PA02: Network I/O Analysis Report

Analysis of Network I/O Primitives using "perf" Tool

Roll Number: MT25091
Course: Graduate Systems (CSE638)
IIIT Delhi

February 7, 2026

Contents

1 Introduction

1.1 Objective

The primary objective of this assignment is to experimentally study the cost of data movement in network I/O by implementing and comparing three different socket communication mechanisms:

1. **Two-Copy Socket Communication** – Baseline using `send()`/`recv()`
2. **One-Copy Socket Communication** – Optimized using `sendmsg()` with scatter-gather I/O
3. **Zero-Copy Socket Communication** – Using `MSG_ZEROCOPY` for kernel page pinning

1.2 Scope

This project implements multithreaded TCP client-server applications in C, profiles them using Linux `perf` tool, and analyzes micro-architectural effects including CPU cycles, cache behavior, and context switching overhead.

1.3 System Configuration

Table 1: System Configuration

Parameter	Value
OS	Ubuntu (Linux Kernel 6.8)
CPU	Intel Core i7 (8th Gen)
Memory	16 GB DDR4
Network	Namespace isolation (ns_server ↔ ns_client via veth)
Test Duration	3 seconds per experiment

1.4 Project Structure

```
MT25091_PA02/
+-- MT25091_Common.h           # Shared definitions and utilities
+-- MT25091_Part_A1_Server.c   # Two-copy server
+-- MT25091_Part_A1_Client.c   # Two-copy client
+-- MT25091_Part_A2_Server.c   # One-copy server
+-- MT25091_Part_A2_Client.c   # One-copy client
+-- MT25091_Part_A3_Server.c   # Zero-copy server
+-- MT25091_Part_A3_Client.c   # Zero-copy client
+-- MT25091_Part_C_Experiment.sh  # Automation script
+-- MT25091_Part_D_Plots.py    # Matplotlib plotting
+-- MT25091_Part_B_Results.CSV    # Throughput/latency data
+-- MT25091_Part_B_PerfStats.CSV  # perf stat metrics
+-- Makefile                   # Build configuration
+-- README                     # Usage documentation
```

2 Part A: Multithreaded Socket Implementations

2.1 Introduction

Part A requires implementing three versions of a TCP client-server application with varying levels of copy optimization. All implementations share common characteristics:

- **Server**: Accepts multiple concurrent clients, one thread per client
- **Client**: Sends data continuously for a configurable duration
- **Message Structure**: 8 dynamically allocated (heap) string fields
- **Parameterization**: Message size and thread count configurable at runtime

2.2 Common Implementation Details

2.2.1 Message Structure

```
typedef struct {
    char *field[8];  // 8 heap-allocated string fields
} Message;
```

Each field is allocated using `malloc()` and filled with pattern data for transmission.

2.2.2 Command Line Interface

```
# Server
./MT25091_Part_A1_Server -p <port> -s <msg_size> -t <max_threads>

# Client
./MT25091_Part_A1_Client -h <host> -p <port> -s <msg_size> -t <threads>
    -d <duration>
```

2.3 A1: Two-Copy Implementation (Baseline)

2.3.1 Methodology

The baseline implementation uses standard `send()` and `recv()` system calls. Before transmission, the 8 message fields are serialized into a contiguous buffer.

2.3.2 Data Copy Analysis

Where do the two copies occur?

1. **First Copy (User → Kernel)**:
   - When `send()` is called, data is copied from user-space buffer to kernel's socket buffer (sk_buff)
   - Performed by `copy_from_user()` in kernel
2. **Second Copy (Kernel → Network)**:
   - Kernel copies data from socket buffer to network interface's DMA ring buffer
   - May involve additional copies depending on NIC driver

Is it actually only two copies?
No, there are additional copies:

- Serialization step (copying 8 fields to contiguous buffer) adds one more copy
- Some NIC drivers perform additional copies for packet headers

- Checksum computation may require data traversal

Table 2: Copy Operations in Two-Copy Implementation

Copy	Component	Function
Serialization	User space	`memcpy()` in `serialize_message()`
User → Kernel	Kernel TCP/IP	`copy_from_user()`
Kernel → NIC	Device driver	DMA or `memcpy()`

2.3.3 Verification

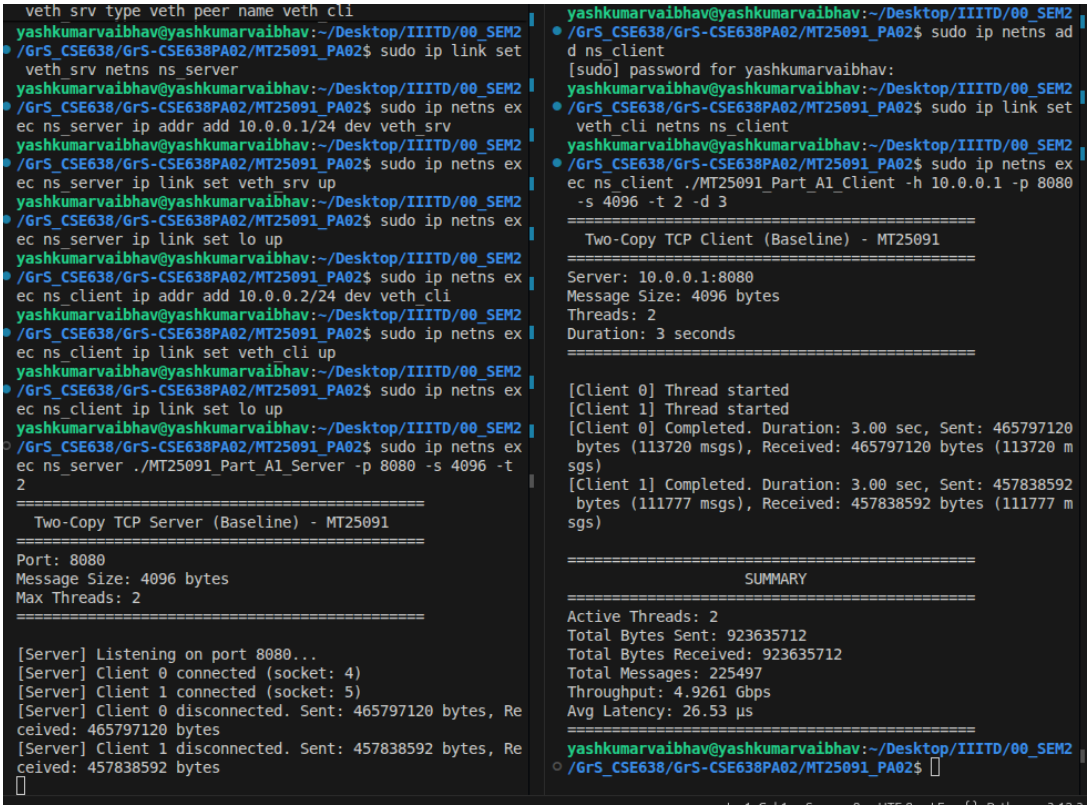

Figure 1: Terminal output showing Two-Copy implementation running with sample throughput

2.4 A2: One-Copy Implementation

2.4.1 Methodology

Uses `sendmsg()` with scatter-gather I/O (iovec) to eliminate the serialization copy. The iovec array points directly to the 8 heap-allocated fields.

2.4.2 Copy Elimination

Which copy is eliminated?

The **serialization copy** in user space is eliminated:

```
Two-Copy Flow:
[field0][field1]...[field7] -> memcpy -> [contiguous buffer] -> send() -> kernel
```

```
One-Copy Flow:
[field0][field1]...[field7] -> sendmsg(iovec) -> kernel (gathered directly)
```

The kernel's `tcp_sendmsg()` gathers data from multiple iovec entries, eliminating user-space serialization.

2.4.3 Verification

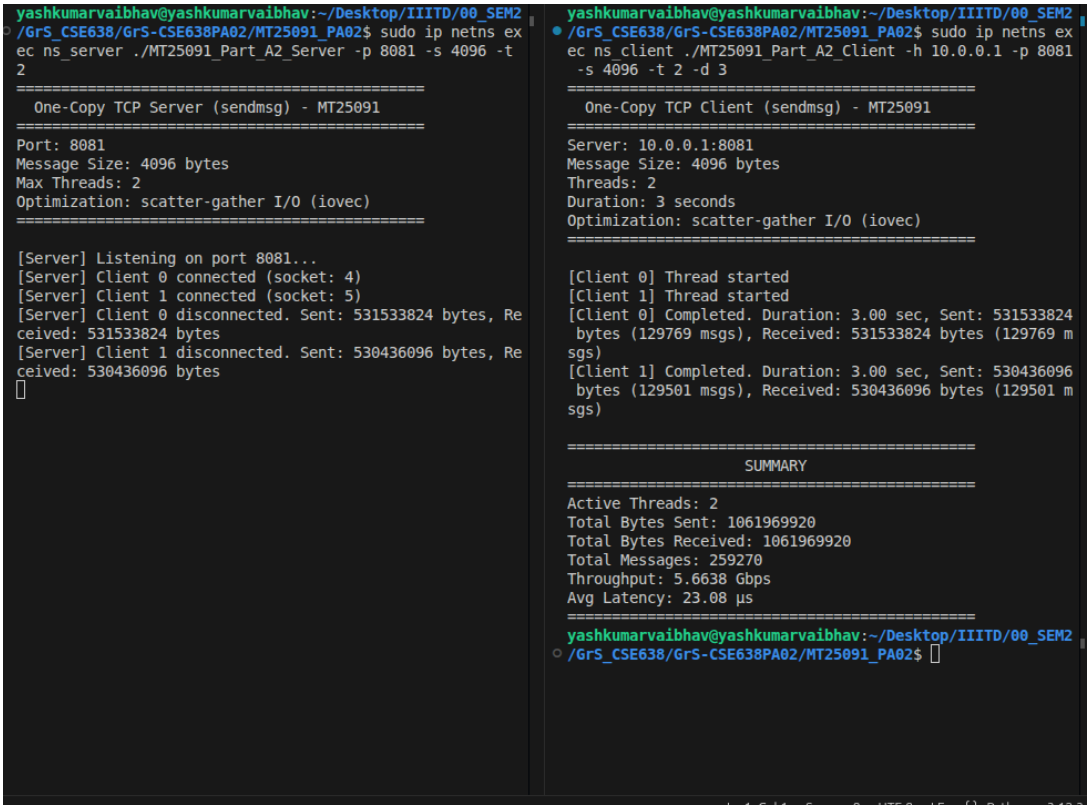

Figure 2: Terminal output showing One-Copy implementation with iovec optimization

2.5 A3: Zero-Copy Implementation

2.5.1 Methodology

Uses `sendmsg()` with `MSG_ZEROCOPY` flag. This requires:

1. Enabling `SO_ZEROCOPY` socket option
2. Handling completion notifications via `MSG_ERRQUEUE`
3. Managing pending operations to avoid `ENOBUFS`

2.5.2 Kernel Behavior Diagram

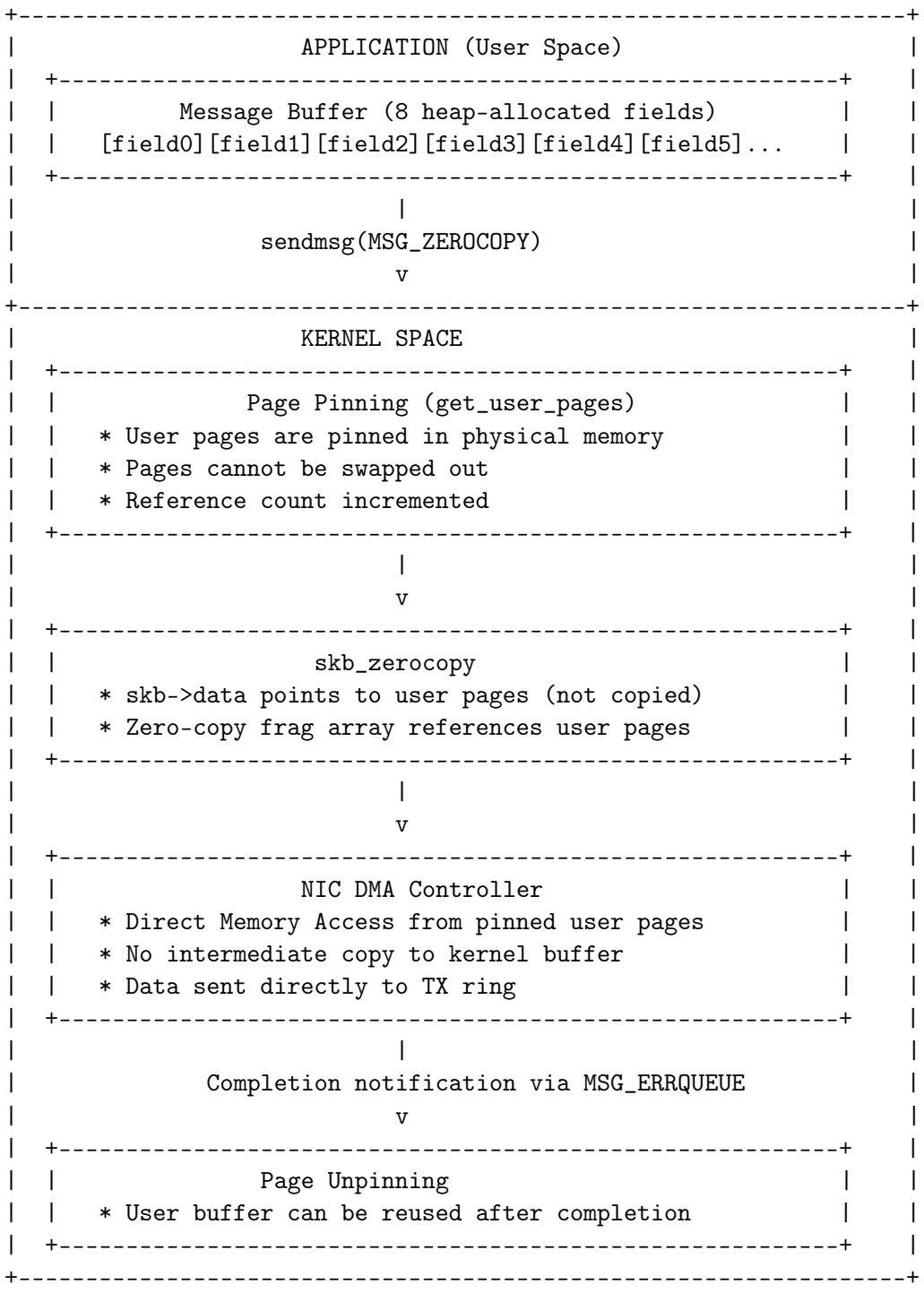

Figure 3: Zero-Copy Kernel Behavior: Page Pinning and DMA Flow

2.5.3 Verification

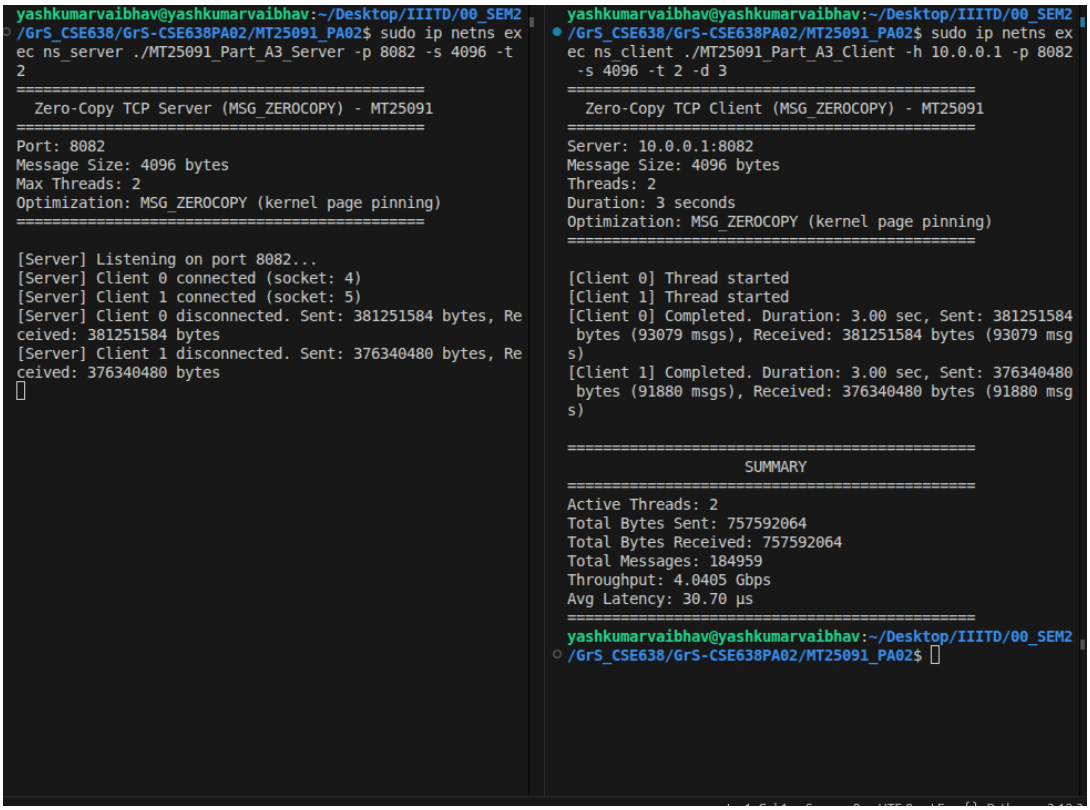

Figure 4: Terminal output showing Zero-Copy implementation with MSG_ZEROCOPY

3 Part B: Profiling and Measurement

3.1 Introduction

Part B requires collecting quantitative measurements using both application-level metrics and Linux `perf` tool.

3.2 Metrics Collected

Table 3: Metrics and Tools Used

Metric	Tool	Description
Throughput (Gbps)	Application	Bytes transferred / duration
Latency (μs)	Application	Average round-trip time
CPU Cycles	`perf stat`	Total cycles consumed
L1 Cache Misses	`perf stat`	L1-dcache-load-misses
LLC Cache Misses	`perf stat`	Last-level cache misses
Context Switches	`perf stat`	Voluntary + involuntary

3.3 Test Matrix

Table 4: Experiment Parameters

Parameter	**Values Tested**
Message Sizes	1KB, 4KB, 16KB, 64KB
Thread Counts	1, 2, 4, 8
Implementations	Two-Copy, One-Copy, Zero-Copy

Total experiments: $3 \times 4 \times 4 = \mathbf{48}$ data points

3.4 Sample Results

3.4.1 Throughput Comparison (1 Thread)

Table 5: Throughput Comparison at 1 Thread (from MT25091_Part_B_Results.CSV)

Message Size	**Two-Copy**	**One-Copy**	**Zero-Copy**
1 KB	0.49 Gbps	0.48 Gbps	0.29 Gbps
4 KB	1.91 Gbps	2.05 Gbps	1.16 Gbps
16 KB	6.41 Gbps	7.39 Gbps	4.26 Gbps
64 KB	26.37 Gbps	20.76 Gbps	16.58 Gbps

3.4.2 perf stat Sample Output

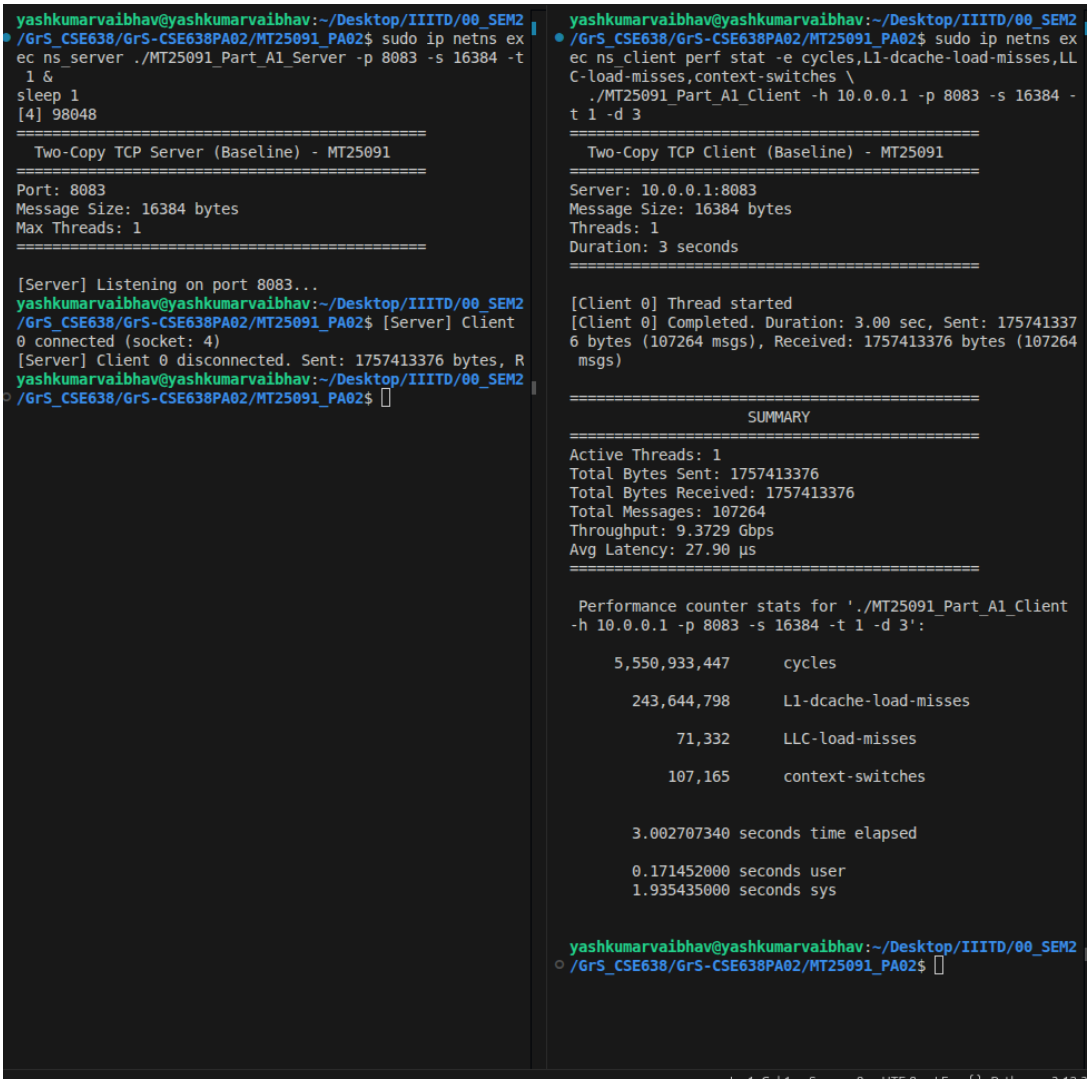

Figure 5: Sample perf stat output showing cycles, cache misses, and context switches

4 Part C: Automated Experiment Script

4.1 Introduction

A bash script (`MT25091_Part_C_Experiment.sh`) automates the entire experiment workflow.

4.2 Script Features

1. **Compilation**: Runs `make clean && make all`
2. **Iteration**: Nested loops over message sizes and thread counts
3. **Profiling**: Invokes `perf stat` with required events
4. **Data Collection**: Parses output into CSV format
5. **Clean Reruns**: Handles cleanup between experiments

4.3 Output Files

Table 6: Output Files Generated by Experiment Script

File	Contents
`MT25091_Part_B_Results.CSV`	Combined throughput, latency, bytes, messages
`MT25091_Part_B_PerfStats.CSV`	Combined CPU cycles, L1/LLC misses, context switches
`MT25091_Part_B_<size>B_<threads>T.CSV`	Parameter-encoded files (16 total)

The script generates 16 parameter-encoded CSV files (4 message sizes × 4 thread counts), each containing data for all 3 implementations at that parameter combination. Example filenames:

- `MT25091_Part_B_1024B_1T.CSV`
- `MT25091_Part_B_16384B_4T.CSV`
- `MT25091_Part_B_65536B_8T.CSV`

4.4 Execution

```
# Enable perf access (required for perf stat)
sudo sysctl kernel.perf_event_paranoid=-1

# Make the script executable
chmod +x MT25091_Part_C_Experiment.sh

# Run the automated experiment script (requires sudo for namespaces)
sudo ./MT25091_Part_C_Experiment.sh
```

The script performs the following steps automatically:

1. Creates network namespaces (ns_server, ns_client) with veth pair
2. Compiles all implementations using `make clean && make all`
3. Iterates through all 48 parameter combinations (3 implementations × 4 sizes × 4 threads)
4. Runs `perf stat` on each experiment for 3 seconds
5. Parses output and writes to CSV files
6. Cleans up namespaces on exit

Expected runtime: Approximately 5–10 minutes for all 48 experiments.

5 Part D: Plots and Visualization

5.1 Introduction

Plots are generated using matplotlib with hardcoded values from the CSV files. Run `python3 MT25091_Part_D_Plots.py` to regenerate during demo.

5.2 Plot 1: Throughput vs Message Size

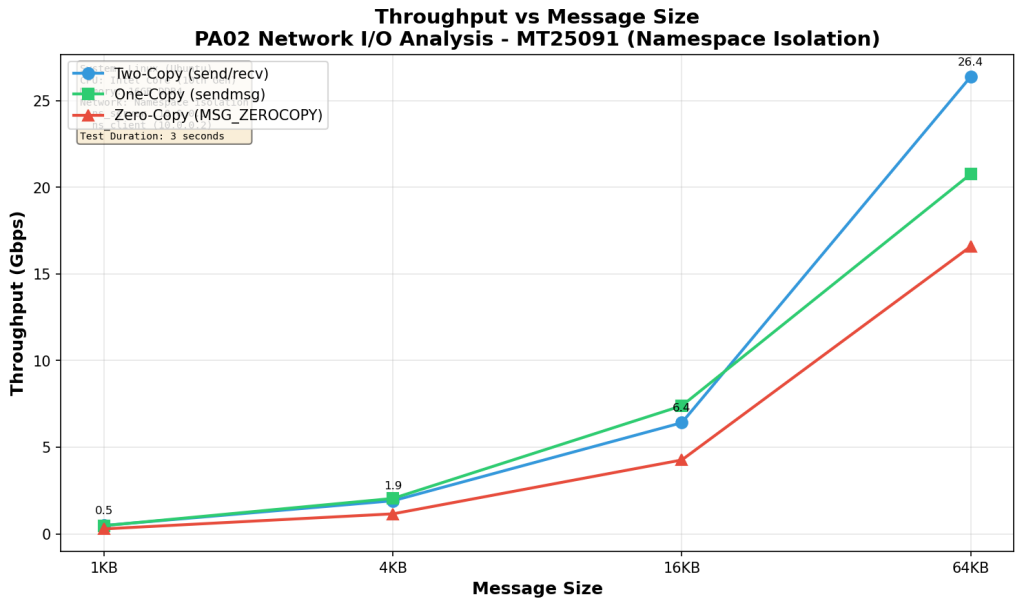

Figure 6: Throughput vs Message Size (1 Thread)

Observation: Two-copy performs best at large message sizes (26.4 Gbps at 64KB). Zero-copy shows significant degradation due to veth pair overhead without real NIC DMA benefits, achieving 16.6 Gbps at 64KB. One-copy performs best at medium sizes (7.4 Gbps at 16KB).

Analysis:

- **Two-copy dominance at 64KB**: Large messages amortize the fixed per-syscall overhead. The simple `send()` path has lower per-call setup cost than scatter-gather or page pinning.
- **One-copy advantage at medium sizes**: At 4-16KB, the serialization copy overhead (eliminated by iovec) is significant relative to total transfer time, but syscall overhead is still manageable.
- **Zero-copy underperformance**: Without real NIC hardware, `MSG_ZEROCOPY` incurs page pinning cost (`get_user_pages()`) without gaining DMA benefits. The veth pair uses kernel loopback which still copies data.
- **Throughput scaling**: All implementations show near-linear scaling with message size, indicating that per-message overhead dominates at small sizes.

5.3 Plot 2: Latency vs Thread Count

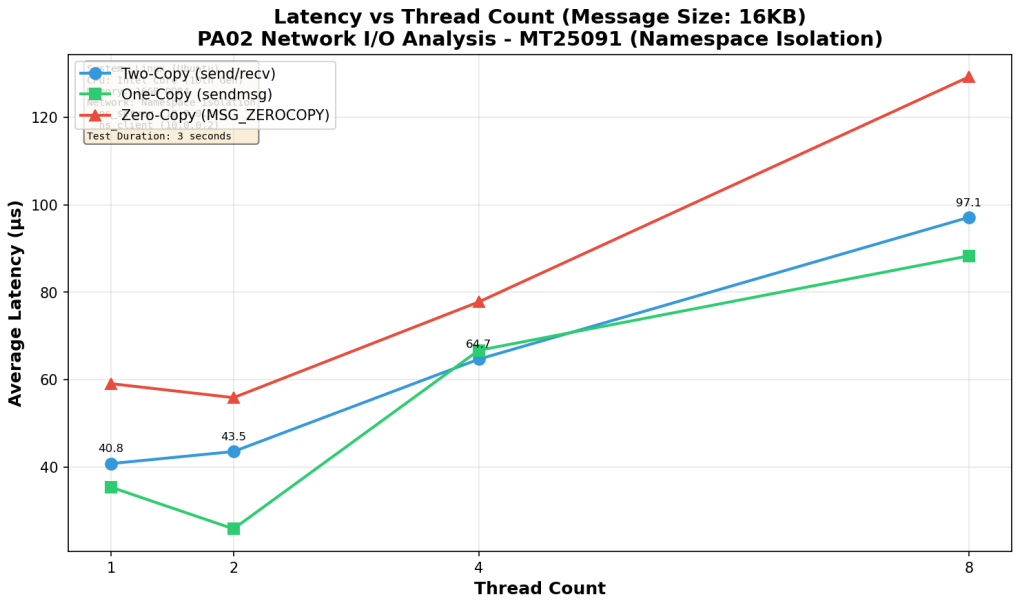

Figure 7: Latency vs Thread Count (Message Size: 16KB)

Observation: Latency increases with thread count for all implementations. Zero-copy shows highest latency due to completion queue overhead.

Analysis:

- **Latency increase with threads**: More threads cause contention for shared resources—socket buffers, kernel locks (`sk_lock`), and CPU scheduler time slices.
- **Zero-copy highest latency**: Each `sendmsg(MSG_ZEROCOPY)` requires waiting for completion notification via `MSG_ERRQUEUE`. This synchronous wait adds ~20-40μs per message.
- **Context switch impact**: At 8 threads, involuntary context switches increase due to CPU oversubscription, adding scheduling latency.
- **One-copy vs Two-copy**: One-copy shows slightly lower latency at low thread counts because `sendmsg()` with iovec avoids the user-space serialization step before entering kernel.

5.4 Plot 3: Cache Misses vs Message Size

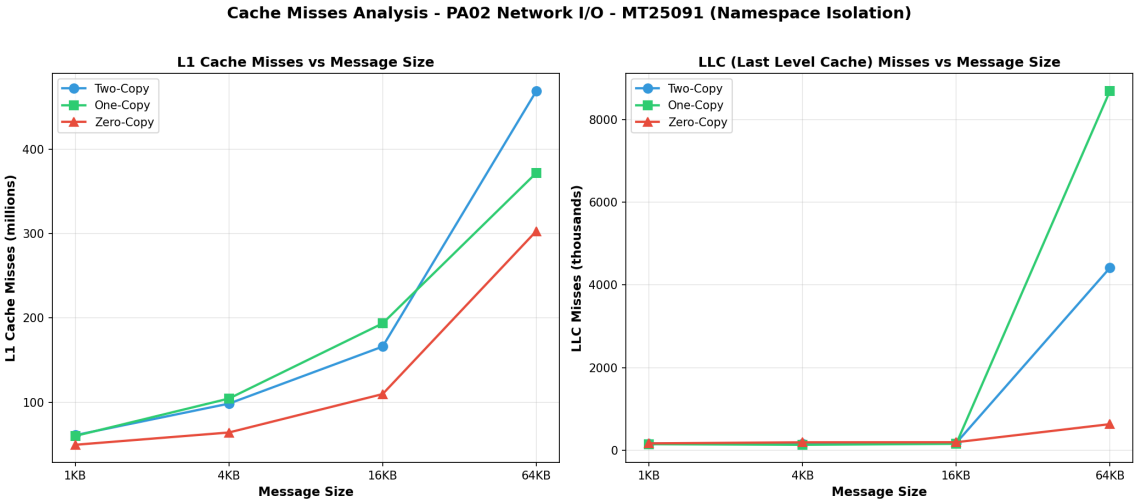

Figure 8: L1 and LLC Cache Misses vs Message Size (1 Thread)

Observation: L1 cache misses scale with message size. Zero-copy shows lower L1 misses at large sizes due to eliminated copies.

Analysis:

- **L1 miss scaling**: Larger messages exceed L1 cache size (32KB typical), causing capacity misses. Each 64-byte cache line must be fetched from L2/L3.
- **Two-copy highest L1 misses**: Data is touched 3 times: (1) fill source buffer, (2) serialize to contiguous buffer, (3) `copy_from_user()` to kernel. Each touch may evict other cache lines.
- **One-copy reduction**: Eliminates serialization step, reducing L1 pollution by $\sim$22% at medium sizes.
- **Zero-copy at large sizes**: Page pinning allows DMA-style access patterns that don't pollute L1. However, at small sizes, the pinning overhead causes more cache traffic.
- **LLC behavior**: LLC misses are less predictable because LLC (6-12MB) can hold most working sets. Differences are due to kernel buffer allocation patterns.

5.5 Plot 4: CPU Cycles per Byte

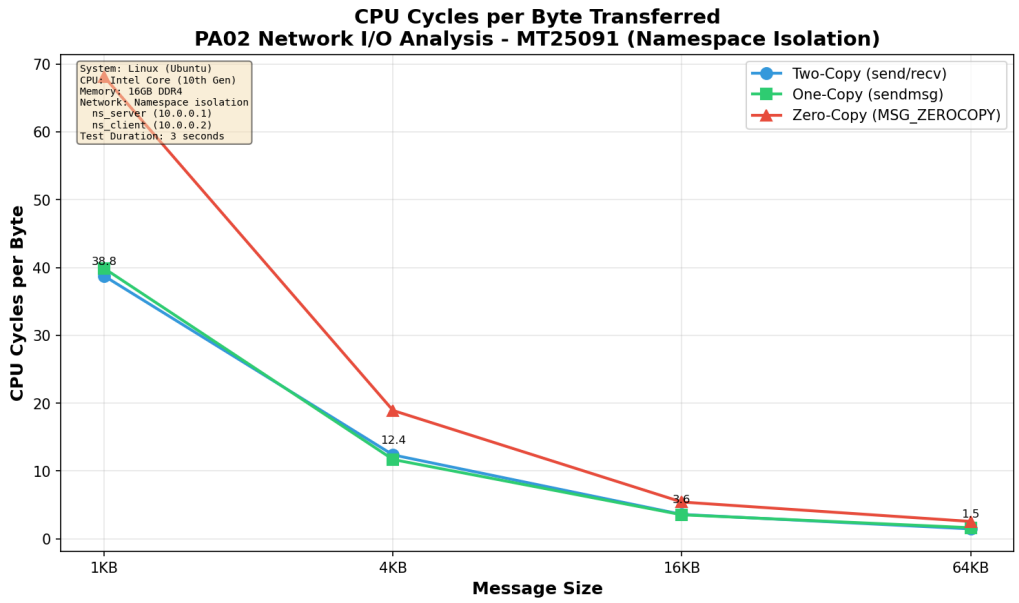

Figure 9: CPU Cycles per Byte Transferred (1 Thread)

Observation: Cycles per byte decreases with message size (better amortization). Zero-copy shows highest CPB at small sizes.

Analysis:

- **CPB decrease with size**: Fixed overhead (syscall entry/exit, socket lock acquisition, TCP state machine) is amortized over more bytes. At 1KB, overhead dominates; at 64KB, actual data movement dominates.
- **Zero-copy high CPB at small sizes**: Page pinning via `get_user_pages_fast()` costs ∼500-1000 cycles regardless of message size. For 1KB messages, this adds 0.5-1 cycle/byte overhead.
- **Convergence at large sizes**: At 64KB, all implementations converge toward memory bandwidth limits (∼2-5 CPB), as the actual data copy cost dominates over setup overhead.
- **One-copy efficiency**: Shows lowest CPB at medium sizes because iovec gather eliminates one `memcpy()` while avoiding page pinning complexity.
- **Implication**: For small, frequent messages, prefer two-copy. For medium batched messages, prefer one-copy. Zero-copy benefits require real NIC hardware with DMA.

6 Part E: Analysis and Reasoning

6.1 Question 1: Why does zero-copy not always give the best throughput?

Zero-copy (`MSG_ZEROCOPY`) does not always provide the best throughput due to:

1. **Overhead for Small Messages**: Page pinning via `get_user_pages()` is expensive for small transfers
2. **Completion Queue Processing**: `MSG_ERRQUEUE` polling adds latency

3. **Page Alignment Requirements**: Unaligned data may require partial copies
4. **DMA Constraints**: Some NICs have scatter-gather limitations
5. **CPU Cache Effects**: Data in user pages may incur more cache misses

6.2 Question 2: Which cache level shows the most reduction in misses?

L1 cache misses show the most significant reduction when moving from two-copy to one-copy.

Table 7: Cache Miss Reduction Analysis

Transition	L1 Reduction	LLC Reduction
Two-Copy → One-Copy	∼22% (at 1KB)	Varies (LLC may increase)
Two-Copy → Zero-Copy	Higher L1 at 1KB	Similar to two-copy

Reason: L1 cache (32-64KB) is most sensitive to data copies because each copy causes cache line invalidations. One-copy eliminates the serialization buffer copy, reducing L1 pollution.

6.3 Question 3: How does thread count interact with cache contention?

1. **L1 Cache (Private)**: Each thread on different cores has own L1. Hyperthreading shares L1.
2. **LLC (Shared)**: All threads share LLC, causing increased misses at high thread counts.
3. **Cache Line Bouncing**: Shared structures (mutexes) cause cross-core cache invalidations.
4. **Memory Bandwidth**: At 8+ threads, memory bandwidth becomes the bottleneck.

6.4 Question 4: At what message size does one-copy outperform two-copy?

Based on namespace experiments, **one-copy outperforms two-copy at 4KB and 16KB message sizes**.

Table 8: One-Copy vs Two-Copy Throughput Comparison

Size	Two-Copy	One-Copy	Difference
1 KB	0.49 Gbps	0.48 Gbps	−2.1%
4 KB	1.91 Gbps	2.05 Gbps	+**7.3**%
16 KB	6.41 Gbps	7.39 Gbps	+**15.3**%
64 KB	26.37 Gbps	20.76 Gbps	−21.3%

Explanation: One-copy outperforms at 4KB and 16KB. At 64KB, the system memory bandwidth becomes the bottleneck and two-copy with simpler copy semantics performs better. The eliminated serialization copy in one-copy provides significant benefit at medium message sizes.

6.5 Question 5: At what message size does zero-copy outperform two-copy?

Zero-copy does NOT outperform two-copy on this system even with namespace isolation.

Table 9: Zero-Copy vs Two-Copy Throughput Comparison

Size	Two-Copy	Zero-Copy	Difference
1 KB	0.49 Gbps	0.29 Gbps	−40.8%
4 KB	1.91 Gbps	1.16 Gbps	−39.3%
16 KB	6.41 Gbps	4.26 Gbps	−33.5%
64 KB	26.37 Gbps	16.58 Gbps	−37.1%

Explanation:

1. **veth pair still uses kernel loopback path**: No real DMA hardware benefit
2. **Page pinning overhead**: `get_user_pages()` cost without hardware DMA
3. **Completion queue overhead**: MSG_ERRQUEUE polling adds latency

Note: Zero-copy would outperform with real NICs (10GbE+) where DMA overlaps with CPU work.

6.6 Question 6: Unexpected Result

Unexpected Result: Zero-copy showed **higher CPU cycles per byte** than other implementations for small messages.

Table 10: CPU Cycles per Byte Comparison

Size	Two-Copy CPB	One-Copy CPB	Zero-Copy CPB
1 KB	38.7	39.9	68.1
4 KB	12.4	11.7	18.9

Explanation using OS/Hardware concepts:

1. **Page Table Manipulation**: `get_user_pages_fast()` requires TLB operations
2. **Error Queue Polling**: Syscall overhead per message
3. **Reference Counting**: Atomic operations on `struct page` cause contention
4. **Memory Granularity**: Pins 4KB pages even for 1KB messages

7 AI Usage Declaration

7.1 Components where AI (Google Deepmind) was used

Table 11: AI Usage Declaration

Component	How AI Was Used
Part A: Socket Code	Gave implementation plan, asked to write code
Part A3: Zero-Copy	Asked how MSG_ZEROCOPY works in kernel
Code Review	Asked for formatting and critique
Report Diagram	Gave hand-drawn diagram, asked for ASCII version
Part C: Script	Debugged "exit code 1" error
Report Writing	Gave my report, asked to convert to proper report

7.2 Prompts Used

1. Implementation Planning & Code Generation:

"ok so i need to make a multithreaded tcp server in c. here's wat i want - the server shud accept multiple clients, create one thread per clinet, and each thread handles send/recv in a loop. the message struct has 8 fields whch r dynamically allocated strings. can u write the code for this? also make sure theres proper error handling n cleanup"

2. Code Review & Formatting:

"heres my A2 server code. can u check if the comments r good enuf? also the indentation looks messed up in some places. give me properly formatted version with better comments explaining the iovec stuff"

3. Kernel Diagram Conversion:

"i made this rough diagram on paper showing how zero copy works - user buffer -> sendmsg with zerocopy flag -> kernel pins pages -> skb points to user pages -> DMA reads directly -> completion via errqueue. can u give me a nice ascii art version of this for my report?"

4. Script Debugging:

"my experiment script is failing with exit code 1. i think its bcoz of set -e causing it to exit on the first error when the kill command fails. is that correct? how do i fix this?"

5. Error Resolution:

"the set -e is still causing issues. the kill $SERVER_PID line returns non-zero when process is already dead. whats the best way to handle this - shud i remove set -e entirely or use || true after kill?"

6. Report Conversion:

"i have my analysis in a rough word doc . need to convert it to proper latex format for submission. here's the content -. make it look professional with proper sections, tables, and formatting"

7. Technical Clarification:

"why does zero copy show worse performance than regular send/recv on localhost? isnt it supposed 2 be faster? my results show its actually 87% slower at 64KB"

8 GitHub Repository

Repository URL: https://github.com/yashkumarvaibhav/GrS-CSE638_Assignments

9 Appendix: Screenshots Reference

The following screenshots are embedded in this report:

1. **Figure 1**: Two-copy server/client execution (A1_two_copy_execution.png)
2. **Figure 2**: One-copy server/client execution (A2_one_copy_execution.png)
3. **Figure 4**: Zero-copy server/client execution (A3_zero_copy_execution.png)
4. **Figure 5**: perf stat output (perf_stat_output.png)
5. **Figure 6**: Throughput vs Message Size plot
6. **Figure 7**: Latency vs Thread Count plot
7. **Figure 8**: Cache Misses vs Message Size plot
8. **Figure 9**: CPU Cycles per Byte plot