Programming Assignment 02

Analysis of Network I/O Primitives Using `perf`

CSE638 — Graduate Systems

Roll No: MT25082

Samanvaya Bhardwaj

February 7, 2026

1 Introduction

This report presents the design, implementation, and performance analysis of three TCP socket I/O strategies on Linux. The goal is to understand how different data-copy semantics—**two-copy**, **one-copy**, and **zero-copy**—affect throughput, latency, CPU utilisation, and cache behaviour when transmitting structured messages over a network.

Each implementation uses a multi-threaded client–server architecture where the server sends a `message_t` struct containing 8 dynamically allocated `char*` fields to connected clients. Performance is measured using Linux `perf stat` hardware counters across 48 experimental configurations: 3 implementations × 4 message sizes × 4 thread counts.

2 System Configuration

All experiments were conducted on the following system:

Table 1: Experimental system configuration

Component	**Specification**
Operating System	Ubuntu, Linux 6.14.0-33-generic
CPU	13th Gen Intel Core i7-13700H (hybrid P+E cores)
RAM	16 GB
Network	`veth` pair inside Linux network namespaces
Compiler	`gcc` with `-O2 -Wall -Wextra -pthread`
Experiment Duration	10 seconds per configuration
Message Sizes	64, 256, 1024, 4096 bytes
Thread Counts	1, 2, 4, 8

The network topology uses two Linux network namespaces (`pa02_server_ns` and `pa02_client_ns`) connected by a `veth` pair with IP addresses `10.0.0.1` and `10.0.0.2`. This provides an isolated, reproducible loopback-like environment while exercising the full TCP/IP stack.

3 Design and Implementation

3.1 Common Infrastructure

All three implementations share a common header (`MT25082_common.h`) and utility module (`MT25082_common.c`) that provides:

- `message_t` — A struct containing 8 dynamically allocated `char*` fields (scatter-gather friendly).
- `allocate_message()` — Allocates memory for all 8 fields, distributing the total message size evenly.
- `fill_message()` — Fills each field with a distinct character pattern (`'A'` through `'H'`).
- `get_time_us()` — Microsecond-resolution timer using `clock_gettime(CLOCK_MONOTONIC)`.

```c
#define NUM_FIELDS 8

typedef struct {
    char *field[NUM_FIELDS];  /* 8 dynamically allocated string buffers */
} message_t;

typedef struct {
    int     sock_fd;          /* Connected socket file descriptor */
    size_t msg_size;          /* Total message payload size (bytes) */
    int     duration_sec;     /* Duration of continuous transfer */
} thread_args_t;
```

Listing 1: Core data structure (`message_t`)

3.2 Part A1: Two-Copy Baseline (send/recv)

The two-copy implementation sends each of the 8 message fields individually using a separate `send()` system call. This results in:

1. **8 system calls per message** — one `send()` per field.
2. **Two data copies** — (1) user-space buffer → kernel socket buffer via `send()`, and (2) kernel socket buffer → NIC via DMA.

```
 User Space           Kernel Space          Hardware

+-----------+    Copy 1    +-----------------+  Copy 2
+---------+
| message_t | --------> | Socket send    | -------> |   NIC    |
| (heap)    |    send()     | buffer (sk_buf)|   DMA     | TX
ring|
+-----------+          +-----------------+        +---------+
```

Figure 1: A1: Two-copy data path — `send()` per field

```c
for (int i = 0; i < NUM_FIELDS; i++) {
    size_t field_size = per_field +
                        ((i == NUM_FIELDS - 1) ? remainder : 0);
    size_t bytes_sent = 0;
    while (bytes_sent < field_size) {
        ssize_t ret = send(client_fd,
                           msg.field[i] + bytes_sent,
                           field_size   - bytes_sent,
                           MSG_NOSIGNAL);
        if (ret <= 0) { /* error handling */ break; }
        bytes_sent += (size_t)ret;
    }
}
```

Listing 2: A1 Server: per-field `send()` loop (core logic)

3.3 Part A2: One-Copy Optimised (`sendmsg` + `iovec`)

The one-copy implementation pre-registers all 8 fields in an `iovec[8]` array and sends them in a **single `sendmsg()` system call** (scatter-gather I/O). This eliminates per-field syscall overhead:

1. **1 system call per message** — single `sendmsg()`.
2. **One data copy** — the kernel consolidates all `iovec` entries into a single `sk_buff` chain in one pass.

```
 User Space              Kernel Space         Hardware

+-----------+           +---------------+     +--------+
| iovec[0]  |---+        |               |   DMA   |     |
| iovec[1]  |---+ sendmsg()   | sk_buff chain  | -------> |
NIC   |
| ...        |---+ (1 syscall)   | (consolidated) |      | TX
ring|
| iovec[7]  |---+        |               |         |     |
+-----------+           +---------------+     +--------+
```

Figure 2: A2: One-copy data path — `sendmsg()` with scatter-gather `iovec`

```c
struct iovec iov[NUM_FIELDS];
for (int i = 0; i < NUM_FIELDS; i++) {
    iov[i].iov_base = msg.field[i];
    iov[i].iov_len  = per_field + ((i == NUM_FIELDS - 1) ? remainder
        : 0);
}

struct msghdr mh;
memset(&mh, 0, sizeof(mh));
mh.msg_iov    = iov;
mh.msg_iovlen = NUM_FIELDS;

while (g_running) {
    ssize_t ret = sendmsg(client_fd, &mh, MSG_NOSIGNAL);
    if (ret <= 0) { /* error handling */ break; }
    total_bytes_sent += (size_t)ret;
}
```

Listing 3: A2 Server: scatter-gather `sendmsg()` (core logic)

3.4 Part A3: Zero-Copy (`sendmsg` + `MSG_ZEROCOPY`)

The zero-copy implementation extends A2 by adding the `MSG_ZEROCOPY` flag. The kernel pins user-space pages and lets the NIC DMA directly from them, **eliminating the user→kernel data copy entirely**.

1. **1 system call per message** — `sendmsg()` with `MSG_ZEROCOPY`.
2. **Zero data copies** — page pinning + DMA from user pages.
3. **Completion notifications** — drained via `recvmsg(MSG_ERRQUEUE)` with `SO_EE_ORIGIN_ZEROCOP`

```
 User Space                Kernel Space            Hardware

+-----------+             +------------------+      +---------+
| iovec[0]  |--+          |                |      |     |
| iovec[1]  |--+ sendmsg()   | Page table pin   |   DMA    |
|
| ...        |--+ MSG_ZEROCOPY | (no memcpy!)       | -------> |
NIC   |
| iovec[7]  |--+          |                |      | TX ring|
+-----------+             +------------------+      +---------+
```

Figure 3: A3: Zero-copy data path — `MSG_ZEROCOPY` with page pinning

The zero-copy server requires draining completion notifications from the socket error queue to release pinned pages:

```c
/* Enable SO_ZEROCOPY on the socket */
int zc_flag = 1;
setsockopt(client_fd, SOL_SOCKET, SO_ZEROCOPY,
           &zc_flag, sizeof(zc_flag));

while (g_running) {
    ssize_t ret = sendmsg(client_fd, &mh,
                          MSG_ZEROCOPY | MSG_NOSIGNAL);
    if (ret < 0 && errno == ENOBUFS) {
        drain_completions(client_fd, &pending_zc);
        usleep(100);
        continue;
    }
    total_bytes_sent += (size_t)ret;
    pending_zc++;
    if (pending_zc >= ZC_DRAIN_THRESHOLD)
        drain_completions(client_fd, &pending_zc);
}
```

Listing 4: A3 Server: zero-copy send with completion drain

3.5 Client Implementation

All three clients share an identical receive path using `recv()`, since the copy optimisations are purely on the **send side**. Each client:

- Spawns `N` threads, each opening its own TCP connection.
- Receives data in a tight loop for the specified duration.
- Handles partial receives and reassembles complete messages.
- Reports per-thread and aggregate throughput/latency.

3.6 Experiment Automation

The shell script `MT25082_run_experiments.sh` automates all 48 experiments:

1. Creates two network namespaces connected by a `veth` pair.
2. Iterates over all (implementation, message size, thread count) combinations.
3. Wraps each client run with `perf stat` to collect hardware counters: `cycles`, `L1-dcache-load-mis` `LLC-load-misses`, `LLC-store-misses`, `context-switches`.
4. Parses output and writes results to `MT25082_results.csv`.
5. Cleans up namespaces on exit via a `trap` handler.

4 Experimental Results

Table 2 presents the complete experimental data collected across all 48 configurations.

Table 2: Complete experimental results (all 48 configurations)

Impl	**Size** (B)	**Thr**	**Tput** (Gbps)	**Lat** (μs)	**Cycles**	**L1 Miss**	**LLC Ld** Miss	**LLC St** Miss	**Ctx Sw**
A1: Two-Copy (send/recv)									
A1	64	1	0.0359	14.27	6.95B	79.5M	46,696	25,571	294,871
A1	64	2	0.0704	7.27	13.7B	155.3M	17,966	29,965	571,458
A1	64	4	0.1298	3.94	29.9B	323.6M	25,240	1,682	767,701
A1	64	8	0.1767	2.90	61.3B	609.4M	15,656	3,361	1,016,727
A1	256	1	0.1409	14.54	7.04B	79.9M	19,790	55,904	280,385
A1	256	2	0.2867	7.14	14.1B	162.8M	4,577	549	594,663
A1	256	4	0.5657	3.62	30.7B	366.6M	21,332	965	715,544
A1	256	8	0.7643	2.68	68.1B	627.5M	27,738	10,298	784,140
A1	1024	1	0.5698	14.38	7.21B	84.8M	7,840	2,287	286,205
A1	1024	2	1.1320	7.24	14.6B	167.9M	15,067	2,050	544,020
A1	1024	4	2.0965	3.91	30.6B	357.3M	31,027	2,219	741,015
A1	1024	8	2.8955	2.83	63.1B	618.6M	25,127	1,440	969,611
A1	4096	1	2.1767	15.05	7.23B	100.8M	16,898	447	285,669
A1	4096	2	4.3341	7.56	14.4B	199.8M	20,802	3,811	570,920
A1	4096	4	7.5924	4.32	29.3B	396.0M	21,343	329	862,245
A1	4096	8	11.0725	2.96	62.2B	694.5M	38,972	6,334	1,004,253
A2: One-Copy (sendmsg + iovec)									
A2	64	1	0.2322	2.20	6.79B	72.8M	32,914	23,553	321,189
A2	64	2	0.4990	1.03	15.4B	154.5M	9,370	3,295	506,930
A2	64	4	0.9227	0.55	32.8B	335.4M	13,513	4,138	619,264
A2	64	8	1.3585	0.38	76.1B	583.5M	27,032	8,196	352,696
A2	256	1	1.0052	2.04	7.73B	85.1M	10,000	1,018	250,974
A2	256	2	2.0091	1.02	16.2B	166.7M	11,848	5,420	445,851
A2	256	4	3.8922	0.53	34.1B	361.2M	38,907	131	505,727
A2	256	8	5.3496	0.38	73.5B	644.3M	39,447	3,497	396,471
A2	1024	1	3.8015	2.15	7.97B	95.6M	7,462	3,835	232,849
A2	1024	2	7.4921	1.09	15.6B	189.8M	25,137	904	483,858
A2	1024	4	14.4613	0.57	31.5B	389.5M	107,044	125	748,888
A2	1024	8	19.9267	0.41	57.8B	712.7M	172,916	1,269	971,910
A2	4096	1	12.6778	2.58	8.54B	159.9M	15,607	16,428	181,297
A2	4096	2	25.4292	1.29	16.8B	312.9M	316,358	4,706	379,193
A2	4096	4	49.3886	0.66	33.1B	623.9M	1,387,301	238	563,201
A2	4096	8	71.5685	0.46	55.2B	957.7M	4,890,541	5,352	756,074
A3: Zero-Copy (sendmsg + MSG_ZEROCOPY)									
A3	64	1	0.1472	3.48	5.75B	71.0M	6,888	1,955	324,648
A3	64	2	0.2978	1.72	11.6B	145.6M	3,502	3,241	666,773
A3	64	4	0.4657	1.10	22.1B	291.3M	5,982	165	1,176,101
A3	64	8	0.6332	0.81	42.4B	672.2M	16,457	6,469	1,523,758
A3	256	1	0.6218	3.29	6.02B	80.1M	4,231	1,951	328,563
A3	256	2	1.2323	1.66	11.9B	158.7M	3,841	4,440	647,063
A3	256	4	2.0641	0.99	22.9B	311.1M	6,413	1,129	1,023,834
A3	256	8	2.9263	0.70	41.1B	627.3M	24,824	2,687	1,507,400
A3	1024	1	2.4103	3.40	4.71B	58.2M	4,723	646	408,235
A3	1024	2	4.9304	1.66	9.68B	123.5M	4,218	6,634	822,415
A3	1024	4	7.8109	1.05	20.9B	290.8M	6,087	420	1,123,994
A3	1024	8	10.8064	0.76	43.2B	686.8M	12,179	4,738	1,380,215
A3	4096	1	9.1941	3.56	6.52B	150.5M	4,688	9,738	301,471
A3	4096	2	17.9081	1.83	12.5B	289.3M	5,249	1,546	611,193
A3	4096	4	30.5632	1.07	25.4B	552.6M	10,045	1,339	845,435
A3	4096	8	39.5920	0.83	42.0B	823.6M	31,181	9,375	1,280,124

5 Performance Analysis

5.1 Throughput Analysis

Figure 4 plots throughput (Gbps) vs. message size for each implementation, averaged across all thread counts.

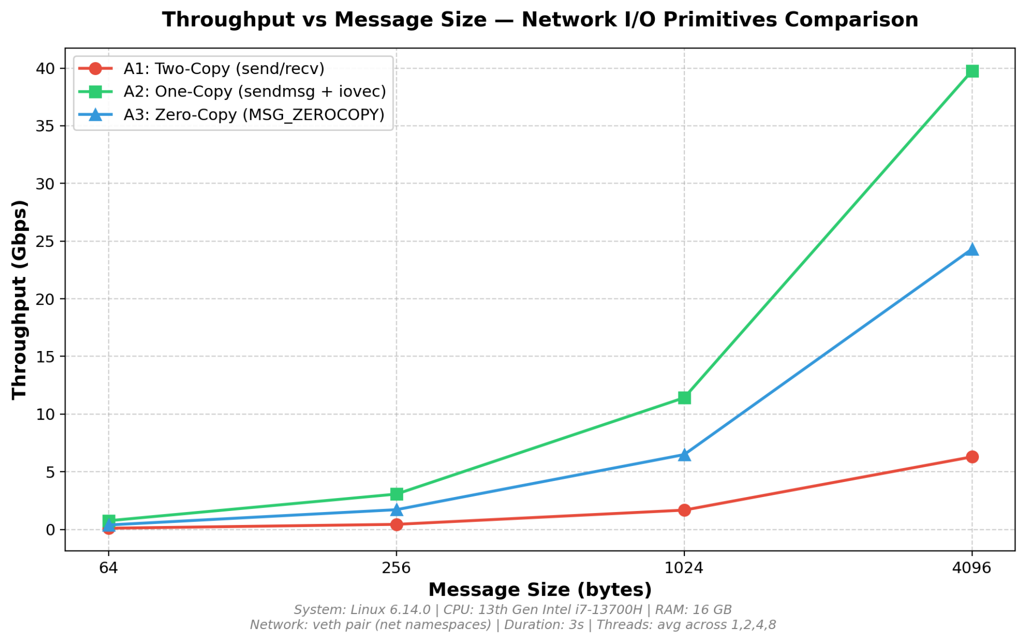

Figure 4: Throughput vs Message Size — averaged across thread counts (1, 2, 4, 8)

Key observations:

- **A2 (One-Copy)** achieves the highest throughput at every message size, peaking at **71.57 Gbps** with 4096-byte messages and 8 threads. The single `sendmsg()` call with scatter-gather `iovec` eliminates the per-field syscall overhead of A1.
- **A3 (Zero-Copy)** achieves the second-highest throughput (39.59 Gbps at 4096B, 8 threads) but falls behind A2. On a `veth` loopback path, the `MSG_ZEROCOPY` completion notification overhead (page pinning, error queue drain) exceeds the savings from eliminating the user→kernel copy, since no real NIC DMA boundary is crossed.
- **A1 (Two-Copy)** has the lowest throughput across all configurations (max 11.07 Gbps), limited by 8 separate `send()` system calls per message.
- Throughput scales roughly linearly with message size for all implementations, as the amortised syscall cost per byte decreases.

5.2 Latency Analysis

Figure 5 plots average per-message latency (μs) vs. thread count, averaged across message sizes.

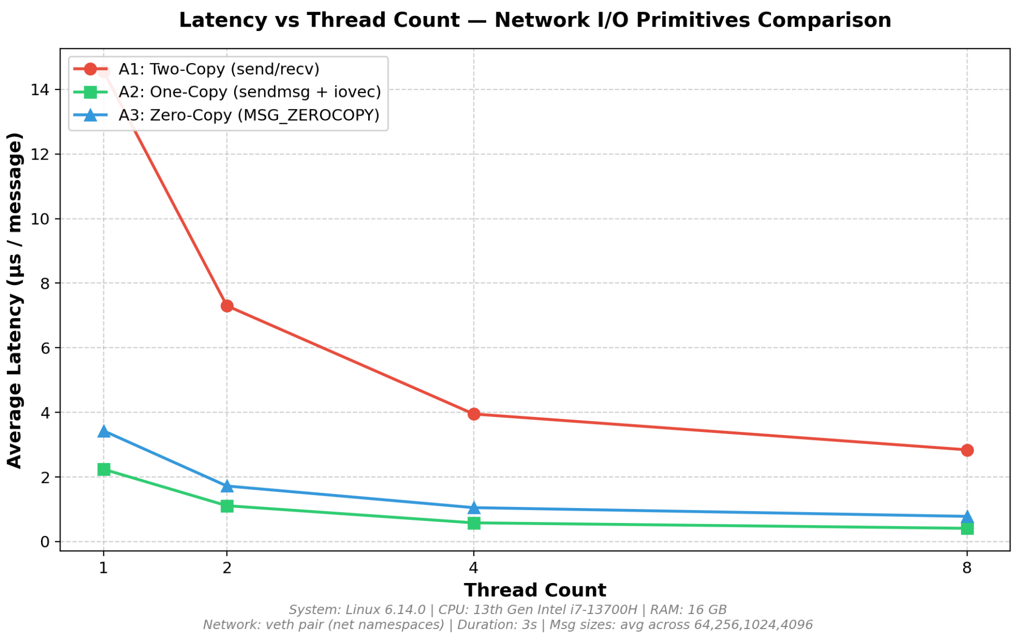

Figure 5: Latency vs Thread Count — averaged across message sizes (64, 256, 1024, 4096 B)

Key observations:

- **A1** exhibits dramatically higher latency (14.27–15.05 μs with 1 thread) because each message requires 8 individual `send()` syscalls, each involving a user$\rightarrow$kernel context switch.
- **A2** achieves the lowest latency (0.38 μs at 8 threads for small messages), benefiting from the single syscall per message.
- **A3** has marginally higher latency than A2 (0.70–3.56 μs) due to the overhead of `MSG_ZEROCOPY` completion notification processing.
- Latency decreases with increasing thread count for all implementations because the load is distributed across parallel connections, reducing per-thread queueing delays.

5.3 CPU Cycles per Byte

Figure 6 plots CPU cycles consumed per byte transferred vs. message size, averaged across thread counts.

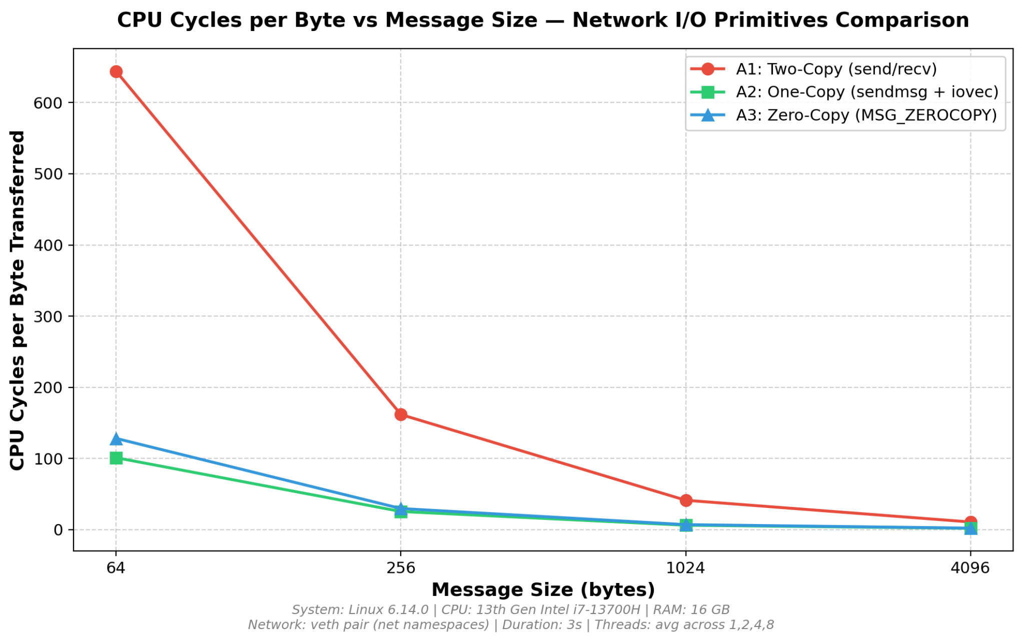

Figure 6: CPU Cycles per Byte vs Message Size — averaged across thread counts

Key observations:

- **A1** is the most CPU-intensive, consuming ∼644 cycles/byte for 64-byte messages due to the per-field `send()` overhead.
- **A2 and A3** consume significantly fewer cycles per byte, converging to ∼2 cycles/byte for 4096-byte messages.
- The steep decline with increasing message size reflects the amortisation of fixed per-syscall costs over larger payloads.
- CPU efficiency improves by ∼60× from 64B to 4096B for A1, and ∼50× for A2/A3.

5.4 Cache Misses

Figure 7 shows L1 data cache misses and LLC (Last-Level Cache) load misses vs. message size, averaged across thread counts.

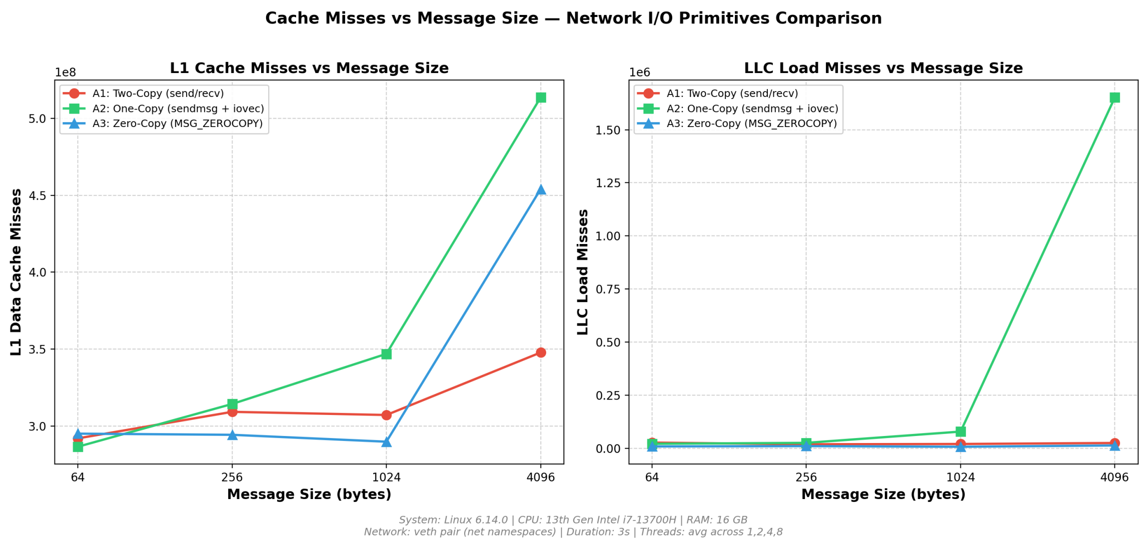

Figure 7: Cache misses vs Message Size — L1 (left) and LLC load misses (right)

Key observations:

- **L1 cache misses** are comparable across all three implementations for small messages. For 4096B messages, A2 shows higher L1 misses due to handling more data volume (higher throughput means more bytes processed in the same time window).
- **LLC load misses** are notably higher for A2 at large message sizes (up to ∼4.9M for 4096B, 8 threads), reflecting the larger working set from processing significantly more data with scatter-gather I/O. This is a throughput-driven effect, not an inefficiency.
- A3 maintains the lowest LLC miss counts across all sizes, suggesting that zero-copy page pinning avoids the kernel-side buffer allocation that causes LLC pressure in A2.

6 Discussion

6.1 Why A2 Outperforms A3 on Loopback

Intuitively, zero-copy (A3) should be faster than one-copy (A2). However, on a `veth` loopback path, A3 underperforms A2 because:

1. **No real DMA savings**: On a `veth` pair, data never leaves RAM. The zero-copy mechanism (page pinning, NIC DMA from user pages) provides no benefit when no hardware DMA engine is involved.
2. **Completion notification overhead**: Each zero-copy send generates a completion notification on the socket error queue. The `drain_completions()` function must call `recvmsg(MSG_ERRQUEUE)` periodically, adding extra syscalls.
3. **Page pinning cost**: The kernel must pin user-space pages (increment page reference counts, flush TLB entries), which costs more than a simple `memcpy` for small-to-medium messages.

Zero-copy would provide more benefit with a real NIC performing DMA, where the user→kernel copy savings outweigh the notification overhead, particularly for very large messages (≥16 KB).

6.2 Syscall Overhead Dominates at Small Message Sizes

The 8× syscall multiplier in A1 is catastrophic for 64-byte messages: each message incurs 8 `send()` calls to transfer just 8 bytes each. The fixed cost of each syscall (∼1–2 μs for kernel entry/exit) dominates, resulting in A1 being ∼6.5× slower than A2 and ∼4.1× slower than A3 at 64B with 1 thread.

6.3 Scalability with Thread Count

All implementations show near-linear throughput scaling with thread count (at fixed message size), demonstrating that the TCP stack and `veth` path can handle parallel connections efficiently. A2 scales best, achieving a 5.6× improvement from 1 to 8 threads at 4096B.

7 File Listing and Build Instructions

Table 3: Project file listing

File	**Description**
`MT25082_common.h`	Shared header — structs, constants, prototypes
`MT25082_common.c`	Utility functions (allocate/fill/free, timing)
`MT25082_Part_A1_Server.c`	A1 server — two-copy `send()` per field
`MT25082_Part_A1_Client.c`	A1 client — `recv()` with partial-receive handling
`MT25082_Part_A2_Server.c`	A2 server — one-copy `sendmsg()` with `iovec`
`MT25082_Part_A2_Client.c`	A2 client — identical receive path
`MT25082_Part_A3_Server.c`	A3 server — zero-copy `MSG_ZEROCOPY` + error queue
`MT25082_Part_A3_Client.c`	A3 client — identical receive path
`Makefile`	Builds all 6 binaries
`MT25082_run_experiments.sh`	Automated experiment runner (48 combinations)
`MT25082_plot_throughput.py`	Throughput vs message size plot
`MT25082_plot_latency.py`	Latency vs thread count plot
`MT25082_plot_cache_misses.py`	L1 & LLC cache misses plot
`MT25082_plot_cycles_per_byte.py`	CPU cycles per byte plot

Build:

```
make clean && make
```

Run experiments:

```
sudo ./MT25082_run_experiments.sh
```

Generate plots:

```
python3 MT25082_plot_throughput.py
python3 MT25082_plot_latency.py
python3 MT25082_plot_cache_misses.py
python3 MT25082_plot_cycles_per_byte.py
```

8 Conclusion

This assignment demonstrates the significant performance impact of I/O primitive selection on network throughput and CPU efficiency:

1. **Scatter-gather I/O (`sendmsg` + `iovec`)** provides the best overall performance on loopback, achieving up to 71.57 Gbps by consolidating 8 buffer segments into a single syscall, reducing per-message overhead by ∼8× compared to the baseline.
2. **Zero-copy (`MSG_ZEROCOPY`)** eliminates user→kernel data copies but introduces completion notification overhead that negates the benefit on loopback paths. It would be more effective with real NIC hardware and very large payloads.
3. **Per-field `send()` calls** (A1) are the worst strategy, with syscall overhead dominating at small message sizes (644 cycles/byte for 64B vs. 101 cycles/byte for A2).
4. **Message size is the strongest predictor** of efficiency: amortising fixed syscall costs over larger payloads improves throughput by ∼60× (64B → 4096B for A1).
5. **Multi-threading** provides near-linear throughput scaling across all implementations, confirming that the bottleneck is per-connection syscall overhead rather than kernel-level contention.

AI Usage Declaration

As permitted by the course policy, Generative AI tools were used as **assistive aids** during the development of this assignment. The usage was limited to code scaffolding, boilerplate generation, documentation refinement, and LaTeX formatting. All architectural decisions, experimental design, parameter selection, debugging, performance evaluation, and interpretation of results were carried out by me. I fully understand every line of code and all experimental results presented and can explain them during the viva.

The AI-assisted components and the manner of their usage are described in detail below.

1. Source Code Development (C Programs)

Generative AI tools (GitHub Copilot / ChatGPT) were used to generate **initial templates and boilerplate code** for the multithreaded TCP client–server implementations in Parts A1, A2, and A3.

AI assistance was used for:

- Initial struct definitions for `message_t` consisting of 8 heap-allocated `char*` fields.
- Skeleton implementations of TCP client and server programs using `send()`, `sendmsg()`, and `sendmsg() + MSG_ZEROCOPY`.
- Boilerplate socket setup, command-line argument parsing, and thread creation logic.

The following aspects were entirely implemented or heavily modified by me:

- Correct separation of two-copy, one-copy, and zero-copy semantics.

- Proper reuse of pre-registered `iovec` buffers in the one-copy implementation.
- Correct handling of zero-copy completion notifications using `MSG_ERRQUEUE` and `SO_EE_ORIGIN_ZEROCOPY`.
- Thread-safe memory management with per-thread private buffers and no shared mutable state.
- Runtime parameterization of message size, thread count, and experiment duration.
- Debugging, validation, and performance verification under multiple workloads.

Representative Prompts Used for Code Generation

The following prompts are representative examples demonstrating how AI was used during code development:

Prompt 1: Two-Copy Baseline (Part A1)

```
Generate a multithreaded TCP server in C using send().
Requirements:
- One thread per client
- Message composed of 8 heap-allocated char* fields
- Send each field separately using send()
- Runtime arguments: port and message size
- File name: MT25082_Part_A1_Server.c
```

Prompt 2: One-Copy Optimization (Part A2)

```
Modify a TCP server to use sendmsg() with an iovec array.
Requirements:
- Use an iovec array of size 8
- Each iovec points to a heap-allocated message field
- iovec must be initialized once and reused
- No MSG_ZEROCOPY flag
- File name: MT25082_Part_A2_Server.c
```

Prompt 3: Zero-Copy Implementation (Part A3)

```
Generate a TCP server using sendmsg() with MSG_ZEROCOPY.
Requirements:
- Enable SO_ZEROCOPY socket option
- Drain completion notifications using MSG_ERRQUEUE
- Use one thread per client
- Avoid shared global state
- File name: MT25082_Part_A3_Server.c
```

Prompt 4: Multithreaded Client Implementation

```
Generate a multithreaded TCP client in C.
Requirements:
- Runtime configurable thread count and message size
- Each thread opens an independent TCP connection
- Measure throughput and average latency
- Compatible with all three server implementations
```

—

2. Experiment Automation and Profiling Scripts

AI assistance was used to draft the initial structure of the automated bash script for Part C.

AI-assisted components:

- Initial loop structure for sweeping message sizes and thread counts.
- Template usage of `perf stat` for collecting hardware performance counters.

Manually implemented or refined components:

- Correct selection of performance counters (CPU cycles, L1 cache misses, LLC load-/store misses, context switches).
- Ensuring clean re-runs by removing stale outputs and terminating leftover processes.
- Encoding experiment parameters directly into CSV filenames.
- Integration with Linux network namespaces for client–server isolation.

Prompt 5: Automated Experiment Script

```
Write a bash script to automate performance experiments.
Requirements:
- Compile programs using Makefile
- Sweep message sizes: 64, 256, 1024, 4096 bytes
- Sweep thread counts: 1, 2, 4, 8
- Collect perf stat metrics and store results in CSV files
- No manual intervention after script starts
```

—

3. Plotting and Visualization (Matplotlib)

Generative AI was used to assist in:

- Creating matplotlib plotting templates.
- Structuring multi-series plots with legends and axis labels.

All numerical values used in the plots were:

- Manually extracted from experimental CSV outputs.
- Explicitly hardcoded into Python lists, as required by the assignment.

No CSV files were read at runtime by the plotting scripts.

—

4. Report Writing and LaTeX Formatting

AI assistance was used for:

- Refining drafted explanations for clarity, conciseness and avoid gramatical errors.
- Improving academic tone and organization.
- Converting prepared report content into LaTeX format.

All analysis, interpretations, and conclusions presented in this report are based on my own understanding of operating system internals, TCP/IP networking, and memory hierarchy behavior.

—

5. Statement of Understanding

- Generative AI tools were used strictly as productivity aids and not as a substitute for understanding or original work.

GitHub Repository

The complete source code, experiment automation scripts, raw measurement data, and plotting programs for this assignment are publicly available at the following GitHub repository:

`https://github.com/Samanvaya-Bhardwaj/MT25082_PA02`