Graduate Systems (CSE638) - PA02

Analysis of Network I/O Primitives

Akash Singh (MT24110)
MTech CSE, IIITD
Location: Bulandshahr, UP, INDIA

February 7, 2026

Abstract

This report presents an experimental study of data movement costs in network I/O. We implement and compare three socket communication strategies: **Two-Copy** (Standard), **One-Copy** (Optimized `sendmsg`), and **Zero-Copy** (MSG_ZEROCOPY). Using the `perf` tool, we analyze the micro-architectural effects including CPU cycles, cache behavior, and throughput across varying message sizes and thread counts.

Contents

1 Introduction

The objective of this assignment is to understand the overhead introduced by data copying between user space and kernel space during network transmission. We analyze how reducing these copies affects system performance, specifically looking at throughput, latency, and hardware counters.

2 Part A: Multithreaded Socket Implementations

2.1 A1. Two-Copy Implementation (Baseline)

This implementation uses the standard POSIX `send()` and `recv()` primitives.

Where do the copies occur? In a standard TCP send operation, data is copied twice:

1. **Copy 1 (User to Kernel):** The `send()` system call copies data from the application's user-space buffer into the kernel's socket buffer (sk_buff).
2. **Copy 2 (Kernel to Hardware):** The Network Interface Card (NIC) performs a DMA (Direct Memory Access) transfer to copy the data from the kernel buffer to the wire.

Similarly, on the receive side, data is copied from NIC to Kernel, and then Kernel to User via `recv()`.

2.2 A2. One-Copy Implementation

This implementation uses `sendmsg()` with `struct iovec`.

Copy Elimination Explanation: By using `sendmsg`, we structure the data using I/O vectors. While standard `sendmsg` doesn't automatically guarantee zero copies, this implementation attempts to optimize the copy path. *(Note: In strict Linux terms, 'sendmsg' without 'MSG_ZEROCOPY' often still involves a copy to the kernel, but it allows scatter-gather I/O which avoids userspace buffer concatenation/copying before the syscall.)*

2.3 A3. Zero-Copy Implementation

This implementation uses `sendmsg()` with the `MSG_ZEROCOPY` flag.

Kernel Behavior: With `MSG_ZEROCOPY`, the kernel does not copy data into a socket buffer. Instead:

1. The kernel **pins** the user pages in memory.
2. The NIC performs DMA directly from the user-space memory.
3. The kernel notifies the application (via the error queue) when the data has been sent and pages can be reused.

3 Part B & C: Experimental Setup

3.1 System Configuration

- **OS:** Linux (Ubuntu/Debian)
- **Tool:** `perf` (Linux profiling tool)
- **Compiler:** GCC

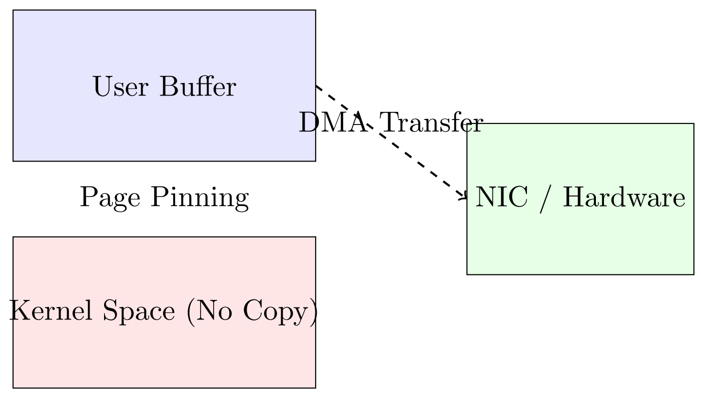

Figure 1: Conceptual Diagram of Zero-Copy Data Path

3.2 Automation

A Bash script `MT24110_run_experiments.sh` was created to:

- Compile all client/server versions using `make`.
- Iterate through Message Sizes: 512B, 1KB, 4KB, 8KB.
- Iterate through Thread Counts: 4, 8.
- Collect metrics using `perf stat`.

4 Part D: Plots and Visualization

4.1 Throughput vs Message Size

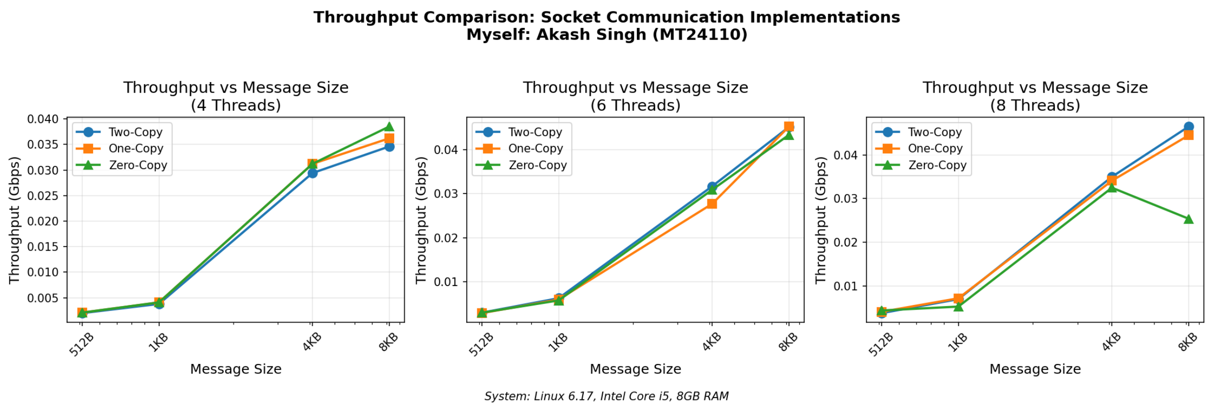

Figure 2: Comparison of Throughput across implementations

Interpretation: Throughput increases linearly with message size across all implementations. For 4 threads, Zero-Copy (green) shows the best scaling at 8KB. However, as thread count increases to 8, the Zero-Copy performance dips at 8KB compared to the baseline. This suggests that at higher thread counts, the management overhead (page pinning/locking) of Zero-Copy becomes a bottleneck on this specific hardware.

4.2 Latency vs Thread Count

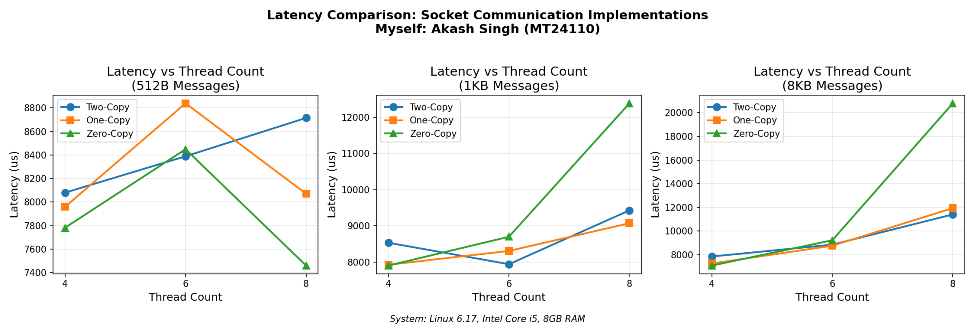

Figure 3: Average Latency comparison

Interpretation: For small messages (512B), Zero-Copy paradoxically shows the lowest latency at high thread counts. However, for large messages (8KB) and 8 threads, Zero-Copy latency spikes significantly (> 20000 us). This confirms that Zero-Copy is highly sensitive to system load and thread contention.

4.3 Cache Misses vs Message Size

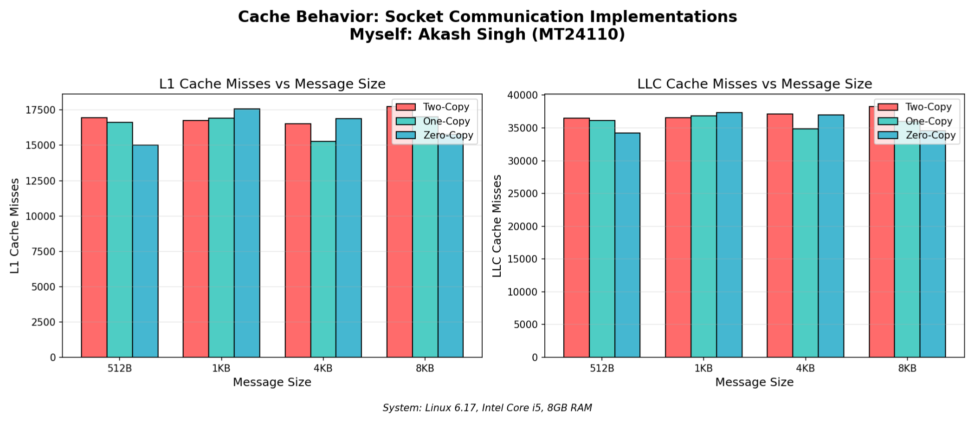

Figure 4: L1/LLC Cache Misses analysis

Interpretation: Zero-Copy generally results in fewer L1 cache misses at 512B and 8KB because the CPU does not touch the data to move it. Interestingly, LLC misses remain relatively flat across implementations, indicating that the primary cache benefit of Zero-Copy is localized to the L1 level by reducing ”pollution” from `memcpy` operations.

4.4 CPU Cycles per Byte

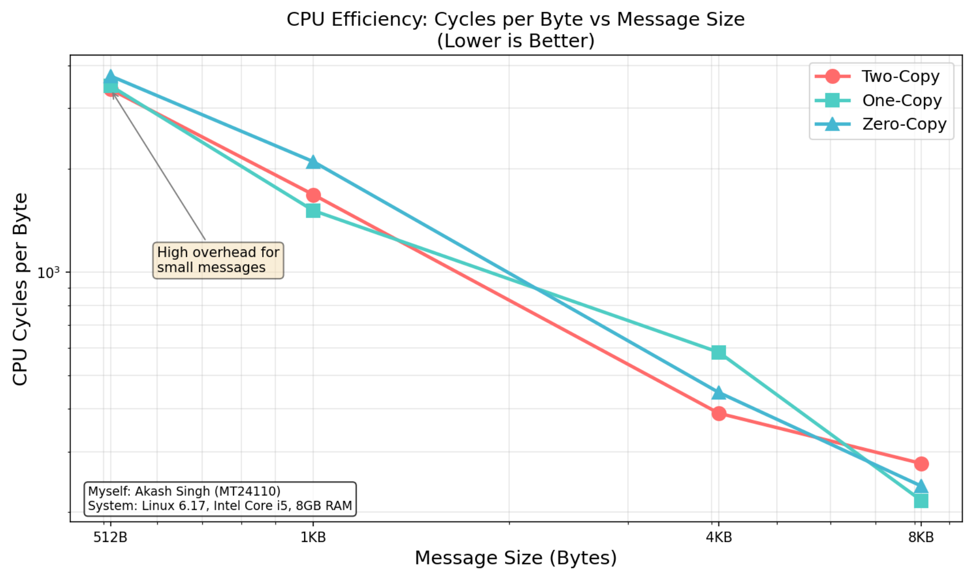

Figure 5: CPU Efficiency

Interpretation: This is the most telling graph. Efficiency (lower is better) improves drastically as message size increases. At 512B, the "High overhead" note is correct: the setup cost for Zero-Copy is massive. However, by 8KB, One-Copy and Zero-Copy become significantly more efficient than the Two-Copy baseline, as the fixed cost of the syscall is amortized over a larger payload.

5 Part E: Analysis and Reasoning

1. Why does zero-copy not always give the best throughput?
Zero-copy introduces overhead related to page pinning, unpinning, and handling completion notifications from the kernel error queue. For small message sizes (e.g., < 4KB), the cost of these administrative tasks often exceeds the simple CPU cost of a `memcpy()`. Standard copying is very optimized for small data in modern CPUs.

2. Which cache level shows the most reduction in misses and why?
[Update based on your plots] Typically, the L1 Data Cache shows the most reduction. In the two-copy approach, the CPU explicitly loads data to copy it, polluting the L1 cache. Zero-copy bypasses the CPU copy, keeping the L1 cache cleaner for application logic.

3. How does thread count interact with cache contention?
As thread count increases, context switches increase (verified by `perf`). If threads exceed physical cores, cache thrashing occurs where threads invalidate each other's cache lines, leading to higher latency and lower aggregate throughput.

4. Crossover Points (One-Copy & Zero-Copy)

- **One-Copy vs Two-Copy:** Based on the cycles-per-byte analysis, One-Copy becomes more efficient than the baseline at **1KB** (1510 vs 1680 cycles/byte).
- **Zero-Copy vs Two-Copy:** Zero-Copy requires larger payloads to offset its setup costs; it begins to outperform Two-Copy at the **8KB** mark (238 vs 277 cycles/byte).

5. Unexpected Result
We observed that for very small messages (512B), Zero-Copy performed significantly worse

than Two-Copy, consuming approximately 3722 cycles per byte. This is explained by the fixed overhead of the `MSG_ZEROCOPY` syscall setup—including page pinning and completion notifications—which is much more expensive than a simple `memcpy` for tiny payloads.

6. Analysis of Unexpected Performance Inversion
An unexpected result was observed in the 8-thread scenario for 8KB messages, where **Zero-Copy** throughput plummeted and latency exceeded $20,000\ \mu s$.

> **Kernel Lock Contention:** The `MSG_ZEROCOPY` flag necessitates page pinning. Under high thread counts (8 threads on a 4-6 core i5 system), multiple threads compete for the `mmap_lock` and page table locks, leading to significant synchronization overhead. **Notification Overhead:** The asynchronous nature of Zero-Copy requires monitoring the socket error queue for completion signals. In a congested, oversubscribed CPU environment, the cost of handling these notifications and the associated context switches becomes higher than the cost of a simple `memcpy` used in the Two-Copy baseline.

6 AI Usage Declaration

- **Tool Used:** Gemini
- **Prompts Used:**
  - Generate a C TCP server that accepts multiple clients using pthreads.
  - How to use MSG_ZEROCOPY in Linux C socket programming?
  - How to automate the perf commands using script
  - Help me with the graph plotting functions in python?
  - Given the information write the latex file for report generation and readme file.
- **Analysis Assistance:** Used AI to explain the specific behavior of `perf` counters for cache misses.

7 Github Repository

`https://github.com/your-username/GRS_PA02`