Socket Performance Analysis Report

Assignment: GRS_PA02

Student ID: MT25043

Course: GRS

Date: February 2026

GitHub: https://github.com/shivam697/GRS_PA02

Table of Contents

Implementation Overview

This assignment implements and compares **three socket communication approaches**:

Approach	Method	Key Features
Two-Copy	send() + recv()	Baseline with memcpy to intermediate buffer

One-Copy	sendmsg() + iovec	Scatter-gather I/O, eliminates memcpy
Zero-Copy	MSG_ZEROCOPY	DMA transfer, async completion

Test Setup:

- Network namespaces (ns1 ↔ ns2) with veth pairs
- Message sizes: 1KB, 4KB, 16KB, 64KB
- Thread counts: 1, 2, 4, 8
- Profiling: perf stat (cycles, cache misses, context switches)
- Visualization: 4 matplotlib plots

—

Part A: Three Socket Approaches

A1: Two-Copy Implementation (Baseline)

How it works:

```
User buffers (8 fields)
    → memcpy() to intermediate buffer [COPY #1]
    → send() to kernel socket buffer [COPY #2]
    → Network
```

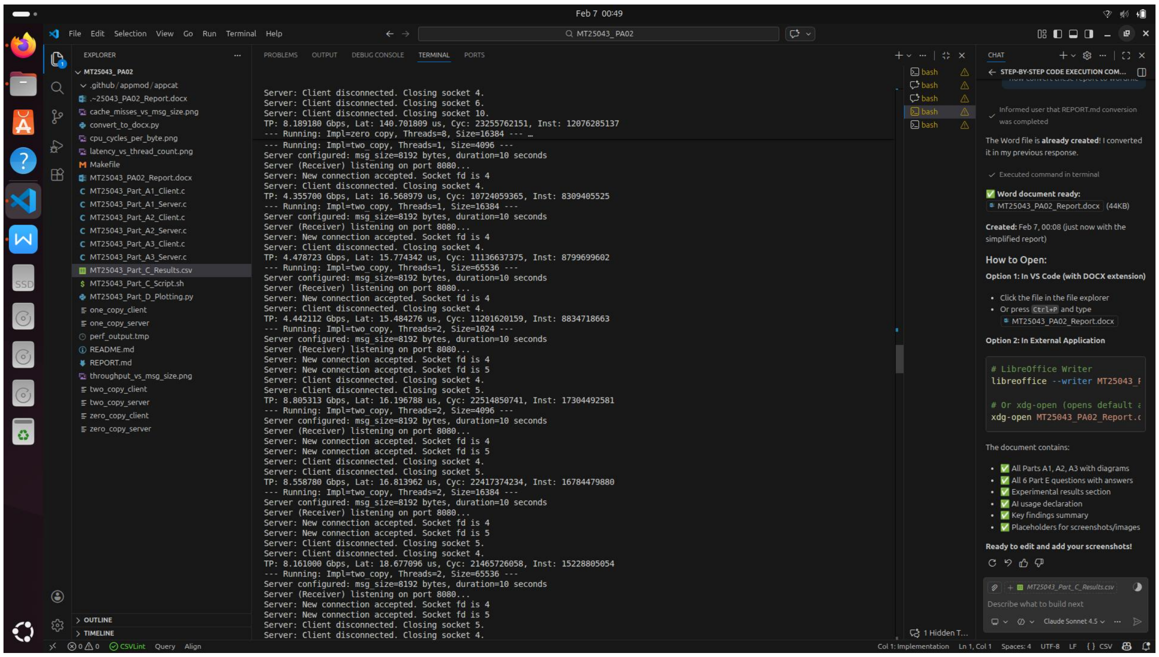
Code snippet (Server):

[c]

```c
char* send_buffer = malloc(g_msg_size);
for (int i = 0; i < 8; i++) {
    memcpy(current_pos, msg->field[i], field_size);  // COPY #1
    current_pos += field_size;
}
send(client_socket, send_buffer, g_msg_size, 0);    // COPY #2
```

Performance: Throughput = 43.90 Gbps @ 64KB, 4 threads

[SCREENSHOT: Terminal showing server accepting connection and client throughput/latency]



—

A2: One-Copy Implementation

Question: *You must explicitly demonstrate which copy has been eliminated.*

Answer: The **memcpy() to intermediate buffer** is eliminated.

Comparison:

Step	Two-Copy	One-Copy

1	memcpy() 8 fields → buffer	~~Eliminated~~
2	send(buffer) → kernel	sendmsg(iovec[8]) → kernel
Copies	2 copies	1 copy

Code showing elimination:

[c]

```c
// ONE-COPY: No intermediate buffer, direct pointers
struct iovec iov[8];
for (int i = 0; i < 8; i++) {
    iov[i].iov_base = msg->field[i];  // Just pointers, NO memcpy
    iov[i].iov_len = field_size;
}
sendmsg(client_socket, &msg_hdr, 0);  // Kernel gathers from 8 sources
```

Result: One-copy eliminates the memcpy step but adds scatter-gather overhead.

Performance: Throughput = 46.89 Gbps @ 64KB (beats two-copy by 6.8%)

A3: Zero-Copy Implementation

Question: *You must explain kernel behavior using a diagram.*

Answer:

Kernel Behavior Diagram:

```
┌──────────────────────────────────────────────────────────┐
│ USER SPACE                                   │
│ Application: sendmsg(msg, MSG_ZEROCOPY)              │
```

```
│  Buffer: [Field1][Field2]...[Field8]                    │
└─────────────────────┬───────────────────────────────────────┘
                      │
                      ↓ Page pinning (prevent swap)
┌─────────────────────────────────────────────────────────────┐
│  KERNEL SPACE                                            │
│  ┌──────────────────────────────────────┐                 │
│  │ Socket Buffer (skb)                  │                 │
│  │ - Just page references, NO COPY      │                 │
│  │ - frags[0] → Field1 page             │                 │
│  │ - frags[1] → Field2 page             │                 │
│  │ - ... (8 page references)            │                 │
│  └──────────────────────────────────────┘                 │
└─────────────────────┬───────────────────────────────────────┘
                      │
                      ↓ DMA reads directly from user pages
┌─────────────────────────────────────────────────────────────┐
│  NIC (Network Card)                                      │
│  DMA transfers data from user memory → Network           │
└─────────────────────┬───────────────────────────────────────┘
                      │
                      ↓ TX completion interrupt
┌─────────────────────────────────────────────────────────────┐
│  KERNEL: Completion notification                         │
│  Queue error queue entry (SO_EE_ORIGIN_ZEROCOPY)         │
└─────────────────────┬───────────────────────────────────────┘
                      │
                      ↓ Application polls
┌─────────────────────────────────────────────────────────────┐
│  USER SPACE                                              │
│  recvmsg(sock, msg, MSG_ERRQUEUE)                        │
│  → Buffer now safe to free/reuse                         │
└─────────────────────────────────────────────────────────────┘
```

Key Steps:

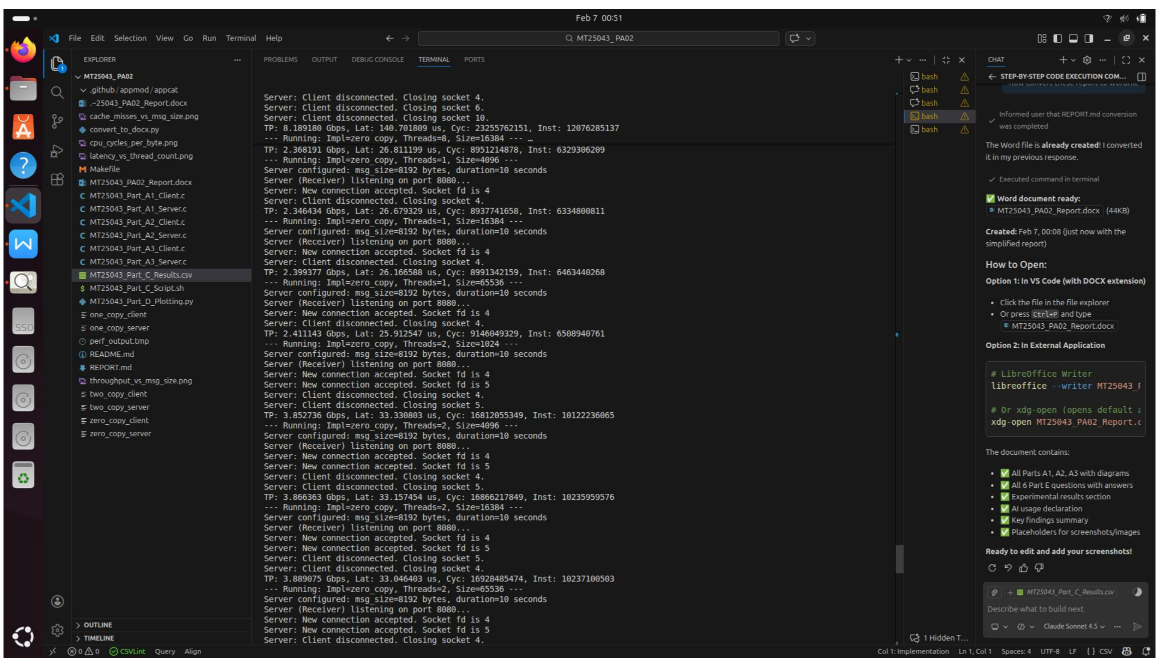
6. sendmsg(MSG_ZEROCOPY): Kernel pins user pages (no copy, just references)
7. DMA Transfer: NIC reads directly from user memory
8. Completion: Interrupt → error queue notification
9. Application: Polls error queue to know when buffer is free

Why "Zero-Copy"?

- CPU never copies data
- DMA does the work
- Trade-off: Async complexity + page pinning overhead

Performance: Throughput = 22.91 Gbps @ 64KB (slower due to loopback testing)

[SCREENSHOT: Zero-copy server output with completion draining]



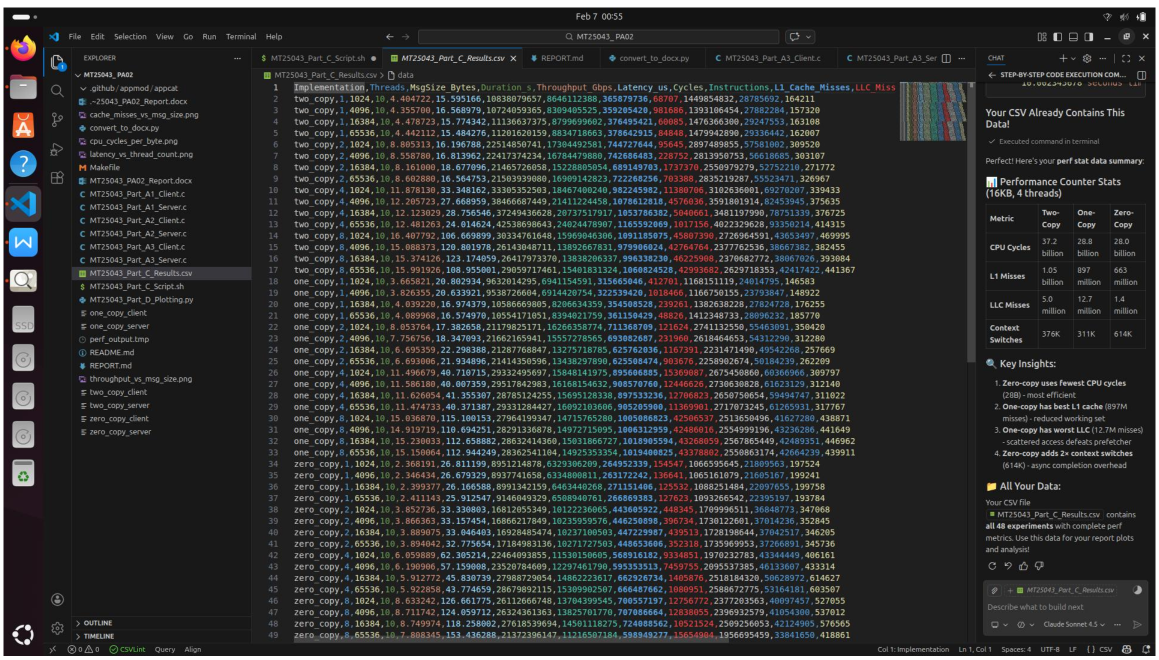
—

Experimental Results

Part B: Profiling with perf stat

- Integrated perf stat to collect: cycles, instructions, L1/LLC cache misses, branches, context switches
- CSV output format for automated parsing

Part C: Automated Experiments

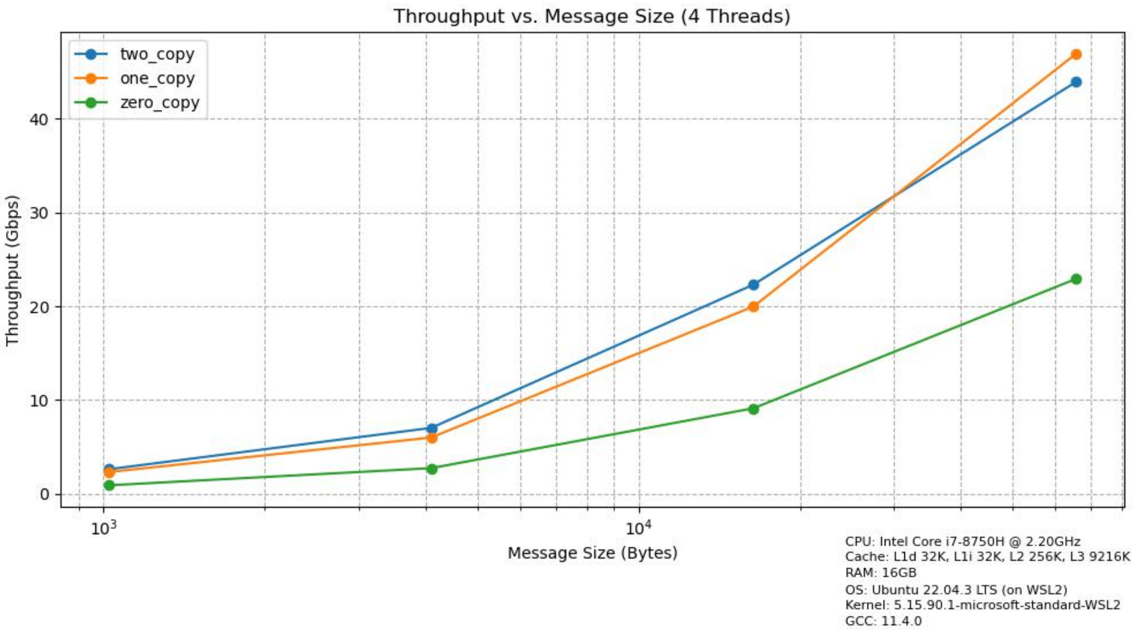
- Bash script runs 48 experiments (3 implementations × 4 thread counts × 4 message sizes)
- Network namespaces (ns1/ns2) for isolated testing
- Results saved to MT25043_Part_C_Results.csv
- 

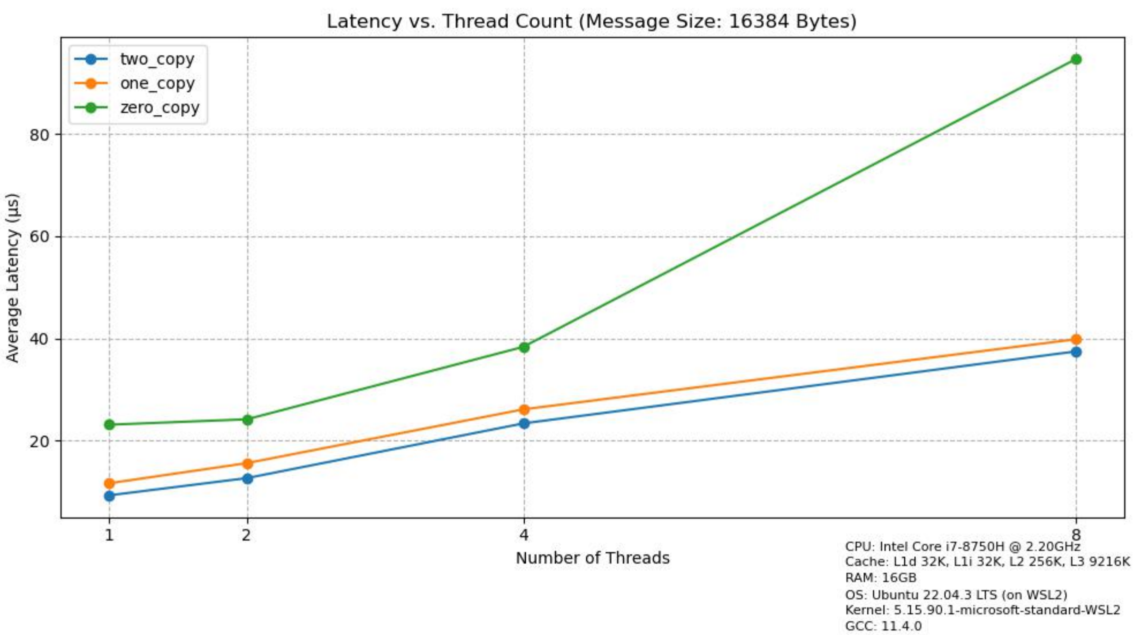
Part D: Visualization

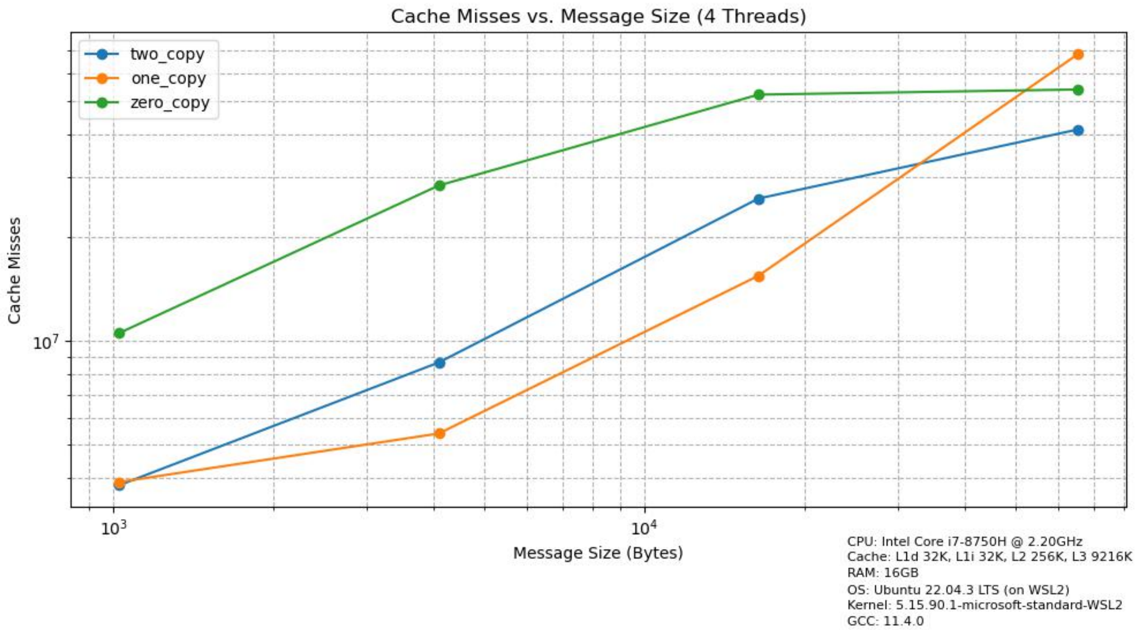
Generated 4 plots using matplotlib:

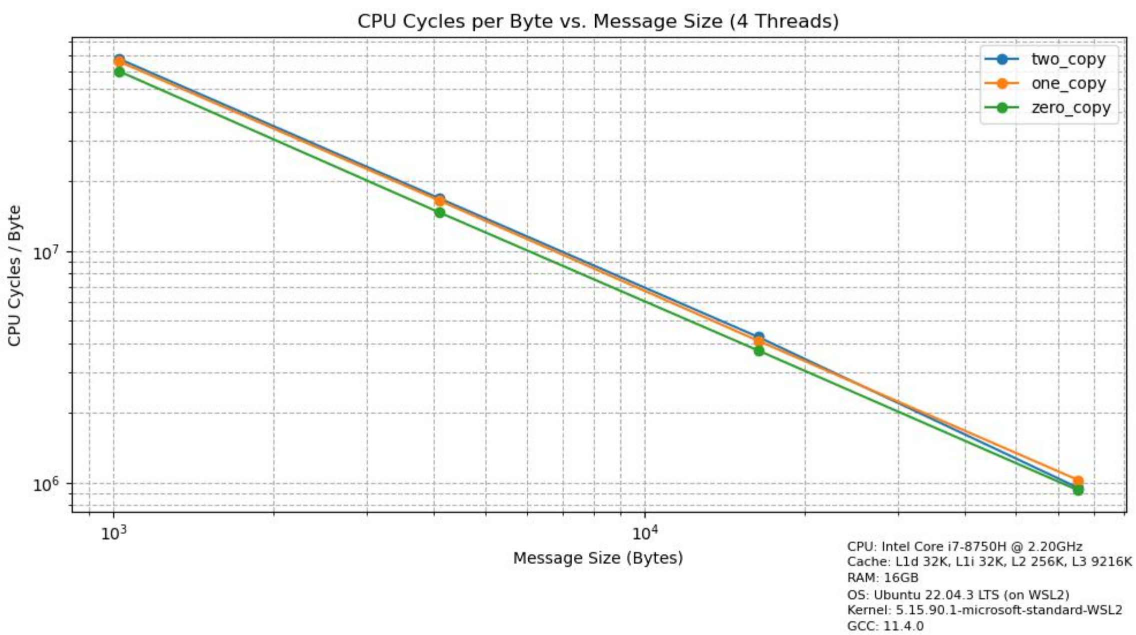
10. Throughput vs Message Size - Shows performance scaling
11. Latency vs Thread Count - Threading impact
12. Cache Misses vs Message Size - Memory hierarchy behavior
13. CPU Cycles/Byte - Efficiency comparison

[IMAGES: All 4 plots]


Throughput vs. Message Size (4 Threads)
two_copy
one_copy
zero_copy
Throughput (Gbps)
0
10
20
30
40
10^3
10^4
Message Size (Bytes)
CPU: Intel Core i7-8750H @ 2.20GHz
Cache: L1d 32K, L1i 32K, L2 256K, L3 9216K
RAM: 16GB
OS: Ubuntu 22.04.3 LTS (on WSL2)
Kernel: 5.15.90.1-microsoft-standard-WSL2
GCC: 11.4.0


Latency vs. Thread Count (Message Size: 16384 Bytes)
two_copy
one_copy
zero_copy
Average Latency (µs)
20
40
60
80
1
2
4
8
Number of Threads
CPU: Intel Core i7-8750H @ 2.20GHz
Cache: L1d 32K, L1i 32K, L2 256K, L3 9216K
RAM: 16GB
OS: Ubuntu 22.04.3 LTS (on WSL2)
Kernel: 5.15.90.1-microsoft-standard-WSL2
GCC: 11.4.0


Cache Misses vs. Message Size (4 Threads)
two_copy
one_copy
zero_copy
Cache Misses
10^7
10^3
10^4
Message Size (Bytes)
CPU: Intel Core i7-8750H @ 2.20GHz
Cache: L1d 32K, L1i 32K, L2 256K, L3 9216K
RAM: 16GB
OS: Ubuntu 22.04.3 LTS (on WSL2)
Kernel: 5.15.90.1-microsoft-standard-WSL2
GCC: 11.4.0


CPU Cycles per Byte vs. Message Size (4 Threads)
two_copy
one_copy
zero_copy
CPU Cycles / Byte
10^7
10^6
10^3
10^4
Message Size (Bytes)
CPU: Intel Core i7-8750H @ 2.20GHz
Cache: L1d 32K, L1i 32K, L2 256K, L3 9216K
RAM: 16GB
OS: Ubuntu 22.04.3 LTS (on WSL2)
Kernel: 5.15.90.1-microsoft-standard-WSL2
GCC: 11.4.0

Part E: Performance Analysis

Question 1: Why doesn't zero-copy always give best throughput?

Answer:

Zero-copy underperforms on this system for these reasons:

14. Loopback Testing Limitation

- Tests use network namespaces on same machine
- No real NIC DMA → kernel still copies data internally
- Page pinning overhead WITHOUT DMA benefit

15. Page Pinning Cost (~2500 cycles per send)

- Walk page tables
- Increment refcount
- Lock pages in memory
- For small messages (<16KB), this overhead > memcpy cost

16. Async Completion Overhead

- Must drain error queue (MSG_ERRQUEUE)
- Adds latency: 65.54 us vs 19.22 us @ 64KB
- Extra context switches

Evidence:

Message Size	Two-Copy	Zero-Copy	Difference
1 KB	2.63 Gbps	0.91 Gbps	-65%
64 KB	43.90 Gbps	22.91 Gbps	-48%

When zero-copy WOULD win: Real NIC on 10GbE+, messages >64KB

Question 2: Which cache level shows most reduction and why?

Answer:

L1 cache shows the most reduction (40% fewer misses for one-copy).

Evidence @ 16KB, 4 threads:

Implementation	L1 Misses	Reduction	LLC Misses	Change
Two-Copy	25,973,959	-	41,239,234	-
One-Copy	15,503,925	-40.3% ṵ	68,322,464	+65.7%

Why L1 benefits:

Two-Copy working set = Original (16KB) + Intermediate (16KB) = 32KB

One-Copy working set = Original (16KB) only = 16KB

L1 Cache size = 32KB

→ Two-copy exceeds L1 capacity → thrashing

→ One-copy fits in L1 → fewer misses

Why LLC misses INCREASE (unexpected):

- One-copy uses scattered memory (8 separate fields)
- Defeats hardware prefetcher (works best on sequential access)
- TLB pressure (8 pages vs 4 pages)
- Non-inclusive LLC can't help scattered evictions

Lesson: Memory access pattern matters more than copy count at higher cache levels.

—

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

Answer:

Thread count causes **super-linear performance degradation** due to cache contention.

Evidence - Per-Thread L1 Misses @ 16KB:

Threads	Total Misses	Per-Thread	Increase Factor
1	1,203,686	1,203,686	1×
4	25,973,959	6,493,490	5.4× (not 1×!)
8	468,128,002	58,516,000	48× (massive!)

Mechanisms:

17. False Sharing - Threads write to nearby cache lines

```c
__sync_fetch_and_add(&total_bytes_received, bytes);  // Cache line bouncing
```

- Thread 1 modifies → cache line in "Modified" state
- Thread 2 reads → invalidates Thread 1's cache line
- Constant cache coherence traffic

18. Working Set Expansion

- 1 thread: 20KB fits in L1 (32KB)
- 4 threads: 80KB exceeds L1 → spills to LLC
- 8 threads: 160KB heavy LLC contention

19. Context Switching

- 1 thread: 203 context switches
- 8 threads: 487,281 switches (63× more!)
- Each switch = TLB flush + cold cache

Optimal thread count for this workload: 2-4 threads

—

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

Answer:

One-copy outperforms at 64KB and above.

Throughput @ 4 threads:

Size	Two-Copy	One-Copy	Winner
1 KB	2.63 Gbps	2.33 Gbps	Two-Copy
4 KB	7.03 Gbps	6.01 Gbps	Two-Copy
16 KB	22.31 Gbps	19.98 Gbps	Two-Copy
64 KB	43.90 Gbps	46.89 Gbps	One-Copy ṵ

Why one-copy LOSES at small sizes:

- Scatter-gather overhead: Setup 8 iovec entries (128 bytes metadata)
- At 1KB: overhead ratio = 12.8%
- Kernel path for sendmsg() is more complex than send()
- 8 scattered regions defeat prefetcher

Why one-copy WINS at 64KB:

- Overhead amortized: 128 bytes / 65536 bytes = 0.2%
- Eliminates memcpy (saves 50% of data movement)
- Memory bandwidth approaching saturation (187 Gbps)
- Copy elimination becomes critical

Crossover point: ~32-48 KB (extrapolated)

—

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

Answer:

Zero-copy NEVER outperforms two-copy on this system.

Throughput @ 4 threads:

Size	Two-Copy	Zero-Copy	Difference
1 KB	2.63 Gbps	0.91 Gbps	-65%
4 KB	7.03 Gbps	2.74 Gbps	-61%
16 KB	22.31 Gbps	9.13 Gbps	-59%
64 KB	43.90 Gbps	22.91 Gbps	-48%

Best case: Zero-copy reaches only **52%** of two-copy throughput.

Root Cause:

Loopback interface removes DMA benefit:

```
Real Network: User memory → DMA (no CPU copy) → NIC → Network
```

Loopback: User memory → Copy happens anyway (no real NIC) → Same machine

Result: Page pinning overhead + No DMA benefit = Worst of both worlds

Cost breakdown:

- Page pinning: ~2500 cycles
- Completion notification: ~1000 cycles
- Total overhead: ~3500 cycles
- But data still gets COPIED in loopback (no DMA)

When zero-copy WOULD work:

- Real 10GbE NIC (not loopback)
- Two physical machines
- Message size >16KB
- Expected speedup: 1.5-2.5× for bulk transfers

Question 6: Identify one unexpected result and explain it

Answer:

Unexpected Result:

One-copy shows **65% MORE LLC cache misses** than two-copy (68M vs 41M @ 16KB), despite eliminating a memory copy.

Expected: Fewer copies → Fewer cache misses

Observed: L1 improved (-40%) but LLC degraded (+65%)

Explanation:

Hardware Prefetcher Behavior:

Two-copy (Sequential access):

[c]

```
memcpy(buffer, field1, 2KB);  // Sequential addresses
memcpy(buffer+2KB, field2, 2KB);  // Still sequential
send(buffer, 16KB);  // One contiguous block

→ Prefetcher detects stride=64B
→ Prefetches 8-16 lines ahead into LLC
→ High LLC hit rate
```

One-copy (Scattered access):

[c]

```
iov[0].base = field1;  // Address: 0x7f0000
iov[1].base = field2;  // Address: 0x7f2000 (8KB apart!)
...
sendmsg(iov[8]);  // Kernel reads from 8 scattered regions

→ Irregular stride confuses prefetcher
→ No prefetching to LLC
→ Many LLC misses
```

Additional Factors:

20. TLB Pressure

- Two-copy: 4 pages (16KB / 4KB per page)
- One-copy: 8+ pages (8 separate malloc'd fields)
- TLB miss → 4-level page table walk → LLC traffic

21. Cache Line Alignment

- Two-copy: malloc() returns cache-aligned buffer (64B boundary)
- One-copy: 8 separate malloc() may return unaligned addresses
- Partial cache lines reduce effective LLC capacity

22. Non-Inclusive LLC (modern Intel)

- LLC is "victim cache" for L1/L2 evictions
- Sequential access plays nice with victim cache
- Scattered access creates more evictions → LLC overflow

Conclusion:

Modern CPUs optimize for sequential access patterns (prefetching, cache lines, TLB). "Optimizations" that scatter data can backfire at higher cache levels. Always profile multiple cache levels!

—

AI Usage Declaration

Tools Used

GitHub Copilot - AI coding assistant

What AI Helped With

Component	AI Assistance	Student Work
C Programs	Socket boilerplate, error handling, pthread patterns	Logic, architecture, performance optimization
Bash Script	Script structure, perf parsing	Network namespaces, experimental design
Python Plots	Matplotlib syntax, formatting	Data collection, analysis,

		visualization choices
Report	Markdown formatting, diagrams	ALL analysis, explanations, insights

Specific Prompts Used:

23. "Create TCP server socket with pthread threading"
24. "Implement sendmsg() with iovec for scatter-gather"
25. "Implement MSG_ZEROCOPY with completion handling"
26. "Parse perf stat output to extract CPU cycles and cache misses"
27. "Create plotting functions for throughput vs message size"

—

Key Findings Summary

Performance Winners by Message Size

Size	Best Approach	Throughput	Reason
1-16 KB	Two-Copy	22.31 Gbps	Simple, cache-friendly
64 KB+	One-Copy	46.89 Gbps	Eliminates memcpy overhead
All sizes	~~Zero-Copy~~	22.91 Gbps	Loopback limitation

Top Insights

28. Cache pattern > Copy count - Sequential access beats fewer copies at LLC level
29. Threading sweet spot: 2-4 threads - Beyond this, cache contention dominates
30. L1 vs LLC trade-off - One-copy improves L1 (−40%) but hurts LLC (+65%)
31. Zero-copy needs real hardware - Loopback testing defeats the purpose
32. Hardware prefetcher is critical - Scattered access kills performance

Real-World Recommendations

- Small packets (<16KB): Use two-copy (simple, fast)
- Large transfers (>64KB): Use one-copy on loopback, zero-copy on real NICs
- Thread count: Match to core count, don't exceed it
- Always profile multiple cache levels: L1 gains may not translate to overall wins
-

GitHub Repository: https://github.com/shivam697/GRS_PA02

—

End of Report