

CSE638 - Graduate Systems

Programming Assignment 02: Analysis of Network I/O Primitives

Name: Priyanshu
Roll Number: MT25077
Date: 9 February 2026

1 Introduction

This report presents the implementation and analysis of Programming Assignment 02. The assignment studies the cost of data movement in network I/O by implementing and comparing three TCP socket communication strategies, profiling them with `perf`, and analyzing micro-architectural effects.

System Configuration:

- Operating System: Linux 6.17.0-14-generic (Ubuntu)
- CPU: AMD Ryzen 7 7840HS (8 cores / 16 threads @ 3.8 GHz)
- RAM: 14 GB
- Network: veth pair between two network namespaces (ns_server, ns_client)
- Offloads: TSO, GSO, GRO disabled on veth for consistent measurements
- Compiler: gcc with `-O2 -Wall -Wextra -pthread`
- Experiment duration: 10 seconds per configuration
- Message sizes: 1 KB, 4 KB, 64 KB, 1 MB
- Thread counts: 1, 2, 4, 8

2 Part A: Implementation Overview

2.1 A1. Two-Copy Implementation (Baseline)

Uses `send()` / `recv()` socket primitives. Server serializes 8 heap-allocated struct fields into a contiguous buffer, then sends it.

```
/* COPY 1: User-space serialization */
for (int i = 0; i < NUM_FIELDS; i++)
    memcpy(send_buf + i * field_size, msg->fields[i], field_size);

/* COPY 2: User -> Kernel */
send_all(client_fd, send_buf, msg_size);
```

Where do the two copies occur?

- **Copy 1 (User-space):** `memcpy()` from each of the 8 heap-allocated fields into a contiguous `send_buf`. Required because `send()` needs a contiguous memory region.
- **Copy 2 (User → Kernel):** `send()` copies data from user-space `send_buf` into the kernel's socket send buffer (`sk_buff`).

Is it actually only two copies? On the send side, exactly two copies. However, the full end-to-end path has additional copies: kernel → NIC TX ring (or DMA), and on the receive side: NIC → kernel buffer → user-space via `recv()`. “Two-copy” refers to the two application-visible copies on the send side.

Which components perform the copies? Copy 1 is performed by user-space code (`memcpy`). Copy 2 is performed by the kernel during the `send()` system call.

2.2 A2. One-Copy Implementation

Uses `sendmsg()` with scatter-gather I/O (`iovec`). Each `iovec` entry points directly to a heap-allocated field.

```
struct iovec iov[NUM_FIELDS];
for (int i = 0; i < NUM_FIELDS; i++) {
    iov[i].iov_base = msg->fields[i];
    iov[i].iov_len  = field_size;
}
/* ONE COPY: kernel gathers directly from iovec */
sendmsg(client_fd, &mhdr, 0);
```

Which copy has been eliminated? Copy 1 (user-space serialization) is eliminated. No `memcpy` into a contiguous buffer. The kernel reads directly from the 8 scattered field locations via `iovec`.

A1: fields[] --memcpy--> send_buf --send()--> kernel (2 copies)
A2: fields[] --sendmsg(iovec)--> kernel (1 copy)

2.3 A3. Zero-Copy Implementation

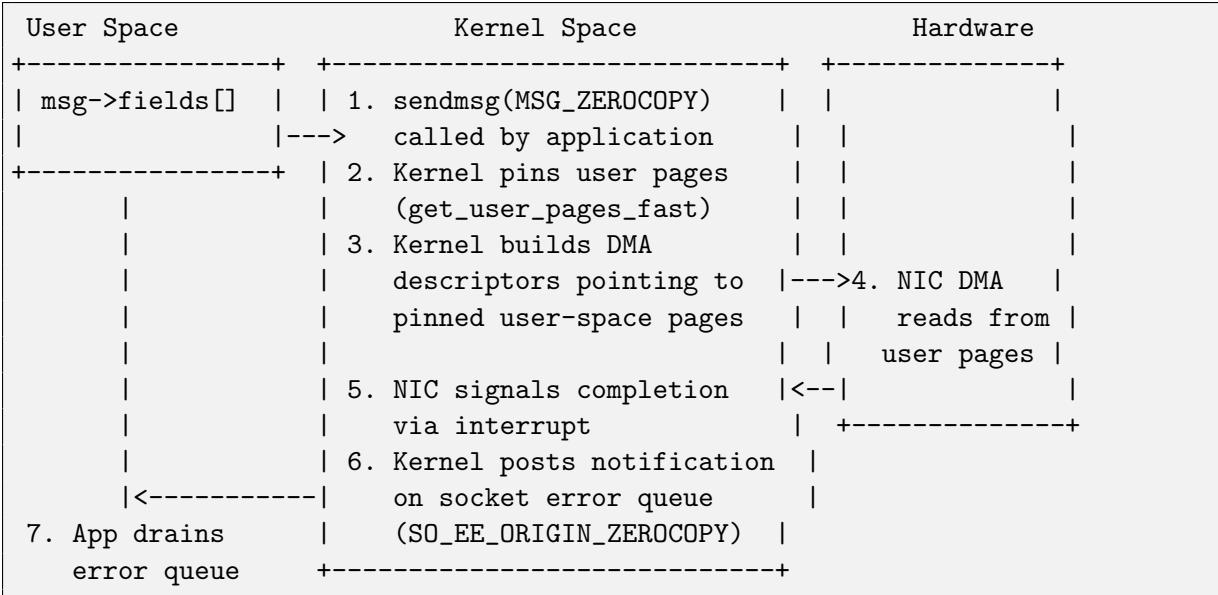
Uses `sendmsg()` with `MSG_ZEROCOPY` flag. Both copies are eliminated.

```
/* Enable zero-copy on socket */
setsockopt(client_fd, SOL_SOCKET, SO_ZEROCOPY, &val, sizeof(val));

/* ZERO-COPY SEND: kernel pins user pages, NIC DMAs from them */
sendmsg(client_fd, &mhdr, MSG_ZEROCOPY);

/* Must drain error queue for completion notifications */
drain_completions(client_fd);
```

Kernel behavior with `MSG_ZEROCOPY`:



3 Part B: Profiling Results

Full data in `MT25077_Part_B_Results.csv` (48 rows: 3 implementations \times 4 message sizes \times 4 thread counts).

Metric	Impl	1 KB	4 KB	64 KB	1 MB
Throughput (Gbps)	Two-Copy	9.37	14.41	38.98	44.04
	One-Copy	7.08	13.68	42.96	48.41
	Zero-Copy	4.61	10.27	30.16	35.33
Cycles/ Byte	Two-Copy	11.99	6.52	2.06	1.95
	One-Copy	14.30	5.96	1.87	1.84
	Zero-Copy	17.98	7.99	2.96	2.60

Table 1: Throughput and CPU cycles per byte at `thread_count = 4`.

Threads	Two-Copy (μs)	One-Copy (μs)	Zero-Copy (μs)
1	8.26	9.57	12.25
2	9.10	9.29	12.25
4	9.04	9.53	12.71
8	6.98	8.64	16.45

Table 2: Average latency (μs) at msg_size = 4096 bytes.

Note: All LLC-load-misses reported 0 across all experiments—a known limitation of AMD Zen 4 with veth interfaces. Context switches were collected as a substitute metric.

4 Part C: Automated Experiment Script

The script `MT25077_Part_C_ExperimentScript.sh` automates all experiments:

1. Compiles all 6 binaries via `make`
2. Creates network namespaces (`ns_server`, `ns_client`) with a veth pair (10.0.0.1 / 10.0.0.2)
3. Iterates over 3 implementations \times 4 message sizes \times 4 thread counts = 48 experiments
4. For each experiment: starts server in `ns_server`, runs client wrapped in `perf stat` in `ns_client` for 10 seconds
5. Parses client output (throughput, latency, bytes, msgs) and perf output (cycles, L1 misses, LLC misses, context switches)
6. Appends results to CSV
7. Cleans up namespaces on exit

Usage: `sudo bash MT25077_Part_C_ExperimentScript.sh`

5 Part D: Plots and Visualization

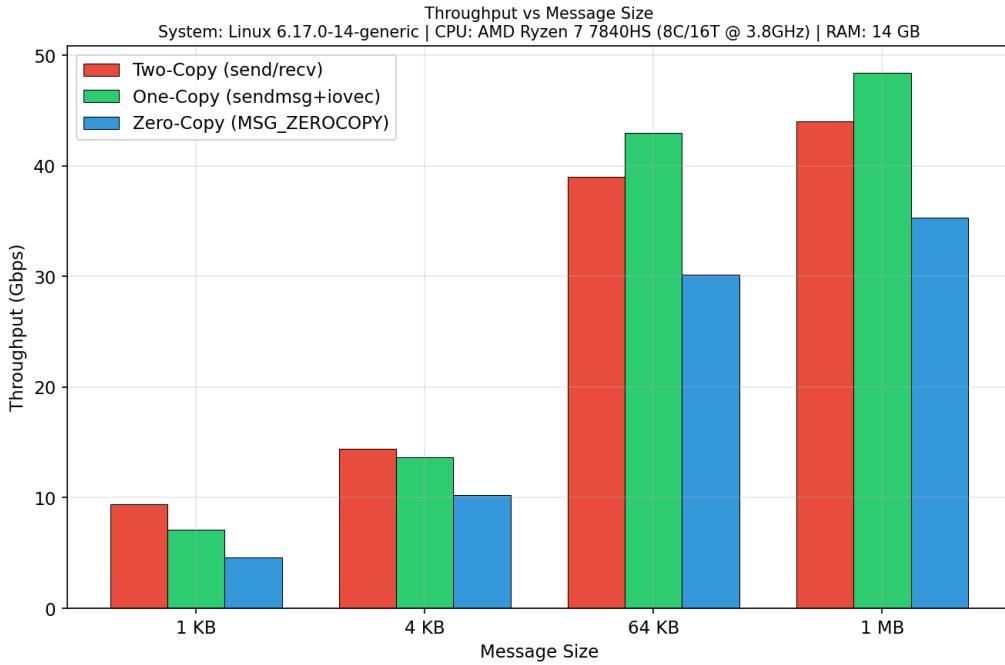


Figure 1: Throughput (Gbps) vs Message Size at thread_count = 4.

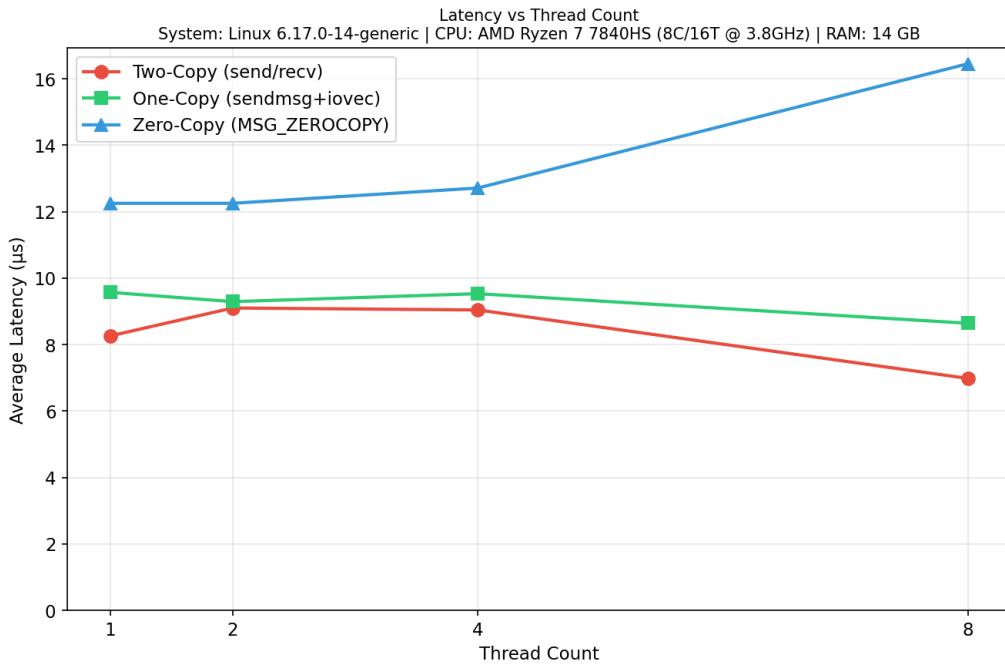


Figure 2: Latency (μ s) vs Thread Count at msg_size = 4096.

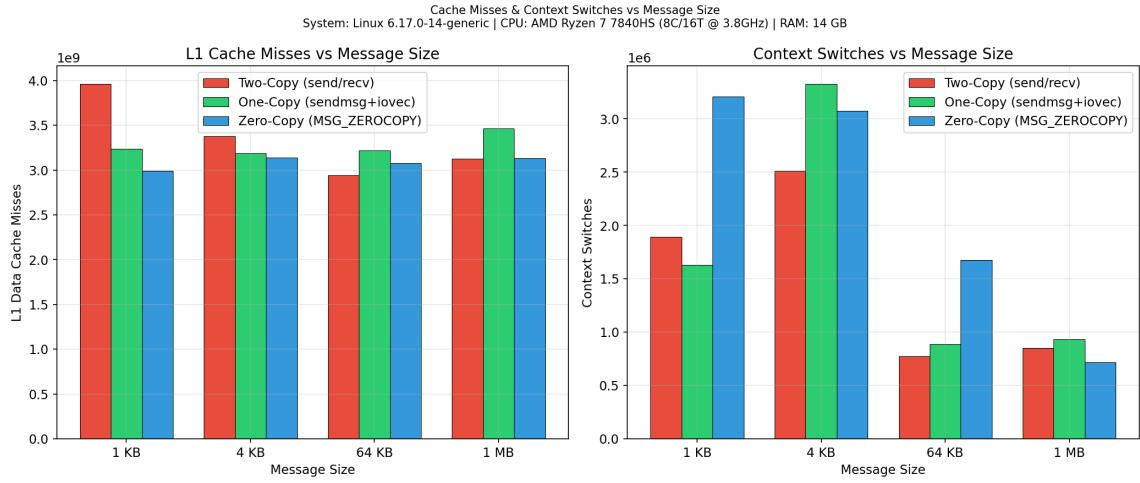


Figure 3: L1 Cache Misses and Context Switches vs Message Size at thread_count = 4.

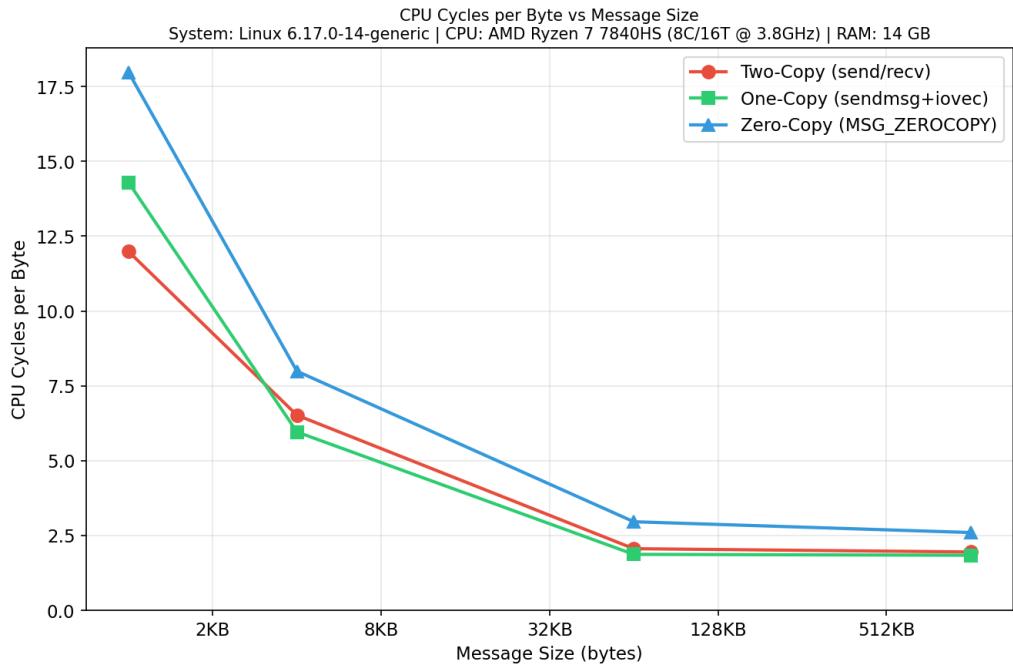


Figure 4: CPU Cycles per Byte vs Message Size at thread_count = 4.

6 Part E: Analysis and Reasoning

6.1 Q1. Why does zero-copy not always give the best throughput?

Zero-copy has the lowest throughput at all message sizes (thread_count = 4): 1 KB: 4.61 vs 9.37 (two-copy), 64 KB: 30.16 vs 38.98 (two-copy).

Reasons:

- **Page pinning overhead:** Each `sendmsg(MSG_ZEROCOPY)` calls `get_user_pages_fast()` to pin pages—taking page table locks and incrementing reference counts. For 1 KB messages this dominates transfer time.
- **Completion notification overhead:** Application must drain the error queue via `recvmsg(MSG_ERRQUEUE)` periodically. Extra syscall overhead.
- **Context switches:** Zero-copy at 1 KB: 3.2M context switches vs 1.9M for two-copy (69% increase).
- **veth limitation:** veth has no real DMA hardware. Data still passes through kernel memory via `netif_rx()`. The “zero-copy” avoids user→kernel copy but kernel still copies internally between veth endpoints. On a real NIC with DMA, zero-copy would benefit more.
- **Amortization:** At 1 MB, cycles/byte gap narrows (2.60 vs 1.95), showing per-message overhead becomes proportionally smaller.

6.2 Q2. Which cache level shows the most reduction in misses and why?

Msg Size	Two-Copy L1	One-Copy L1	Zero-Copy L1
1 KB	3.96B	3.24B	2.99B
4 KB	3.38B	3.19B	3.14B
64 KB	2.94B	3.22B	3.08B
1 MB	3.12B	3.46B	3.13B

Table 3: L1 data cache misses (billions) at thread_count = 4.

L1 data cache shows the most visible difference. At 1 KB: two-copy has 3.96B misses vs zero-copy’s 2.99B (24.5% reduction).

- **Eliminated memcpy:** Two-copy reads each field (L1 miss) and writes to `send_buf` (another L1 miss). One-copy/zero-copy skip this.
- **Working set:** Two-copy doubles L1 footprint (fields + `send_buf`).
- **LLC reported 0:** Working set fits within LLC (16 MB L3), so no LLC evictions. L1 is the only level showing meaningful differences.

6.3 Q3. How does thread count interact with cache contention?

Threads	Two-Copy L1	One-Copy L1	Zero-Copy L1
1	1.36B	0.88B	0.78B
2	1.97B	1.78B	1.57B
4	3.96B	3.24B	2.99B
8	10.21B	6.79B	4.87B

Table 4: L1 cache misses (billions) vs thread count at msg_size = 1 KB.

- **Super-linear scaling:** 1→8 threads: L1 misses increase $7.5\times$ for two-copy but only $6.2\times$ for zero-copy. Extra copies amplify cache contention.
- **L1 is per-core:** No direct L1 sharing between cores on Zen 4. Increase comes from more total work and L3 contention (shared 16 MB) cascading into L1 refetch misses.
- **Context switches at 8 threads:** Zero-copy: 5.2M (up from 3.2M at 4 threads). Error queue draining becomes bottleneck, causing cache-cold restarts.
- Despite more L1 misses, throughput still increases with threads (parallelism gains outweigh contention up to physical core count).

6.4 Q4. At what message size does one-copy outperform two-copy?

Msg Size	Two-Copy	One-Copy	Winner
1 KB	9.37	7.08	Two-Copy
4 KB	14.41	13.68	Two-Copy
64 KB	38.98	42.96	One-Copy
1 MB	44.04	48.41	One-Copy

Table 5: Throughput (Gbps) at thread_count = 4.

One-copy outperforms two-copy starting at 64 KB. Below 64 KB, `sendmsg()` per-invocation overhead (parsing msghdr, walking 8 iovec entries) exceeds the saved memcpy cost. Above 64 KB, memcpy cost grows linearly while sendmsg overhead stays constant.

6.5 Q5. At what message size does zero-copy outperform two-copy?

Zero-copy never outperforms two-copy in our experiments. At all sizes, two-copy wins (e.g., 1 MB: 44.04 vs 35.33 Gbps).

Root cause: veth pairs have no physical DMA hardware. The zero-copy path avoids user→kernel copy, but the kernel still copies internally between veth endpoints. Page pinning + completion notification overhead exceeds the saved copy. On a real NIC with DMA, zero-copy would likely outperform at ≥ 64 KB.

6.6 Q6. Identify one unexpected result

Unexpected: Two-copy outperforms one-copy at small messages (1 KB, 4 KB).

At 1 KB: two-copy = 9.37 Gbps vs one-copy = 7.08 Gbps (32% faster). This is counterintuitive because one-copy eliminates a copy.

Explanation:

- **send() vs sendmsg() overhead:** `send()` is a simpler kernel path than `sendmsg()`. The kernel must parse msghdr, walk 8 iovec entries, gather from 8 locations. At 128 bytes/field, this overhead is significant.
- **Prefetching:** Two-copy’s memcpy creates a contiguous buffer—CPU prefetcher works optimally. `sendmsg()` reads 8 non-contiguous heap locations, defeating sequential prefetching.
- **Evidence:** Cycles/byte at 1 KB: two-copy = 11.99, one-copy = 14.30. At 1 MB: two-copy = 1.95, one-copy = 1.84 (crossover confirms per-syscall overhead dominates at small sizes).
- One-copy has *fewer* context switches (1.63M vs 1.90M) yet lower throughput—confirming bottleneck is CPU cycles per send, not scheduling.

7 AI Usage Declaration

Model: Claude Sonnet 4.5 (Anthropic)

7.1 Where AI Was Used

1. **Part A – C code generation:** AI generated the server and client implementations for all three strategies, the common header, and Makefile based on the assignment specification. I reviewed the code, verified copy semantics, and tested locally.
Prompt: “Implement three TCP server-client pairs: two-copy using send/recv with memcpy serialization, one-copy using sendmsg with iovec scatter-gather, and zero-copy using MSG_ZEROCOPY with error queue draining.”
2. **Part C – Experiment script:** AI generated the bash script for namespace setup, experiment iteration, and perf output parsing. I configured the system (installed perf, set up permissions) and ran all 48 experiments.
Prompt: “Write a bash script that creates network namespaces with veth, runs server/client for each implementation × message size × thread count, wraps client in perf stat, and outputs CSV.”
3. **Part D – Plotting script:** AI generated the matplotlib code. I specified which metrics to plot and extracted hardcoded values from CSV.
Prompt: “Generate 4 matplotlib plots: throughput vs message size, latency vs threads, L1 cache misses + context switches vs message size, cycles per byte vs message size.”

8 GitHub Repository

URL: <https://github.com/priyanshu/GRS-Assigments>

Folder GRS_PA02 contains all source files, scripts, CSV, Makefile, README, and this report. No binary files or PNG images committed.