CS6501-003: Datacenter Infrastructure

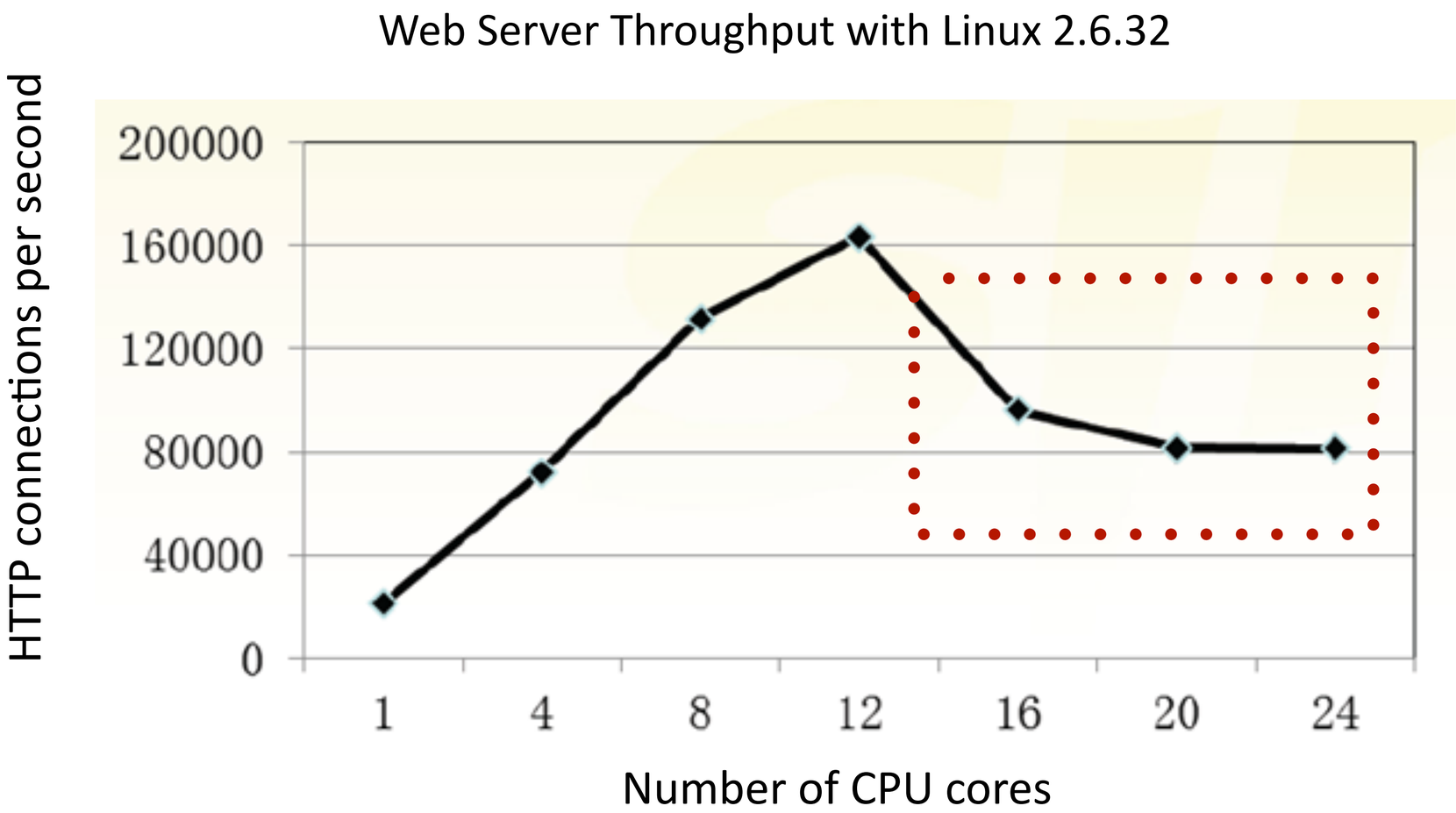
- How to profile your systems

Qizhe Cai

The importance of profiling systems

- Examples of research closely related to profiling
- Profiling tools that we can use today:
  - htop — interactive system view
    - Quick look at hot processes/threads, CPU and memory pressure, load distribution
  - lstopo — hardware/NUMA topology
    - See sockets, cores, caches, and NUMA distances; map threads to cores
  - perf — CPU sampling and stats
    - Hot functions, call stacks, cycles, cache misses; good for flamegraphs
  - eBPF —low-overhead kernel tracing
    - Trace syscalls, TCP events, scheduler latencies; per-flow attribution

Example 1: Fastsocket (ASPLOS 16)

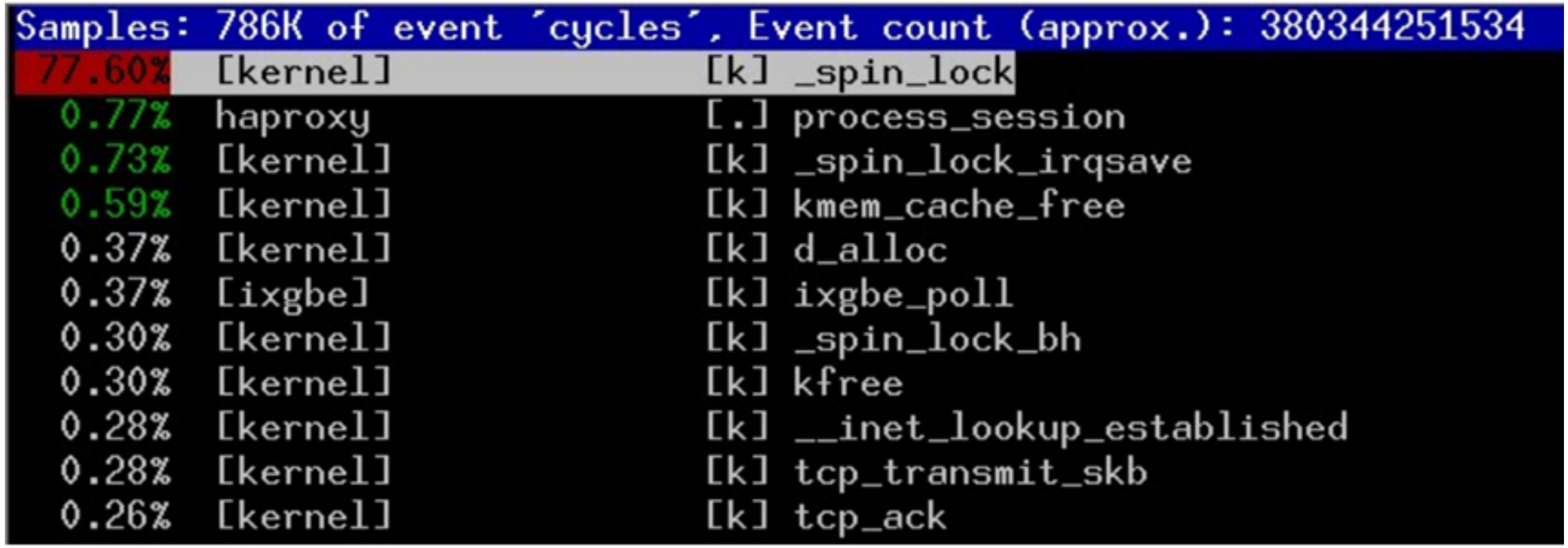


- The figure shows HTTP connection per second (throughput) with increasing number of CPU cores

| The throughput decreases after #CPU cores >= 12

Example 1: Fastsocket: A network socket (ASPLOS 16)

CPU Profiling



```
Samples: 786K of event 'cycles', Event count (approx.): 380344251534
 77.60%  [kernel]                [k] _spin_lock
  0.77%  haproxy                 [.] process_session
  0.73%  [kernel]                [k] _spin_lock_irqsave
  0.59%  [kernel]                [k] kmem_cache_free
  0.37%  [kernel]                [k] d_alloc
  0.37%  [ixgbe]                 [k] ixgbe_poll
  0.30%  [kernel]                [k] _spin_lock_bh
  0.30%  [kernel]                [k] kfree
  0.28%  [kernel]                [k] __inet_lookup_established
  0.28%  [kernel]                [k] tcp_transmit_skb
  0.26%  [kernel]                [k] tcp_ack
```

- Almost 80% CPU time is spent on locking!

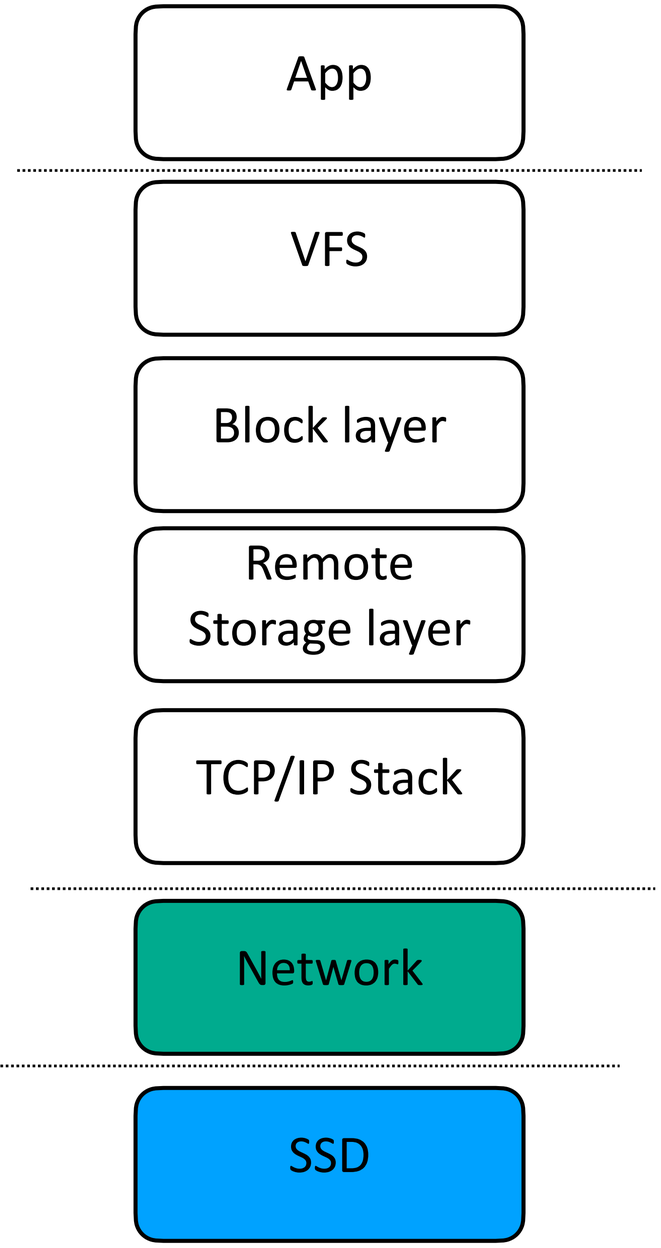
| Multiple cores are accessing global objects and critical sections!

Example 2: i10 (NSDI'20)

Linux CPU overheads

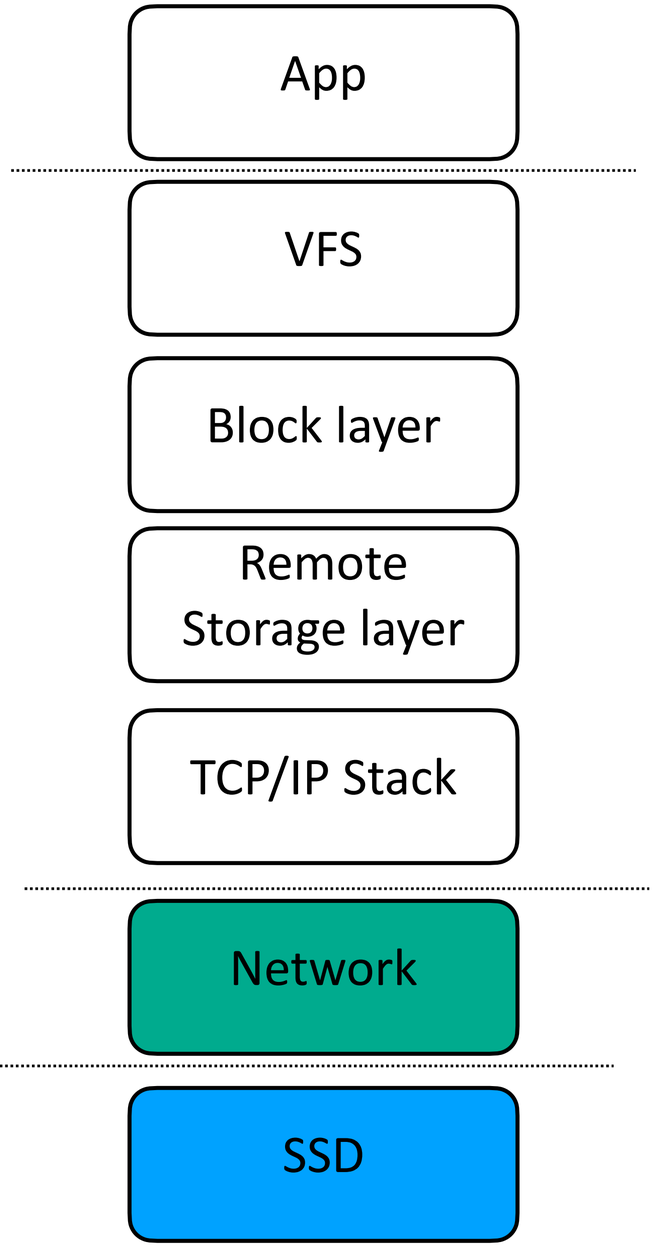
```
Samples: 821  of event 'cpu-clock:pppH', 4000 Hz, Event count (approx.)
Overhead  Shared Object          Symbol
   7.36%  [kernel]               [k] kallsyms_expand_symbol.constprop.0
   6.03%  [kernel]               [k] vsnprintf
   4.82%  [kernel]               [k] number
   4.46%  [kernel]               [k] format_decode
   3.86%  perf                   [.] 0x00000000000e0997
   3.62%  [kernel]               [k] string
   3.26%  perf                   [.] 0x00000000000ea44d
   2.65%  libc.so.6              [.] __strchr_avx2
   2.41%  [kernel]               [k] memcpy_erms
   1.93%  [kernel]               [k] do_user_addr_fault
   1.93%  libc.so.6              [.] __strcmp_avx2
   1.93%  perf                   [.] 0x00000000000ec805
   1.57%  [kernel]               [k] s_show
   1.57%  libc.so.6              [.] __libc_calloc
   1.57%  libc.so.6              [.] sysmalloc
   1.57%  perf                   [.] 0x00000000000e03c0
   1.33%  [kernel]               [k] clear_page_erms
   1.33%  [kernel]               [k] s_next
   1.33%  perf                   [.] 0x00000000000d43eb
   1.21%  [kernel]               [k] seq_read_iter
   1.09%  [kernel]               [k] module_get_kallsym
   1.09%  libc.so.6              [.] __mprotect
   0.96%  libc.so.6              [.] _int_malloc
   0.96%  perf                   [.] 0x00000000000d4408
   0.84%  [kernel]               [k] seq_printf
   0.84%  [kernel]               [k] update_iter
   0.84%  perf                   [.] 0x00000000000e096b
   0.74%  [kernel]               [k] _raw_spin_unlock_irqrestore
   0.72%  perf                   [.] 0x00000000000ea456
   0.72%  perf                   [.] 0x00000000000ec80f
   0.60%  [kernel]               [k] syscall_enter_from_user_mode
   0.60%  perf                   [.] 0x00000000000d4225
   0.60%  perf                   [.] 0x00000000000d422e
   0.60%  perf                   [.] 0x00000000000d4413
```

- CPU profiling seems messy



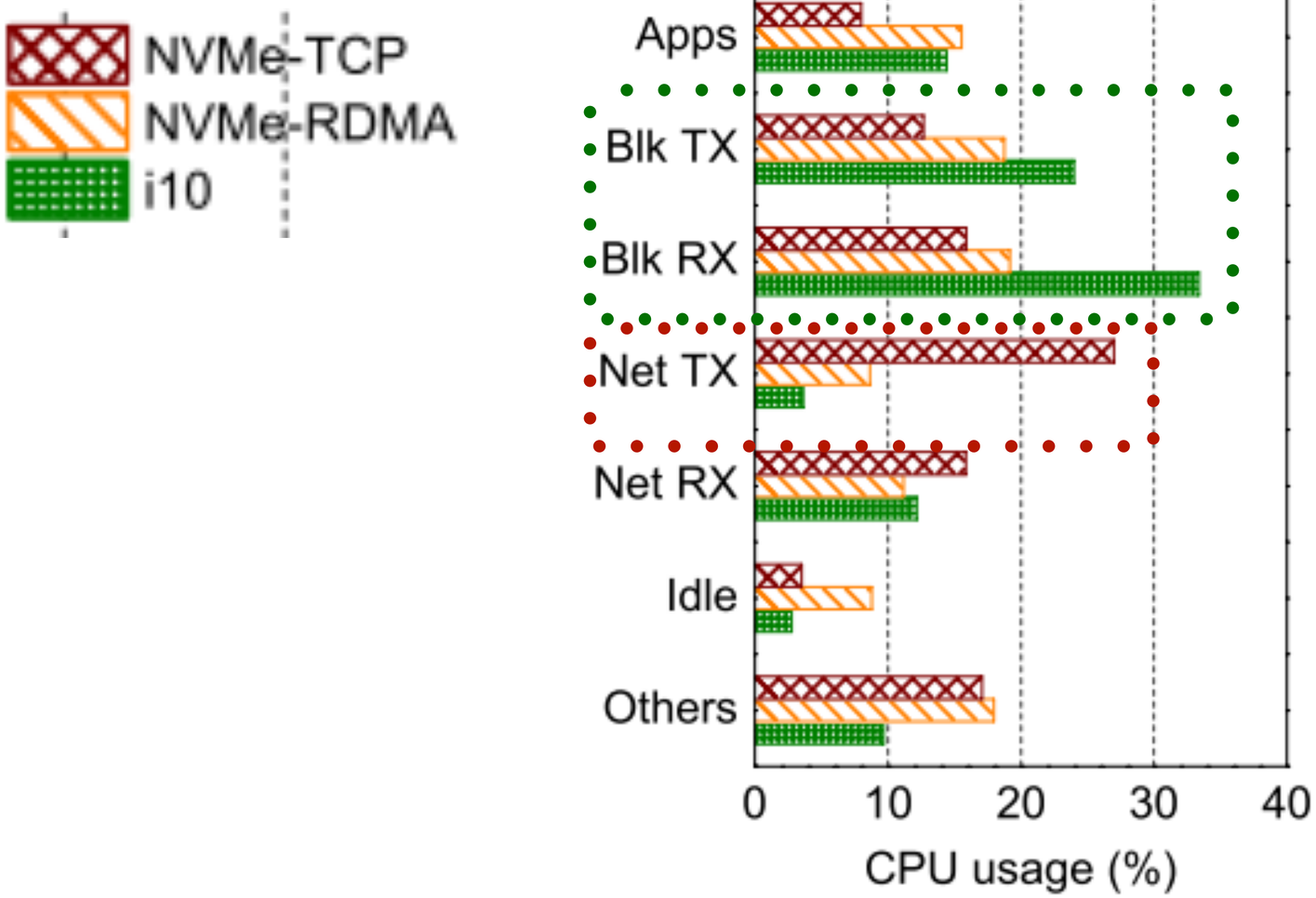
Example 2: i10: remote storage stack (NSDI'20)

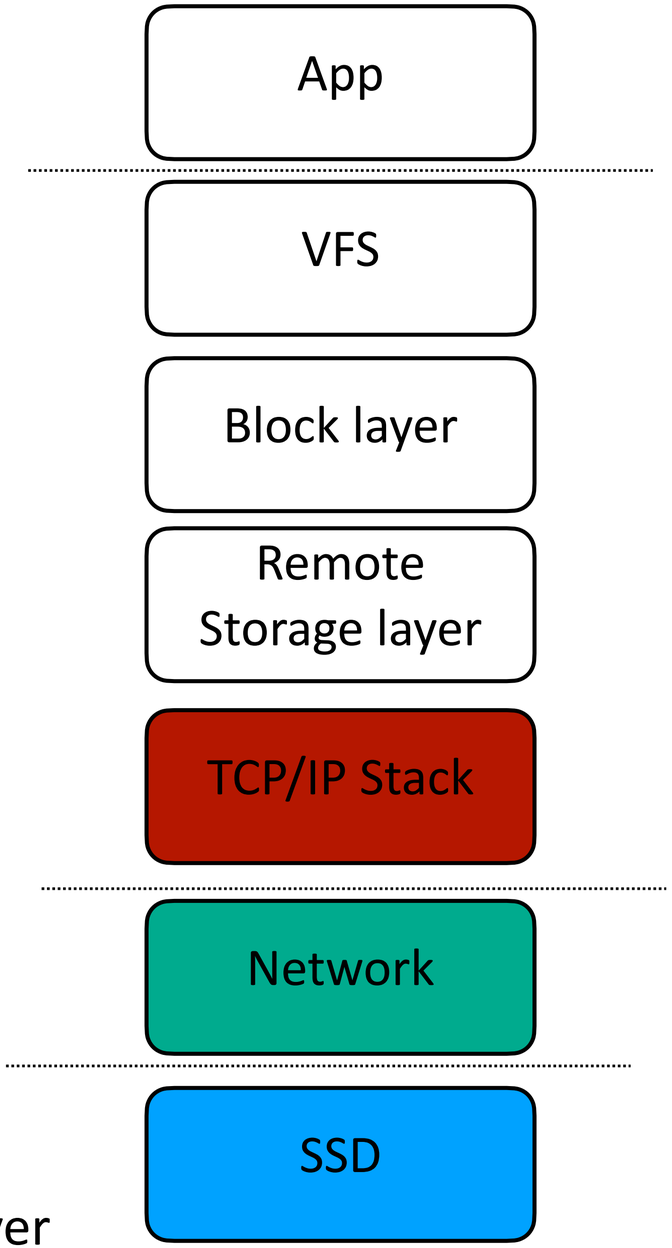
Component	Description
Applications	Submit and receive requests/responses via I/O system calls (host). Ideally, all cycles would be spent in this component.
Block TX	Process the requests at the blk-mq layer and ring the doorbells to the remote storage access layer (host) or the local storage device (target).
Block RX	Receive the requests/responses from the network Rx queues or the local storage device.
Net TX	Send the requests/responses from the I/O queues (NVMe-RDMA uses Queue Pairs in the NIC).
Net RX	Process packets and insert into the network Rx queues (by the network interrupt handler).
Idle	Enter the CPU *Idle* mode.
Others	Include all the remaining overheads such as task scheduling, IRQ handling, spin locks, and so on.



- We attribute each functions into different categories/component.

Example 2: i10 (NSDI'20)





- CPU bottleneck is mainly on Net TX for NVMe-TCP.
- I10 resolves this bottleneck and so more CPU cycles can be spent on block layer

Example 3: Understanding host network stack overheads (SIGCOMM'21)

Visualization

```
Samples: 821  of event 'cpu-clock:pppH', 4000 Hz, Event count (approx.)
Overhead  Shared Object                Symbol
   7.36%  [kernel]                     [k] kallsyms_expand_symbol.constprop.0
   6.03%  [kernel]                     [k] vsnprintf
   4.82%  [kernel]                     [k] number
   4.46%  [kernel]                     [k] format_decode
   3.86%  perf                         [.] 0x000000000000e0997
   3.62%  [kernel]                     [k] string
   3.26%  perf                         [.] 0x00000000000ea44d
   2.65%  libc.so.6                    [.] __strchr_avx2
   2.41%  [kernel]                     [k] memcpy_erms
   1.93%  [kernel]                     [k] do_user_addr_fault
   1.93%  libc.so.6                    [.] __strcmp_avx2
   1.93%  perf                         [.] 0x00000000000ec805
   1.57%  [kernel]                     [k] s_show
   1.57%  libc.so.6                    [.] __libc_calloc
   1.57%  libc.so.6                    [.] sysmalloc
   1.57%  perf                         [.] 0x00000000000e03c0
   1.33%  [kernel]                     [k] clear_page_erms
   1.33%  [kernel]                     [k] s_next
   1.33%  perf                         [.] 0x00000000000d43eb
   1.21%  [kernel]                     [k] seq_read_iter
   1.09%  [kernel]                     [k] module_get_kallsym
   1.09%  libc.so.6                    [.] __mprotect
   0.96%  libc.so.6                    [.] _int_malloc
   0.96%  perf                         [.] 0x00000000000d4408
   0.84%  [kernel]                     [k] seq_printf
   0.84%  [kernel]                     [k] update_iter
   0.84%  perf                         [.] 0x000000000000e096b
   0.74%  [kernel]                     [k] _raw_spin_unlock_irqrestore
   0.72%  perf                         [.] 0x00000000000ea456
   0.72%  perf                         [.] 0x00000000000ec80f
   0.60%  [kernel]                     [k] syscall_enter_from_user_mode
   0.60%  perf                         [.] 0x00000000000d4225
   0.60%  perf                         [.] 0x00000000000d422e
   0.60%  perf                         [.] 0x00000000000d4413
```

perf data

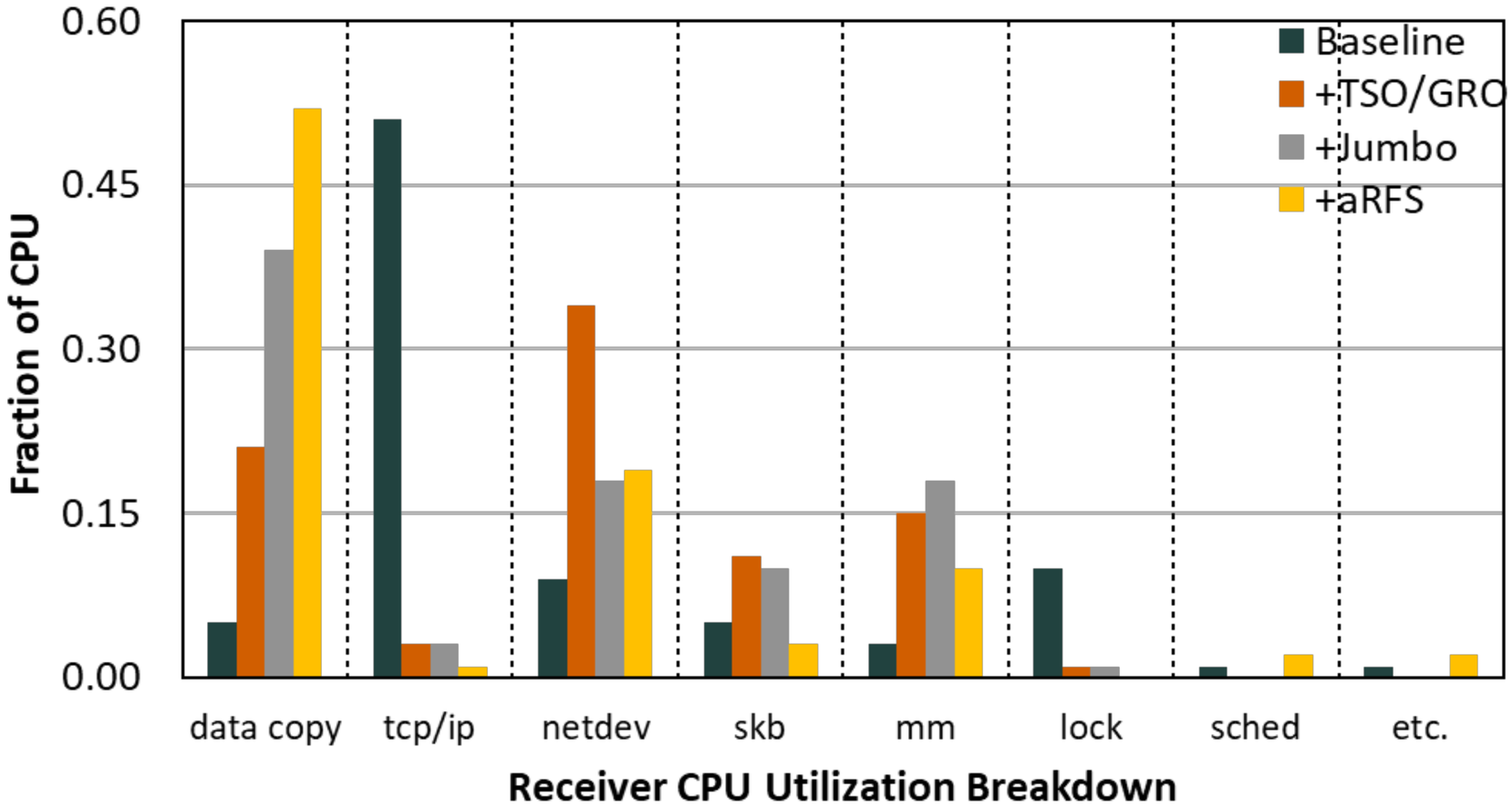
Component	Description
Data copy	From user space to kernel space, and vice versa.
TCP/IP	All the packet processing at TCP/IP layers.
Netdevice subsystem	Netdevice and NIC driver operations (*e.g.*, NAPI polling, GSO/GRO, qdisc, etc.).
skb management	Functions to build, split, and release skb.
Memory de-/alloc	skb de-/allocation and page-related operations.
Lock/unlock	Lock-related operations (*e.g.*, spin locks).
Scheduling	Scheduling/context-switching among threads.
Others	All the remaining functions (*e.g.*, IRQ handling).

Classification

```
1   PageHuge        etc
2   ___slab_alloc   mm
3   __alloc_pages_nodemask  mm
4   __alloc_skb     skb
5   __build_skb     skb
6   __build_skb_around      skb
7   __cgroup_bpf_run_filter_skb     netdev
8   __check_object_size     data_copy
9   __copy_skb_header       skb
10  __dev_queue_xmit        netdev
11  __do_softirq    etc
12  __fget  etc
13  __free_pages_ok mm
14  __get_xps_queue_idx     netdev
15  __inc_numa_state        mm
16  __inet_lookup_established       tcp/ip
17  __ip_finish_output      tcp/ip
18  __ip_local_out  tcp/ip
19  __ip_queue_xmit tcp/ip
20  __kfree_skb     skb
21  __kmalloc_node_track_caller     mm
22  __kmalloc_reserve       mm
23  __ksize mm
24  __libc_recv     etc
25  __local_bh_enable_ip    sched
26  __lock_text_start       lock
27  __memcpy        data_copy
28  __mod_zone_page_state   mm
29  __napi_alloc_skb        netdev
30  __napi_schedule netdev
```

Symbol mapping*

Example 3: Understanding host network stack overheads (SIGCOMM'21)



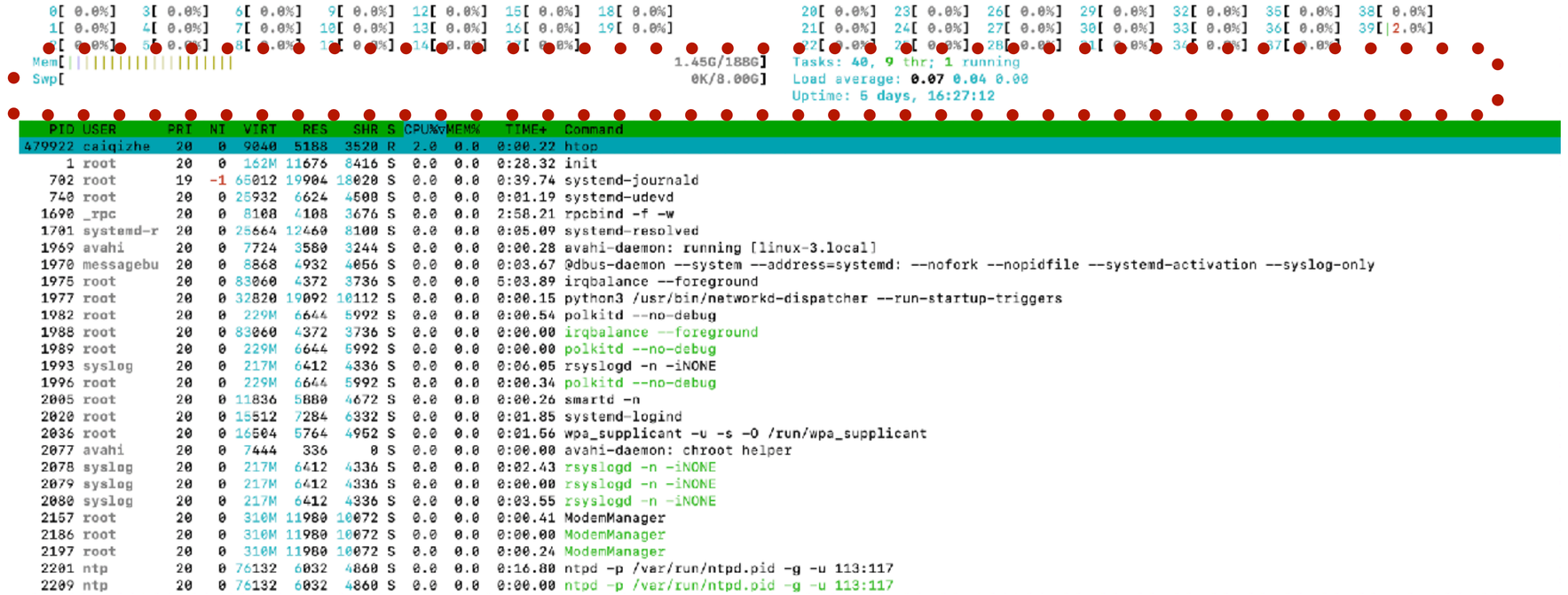
- Understand the overheads of network stacks and how different optimizations resolve bottlenecks
- We'll read this paper later this semester

htop

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
479922	caiqizhe	20	0	9040	5188	3520	R	2.0	0.0	0:00.22	htop
1	root	20	0	162M	11676	8416	S	0.0	0.0	0:28.32	init
702	root	19	-1	65012	19904	18020	S	0.0	0.0	0:39.74	systemd-journald
740	root	20	0	25932	6624	4508	S	0.0	0.0	0:01.19	systemd-udevd
1690	_rpc	20	0	8108	4108	3676	S	0.0	0.0	2:58.21	rpcbind -f -w
1701	systemd-r	20	0	25664	12460	8100	S	0.0	0.0	0:05.09	systemd-resolved
1969	avahi	20	0	7724	3580	3244	S	0.0	0.0	0:00.28	avahi-daemon: running [linux-3.local]
1970	messagebu	20	0	8868	4932	4056	S	0.0	0.0	0:03.67	@dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
1975	root	20	0	83060	4372	3736	S	0.0	0.0	5:03.89	irqbalance --foreground
1977	root	20	0	32820	19092	10112	S	0.0	0.0	0:00.15	python3 /usr/bin/networkd-dispatcher --run-startup-triggers
1982	root	20	0	229M	6644	5992	S	0.0	0.0	0:00.54	polkitd --no-debug
1988	root	20	0	83060	4372	3736	S	0.0	0.0	0:00.00	irqbalance --foreground
1989	root	20	0	229M	6644	5992	S	0.0	0.0	0:00.00	polkitd --no-debug
1993	syslog	20	0	217M	6412	4336	S	0.0	0.0	0:06.05	rsyslogd -n -iNONE
1996	root	20	0	229M	6644	5992	S	0.0	0.0	0:00.34	polkitd --no-debug
2005	root	20	0	11836	5880	4672	S	0.0	0.0	0:00.26	smartd -n
2020	root	20	0	15512	7284	6332	S	0.0	0.0	0:01.85	systemd-logind
2036	root	20	0	16504	5764	4952	S	0.0	0.0	0:01.56	wpa_supplicant -u -s -O /run/wpa_supplicant
2077	avahi	20	0	7444	336	0	S	0.0	0.0	0:00.00	avahi-daemon: chroot helper
2078	syslog	20	0	217M	6412	4336	S	0.0	0.0	0:02.43	rsyslogd -n -iNONE
2079	syslog	20	0	217M	6412	4336	S	0.0	0.0	0:00.00	rsyslogd -n -iNONE
2080	syslog	20	0	217M	6412	4336	S	0.0	0.0	0:03.55	rsyslogd -n -iNONE
2157	root	20	0	310M	11980	10072	S	0.0	0.0	0:00.41	ModemManager
2186	root	20	0	310M	11980	10072	S	0.0	0.0	0:00.00	ModemManager
2197	root	20	0	310M	11980	10072	S	0.0	0.0	0:00.24	ModemManager
2201	ntp	20	0	76132	6032	4860	S	0.0	0.0	0:16.80	ntpd -p /var/run/ntpd.pid -g -u 113:117
2209	ntp	20	0	76132	6032	4860	S	0.0	0.0	0:00.00	ntpd -p /var/run/ntpd.pid -g -u 113:117

- Get CPU usages of your servers

htop

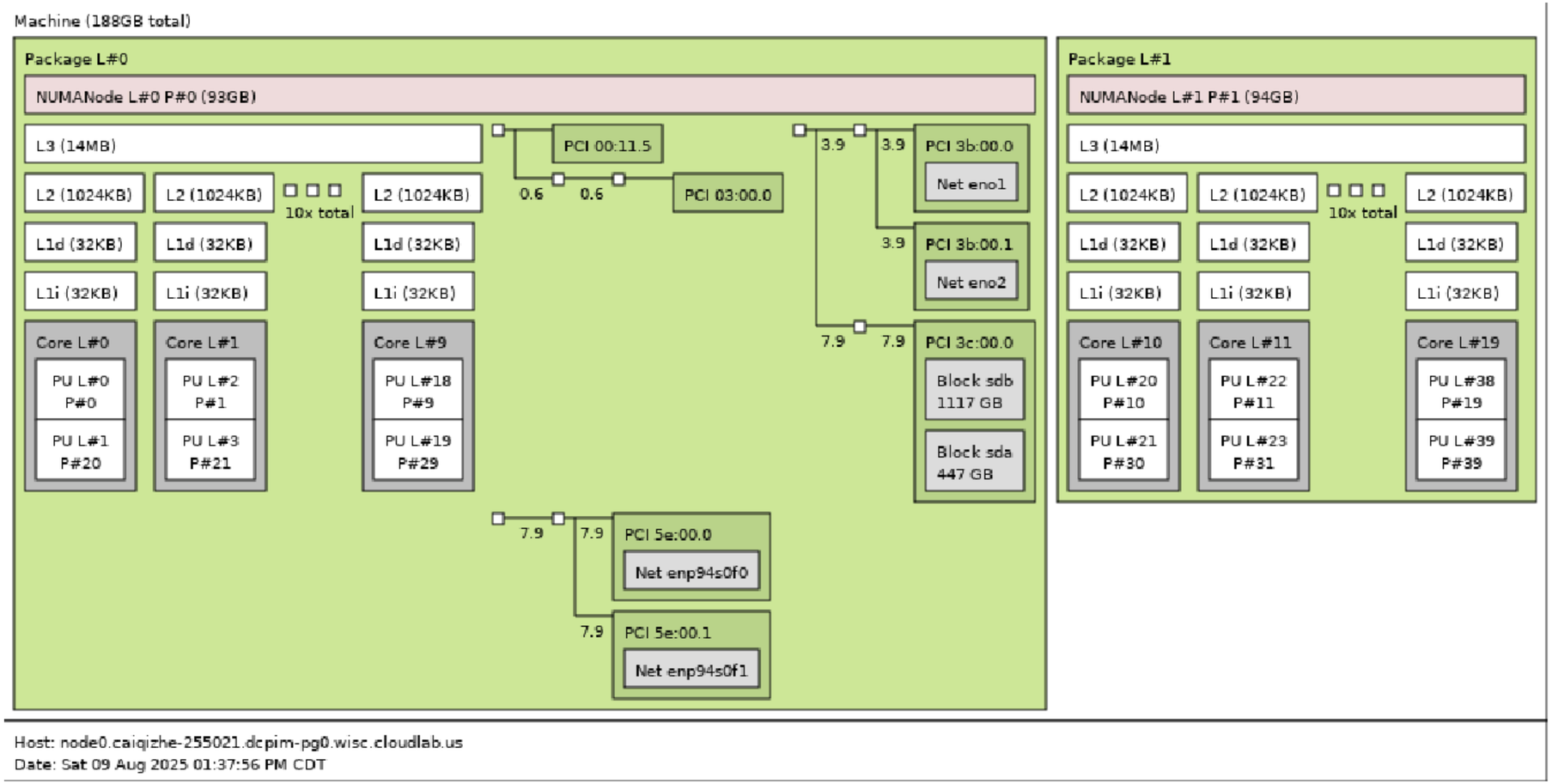


- Get CPU usages of your servers
- Get memory usages of the servers

htop

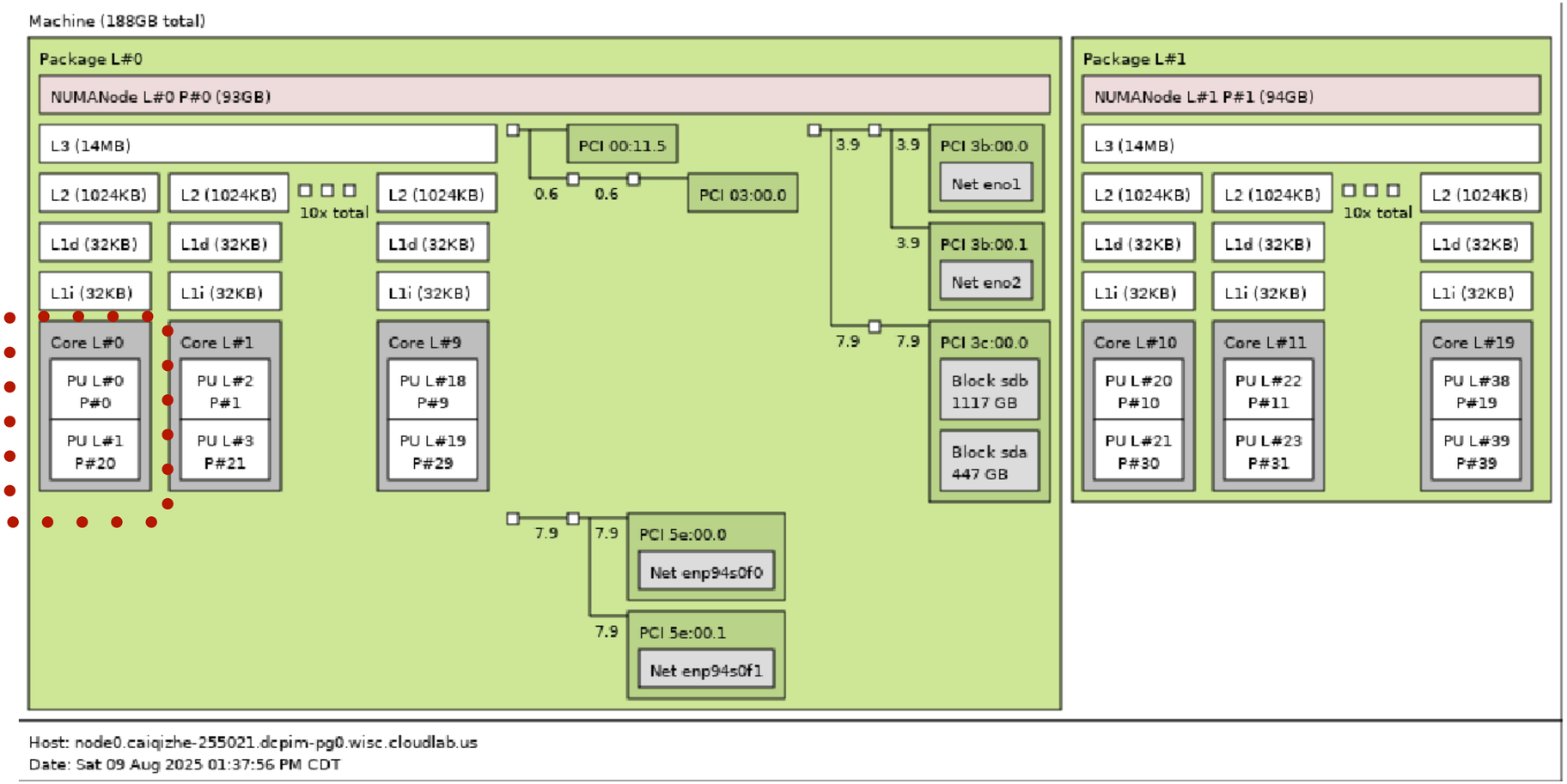
- Get CPU usages of your servers
- Get memory usages of the servers
- See per-application resource usages

lstopo



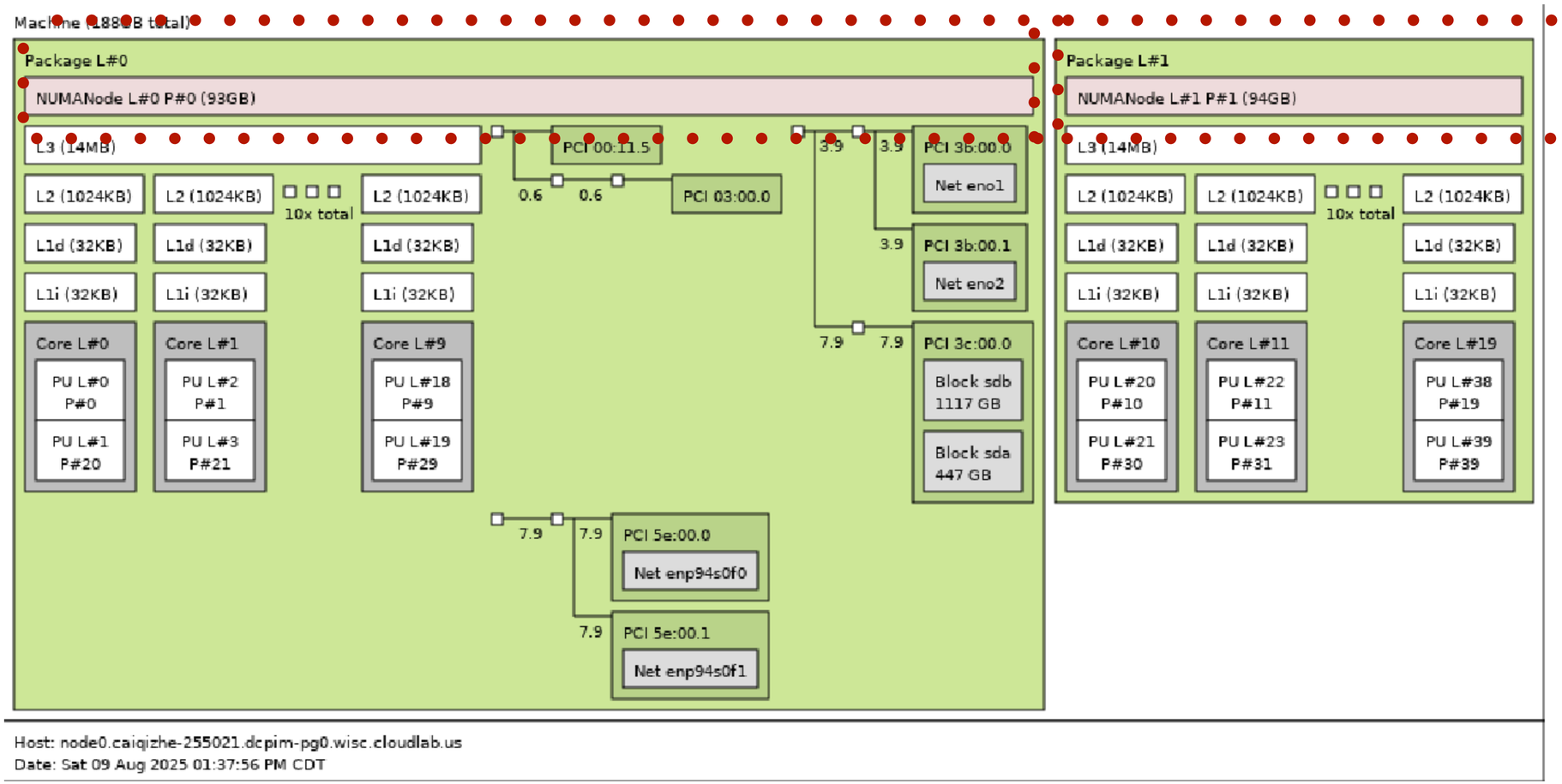
- Check server’s hardware topologies/configuration

lstopo



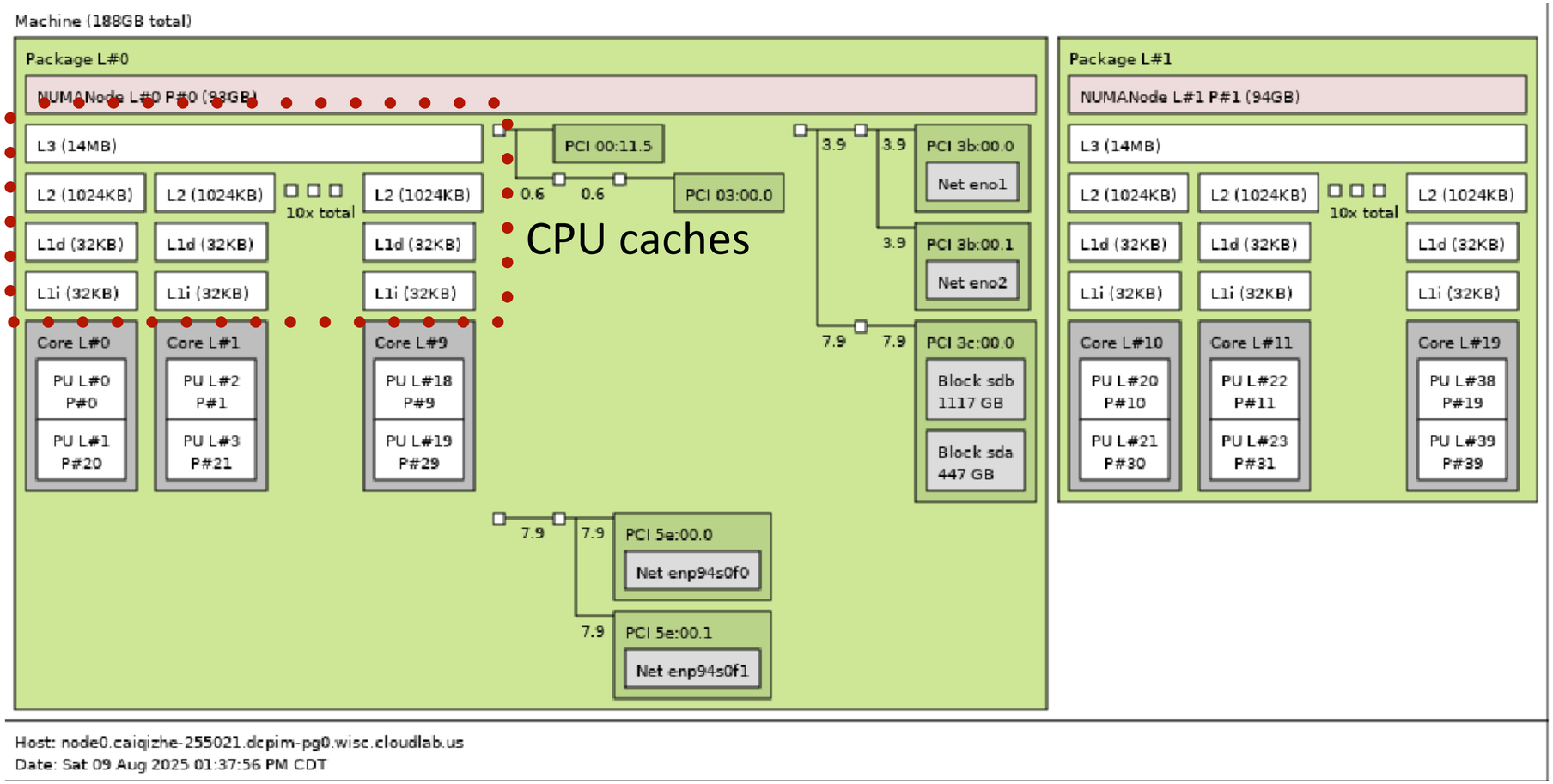
- Check server's hardware topologies/configurations
- Hyper-threading: Logical core #0 and core #20 share the same physical core #0.

lstopo



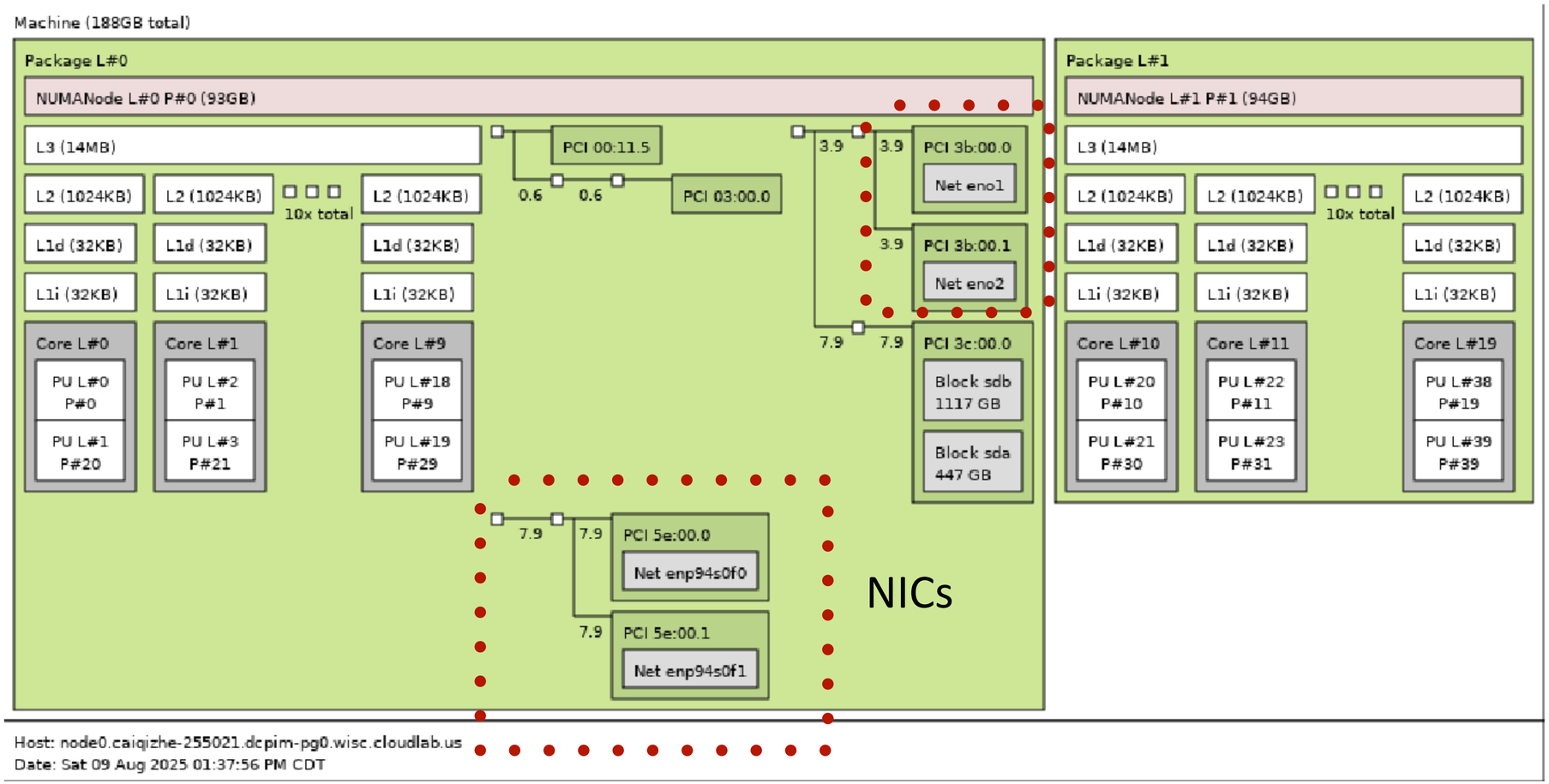
- Check server's hardware topologies/configurations
- Hyper-threading: Logical core #0 and core #20 share the same physical core #0
- Two CPU sockets/NUMA memories

lstopo



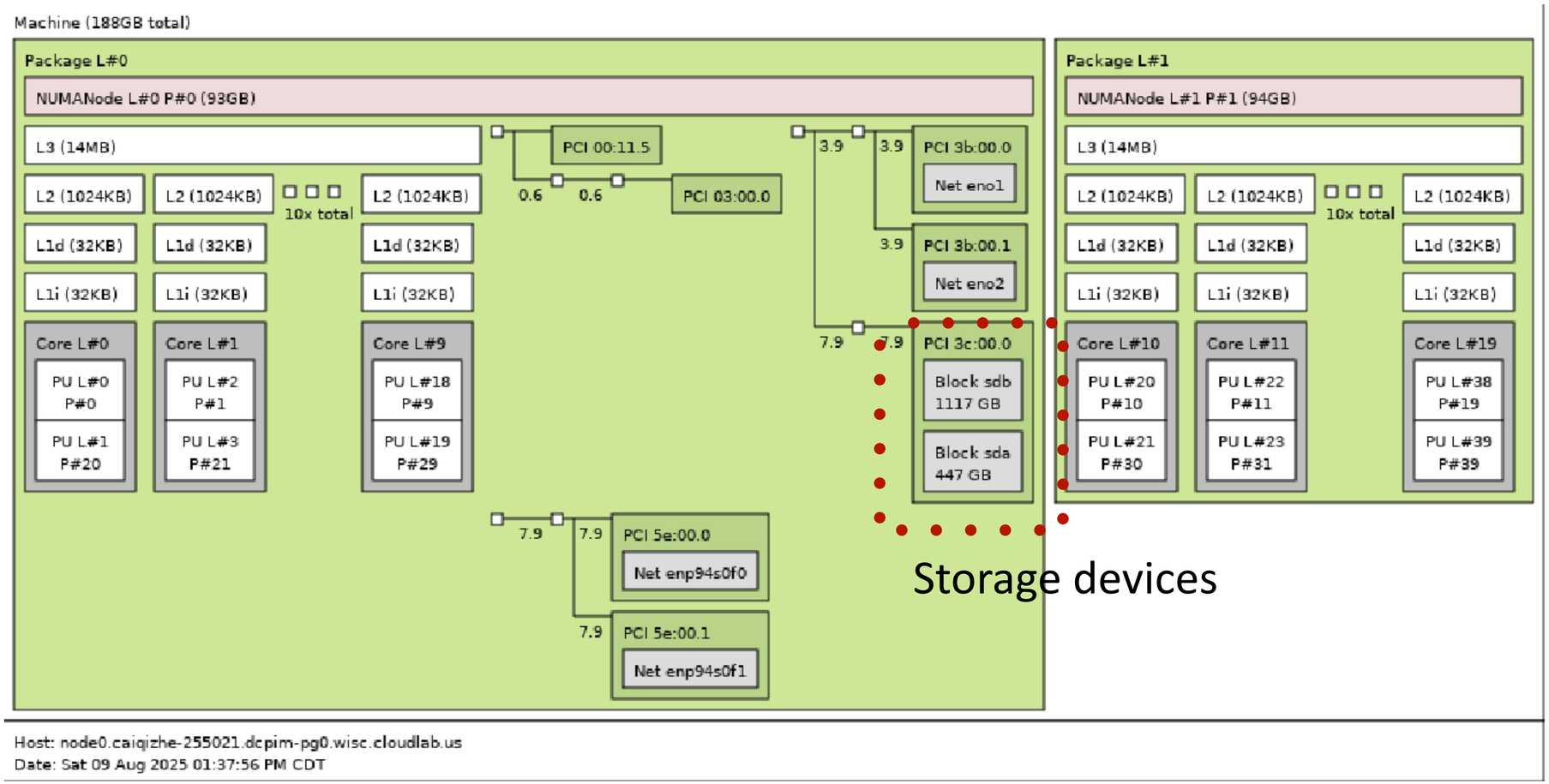
- Check server's hardware topologies/configurations
- Hyper-threading: Logical core #0 and core #20 share the same physical core #0
- Two CPU sockets/NUMA memories

lstopo



- Check server's hardware topologies/configurations
- Hyper-threading: Logical core #0 and core #20 share the same physical core #0
- Two CPU sockets/NUMA memories

lstopo



- Check server's hardware topologies/configurations
- Hyper-threading: Logical core #0 and core #20 share the same physical core #0
- Two CPU sockets/NUMA memories

Perf

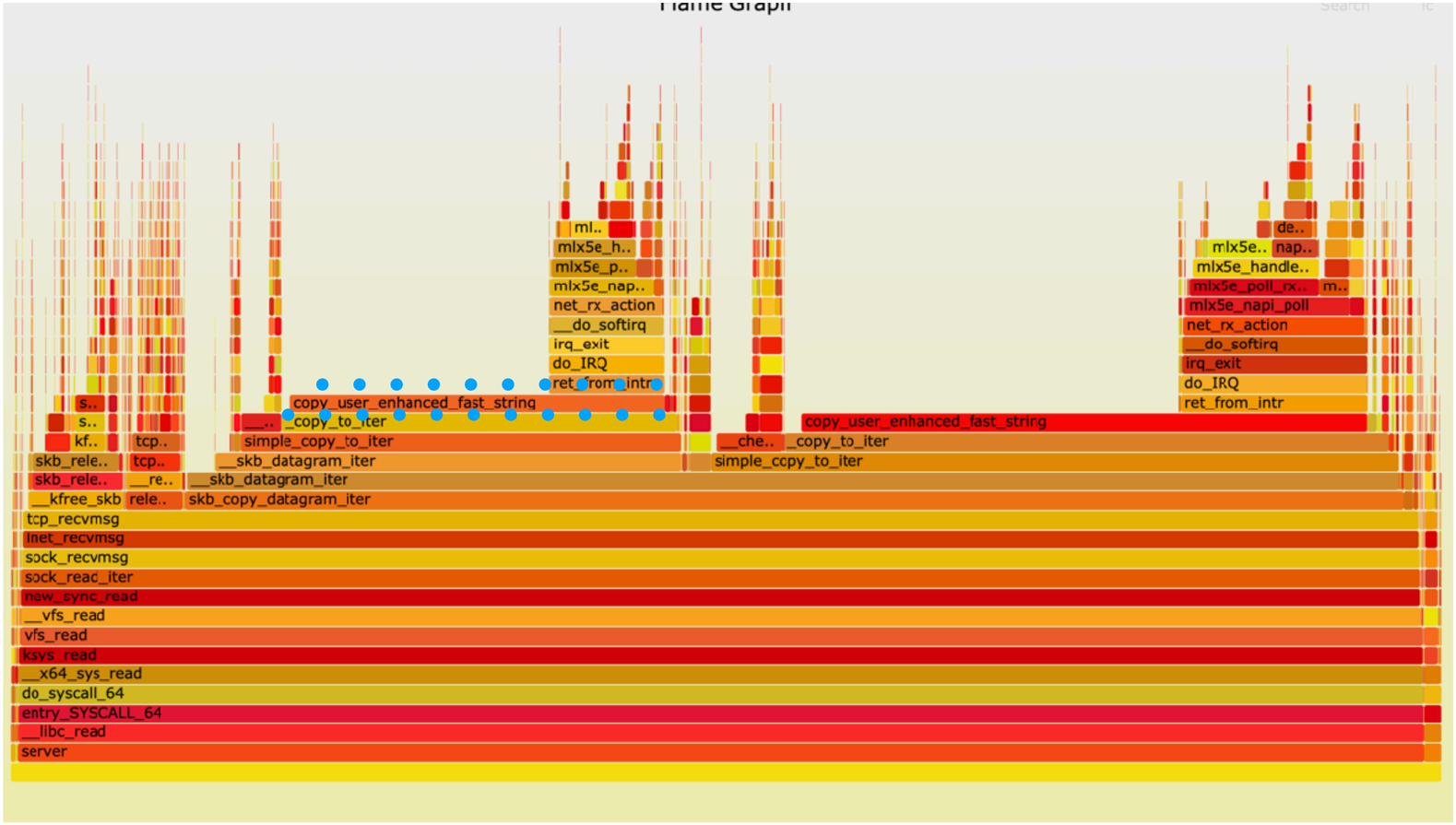
Overhead	Shared Object	Symbol
32.31%	[kernel]	[k] copy_user_enhanced_fast_string
16.86%	[kernel]	[k] clear_page_erms
2.08%	[kernel]	[k] kernel_init_free_pages.part.0
1.65%	[kernel]	[k] tcp_sendmsg_locked
1.57%	[kernel]	[k] psi_group_change
1.53%	[kernel]	[k] syscall_return_via_sysret
1.10%	[kernel]	[k] syscall_exit_to_user_mode
0.94%	[kernel]	[k] _raw_spin_lock_irqsave
0.91%	[kernel]	[k] skb_release_data
0.86%	[kernel]	[k] entry_SYSCALL_64_after_hwframe
0.84%	[kernel]	[k] __check_object_size.part.0
0.77%	[kernel]	[k] tcp_recvmsg_locked
0.75%	[kernel]	[k] prep_compound_page
0.65%	[kernel]	[k] __tcp_transmit_skb
0.62%	[kernel]	[k] get_page_from_freelist
0.62%	[kernel]	[k] _raw_spin_lock
0.57%	[kernel]	[k] entry_SYSCALL_64
0.55%	[kernel]	[k] __skb_datagram_iter
0.52%	[kernel]	[k] __inet_lookup_established
0.51%	[kernel]	[k] menu_select
0.46%	[kernel]	[k] read_tsc
0.43%	[kernel]	[k] dst_release
0.39%	[kernel]	[k] skb_page_frag_refill
0.39%	[kernel]	[k] __schedule
0.38%	[kernel]	[k] native_sched_clock
0.38%	[kernel]	[k] _raw_spin_lock_bh
0.36%	[kernel]	[k] tcp_v4_rcv
0.35%	[kernel]	[k] __alloc_pages
0.35%	[kernel]	[k] wait_woken
0.33%	[kernel]	[k] tcp_rcv_established

- sudo perf top
  - Get per-function CPU costs (excluding child function CPU costs)

Perf with flamegraph

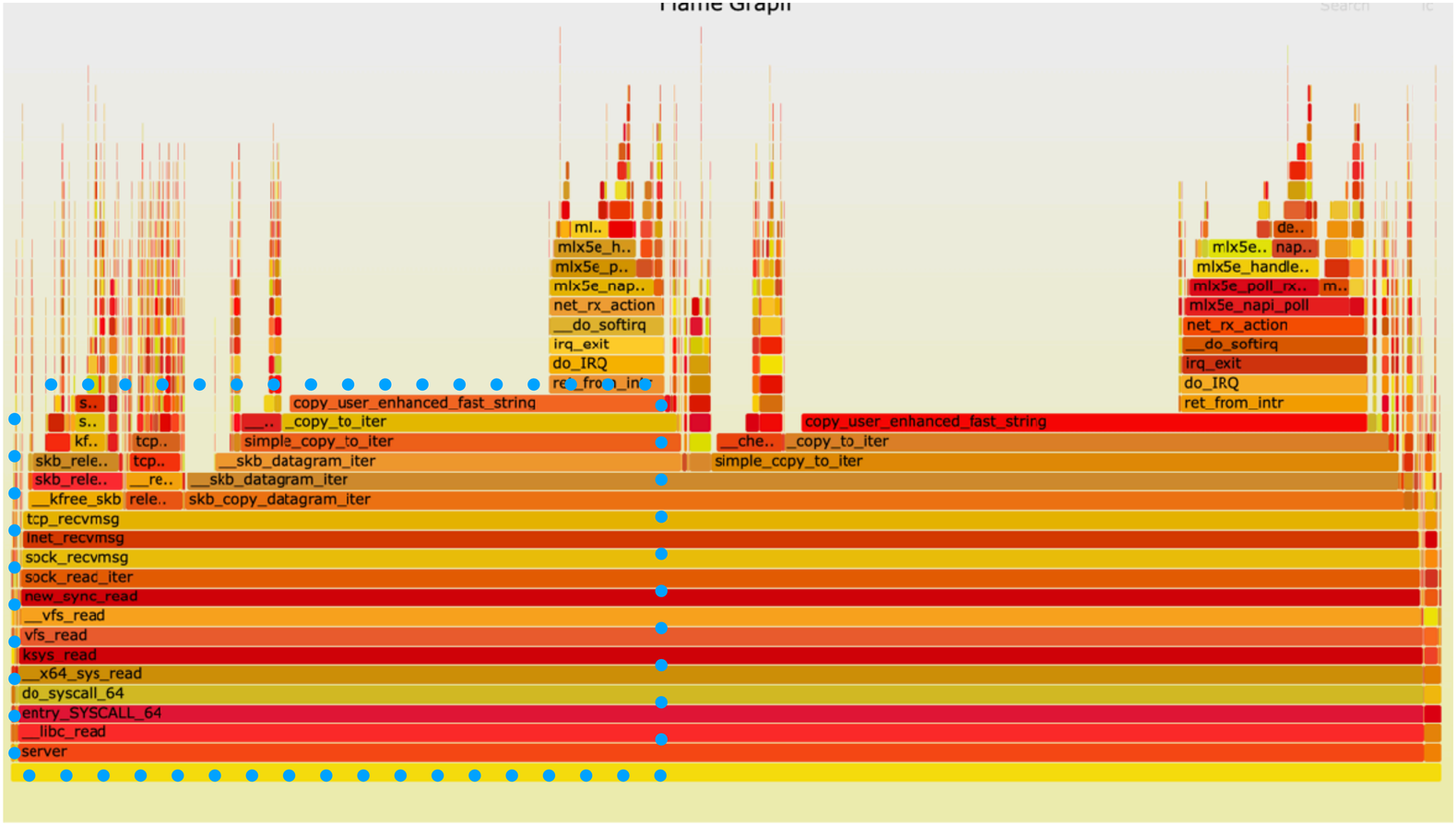
- Understand the data paths

Perf with flamegraph



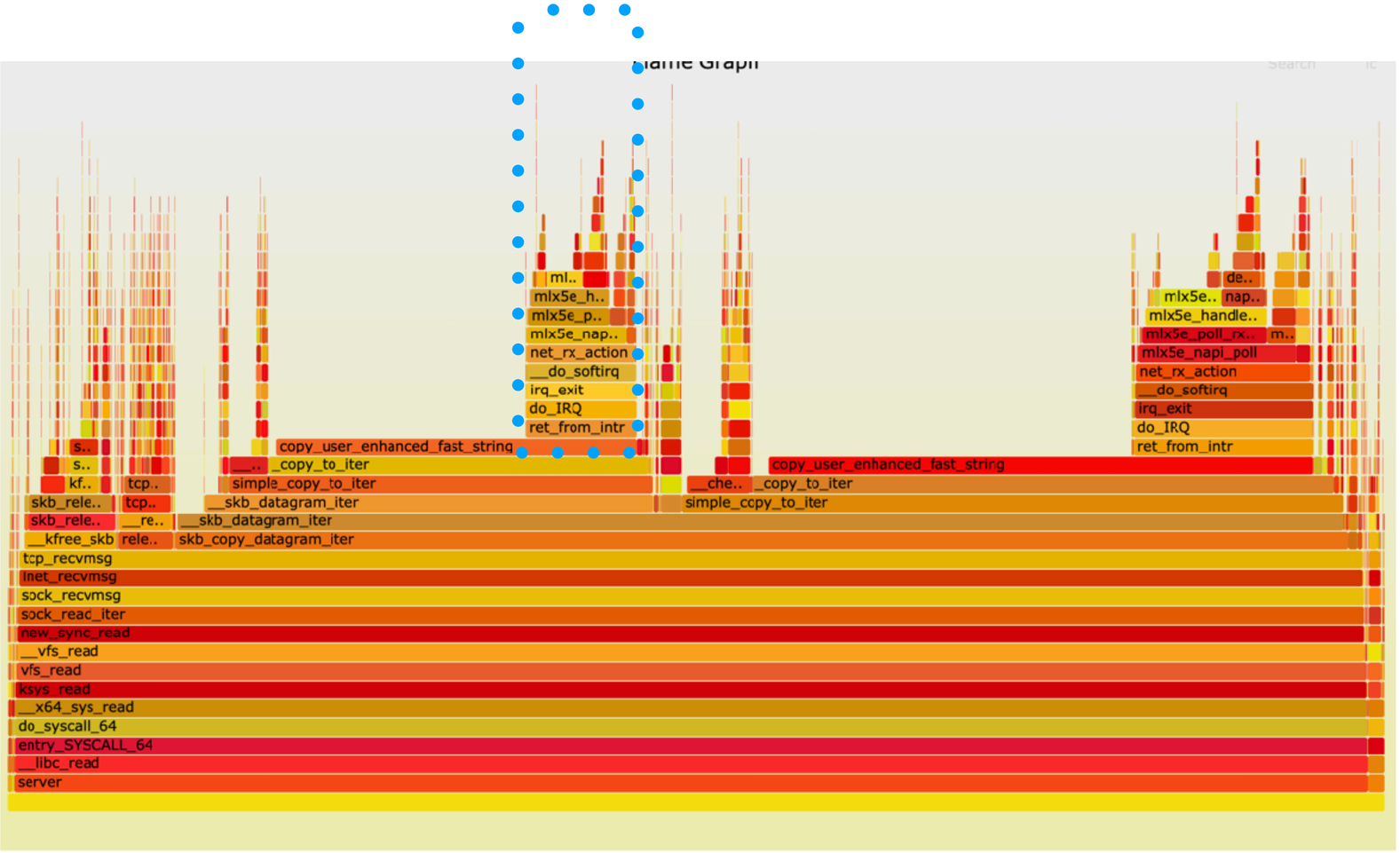
- The Flamegraph allows users to view calling traces and function usage
  - https://github.com/brendangregg/FlameGraph

Perf with flamegraph



- copy_user_enhanced_fast_string: server -> syscall -> vfs -> tcp layer.
  - Copying the data from kernel to OS on the receiver side

Perf with flamegraph



- Interrupts are triggered within the copy function.
  - Network packets are arrived at the receiver side.

Perf sched

`perf sched record -- sleep 1`

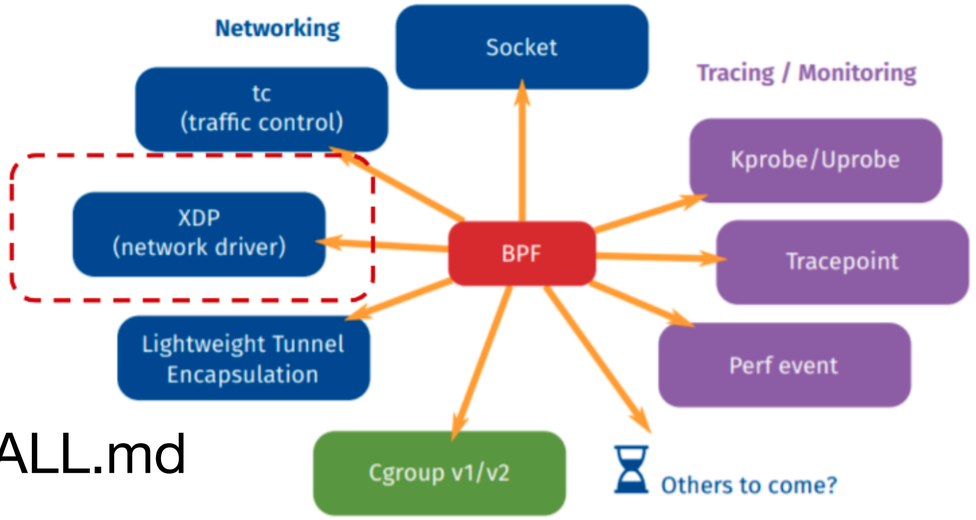
```
# perf sched latency

 -----------------------------------------------------------------------------------------------------------------
  Task                  |   Runtime ms  | Switches | Average delay ms | Maximum delay ms | Maximum delay at       |
 -----------------------------------------------------------------------------------------------------------------
  cat:(6)               |     12.002 ms |        6 | avg:    17.541 ms | max:    29.702 ms | max at: 991962.948070 s
  ar:17043              |      3.191 ms |        1 | avg:    13.638 ms | max:    13.638 ms | max at: 991963.048070 s
  rm:(10)               |     20.955 ms |       10 | avg:    11.212 ms | max:    19.598 ms | max at: 991963.404069 s
  objdump:(6)           |     35.870 ms |        8 | avg:    10.969 ms | max:    16.509 ms | max at: 991963.424443 s
  :17008:17008          |    462.213 ms |       50 | avg:    10.464 ms | max:    35.999 ms | max at: 991963.120069 s
  grep:(7)              |     21.655 ms |       11 | avg:     9.465 ms | max:    24.502 ms | max at: 991963.464082 s
  fixdep:(6)            |     81.066 ms |        8 | avg:     9.023 ms | max:    19.521 ms | max at: 991963.120068 s
  mv:(10)               |     30.249 ms |       14 | avg:     8.380 ms | max:    21.688 ms | max at: 991963.200073 s
  ld:(3)                |     14.353 ms |        6 | avg:     7.376 ms | max:    15.498 ms | max at: 991963.452070 s
  recordmcount:(7)      |     14.629 ms |        9 | avg:     7.155 ms | max:    18.964 ms | max at: 991963.292100 s
  svstat:17067          |      1.862 ms |        1 | avg:     6.142 ms | max:     6.142 ms | max at: 991963.280069 s
  cc1:(21)              |   6013.457 ms |     1138 | avg:     5.305 ms | max:    44.001 ms | max at: 991963.436070 s
  gcc:(18)              |     43.596 ms |       40 | avg:     3.905 ms | max:    26.994 ms | max at: 991963.380069 s
  ps:17073              |     27.158 ms |        4 | avg:     3.751 ms | max:     8.000 ms | max at: 991963.332070 s
...]
```

- Getting runtime of each app
- Getting number of context switches
- Getting delay of apps
  - The CPU core is busy running other tasks, causing the task to wait.

eBPF

- **Berkeley Packet Filter (BPF)**
  - BPF lets user-space programs attach filters to sockets, controlling which data passes through.
  - Without requiring device changes or kernel recompilation.
  - User programs (e.g., C) are compiled into BPF bytecode, which runs in the kernel

- **Extended BPF (eBPF)**
  - People have extended BPF to use this useful tool in many places.
  - Linux kernel has now many “hook” points and runs an eBPF program whenever reaching the point
    - **Technically, these hook points can be placed anywhere in the kernel.**
  - Two common tools:
    - BCC: complex tools and daemons
    - Bpftrace: command-line tools



https://github.com/bpftrace/bpftrace/blob/master/INSTALL.md

Example

- **sudo bpftrace -e 'kprobe:tcp_recvmsg { printf("pid=%d comm=%s len=%d\n", pid, comm, arg2);print(kstack); }'**
- **Get the call stack of tcp_recvmsg**

```
pid=2950 comm=iperf len=131072

        tcp_recvmsg+1
        sock_recvmsg+113
        __sys_recvfrom+183
        __x64_sys_recvfrom+36
        do_syscall_64+89
        entry_SYSCALL_64_after_hwframe+99
```

https://github.com/bpftrace/bpftrace/blob/master/INSTALL.md

Questions?