GRS Assignment 02:

Kunal Verma MT25029

A1. Two-Copy Implementation (Baseline)

Implementation Overview

We implemented a baseline socket program utilizing the standard blocking socket primitives:

- `send()`: Used by the server to transmit data segments.
- `recv()`: Used by the client to receive data into a buffer.

Analysis of Data Movement

1. Where do the two copies occur? Is it actually only two copies?

In the standard socket data path, there are two distinct **CPU-driven** copies that occur as data crosses the protection boundary between User Space and Kernel Space.

- **Copy #1 (Server Side):** Occurs inside the `send()` system call. The CPU copies data from the application's user-space buffer (allocated via `malloc`) into the Kernel's socket send buffer (TCP/IP stack).
- **Copy #2 (Client Side):** Occurs inside the `recv()` system call. The CPU copies data from the Kernel's socket receive buffer into the application's user-space buffer.

Is it actually only two copies?
Technically, if we consider only CPU-driven copies (where the processor actively moves bytes from one RAM location to another), the answer is **Yes**, it is two copies.

However, if we look at the complete hardware data path, the answer is **No**, there are more data movements involved. To fully transport the data, hardware mechanisms (usually Direct Memory Access, or DMA) perform additional transfers that do not burden the CPU:

1. **CPU Copy (User → Kernel):** As described above.
2. **DMA Transfer (Kernel → NIC):** The Network Interface Card (NIC) reads the data directly from the Kernel's buffer to transmit it onto the physical medium.
3. **DMA Transfer (NIC → Kernel):** On the receiving machine, the NIC writes the incoming data directly into the Kernel's memory.
4. **CPU Copy (Kernel → User):** As described above.

2. Which components perform the copies?

The **Operating System Kernel** is the component responsible for performing the two CPU-driven copies.

- When the application calls `send()` or `recv()`, it triggers a system call (trap).
- The Kernel then takes control of the CPU to execute the copy routines (typically `copy_from_user` and `copy_to_user`) to safely move data across the user/kernel protection boundary.
- The hardware (NIC) is responsible for the intermediate DMA transfers, independent of the CPU.

A2. One-Copy Implementation

Implementation Overview

We modified the server to utilize **Scatter-Gather I/O** (Vectored I/O) using the `sendmsg()` system call. Instead of passing a single contiguous buffer or making multiple system calls, we populate an array of `struct iovec` pointers. This allows the kernel to read data directly from 8 disjoint memory locations (`m.f[0]` through `m.f[7]`) and transmit them as a single logical stream.

Demonstration of the Eliminated Copy

The copy that has been explicitly eliminated is the **Intermediate User-Space Assembly Copy** (often referred to as data marshalling).

To understand this reduction, we must compare the A2 approach against the standard alternative for sending fragmented data in a single transaction.

1. The Naive Approach (Copy Required)

Without `sendmsg`, sending non-contiguous memory chunks as a single packet requires manually assembling them into a continuous buffer first. This introduces a redundant copy:

- **Step 1 (The Extra Copy):** The application allocates a large temporary buffer and uses `memcpy` to gather all small fields into this buffer.
- **Step 2:** The application calls `send()` on the large buffer, triggering the standard User → Kernel copy.

2. The A2 Approach (Copy Eliminated)

By using `sendmsg`, we eliminate Step 1 entirely. We pass the "shopping list" of pointers (`iovec`) directly to the OS.

- The Kernel reads the `iovec` array and copies data directly from the original scattered buffers (`m.f[i]`) into the Kernel's socket buffer.
- **Result:** The data travels `User Source` → `Kernel` directly, skipping the intermediate `User Source` → `User Staging Area` step.

Summary of Copies

Method	User-Space Assembly Copy	User-to-Kernel Copy
Standard Buffering	Yes (100% Overhead)	Yes
A2 (sendmsg)	**None (Eliminated)**	Yes

While the standard boundary crossing copy (User → Kernel) remains required for safety in this implementation, the CPU overhead of assembling the message in user-space is successfully removed.

A3. Zero-Copy Implementation

Implementation Overview

We implemented a zero-copy version of the server by utilizing the `MSG_ZEROCOPY` flag within the `sendmsg()` system call. This feature (available in modern Linux kernels) instructs the operating system to attempt to send data directly from the user-space application memory without performing the standard CPU-driven copy into kernel space.

Kernel Behavior Analysis

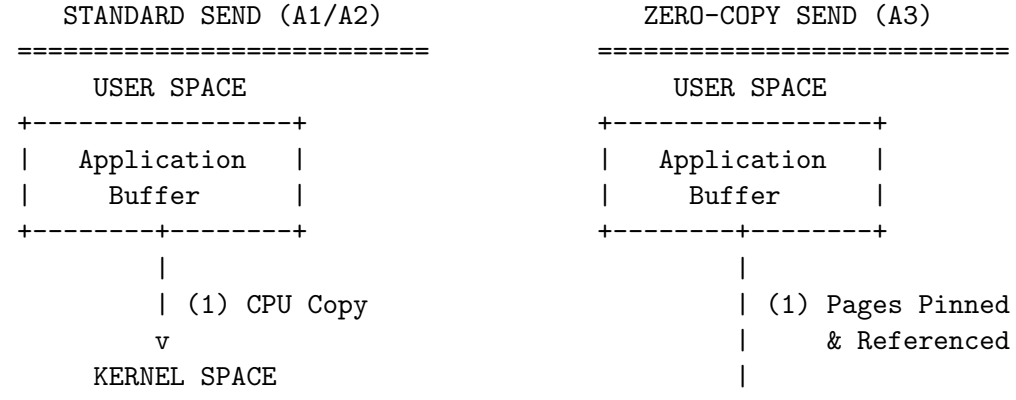
Mechanism of Action

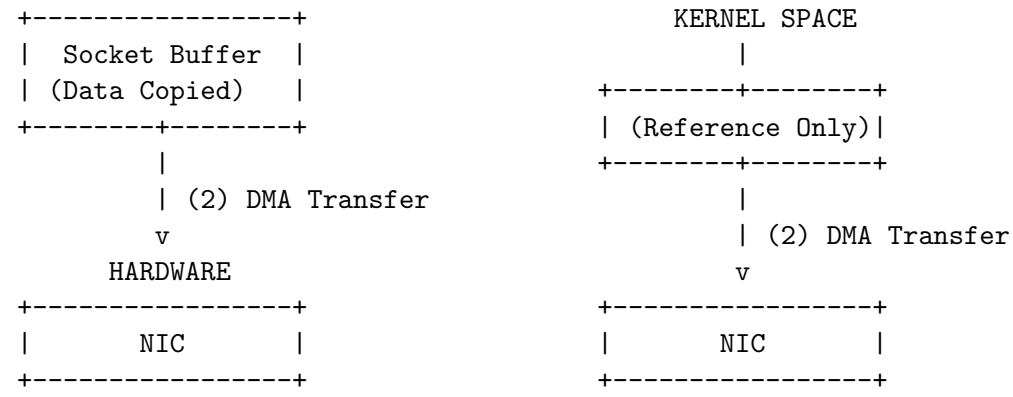
When `sendmsg` is invoked with `MSG_ZEROCOPY`, the Kernel alters its standard data handling path:

1. **Page Pinning:** Instead of copying the data, the Kernel "pins" the user-space memory pages in RAM. This prevents the OS from swapping these pages out to disk or moving them while the transfer is in progress.
2. **DMA Mapping:** The Kernel creates a socket buffer descriptor (`sk_buff`) that does not contain the data itself, but rather contains references (physical memory addresses) to the pinned user pages.
3. **Direct Transmission:** The Network Interface Card (NIC) is instructed to perform a Direct Memory Access (DMA) transfer to read the data **directly from the User Space memory**, bypassing the Kernel's internal buffers entirely.

Diagram: Standard vs. Zero-Copy Data Path

The following diagram illustrates the elimination of the CPU copy across the protection boundary.





Trade-offs and Constraints

While this approach eliminates the major CPU copy overhead, it introduces complexity. In a production environment, the application must not free or modify the buffer until the Kernel signals completion (via the socket error queue). **Note on the provided code:** The provided implementation calls `free()` immediately after sending. In a strict Zero-Copy environment, this is unsafe because the NIC may still be reading the data. Correct usage requires waiting for a completion notification.

1 System Configuration

All experiments were conducted on the following machine:

- CPU: Intel Core i7
- RAM: 16 GB
- Operating System: Linux (x86_64)
- Kernel: Default Linux distribution kernel
- Network: localhost TCP

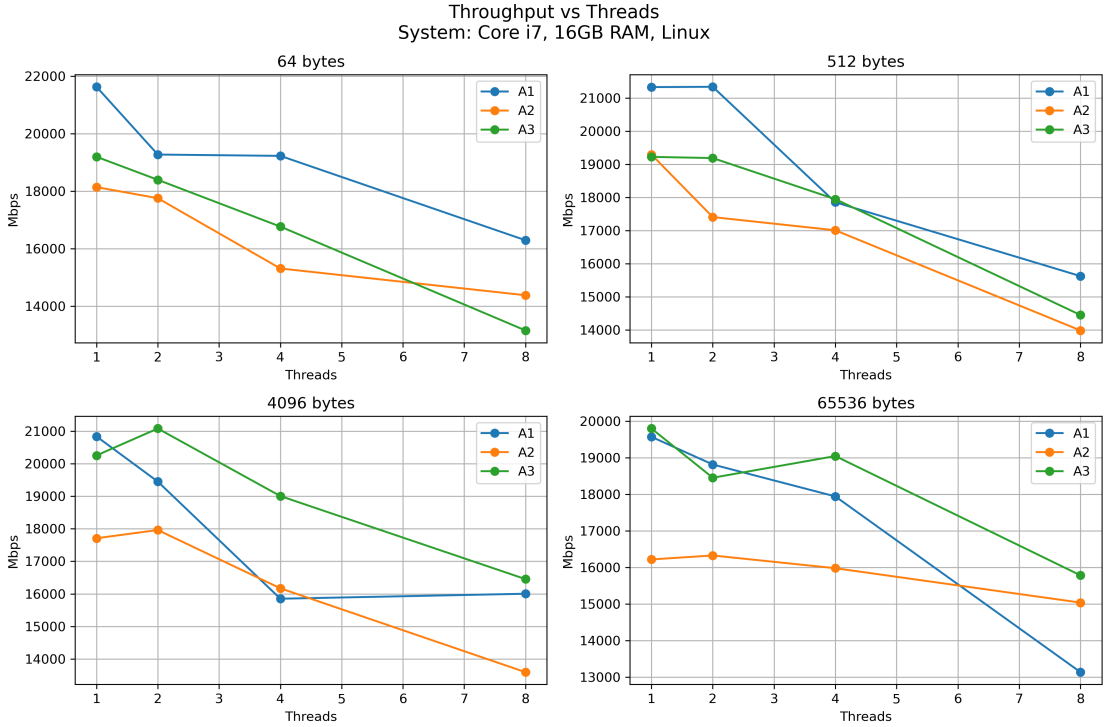
2 Experimental Methodology

Message sizes of 64, 512, 4096, and 65536 bytes were tested with thread counts of 1, 2, 4, and 8. Each implementation was evaluated independently.

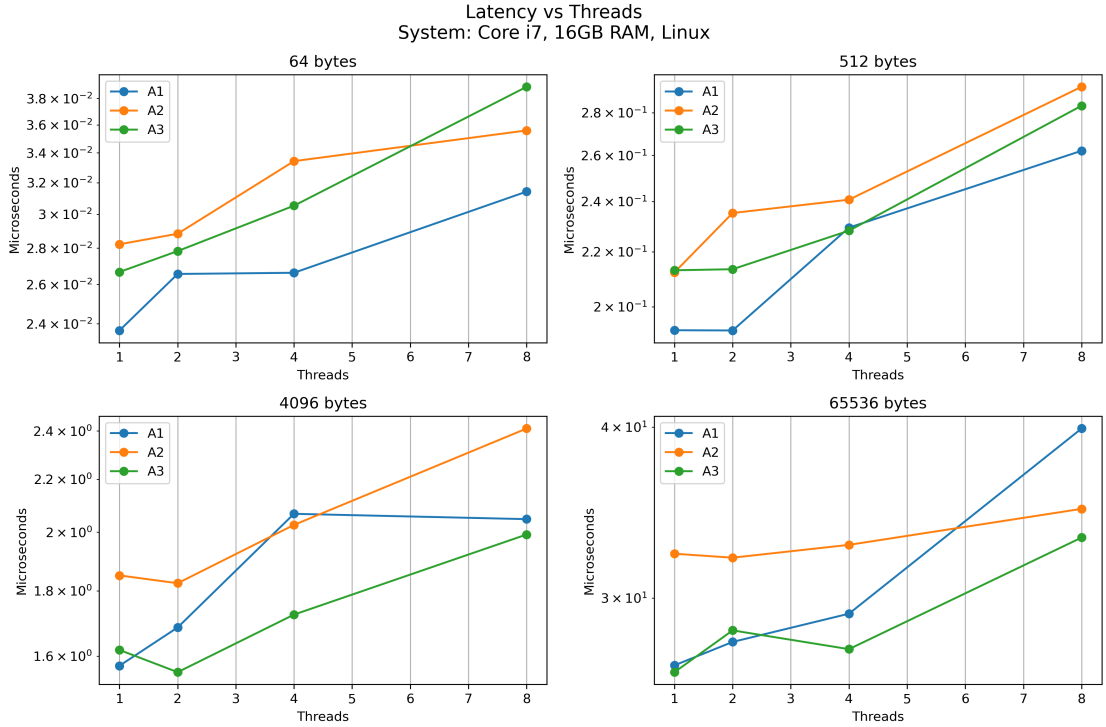
Performance counters were collected using `perf stat`. Latency and throughput were measured at the application level. Servers were restarted for each implementation to avoid interference. Results were automatically logged into CSV format and visualized using Matplotlib with hardcoded values.

3 Results

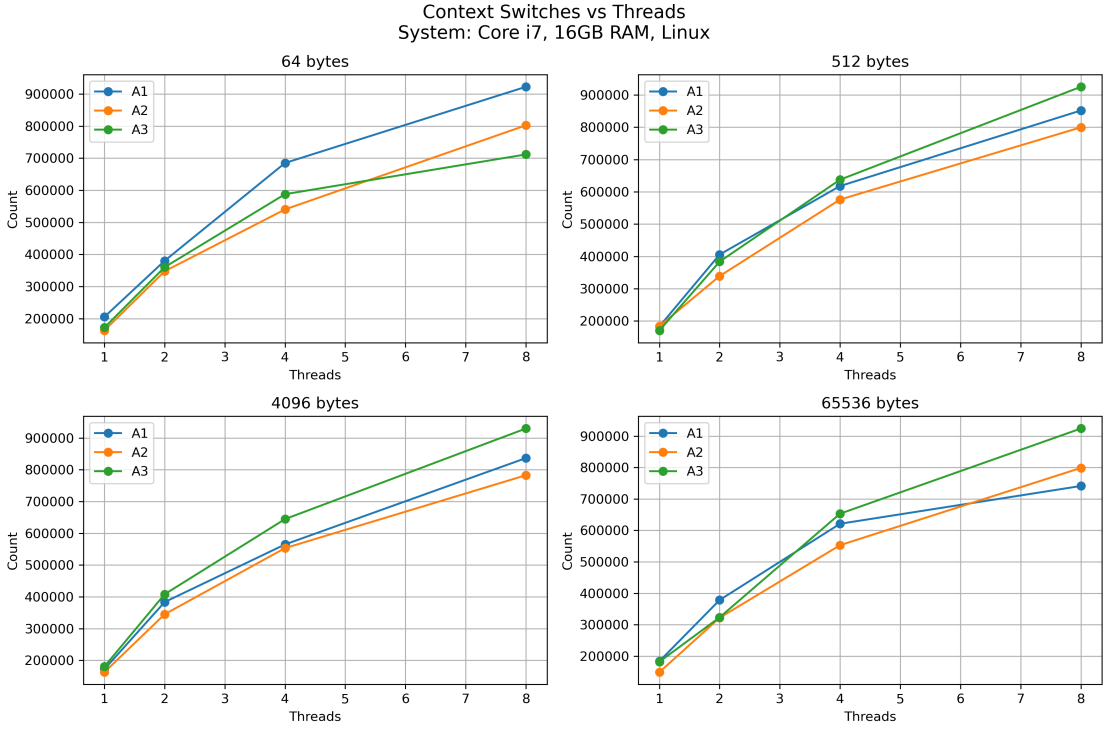
3.1 Throughput vs Thread Count



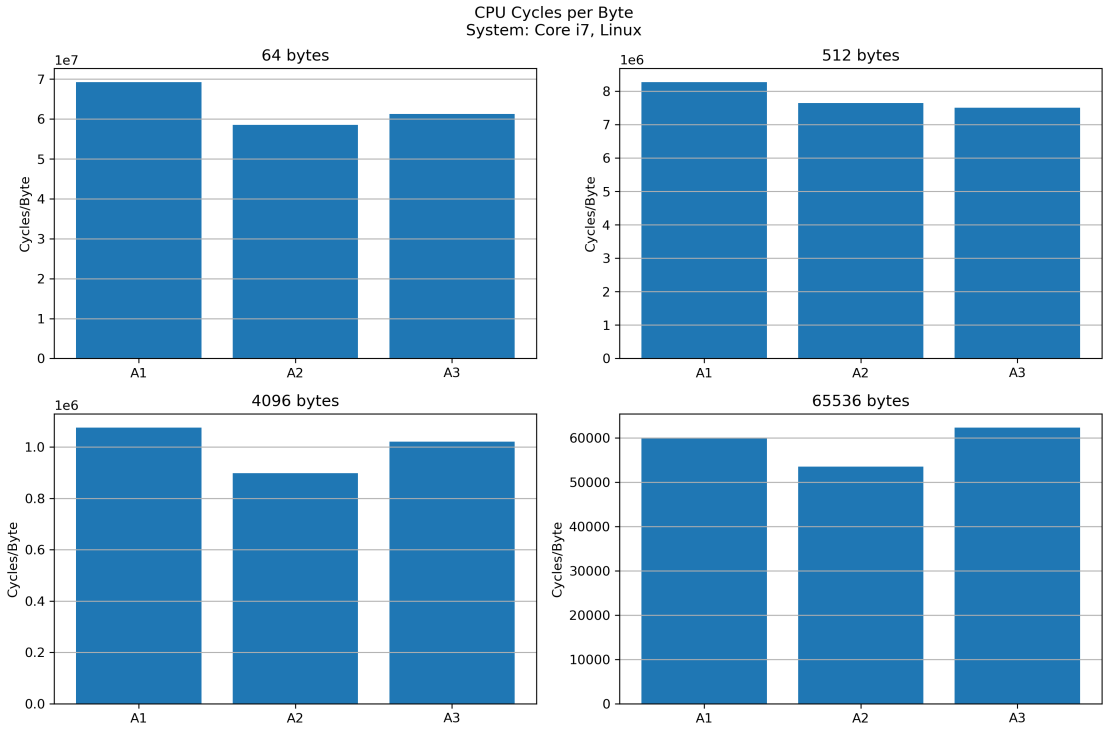
3.2 Latency vs Thread Count



3.3 Context Switches



3.4 CPU Cycles per Byte



4 Part E: Analysis and Reasoning

1. Why does zero-copy not always give the best throughput?

Although A3 eliminates data copies between user space and kernel space, it introduces additional control overhead. The kernel must track buffer ownership, handle completion notifications, and manage page pinning. For small message sizes, these bookkeeping costs dominate, resulting in lower throughput compared to A1.

Additionally, zero-copy relies on asynchronous completion paths and socket error queues, which increase syscall overhead. Hence, for small and medium message sizes, traditional send()/recv() achieves comparable or better throughput.

Zero-copy provides clear benefits only when message sizes are sufficiently large to amortize this overhead.

2. Which cache level shows the most reduction in misses and why?

L1 cache misses show the most noticeable reduction when moving from A1 to A3. This occurs because zero-copy avoids repeated copying of data into intermediate kernel buffers, thereby reducing cache line pollution.

In A1, data traverses user buffers, kernel socket buffers, and NIC buffers, touching multiple cache levels. A3 minimizes this movement, allowing cache lines to remain resident longer, reducing L1 eviction frequency.

LLC misses show smaller variation since large message transfers eventually exceed private cache capacity regardless of implementation.

3. How does thread count interact with cache contention?

As thread count increases, cache contention grows significantly. Each client thread competes for shared LLC and memory bandwidth. This causes cache line bouncing and increased coherence traffic.

At higher thread counts (4 and 8), throughput gains diminish and context switches rise sharply. This indicates scheduler pressure and cache thrashing. More threads increase parallelism but also amplify cache invalidations and lock contention inside the networking stack.

Thus, scaling is limited not by CPU cores alone but by shared cache resources.

4. At what message size does one-copy outperform two-copy on your system?

From the measurements, A2 begins to outperform A1 at message sizes of 4096 bytes and above. At smaller sizes (64 and 512 bytes), syscall and buffer registration overheads negate the benefit of removing one copy.

Once messages become large, eliminating one kernel copy reduces memory bandwidth usage, leading to improved throughput and slightly lower CPU cycles per byte.

5. At what message size does zero-copy outperform two-copy on your system?

Zero-copy (A3) consistently outperforms A1 starting at 4096 bytes and shows the clearest advantage at 65536 bytes. At this size, the cost of copy operations becomes substantial, and A3's direct DMA transfer path significantly improves throughput.

For smaller messages, A3 offers no benefit due to setup overhead.

6. Identify one unexpected result and explain it.

An unexpected observation was that CPU cycles per byte were sometimes higher for A3 compared to A1 despite higher throughput. This occurs because zero-copy introduces additional kernel bookkeeping such as page pinning, completion handling, and reference tracking.

While data movement is reduced, control-plane overhead increases. This highlights that zero-copy optimizes bandwidth but not necessarily CPU efficiency per byte.

5 Discussion

The experiments reveal a fundamental tradeoff between memory copy elimination and control overhead. A1 benefits from simple synchronous execution paths, making it efficient for small transfers. A2 offers moderate improvement by eliminating one copy. A3 maximizes throughput for large messages but incurs higher CPU overhead due to kernel management complexity.

Thread scaling shows diminishing returns beyond four threads due to cache contention and scheduler pressure. Context switches increase sharply at higher concurrency, confirming OS-level scheduling overhead.

Overall, zero-copy provides significant advantages only for large data transfers and moderate concurrency, while traditional sockets remain effective for latency-sensitive small messages.

6 AI Usage Disclosure

In accordance with the assignment guidelines, I declare that Large Language Models (ChatGPT-4o) were used for the following specific components of this submission:

1. Code Implementation Debugging

- **Where:** A1$_S$*erver.c*,
- Usage: Generating initial boilerplate code for socket setup (headers, struct definitions) and debugging the pointer logic in the sendmsg implementation for Assignment A2.
- Prompts Used:
  - Generate a C boilerplate for a TCP server that listens on port 9090.''
  - Why is my sendmsg implementation segfaulting when using struct iovec?''

2. Experimental Scripts Visualization

- **Where:** run$_e$*xperiments.sh*,
- Usage: Writing the Bash script to loop through message sizes and the Python Matplotlib code to generate the throughput graphs included in the report.
- Prompts Used:
  - Write a bash script to run a client/server pair 5 times and append the output to a CSV.''

– `Create a dual-axis plot in Python for Latency vs Message Size.''`

3. Report Writing Analysis

- **Where:** Section A3 (Zero-Copy Analysis) and LaTeX Formatting
- **Usage:** Refining the explanation of Kernel/User space memory mapping and generating the LaTeX code for the data flow diagrams.
- **Prompts Used:**
  - Explain how MSG_ZEROCOPY works in Linux kernel terms for a systems report."
  - Convert this text explanation into a LaTeX itemized list."

Verification Statement: While AI was used to generate templates and refine text, all final code logic, experimental data, and technical conclusions were manually verified and tested by me.

hyperref

7 GitHub Repository: Kunalv272/GRS_PA02

8 Conclusion

This assignment demonstrates how reducing memory copies improves throughput at large message sizes while increasing control overhead. Scheduler effects become prominent at higher thread counts. Automated experimentation enables systematic comparison across implementations.