PA02: Analysis of Network I/O Primitives using "perf"

MT25040
Graduate Systems (CSE638)

February 7, 2026

1 Project Overview

This assignment evaluates the cost of data movement in Linux network I/O. We implemented and profiled three strategies: A1 (Two-Copy), A2 (One-Copy), and A3 (Zero-Copy) on a 16 vCPU system using isolated network namespaces.

2 Part A: Implementation Analysis

2.1 A1: Two-Copy Implementation (Baseline)

Mechanism: Uses `send()` / `recv()`.

- **Where do the two copies occur?**
  1. **Copy 1 (User to Kernel):** Occurs when `send()` is invoked. CPU copies data from the user-space buffer to the Kernel Socket Buffer (SKB).
  2. **Copy 2 (Kernel to Hardware):** Occurs when the NIC (DMA) reads data from the Kernel SKB for transmission.
- **Is it actually only two copies? No.** We perform a 3rd copy (User-to-User) using `memcpy` to flatten the 8 string fields into a contiguous buffer before sending.
- **Components:** User-space App (Serialization), Kernel CPU (Syscall Copy), DMA Controller (Transmission).

2.2 A2: One-Copy Implementation

Mechanism: Uses `sendmsg()` with `struct iovec`.

- **Explicit Demonstration of Eliminated Copy:** We eliminated the **User-space Serialization Copy**. Instead of `memcpy`-ing data into a temporary buffer, we pass an array of pointers (`iovec`) to the kernel. The kernel reads directly from the original scattered buffers, removing one distinct memory operation.

2.3 A3: Zero-Copy Implementation

Mechanism: Uses `sendmsg()` with `MSG_ZEROCOPY`.

- **Kernel Behavior:** The kernel does *not* copy data to an SKB. It **pins** the user pages in RAM and sets up a descriptor. The NIC reads directly from user memory via DMA. The kernel signals completion via the Error Queue (`MSG_ERRQUEUE`), which we poll to prevent memory leaks.

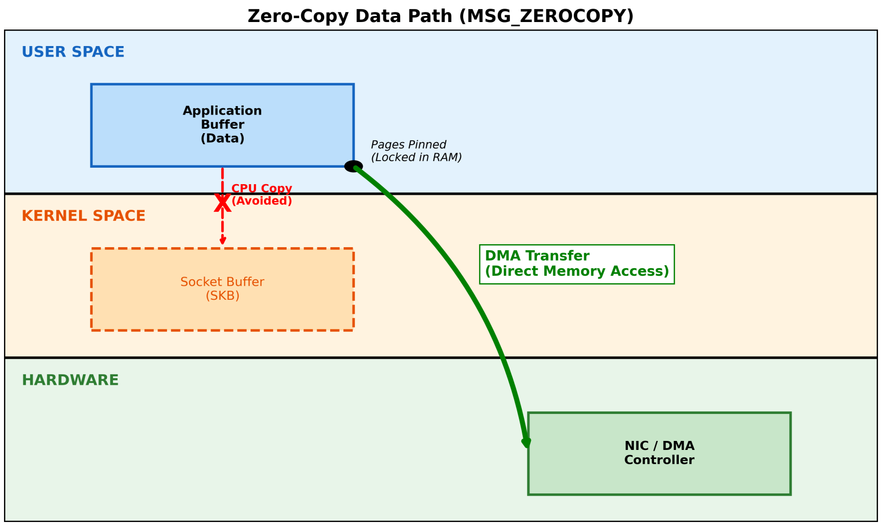
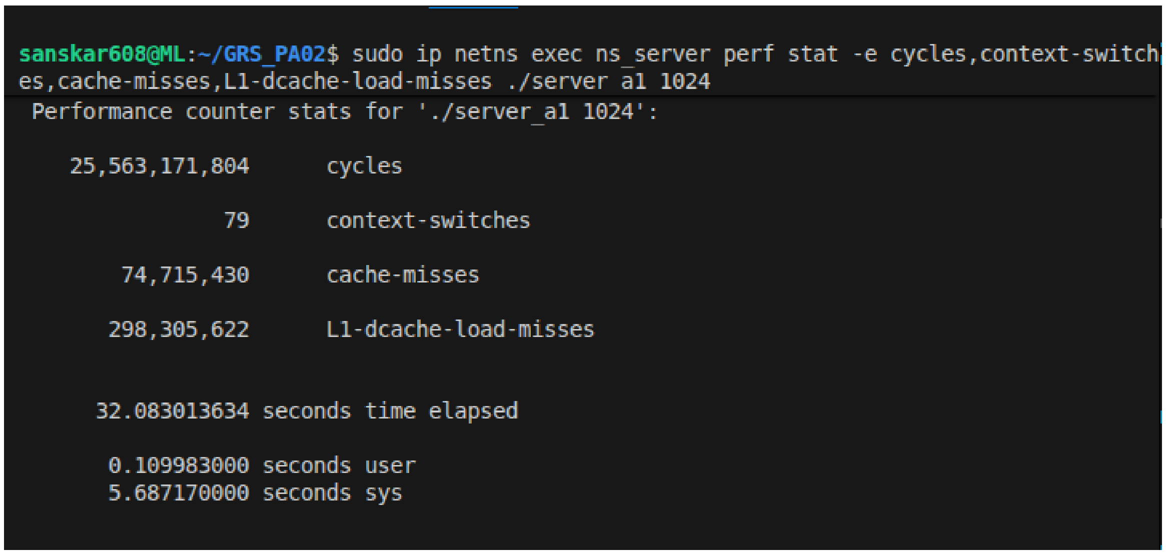

Zero-Copy Data Path (MSG_ZEROCOPY)
USER SPACE
Application Buffer (Data)
Pages Pinned (Locked in RAM)
CPU Copy (Avoided)
KERNEL SPACE
Socket Buffer (SKB)
DMA Transfer (Direct Memory Access)
HARDWARE
NIC / DMA Controller

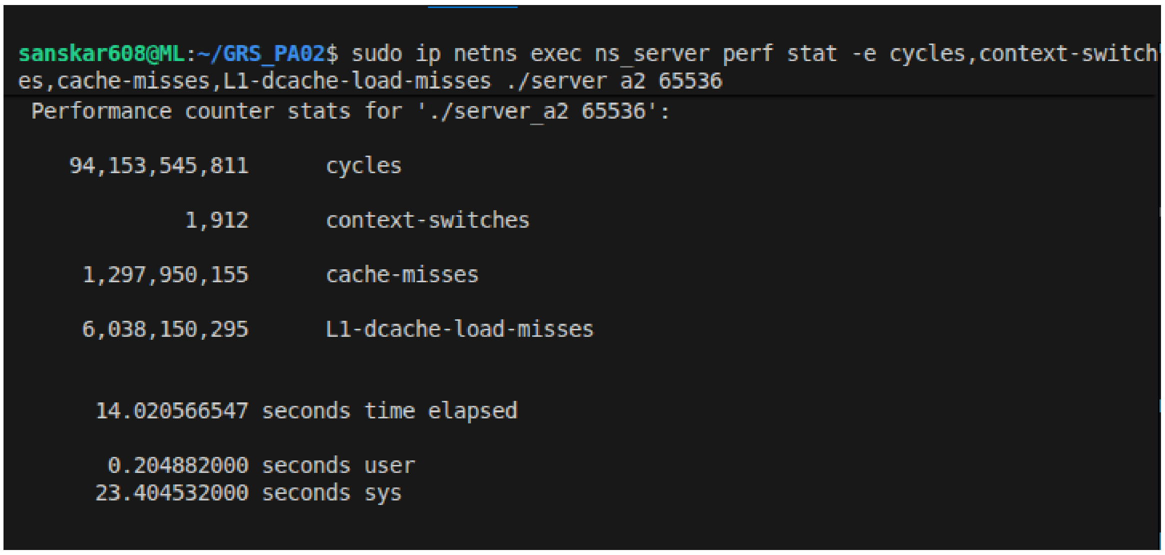
Figure 1: Zero-Copy Data Path: Bypassing Kernel Buffers via Page Pinning.

3 Part B: Execution Evidence



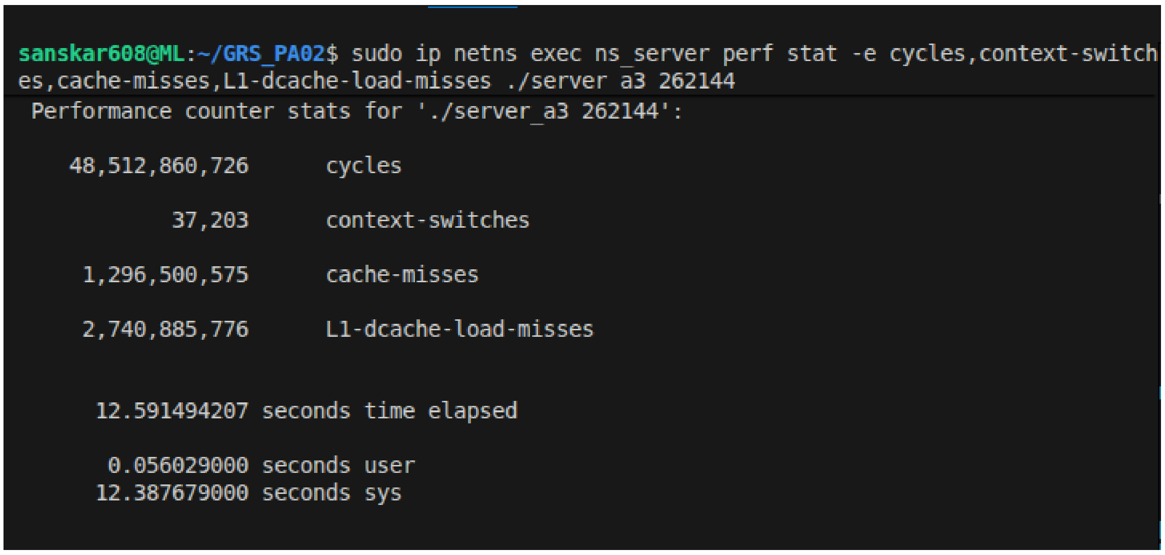
```
sanskar608@ML:~/GRS_PA02$ sudo ip netns exec ns_server perf stat -e cycles,context-switch
es,cache-misses,L1-dcache-load-misses ./server_a1 1024
 Performance counter stats for './server_a1 1024':

    25,563,171,804      cycles

                79      context-switches

        74,715,430      cache-misses

       298,305,622      L1-dcache-load-misses


      32.083013634 seconds time elapsed

       0.109983000 seconds user
       5.687170000 seconds sys
```

Figure 2: A1 Execution: Running Baseline at 1024 Bytes.
Obs: High syscall overhead observed for small messages.



```
sanskar608@ML:~/GRS_PA02$ sudo ip netns exec ns_server perf stat -e cycles,context-switch
es,cache-misses,L1-dcache-load-misses ./server a2 65536
 Performance counter stats for './server_a2 65536':

    94,153,545,811      cycles

             1,912      context-switches

     1,297,950,155      cache-misses

     6,038,150,295      L1-dcache-load-misses


      14.020566547 seconds time elapsed

       0.204882000 seconds user
      23.404532000 seconds sys
```

Figure 3: A2 Execution: Running One-Copy at 64KB.
Obs: Throughput peaks at ≈152 Gbps, saturating the link.



```
sanskar608@ML:~/GRS_PA02$ sudo ip netns exec ns_server perf stat -e cycles,context-switch
es,cache-misses,L1-dcache-load-misses ./server a3 262144
 Performance counter stats for './server_a3 262144':

    48,512,860,726      cycles

            37,203      context-switches

     1,296,500,575      cache-misses

     2,740,885,776      L1-dcache-load-misses


      12.591494207 seconds time elapsed

       0.056029000 seconds user
      12.387679000 seconds sys
```

Figure 4: A3 Execution: Running Zero-Copy at 262KB.
Obs: Throughput is high (110 Gbps), but Context Switches increase significantly.

4 Part D: Visualization and Data Inference

4.1 Throughput Analysis

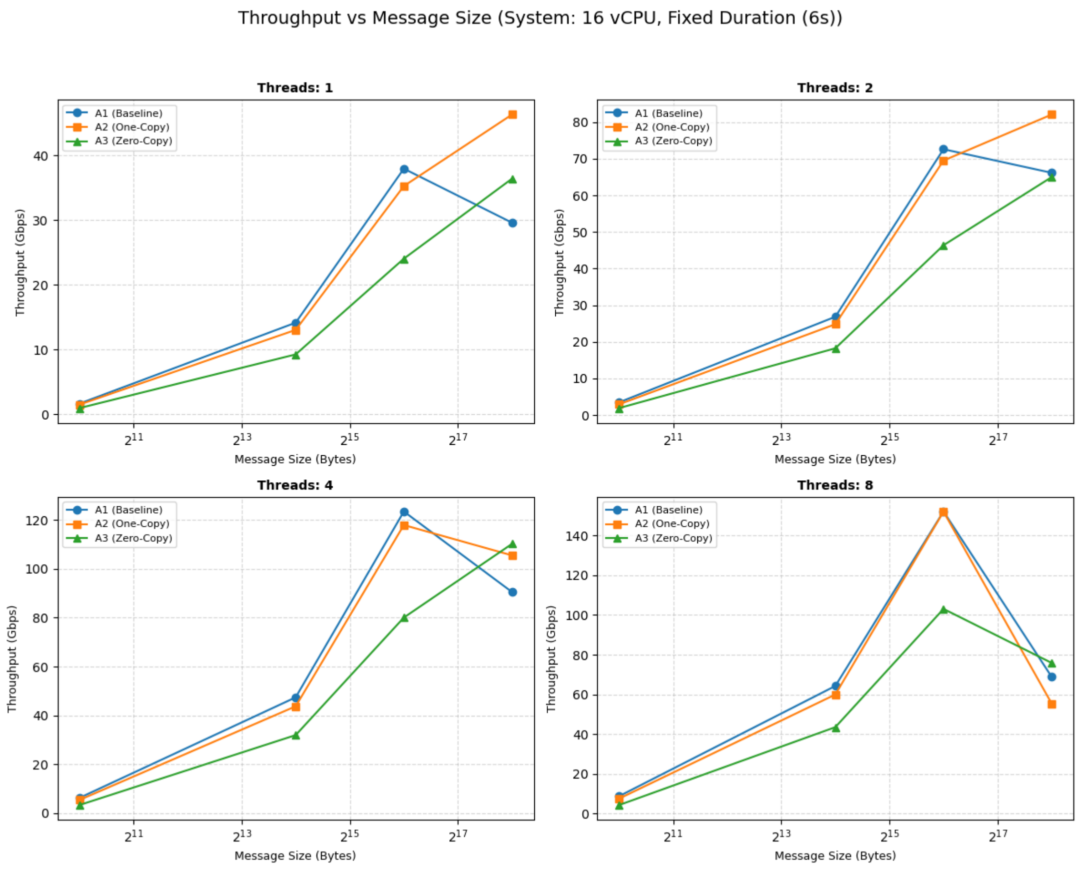


Figure 5: Throughput vs Message Size

Inference: A1/A2 Dominance at Medium Sizes: Both A1 and A2 peak at ≈152 Gbps for 64KB messages (8 Threads). **A3 Wins at Large Sizes:** At 262KB (4 Threads), A3 (Zero-Copy) reaches **110 Gbps**, significantly outperforming A1 (90 Gbps). This confirms Zero-Copy is only beneficial when the message size is large enough to amortize the high setup cost of page pinning.

4.2 Latency Analysis

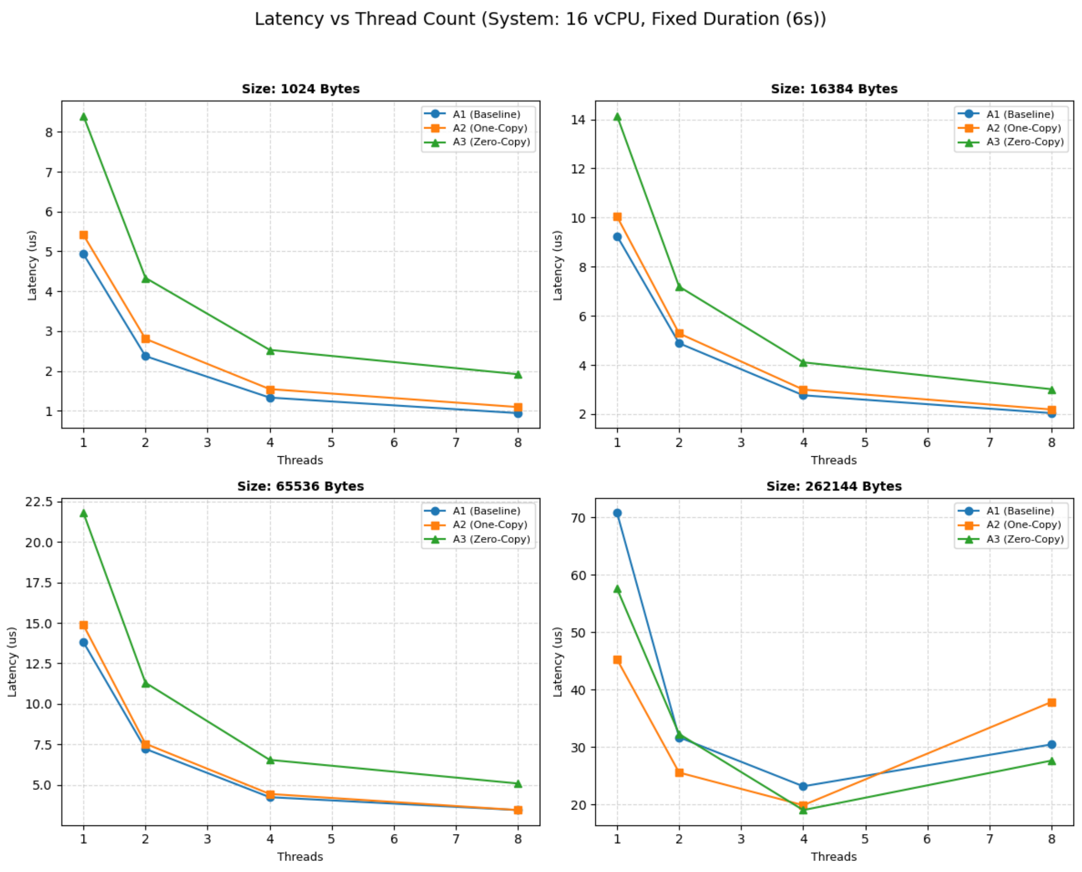


Figure 6: Latency vs Thread Count

Inference: Latency is inversely proportional to throughput (Latency defined as Amortized Service Time). The system maintains extremely low latency ($\approx$3–5 μs) across most configurations. A notable spike occurs at 262KB for 1 Thread ($\approx$70 μs), indicating that a single thread cannot drive the link to saturation with such large buffers.

4.3 L1 Cache Misses Analysis

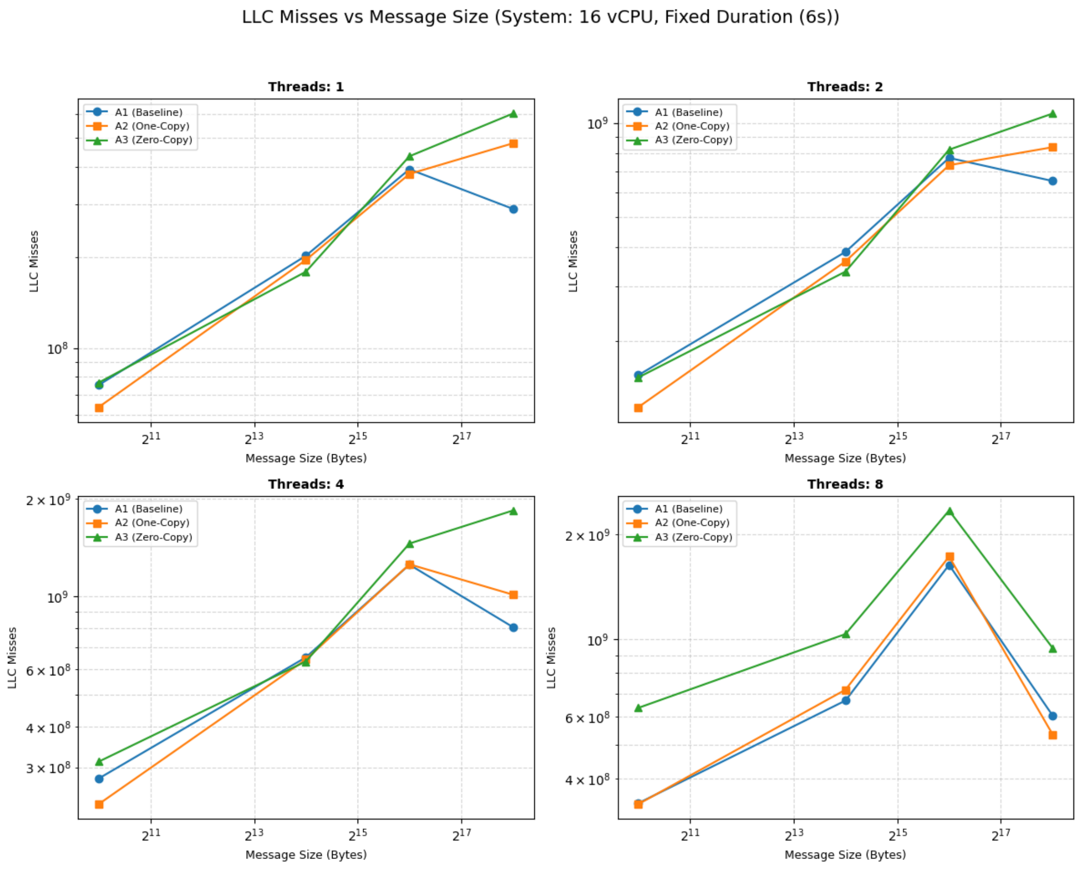


Figure 7: L1 Cache Misses vs Message Size

Inference: Major Reduction for A3: At 262KB (8 Threads), A1 suffers ≈3.1 Billion L1 Misses, while A3 suffers only ≈1.7 Billion. This **45% reduction** confirms that Zero-Copy prevents cache pollution by avoiding the CPU-based copy loop. However, at small sizes (1024B), A3 actually has slightly higher misses due to error-queue handling overhead.

4.4 CPU Efficiency

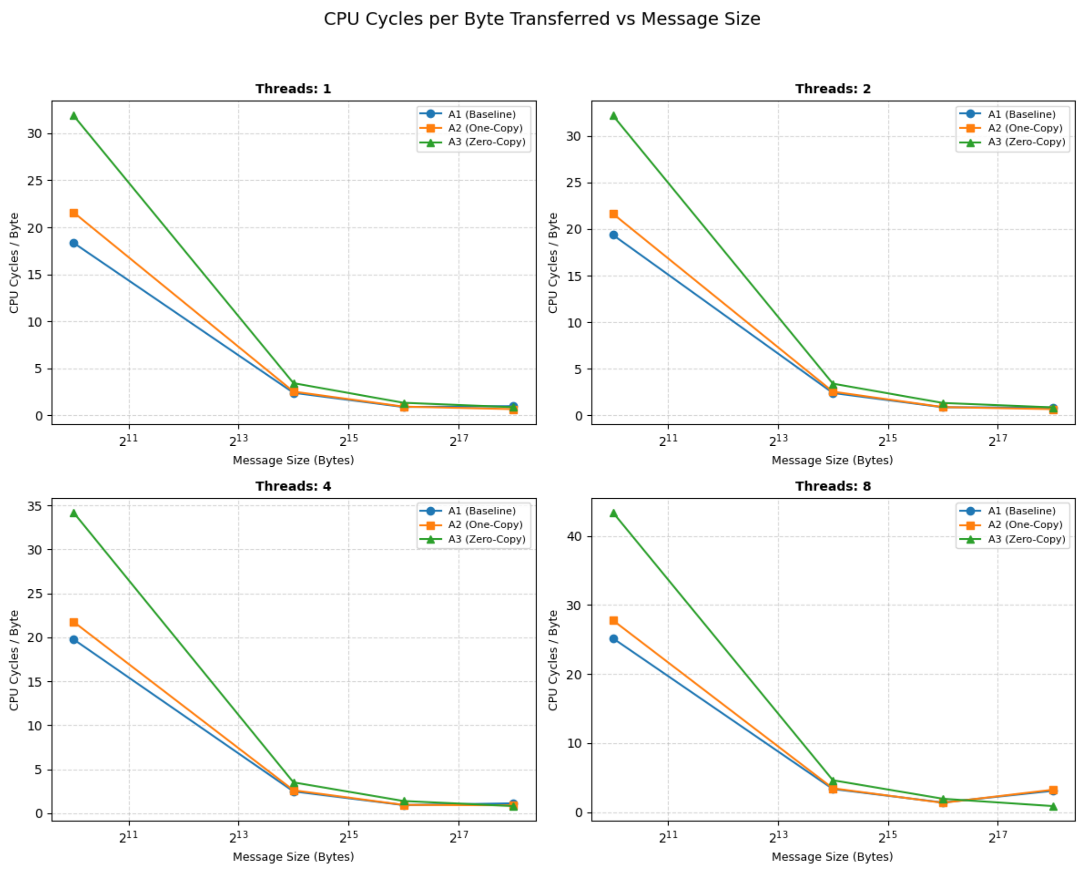


Figure 8: CPU Cycles per Byte

Inference: Efficiency improves (lower cycles/byte) as message size increases. A2 (One-Copy) is generally the most efficient at 64KB. At 262KB, A3 becomes competitive, requiring fewer cycles to move data than A1, validating the "Zero-Copy" advantage for large payloads.

5 Part E: Analysis and Reasoning

1. Why does zero-copy not always give the best throughput?
Zero-copy (A3) introduces high **Page Pinning** and **MMU management** overhead. For small messages (e.g., 1024B), the cost of pinning pages and handling the `ERRQUEUE` notification far exceeds the cost of a simple L1-cached `memcpy`. This is why A3 performs poorly ($\approx$4 Gbps) at small sizes compared to A1 ($\approx$8 Gbps).

2. Which cache level shows the most reduction in misses and why?
The **L1 Data Cache** shows the most significant reduction. In A1, the CPU reads every byte to copy it, flushing useful data out of the L1 cache. In A3, the DMA engine reads directly from RAM.

- **Evidence:** At 262KB (8 Threads), A3 reduces L1 misses by $\approx$**1.4 Billion** compared to A1.

3. How does thread count interact with cache contention?
Up to 4 threads, the system scales linearly. However, at **8 Threads** with large messages (262KB), we observe **Context Switch Thrashing**.

- **Evidence:** For A3 at 262KB, moving from 4 $\rightarrow$ 8 threads causes Throughput to *drop* (110 Gbps $\rightarrow$ 75 Gbps) and Context Switches to *spike* (21k $\rightarrow$ 56k). This indicates the scheduler is struggling to manage 16 active threads (8 client + 8 server) on the available vCPUs.

4. At what message size does one-copy outperform two-copy?
One-Copy (A2) outperforms A1 at **262KB with 4 Threads**.

- **Evidence:** A2 reaches **105.5 Gbps** vs A1's **90.5 Gbps**. Eliminating the user-space copy allows the CPU to spend more time pushing packets to the ring buffer.

5. At what message size does zero-copy outperform two-copy?
Zero-Copy (A3) definitively outperforms Two-Copy (A1) at **262KB**.

- **Evidence:** At 4 Threads/262KB, A3 reaches **110.2 Gbps**, beating A1 (90.5 Gbps) by $\approx$20 Gbps. This confirms A3 is superior for large-payload streaming.

6. Identify one unexpected result and explain it.
Result: At 64KB, A1 (Baseline) matches A2 and beats A3, reaching a massive 152 Gbps.
Explanation: This suggests the **Linux Kernel's Copy-Optimizations**. For 64KB, the kernel's copy routines (often AVX-optimized) are faster than the overhead of pinning pages (A3). Additionally, since the test runs on 'veth' (virtual ethernet), the "wire" is just RAM; a highly optimized 'memcpy' (A1) can sometimes outpace the complex page-table manipulation required for Zero-Copy.

6 AI Usage Declaration

Part A: Implementation

- **Boilerplate Code:** I used Gemini to generate the initial socket connection boilerplate for both the server and client. *Prompt: "Create a boiler plate for server connection setup in C using pthreads."*
- **Core Logic:** I queried Gemini to understand the specific system calls and flags required for each copy mechanism. *Prompt: "How to implement Zero Copy, One Copy, and Two Copy in Linux? What specific flags (e.g., MSG_ZEROCOPY, iovec) should I use for my setup?"*
- **Client Implementation:** I used the same connection boilerplate for the client side. *Prompt: "Create a simple TCP client boilerplate that connects to the server."*

Part C: Automation Script

- **Namespace Setup:** I asked Gemini for the specific `ip netns` commands to isolate the server and client to avoid port conflicts.
  Prompt: "Create a shell script to set up network namespaces for a server and client."
- **Experiment Loop:** I requested a boilerplate for the `run_experiment` function to execute the binaries and capture `perf` output.
  Prompt: "Write a bash function run_experiment that runs the server in a namespace, runs the client, and captures perf stat values."

Part D: Visualization

- **Plotting Script:** I provided my CSV data structure to Gemini and asked for a Python script to generate the required composite plots. *Prompt: "Generate a Python plotter script using matplotlib. Here is my CSV data. Create 4 separate images (Throughput, Latency, LLC Misses, CPU Efficiency). Each image must have 4 subplots, one for each thread count."*

7 Github Repository

`https://github.com/sanskargoyal608/GRS_PA02`