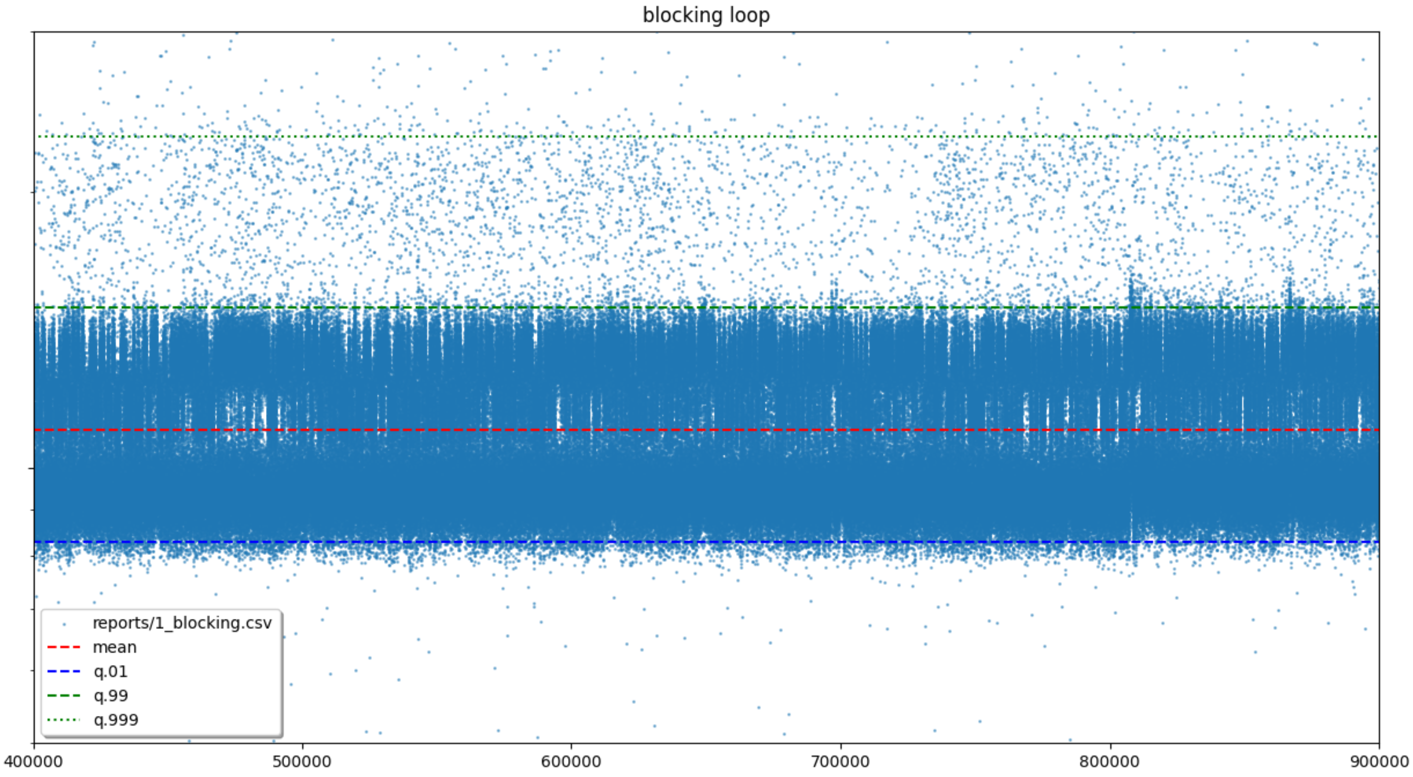
bincode

Measurement method




```
$ bin/receive -c ${PRIVATE_IP}:3000 -n 100000 > 1_blocking.csv &

$ bin/send -c ${PRIVATE_IP}:3000 -s ${PRIVATE_IP}:3001 -t 1000 -n 1000000
```

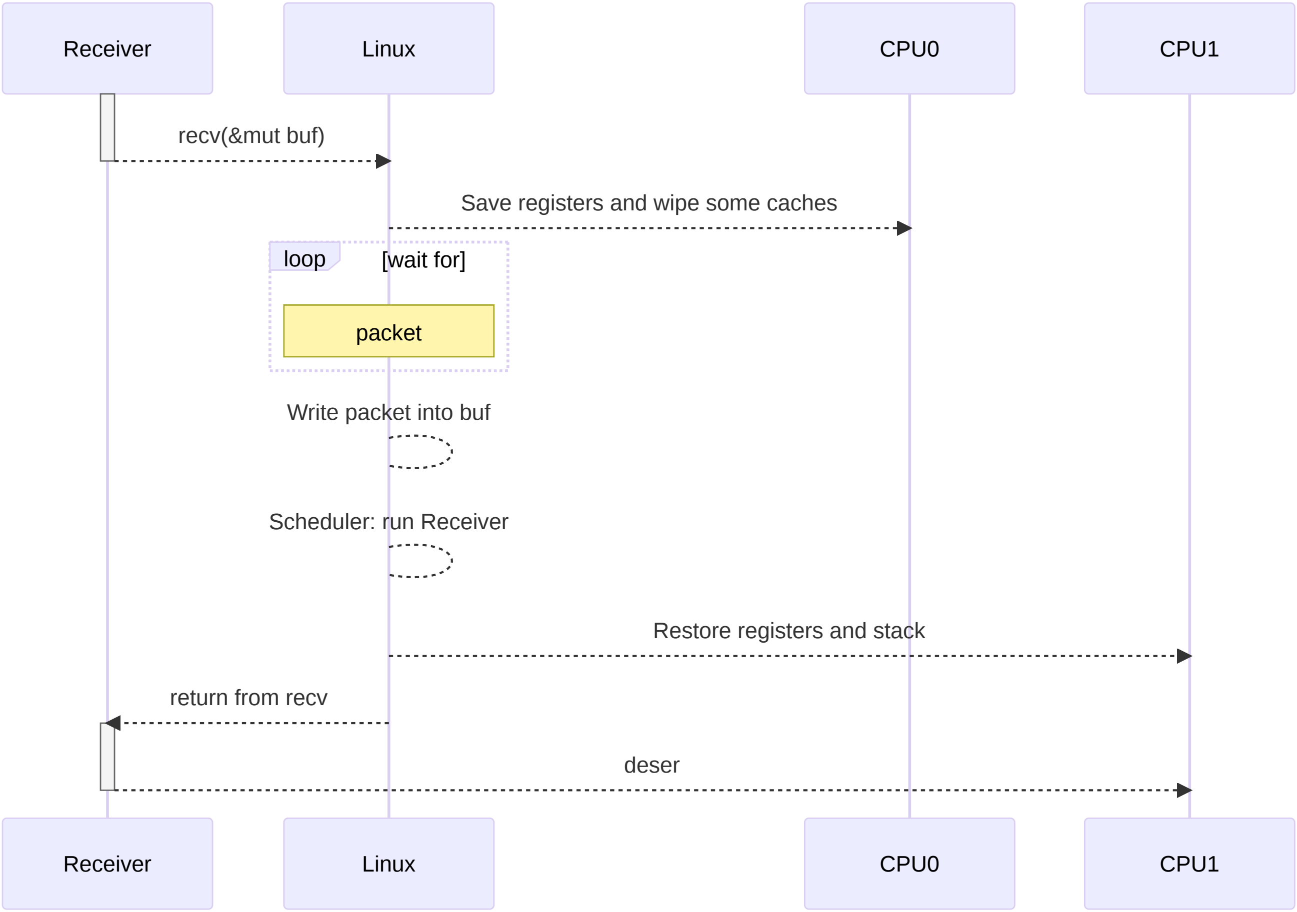
Measurement results



Profile

```
$ perf stat bin/receive -c ${PRIVATE_IP}:3000 -n 100000

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

Context switching



Let’s pin our receiver to CPU Core

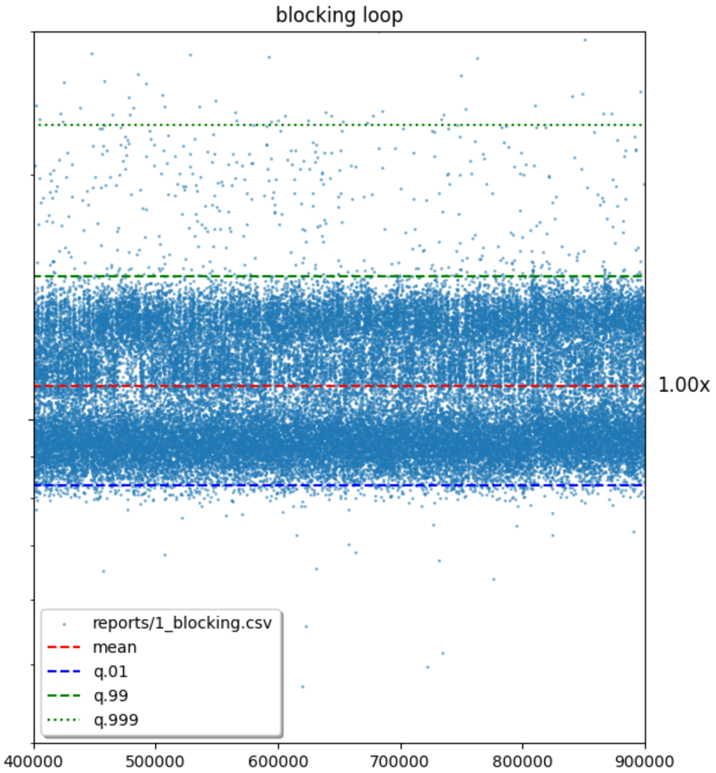
```
let channel: std::net::UdpSocket;
core_affinity::set_for_current(core_id);
loop {
    match channel.recv(&mut buf) /* .await */ {
        Ok(len) => handle_message(&buf[..len]),
        Err(err) => handle_error(err),
    }
}
```

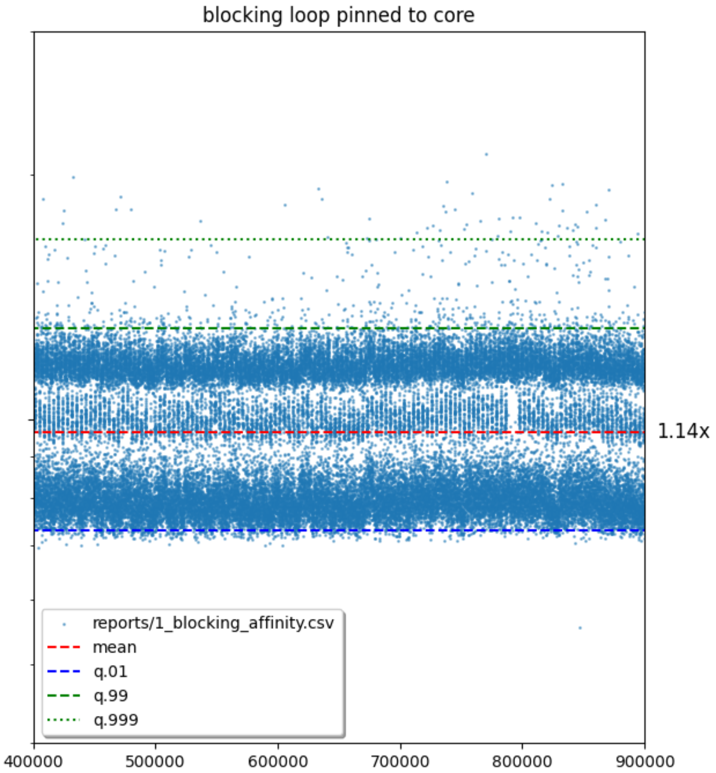
core_affinity

libc::sched_setaffinity

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

Measure and compare





crossbeam_channel::array

```rust
pub(crate) fn recv(&self, deadline: Option<Instant>) -> Result<T, RecvTimeoutError> {
    ...
    loop {
        if self.start_recv(token) {
            return unsafe { self.read(token) };
        }
        if backoff.is_completed() {
            break;
        } else {
            backoff.snooze();
        }
    }
    ...
    // Block the current thread.
    let sel = cx.wait_until(deadline);
    ...
}
```

crossbeam_utils::backoff

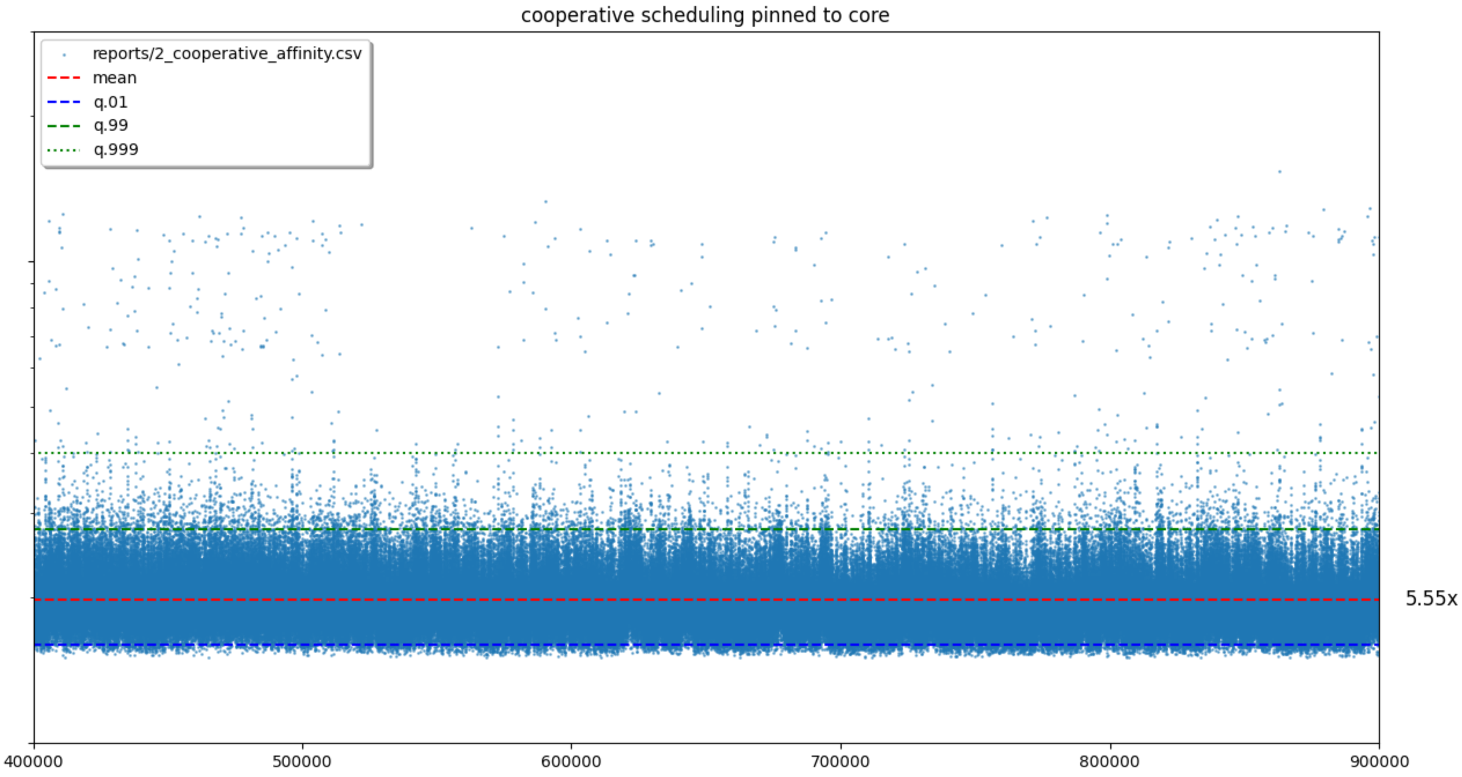
```rust
pub fn snooze(&self) {
    if self.step.get() <= 6 {
        for _ in 0..1 << self.step.get() {
            ::std::hint::spin_loop(); // Busy loop
        }
    } else {
        ::std::thread::yield_now(); // Cooperative scheduling
    }
    if self.step.get() <= 10 {
        self.step.set(self.step.get()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

sched_yield

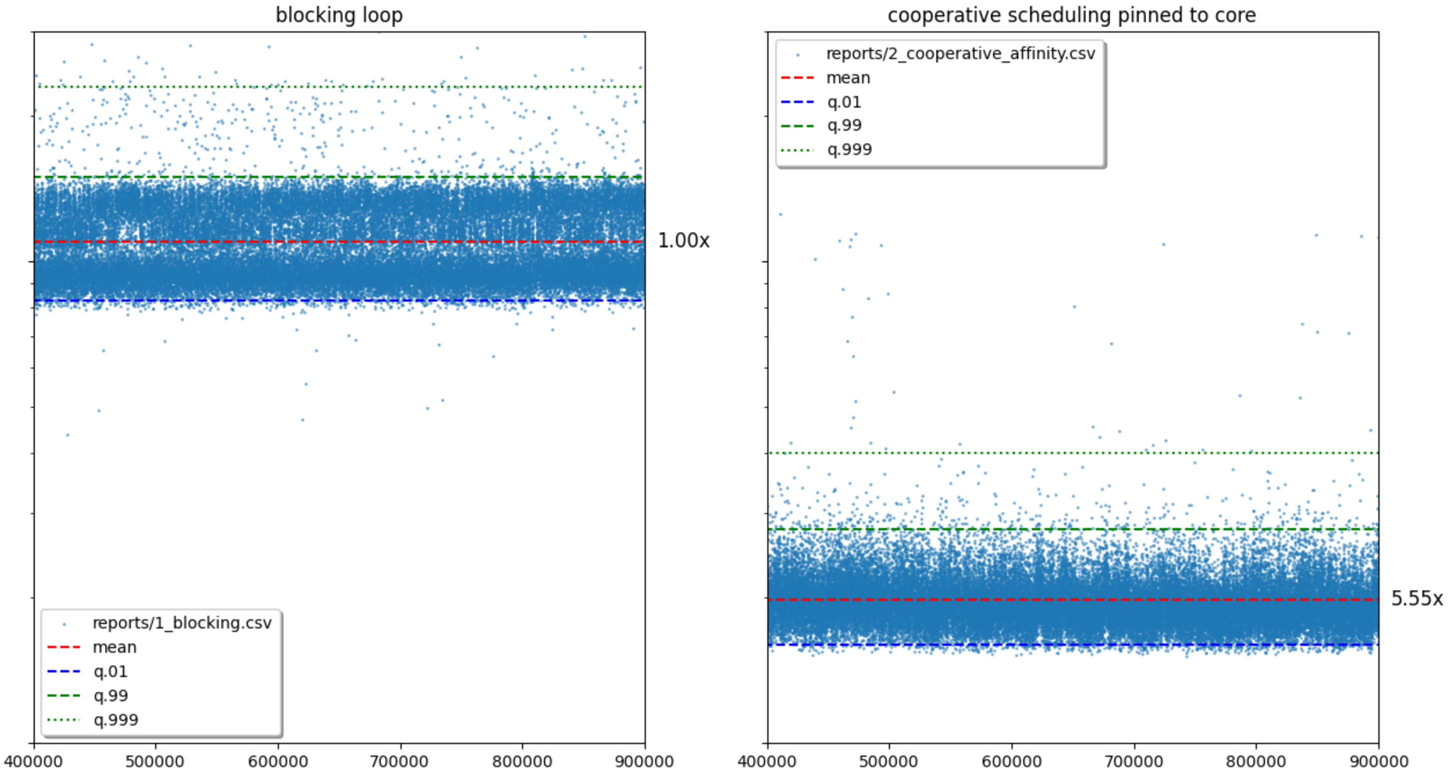
Use cooperative scheduling

```
let sock: std::net::UdpSocket;
core_affinity::set_for_current(core_id);
sock.set_nonblocking(true);
loop {
    match sock.recv(&mut buf) {
        Ok(len) => handle_message(&buf[..len]),
        Err(err) if err.kind() == ErrorKind::WouldBlock ||
            err.kind() == ErrorKind::TimedOut => { }
        Err(err) => handle_error(err),
    }
    std::thread::yield_now();
}
```

Measurement results



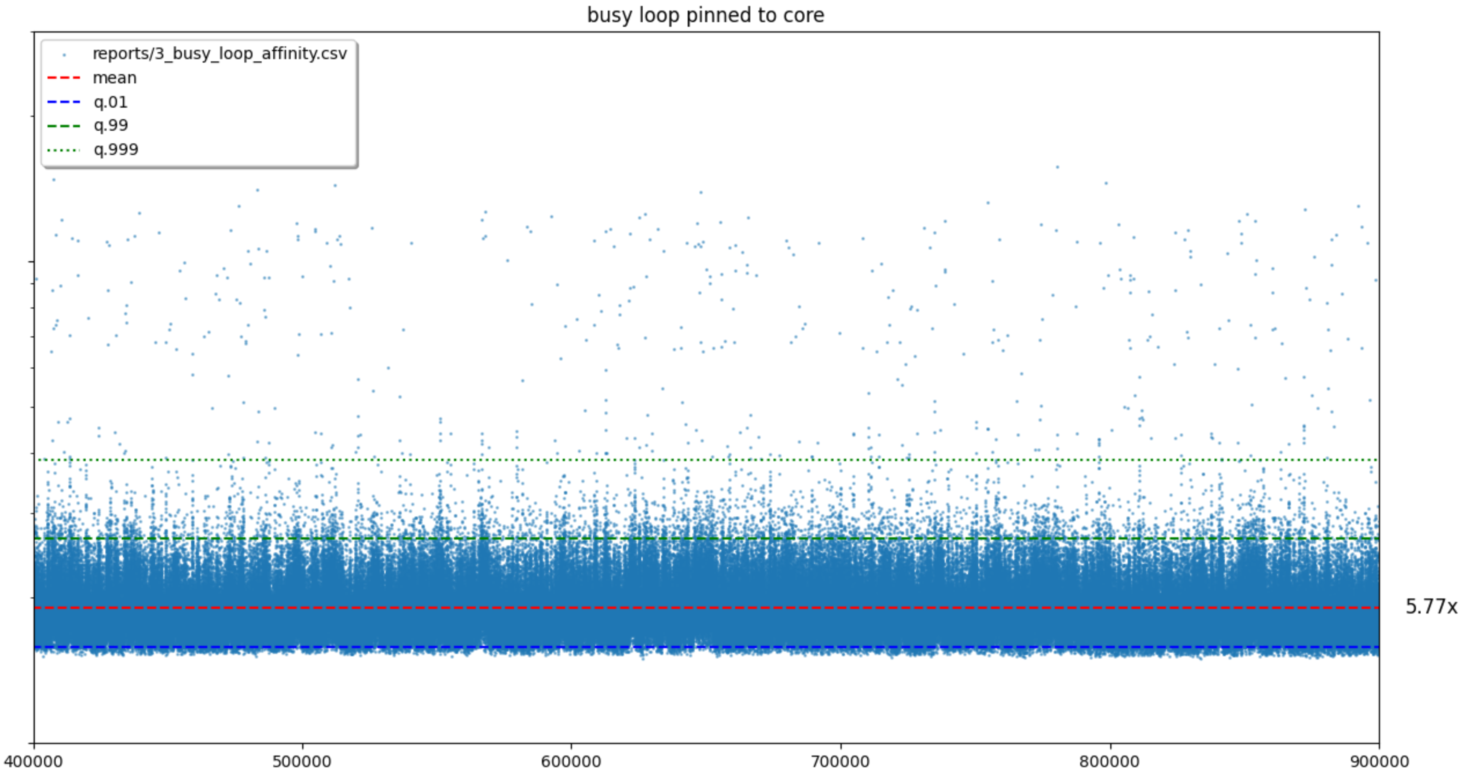
Compare with blocking



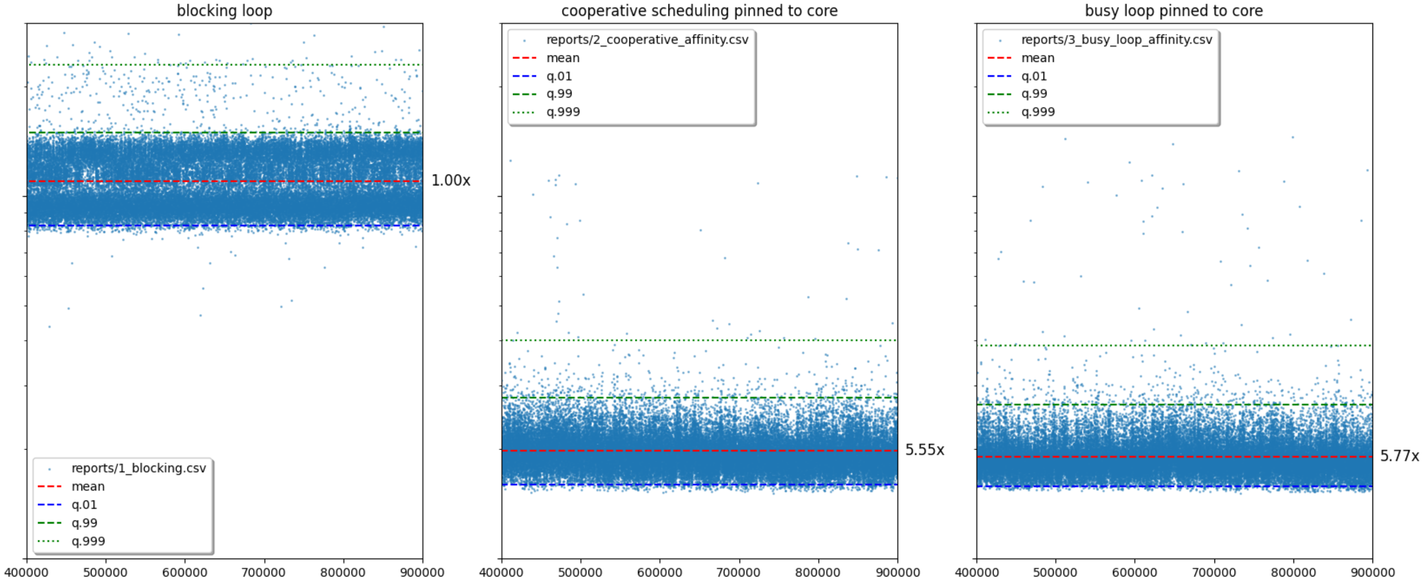
Busy loop

```
let sock: std::net::UdpSocket;
sock.set_nonblocking(true);
loop {
    match channel.recv(&mut buf) {
        Ok(len) => handle_message(&buf[..len]),
        Err(err) if err.kind() == ErrorKind::WouldBlock ||
            err.kind() == ErrorKind::TimedOut => { }
        Err(err) => handle_error(err),
    }
    for i in 0..128 {
        std::hint::spin_loop();
    }
}
```

Measurement results



Compare measurement results



Profiling summary

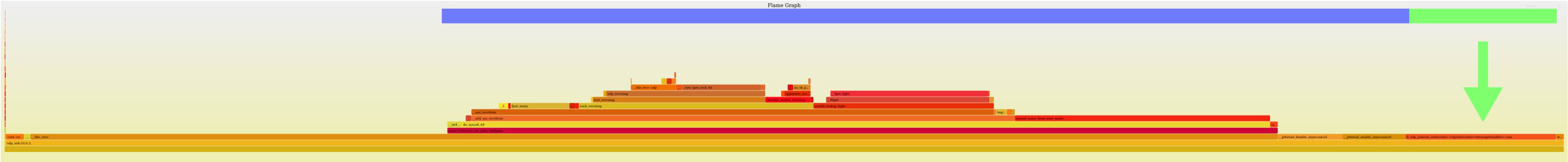
```
$ perf stat taskset -c 1-4 bin/receive -c ${PRIVATE_IP}:3000 -n 100000 --non-blocking --core 1

        106103.94 msec task-clock                #    1.000 CPUs utilized
              253      context-switches          #    2.384 /sec
                2      cpu-migrations            #    0.019 /sec
              789      page-faults               #    7.436 /sec
```

PREVIOUS RESULT:

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

Profiling flame chart

- green area - time within busy loop handlers
- blue area - time within linux kernel recv (syscall)



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