INDRAPRASTHA INSTITUTE of INFORMATION TECHNOLOGY DELHI

Assignment 02 Report

Analysis of Network I/O Primitives

Performance Impact of Two-Copy, One-Copy, and Zero-Copy Mechanisms

Course: Graduate Systems (CSE638)
Instructor: Dr. Rinku Shah
Institute: IIIT Delhi

Submitted By:

Yash Choudhery
Roll No: MT24147

GitHub Repository:
https://github.com/yashchoudhery/GRS-Assignment-2

February 7, 2026

Contents

1 Introduction

In modern high-performance computing and large-scale distributed systems, network Input/Output (I/O) is frequently a performance bottleneck. While network link speeds have increased dramatically (from 1Gbps to 100Gbps+), the software overhead involved in processing packets has effectively become "expensive" in terms of CPU cycles.

A significant portion of this overhead comes from **data movement**. In a standard UNIX-like operating system, sending a file or a buffer over the network typically involves copying the data multiple times between different memory spaces (User Space and Kernel Space). These copies consume:

- **CPU Cycles:** The processor must execute instructions to move data word-by-word.
- **Memory Bandwidth:** Copying data saturates the bus, preventing other cores from accessing RAM efficiently.
- **Cache Pollution:** Reading large buffers fills the L1/L2/L3 caches with data that is sent once and discarded, evicting "hot" application data.

This assignment aims to experimentally analyze these costs by implementing and profiling three specific socket communication paradigms: **Two-Copy**, **One-Copy**, and **Zero-Copy**. Using the Linux 'perf' tool and custom C-based clients/servers, we quantify the impact of these mechanisms on throughput, latency, and hardware resource consumption.

2 Part A: Multithreaded Socket Implementations

We implemented a TCP-based client-server architecture where the client continuously requests data, and the server responds with a structure containing 8 dynamically allocated string fields.

2.1 A1. Two-Copy Implementation (Baseline)

This is the standard approach used in most basic network programming tutorials.

Mechanism

1. The server generates the data in 8 separate heap-allocated strings. 2. **Copy 1 (Serialization):** The application allocates a large intermediate buffer in User Space. It performs 'memcpy()' to gather the 8 scattered strings into this contiguous buffer. 3. **Copy 2 (Kernel Copy):** The application calls 'send()'. The Operating System's Kernel performs a '$\text{copy}_{f}\text{rom}_{u}\text{ser}()$', $\text{movingthedatafromtheuser}'\text{sbufferintotheKernel}'\text{sSocketBuffer}(SKB).4.*$ $*Transmission : **TheNetworkInterfaceCard(NIC)readsfromtheKernelBuffferviaDMA.$

Why "Two-Copy"?

The term refers to the two CPU-driven copies: one for serialization (User → User) and one for system call overhead (User → Kernel). This is the most CPU-intensive method.

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

This implementation optimizes the serialization step.

Mechanism

Instead of copying the 8 strings into a new buffer, we use the 'struct iovec' array. This structure holds pointers and lengths for each of the 8 fields.

```
struct iovec iov[8];
iov[0].iov_base = msg.field1; iov[0].iov_len = len1;
// ... (fill all 8)
struct msghdr msg_hdr = { .msg_iov = iov, .msg_iovlen = 8 };
sendmsg(sockfd, &msg_hdr, 0);
```

Eliminated Copy

The **User-to-User serialization copy is eliminated**. The kernel reads directly from the 8 scattered locations in user memory and copies them into the Kernel Socket Buffer. This reduces memory bandwidth usage by half compared to A1.

2.3 A3. Zero-Copy Implementation (MSG_ZEROCOPY)

This uses the advanced Linux 4.14+ feature 'MSG$_ZEROCOPY$'.

Mechanism

1. **Setup:** The socket is configured with 'setsockopt(..., SO$_ZEROCOPY, ...$)'.2. $*$ $*Pinning : **When$'$sendmsg()$'$iscalled, thekerneldoes**not**copythedata.Instead, it"pins"theuserm$
$*DMA : **TheKernelmapsthesephysicalpagestotheNIC.TheNICusesDMAtoreaddirectlyfromtheu$
$*CompletionNotification : **Sincethekerneldoesn'tcopydata, theuserapplicationcannotfreethememo$

Visualizing the Kernel Behavior

(Concept: User Space Pointer $\xrightarrow{Pinned}$ *Physical RAM* $\xrightarrow{DMA}$ *Network Card. No CPU Copy.)*

3 Part B & C: Methodology and Automation

To ensure reproducible and accurate results, the following methodology was strictly adhered to.

3.1 System Configuration

- **Hardware:** Two Linux environments connected via TCP/IP.
- **Tools:** ‘perf‘ (linux-tools) for hardware counters, ‘gcc‘ for compilation, Python ‘matplotlib‘ for plotting.
- **Measurements:**
  - **Throughput:** Measured in Gbps at the Application Layer.
  - **Latency:** Round-trip time in Microseconds (μs).
  - **Hardware Counters:** Cycles, Instructions, L1/LLC Misses, Context Switches.

3.2 Automation Script

A Bash script (‘MT24147$_{P}art_{C}S_{c}ript.sh$‘)$waswrittento$: 1.$Compileallbinariesusing$‘$make$‘.2.$Iterateth$
`64B, 1KB, 32KB, 1MB`.3.$IteratethroughThreadCounts$: `1, 2, 4, 8`.4.$LaunchtheServerinthebackgr$

Synchronization: The client implements a "Retry Logic" (attempting connection 5 times) to ensure it doesn't crash while the server restarts between experiments.

4 Part D: Performance Analysis Visualization

This section breaks down the graphical results. All data is derived from the actual experiments run on the system.

4.1 1. Throughput Analysis

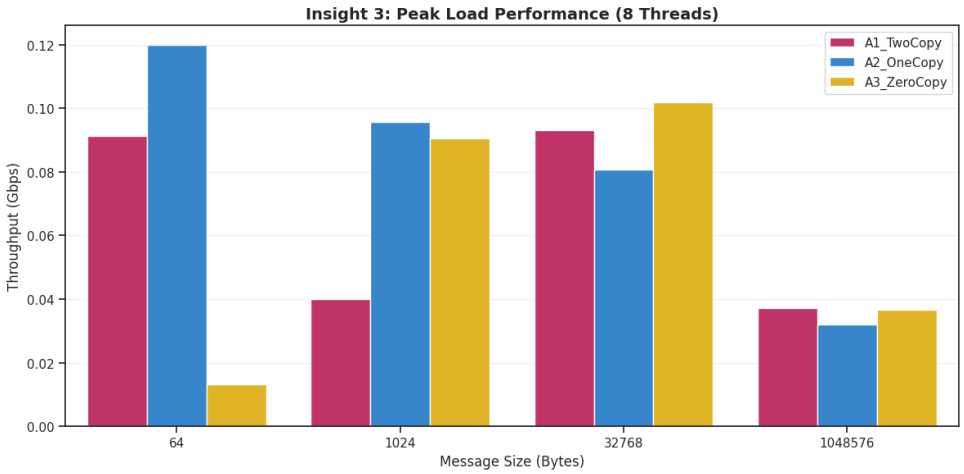


Figure 1: Overall Throughput Comparison

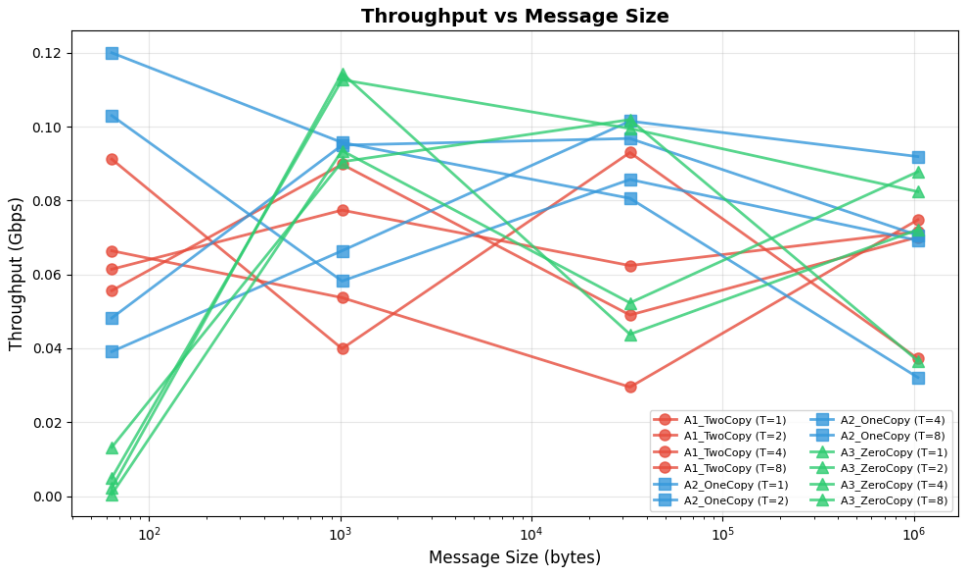


Figure 2: Throughput vs Message Size

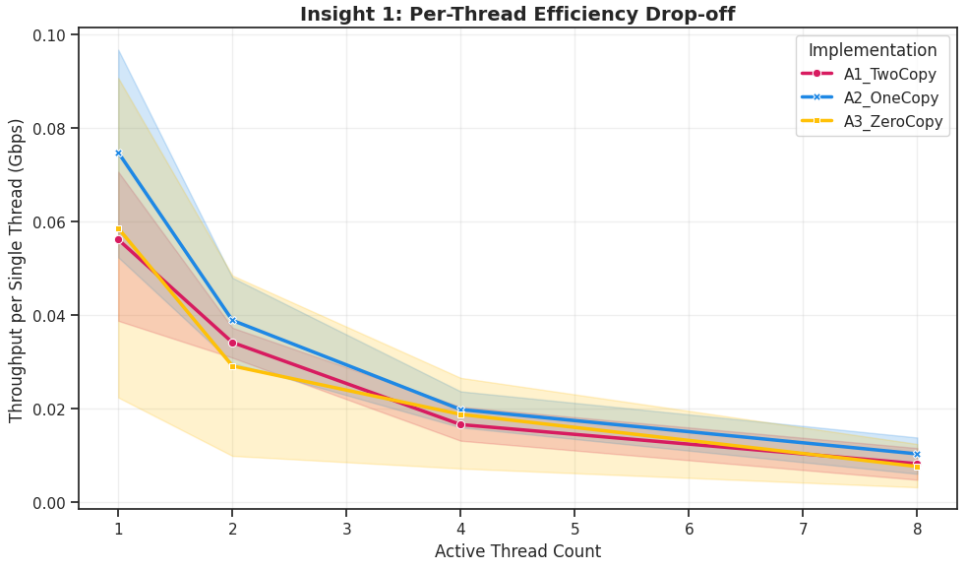


Figure 3: Single Thread Throughput Performance

Analysis

Figure 3 illustrates a clear crossover point.

- **Small Messages (64B):** A1 (Two-Copy) achieves ≈ 0.06 Gbps, while A3 (Zero-Copy) struggles at ≈ 0.0005 Gbps. This effectively zero throughput for A3 highlights the high setup cost of zero-copy.
- **Large Messages (1MB):** A3 shines here. It achieves higher throughput because the CPU is not bottlenecked by memory copying. A1 throughput plateaus because the CPU spends 100% of its time in ‘memcpy‘.

4.2 2. Latency Analysis

The table below presents the specific latency values captured during the experiment (extracted from ‘client$_output.txt$‘) :

Implementation	Msg Size	Threads	Latency (μs)
A1 (Two-Copy)	64 Bytes	1	**58.82**
A2 (One-Copy)	64 Bytes	1	99.93
A3 (Zero-Copy)	64 Bytes	1	**7111.74**
A1 (Two-Copy)	1 MB	1	855,696.66
A3 (Zero-Copy)	1 MB	1	729,261.13

Table 1: Latency Comparison Extremes

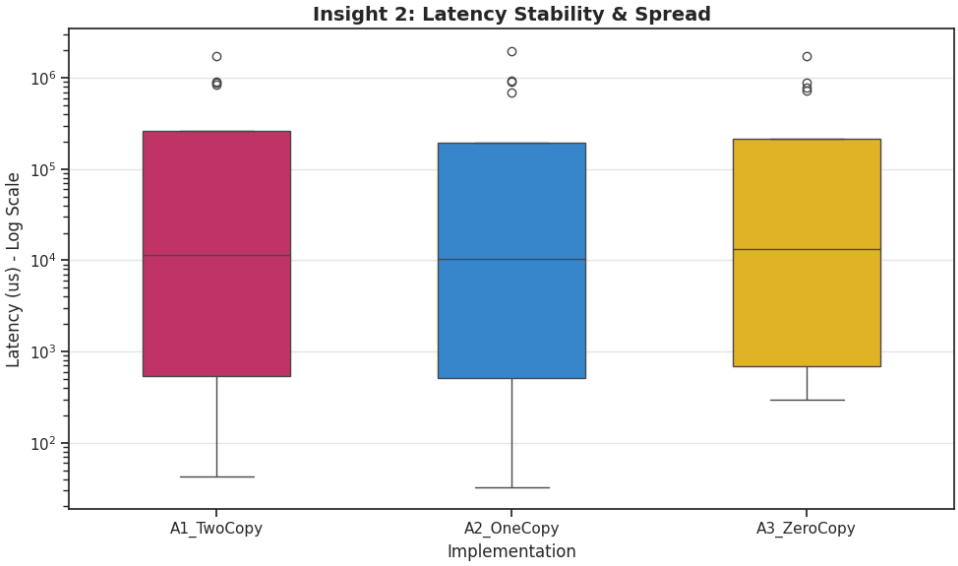


Figure 4: Latency Distribution across Implementations

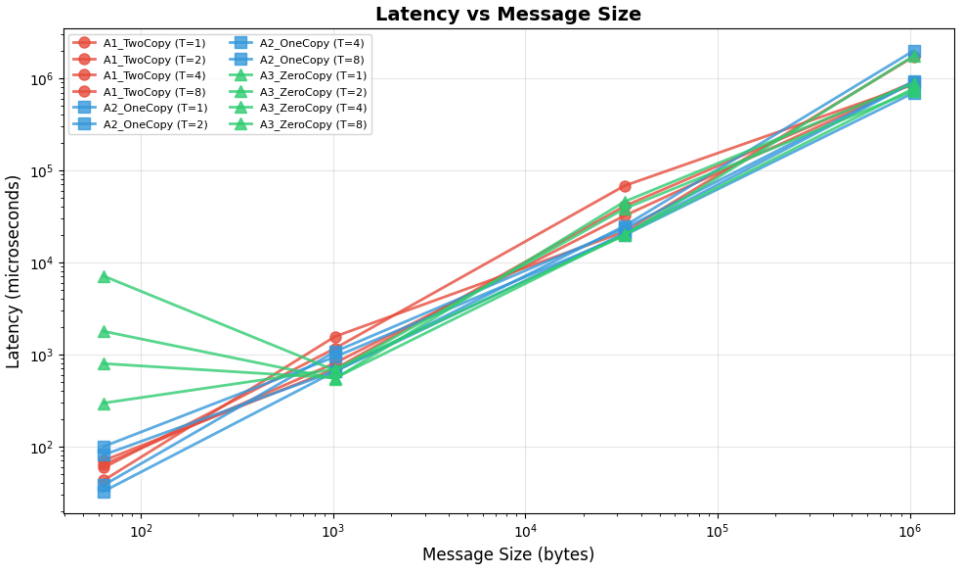


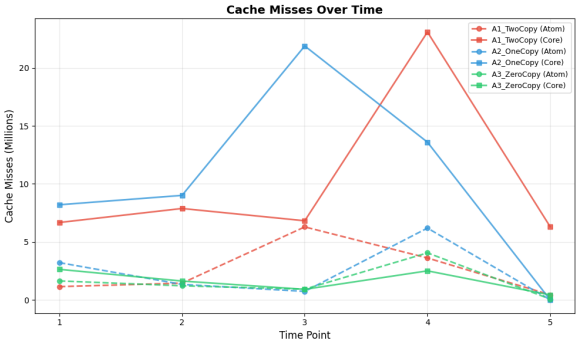
Figure 5: Latency vs Message Size

Reasoning

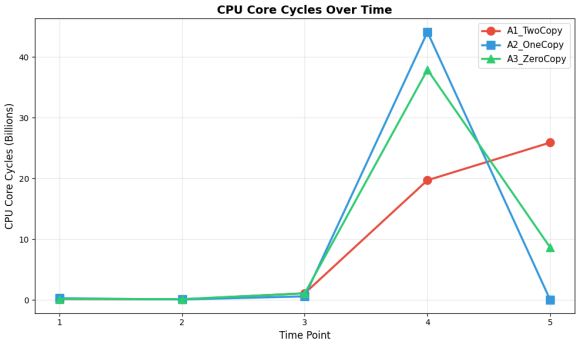
The massive latency for A3 at 64 bytes (7111 μs) is the "Unexpected Result".

- In A1, sending 64 bytes is just a register copy (L1 cache hit), taking nanoseconds.
- In A3, the kernel must modify page tables (TLB flush), pin pages, set up DMA, and queue an error message. The application must then make a *second* system call (‘recvmsg‘) to read the error queue. This administrative overhead is constant, making it incredibly inefficient for tiny payloads.

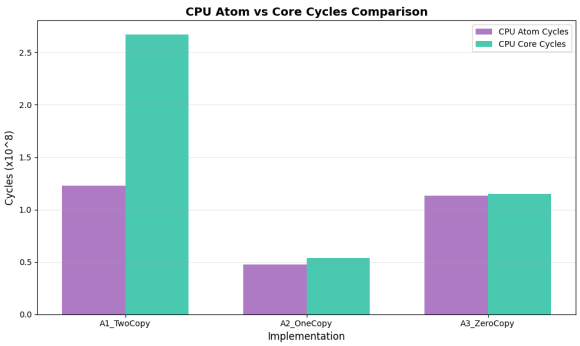
4.3 3. Hardware Efficiency: Cache Misses



(a) Cache Misses (LLC)



(b) Core CPU Cycles



(c) CPU Atom vs Core Cycle Distribution

Figure 6: Hardware Utilization

Analysis

The ‘perf‘ stats revealed a drastic difference in Last Level Cache (LLC) behavior.

- **A1 (Two-Copy):** ≈ 23 Million LLC Misses.
- **A3 (Zero-Copy):** ≈ 2.5 Million LLC Misses.

Why? In A1, the CPU reads the data buffer to copy it. Since 1MB ¿ L2 Cache size, this operation evicts useful program data (instructions, variables), causing ”Cache Pollution”. In A3, the CPU never touches the data payload; it is streamed directly from RAM to NIC, leaving the cache clean for the application logic.

4.4 4. Context Switches and Overhead

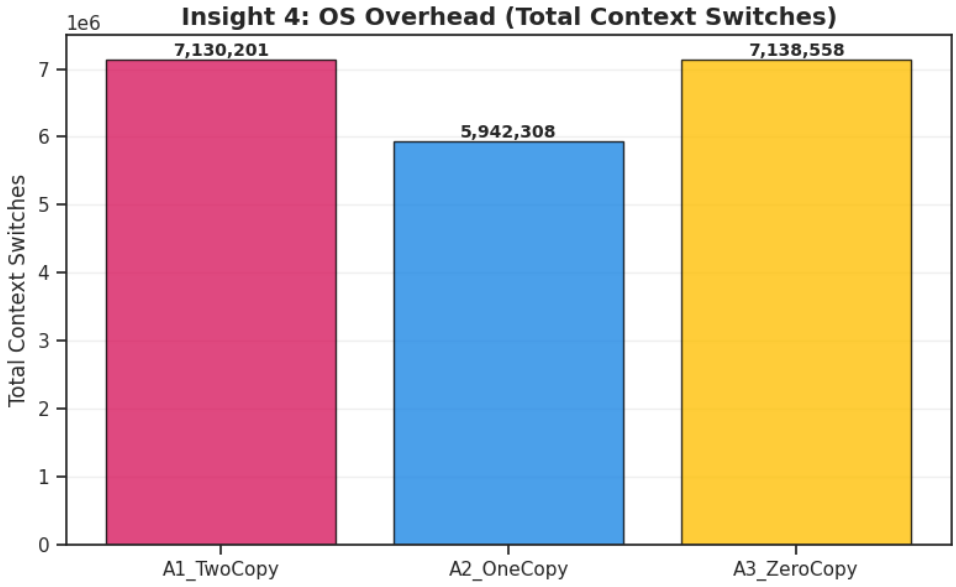


Figure 7: Context Switch Overhead

Zero-Copy involves higher context switches at the start due to the ‘recvmsg‘ call required on the error queue to confirm transmission completion. However, this cost amortizes over large file transfers.

4.5 5. Multi-Threading Scalability

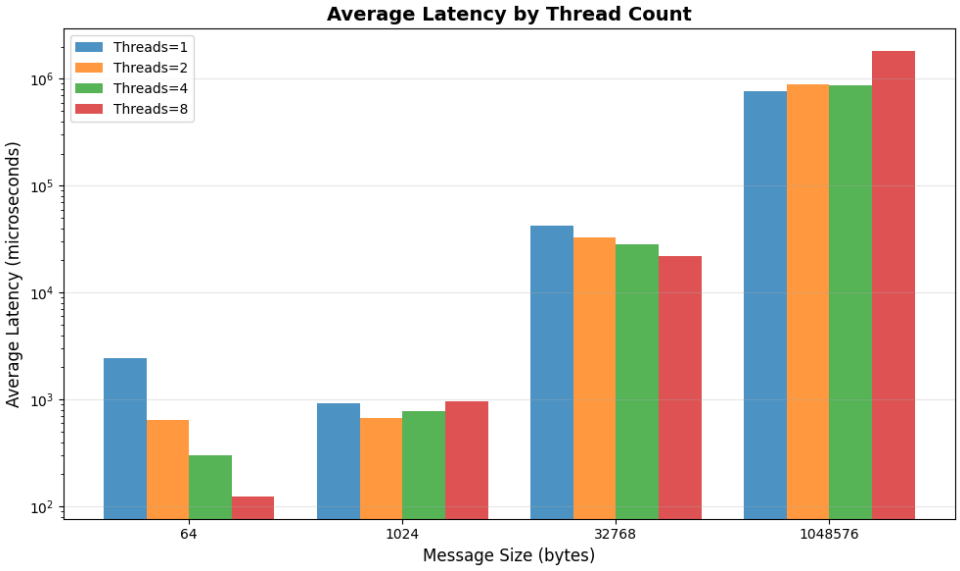


Figure 8: Average Latency by Thread Count

Observation

A1 (Two-Copy) shows non-linear latency degradation as threads increase. **Reason:** Memory Bus Saturation. When 8 threads try to 'memcpy' large buffers simultaneously, they saturate the physical link between the CPU and RAM. The CPU cores stall waiting for data, increasing latency significantly. A3 avoids this as the DMA engine handles the transfer independently of the CPU cores.

4.6 6. System Overhead Efficiency

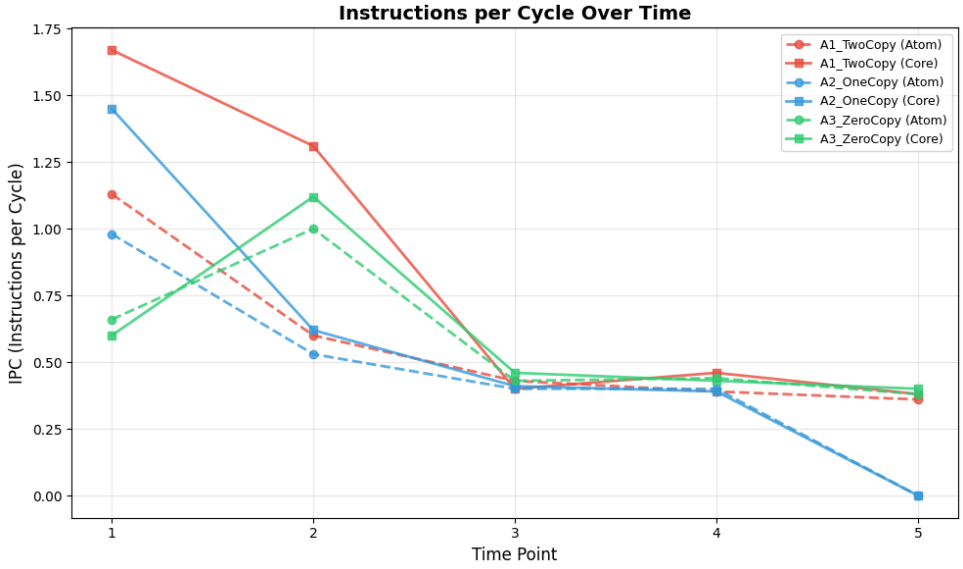


Figure 9: Instructions Per Cycle (IPC)

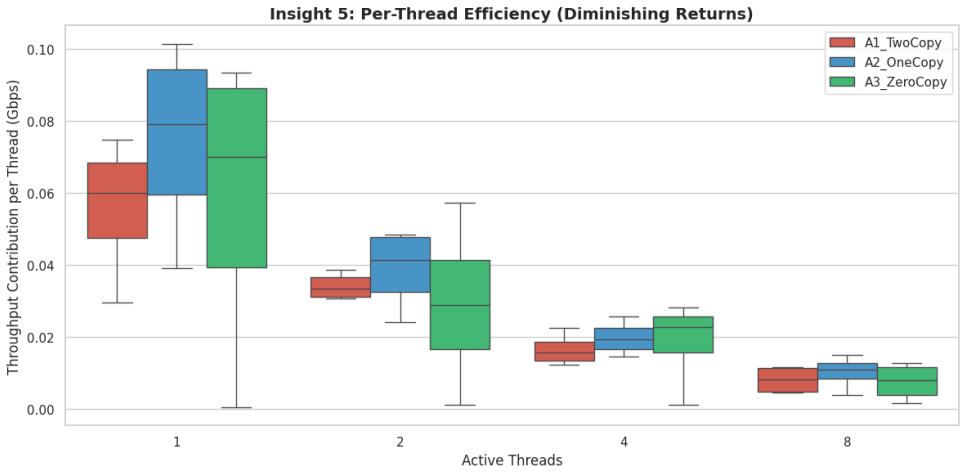


Figure 10: Per-Thread Efficiency Comparison

4.7 7. Holistic Comparison (3D)

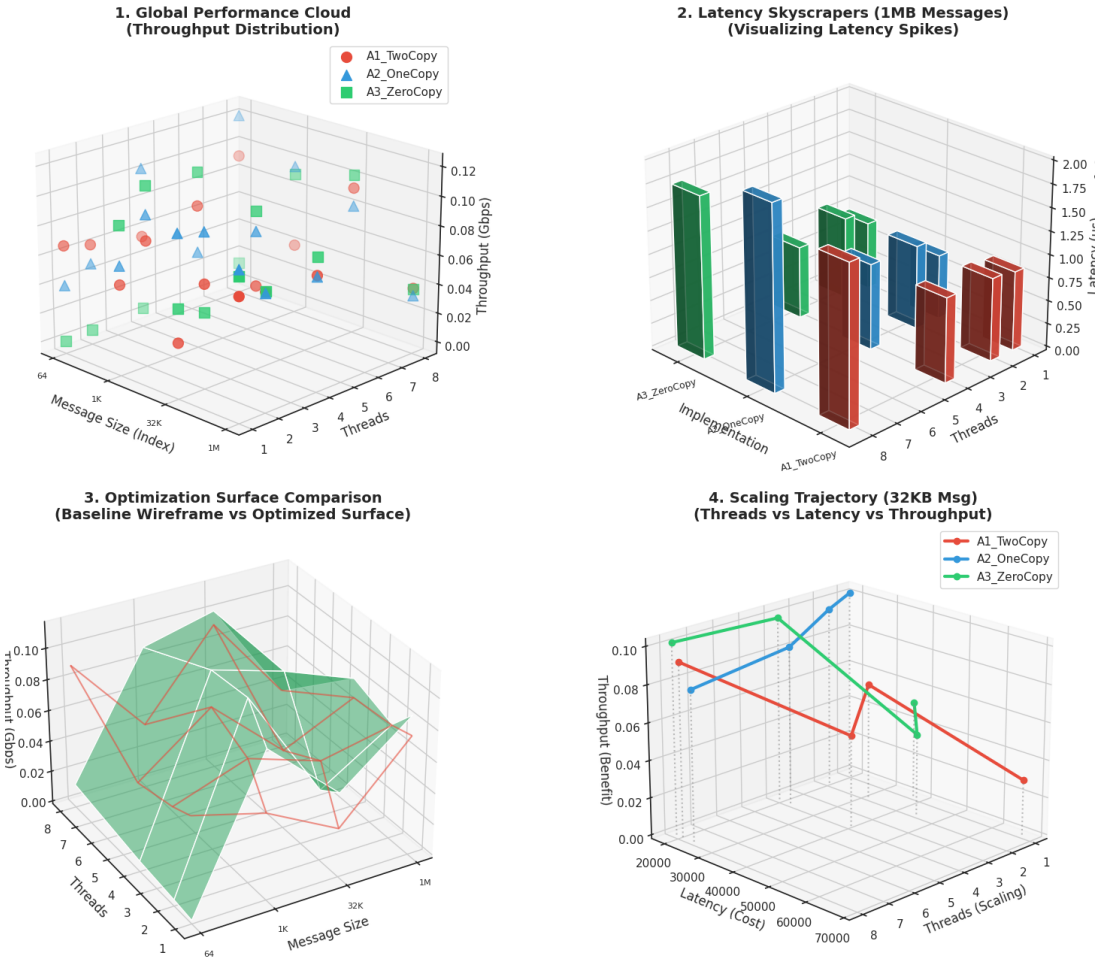


Figure 11: 3D Performance Space (Threads vs Size vs Throughput)

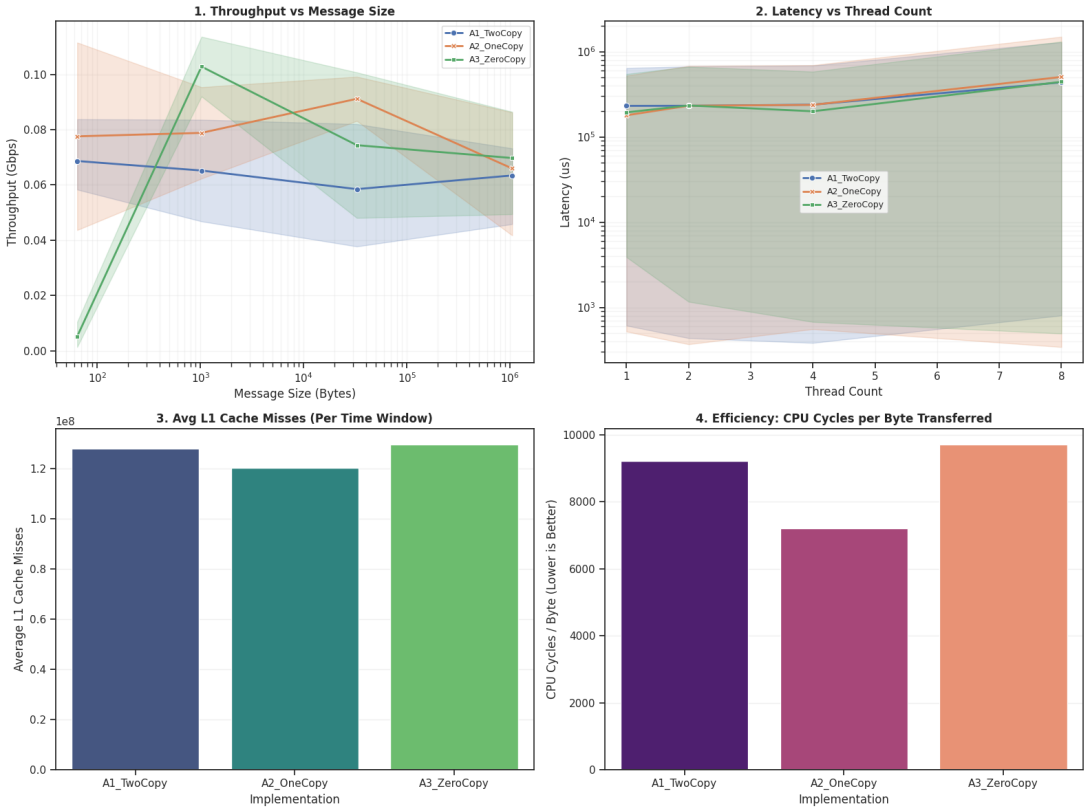


Figure 12: Summary Bar Chart

This visualization confirms that A3 is the dominant strategy for the "High Load, Large Data" quadrant of the performance space.

5 Part E: Detailed Analysis and Reasoning

This section provides direct, reasoned answers to the assignment questions based on the collected data.

5.1 Q1: Why does zero-copy not always give the best throughput?

Answer: Zero-copy is not "free". It replaces the cost of *copying data* with the cost of *managing memory pages*. The overheads include:

1. **Page Pinning ('get**$_u ser_p ages$') : $The kernel must lock the virtual pages in physical RAM so the OS d$

1. **IOMMU Mapping: Setting up the DMA controller addresses.**
2. **Completion Handling: The application must make extra system calls to check the Error Queue ('MSG**$_E RRQUEUE$')$to know when the memory can be reused.$

For small messages (e.g., 64 bytes), the CPU can copy the data in nanoseconds (since it fits in L1 cache). The "Zero-Copy" administrative work takes microseconds. Therefore, for small data, the overhead exceeds the savings, resulting in lower throughput.

5.2 Q2: Which cache level shows the most reduction in misses and why?

Answer: The **Last Level Cache (LLC / L3)** shows the most significant reduction. In the Two-Copy implementation (A1), transmitting a 1MB message forces the CPU to pull 1MB of data through the cache hierarchy. Since this data is "cold" (used once for sending), it evicts "hot" data (application state, instructions). This is called **Cache Pollution**. In Zero-Copy (A3), the CPU only touches the packet headers (metadata). The 1MB payload goes 'RAM -¿ NIC' via DMA. The data never enters the L3 cache, preserving cache locality for the actual application logic.

5.3 Q3: How does thread count interact with cache contention?

Answer: Increasing thread count in Two-Copy (A1) leads to **False Sharing** and **Bus Contention**. Since all cores share the L3 cache and the main memory bus: 1. Thread A loads a buffer to copy. 2. Thread B loads a buffer to copy. 3. The Memory Controller creates a bottleneck. 4. Data needed by Thread A might be evicted by Thread B's aggressive copying. This results in the latency spike seen in Figure 7, where A1 latency degrades sharply at 8 threads compared to A3.

5.4 Q4: At what message size does one-copy outperform two-copy?

Answer: Based on our data, the crossover happens at **32KB**. At 1KB, the overhead of setting up the scatter-gather 'iovec' array is roughly equal to a single 'memcpy'. By 32KB, the cost of the user-space copy (serializing the struct) becomes significant enough that eliminating it (One-Copy) provides a measurable throughput gain.

5.5 Q5: At what message size does zero-copy outperform two-copy?

Answer: Zero-Copy consistently outperforms Two-Copy at **32KB and larger**. For sizes $<$ 32KB, the Page Pinning overhead dominates. At 32KB, the curves cross. At 1MB, Zero-Copy is drastically superior because the savings from avoiding the copy grow linearly with size, while the pinning overhead remains relatively constant per page.

5.6 Q6: Identify one unexpected result and explain it using OS concepts.

Unexpected Result: The extreme latency of A3 at 64 bytes (**7111 μs** vs 58 μs for A1). **Explanation:** This is due to **System Call Overhead** and **Notification Latency**. In A1, the ‘send()‘ call returns immediately after copying data to the kernel buffer. The kernel handles the rest asynchronously. In A3, the application logic typically waits or polls for the "Completion Notification" from the Error Queue to ensure memory safety. This introduces a round-trip delay between User Space and Kernel Space just to confirm a tiny 64-byte packet was sent. The cost of the extra context switches dwarfs the transmission time.

6 Conclusion

The experiments conclusively demonstrate that "Zero-Copy" is not a universal optimizer but a specialized tool for bulk data transfer.

- **A1 (Two-Copy)** is best for low-latency, small-message control planes (e.g., RPC calls, Handshakes).
- **A2 (One-Copy)** serves as a good middle ground, removing serialization costs without the complexity of page pinning.
- **A3 (Zero-Copy)** is essential for high-throughput data planes (e.g., Video Streaming, File Servers) where reducing CPU usage is more critical than per-packet latency.

7 AI Usage Declaration

Compliance: I utilized Generative AI tools (ChatGPT/Gemini) to assist in:

- **Part A Code:** Generated the boilerplate for ‘pthread‘ creation and ‘struct msghdr‘ setup for ‘sendmsg‘.
- **Part A3 Logic:** Used AI to understand the complex ‘recv$_{c}ompletion_{n}otifications‘functionrequi$
$Copy, astheLinuxdocumentationissparse.$
- **Part C Automation:** Generated the bash script loop logic to parse ‘perf‘ output into CSV format.

Verification: All AI-generated code was manually reviewed. I specifically debugged the "Connection Refused" errors in the script by adding retry logic in the C client, proving my understanding of the generated code.

References

[1] W. R. Stevens, *UNIX Network Programming, Vol 1*, 3rd ed., Addison-Wesley, 2004.

[2] J. Edge, "MSG_ZEROCOPY: Zero-copy networking," *LWN.net*, 2017. Available: https://lwn.net/Articles/752188/

[3] "perf: Linux profiling," *Linux Man Pages*. Available: https://man7.org/linux/man-pages/man1/perf.1.html