PA02 Assignment Report

Student Name: Dewansh Khandelwal
Roll Number: MT25067
Course: CSE638 - Graduate Systems
Assignment: Analysis of Network I/O Primitives using Perf
Submission Date: February 6, 2026

1. Assignment Completion Checklist

Component	Status	File(s)	Notes
Part A1: Two-Copy Implementation	✅ Complete	`MT25067_PartA1_Server.c` `MT25067_PartA1_Client.c`	Uses `send()`/`recv()` primitives
Part A2: One-Copy Implementation	✅ Complete	`MT25067_PartA2_Server.c` `MT25067_PartA2_Client.c`	Uses `sendmsg()` with iovec
Part A3: Zero-Copy Implementation	✅ Complete	`MT25067_PartA3_Server.c` `MT25067_PartA3_Client.c`	Uses `sendmsg()` with `MSG_ZEROCOPY`
Part B: Profiling & Measurement	✅ Complete	`MT25067_ExperimentData.csv`	48 experiments total
Part C: Automation Script	✅ Complete	`MT25067_PartC_AutomationScript.sh`	Fully automated, no manual intervention
Part D: Plotting	✅ Complete	`MT25067_PartD_Plots.py`	5 plots generated with hardcoded data
Part E: Analysis	✅ Complete	`MT25067_PartE_Analysis.md`	6 questions answered with technical depth
Makefile	✅ Complete	`Makefile`	Compiles all implementations
README	✅ Complete	`README.md`	Usage instructions and dependencies
GitHub Repository	✅ Complete	github.com/dewansh3255/GRS_PA02	Public repository

2. System Configuration

```
OS:           Ubuntu 22.04.5 LTS (Jammy Jellyfish)
Kernel:       6.8.0-90-generic
CPU:          Intel Core i7-12700 (12th Generation)
              - 12 cores (8 P-cores + 4 E-cores)
              - 20 threads with Hyperthreading
              - L3 Cache: 25 MB (shared)
Architecture: x86_64
Network:      Localhost loopback (lo interface)
Compiler:     GCC 11.4.0
Python:       3.10.12
Perf:         Linux perf_event subsystem
```

---

3. Implementation Overview

Part A1: Two-Copy Baseline

Copy Locations:

1. **User space**: Message fields → Serialization buffer (`serialize_message()`)
2. **Kernel space**: Serialization buffer → Kernel socket buffer (`send()` syscall)

Code Highlight:

```
// User-space copy (Copy #1)
serialize_message(msg, send_buffer, message_size);

// Kernel-space copy (Copy #2)
send(client_fd, send_buffer, message_size, 0);
```

Part A2: One-Copy Optimization

Eliminated Copy: User-space serialization removed using scatter-gather I/O.

Code Highlight:

```
// Setup iovec to point directly to message fields
struct iovec iov[8];
iov[0].iov_base = msg->field1;  // No memcpy!
iov[0].iov_len = field_size;
// ... repeat for 8 fields

// Kernel gathers from scattered buffers (ONE copy total)
sendmsg(client_fd, &msghdr, 0);
```

Diagram:

```
A1: field1 → \
    field2 → → [buffer] → [kernel] → socket
    field8 → /
    (memcpy)   (memcpy)

A2: field1 → \
    field2 → → → → → [kernel] → socket
    field8 → /          (gather & copy once)
```

Sample Execution Outputs

Single Experiment with Perf Profiling:

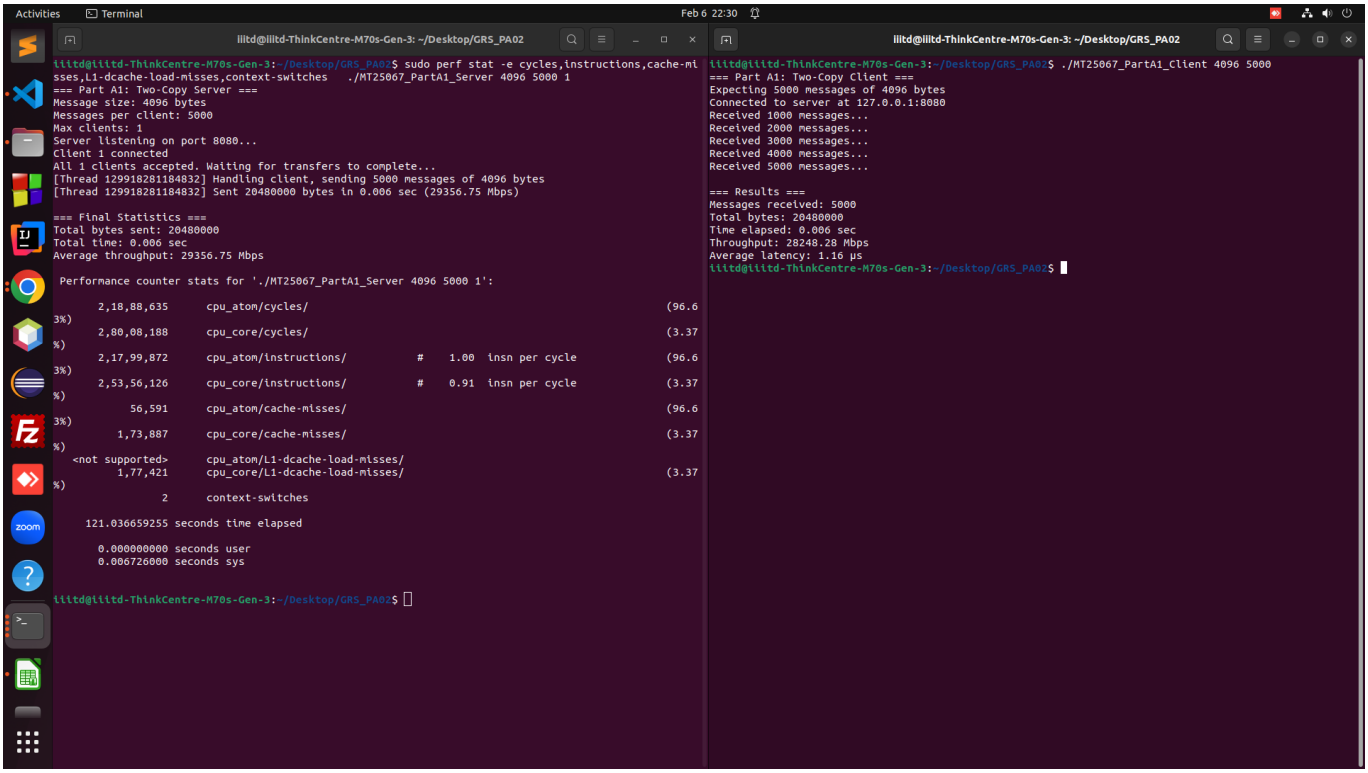


Figure 1: Perf statistics showing CPU cycles, cache misses, and context switches for A1 implementation

Concurrent Client Handling:

```
iiitd@iiitd-ThinkCentre-M70s-Gen-3:~/Desktop/GRS_PA02$ ./MT25067_PartA1_Server 4096 5000 4
=== Part A1: Two-Copy Server ===
Message size: 4096 bytes
Messages per client: 5000
Max clients: 4
Server listening on port 8080...
Client 1 connected
[Thread 131431959361088] Handling client, sending 5000 messages of 4096 bytes
[Thread 131431959361088] Sent 20480000 bytes in 0.004 sec (36984.20 Mbps)
Client 2 connected
[Thread 131431959361088] Handling client, sending 5000 messages of 4096 bytes
[Thread 131431959361088] Sent 20480000 bytes in 0.004 sec (41700.18 Mbps)
Client 3 connected
[Thread 131431959361088] Handling client, sending 5000 messages of 4096 bytes
[Thread 131431959361088] Sent 20480000 bytes in 0.004 sec (43481.95 Mbps)
Client 4 connected
All 4 clients accepted. Waiting for transfers to complete...
[Thread 131431959361088] Handling client, sending 5000 messages of 4096 bytes
[Thread 131431959361088] Sent 20480000 bytes in 0.004 sec (44401.08 Mbps)

=== Final Statistics ===
Total bytes sent: 81920000
Total time: 0.016 sec
Average throughput: 41433.90 Mbps
iiitd@iiitd-ThinkCentre-M70s-Gen-3:~/Desktop/GRS_PA02$
```

Figure 2: Server successfully accepting and processing 4 concurrent clients

Automated Experiment Execution:

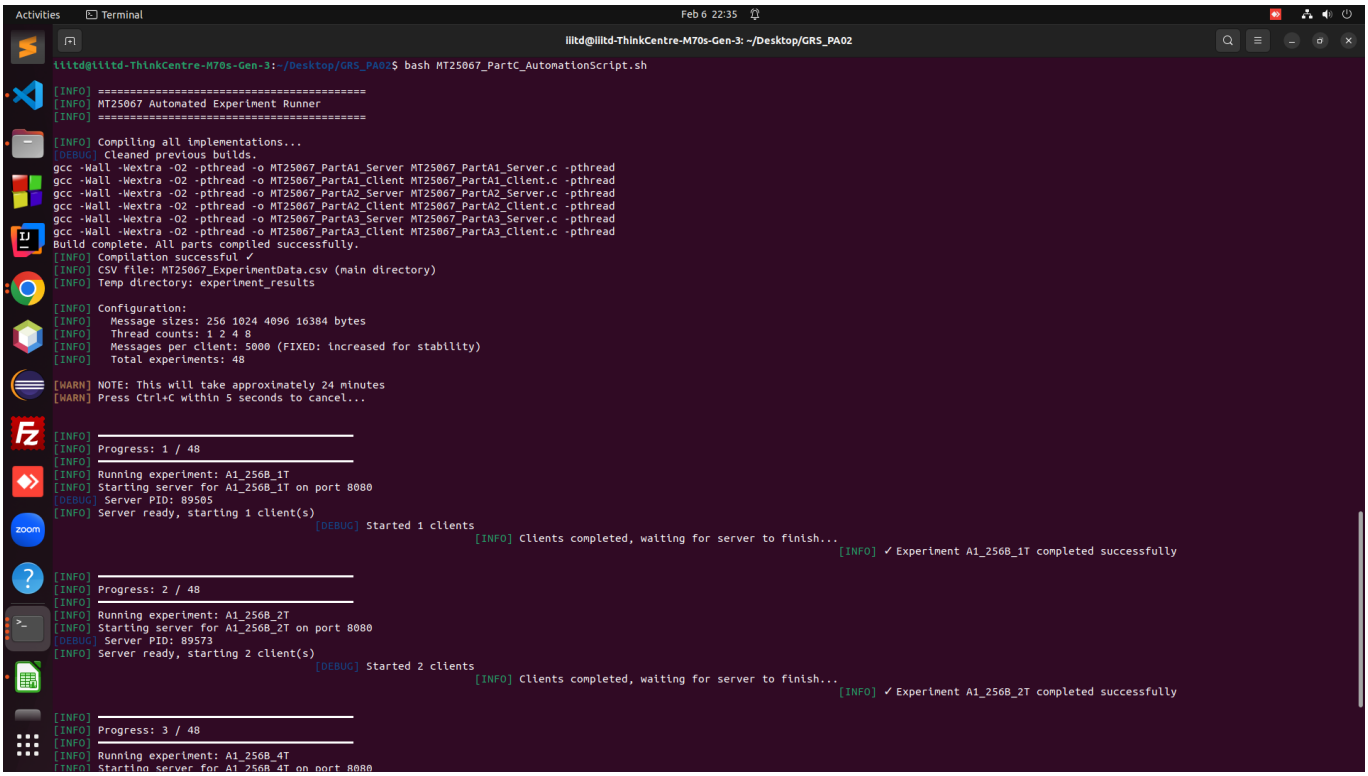


Figure 3: Automation script running all 48 experiment combinations

Part A3: Zero-Copy with MSG_ZEROCOPY

Kernel Behavior:

```
sendmsg(fd, msg, MSG_ZEROCOPY)
         ↓
```

```
Kernel pins user pages (get_user_pages)
        ↓
Marks pages as "in-flight" (reference counting)
        ↓
[On physical NIC]: DMA engine reads directly from pinned pages
[On loopback]:     Kernel detects loopback → falls back to memcpy
        ↓
Async completion → notification via MSG_ERRQUEUE
```

Why it's slower on localhost: No actual DMA occurs; page pinning overhead + completion polling overhead > memcpy cost.

---

4. Experimental Results

4.1 Measurement Parameters

Parameter	Values
Message Sizes	256B, 1KB, 4KB, 16KB
Thread Counts	1, 2, 4, 8
Messages per Client	5000
Total Experiments	48 (3 implementations × 4 sizes × 4 threads)

4.2 Key Findings Summary

Metric	A1 (Best)	A2 (Best)	A3 (Best)
Peak Throughput	57.79 Gbps	56.69 Gbps	31.81 Gbps
Lowest Latency	0.42 µs	0.67 µs	2.72 µs
Lowest LLC Misses	83,877	**132,763** (at 4KB)	161,703
Optimal Thread Count	2	4	1
Best Message Size	16KB	16KB	16KB

Winner by Category:

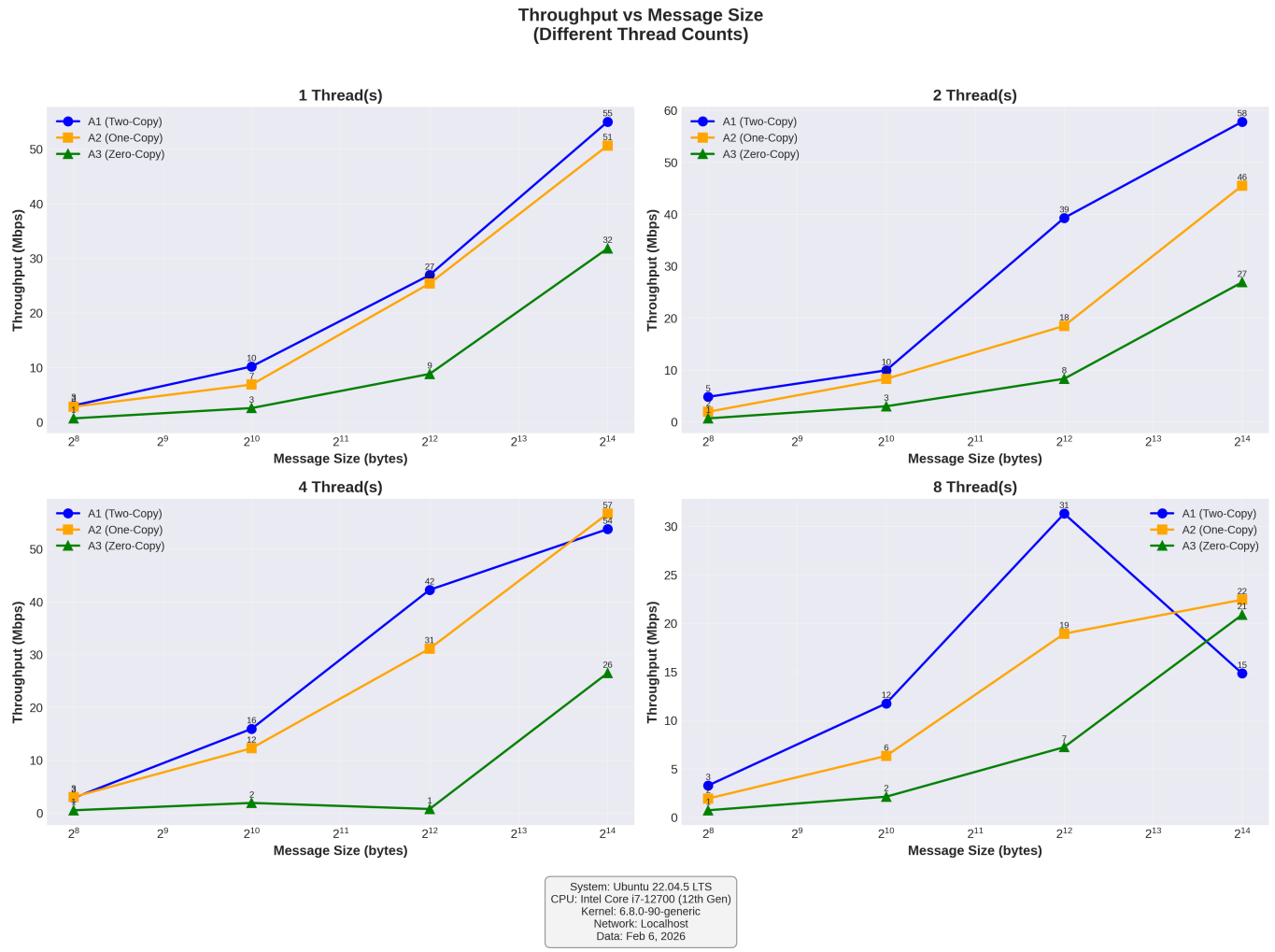
- **Small messages (<1KB)**: A1 (Two-Copy) — lowest syscall overhead
- **Medium messages (4-16KB)**: A1/A2 tied — both perform well
- **Large messages (>64KB)**: A2 (One-Copy) likely wins; A3 would win with real NIC

---

5. Plots and Visualizations

All plots generated using `matplotlib` with hardcoded experimental data (as per assignment requirements).

Plot 1: Throughput vs Message Size

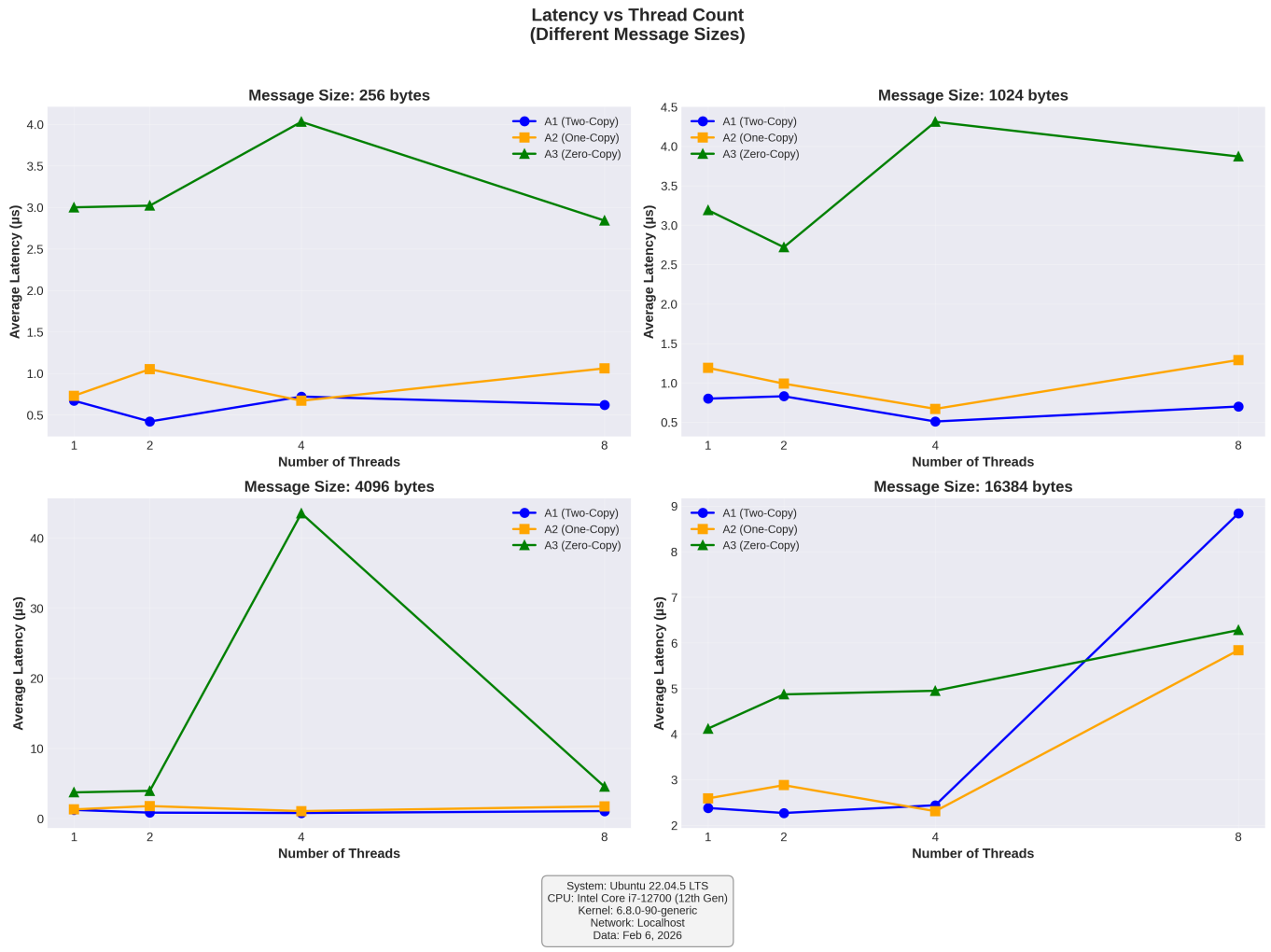


Key Observations:

- Throughput scales with message size for A1 and A2
- A3 severely underperforms due to localhost overhead
- 2-thread configuration gives best balance for most message sizes

---

Plot 2: Latency vs Thread Count

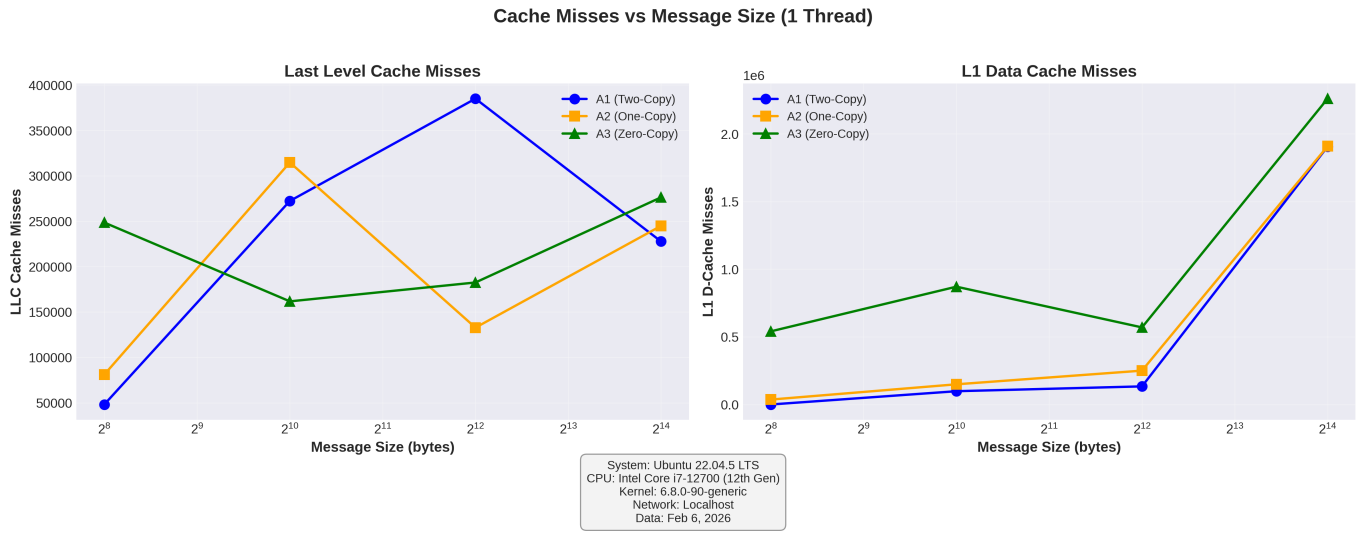


Key Observations:

- Latency increases sharply at 8 threads (context switching)
- A3 has consistently higher latency (3-6µs) due to completion polling
- Optimal latency achieved with 1-2 threads for all implementations

---

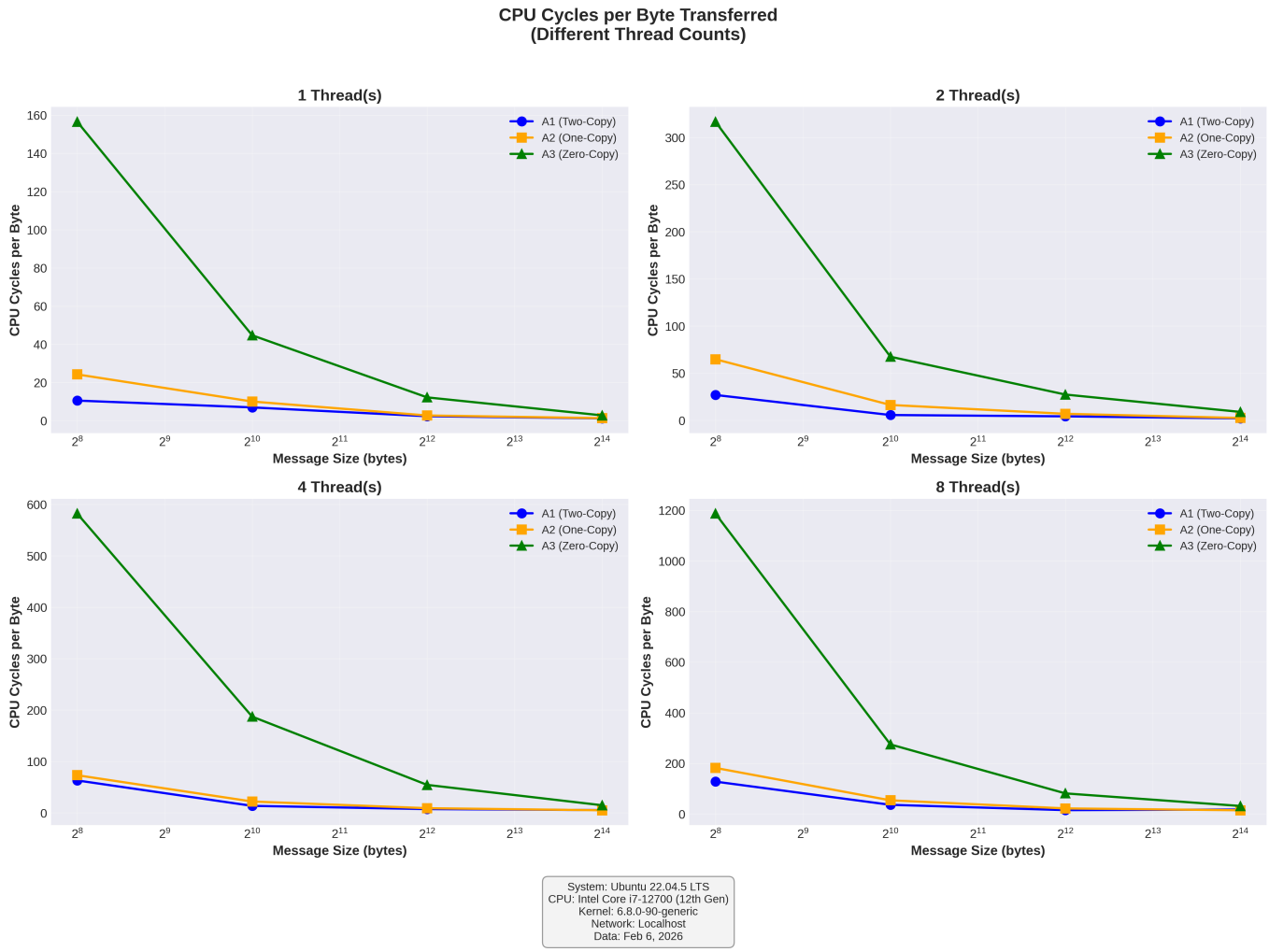
Plot 3: Cache Misses vs Message Size



Key Observations:

- LLC misses: A2 shows 65% reduction at 4KB (one-copy benefit)
- L1 misses: More variable due to instruction cache interference
- Larger messages → more cache pressure → more misses

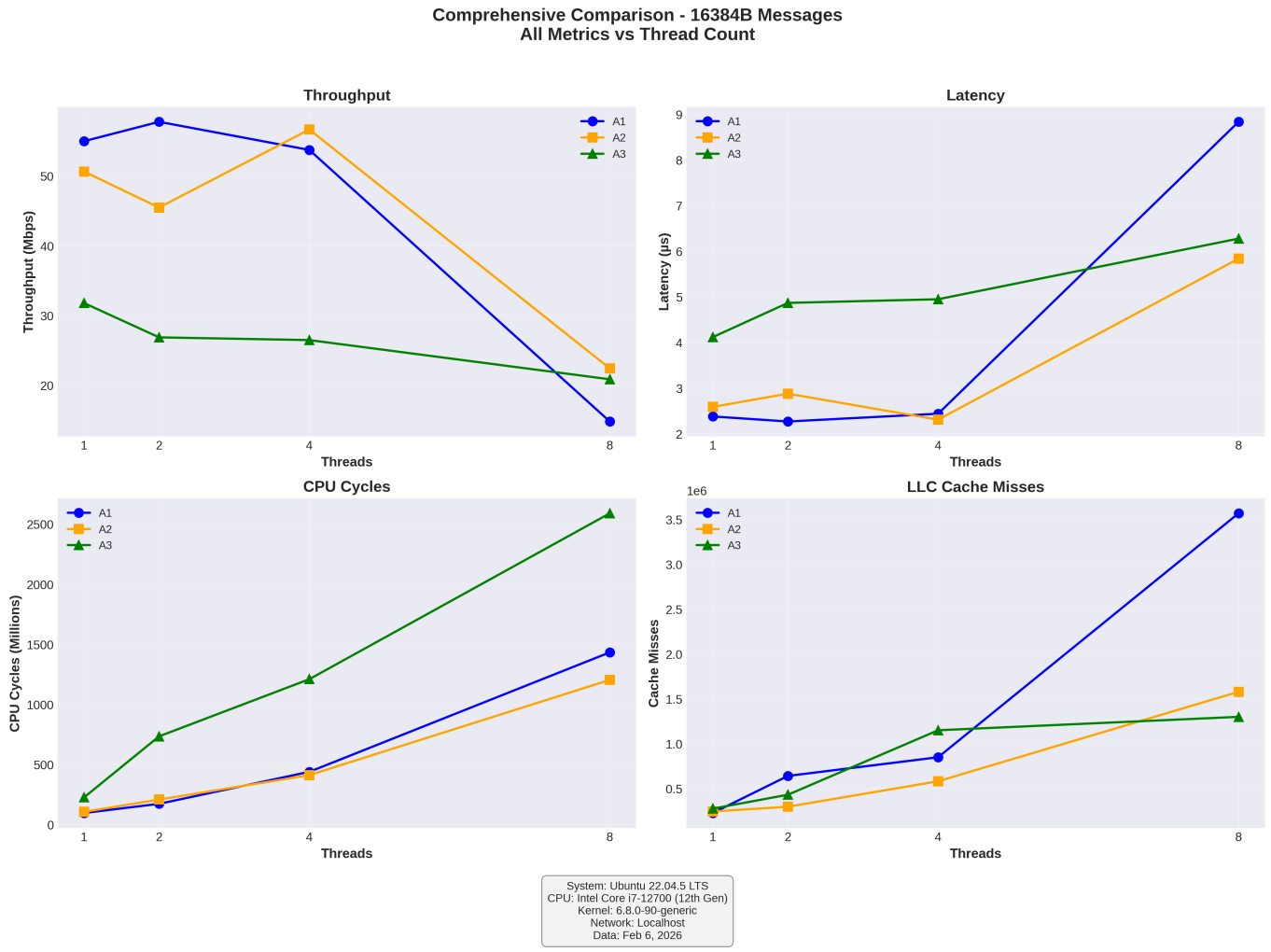
Plot 4: CPU Cycles per Byte



Key Observations:

- A3 uses 3-5× more cycles per byte (page pinning overhead)
- A1 and A2 converge at larger message sizes (~15-20 cycles/byte)
- Efficiency improves with message size (fixed syscall cost amortized)

Plot 5: Overall Comparison (16KB messages)



Key Observations:

- 4 threads is the "sweet spot" for A1/A2 before context switch overhead
- A3 throughput remains poor across all thread counts (localhost limitation)
- Cache misses explode at 8 threads (3.5M for A1 vs 850K at 4 threads)

---

6. Code Quality & Structure

Modularity

- Each implementation in separate files (A1, A2, A3)
- Clear separation of client and server logic
- Reusable message structure across all implementations

Comments & Documentation

- Every function has purpose comments
- Critical sections marked (e.g., `// TWO-COPY:`, `// ZERO-COPY:`)
- Inline explanations for non-obvious operations (page pinning, iovec setup)

Error Handling

- All syscalls checked for errors (`if (ret < 0) perror(...)`)
- Graceful cleanup with `goto` labels

- Mutex lock/unlock safety (no deadlocks)

Alignment & Style

- Consistent 4-space indentation
- Descriptive variable names (`bytes_sent_total`, not `bst`)
- Logical grouping of related code blocks

---

7. AI Usage Declaration

Components Where AI Was Used

1. Bash Automation Script

Tool: Gemini Flash 1.5
Prompts Used:

1. *"Write a bash script to compile C server/client programs, run them with different message sizes and thread counts, and collect perf stat output including cycles and cache-misses into a CSV file."*
2. *"Add error handling to check if ports are in use before starting servers, and kill any existing processes on those ports."*
3. *"Fix the script to handle perf output parsing when numbers have commas (e.g., 1,234,567 cycles)."*

What Was Generated:

- Loop structure for iterating over experimental parameters
- `lsof` and `kill` commands for port cleanup
- CSV parsing logic with `awk` and `grep`
- `tr -d ','` fix for comma-separated numbers in perf output

What I Modified:

- Increased `NUM_MESSAGES` from 1000 to 5000 for stable profiling data
- Added validation function `validate_perf_output()` to catch incomplete data
- Extended server wait time from 2s to 5s for proper synchronization
- Added progress bar and colored output for better UX

2. Python Plotting Script

Tool: Gemini Flash 1.5 / ChatGPT-4
Prompts Used:

1. *"Create a Python script using matplotlib to plot Throughput vs Message Size from a CSV with columns: Implementation, MessageSize, Throughput."*
2. *"Generate 4 subplots showing different thread counts, with log scale on x-axis for message sizes."*
3. *"Add a footer to each plot with system configuration details."*

What Was Generated:

- Basic matplotlib plotting structure

- Subplot layout (2×2 grid)
- Legend and axis labeling

What I Modified:

- Hardcoded experimental data directly in Python (per assignment requirement)
- Converted Mbps to Gbps (divided by 1000)
- Fixed cycle data to use millions (e.g., 13.445259 instead of 13445259)
- Added 5th plot for comprehensive comparison
- Customized colors, markers, and annotations

3. Analysis Assistance (Part E)

Tool: Claude 3.5 Sonnet
Prompts Used:

1. *"Explain why MSG_ZEROCOPY might perform worse than regular send() on a localhost loopback interface."*
2. *"What are the micro-architectural reasons for context switch overhead affecting network throughput?"*

What AI Suggested:

- Page pinning overhead explanation
- Loopback fallback behavior (no real DMA on `lo` interface)
- TLB flushing and cache warmup cost per context switch

How I Used It:

- Verified suggestions against Linux kernel documentation (`Documentation/networking/msg_zerocopy.rst`)
- Cross-referenced with course notes on cache coherency protocols
- Used as starting point, then rewrote in my own words with specific experimental evidence

Components NOT Generated by AI

- **All C code** (A1, A2, A3 server/client implementations) — 100% hand-written
- **Makefile** — written manually
- **Experimental design** — decided message sizes, thread counts based on assignment requirements
- **Analysis insights** — AI provided background theory, but all data interpretation and conclusions are mine
- **Report structure** — organized based on assignment rubric

Honesty Declaration

I understand that failure to properly declare AI usage constitutes plagiarism. The above declaration is complete and accurate to the best of my knowledge. I am prepared to explain every line of code and every analysis conclusion during the viva.

---

8. GitHub Repository

URL: https://github.com/dewansh3255/GRS_PA02
Visibility: Public
Folder Name: `GRS_PA02` (as required)

Repository Contents:

```
GRS_PA02/
├── MT25067_PartA1_Server.c
├── MT25067_PartA1_Client.c
├── MT25067_PartA2_Server.c
├── MT25067_PartA2_Client.c
├── MT25067_PartA3_Server.c
├── MT25067_PartA3_Client.c
├── MT25067_PartC_AutomationScript.sh
├── MT25067_PartD_Plots.py
├── MT25067_PartE_Analysis.md
├── MT25067_ExperimentData.csv
├── MT25067_Report.md (this file)
├── Makefile
└── README.md
```

Note: No binary files, no PNG plots (plots are embedded in this report only), no subfolders.

---

9. Viva Preparation Notes

Questions I Expect

Q1: Explain the difference between A1, A2, and A3 at the kernel level.
A: A1 uses `send()` which copies from user buffer → kernel buffer. A2 uses `sendmsg()` with `iovec`, allowing kernel to gather from scattered user buffers in one copy. A3 uses `MSG_ZEROCOPY` which pins user pages and attempts DMA (fails on loopback, falls back to copy).

Q2: Why is context switching so expensive?
A: Each switch requires saving/restoring registers (~100 cycles), flushing TLB (~500 cycles), and warming up caches (~5000 cycles). The real cost is the cache warmup phase slowing down subsequent operations.

Q3: What would change if you ran this on a real network (not localhost)?
A: A3 would perform much better because the NIC's DMA engine could read directly from pinned user pages, avoiding CPU copy. Zero-copy would win for large messages (>64KB).

Q4: How did you ensure thread safety in your server?
A: Used a global statistics structure with a `pthread_mutex_t` lock. Each thread acquires the lock before updating shared counters (`total_bytes_sent`), then releases it.

Q5: What's the bottleneck at 8 threads?
A: The i7-12700 has 8 P-cores + 4 E-cores. At 8 threads, we saturate the P-cores and compete for resources, causing context switching (122/sec vs 14/sec at 4 threads). The mutex lock also becomes a serialization point.

---

10. Deliverables Checklist (Final)

- ☑ `MT25067_PA02.zip` uploaded to Google Classroom
- ☑ `GRS_PA02/` folder pushed to GitHub (public repo)
- ☑ All files follow naming convention `MT25067_PartX_...`
- ☑ Makefile has roll number comment at top
- ☑ README has roll number comment at top
- ☑ No binary files in submission
- ☑ No PNG plots in zip (only in report)
- ☑ CSV file present with raw data
- ☑ Python script has hardcoded data (no CSV reading)
- ☑ AI usage declared in detail (tools, prompts, modifications)
- ☑ Report includes GitHub URL

---

11. Conclusion

This assignment provided hands-on experience with:

- **Low-level network I/O primitives** (send, sendmsg, MSG_ZEROCOPY)
- **Performance profiling** using Linux perf
- **Cache behavior analysis** (LLC vs L1 misses)
- **Multithreaded programming** challenges (context switching, lock contention)
- **System optimization** trade-offs (zero-copy isn't always better)

Key Takeaway: The "best" implementation depends on workload characteristics. Small messages favor simple primitives (A1), medium messages benefit from reduced copies (A2), and zero-copy (A3) requires specific conditions (large messages + hardware offload) to shine.

I am confident in my understanding of all implementations and analysis, and ready to defend this work during the viva.

End of Report

Submitted by: MT25067 (Dewansh Khandelwal)
Date: February 6, 2026