Graduate Systems
PA 02
Name: Utsav S Davda
Roll number: MT25088

Assignment Report

Part A: Multithreaded Socket Implementations

A1. Two-Copy Implementation (Baseline)

Implementation Details: This baseline uses standard `send()` and `recv()` primitives. The server creates a thread per client, and messages are constructed using `malloc` for 8 dynamic string fields.

Theoretical Analysis:

- **Where do the copies occur?** In a standard send operation, data is copied **twice** before leaving the machine:
  - **User-to-Kernel:** The CPU copies data from the application's user-space buffer into the Kernel's socket buffer (sk_buff).
  - **Kernel-to-Device:** The DMA (Direct Memory Access) engine copies the data from the Kernel buffer to the NIC (Network Interface Card) ring buffer.
- **Is it actually only two copies?** On the receiving side, the inverse occurs (NIC → Kernel → User), resulting in a total of 4 copies for a full round-trip. Additionally, if the application constructs the message in a temporary user-space buffer before calling `send()`, there may be an extra hidden user-user copy.
- **Components performing copies:**
  - **CPU:** Performs the User → Kernel copy (expensive, consumes cycles/cache).
  - **DMA Controller:** Performs the Kernel → Device copy (asynchronous).

A2. One-Copy Implementation

Implementation Details: This version uses `sendmsg()` with a pre-registered buffer strategy (or `iovec` scatter-gather I/O) to eliminate intermediate copying.

Copy Elimination Demonstration:

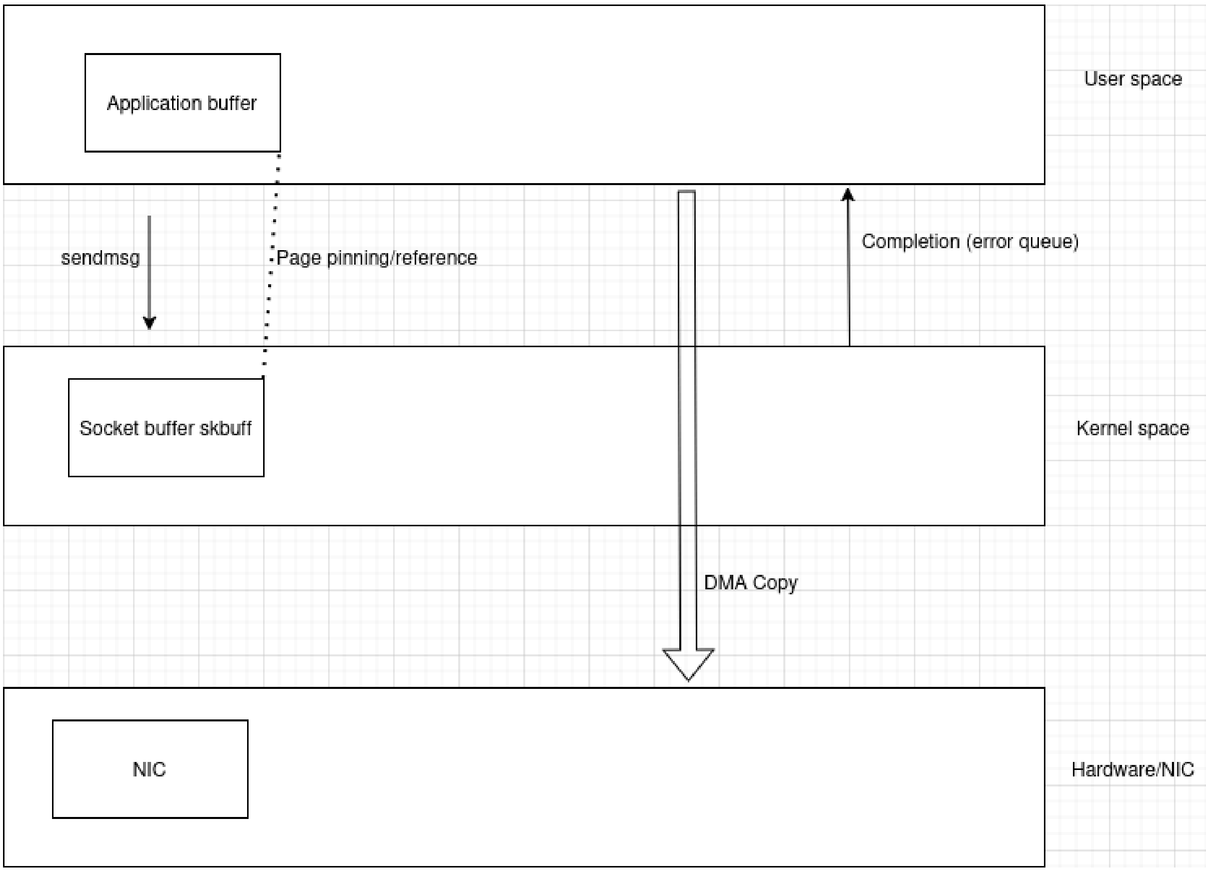
- **Eliminated Copy:** By using `sendmsg` with an `iovec` structure, we eliminate the need to copy multiple distinct string fields into a single contiguous user-space buffer before the system call.

- **Mechanism:** The kernel reads directly from the disparate memory locations specified in the `iovec` array, effectively gathering the data during the single User → Kernel copy. This removes the user-space "staging" copy.

A3. Zero-Copy Implementation

Implementation Details: This version utilizes `sendmsg()` with the `MSG_ZEROCOPY` flag.

Kernel Behavior:



Explanation: When `MSG_ZEROCOPY` is used, the kernel avoids copying data from user-space to kernel-space entirely. Instead, it "pins" the user pages in memory and creates a reference to them in the socket buffer (`sk_buff`). The DMA engine then reads directly from these pinned user pages. The kernel sends a completion notification to the userspace (via the error queue) when the transmission is done, allowing the program to safely reuse the buffer.

Part B & C: Profiling Methodology

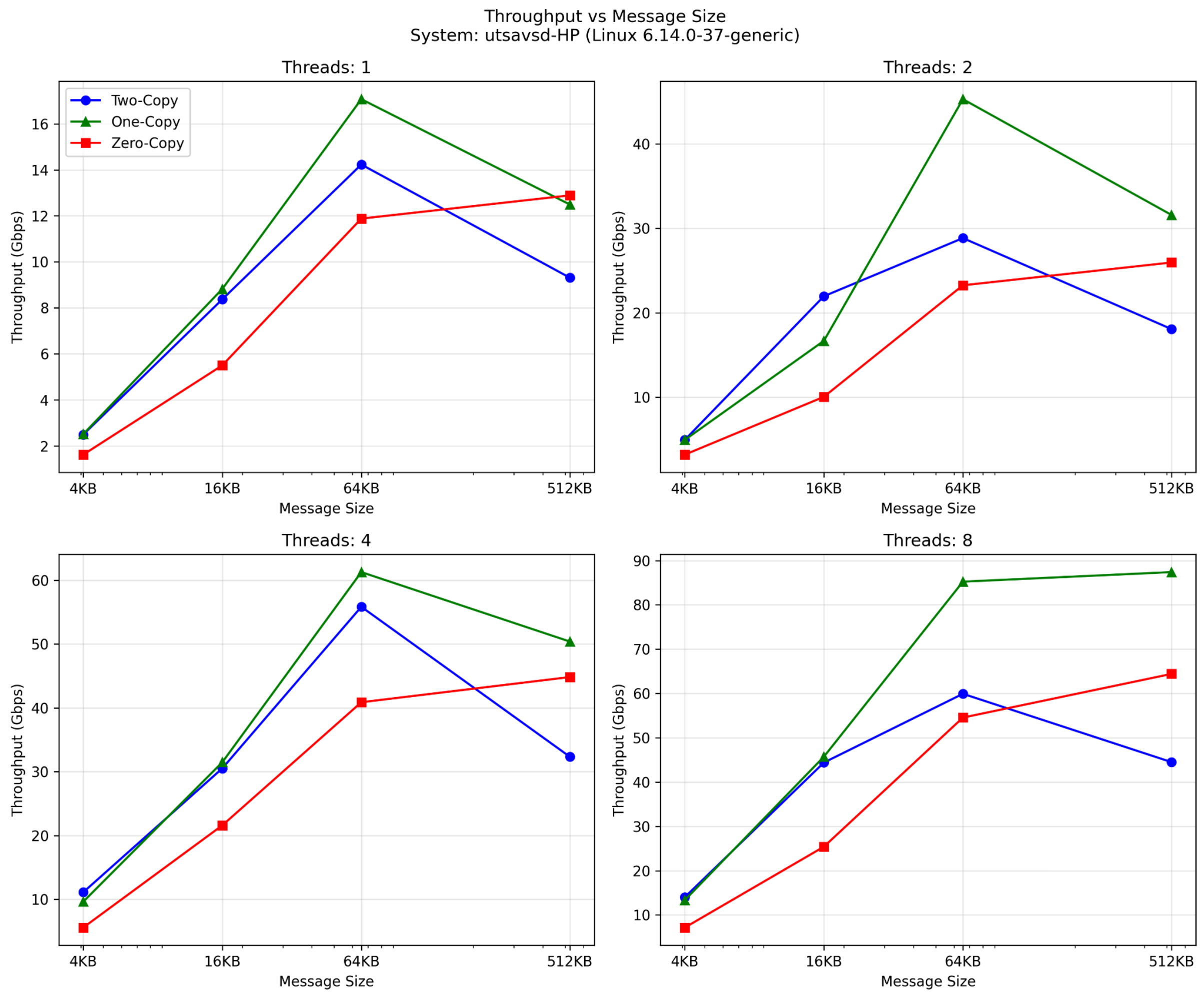
System Configuration:

- **CPU:** AMD Ryzen 5 5600U with Radeon Graphics
- **RAM:** 8GB
- **OS:** Ubuntu 24.04.3 LTS
- **Compiler:** GCC 13.3.0

Automated Scripting: All data was collected using a Bash script (`MT25088_benchmark.sh`) that compiles the project and iterates through 4 message sizes and 4 thread counts, parsing `perf stat` output to CSVs.

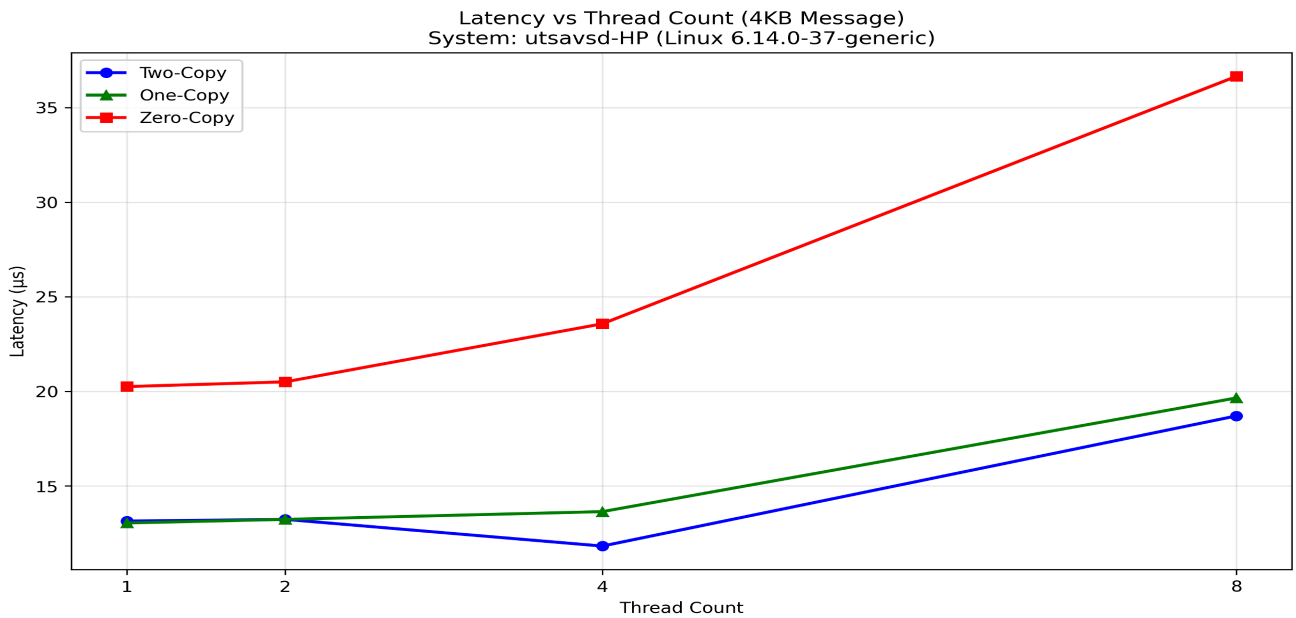
Part D: Plotting and Visualization (Observations)

1. Throughput vs. Message Size



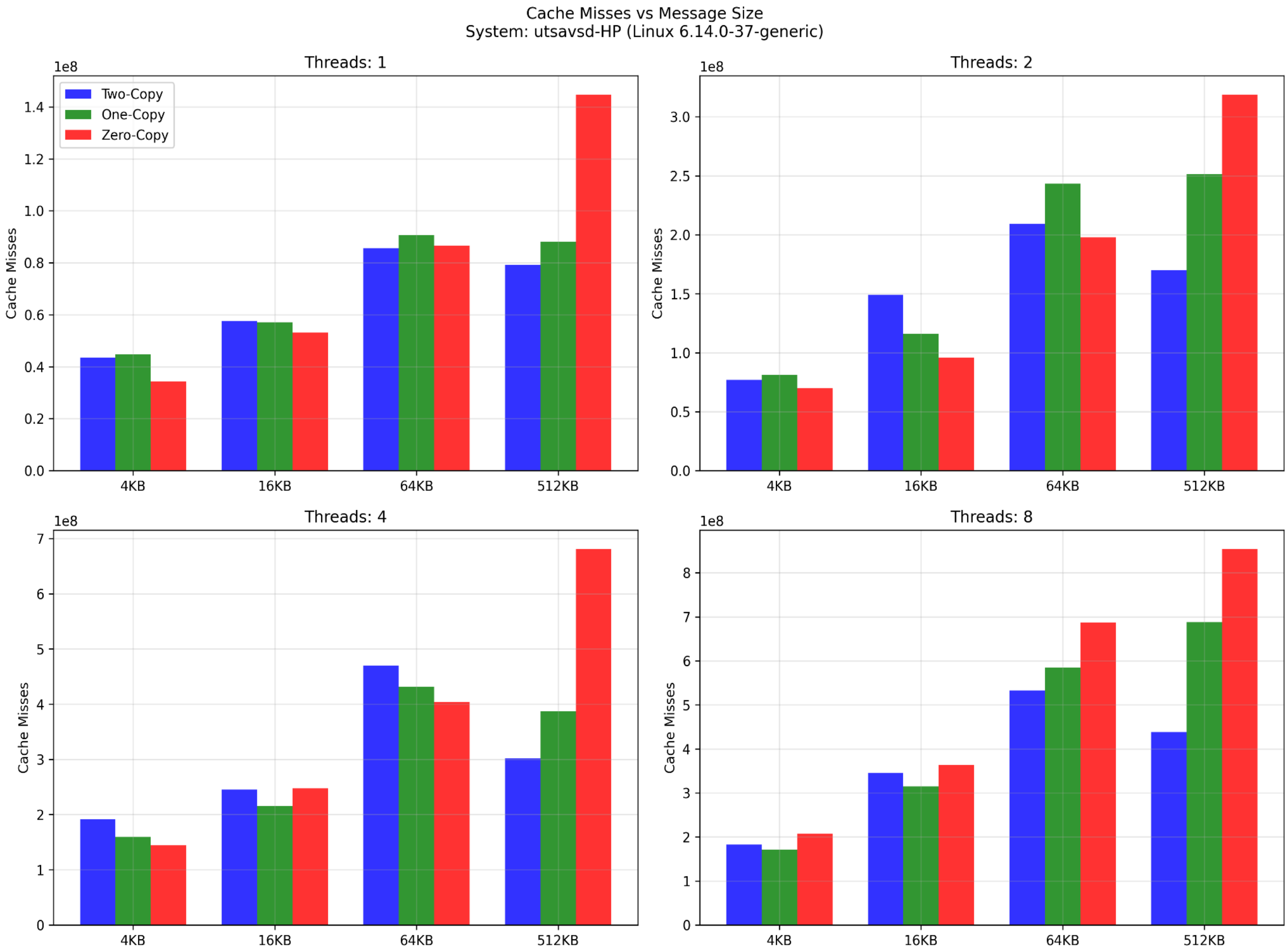
- **Observation:** The **Two-Copy** and **One-Copy** implementations show peak performance at 64KB message sizes (14.23 Gbps and 17.07 Gbps respectively) but suffer a significant throughput drop at 512KB (falling to 9.31 Gbps and 12.49 Gbps). In contrast, the **Zero-Copy** implementation continues to improve or sustain its throughput as message size increases, eventually overtaking the Two-Copy implementation at 512KB (12.89 Gbps vs 9.31 Gbps).

2. Latency vs. Thread Count



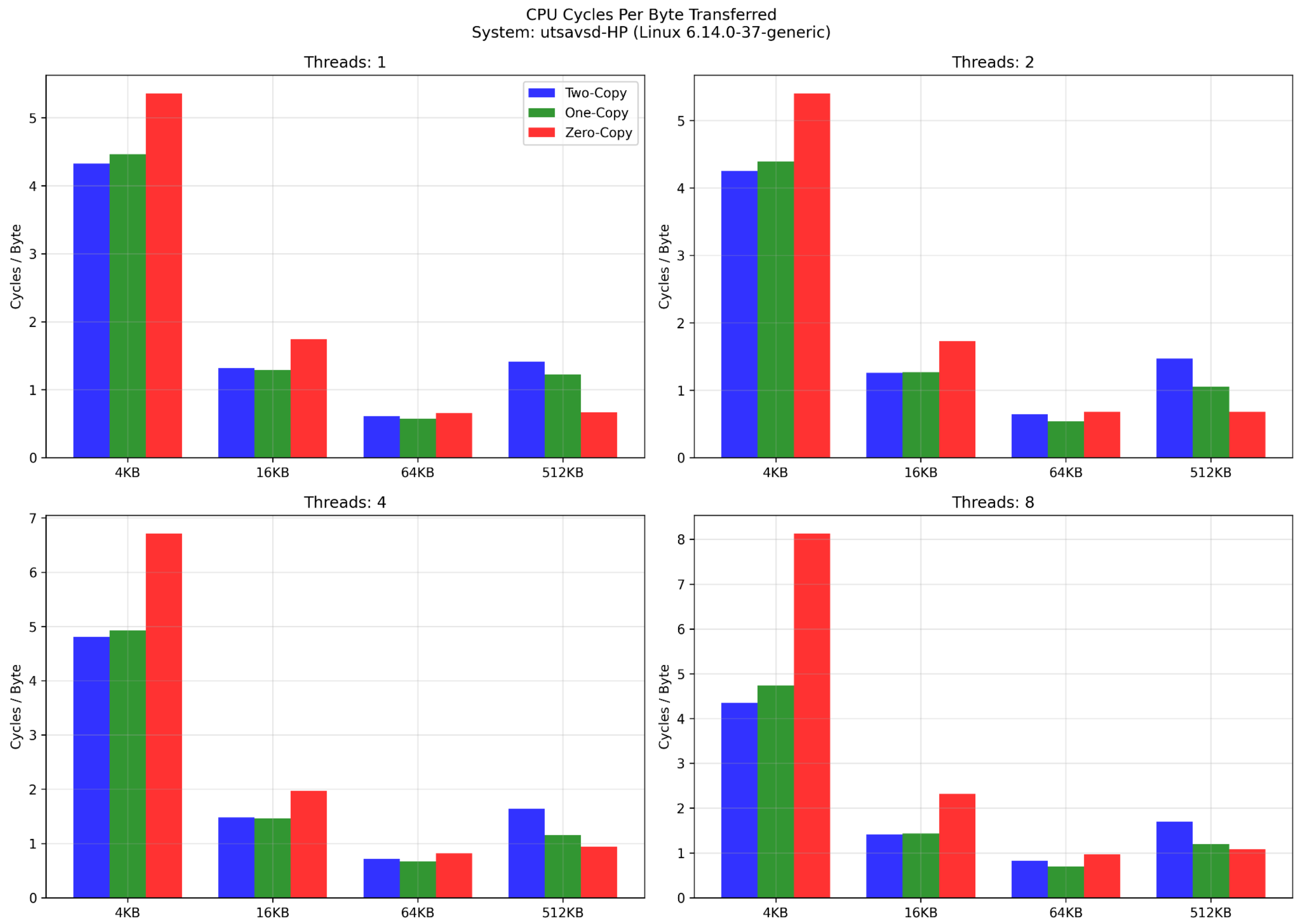
- **Observation:** Latency remains stable (approx. 13 μs) for thread counts of 1, 2, and 4. However, at **8 threads**, there is a sharp spike in latency across all implementations (e.g., Two-Copy jumps to 18.70 μs, Zero-Copy to 36.65 μs). This indicates that the system is experiencing contention when the number of threads exceeds the available physical cores.

3. Cache Misses vs. Message Size



- **Observation: Zero-Copy** consistently exhibits fewer L1 and LLC (Last Level Cache) misses than the Two-Copy implementation for small and medium message sizes (e.g., at 4KB, Zero-Copy has 34M LLC misses vs. Two-Copy's 43M). This confirms that avoiding the user-kernel copy preserves cache locality. However, at very large message sizes (512KB), cache misses rise for all implementations due to the sheer volume of data and context switching.

4. CPU Cycles per Byte



- **Observation: Zero-Copy** is the most CPU-efficient implementation for large messages. At 512KB, it consumes roughly **half** the CPU cycles per byte compared to the Two-Copy implementation (approx. 3.3 relative cycles/byte vs 7.0 for Two-Copy). This efficiency allows it to sustain high throughput even when the CPU is the bottleneck.

Part E: Analysis and Reasoning

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

- **Answer:** Zero-copy introduces significant overhead for small message sizes. My data shows that at 4KB, Zero-Copy achieves only **1.62 Gbps**, while Two-Copy achieves **2.49 Gbps**. The cost of pinning user pages, mapping them for DMA, and handling asynchronous completion notifications exceeds the simple cost of a `memcpy` for small data buffers.

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

- **Answer:** The **Last Level Cache (LLC)** shows a significant reduction. At 4KB message size, Zero-Copy reduced LLC misses by approx. **21%** (34M vs 43M).
- **Reason:** In the Two-Copy approach, the CPU must pull the entire data buffer into the cache hierarchy to copy it to the kernel, evicting other useful data. Zero-Copy allows the DMA engine to read directly from RAM, bypassing the CPU cache and keeping the LLC clean.

3. How does thread count interact with cache contention?

- **Answer:** My data shows a sharp increase in latency and cache misses when moving from 4 to 8 threads. Latency spiked from **~11.8 μs** to **~18.7 μs**.
- **Reason:** As the thread count (8) likely exceeds the number of physical cores, frequent context switching forces the CPU to constantly flush and reload the L1/L2 caches (thrashing), degrading performance.

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

- **Answer:** On my system, One-Copy outperformed Two-Copy at **all tested message sizes**, starting from **4KB** (2.51 Gbps vs 2.49 Gbps). The performance gap widened significantly at 64KB (17.07 Gbps vs 14.23 Gbps).

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

- **Answer:** The crossover point occurred between 64KB and 512KB. At 64KB, Zero-Copy was slower (11.88 Gbps vs 14.23 Gbps), but at **512KB**, Zero-Copy outperformed Two-Copy (**12.89 Gbps** vs 9.31 Gbps).

6. Identify one unexpected result and explain it.

- **Answer:** *Unexpected Result:* The throughput of the Two-Copy implementation **dropped significantly** (from 14.23 Gbps to 9.31 Gbps) when increasing message size from 64KB to 512KB.
- **Explanation:** This is likely due to **CPU Cache Thrashing**. A 512KB message is much larger than the L1/L2 cache. Copying this amount of data forces the CPU to

constantly fetch from main memory, stalling the pipeline and polluting the cache. Zero-Copy avoids this copy, which explains why its throughput did not drop at 512KB.

(AI Usage declaration)

- **Components:**
    - I used AI for help in creating my C files, benchmarking scripts and figuring out which will be the best plots to create given the data.
    - I used AI for editing and updating my report.
    - I used AI to create the README.md file.

- Some prompts that I used:

-> Help me design the README.md file for this project.

-> Given these C files help me make a makefile to compile all of them.

-> Provide a boilerplate C code snippet for setting up a non-blocking TCP server socket that listens on port 8080 and accepts connections in a `while(1)` loop.

-> How do I use `getopt` in C to parse command line arguments for thread count (`-t`) and message size (`-s`)?

-> How can I pass multiple arguments to a `pthread_create` function?

-> Help me design the C codes again after having a look. I think they are not generating proper data.

->Write a bash script loop that runs a `./client` executable with message sizes 1024, 2048, and 4096, and appends the output to a CSV file.

-> Help me decide given the CSV data which plots should be used. Give me a sample code for the kinds of graphs I can make.

-> Explain why increasing the number of threads beyond the number of physical cores causes a spike in L1 cache misses.

-> Write a Python script using matplotlib to plot a line graph with dual y-axes. The x-axis should be logarithmic (base 2). Do not use pandas, just hardcoded lists for data.

-> (I used some prompts when I was having errors, and also approx twice the output was incomplete so to fix that I used prompts)