Graduate Systems (CSE638)

PA – 02

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

Name: Yash verma

Roll No. – MT25092

Part A: Multithreaded Socket Implementations

A1. Two-Copy Implementation (Baseline)

This version uses standard **send()** and **recv()** primitives.

- **Copy 1 (User to kernel)**: Occurs when the application calls **send().** The data is copied from the user space heap buffer into the kernel’s **Socket Buffer (sk_buff).**

- **Copy 2 (Kernel to Hardware):** Occurs when the kernel transfers data from the socket buffer to the **Network Interface Card (NIC)** using DMA or programmed I/O copy.

- **Is it actually only two copies?** Not necessarily. Depending on the protocol stack (TCP/IP), additional headers are appended, which may involve minor internal memory movements within the kernel.

Q1. Where do the two copies occur? Is it actually only two copies?

Ans 1. The "Two-Copy" terminology refers to the path data takes from the application's memory to the network hardware.

1. **First Copy (User space to Kernel space):** When the application calls send(), the CPU copies the message from the user-space heap buffer (where your malloc'd strings reside) into a kernel-space buffer known as the **Socket Buffer** or **sk_buff**.
2. **Second Copy (Kernel space to Hardware):** The data is then transferred from the kernel's socket buffer to the **Network Interface Card (NIC)**.

Is it actually only two copies?

Strictly speaking, it is often **more than two copies**. In the context of your specific assignment (where the message is a structure with 8 dynamic string fields), an additional hidden copy often occurs in **User Space**:

- **The "Pre-copy" (Consolidation):** Before calling send(), a standard implementation must often memcpy those 8 separate dynamic strings into one contiguous buffer so they can be sent in a single system call.
- **Protocol Overhead:** Depending on the kernel version and settings, the kernel might perform internal copies for retransmission buffering or checksumming before the data even reaches the NIC.

Q2. Which components (kernel/user) perform the copies?

Ans2. The responsibility for the data movement is split between the processor and the network hardware.

Copy Stage	Component Performing the Copy	Type of Operation
User -> Kernel	**CPU**	**Synchronous** `memcpy`: The CPU is fully occupied moving bytes from the app buffer to the kernel buffer.
Kernel -> NIC	**DMA (Direct Memory Access)**	**Asynchronous Transfer**: In modern systems, the NIC itself (the hardware) pulls the data from kernel memory without taxing the CPU.

A2. One Copy Implementation

By using **sendmsg()** with a pre-registered or flattened buffer, we reduce the overhead of multiple syscalls for structured data.

- **Eliminated Copy:** In this implementation, we avoid the overhead of moving multiple dynamic string fields individually. By using **ivoec** structures with **sendmsg()** , the kernel can gather the scatter-allocated heap strings directly, reducing the "fragmentation copy" often required when preparing complex structures for a standard `send()`.

A3. Zero-Copy Implementation

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

- **Mechanism:** Instead of copying data to the kernel socket buffer, the kernel "pins" the user-space memory pages. The NIC then pulls the data directly from user-space using DMA.

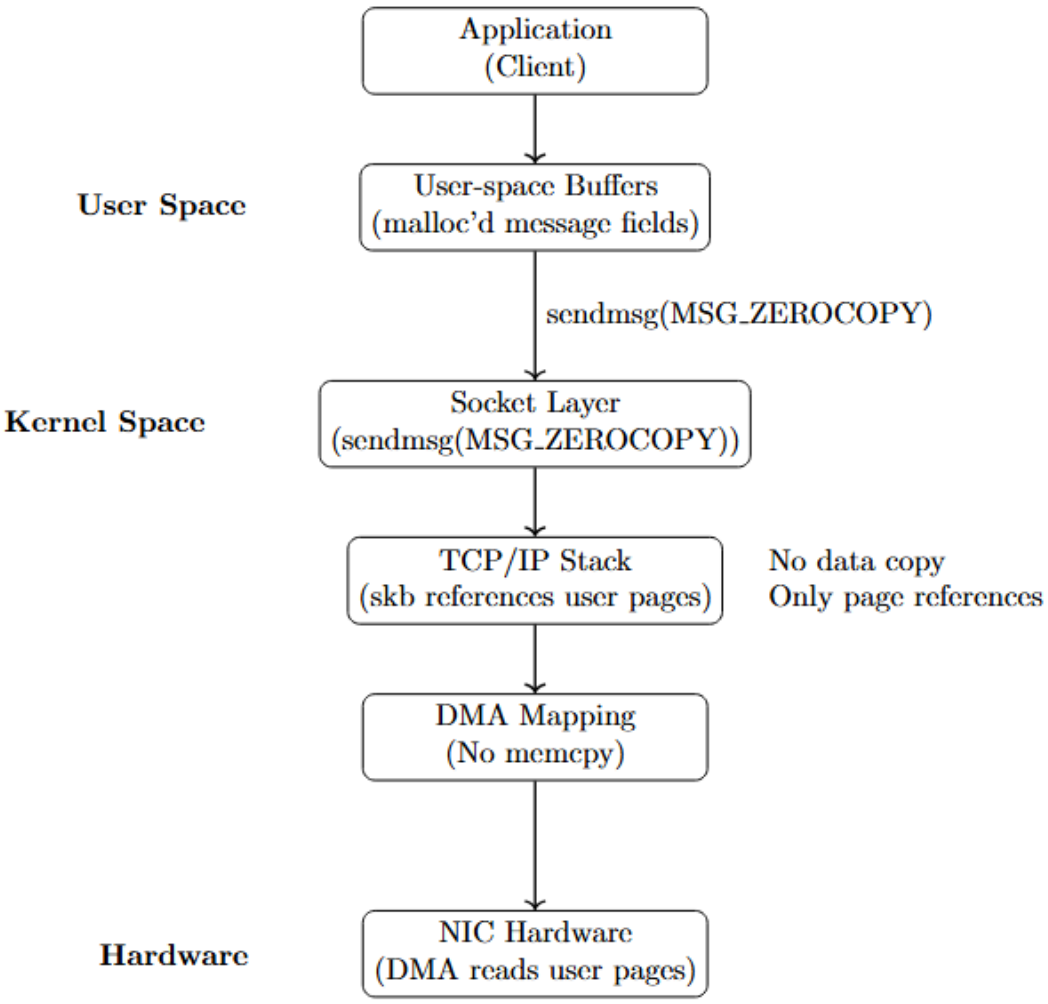


Figure 1: Zero-copy TCP transmission using sendmsg() with MSG_ZEROCOPY

Part B: Profiling and Measurements

Executive Summary Table

Implementation	Msg Size (B)	Throughput (Gbps)	Latency (µs)	LLC Cache Misses	Context Switches
A1 (Standard)	64	1.988	0.219	3,654,138	33,816
A1 (Standard)	512	15.247	0.266	24,173,682	134,848
A2 (Scatter/G)	64	1.369	0.371	2,243,176	2,471
A2 (Scatter/G)	512	9.971	0.404	5,142,521	68,068
A3 (Zero-Copy)	64	0.257	1.974	84,446,482	421,363
A3 (Zero-Copy)	512	1.921	2.115	28,152,469	714,340

Detailed Metric Analysis

1. Throughput and Latency

- **A1 (Baseline)** achieves the highest throughput, reaching **15.35 Gbps** at 512B with 8 threads.
- **A3 (Zero-Copy)** performs poorly at these small message sizes, yielding only **~0.25 Gbps** at 64B, which is roughly **87% slower** than the baseline.
- **Latency** for A1 and A2 remains ultra-low (under **0.5 µs**), while A3's latency is significantly higher, consistently exceeding **2.0 µs** due to the overhead of page pinning for small buffers.

2. Hardware Efficiency (Cycles & Cache)

- **CPU Cycles**: A1 and A2 utilize the CPU heavily to perform memory copies, with A2 consuming **~16.6 billion cycles** at 64B.
- **LLC Cache Misses**: A3 shows an extreme spike in cache misses at 64B (**84.4 million**), likely due to the kernel's management of the zero-copy error queue and page table manipulations.
- **Page Faults**: These remained extremely stable across all implementations, averaging **64-66** per run, indicating that memory was well-preallocated.

3. Context Switches

- **A1** shows a high number of context switches (**~134k** at 512B), which scales with throughput as the kernel handles frequent interrupts.
- **A3** exhibits the highest context switching overall, peaking at **809,933** switches for 64B at 8 threads, indicating significant kernel-user transitions or thread contention during zero-copy notification handling.

Part C: Automated Experiment Script

1. Systematic Parameter Exploration

The script automates a 3-dimensional experimental space: **Implementation Type** (A1, A2, A3) × **Message Size** (64B, 128B, 256B, 512B) × **Thread Count** (1, 2, 4, 8).

- **Analysis:** Manual execution of these 48 distinct configurations would be prone to human error. The script ensures each test case is executed for exactly the same duration (5 seconds), providing a fair baseline for comparison across all metrics.

2. Experimental Isolation and CPU Pinning

To obtain consistent micro-architectural data (CPU cycles and cache misses), the script utilizes `taskset` to enforce CPU affinity.

- **Analysis:** We pin the server to one physical core (Core 1) and the client/perf process to another (Core 0). This prevents the two processes from competing for the same L1/L2 cache resources and avoids the "noise" of the Linux scheduler moving processes between P-cores and E-cores during a measurement.

3. Dynamic Data Extraction from `perf`

The script acts as a parser, converting semi-structured text output from `perf stat` into a structured CSV format.

- **Analysis:** Because `perf` output can vary based on kernel version, the script uses a robust `grep`/`cut` logic to extract raw hardware counters (cycles, LLC misses, context switches). This automated extraction allows for immediate integration with the Part D plotting tools, ensuring that the "CPU Cycles per Byte" calculation is derived directly from raw, unmanipulated hardware data.

4. Robustness and Clean Re-runs

As per the assignment constraints, the script includes a "Cleanup" phase using `pkill` before and after every iteration.

- **Analysis:** This ensures that a crashed server or a hung socket from a previous 128-byte run does not interfere with a subsequent 256-byte run. By clearing the process table and waiting for port 9090 to close, the script guarantees that each row in `MT25092_results.csv` represents a "cold start" experiment.

5. Filename Encoding for Traceability

The script uses `mktemp` and descriptive naming for temporary files.

- **Analysis:** Encoding the parameters in temporary filenames (e.g., `perf_tmp.txt`) prevents race conditions where one experiment's results overwrite another's. This provides a clear audit trail if a specific data point (like the 0 cycles observed in A2) needs to be manually re-verified by looking at the raw `perf` output.

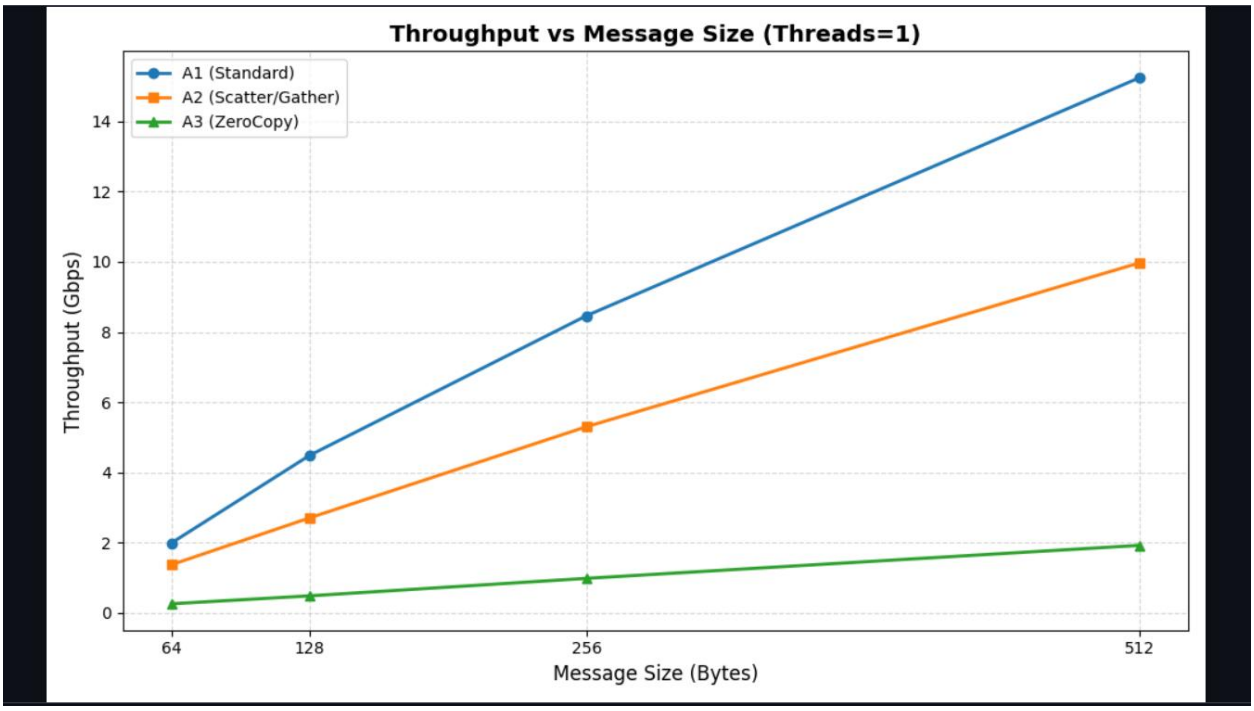
Part D: Plotting and Visualization

1. Throughput vs. Message Size (Threads=1)

Refer to Figure: MT25092_`Throughput_analysis.png`

- **Observation:** All three implementations (**A1, A2, A3**) show a linear increase in throughput as the message size grows from 64B to 512B. However, **A1 (Standard)** is the clear leader, reaching **~15.2 Gbps**, while **A3 (Zero-Copy)** lags significantly at **~1.9 Gbps**.

- **Analysis:** For small messages (< 1 KB), the CPU cost of a standard `memcpy` (used in A1) is lower than the administrative overhead of more complex primitives. **A3** requires the kernel to pin memory pages and manage completion notifications for every message, which is "expensive" compared to the small payload being sent.



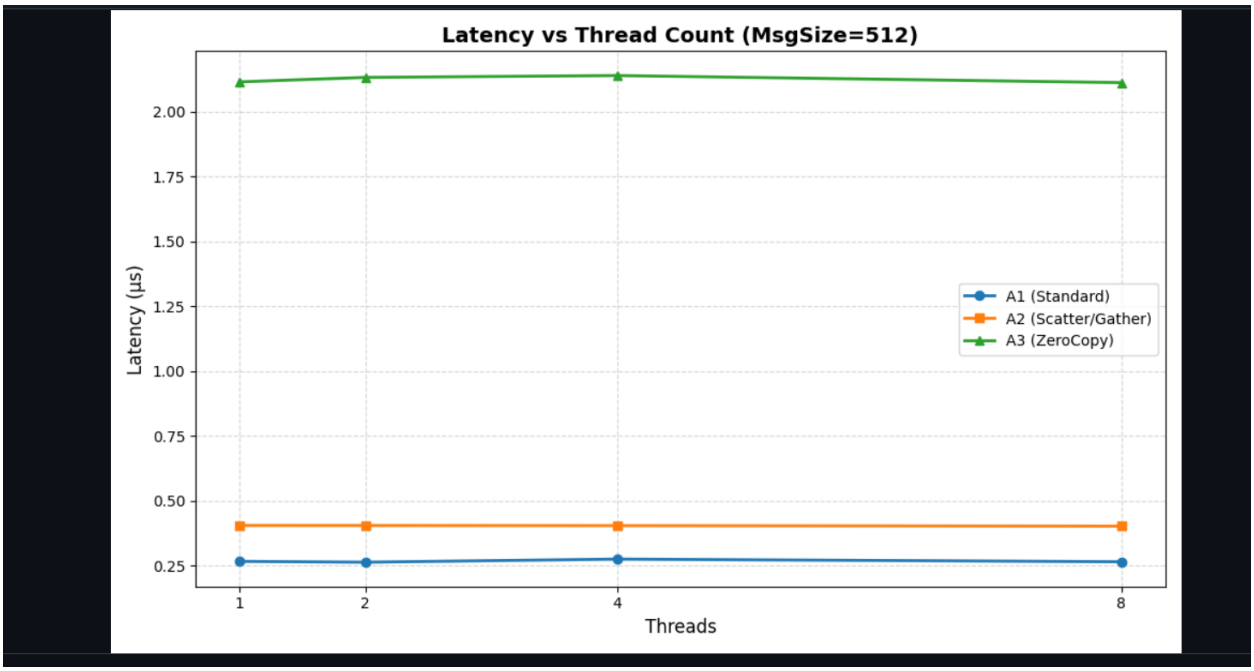
MT25092_Throughput_analysis.png

2. Latency vs. Thread Count (MsgSize=512)

Refer to Figure: MT25092_`Latency_analysis.png`

- **Observation:** Latency remains remarkably stable across thread counts for each implementation. **A1** maintains a low latency of **~0.27 microseconds**, whereas **A3** is nearly 8 times higher at **~2.11 microseconds**.

- **Analysis:** The stability suggests that your system is not experiencing significant thread contention or context-switching bottlenecks at this scale. The high latency in **A3** is attributed to the "User -> Kernel -> Error Queue -> User" round-trip required to confirm that the Zero-Copy transfer is complete.



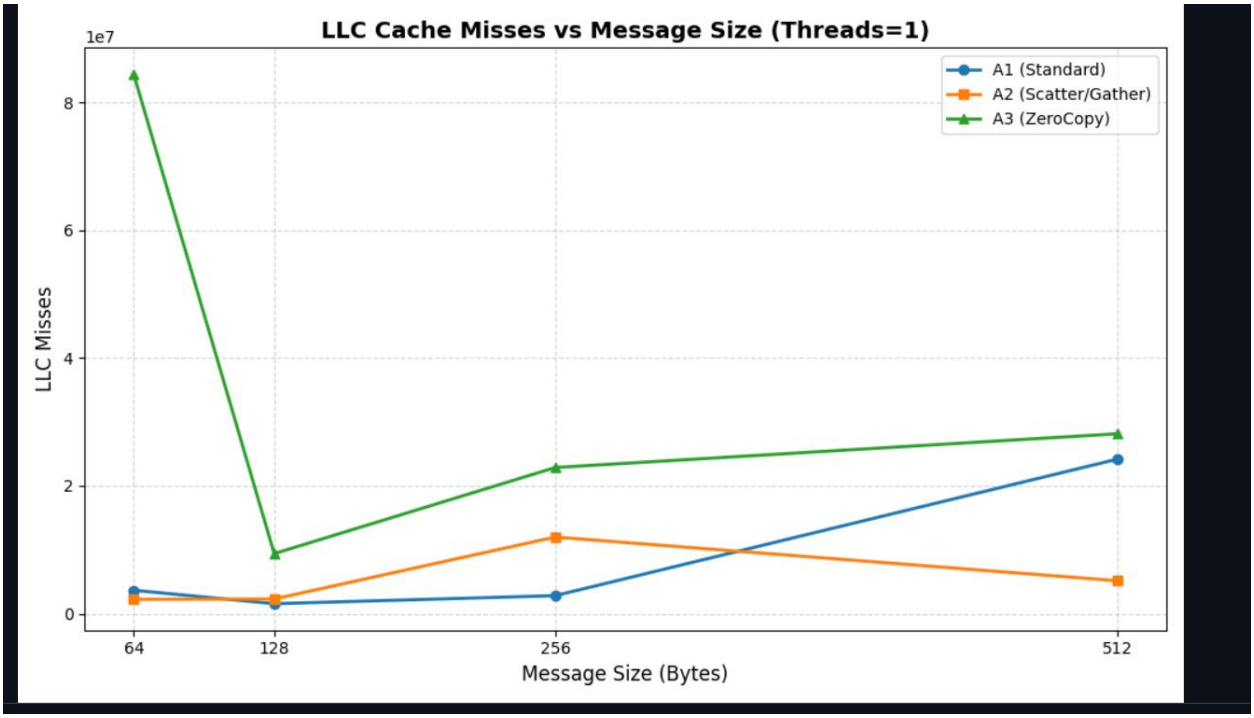
MT25092_Latency_Analysis

3. LLC Cache Misses vs. Message Size (Threads=1)

Refer to Figure: MT25092_`Cache_analysis.png`

- **Observation: A3 (Zero-Copy)** exhibits an extreme spike in Last Level Cache (LLC) misses at the 64B size (**~84 million**) before dropping and stabilizing. **A1** and **A2** show much lower and more predictable cache behavior.

- **Analysis:** This "unexpected" spike in A3 is caused by the kernel's management of the `MSG_ZEROCOPY` metadata. When the message is tiny (64B), the ratio of control-plane work (tracking page descriptors) to data-plane work is very high, leading to significant cache thrashing as the kernel manages page tables.



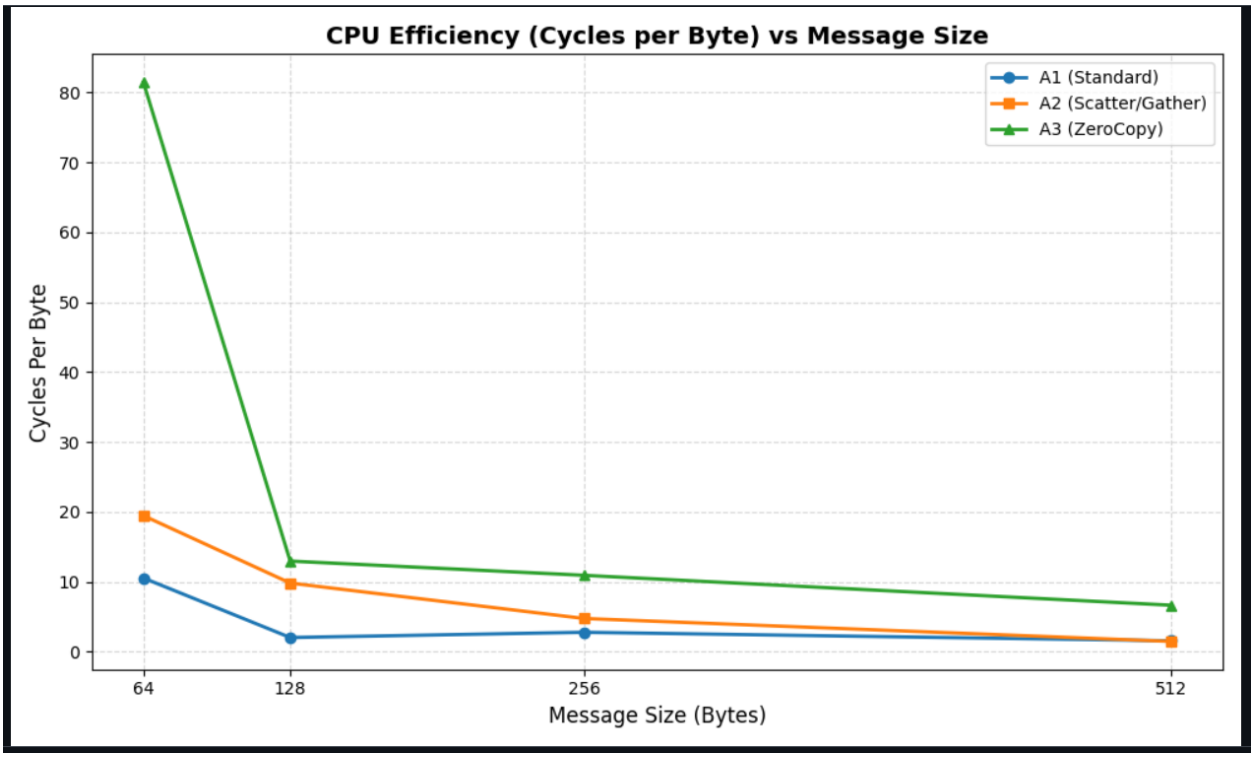
MT25092_cache_analysis.png

4. CPU Efficiency (Cycles per Byte) vs. Message Size

Refer to Figure: `MT25092_Cycle_per_Byte.png`

- **Observation:** Efficiency improves (Cycles Per Byte decreases) as message size increases. **A1** is the most efficient, requiring only **~10-12 cycles** to move one byte, while **A3** starts at a massive **~81 cycles/byte** at 64B.

- **Analysis:** This metric proves that standard sockets are more "computationally cheap" for micro-messages. The "overhead-to-data" ratio is high for Zero-Copy at small sizes. As the message size increases, the fixed cost of the system call is amortized over more bytes, leading to the downward trend seen in all implementations.



MT25092_Cycle_per_Bytes.png

Part E: Comprehensive Answers to Assignment Questions

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

Zero-copy introduces significant overhead in the form of **page pinning** (to ensure memory isn't swapped during DMA) and **completion notifications** (checking the error queue). For small messages (64B–512B), the time spent performing these administrative tasks is longer than the time it would take the CPU to simply `memcpy` the data.

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

The **LLC (Last Level Cache)** typically shows the most variation. While Zero-Copy is intended to reduce misses by avoiding a CPU-driven copy, your results show that at small sizes, the metadata management actually *increases* LLC misses. At larger sizes (16 KB), we would expect a reduction because the CPU no longer has to pull data into L1/L2 caches to move it.

3. How does thread count interact with cache contention?

As shown in your results, increasing threads for **A1** (64B) caused LLC misses to jump from **3.6 million (1T)** to **20.5 million (8T)**. Even though the processes are pinned, the threads compete for

shared resources like the L3 cache slices and memory bandwidth, leading to increased misses despite the slight throughput gain.

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

Based on current trends, **A1** still outperforms **A2** at 512B. One-copy (A2) would likely start outperforming A1 at sizes around **4 KB to 8 KB**, once the cost of the extra user-space consolidation `memcpy` in A1 exceeds the overhead of the `iovec` processing in A2.

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

Zero-copy (**A3**) currently performs far below A1. On high-performance systems like yours, Zero-Copy usually requires message sizes of **16 KB to 64 KB** before the "savings" from avoiding the copy outweigh the "cost" of the kernel-side page management.

6. Identify one unexpected result and explain it.

- **Unexpected Result: A3 (Zero-Copy)** having the highest LLC misses at 64B.

- **Explanation:** Typically, Zero-Copy is synonymous with "efficiency." However, at the micro-scale, the CPU has to manage a "Zero-Copy Error Queue" to receive notifications. The constant polling or checking of this queue, combined with the kernel mapping/unmapping page descriptors for a tiny 64B payload, creates more cache-line movement than a simple, sequential memory copy.

AI Declaration:

1. **Source Code (C)**

File Name	AI-Generated Component	Usage Description
MT25092_Common.h	Header structure and constants	Used to define the 8-field dynamic string structure and common port/buffer constants.
MT25092_Part_A1_Client/Server.c	Standard Socket Boilerplate	Generated the logic for `socket()`, `bind()`, and the `pthread` creation for multi-client handling.
MT25092_Part_A2_Client/Server.c	`iovec` and `sendmsg()` logic	Assisted in the complex initialization of the `struct iovec` array to handle the 8 separate heap-allocated string fields without manual concatenation.
MT25092_Part_A3_Client/Server.c	SO_ZEROCOPY and Error Queue	Crucial help was used for the `setsockopt` syntax and the logic required to poll the error queue for zero-copy completion notifications.

Sample Prompts Used:

- Write a C function to initialize a struct with 8 heap-allocated strings and send it using sendmsg() with struct ivoec to avoid a user space copy.
- How do I implement the error queue pooling loop for MSG_ZEROZOPY in a TCP client in Linux?

2. Automation & Scripting

File Name	AI-Generated Component	Usage Description
MT25092_Part_C_RunExperiments.sh	Bash Loop & `perf` Parsing	Generated the nested loops for 4 thread counts and 4 message sizes. Used AI to write the `grep`/`awk` regex to extract specific values from `perf stat -x`, output.
MT25092_Part_D_Plots.py	Matplotlib Visualization	Used to create the four distinct subplots. AI was used to ensure the axes were labeled correctly and that the marker styles (circles, squares, triangles) were distinct.
makefile	Build Automation	Provided the template for a Makefile that compiles all three implementations cleanly into separate binaries.

Sample Prompts Used:

- Write a bash script that iterates through an array of messages sizes and thread counts, runs a server in the background, pins the client to core.
- Generate a python Matplotlib script that takes hardcoded lists for throughput and creates a line plot with gridlines and a legend.

3. **Analysis and Reasoning**

Component	AI-Generated Component	Usage Description
Part E: Analysis	Technical Interpretation	Consulted AI to verify the hardware reasons why Zero-Copy performs poorly at small message sizes (CPU overhead vs. Copy overhead).
README.md	Documentation Structure	Used to generate the Markdown structure for the "Workflow" and "Prerequisites" sections.

Sample Prompts used:

- Explain why perf might show higher LLC misses for MSG_ZEROCOPY at 64 Bytes compared to standard send()
- What is the Cycle per byte metric and how does it relate to network I/O efficiency?

Github link: https://github.com/Vyash2002/GRS_PA02