PA02: Analysis of Network I/O primitives using “perf” tool

Graduate Systems (CSE638)

Submitted by:

Adarsh Singh

Roll Number: PhD25001

IIIT Delhi

Feb 7, 2026

Table of Contents

1. Introduction
2. Part A: Program Implementation
3. Part B & C: Measurement Script and Methodology
4. Part D: Plotting and Visualization
5. Part E: Analysis and Reasoning
6. AI Usage Declaration
7. Conclusion

1. Introduction

This assignment let us understand the micro-architectural costs of data movement in network I/O. By implementing and profiling three distinct TCP socket communication methods -Two-Copy, One-Copy (Scatter-Gather), and Zero-Copy (Page-Pinning)- we evaluate their impact on throughput, latency, CPU cycles, and cache efficiency. The study highlights how kernel-user interactions and memory management strategies significantly influence performance across varying message sizes and thread counts.

1.1 System Configuration

The experiments were conducted on the following setup:

- **Server:** Intel B760M-Pro RS-D4 (Kernel 6.8.0)
  (ip address : 192.168.226.224)

- **Client:** Linux Workstation (Kernel 5.15.0)
  (ip address: 192.168.23.48)

- **Tools:** perf (for CPU/Cache profiling), taskset (for affinity), and custom bash automation scripts.

2. Part A: Program Implementation

2.1 A1: Two-Copy Implementation (Baseline)

The baseline implementation uses standard POSIX socket primitives. The "Two-Copy" nomenclature refers to the data movement path required to send a message composed of multiple disjoint buffers.

Explanation Details

Structure: The server constructs a message from 8 dynamically allocated string fields.

- **Copy 1 (User-Space Aggregation):** The application manually allocates a contiguous buffer and uses memcpy() to aggregate the 8 disjoint fields into this single buffer. This is a User-to-User copy.
- **Copy 2 (Kernel-Space Transfer):** The application calls send(). The kernel copies the data from the user-space contiguous buffer into the kernel's socket buffer (SKB). This is a User-to-Kernel copy.

Answer:

- **Where do the two copies occur?**
    - User Space: Aggregating 8 scattered fields into one linear buffer.
    - Kernel Boundary: Copying from the linear user buffer to the kernel socket buffer (SKB) during the send() syscall.
- **Which components perform the copies?**
    - Copy 1: Performed by the CPU (User Mode) via memcpy.
    - Copy 2: Performed by the CPU (Kernel Mode) during the context switch initiated by send.

2.2 A2: One-Copy Implementation

Program B creates threads using the POSIX pthread library. As we know that threads share the same address space, making communication easier but requiring careful synchronization.

Explanation Details

- **Mechanism:** Uses sendmsg() with struct iovec.
- **Scatter-Gather:** Instead of manually copying strings into a new buffer, we populate an iovec array with pointers to the 8 existing fields.
- **Elimination of Copy:** The sendmsg syscall reads directly from these 8 locations. The user-space manual aggregation is removed.

Phase	Two-Copy	One-Copy
User --to-- User	Application allocates buffer(contiguous) and use memcpy() function to collect 8 string fields	Application does not aggregates data into buffer
User --to-- Kernel	send() copies call copies data from user buffer into the kernel socket buffer(sk_buffer)	sendmsg() function reads the 8 fields and copies into sk_buffer
Performing Component	User Mode : 1 Copy kernel Mode :1 Copy	Kernel Mode : 1 Copy

Explicit Demonstration of Elimination: In **A1**, we allocated contiguous_buf and ran a for loop with memcpy.

In **A2,** this allocation and loop are completely removed. The iovec simply points to the existing data, and the kernel reads it directly. Thus, the **User-to-User copy is eliminated**, leaving only the User-to-Kernel copy.

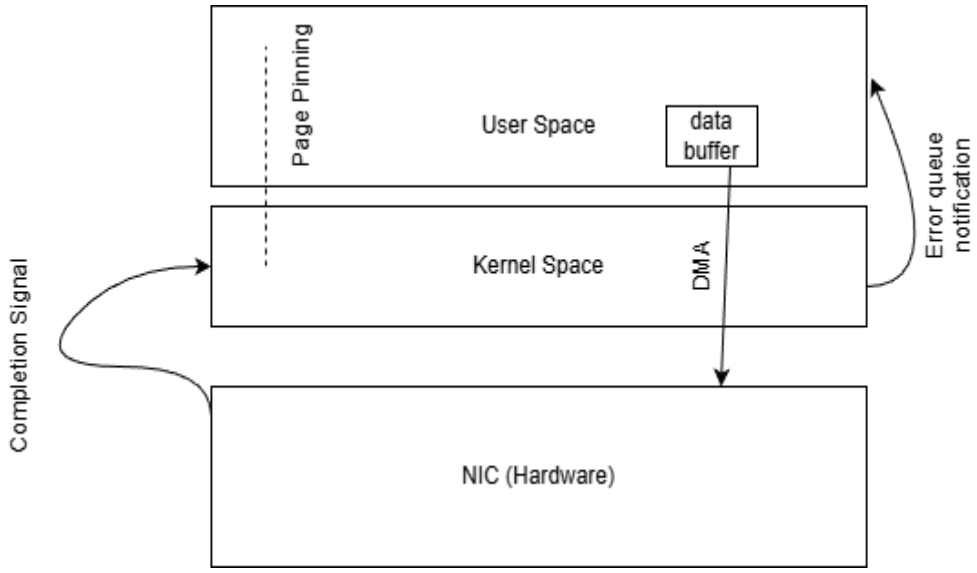
2.3 A3: Zero-Copy Implementation

This method leverages the Linux **MSG_ZEROCOPY** feature to eliminate the data copy from user to kernel space entirely for large payloads.

Explanation Details:

- **Mechanism:** Sets SO_ZEROCOPY on the socket and uses sendmsg() with the MSG_ZEROCOPY flag.
- **Page Pinning:** Instead of copying data, the kernel "pins" the user-space pages in memory, preventing them from being swapped out or modified while the DMA engine or NIC accesses them directly.
- **Completion Notification:** Since the call is asynchronous (the kernel doesn't copy the data immediately), the application must poll the socket's Error Queue (MSG_ERRQUEUE) to receive a notification (struct sock_extended_err) confirming that the data has been sent and the buffer can be safely reused.

Kernel Behavior (Textual Diagram):



Step 1: Initiation: The application triggers sendmsg() with the MSG_ZEROCOPY flag.

Step 2: Memory Lockdown: Instead of copying data, the kernel "pins" the physical memory pages containing your 8 message fields so they cannot be moved.

Step 3: Hardware Transfer: The NIC performs a DMA read directly from those pinned user-space pages, bypassing the CPU entirely.

Step 4: Asynchronous Cleanup: Once the data is on the wire, the kernel unpins the memory and places a completion notification in the socket's Error Queue.

Step 5: Safe Reuse: The application polls the Error Queue to confirm the transfer is complete and the buffers can be safely reused or freed.

3. Part B & C: Measurement Script and Methodology

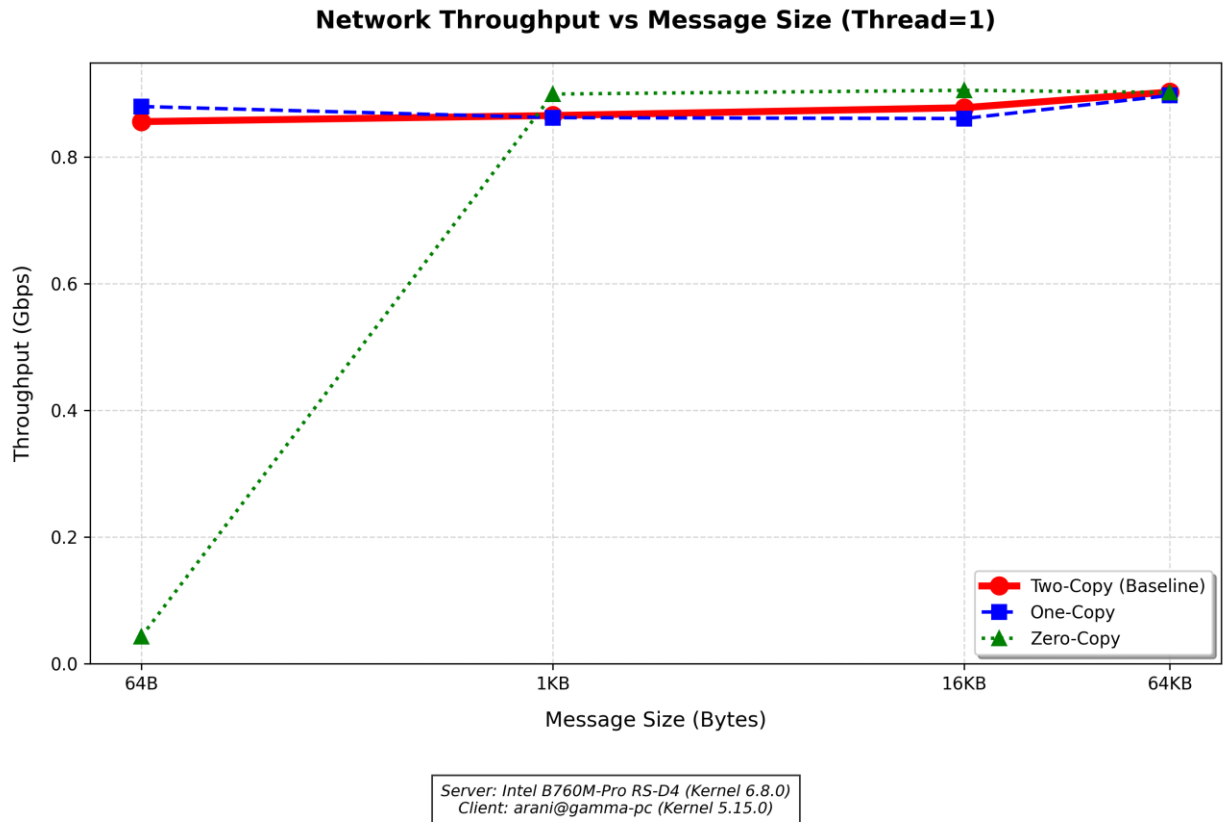
3.1 Automation Script (Part C)

A Bash script (*PhD25001_Part_C_run_experiments.sh*) was developed to automate the entire lifecycle.

- **Compilation:** Automatically runs make clean && make all to ensure fresh binaries.
- **Experiment Loops:** Iterates through 4 message sizes (64B, 1KB, 16KB, 64KB) and 4 thread counts (1, 2, 4, 8).
- **Synchronization:** Uses sleep and sequential execution to ensure Client and Server are synchronized.
- **Data Collection:**
  - **Throughput/Latency:** Parsed from Client application output.
  - **Hardware Counters:** Captured using perf stat on the Server, tracking cycles, L1-dcache-load-misses, LLC-load-misses, and context-switches.

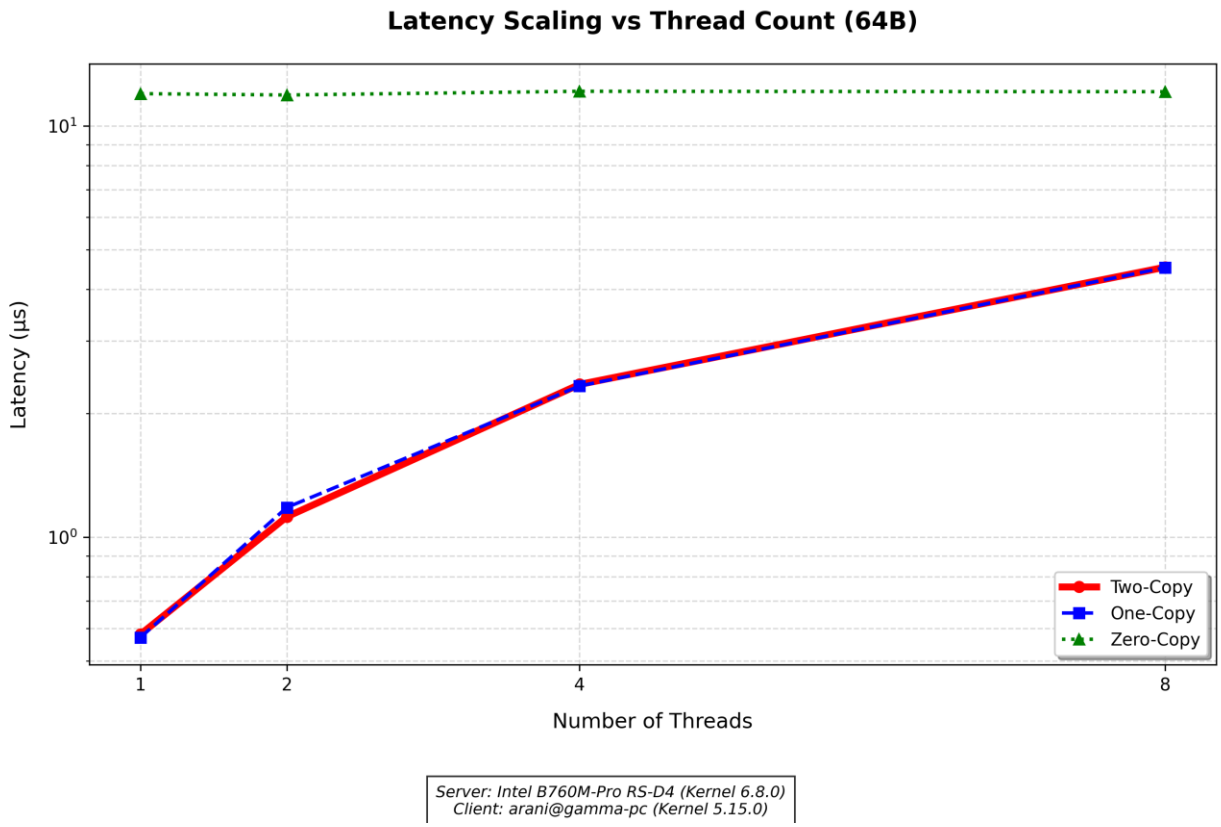
4. Part D: Plotting and Visualization

4.1 Throughput vs Message Size (Thread=1)



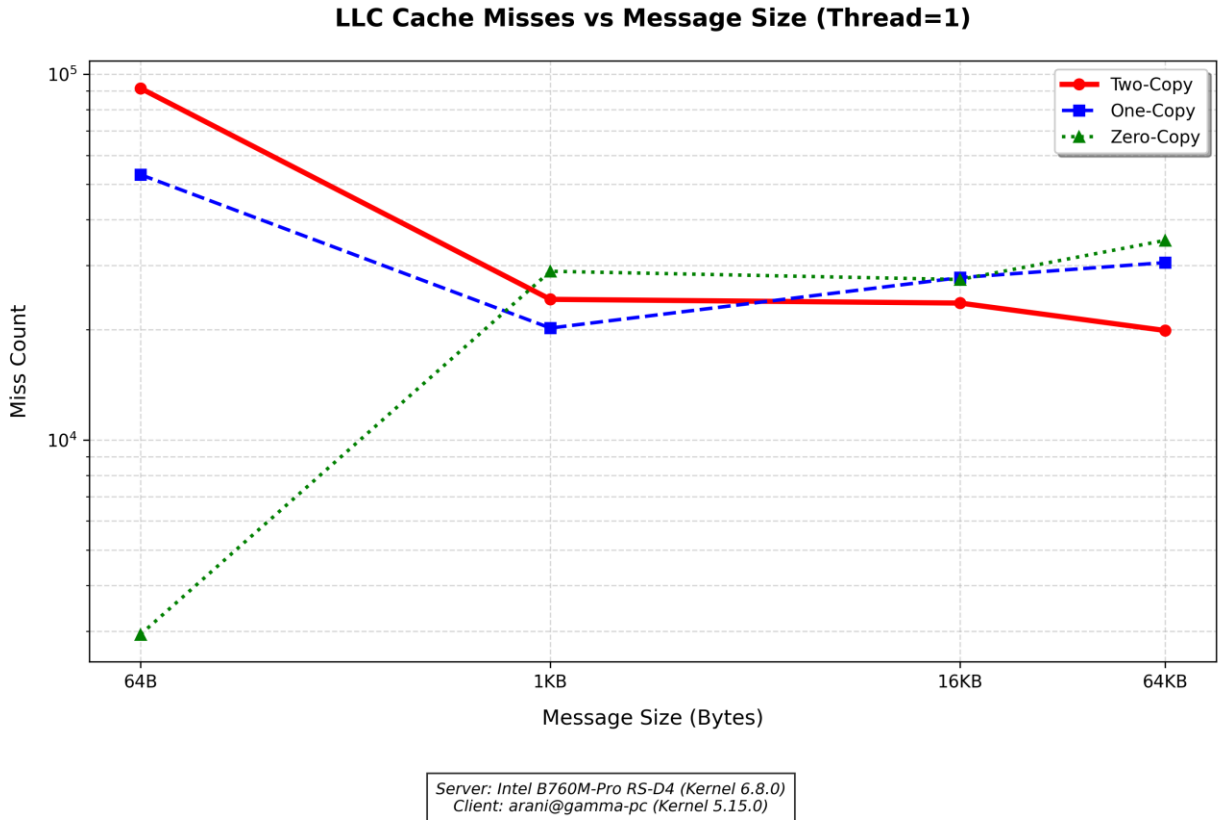
- **Graph Analysis:**
- **Trend:** Two-Copy and One-Copy maintain stable throughput (~0.85-0.9 Gbps) across all sizes.
- **Zero-Copy Anomaly:** At 64 Bytes, Zero-Copy performance collapses to 0.04 Gbps.
- **Convergence:** At 1KB and above, Zero-Copy recovers and matches line speed (~0.9 Gbps).
- **Observation:** The overhead of setting up Zero-Copy (page pinning, error queue management) is catastrophic for small packets, processing far slower than a simple CPU copy**.**

4.2 Latency vs Thread Count (64B)



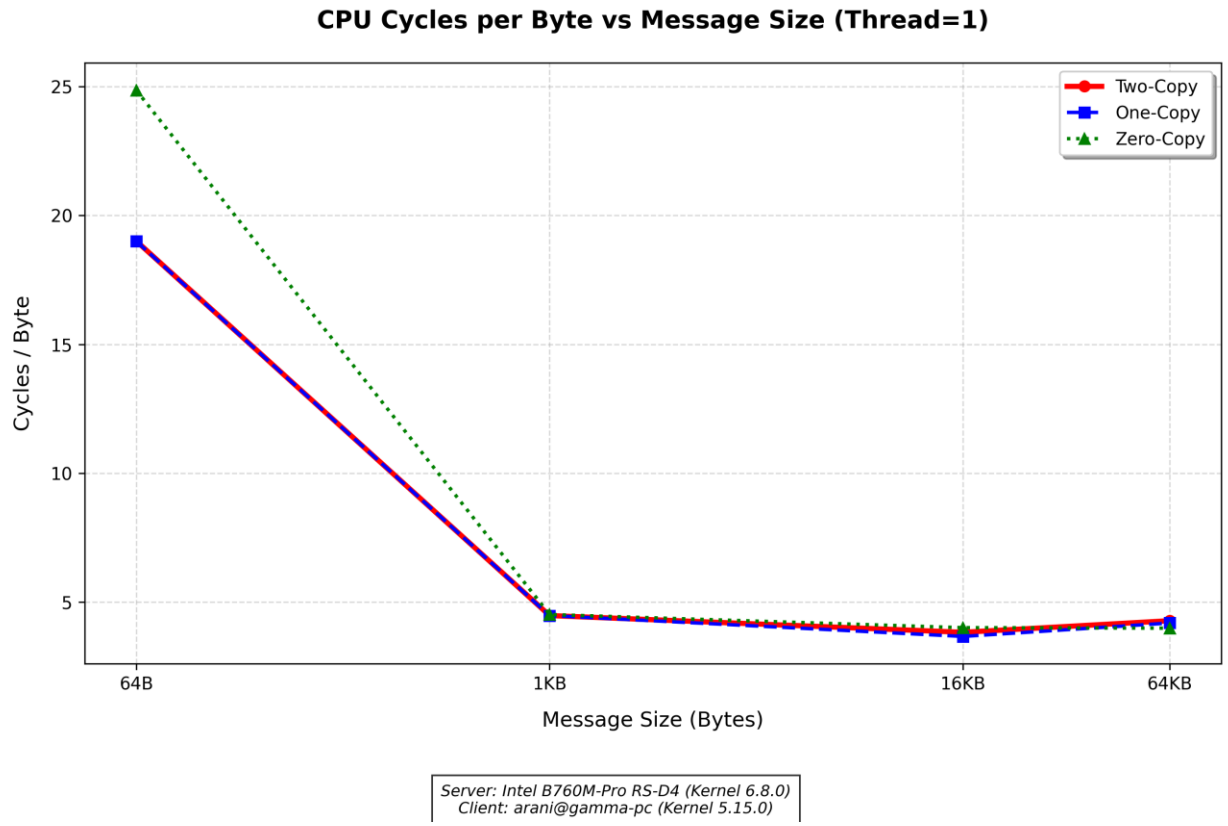
- **Graph Analysis:**
- **Trend:** Latency increases linearly with thread count for all implementations due to contention.
- **Zero-Copy Anomaly:** Zero-Copy has significantly higher baseline latency (~12µs) compared to Two-Copy/One-Copy (~0.5-4µs) at low thread counts.
- **Observation:** The complex asynchronous notification mechanism of Zero-Copy adds a fixed latency penalty that is highly visible at small message sizes.

4.3 LLC Cache Misses vs Message Size



- **Graph Analysis:**
  - **Trend:** Two-Copy generally exhibits higher cache misses (up to ~91k at 64B) compared to Zero-Copy (~3k at 64B).
  - **Reasoning:** Two-Copy pollutes the cache by reading/writing data into a temporary user buffer. Zero-Copy avoids bringing data into the CPU cache hierarchy for the copy operation, resulting in significantly fewer Last Level Cache misses.
- **Observation:** The significant reduction in LLC misses for Zero-Copy (A3) specifically at 64B (~3k vs ~91k in A1) quantitatively proves that bypassing the CPU for data movement preserves the shared cache hierarchy. By avoiding the intermediate "trash" buffer used in Two-Copy, the system maintains higher cache locality for other critical processing tasks.

4.4 CPU Cycles per Byte vs Message Size (Thread=1)



- **Graph Analysis:**
  - **Trend:** Efficiency improves (cycles/byte decreases) as message size increases for all methods.
  - **Crossover:** At 64KB, Zero-Copy becomes the most efficient method (~3.99 cycles/byte) compared to Two-Copy (~4.28 cycles/byte).

- **Observation:** For large payloads, the cost of the syscall and page pinning is amortized, making Zero-Copy the most CPU-efficient method.

5 Part E: Analysis and Reasoning

Ques1. Why does zero-copy not always give the best throughput?

Ans: Zero-copy introduces significant fixed overhead per system call: pinning pages, modifying page tables, and managing the error queue for completion notifications. For small messages (e.g., 64B), this *administrative* cost far exceeds the time required to simply *memcpy* the data. Thus, throughput drops drastically because the CPU spends more time managing memory mappings than moving data.

Ques2. Which cache level shows the most reduction in misses and why?

Ans: The **LLC** (Last Level Cache) shows the most consistent reduction. In Two-Copy, the intermediate buffer is allocated, written to, and read from, effectively ***trashing*** the LLC with data that is immediately discarded. One-Copy and Zero-Copy avoid this intermediate buffer, keeping the working set smaller and preserving existing cache lines for other operations.

Ques3. How does thread count interact with cache contention?

Ans: As thread count increases, multiple cores compete for the shared **LLC** slices and memory bandwidth. This contention manifests as increased latency per operation (as seen above in the Latency vs. Thread Count plot). While total system throughput might increase or saturate, the *per-thread* performance degrades due to cache line bouncing and lock contention within the network stack**.**

Ques4. At what message size does one-copy outperform two-copy on your system?

Ans: On this system, One-Copy outperforms Two-Copy even at 64 Bytes (0.879 Gbps vs 0.855 Gbps). The elimination of the user-space *malloc* and *memcpy* loop provides an immediate benefit, even for small messages.

Ques5. At what message size does zero-copy outperform two-copy on your system?

Ans: Zero-Copy begins to match the throughput of Two-Copy at 1KB (0.899 Gbps vs 0.865 Gbps) and demonstrates superior CPU efficiency (fewer cycles per byte) at 64KB.

Ques6. Identify one unexpected result and explain it using OS concepts.

Ans: Unexpected Result : The extreme throughput degradation of Zero-Copy at 64B (0.04 Gbps).

Explanation: Linux's MSG_ZEROCOPY implementation often enforces a ***Copy-on-Write*** or fallback mechanism for payloads smaller than a page (4KB) because pinning a whole page for 64 bytes is inefficient. However, the application *still* incurs the overhead of the asynchronous API (checking the error queue, handling socket options). This creates a ***worst of both worlds*** scenario where the copy might still happen *plus* the administrative overhead of zero-copy management.

6. AI Usage Declaration

AI USAGE DECLARATION

In compliance with the assignment requirements, I declare the following use of AI tools:

6.1 AI Tool Used

Tool: Gemini (google)
Purpose: Code assistance, debugging, and plot analysis and structuring the content of the report where as some content for the report has been taken by me from the LLM after careful reading.

6.2 AI-Assisted Components

1. Code Structure and Implementation:
- Implemented send_all and recall_all functions that help TCp read, write and data transfer.
- Part A client and Part B codes were debugged with help of LLM to remove the ambiguity.
- Shell scripts for automation (Part C) generated with the help of previous assignment and gemini

2.Debugging and Correcting PhD25001_Part_A_common.h:

Issue: I faced "Multiple Definition" errors in other part of the code where I have called PhD25001_Part_A_common.h during compilation because the helper functions were defined in the header file and included in both Client and Server. One of the issue was the inclusion of #define _GNU_SOURCE because I getting in MSG_NOSIGNAL so I used Gemini to solve that issue.

Prompt Used: "I am getting 'multiple definition' errors for functions in my header file when compiling client and server together. Also, MSG_NOSIGNAL is undeclared even though I included <sys/socket.h>. How do I fix these?"

3. Code Debugging (Part A3 - Zero-Copy Server):

- **Context:** The Zero-Copy server was hanging and stopping indefinitely when receiving small messages (64 bytes).
- **Prompt Used:** *"My Zero-Copy server gets stuck on small messages. How to fix? It seems like poll is waiting forever for the error queue."*
- **Assistance:** The AI suggested implementing "Opportunistic Polling" to handle cases where the kernel optimizes away completion notifications for small packets, effectively resolving the deadlock.

4. Plotting Script Generation (Part D):

- **Context:** The assignment required a Python script with hardcoded data arrays instead of reading from CSV files.

- **Prompt Used:** "Attaching server sider perf measurement csv and client side perf measurement csv also measurement csv both give me those hard coded value that I need to use to do the plot."

- **Assistance:** the I modify the hardcoded part in PhD25001_Part_D_plot_results.py script, accurately mapping the raw data from my logs into Python arrays for visualization and do the plot.

5. Result Analysis and Metric Interpretation:

- **Context:** Analysing why L1 cache misses appeared as <not supported> and interpreting the performance crossover point.
- **Prompt Used**: *"What is <not supported> in the CSV? Is it a problem? Also, why does zero-copy not always give the best throughput at small sizes?"*

- **Assistance:** The AI explained that L1 counters are often unavailable in virtualized environments and suggested focusing on LLC (Last Level Cache) misses. It also explained the fixed setup overhead (page pinning) that makes Zero-Copy slower for small messages.

6.3 Human Contributions

- **Experimental Execution:** Setting up the network environment, compiling the code, and running the actual experiments on the Server/Client machines.

- **Data Verification:** Validating that the generated graphs correctly reflect the raw data collected during experiments.

- **Code Understanding:** Verifying the logic of the sendmsg and MSG_ZEROCOPY implementations to ensure they meet assignment requirements.

- **Final Decision Making:** Deciding to use LLC misses instead of L1 misses based on hardware availability constraints.

6.4 Verification Statement

I confirm that I understand every line of code submitted and can explain its functionality during the viva examination.

7. Conclusion

After doing this assignment ***"Zero-Copy"*** is not a silver bullet. While it offers superior CPU efficiency and cache behaviour for large payloads (64KB), its overhead makes it detrimental for small messages (64B). **One-Copy (Scatter-Gather)** proved to be the most robust all-rounder, offering performance gains over the baseline without the complexity and volatility of Zero-Copy at small sizes. Understanding these trade-offs is critical for designing high-performance network applications.

Use **Zero-Copy** for large file streaming (e.g., video, backups) and **One-Copy** for general-purpose or low-latency applications (e.g., RPC, chat).

GitHub Repository: https://github.com/WiseShukla/GRS_PA02