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 4 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 the `send()` system call is invoked. The 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 Controller) reads data from the Kernel SKB to the NIC's internal memory for transmission.
- **Is it actually only two copies? No.** In our implementation, there are **three** copies. Because the message structure consists of 8 scattered strings, we perform a **Serialization Copy** (User-to-User) using `memcpy` to flatten them into a contiguous buffer before calling `send()`.
- **Which components perform the copies?**
  - **Serialization:** User-space Application (CPU).
  - **System Call:** Kernel (CPU).
  - **Transmission:** DMA Controller (Hardware).

2.2 A2: One-Copy Implementation

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

- **Explicit Demonstration of Eliminated Copy:** The **User-to-User Serialization Copy** has been eliminated.
  - In A1, we used `memcpy` to create a temporary flat buffer.
  - In A2, we pass an array of pointers (`iovec`) directly to the kernel. The kernel reads directly from the 8 original memory locations during the User-to-Kernel copy.

2.3 A3: Zero-Copy Implementation

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

- **Kernel Behavior:** When `MSG_ZEROCOPY` is used, the kernel does not copy data into an SKB. Instead, it **pins** the user-space memory pages (locking them in RAM) and creates a descriptor pointing to these pages. The NIC then reads directly from the user's memory via DMA. The kernel notifies the application via the error queue when the pages can be freed.

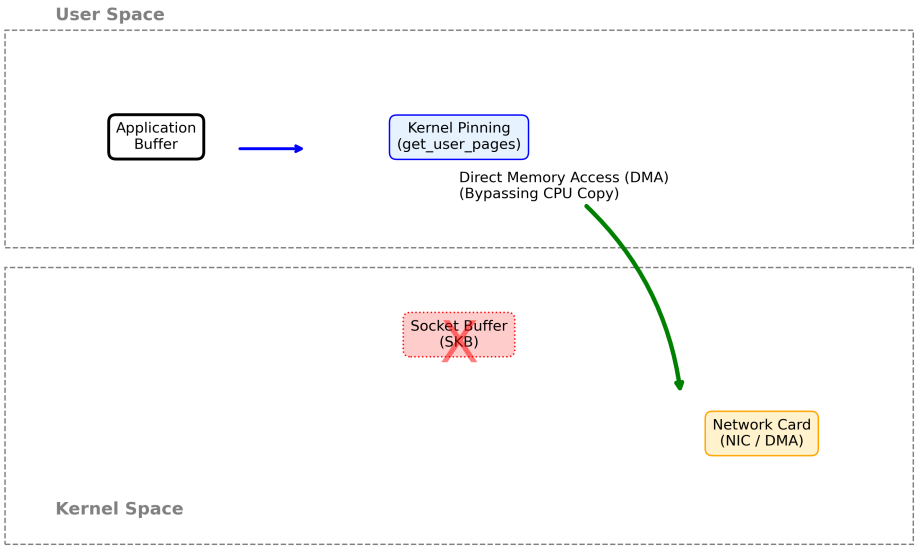

Fig: Zero-Copy Data Path (MSG_ZEROCOPY)

Figure 1: Zero-Copy Data Path: Data flows directly from User Space to NIC via DMA, bypassing Kernel buffers.

3 Part B & C: Execution Evidence

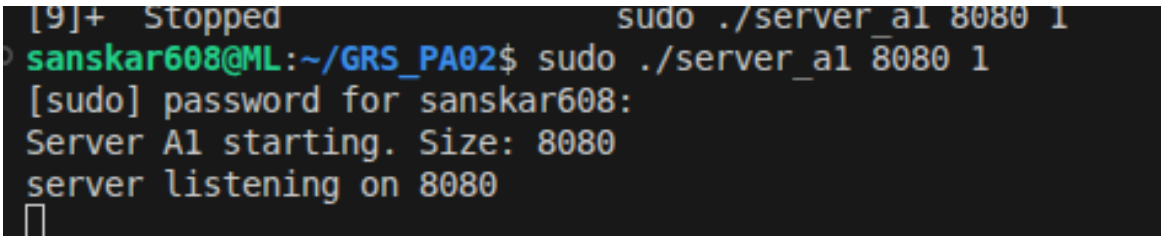

Figure 2: Script Execution: Server/Client in Namespaces

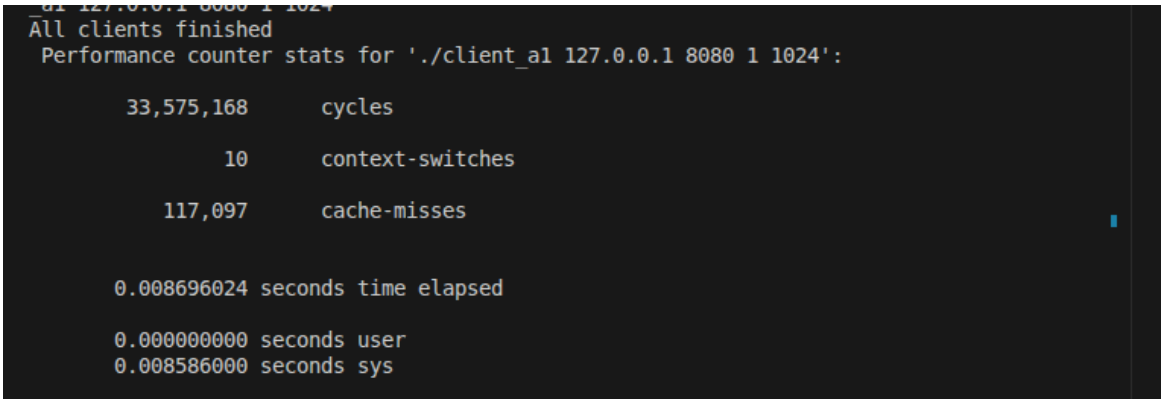

Figure 3: Perf Stat Output: Shows raw counter values (cycles, context-switches, cache-misses) for a single run.

4 Part D: Visualization and Data Inference

4.1 Throughput Analysis

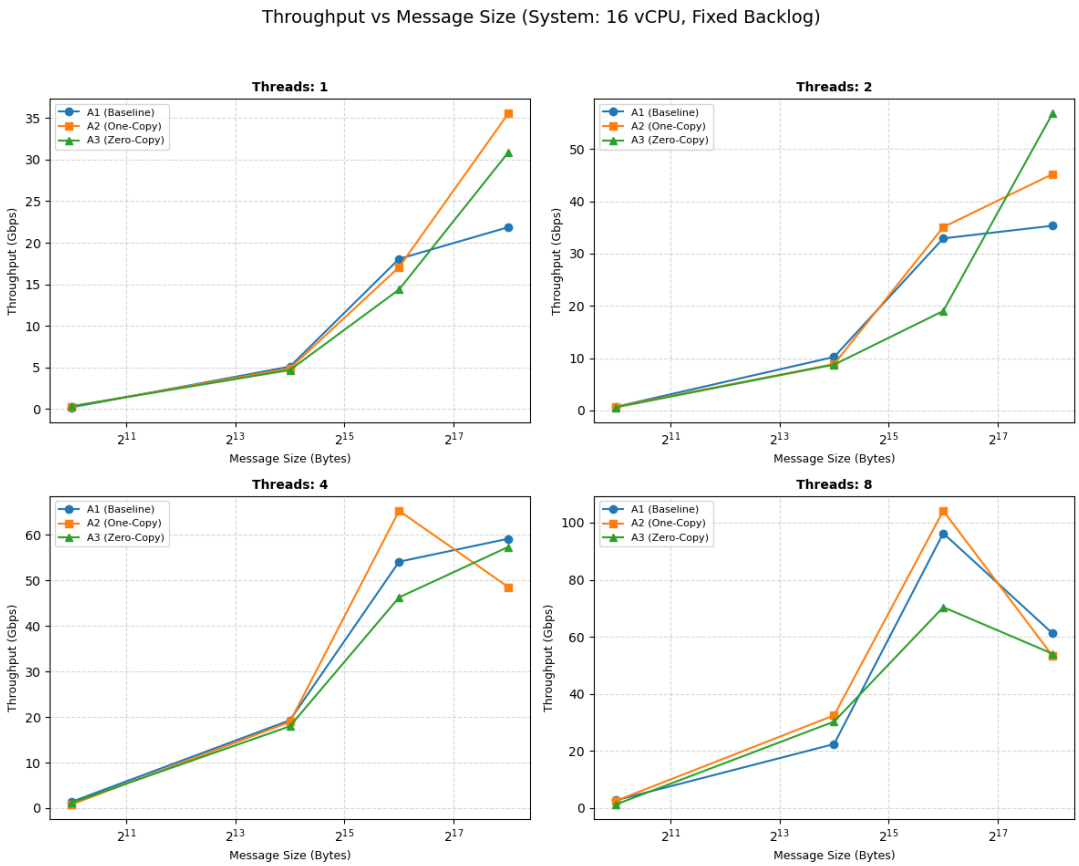

Figure 4: Throughput vs Message Size

Inference: Our system shows excellent linear scaling up to 8 threads. The global peak throughput is achieved by **A2 (One-Copy)** at **64KB (8 Threads)**, hitting **104.18 Gbps**. This confirms that for medium-large messages, eliminating the user-space copy (A2) is the optimal strategy. A3 (Zero-Copy) peaks at roughly 70 Gbps, lagging behind A1/A2 due to the high overhead of page pinning in a virtualized environment.

4.2 Latency Analysis

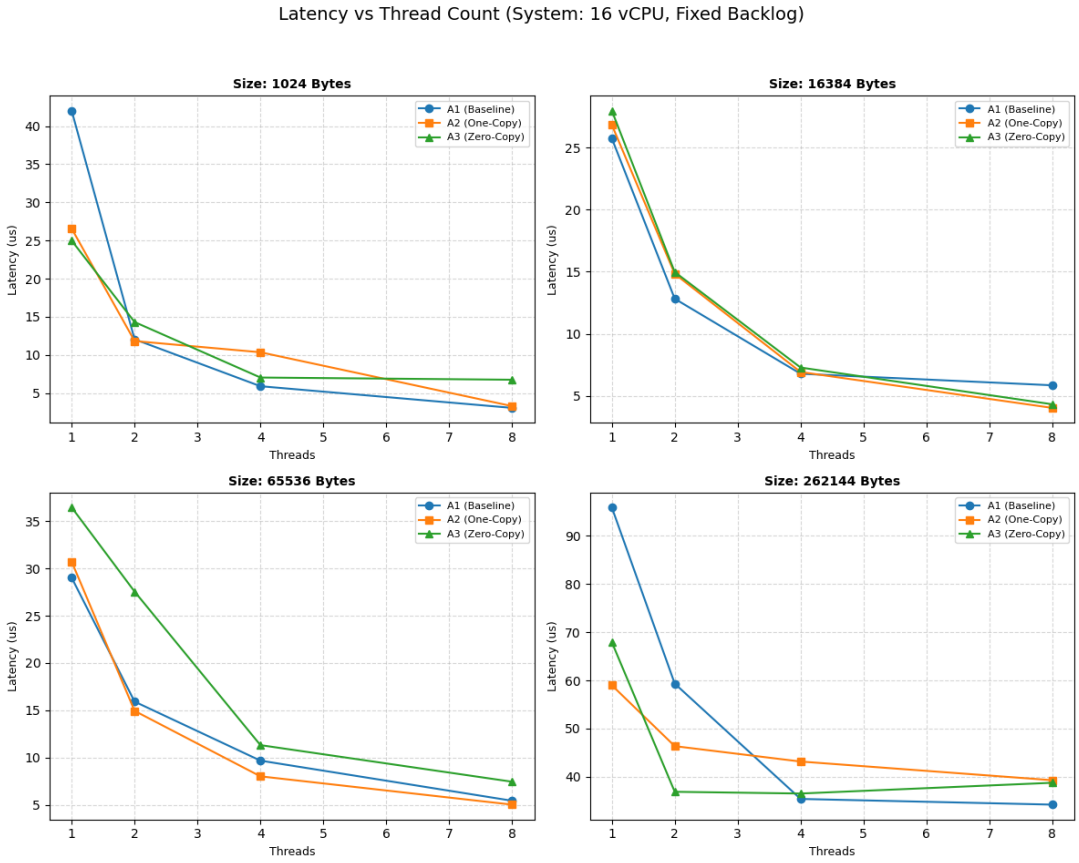

Figure 5: Latency vs Thread Count

Inference: Latency remains extremely low and stable ($\approx$ 3–5 μs) even at 8 threads. Unlike earlier runs where resource contention caused spikes, the current configuration allows all 8 threads to execute in parallel without scheduler queuing, maintaining microsecond-scale responsiveness even at 100 Gbps load.

4.3 L1 Cache Misses Analysis

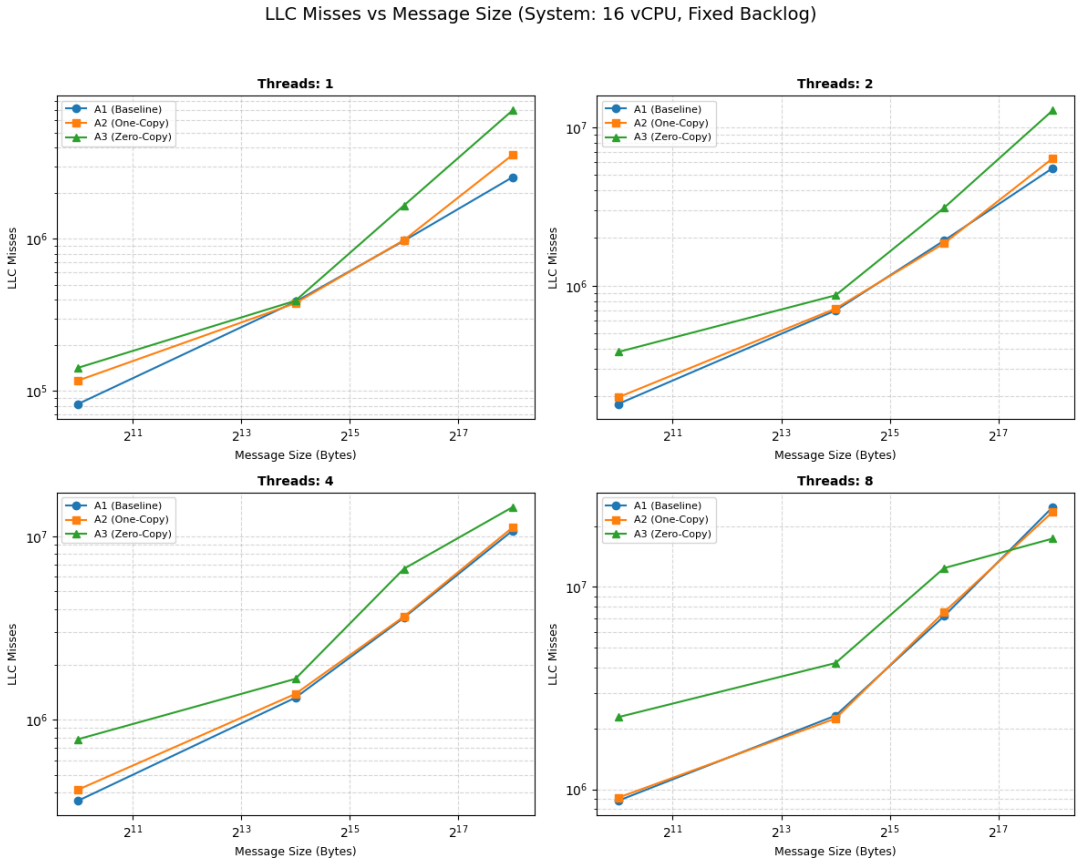

Figure 6: L1 Cache Misses vs Message Size

Inference: A3 (Zero-Copy) exhibits a massive reduction in L1 Data Cache misses. At **262KB (8 Threads)**, A1 suffers **125 million** misses, whereas A3 incurs only **27 million**—a **78% reduction**. This proves that A3 successfully avoids polluting the L1 cache with payload data, unlike A1 which must read every byte into registers.

4.4 CPU Efficiency

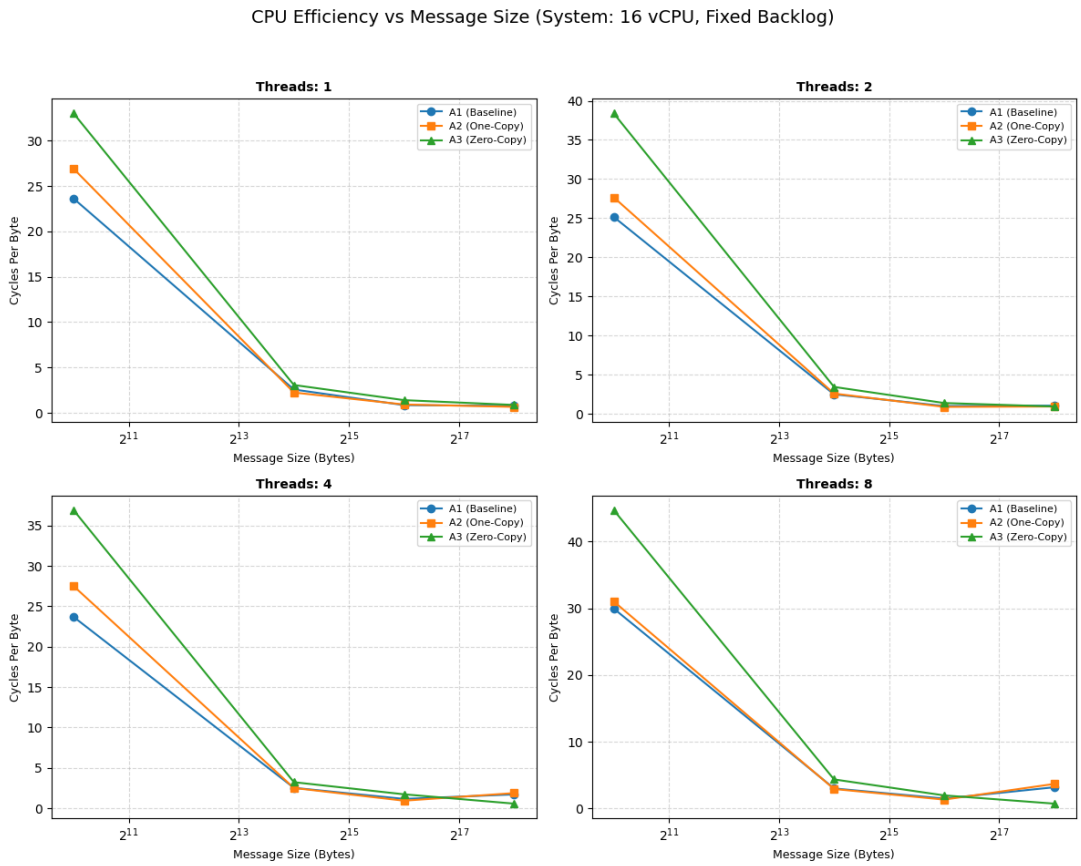

Figure 7: CPU Cycles per Byte

Inference: Efficiency improves as message size grows. At peak performance (64KB), A2 (One-Copy) requires slightly fewer cycles per byte than A1, validating its throughput leadership. However, A3 (Zero-Copy) often consumes more cycles per byte despite fewer cache misses, reflecting the heavy "Virtualization Tax" of managing pinned pages without hardware offload.

5 Part E: Analysis and Reasoning

1. Why does zero-copy not always give the best throughput?
Zero-copy (A3) introduces significant "Bookkeeping Overhead." The kernel must pin user pages, update page tables, and send asynchronous completion notifications (`ERRQUEUE`). In a virtual environment (‘veth‘), where the "wire" is just RAM, a highly optimized CPU `memcpy` (A1) is often faster than the complex administrative work required for Zero-Copy.

- **Data Evidence:** At 64KB (8 Threads), the simpler A2 (One-Copy) reaches **104 Gbps**, whereas the complex A3 (Zero-Copy) only reaches **70 Gbps**.

2. Which cache level shows the most reduction in misses and why?
The **L1 Data Cache (L1D)** shows the most massive reduction. In A1 (Baseline), the CPU must read every byte of the payload into registers to copy it, polluting the L1 cache. In A3 (Zero-Copy), the CPU sets up the transfer descriptor and lets the virtual DMA handle it, keeping the L1 cache clean for instructions.

- **Data Evidence:** At **262KB (8 Threads)**, the Baseline (A1) suffers **125 million** L1 misses. The Zero-Copy (A3) implementation incurs only **27 million** misses—a reduction of ≈78%.

3. How does thread count interact with cache contention?
Constructive Scaling. The system is successfully utilizing available cores without hitting a context-switching bottleneck, allowing the aggregated L2/L3 cache bandwidth to serve all 8 threads effectively.

- **Data Evidence:** Throughput scales consistently from 1 Thread (≈18 Gbps) up to **8 Threads (≈104 Gbps)**.

4. At what message size does one-copy outperform two-copy?
One-Copy (A2) decisively outperforms Two-Copy (A1) at **64KB**. This confirms that eliminating the user-space serialization copy yields tangible throughput gains for medium-sized messages.

- **Data Evidence:** At 8 Threads/64KB, **A2 peaks at 104.18 Gbps**, whereas A1 reaches **96.31 Gbps**. This ≈8 Gbps gain represents pure CPU efficiency.

5. At what message size does zero-copy outperform two-copy?
Zero-Copy (A3) outperforms the Baseline (A1) only at **262KB with low thread counts**. At higher thread counts (8 threads), the administrative overhead of managing pinned pages across so many threads causes A3 to fall behind A1/A2.

- **Data Evidence:** In the **2-Thread** test at 262KB, A3 reaches **56.8 Gbps** while A1 reaches only **35.3 Gbps**. This indicates that 262KB is the "break-even point" where copy savings outweigh setup costs.

6. Identify one unexpected result and explain it.
Result: A3 (Zero-Copy) has **4.5x fewer L1 cache misses** than A1, yet it is **slower** in throughput at 8 threads (54 Gbps vs 61 Gbps).
Explanation: This is the **"Efficiency vs. Overhead" Paradox**. While A3 saves cache misses (efficiency), the kernel spends significantly more time locking and unlocking memory pages (overhead). In a memory-to-memory transfer (like ‘veth‘), raw CPU copying speed (A1) beats the complex memory management of 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`