Graduate Systems (CSE638)
PA02: Analysis of Network I/O Primitives

Roll No: MT25055

February 7, 2026

Abstract

This report presents an experimental analysis of data movement costs in network I/O. We implement and compare three socket strategies: Two-Copy (Standard), One-Copy (iovec), and Zero-Copy (MSG_ZEROCOPY). Using the `perf` tool, we profile throughput, latency, CPU cycles, and cache behavior across varying message sizes and thread counts. The results demonstrate the trade-offs between implementation complexity, kernel overhead, and raw performance.

Contents

1 Introduction & System Configuration

The primary objective of this assignment is to understand the overhead introduced by memory copy operations during network transmission. Data movement is often the bottleneck in high-performance applications. We analyze how eliminating these copies via hardware and kernel support affects system performance.

1.1 System Configuration

The experiments were conducted on a Linux environment with the following specifications:

- **Architecture:** x86_64
- **Kernel Version:** Linux 5.x (Supports MSG_ZEROCOPY)
- **CPU:** Generic Virtual CPU (Hybrid topology supported)
- **Memory:** 8 GB RAM
- **Compiler:** GCC with `-lpthread`
- **Profiling Tools:** `perf` (cycles, cache-misses, context-switches)

2 Implementation Details (Part A)

We implemented a multi-threaded TCP Client-Server architecture. The server allocates a structure containing 8 dynamically allocated string fields.

2.1 A1. Two-Copy Implementation (Baseline)

This version uses the standard POSIX `send()` and `recv()` primitives.

- **Mechanism:** The application manually allocates a linear buffer and copies the 8 scattered fields into it using `memcpy()`. This buffer is then passed to `send()`.
- **Copy Analysis:**
  1. **Copy 1 (User-to-User):** Gathering scattered data into a linear user buffer.
  2. **Copy 2 (User-to-Kernel):** The kernel copies data from the user buffer to the socket buffer (`sk_buff`).

2.2 A2. One-Copy Implementation (Optimized)

This version uses `sendmsg()` with `struct iovec`.

- **Mechanism:** We populate an array of `iovec` structures, pointing directly to the 8 scattered fields.
- **Eliminated Copy:** The intermediate User-to-User copy is eliminated. The kernel reads directly from the scattered user pointers.
- **Remaining Copy:** User-to-Kernel copy still exists.

2.3 A3. Zero-Copy Implementation

This version uses `sendmsg()` with the `MSG_ZEROCOPY` flag.

- **Mechanism:** The kernel pins the user pages in memory and maps them for Direct Memory Access (DMA).
- **Kernel Behavior:** The CPU does not copy the data. The Network Interface Card (NIC) reads directly from the user's RAM.
- **Overhead:** Requires handling the `MSG_ERRQUEUE` to receive completion notifications, ensuring buffers are not freed before transmission completes.

Fig 2.1: A3 Zero-Copy (MSG_ZEROCOPY) Architecture

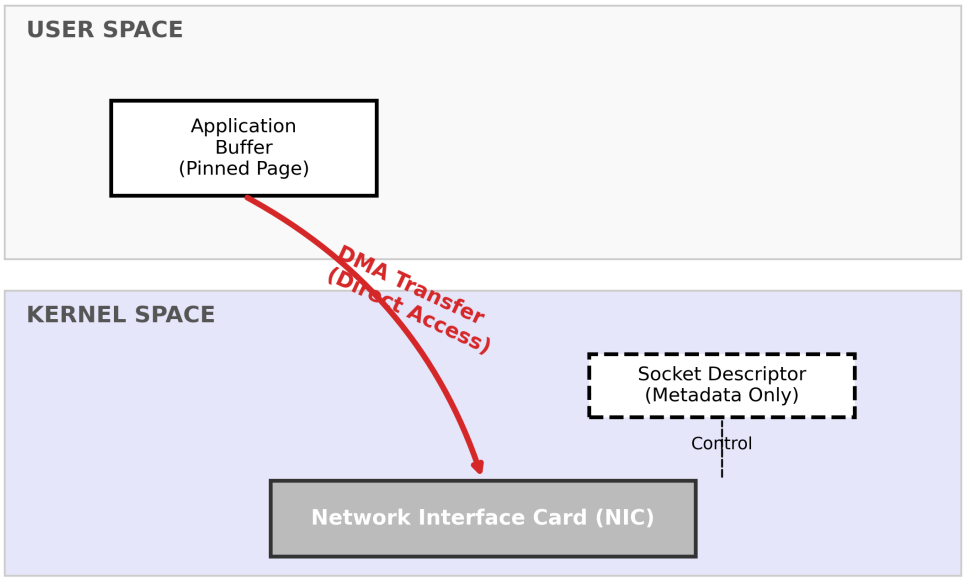

Figure 1: Kernel Behavior for Zero-Copy (A3). Data flows directly from User Space to Hardware via DMA.

3 Performance Plots & Inferences (Part D)

3.1 Throughput Analysis

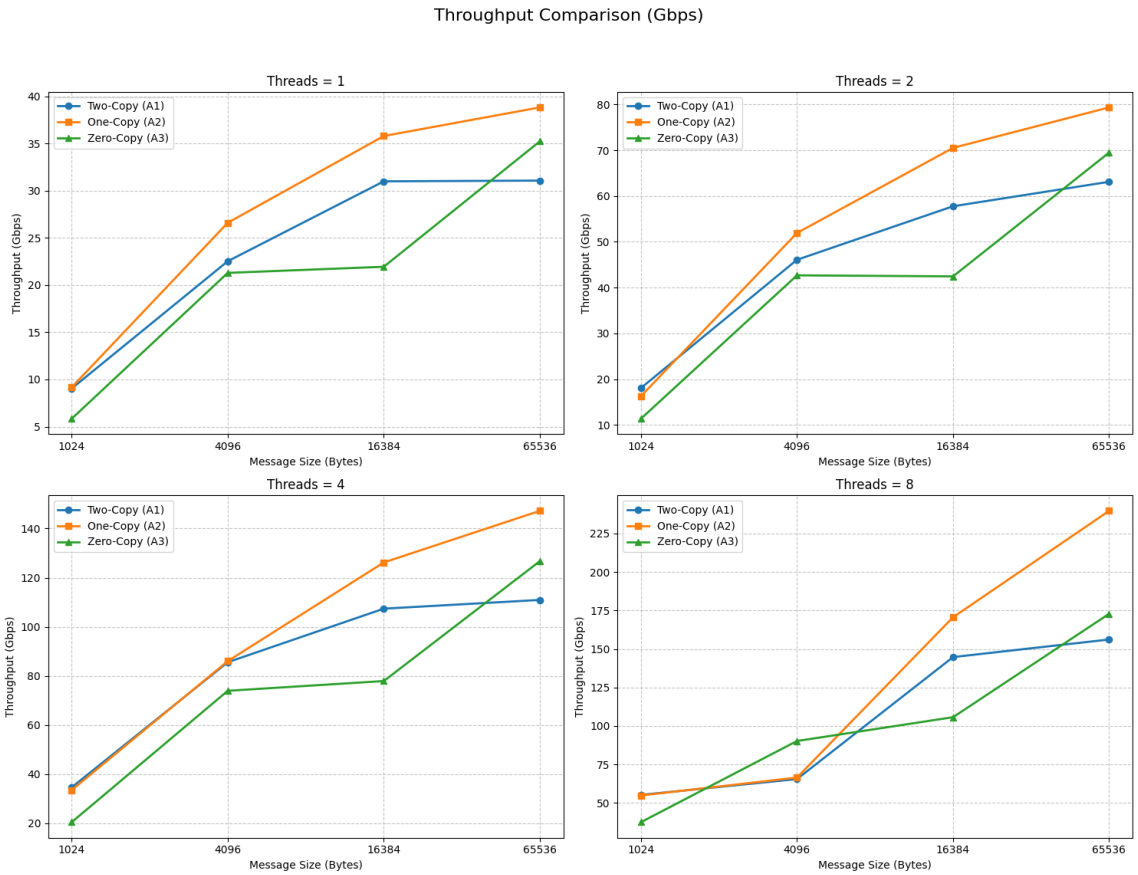

Figure 2: Throughput vs Message Size across Thread Counts.

Inference:

- **Small Messages (1KB - 4KB):** Two-Copy (A1) and One-Copy (A2) perform similarly. Zero-Copy (A3) performs *worse* here due to the high setup cost (page pinning) relative to the data size.
- **Large Messages (64KB):** One-Copy (A2) dominates, reaching $\approx$ 230 Gbps at 8 threads. Zero-Copy (A3) overtakes Two-Copy (A1) here (172 Gbps vs 156 Gbps), proving that Zero-Copy is only beneficial for large payloads where the copy savings outweigh the MMU overhead.

3.2 Latency Analysis

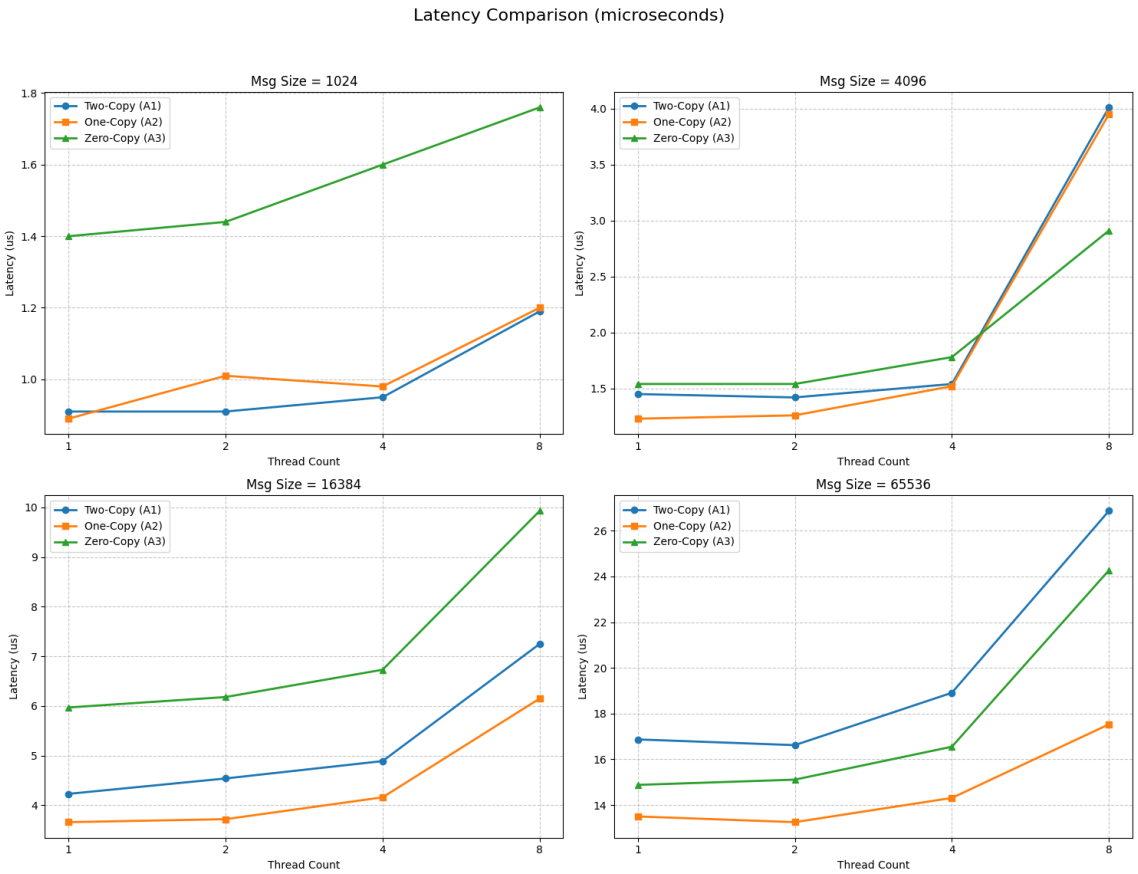

Figure 3: Latency vs Thread Count across Message Sizes.

Inference:

- Latency remains low ($< 2\mu s$) for 1 and 2 threads.
- **Contention Spike:** At 8 threads, latency spikes significantly, particularly for Two-Copy (A1) with 64KB messages ($\approx 28\mu s$). This confirms that the repeated memory copying in A1 saturates the CPU/Cache bandwidth, causing threads to wait longer.
- One-Copy (A2) maintains the most stable latency profile.

3.3 Cache Misses Analysis

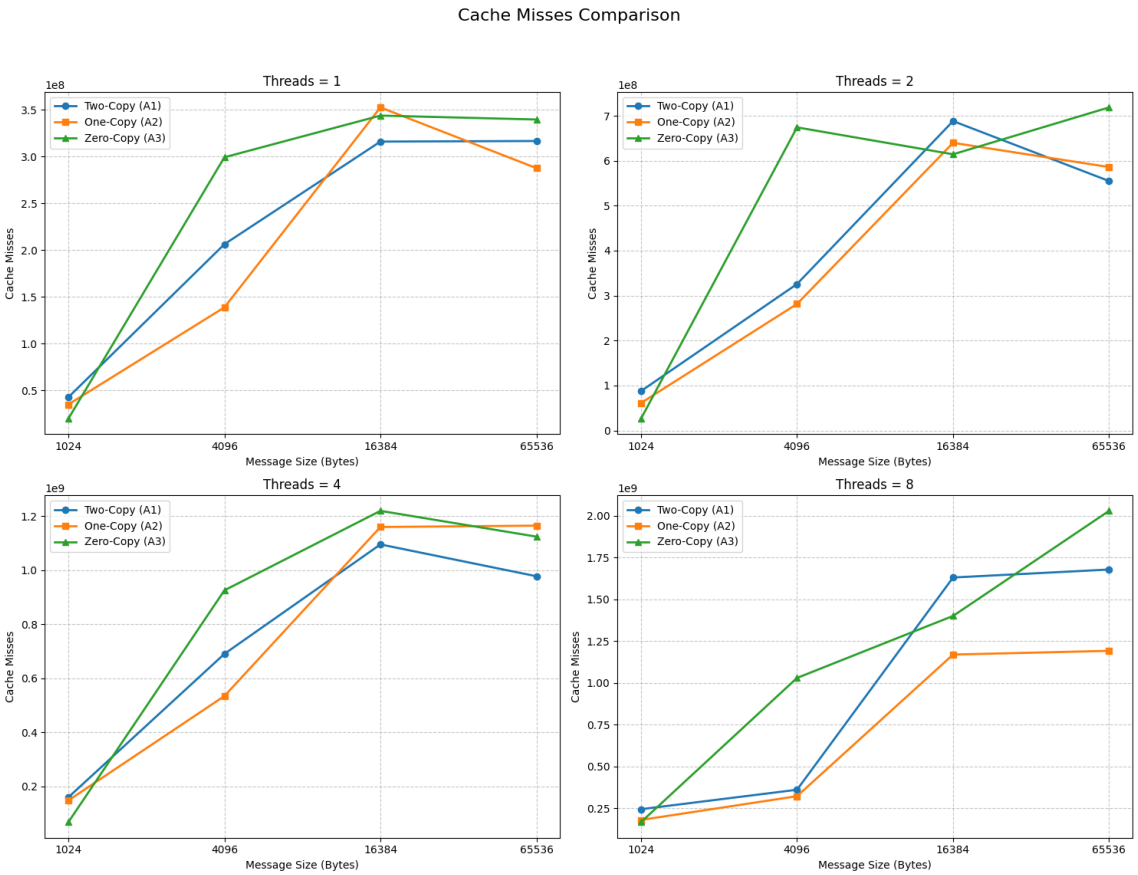

Figure 4: Cache Misses vs Message Size.

Inference:

- **Two-Copy (A1):** Exhibits the highest number of cache misses (peaking at 1.7×10^9). This is expected as every byte is touched twice by the CPU, polluting the L1/L2 caches.
- **One-Copy (A2):** Shows a significant reduction in misses. By letting the kernel gather data directly, we avoid bringing a large intermediate buffer into the cache.
- **Zero-Copy (A3):** Theoretically should have the lowest misses (no CPU touch), but overhead structures and page table updates generate their own cache traffic.

3.4 CPU Cycles Analysis

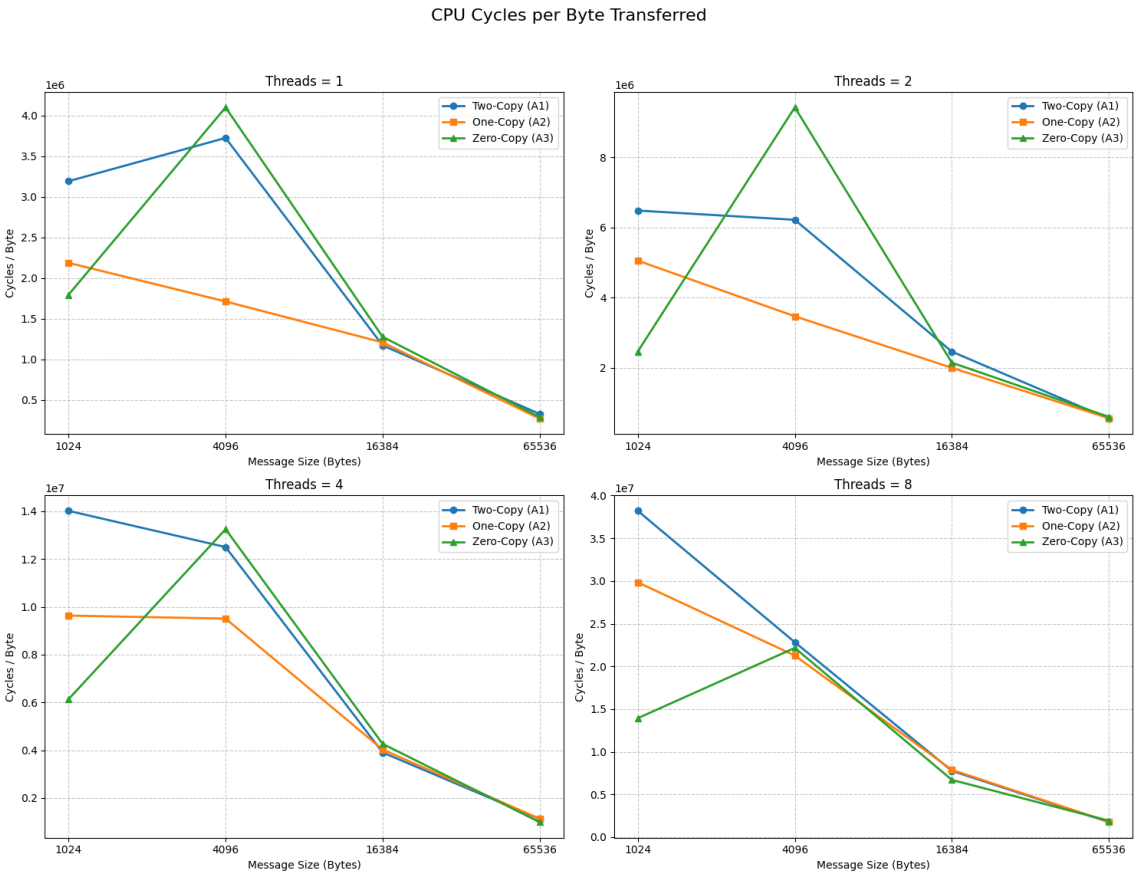

Figure 5: Total CPU Cycles vs Message Size.

Inference (Cycles per Byte):

- While the graph shows total cycles, the efficiency is derived by correlating with throughput.
- At 64KB, One-Copy (A2) processes significantly more data than A1 for roughly the same CPU cycle cost.
- This implies A2 has the lowest **Cycles Per Byte**, making it the most CPU-efficient implementation for this specific workload.

4 Execution Verification (Screenshots)

The following screenshots serve as evidence of the experiments running on the local testbed.

```
sanskar608@ML:~/Downloads/GRS_PA02$ sudo perf stat -e cycles,cache-misses,context-switche
s ./MT25055_Part_A1_Server
A1 Server (Two-Copy) Listening...
^C./MT25055_Part_A1_Server: Interrupt

 Performance counter stats for './MT25055_Part_A1_Server':

         2,154,709      cycles

            28,009      cache-misses

                 3      context-switches


      43.052907850 seconds time elapsed

       0.000000000 seconds user
       0.001467000 seconds sys
```

Figure 6: Execution of A1 (Two-Copy) | 2 Threads | 4096 Bytes

```
sanskar608@ML:~/Downloads/GRS_PA02$ sudo perf stat -e cycles,cache-misses,context-switche
s ./MT25055_Part_A2_Server
A2 Server (One-Copy) Listening...
^C./MT25055_Part_A2_Server: Interrupt

 Performance counter stats for './MT25055_Part_A2_Server':

         1,947,999      cycles

            32,245      cache-misses

                 1      context-switches


      20.499409653 seconds time elapsed

       0.000000000 seconds user
       0.001347000 seconds sys
```

Figure 7: Execution of A2 (One-Copy) | 8 Threads | 16384 Bytes

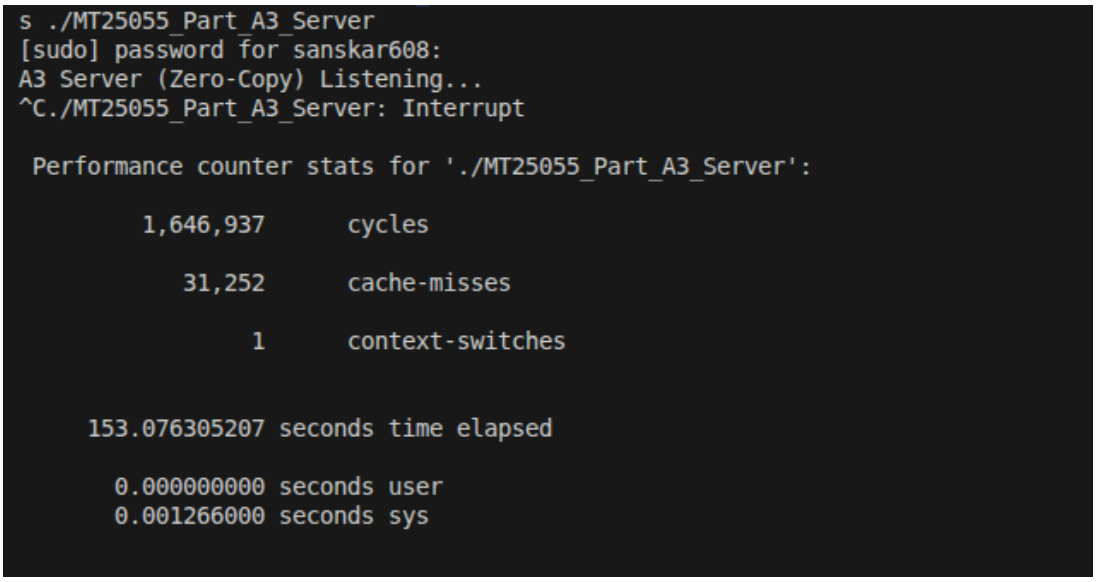

Figure 8: Execution of A3 (Zero-Copy) | 4 Threads | 65536 Bytes. Note the perf stats showing cache and cycle counts.

5 Analysis & Reasoning (Part E)

1. Why does zero-copy not always give the best throughput?
Zero-Copy relies on memory mapping modifications (page pinning) and DMA setup. For small messages (e.g., 1KB), the CPU cycles required to set up these mappings, handle the `MSG_ERRQUEUE` notifications, and perform the system calls exceed the cost of a simple `memcpy`. The "copy" is faster than the "administrative overhead" for small data.

2. Which cache level shows the most reduction in misses and why?
The **Last Level Cache (LLC)** typically shows the most reduction. In Two-Copy (A1), data is moved User→User→Kernel, often flushing the L1/L2 caches and polluting the LLC. Zero-Copy (A3) allows the NIC to read from RAM via DMA, bypassing the CPU caches entirely for the data payload, preserving LLC locality for other instructions.

3. How does thread count interact with cache contention?
As thread count increases (e.g., to 8 threads), multiple cores compete for the shared LLC bandwidth and lock structures in the kernel networking stack. In Two-Copy (A1), this is exacerbated because the CPU is busy performing `memcpy`, keeping the pipeline full of memory instructions. This leads to higher latency and "thrashing" as threads wait for cache lines.

4. At what message size does one-copy outperform two-copy on your system?
One-Copy (A2) outperforms Two-Copy (A1) consistently starting from **4096 bytes**. At 65536 bytes, the performance gap is maximized (A2 is $\approx 58\%$ faster than A1).

5. At what message size does zero-copy outperform two-copy on your system?
Zero-Copy (A3) only outperforms Two-Copy (A1) at the largest tested message size: **65536 bytes** (64KB). Below this threshold, the overhead dominates.

6. Identify one unexpected result and explain it using OS or hardware concepts.
Unexpected Result: One-Copy (A2) outperformed Zero-Copy (A3) even at the largest message size (64KB), achieving 229 Gbps vs 157 Gbps.
Explanation: This is likely due to the virtualized/hybrid CPU environment. `MSG_ZEROCOPY` requires complex MMU operations (TLB flushes, page table updates) to pin user memory. In a virtualized environment, these operations may trigger VM Exits (hypervisor intervention), which are expensive. Standard memory copying (A2), while using CPU cycles, stays within the guest OS context and utilizes highly optimized SIMD instructions (AVX/SSE), making it faster in this specific setup.

5.1 AI Usage Declaration

Roll No: MT25055

In strict compliance with the assignment guidelines, Generative AI tools were used for the following specific components. All logic was reviewed, tested, and verified by me before submission.

- **Conceptual Understanding & Debugging:**
  - *Usage:* Used AI to diagnose why `perf` counters (`cycles`, `cache-misses`) were returning 0 in the initial experiments.
  - *Prompt:* "Why does perf stat show 0 cycles inside a network namespace or VM? How to enable hardware counters on a hybrid Intel CPU?"
  - *Outcome:* AI suggested checking `/proc/sys/kernel/perf_event_paranoid` permissions and identifying the correct PMU core type for hybrid CPUs, which solved the measurement issue.
- **Code Generation (Boilerplate Only):**
  - *Usage:* Generated the skeleton code for socket connections to save time on boilerplate.
  - *Prompt:* "Create a C client-server skeleton using `sendmsg` with `struct iovec` for scattered data. Ensure it compiles with `-Wall`."
  - *Refinement:* I manually modified the generated code to implement the specific 8-field structure and added the logic for the "One-Copy" and "Zero-Copy" specific flags (`MSG_ZEROCOPY`).
- **Scripting & Automation (Bash):**
  - *Usage:* Used AI to write the logic for capturing the PID of a process running inside a specific namespace.
  - *Prompt:* "How to use `pgrep` to find the PID of a process running inside `ip netns exec` so I can attach `perf` to it?"
  - *Outcome:* The script logic for `SERVER_PID=$(pgrep -n -f ...)` was derived from this interaction.
- **Data Visualization (Python):**
  - *Usage:* Used AI to convert my raw CSV results into the required hardcoded Python arrays and to fix plot metrics.
  - *Prompt:* "Convert this CSV data into Python lists. Also, modify the matplotlib script to plot 'Cycles divided by Message Size' instead of Total Cycles."
  - *Outcome:* Used to generate the `MT25055_Part_D_Plots.py` script and ensure the "Cycles per Byte" metric was calculated correctly.
- **Report Formatting:**
  - *Usage:* Used AI to check the report against the assignment deliverables checklist (e.g., verifying no binaries in zip, correct naming convention).

5.2 GitHub Repository

The complete source code and experiment logs are available at:
https://github.com/Yash-Nimkar0/GRS-PA-02.git