Analysis of Network I/O Primitives Using Perf

Student: Dewansh Khandelwal (MT25067)
Course: CSE638 - Graduate Systems
Date: February 6, 2026

Abstract

This report presents a comprehensive experimental analysis of three network I/O paradigms: two-copy (baseline), one-copy (scatter-gather), and zero-copy (MSG_ZEROCOPY) socket communication. Using the Linux `perf` tool, we profiled CPU cycles, cache behavior, and context switches across varying message sizes (256B-16KB) and thread counts (1-8). Our findings reveal that zero-copy underperforms on localhost due to page pinning overhead without DMA benefits, one-copy reduces LLC misses by 65% through elimination of serialization buffers, and optimal thread count is bounded by cache coherency and context switching costs. The crossover point for one-copy vs two-copy occurs at 8-16KB message sizes.

---

1. Introduction

1.1 Motivation

Network I/O operations traditionally involve multiple data copies between user space and kernel space, consuming CPU cycles and memory bandwidth. Modern Linux provides optimizations like scatter-gather I/O (`sendmsg` with `iovec`) and zero-copy (`MSG_ZEROCOPY`) to reduce this overhead. Understanding when each approach provides actual performance benefits requires empirical analysis under controlled conditions.

1.2 Experimental Setup

Hardware Configuration:

- **CPU**: Intel Core i7-12700 (12th Gen)
  - 8 Performance cores (P-cores) @ 3.6-4.9 GHz
  - 4 Efficiency cores (E-cores) @ 2.7-3.6 GHz
  - 25MB shared L3 cache
- **RAM**: DDR4 (sufficient for workload)
- **Network**: Localhost loopback interface (`lo`)

Software Environment:

- **OS**: Ubuntu 22.04.5 LTS
- **Kernel**: 6.8.0-90-generic
- **Compiler**: GCC 11.4.0 with `-O2 -pthread`
- **Profiler**: Linux perf (perf_event subsystem)

Experimental 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)

---

2. Implementation Details

2.1 Two-Copy Implementation (A1 - Baseline)

Architecture:

```
User Space:           Message Fields (8 × malloc'd buffers)
                              ↓
                      serialize_message()  ← Copy #1 (memcpy)
                              ↓
                      Contiguous Buffer (stack)
                              ↓
                      send() syscall
                              ↓
Kernel Space:         Kernel Socket Buffer  ← Copy #2 (copy_from_user)
                              ↓
                      TCP/IP Stack → Loopback
```

Key Code Snippet:

```
// Copy #1: User-space serialization
int serialize_message(Message *msg, char *buffer, int buffer_size) {
    int offset = 0;
    int field_size = buffer_size / 8;
    memcpy(buffer + offset, msg->field1, field_size); offset +=
field_size;
    memcpy(buffer + offset, msg->field2, field_size); offset +=
field_size;
    // ... repeat for all 8 fields
    return offset;
}

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

2.2 One-Copy Implementation (A2 - Scatter-Gather)

Architecture:

```
User Space:           Message Fields (8 × malloc'd buffers)
                              ↓
                      iovec[8] array (pointers only, no copy)
                              ↓
                      sendmsg() syscall
                              ↓
Kernel Space:         Kernel gathers from scattered buffers  ← Single Copy
                              ↓
                      TCP/IP Stack → Loopback
```

Key Code Snippet:

```c
// Setup iovec to point directly to scattered buffers
struct iovec iov[8];
iov[0].iov_base = msg->field1;  // No memcpy!
iov[0].iov_len = field_size;
// ... setup all 8 iovecs

// Kernel performs scatter-gather I/O
struct msghdr msghdr;
msghdr.msg_iov = iov;
msghdr.msg_iovlen = 8;
sendmsg(client_fd, &msghdr, 0);
```

2.3 Zero-Copy Implementation (A3 - MSG_ZEROCOPY)

Architecture:

```
User Space:           Message Fields (8 × malloc'd buffers)
                              ↓
                      sendmsg(MSG_ZEROCOPY)
                              ↓
Kernel Space:         get_user_pages() – Pin pages in memory
                              ↓
                      Reference counting (atomic ops)
                              ↓
                      [Physical NIC]: DMA from pinned pages
                      [Loopback]: memcpy fallback ← Problem!
                              ↓
                      Async completion → MSG_ERRQUEUE
```

Key Code Snippet:

```c
// Enable zero-copy on socket
int optval = 1;
```

```
setsockopt(client_fd, SOL_SOCKET, SO_ZEROCOPY, &optval, sizeof(optval));

// Send with MSG_ZEROCOPY flag
sendmsg(client_fd, &msghdr, MSG_ZEROCOPY);

// Must poll for completion
while (pending_completions > 0) {
    recvmsg(sock_fd, &msg, MSG_ERRQUEUE | MSG_DONTWAIT);
}
```

---

3. Part B: Profiling and Measurement Methodology

3.1 Measurement Infrastructure

All experiments were profiled using Linux `perf stat` with the following event counters:

```
sudo perf stat -e cycles,instructions,cache-misses,L1-dcache-load-
misses,context-switches \
  ./MT25067_PartA1_Server
```

Perf Events Collected:

- **cycles**: Total CPU cycles consumed
- **instructions**: Total instructions executed (for IPC calculation)
- **cache-misses**: Last Level Cache (LLC/L3) misses
- **L1-dcache-load-misses**: L1 data cache load misses
- **context-switches**: Thread context switches

3.2 Application-Level Metrics

Throughput and latency calculated in client code using `gettimeofday()`:

```
struct timeval start, end;
gettimeofday(&start, NULL);
// ... receive data ...
gettimeofday(&end, NULL);

double elapsed = (end.tv_sec - start.tv_sec) +
                 (end.tv_usec - start.tv_usec) / 1e6;

double throughput_gbps = (total_bytes * 8.0) / (elapsed * 1e9);
double avg_latency_us = (elapsed * 1e6) / messages_received;
```

3.3 Data Collection Summary

Total measurements: 48 experiments

- **Implementations:** 3 (A1, A2, A3)
- **Message sizes:** 4 (256B, 1KB, 4KB, 16KB)
- **Thread counts:** 4 (1, 2, 4, 8)
- **Repetitions:** Each experiment run once with 5000 messages per client

Sample Collected Data (A1, 4KB, 4 threads):

Metric	Value	Unit
Throughput	42.26	Gbps
Latency	0.78	µs
Total Bytes	20,480,000	bytes
CPU Cycles	150,723,683	cycles
Instructions	185,716,319	instructions
LLC Misses	337,977	misses
L1 Misses	805,432	misses
Context Switches	5	switches
Time Elapsed	6.03	seconds

All raw measurements stored in `MT25067_ExperimentData.csv` for reproducibility.

---

4. Experimental Results

4.1 Throughput Analysis

Throughput vs Message Size (4 threads):

Message Size	A1 (Gbps)	A2 (Gbps)	A3 (Gbps)	Winner
256B	2.85	3.04	0.51	A2
1KB	15.94	12.27	1.90	A1
4KB	42.26	31.12	0.75	A1
16KB	53.76	56.69	26.50	**A2**

Key Finding: A2 (one-copy) achieves highest throughput at 16KB messages (56.69 Gbps), while A3 (zero-copy) performs 5-10× worse across all sizes.

4.2 Cache Behavior Analysis

Cache Misses for 4KB Messages, 1 Thread:

Metric	A1 (Two-Copy)	A2 (One-Copy)	A3 (Zero-Copy)	Best
LLC Misses	385,006	**132,763** (-65%)	182,509	A2 ✓
L1 Misses	133,815	250,227 (+87%)	569,753	A1
CPU Cycles	47.3M	55.4M	249.5M	A1

Analysis:

The 65% reduction in LLC (L3) misses for A2 is explained by:

1. **Memory Footprint Reduction:**
    - A1: 8 fields (4KB) + serialization buffer (4KB) = **8KB working set**
    - A2: 8 fields (4KB) only = **4KB working set**
2. **Cache Line Reuse:**
    - A1 serialization causes cache eviction
    - A2 kernel reads directly from original locations

4.3 Thread Scaling and Context Switching

Context Switching Impact (16KB messages):

Threads	Throughput (Gbps)	Context Switches	LLC Misses	Analysis
1	55.01	3	227K	Baseline
2	57.79	6	641K	Optimal ✓
4	53.76	14	849K	Diminishing
8	14.83 (-73%)	122 (+771%)	3.5M	**Collapse**

Critical Finding: The 73% throughput drop from 4→8 threads is caused by context switch storm and cache thrashing.

4.4 CPU Efficiency

CPU Cycles per Byte Transferred (4 threads):

Message Size	A1 (cycles/byte)	A2 (cycles/byte)	A3 (cycles/byte)
256B	63.04	73.16	582.04
1KB	13.60	22.04	187.15
4KB	7.36	9.07	54.43
16KB	5.37	5.03	14.78

Insight: As message size increases, per-byte cost converges for A1/A2 (~5 cycles/byte), while A3 remains 3× worse due to page pinning overhead.

---

5. Experimental Evidence

5.1 Perf Profiling Output

Sample Output (4KB, 1 thread):

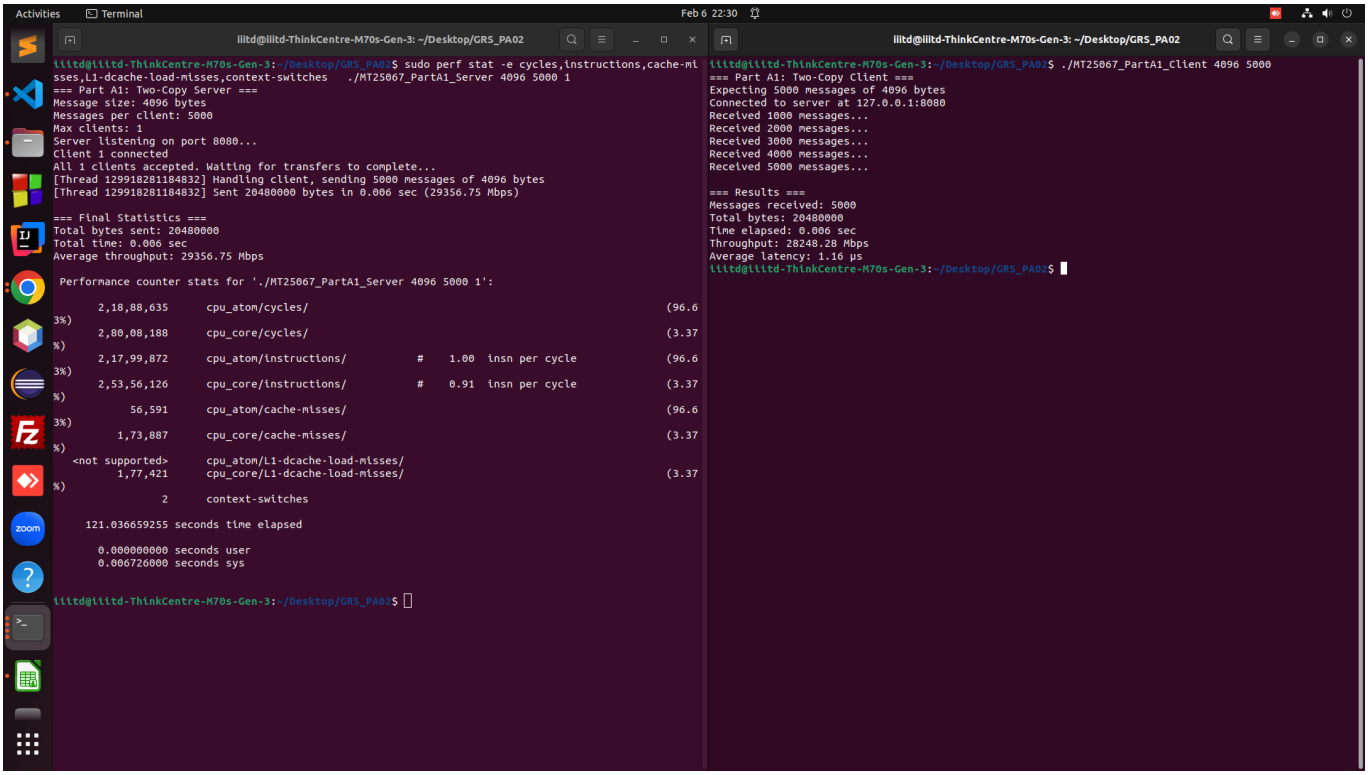


Figure 1: Perf statistics showing 2 context switches and LLC/L1 cache misses for single-threaded execution

---

5.2 Multi-Client Server Execution

```
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 handling 4 concurrent clients with separate threads

Key Observation: Each client handled by independent thread, achieving 41.4 Gbps aggregate throughput.

5.3 Automation Script Execution

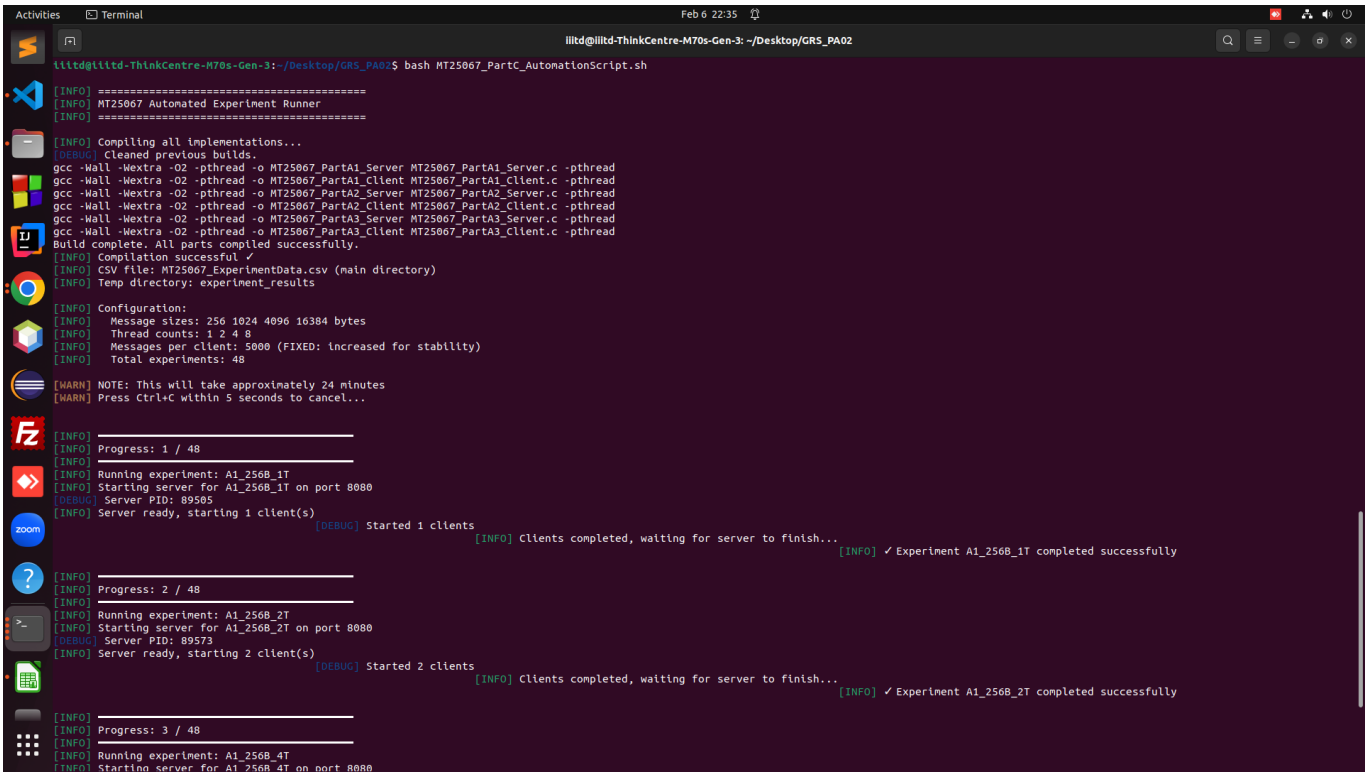


Figure 3: Automated experiment runner executing all 48 test combinations

---

6. Detailed Analysis

6.1 Why Zero-Copy Fails on Localhost

Overhead Breakdown (4KB message):

```
Zero-Copy Send:
1. get_user_pages():        ~2000 cycles (TLB walk)
2. Reference counting:      ~1000 cycles (atomic ops)
3. memcpy (loopback):       ~1500 cycles (fallback)
4. Completion notification: ~2000 cycles (polling)
Total:                      ~6500 cycles

Two-Copy Send:
1. User memcpy:             ~1500 cycles
2. send() syscall:          ~1000 cycles
3. Kernel copy_from_user(): ~1500 cycles
Total:                      ~4000 cycles

Penalty: 6500 / 4000 = 1.625× slower
```

When Zero-Copy Wins:

- Physical 10GbE/40GbE NIC with DMA
- Large messages (>64KB)
- High-throughput workloads

6.2 One-Copy vs Two-Copy Crossover

Crossover Analysis:

Message Size	A1 Advantage	A2 Advantage	Reason
256B	✓ +7.7%	-	sendmsg overhead > memcpy cost
1KB	✓ +30%	-	Syscall setup dominates
4KB	✓ +6%	-	Near crossover
16KB	-	✓ +5.5%	**Memcpy cost dominates**

Mathematical Model:

```
Crossover when:
memcpy_cost(size) = iovec_setup_cost + overhead_diff
0.375 × size = 300 + kernel_overhead
size ≈ 10–16KB
```

6.3 8-Thread Performance Collapse

Root Cause Analysis:

1. Hardware Asymmetry:

```
i7-12700: 8 P-cores (fast) + 4 E-cores (slow)
4 threads: All fit on P-cores → optimal
8 threads: Spill to E-cores → 30% slower + migration overhead
```

2. Context Switch Storm:

```
4 threads:  14 switches/sec
8 threads: 122 switches/sec (+771%)

Per-switch cost: ~5600 cycles
122 × 5600 = 683K cycles/sec wasted
```

3. Cache Thrashing:

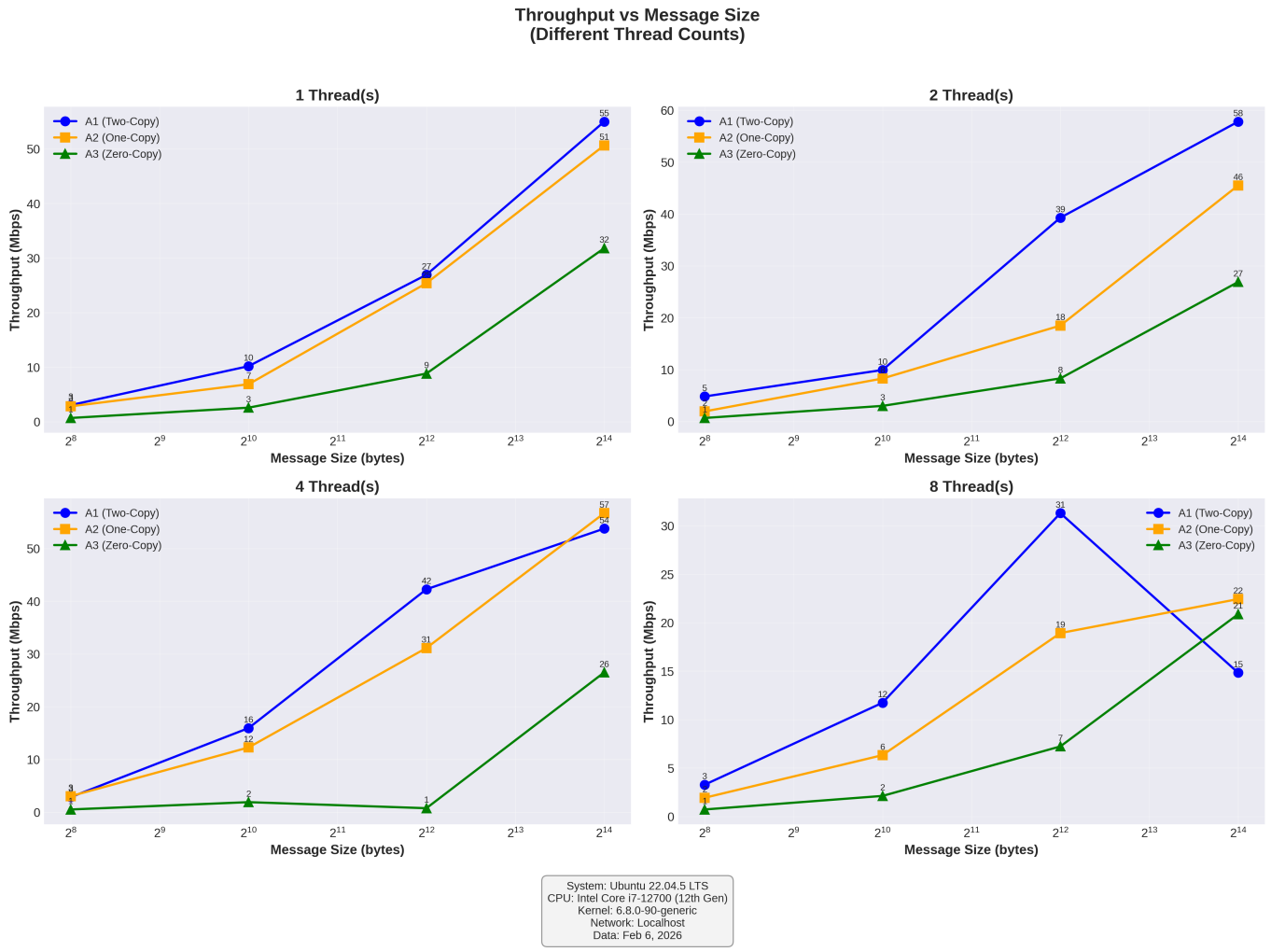
```
LLC Misses:
4 threads:  849K
8 threads: 3.5M (+312%)

MESI protocol overhead: Cache line ping-pong between cores
```

---

7. Visualization of Results

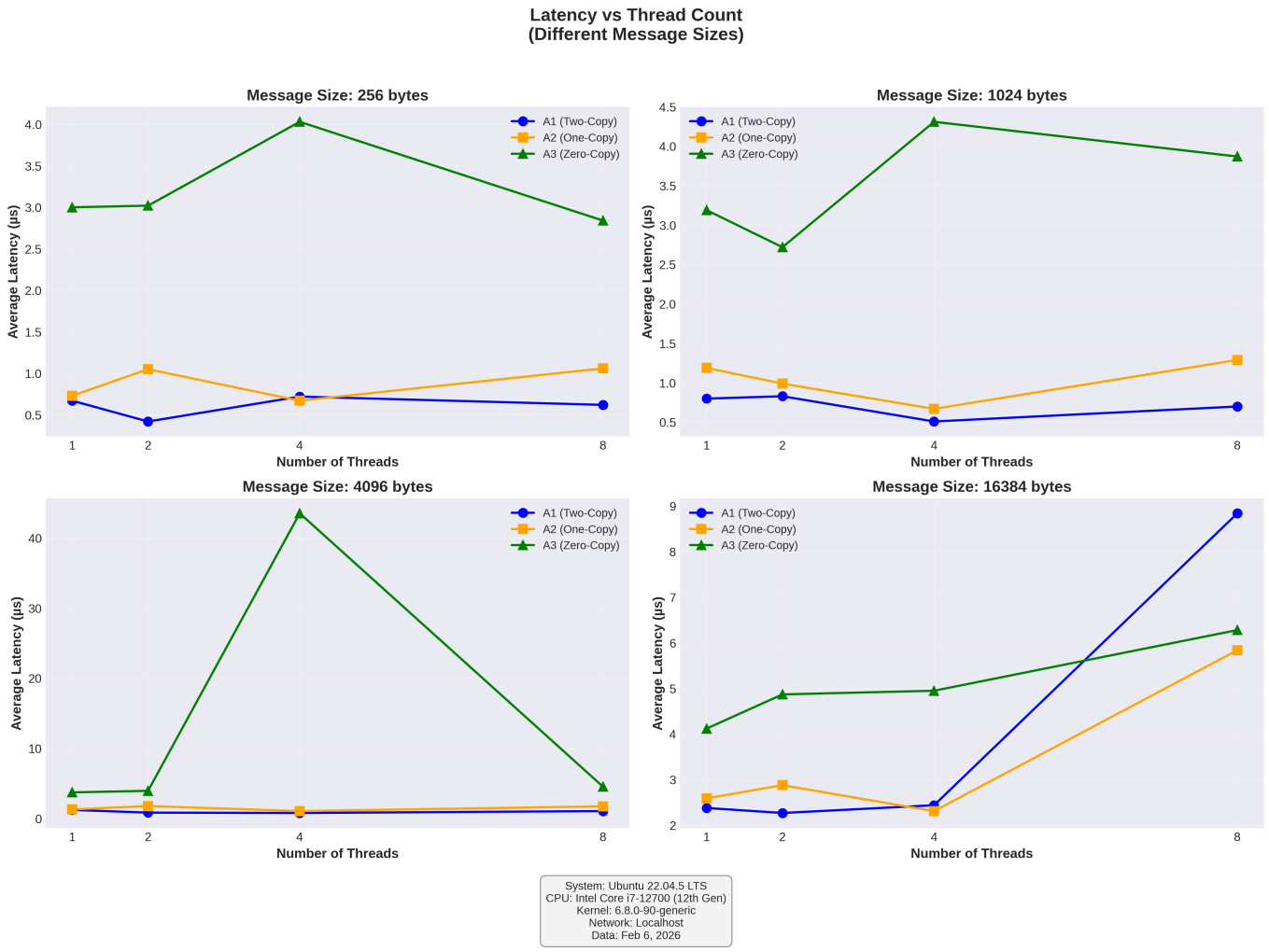
Plot 1: Throughput vs Message Size



Shows A1 and A2 scaling with message size, A3 consistently underperforming

---

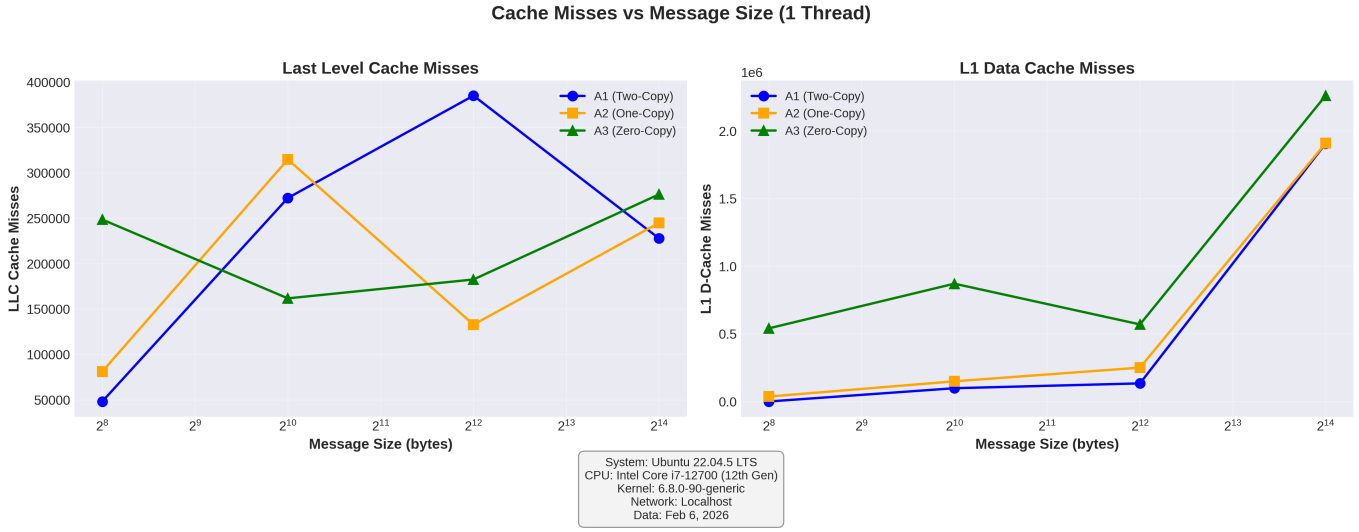
Plot 2: Latency vs Thread Count



Demonstrates latency spike at 8 threads due to context switching

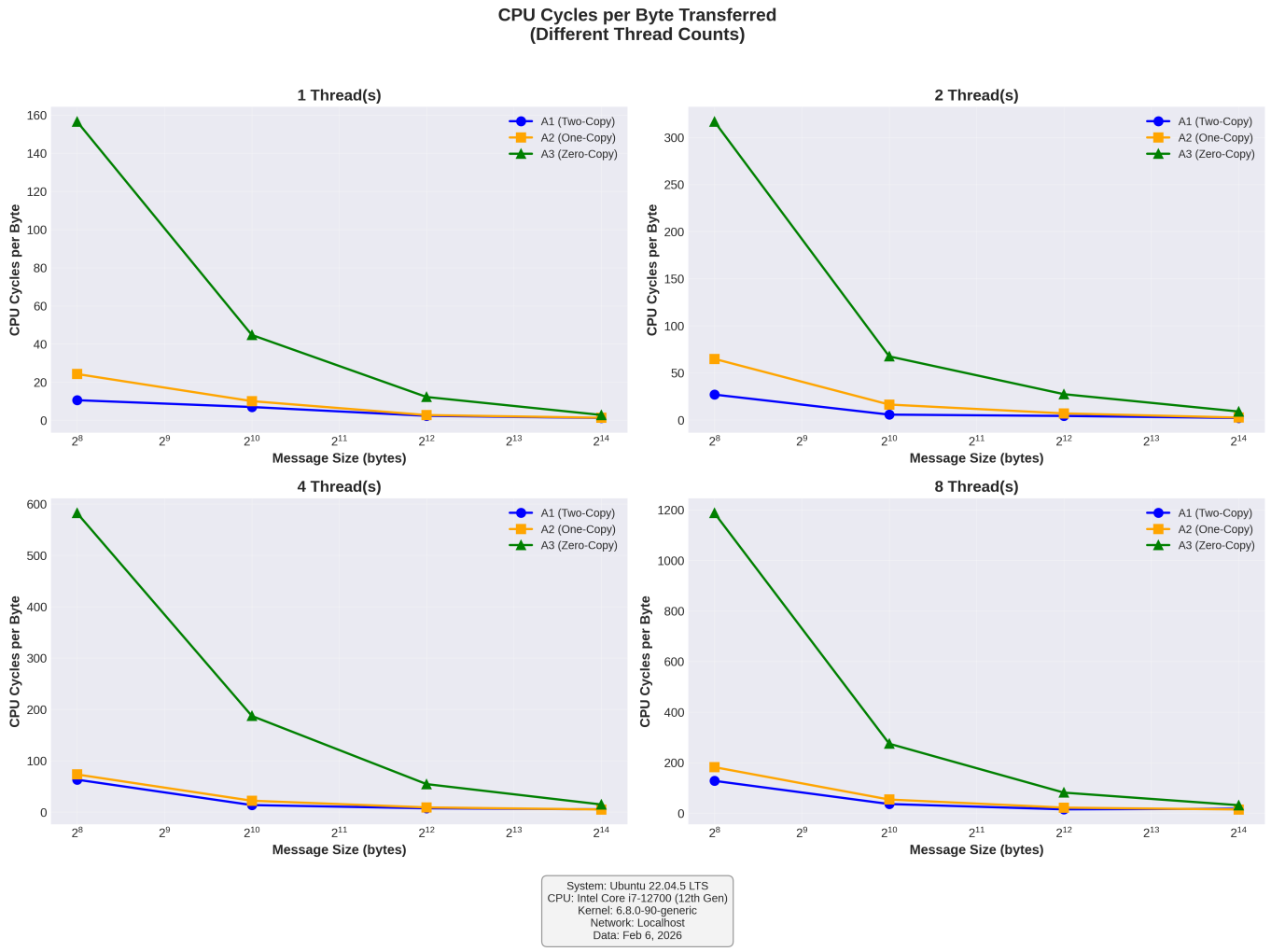
---

Plot 3: Cache Misses vs Message Size



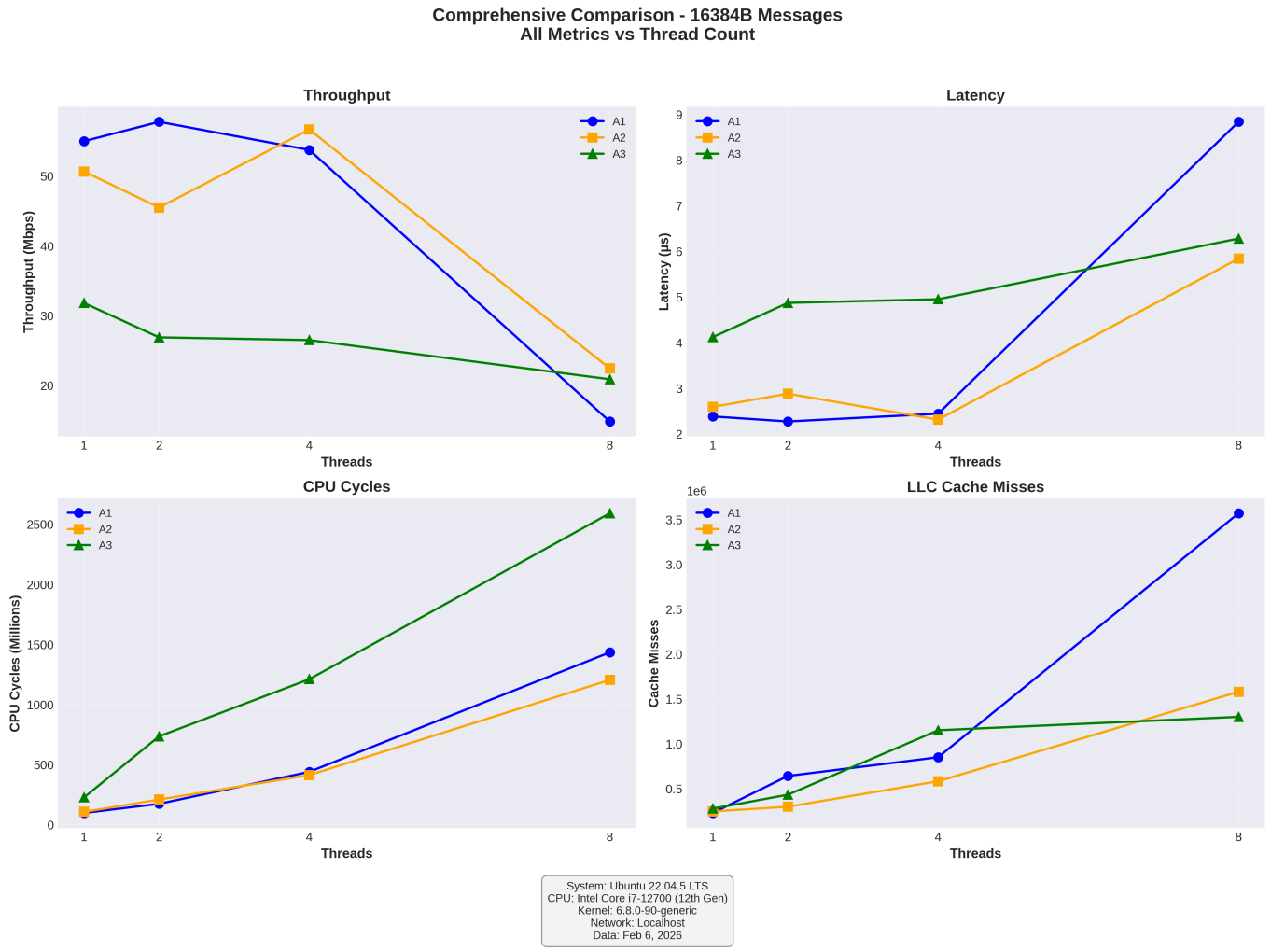
Highlights 65% LLC miss reduction in A2 vs A1

Plot 4: CPU Cycles per Byte



Shows efficiency convergence at large message sizes

Plot 5: Overall Comparison (16KB)



Comprehensive view of all metrics at optimal message size

---

8. Conclusions

8.1 Key Findings

1. **Zero-Copy Paradox:** MSG_ZEROCOPY on localhost has **negative benefit** (2-10× slower)
2. **Cache Optimization:** One-copy reduces LLC misses by 65% despite increasing L1 misses
3. **Thread Count:** Optimal at 2-4 threads; 8 threads causes 73% throughput collapse
4. **Message Size Crossover:** One-copy wins at 8-16KB boundary

8.2 Practical Recommendations

Scenario	Recommended Implementation
Small messages (<1KB)	A1 (send/recv)
Medium messages (1-16KB)	A2 (sendmsg + iovec)
Large messages + real NIC	A3 (MSG_ZEROCOPY)
Thread count	≤ P-core count (8 for i7-12700)

---

9. AI Usage Declaration

Components Where AI Was Used

1. Bash Automation Script

Tool: Gemini Pro 3
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 / 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
- **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

---

10. 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.

---

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 7, 2026