

swiotlb Under Pressure

High-Speed NICs in Confidential VMs

Dragoş Tătulea
NVIDIA



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Confidential Compute VMs

Memory

Hypervisor is not trusted

Guest RAM starts private and encrypted

The host cannot read private pages

Hardware support:
AMD SEV-SNP, Intel TDX, ARM CCA

Devices (PCI)

Can be trusted or not trusted

Trusted devices need hardware support:
AMD SEV-TIO, Intel TDX Connect, ARM DA

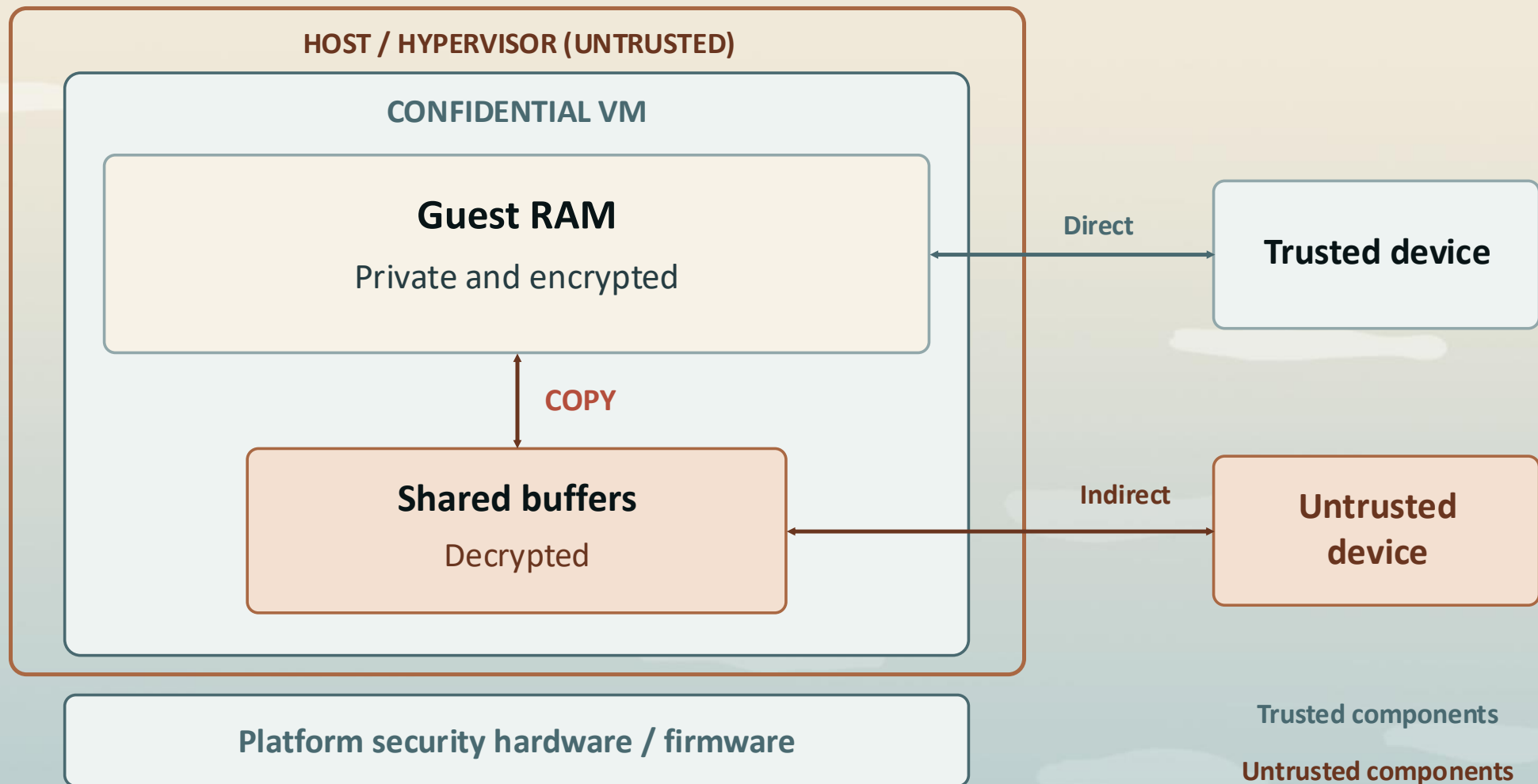
Platform decides if device is trusted

Trusted devices can access private memory
directly

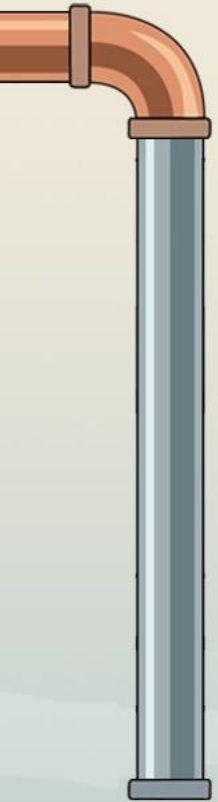
Our focus: **untrusted devices**



Confidential VM trust boundary



Shared Buffers in Linux



Streaming mappings

Shared memory achieved by swiotlb bounce buffers

Managed by DMA layer

dma map: swiotlb allocation

dma sync: copy

dma unmap: swiotlb deallocation

Coherent mappings

Not backed by swiotlb

Sets memory as decrypted

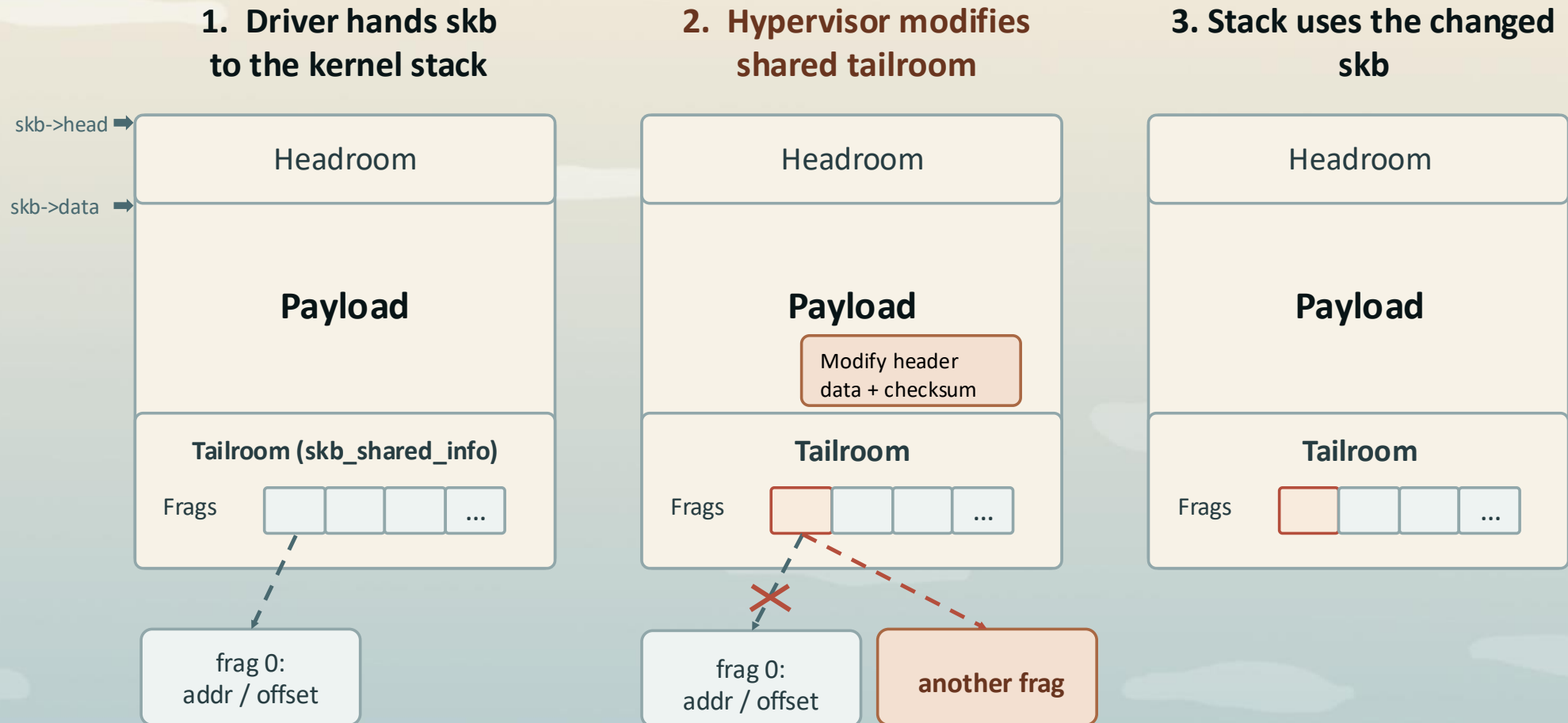


LINUX
PLUMBERS CONFERENCE

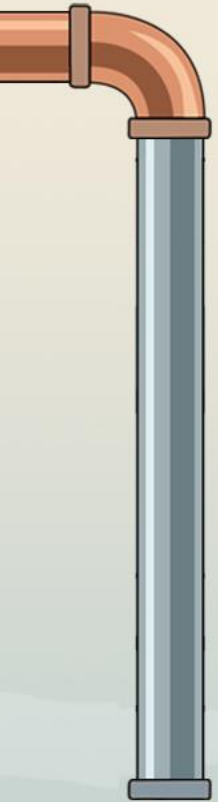
PRAGUE, CZECHIA / OCT. 5-7, 2026

Why is the copy needed?

TOCTOU attack example:



swiotlb pools and configuration



Pool structure

Split in areas (default: 1 area / CPU)

Allocation unit: 2 KB slabs

Areas partition access and carry per-area locks

Types of pools: global, dynamic, transient, per-device

Configuration choices

Boot-time pool sizing and area count

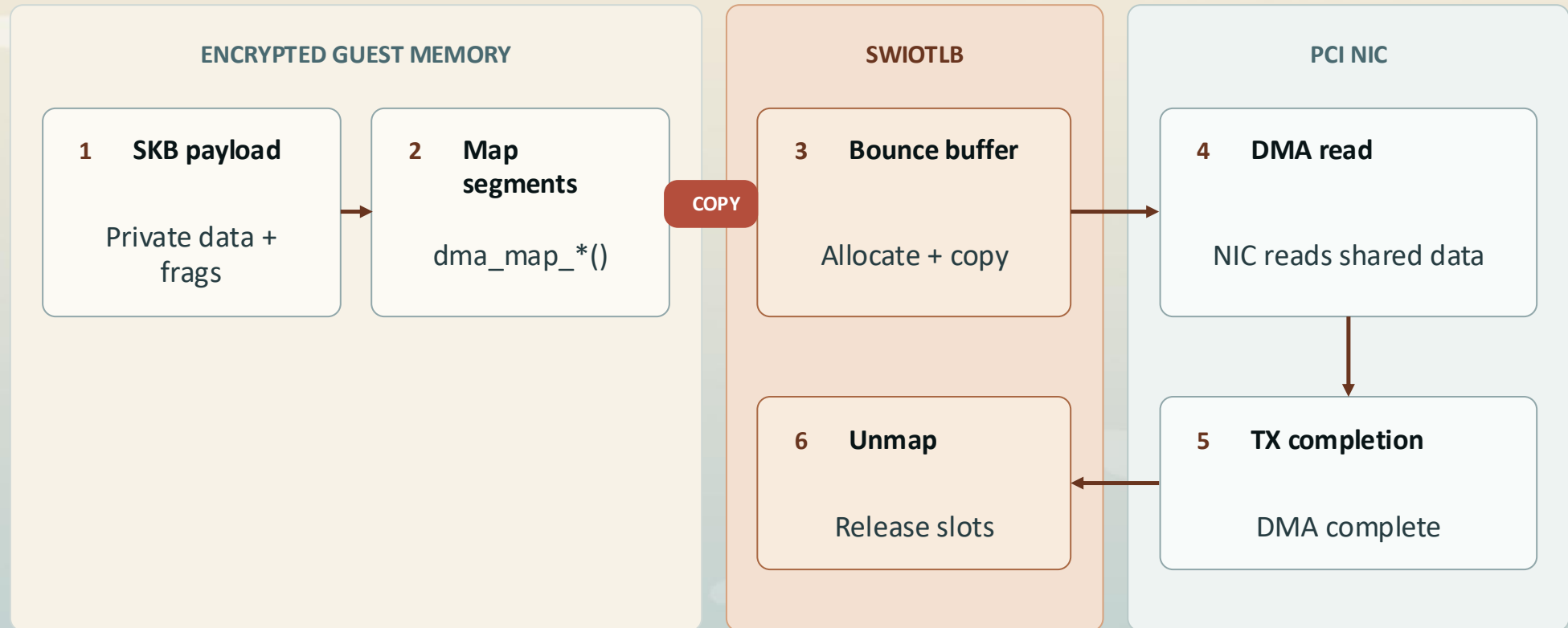
There is a max reservation size (1G, platform dependent)

Reservation is not guaranteed

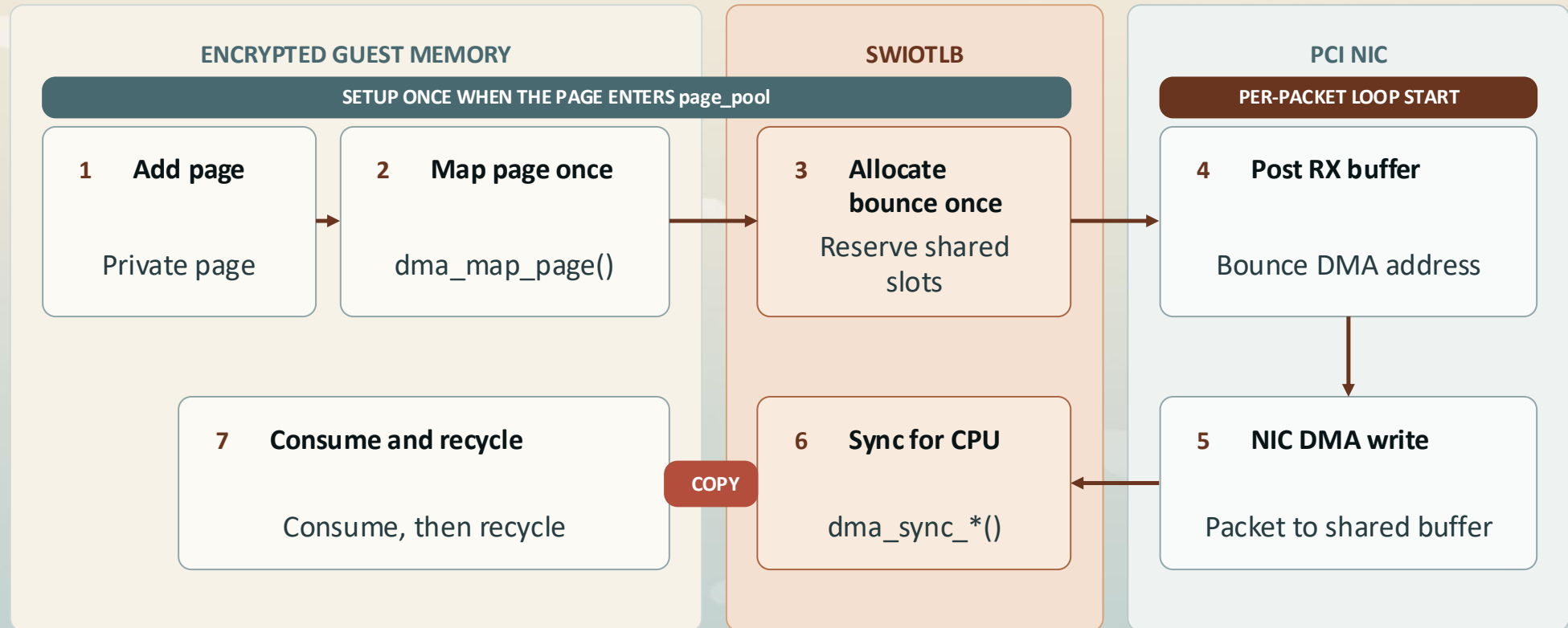
Dynamic pools can expand capacity but not used here



swiotlb TX data path



swiotlb RX data path



DMA unmap called when page exits the page_pool



RX ring occupancy per swiotlb area

1 GB swiotlb mem ÷ 64 areas

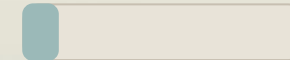
=

16 MB per area

AREA-EQUIVALENT OCCUPANCY

1K ring, MTU 1500

$1024 \times 2 \text{ KiB} = 2 \text{ MB}$



12.5%

4K ring, MTU 1500

$4096 \times 2 \text{ KiB} = 8 \text{ MB}$



50%

1K ring, MTU 9000

$1024 \times 3 \times 4096 \text{ B} = 12 \text{ MB}$



75%

2K ring, MTU 9000

$2048 \times 3 \times 4096 \text{ B} = 24 \text{ MB}$



150%



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Benchmark environment

Setup – x86

x86 AMD SEV-SNP guest

ConnectX-8 (400Gbps port) with PCI passthrough

CoCo VM with 64 CPUs and 64 GB RAM

swiotlb=524288,64 (1GB, max)

Workload

iperf3 TCP, 1 stream per queue

Stream pinning - all on same CPU

RX, TX direction

MTU 1500:

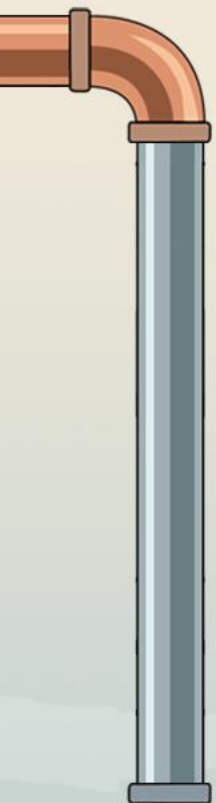
- 1, 2, 4, 8, and 16 streams
- ring size: 4K

MTU 9000:

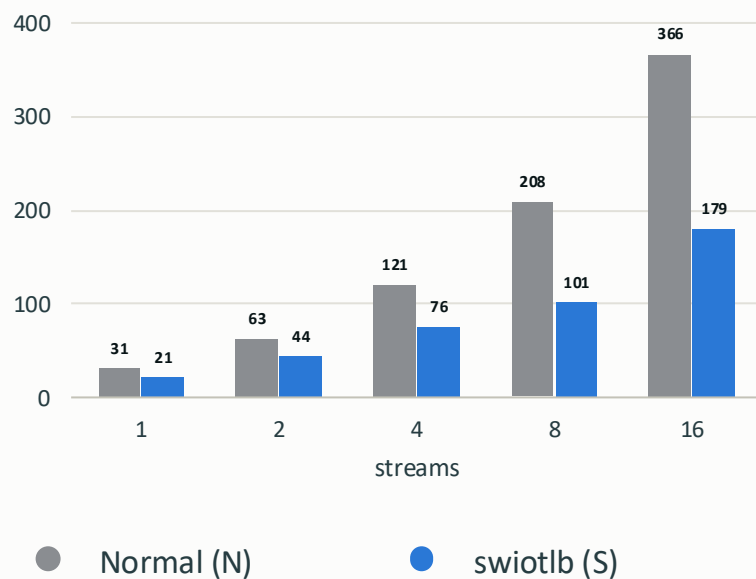
- 1, 2, 4, 8 and 16 streams
- ring size: 1K



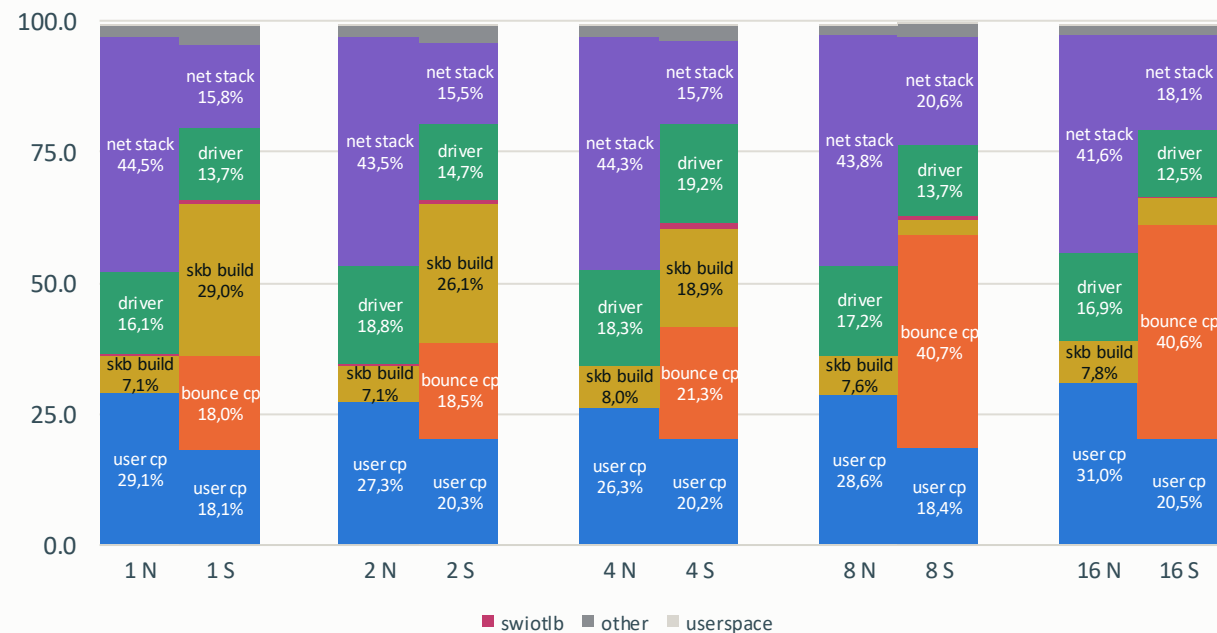
x86 RX at MTU 1500



Throughput (Gb/s)



CPU usage (%)



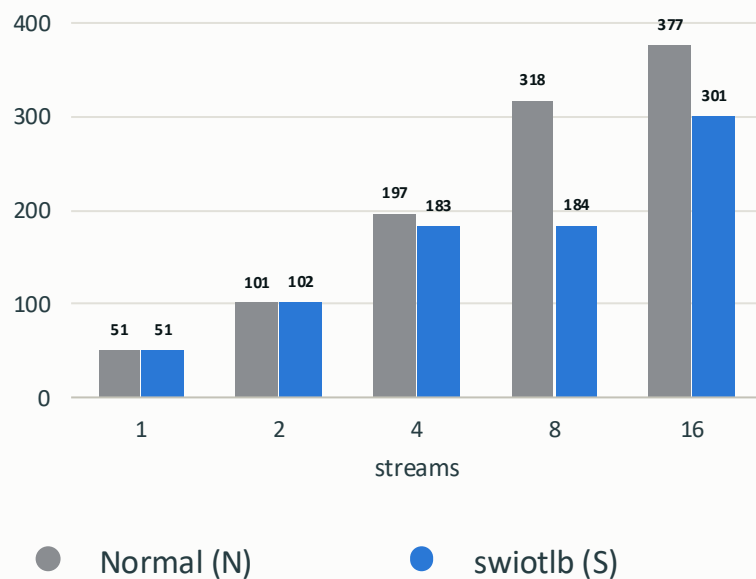
LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

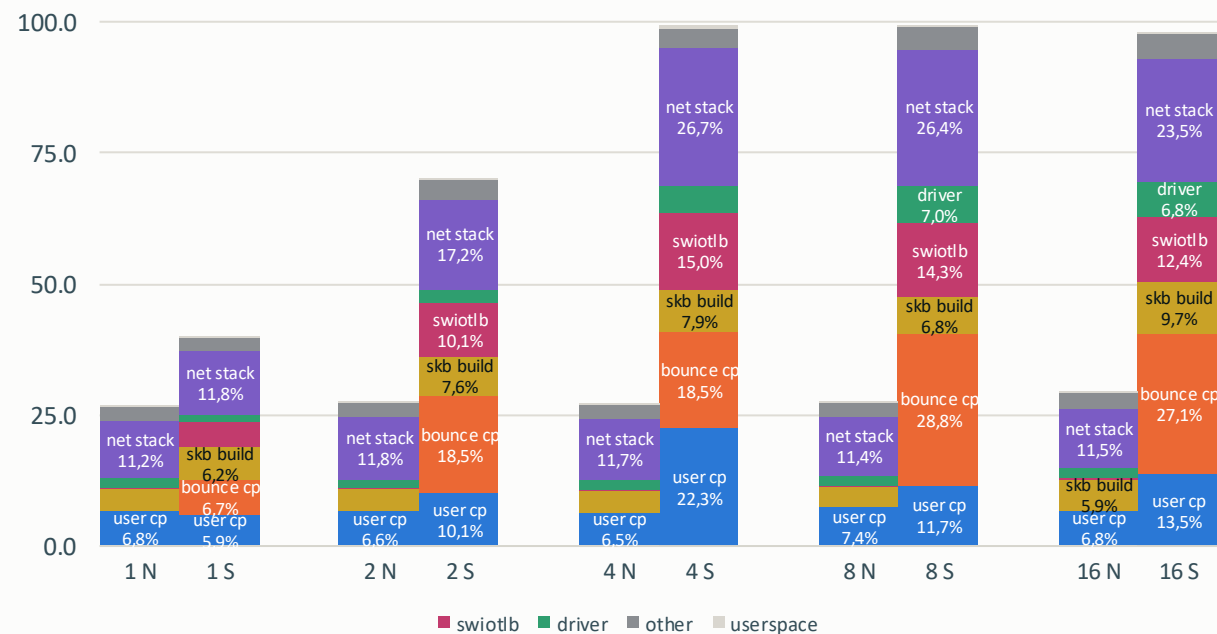
What we see: mostly bounce copy, swiotlb alloc only for ACKs

x86 TX at MTU 1500

Throughput (Gb/s)



CPU usage (%)



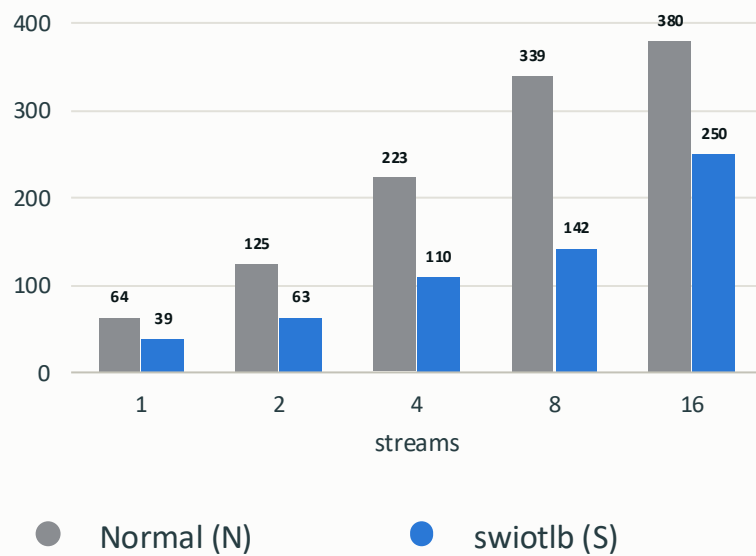
LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

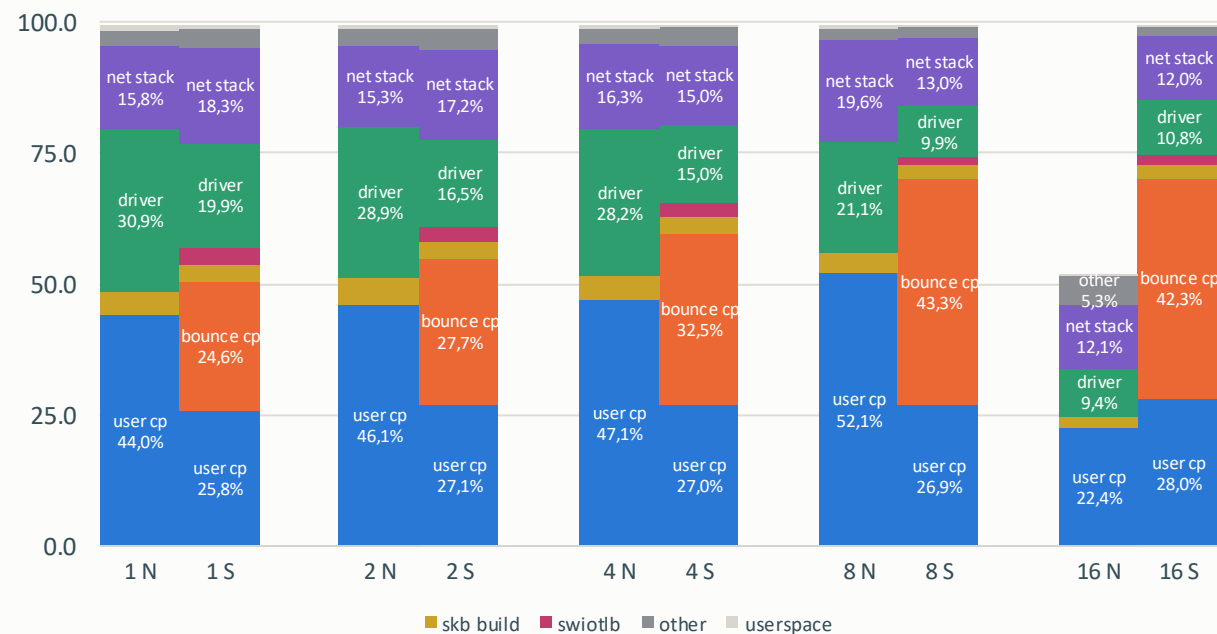
What we see: bounce copies + swiotlb alloc overhead

x86 RX at MTU 9000

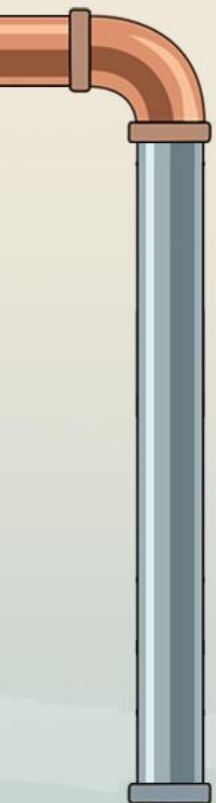
Throughput (Gb/s)



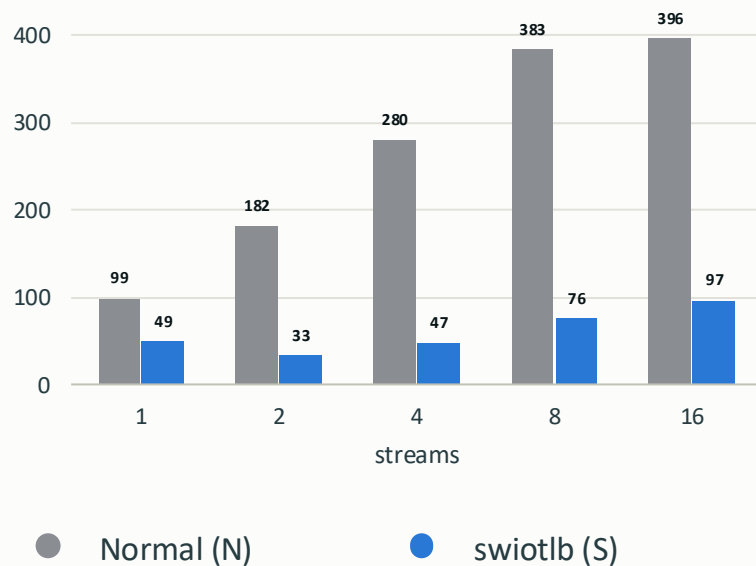
CPU usage (%)



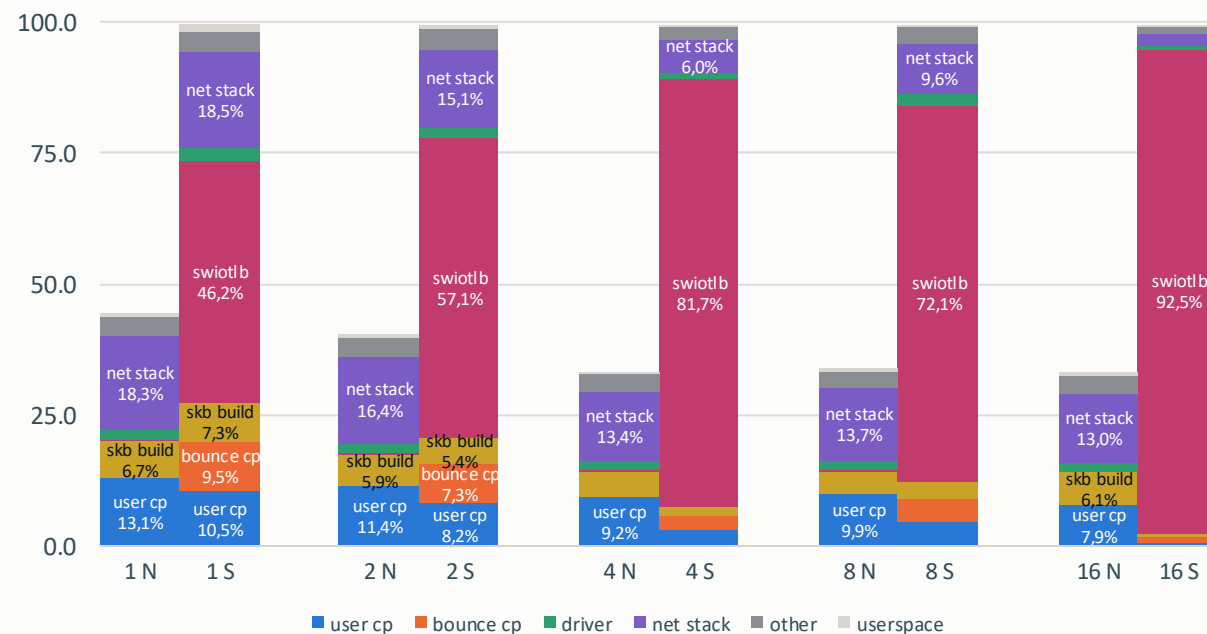
x86 TX at MTU 9000



Throughput (Gb/s)



CPU usage (%)



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

What we see: swiotlb memory almost depleted

Where time is spent

X86 CoCo VM + swiotlb, TX traffic, MTU 9000, 16 streams,

Overhead	Shared Object	Symbol
76.35%	[kernel]	[k] __pv_queued_spin_lock_slowpath
14.77%	[kernel]	[k] swiotlb_search_pool_area
1.44%	[kernel]	[k] __irqentry_text_start
1.23%	[kernel]	[k] __raw_callee_save__pv_queued_spin_unlock
0.95%	[kernel]	[k] __raw_spin_lock_irqsave
0.64%	[kernel]	[k] __pi_memcpy
0.41%	[kernel]	[k] pvclock_clocksource_read_nowd
0.36%	[kernel]	[k] _raw_spin_lock
0.28%	[kernel]	[k] _copy_from_iter
0.25%	[kernel]	[k] sev_es_ghcb_hv_call
0.12%	[kernel]	[k] swiotlb_release_slots
0.10%	[kernel]	[k] asm_sysvec_apic_timer_interrupt

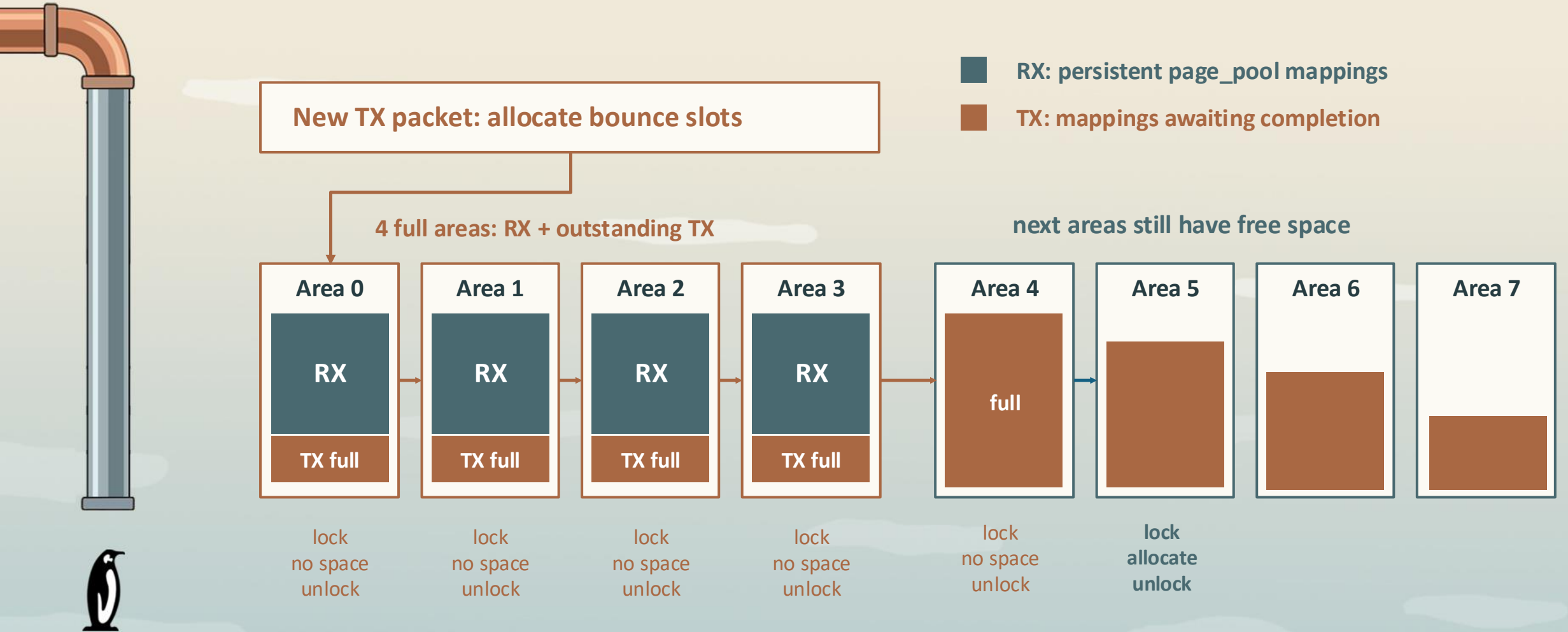
Low memory, searching in other areas



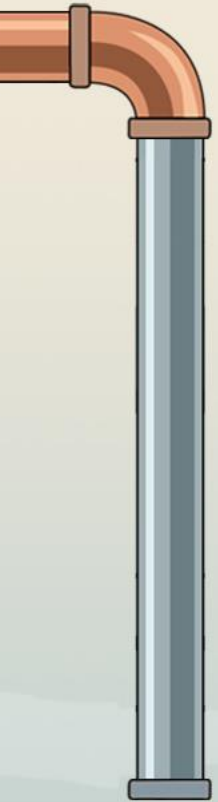
LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Contention: RX occupancy pushes TX into other areas



The Problems



TX has high overhead due to per-packet swiotlb allocations

RX page_pool can drain swiotlb areas

This makes TX overhead worse: high memory usage in swiotlb areas usage triggers search in next areas

Or: run out of memory from the start

What if there are multiple untrusted devices?



Next Step: Avoid the swiotlb

Implement bounce buffers inside the driver

RX:

- Ring uses static coherent DMA
- page_pool without DMA: allocates encrypted memory
- During packet RX, copy from ring into page_pool page

TX:

- TX pool: pre-allocated pool of coherent DMA buffers
- During xmit, copy from SKB into coherent DMA buffers from pool
- TX pool memory released on completion

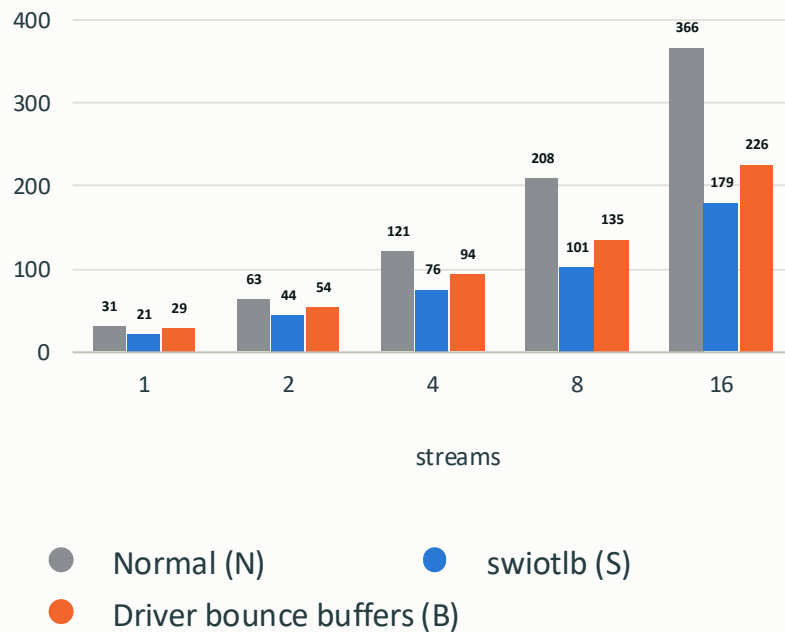


LINUX
PLUMBERS CONFERENCE

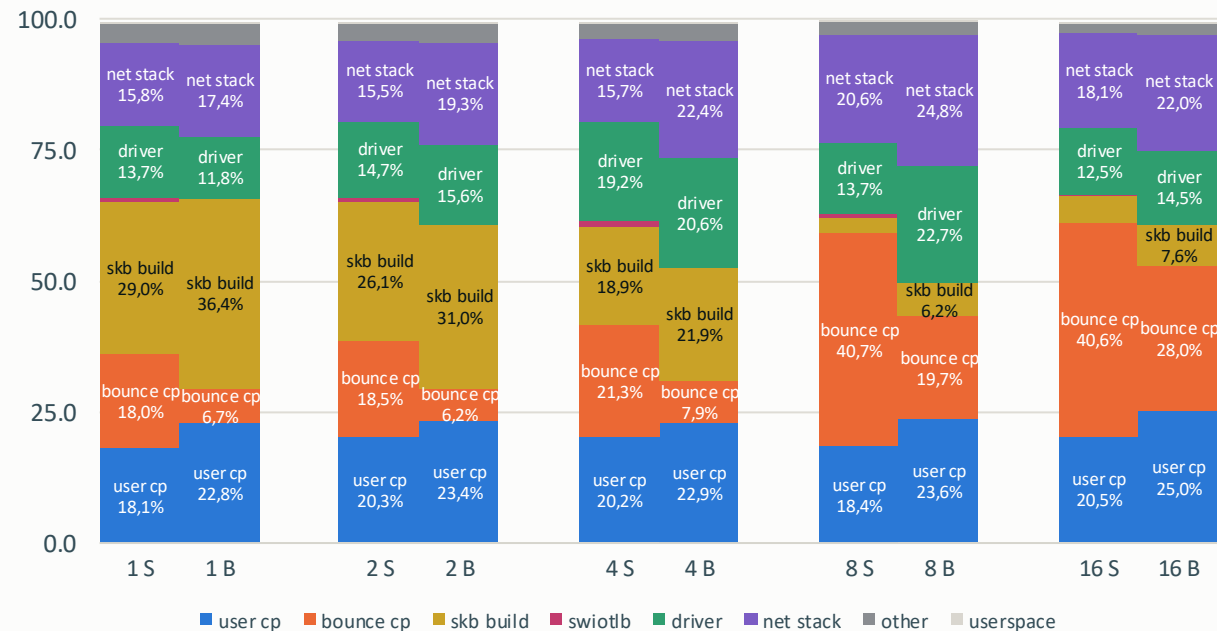
PRAGUE, CZECHIA / OCT. 5-7, 2026

x86 RX at MTU 1500

Throughput (Gb/s)



CPU usage (%)



What we see: mostly bounce copy, no swiotlb alloc for ACKs

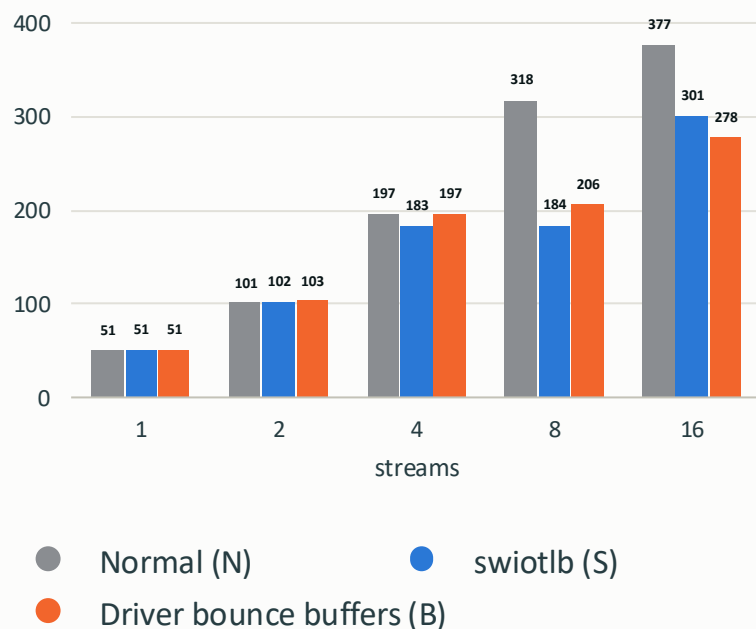


LINUX
PLUMBERS CONFERENCE

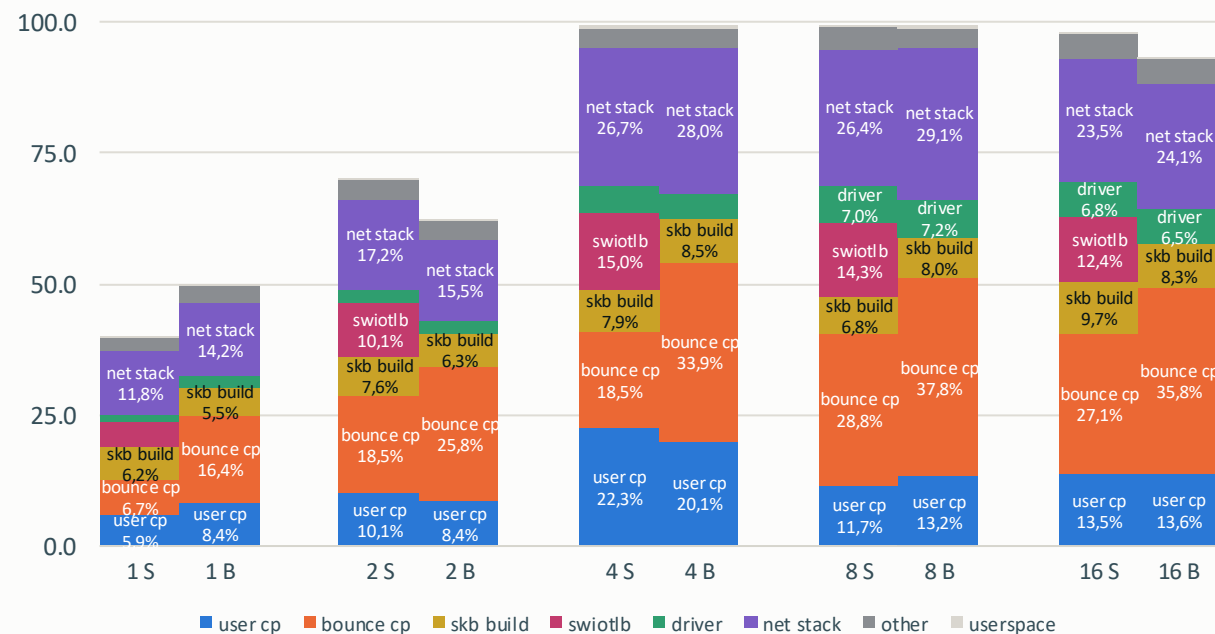
PRAGUE, CZECHIA / OCT. 5-7, 2026

x86 TX at MTU 1500

Throughput (Gb/s)



CPU usage (%)



What we see: mostly bounce copy, no swiotlb overhead



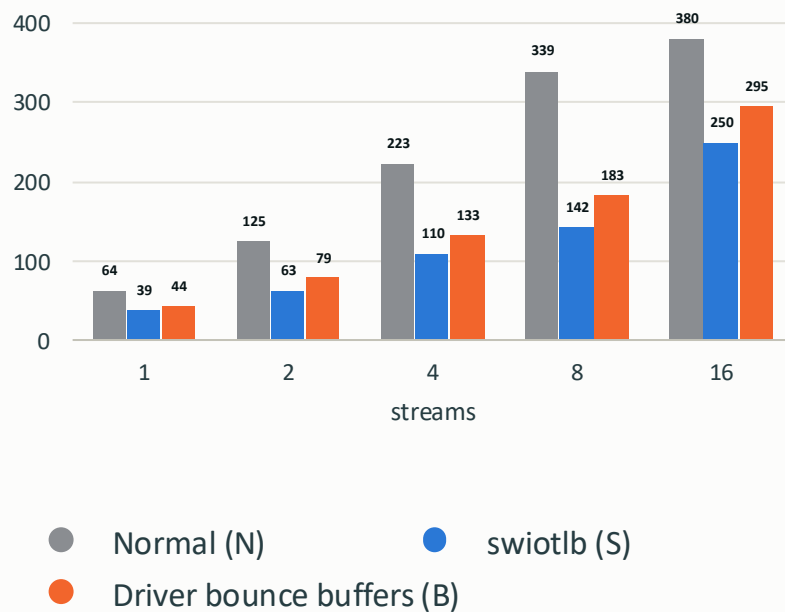
LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

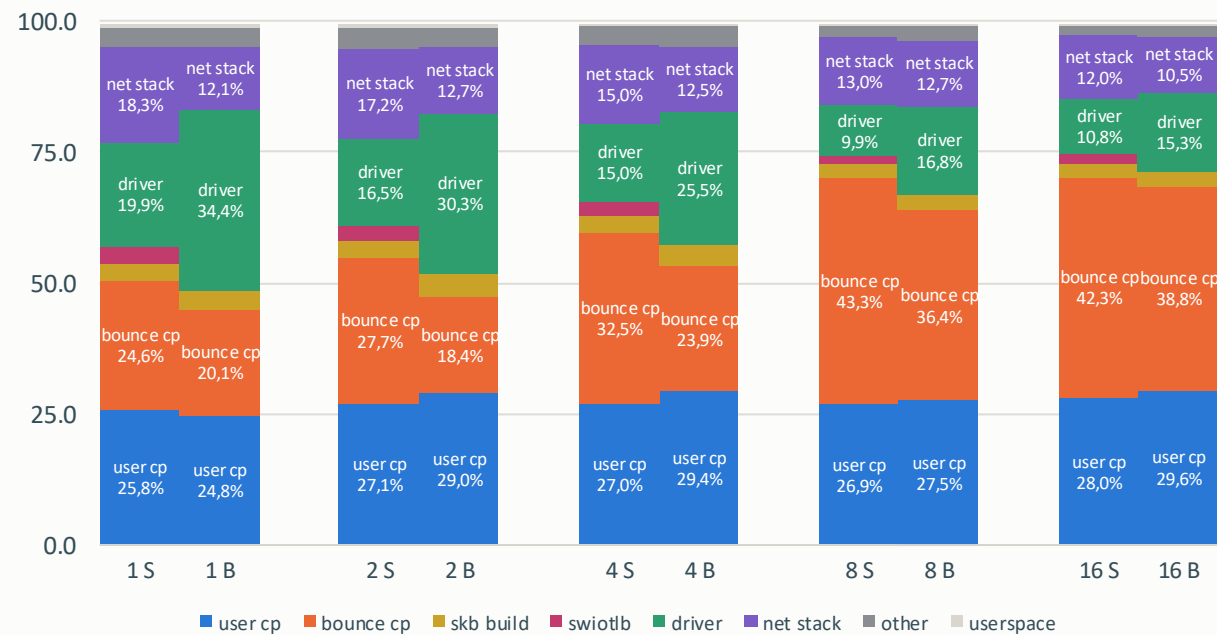


x86 RX at MTU 9000

Throughput (Gb/s)

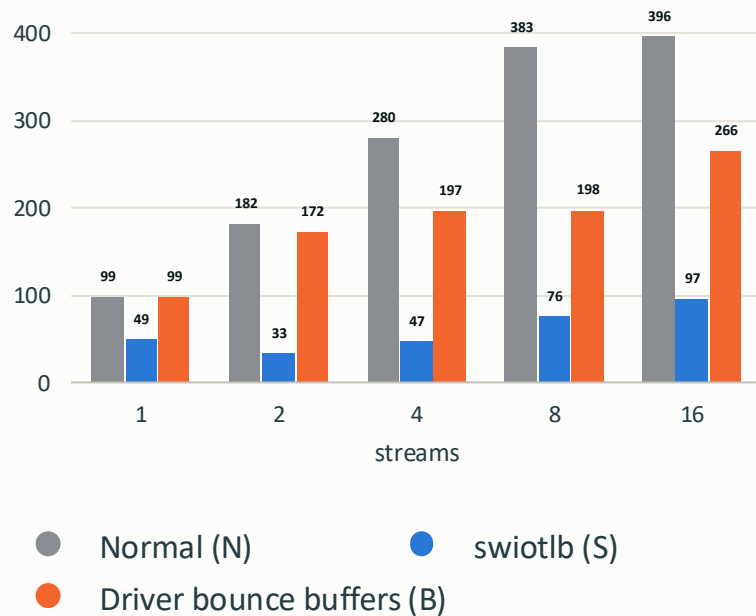


CPU usage (%)

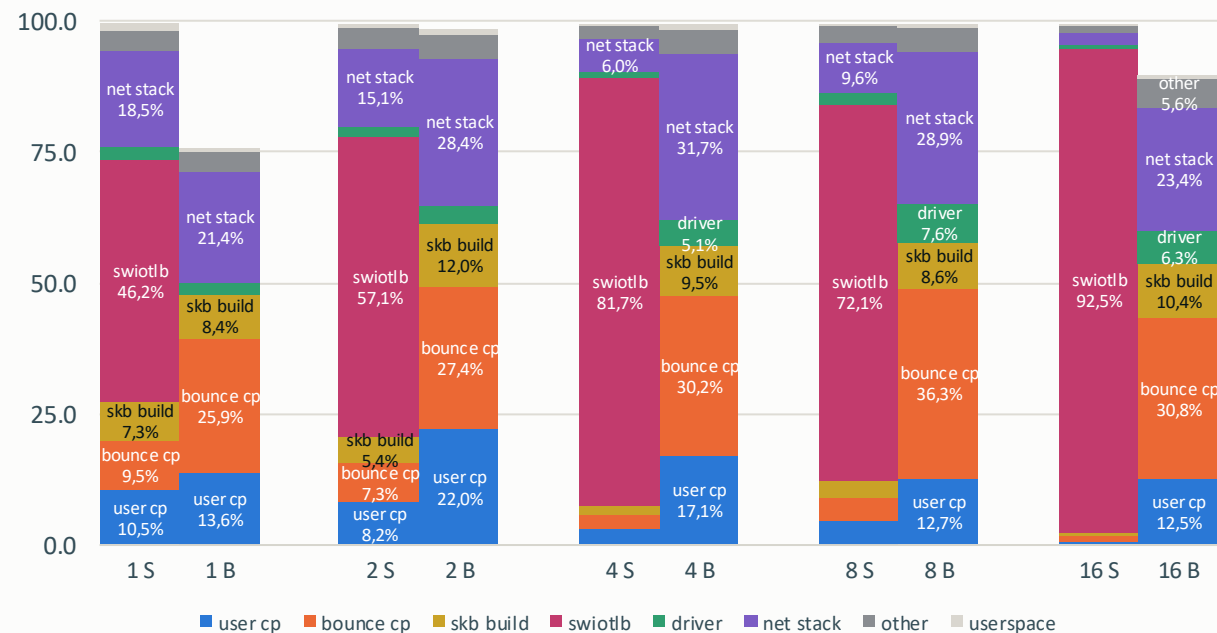


x86 TX at MTU 9000

Throughput (Gb/s)



CPU usage (%)



What we see: no overhead for driver bounce-buffer



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Platform considerations

64K page systems with page sized DMA mapping granule: more swiotlb memory fragmentation

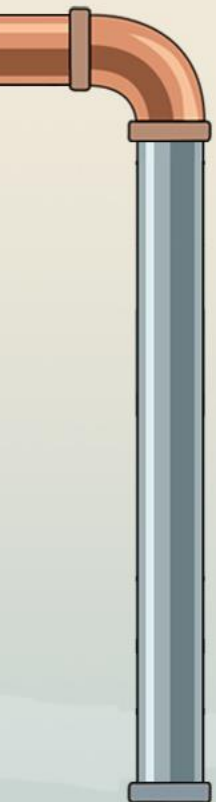
Driver bounce buffer avoids TX DMA map/unmap



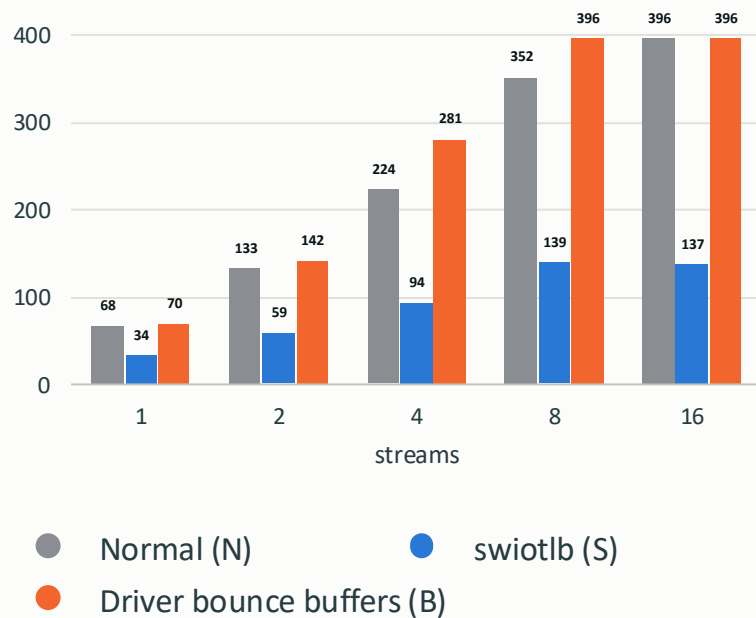
LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

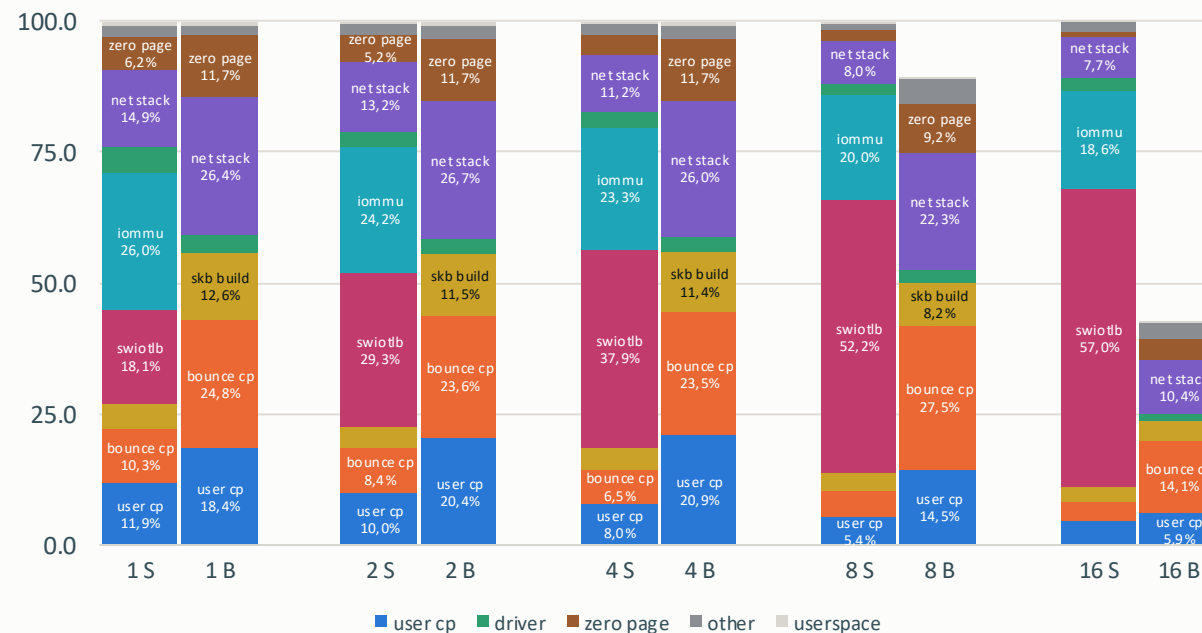
Example: aarch64 TX at MTU 9000



Throughput (Gb/s)



CPU usage (%)



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Grace CPU system, simulated CoCo env, iommu mode: strict, page size: 64K

Existing work: avoid swiotlb copy

[swiotlb: avoid swiotlb copy on network sockets](#) by Luigi Rizzo

Main idea: Instead of 2 copies, copy userspace directly in shared memory

TX side:

- Divert socket allocations to use swiotlb

RX side:

- Allocate page_pool pages directly from swiotlb

Relaxes threat model: drivers that use napi_build_skb() open for TOCTOU attack on linear area

A more extensive series was sent a few days ago:

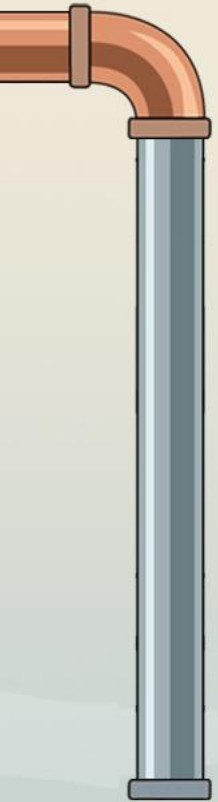
[DMA_PMD: PMD_SIZE-backed IO buffer pools](#)



**LINUX
PLUMBERS CONFERENCE**

PRAGUE, CZECHIA / OCT. 5-7, 2026

Swiotlb under pressure



Swiotlb has issues at scale. Fixing it doesn't seem viable:

- Generic pool shared
- Dynamic pools too small also shared
- Per-device swiotlb pool:
 - How and when to provision?
 - Needs manual config

Per device bounce buffer control from driver:

- Driver knows at runtime how much memory it needs
- DMA map/unmap avoidance can a bit of perf back
- How to make it generic. TBD



Discussion / Q&A



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA
OCT. 5-7, 2026

More numbers



**LINUX
PLUMBERS
CONFERENCE**

PRAGUE, CZECHIA
OCT. 5-7, 2026

Benchmark environment

Setup – aarch64

Grace CPU (Grace Blackwell system)

ConnectX-8 (400Gbps port) with PCI passthrough

Simulated CoCo:

- Set device untrusted for swiotlb
- Hack page_pool to allocate swiotlb memory

swiotlb=524288,64 (1GB, max)

iommu mode: strict

workload

iperf3 TCP, 1 stream per queue

Stream pinning - all on same CPU

RX, TX tests

MTU 1500:

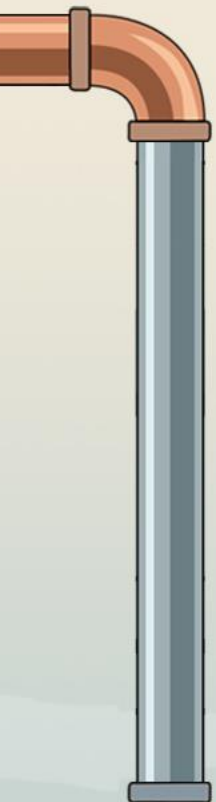
- 1, 2, 4, 8, and 16 streams
- ring size 4K

MTU 9000:

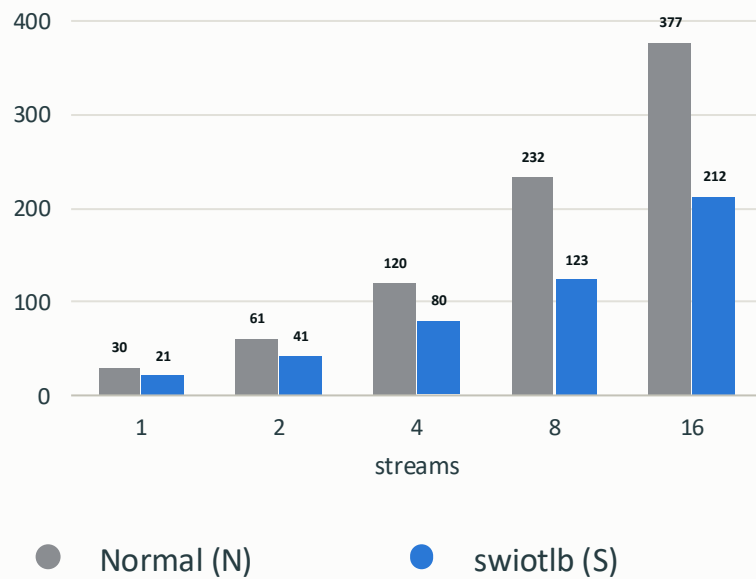
- 1, 2, 4, 8 and 16 streams
- ring size: 1024



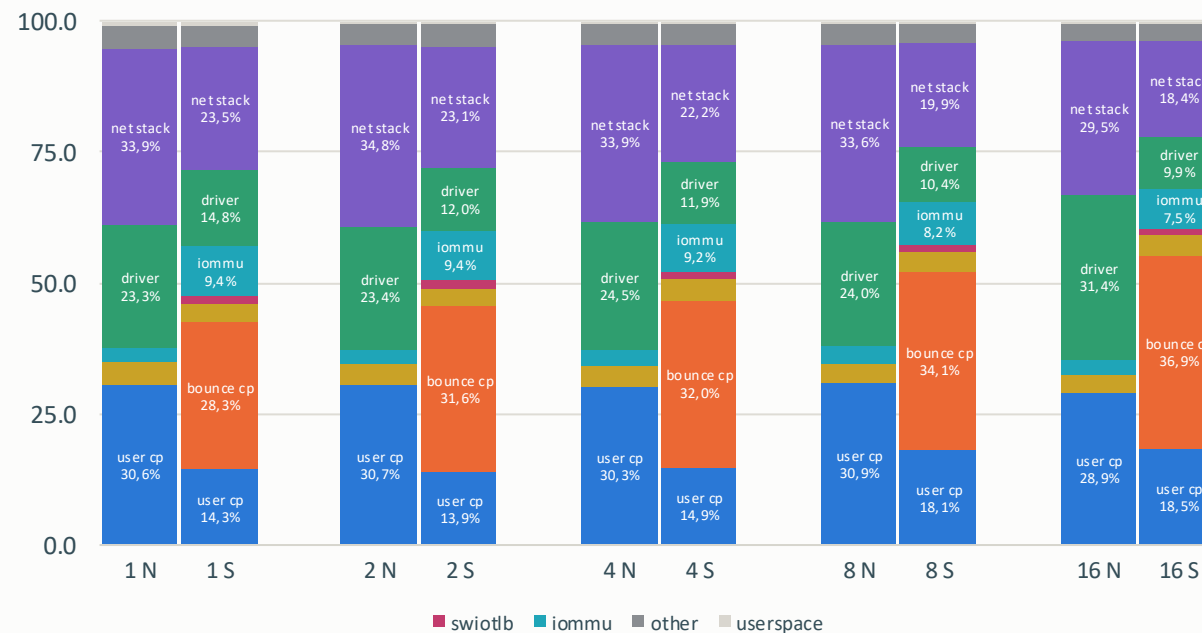
aarch64 RX at MTU 1500



Throughput (Gb/s)

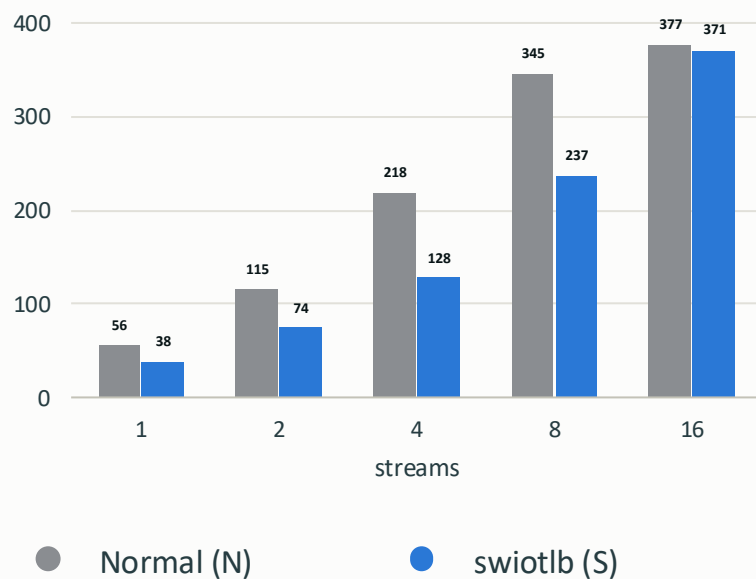


CPU usage (%)

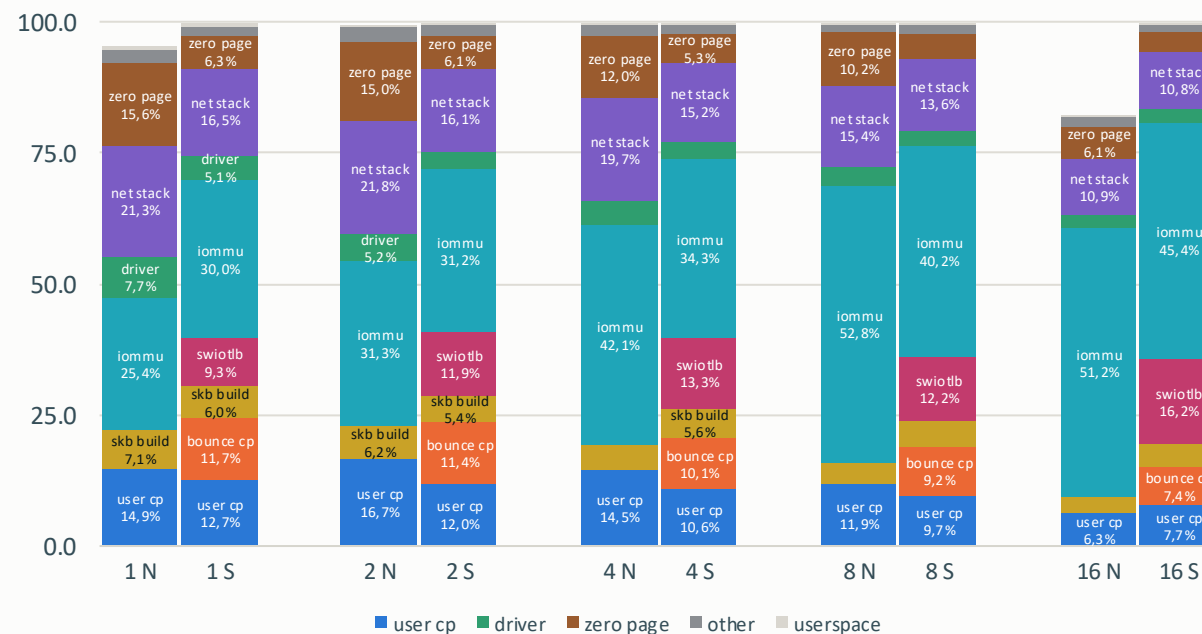


aarch64 TX at MTU 1500

Throughput (Gb/s)

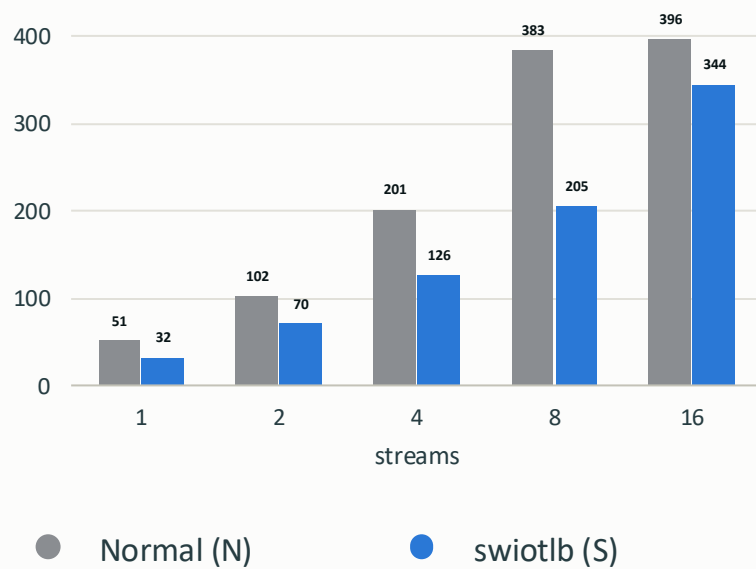


CPU usage (%)

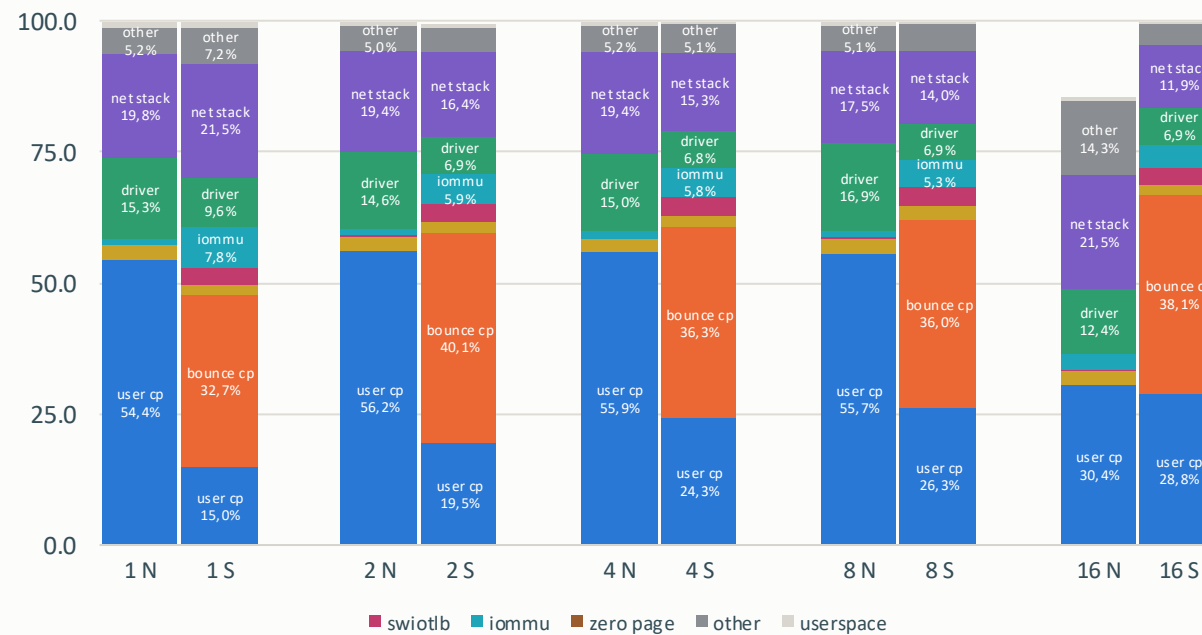


aarch64 RX at MTU 9000

Throughput (Gb/s)

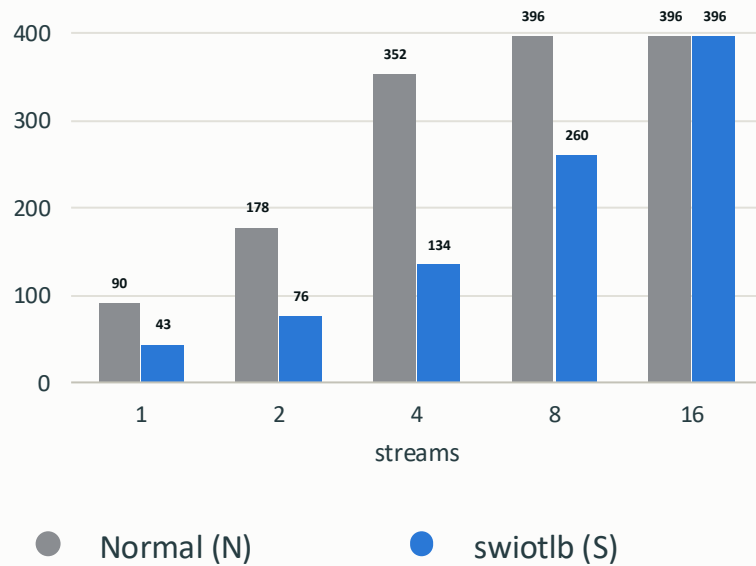


CPU usage (%)

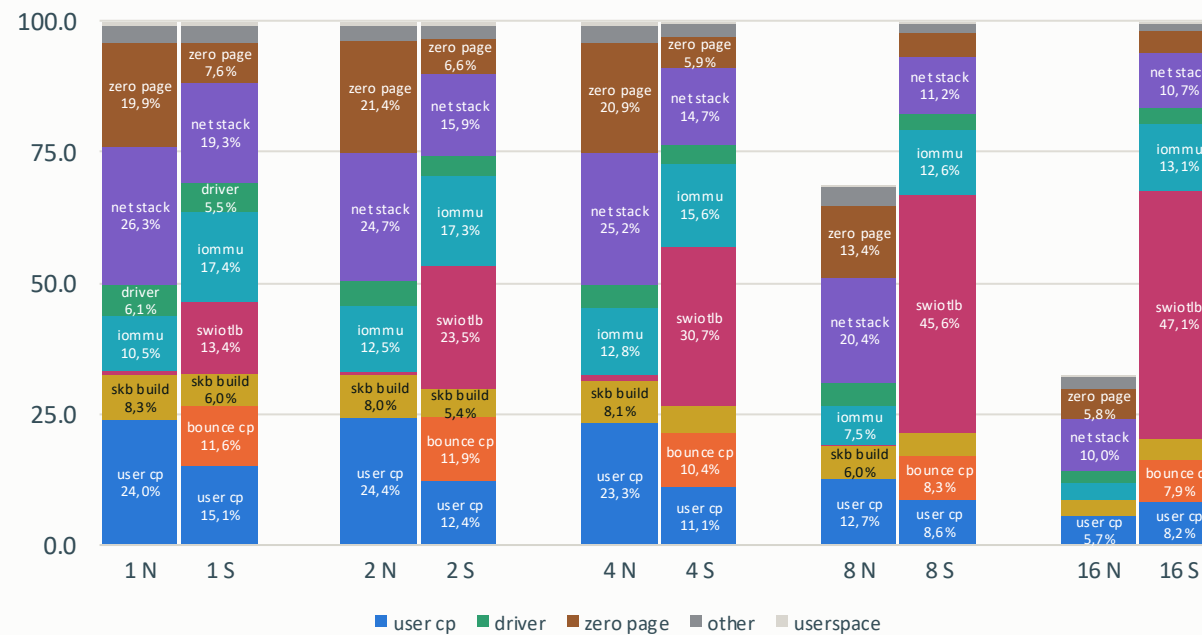


aarch64 TX at MTU 9000

Throughput (Gb/s)

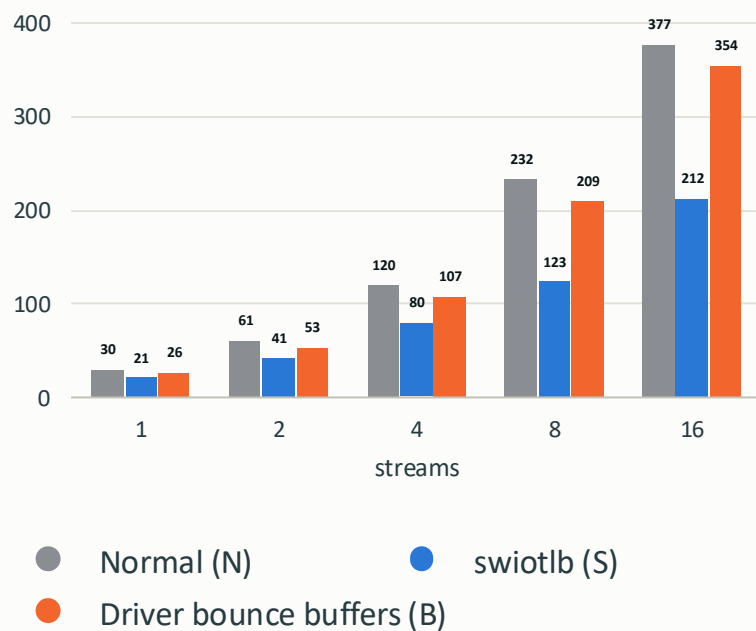


CPU usage (%)

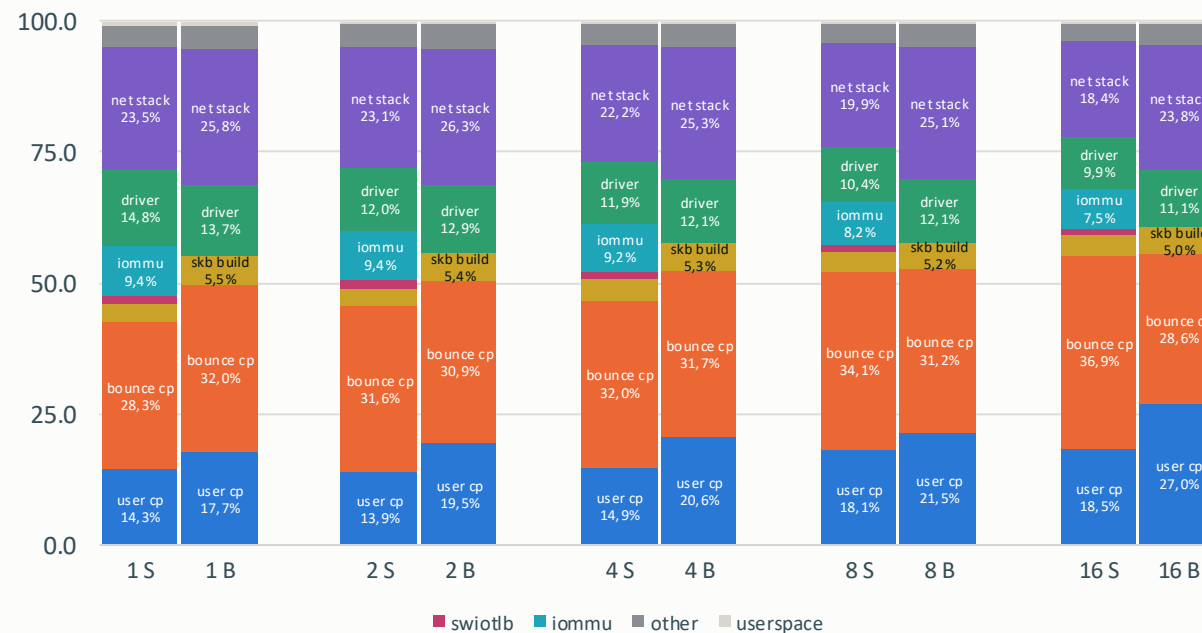


aarch64 RX at MTU 1500

Throughput (Gb/s)

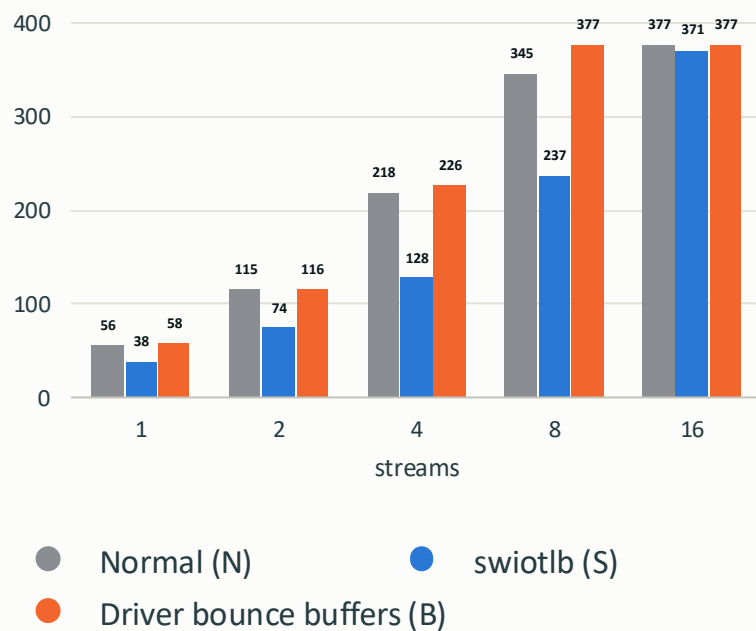


CPU usage (%)

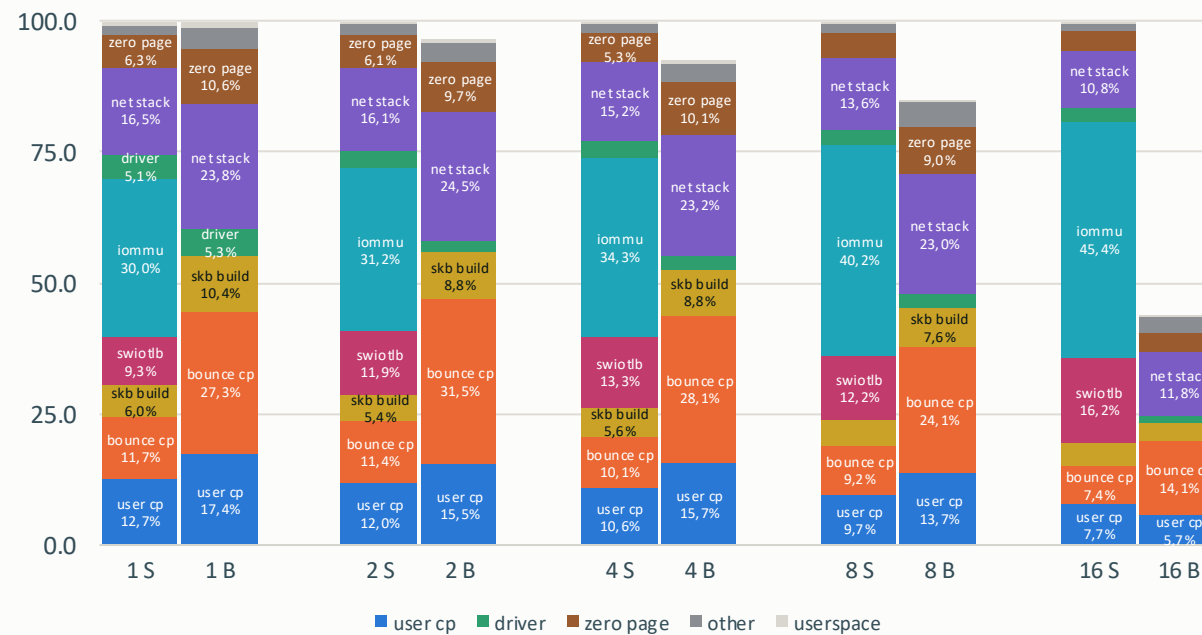


aarch64 TX at MTU 1500

Throughput (Gb/s)

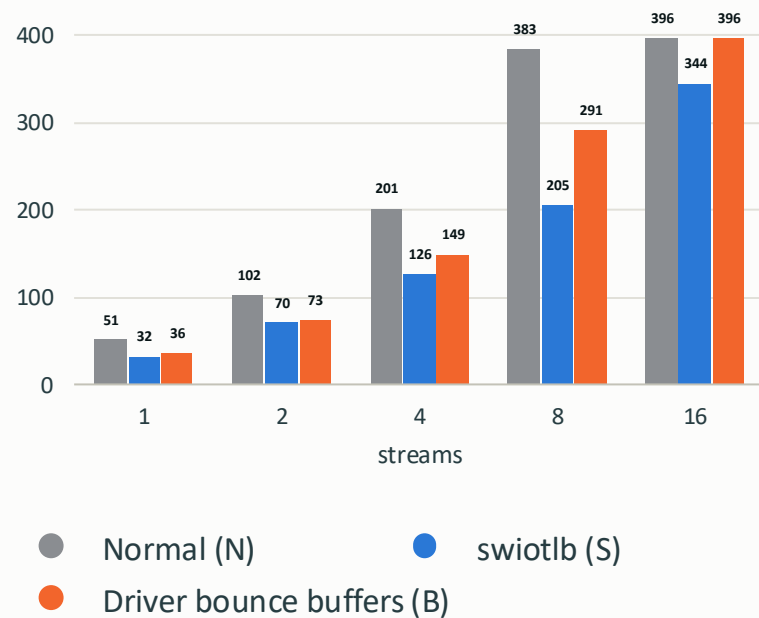


CPU usage (%)

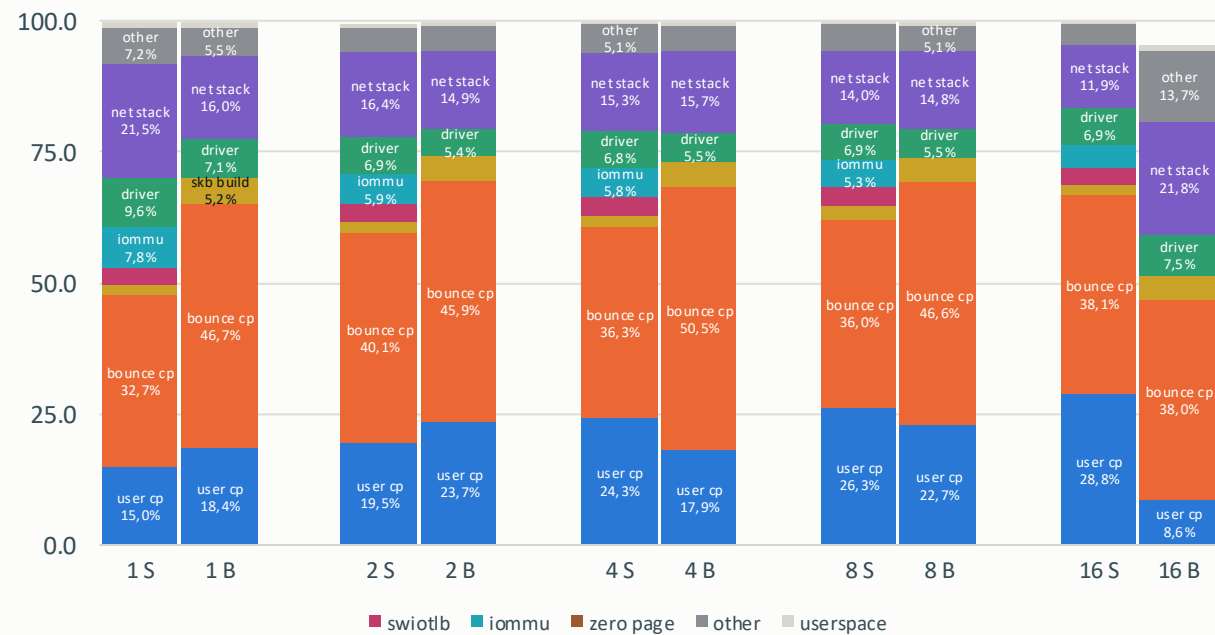


aarch64 RX at MTU 9000

Throughput (Gb/s)

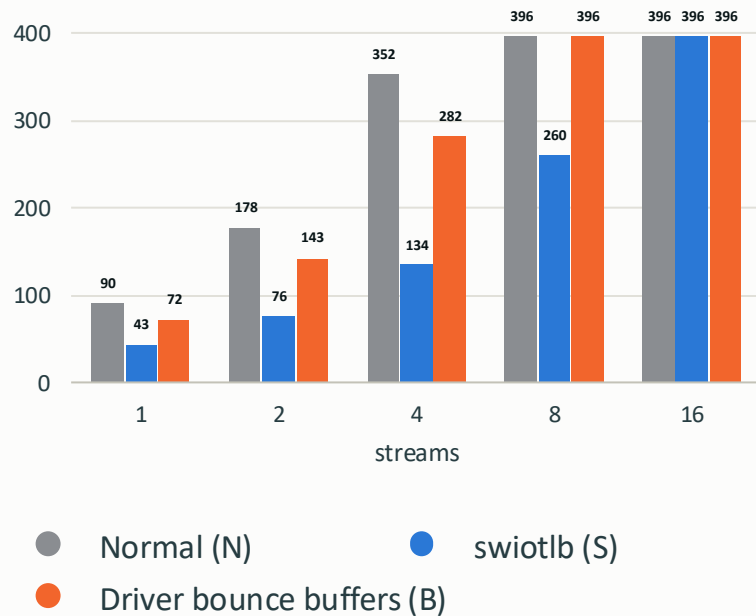


CPU usage (%)

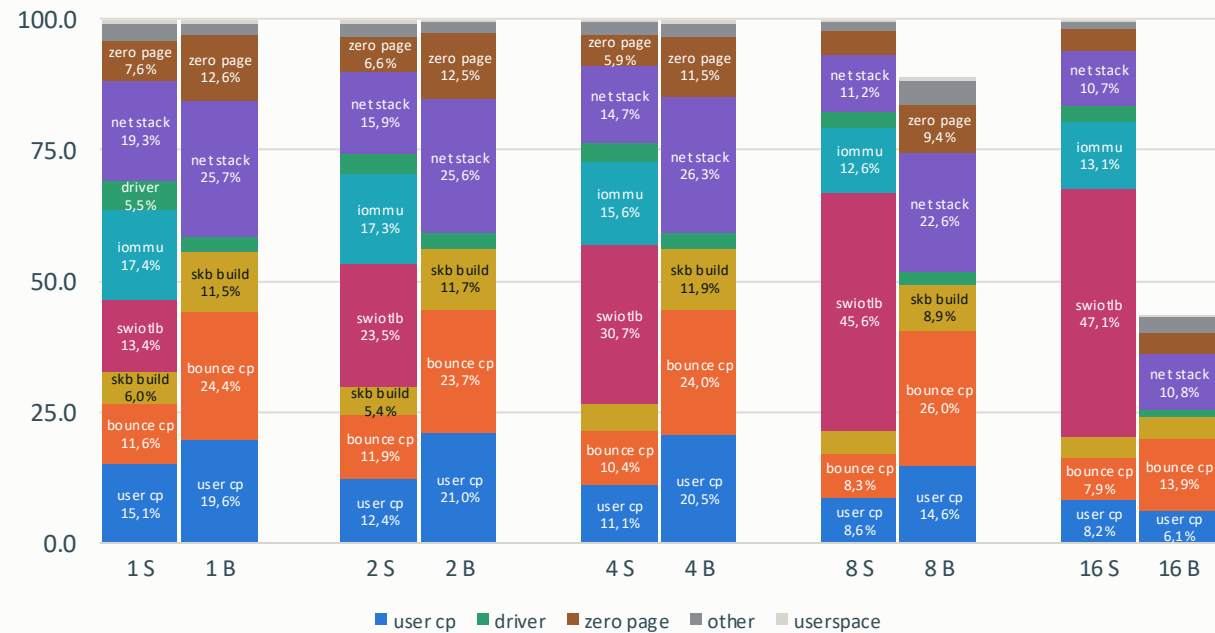


aarch64 TX at MTU 9000 (iommu lazy)

Throughput (Gb/s)



CPU usage (%)



Even worse

CoCo + swiotlb, MTU 9000, 16q, TX, aarch64

Overhead	Shared	Object	Symbol
47.79%	[kernel]	[k]	__swiotlb_find_pool
13.25%	[kernel]	[k]	_raw_spin_unlock_irqrestore
5.38%	[kernel]	[k]	__arch_copy_from_user
5.27%	[kernel]	[k]	__pi_memcpy_generic
4.96%	[kernel]	[k]	__pi_memset_generic
2.84%	[kernel]	[k]	__pi_clear_page
2.20%	[kernel]	[k]	swiotlb_find_slots
1.22%	[kernel]	[k]	__arm_lpae_iopte_walk
0.82%	[kernel]	[k]	_raw_spin_lock
0.69%	[kernel]	[k]	dst_release
0.68%	[kernel]	[k]	__arm_lpae_map
0.53%	[kernel]	[k]	swiotlb_bounce
0.43%	[kernel]	[k]	__iommu_dma_unmap
0.32%	[kernel]	[k]	mlx5e_xmit
0.31%	[kernel]	[k]	__nf_conntrack_find_get
0.30%	[kernel]	[k]	iommu_dma_unmap_phys
0.29%	[kernel]	[k]	mlx5e_napi_poll
0.28%	[kernel]	[k]	kmem_cache_free

Low memory, searching for other pools



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026