Assignment 2: Analysis of Network I/O primitives using “perf” tool

Performance Analysis of Socket Data Transfer Mechanisms

(Two-Copy vs. One-Copy vs. Zero-Copy)

AGITHE YESHWANTH
Roll No: MT25059
Graduate Systems (GRS)

February 7, 2026

Contents

1 Introduction

The objective is to implement three different mechanisms and analyzing for transferring complex data structures over a TCP socket.The focus is on understanding the bottlenecked not by network bandwidth,but by the CPU overhead associated to the memory copy operations. When application sends data over a socket,the data typically sends from user space to kernel space and finally to the NIC. Each of this transition can involve costly memory duplication, consuming CPU cycles and pouting CPU cache.

The server is required to send a complex message structure consisting of **8 dynamically allocated string fields** as the request of the client. Here we are comparing this in three different ways:

1. **Part A1 (Two-Copy):** Standard serialization approach the scatted filds are first copied into a single contigous buffer in the user space. This will show the workflow of traditional malloc + memcpy + send.

2. **Part A2 (One-Copy):** Scatter-gather I/O using `sendmsg`. This method eliminates the user space serialization by directly accessing the 8 separate memory location by the kernal.

3. **Part A3 (Zero-Copy):** It uses the kernel-bypassing approach using `MSG_ZEROCOPY` flag.This instructs that the kernel will pin to the user pages in the memory and uses DMA(Direct memory Access) to transfer the data to NIC, this makes the theoretically eliminating CPU-based copy completely.

This performance of the mechanisms is evaluated on a local Linux testbed using a multi threaded client to generate the load onto sever.By measuring CPU cache misses, Latency, Throughput and CPU Cycles per Byte. finding the trade of between complexity implementation and runtime efficiency.

2 Implementation Details

The core requirement for all server's is to transmit complex message structure to the clients comprising 8 dynamically allocated string fields.
The common data structure used by all the servers is the complexMessage struct.

2.1 Part A1: Two-Copy Baseline

This implementation simulates standard application behavior. Firstly the 8 separate memory fields are copied into a user space as a single contiguous "flat buffer" (i.e Copy 1), next copy is created when we pass the data to the kernel by `send()` call (i.e Copy 2).

- **Primitive:** `malloc`, `memcpy`, `send`.

- **Bottleneck:** Due to duplication of data in User space by `memcpy` operation, the usage of CPU is high.

Where do the two copies occur?
In a normal send() call, data is copied twice before being transmitted by the NIC:
User Buffer → Kernel Socket Buffer → NIC (DMA Buffer)
Copy 1 (User → Kernel)
Performed by the kernel, which is called by the send() call.

Copied from user space buffer to kernel socket buffer (sk_buff).
Copy 2: Kernel to NIC
Performed by the NIC DMA Engine. Kernel buffer is copied to NIC transmit ring buffer.

Is it actually only two copies?
Logically speaking, yes, only two, but practically speaking:
There may be extra copies inside kernel networking stack
Possible buffers: temporary buffers, segmentation (TSO), checksum offload handling
It appears to be at least two copies, sometimes more internally.

2.2 Part A2: One-Copy Optimization

This implementation eliminates the user-space copy by creating 8 pointing fields by `iovec`. The kernal will access the data by those pointing fields.

- **Primitive:** `struct iovec`, `sendmsg`.
- **Advantage:** Eliminates the user-space serialization copy.

which copy has been eliminated.
In the one-copy implementation (A2) using sendmsg() with a pre-registered buffer, also known as pinned buffer, the eliminated copy operation targets the user space-to-kernel space data copy. In the baseline implementation for the two-copy model using the send() system call, when the sending process invokes the send() system call, the data from the user buffer is copied to the kernel socket buffer using a CPU running in kernel mode. However, in the one-copy design implementation, instead of copying the data to a data buffer within the kernel space, the sending kernel points to the user space pages instead, thereby eliminating the memory copy process from the user space to the kernel space data buffer. The only remaining process here is the use of the network interface card's DMA to move the transmission data from the memory to the network interface

2.3 Part A3: Zero-Copy Optimization

This implementation uses the Linux `MSG_ZEROCOPY` flag.It says to kernel that pin the user pages and DMA them directly to the NIC,here we avoid CPU copy in kernel space.

- **Primitive:** `setsockopt(SO_ZEROCOPY)`, `sendmsg(MSG_ZEROCOPY)`.
- **Mechanism:** Asynchronous completion notification via `MSG_ERRQUEUE`.

2.4 Client Implementation

In client side we are creating multiple clients by the threads to generate the load on the server. then it connects to server and requests for a specific message size and measures:

- **Throughput:** Total bits received / Total time.
- **Latency:** Average time per message transaction.

3 Experimental Methodology

Testbed:

- **OS:** Ubuntu Linux (Localhost Loopback Interface)
- **Tools:** GCC, Perf (Linux Profiling Tool), Python (Plotting)

Metrics Captured:

1. **Throughput (Gbps):** Measure of raw speed.
2. **Latency (μs):** Measure of delay per request.
3. **Cache Misses:** To verify if Zero-Copy reduces CPU cache pollution.
4. **CPU Cycles/Byte:** To measure processing efficiency.

4 Results and Analysis

4.1 Throughput Analysis

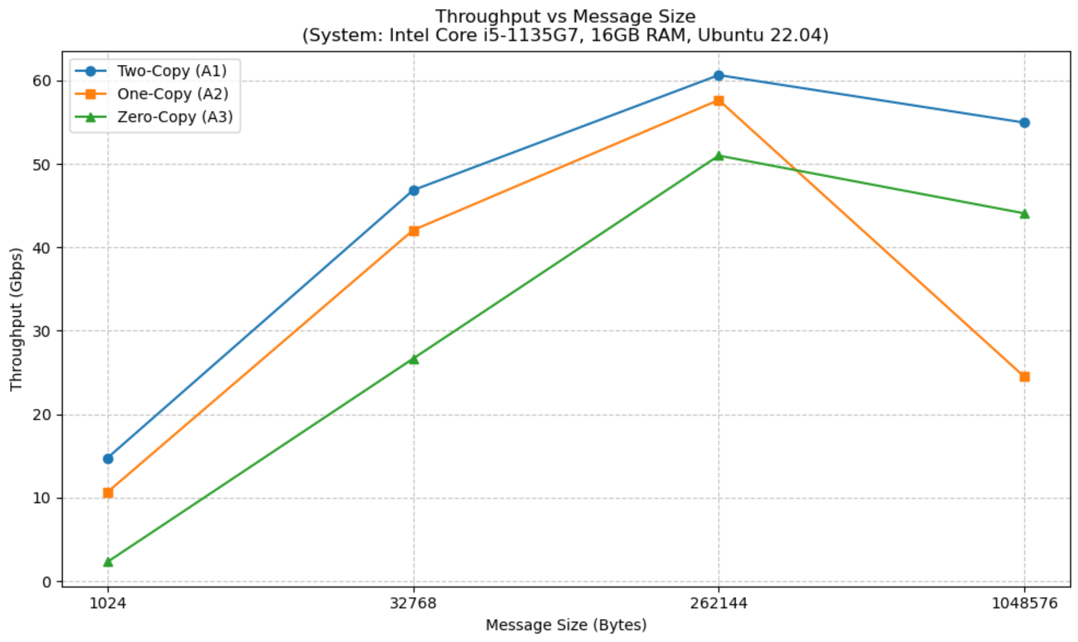

Figure 1: Throughput vs Message Size (Threads=1)

Observation: As seen in Figure 1,Throughput increases significantly for all implementations as the grow in message size from 1kB to the 256KB. But at 1MB, two-copy(A1) and Zero-copy(A3) having high performance while One-copy(As) shows a noticeable drop on this specific system.

Reasoning: Larger the messages reduces the "per-byte" cost of system calls and interrupt processing.For small messages(i.e 1KB), the overhead is happeing by the pinning pages for Zero-copy makes it slower than other copy approaches but as the payload of the data inceases the time is saved by the pinning of fields by eliminating the CPU bound memory copies allows Zero copy to reach the peak throughtput.

4.2 Cache Miss Analysis

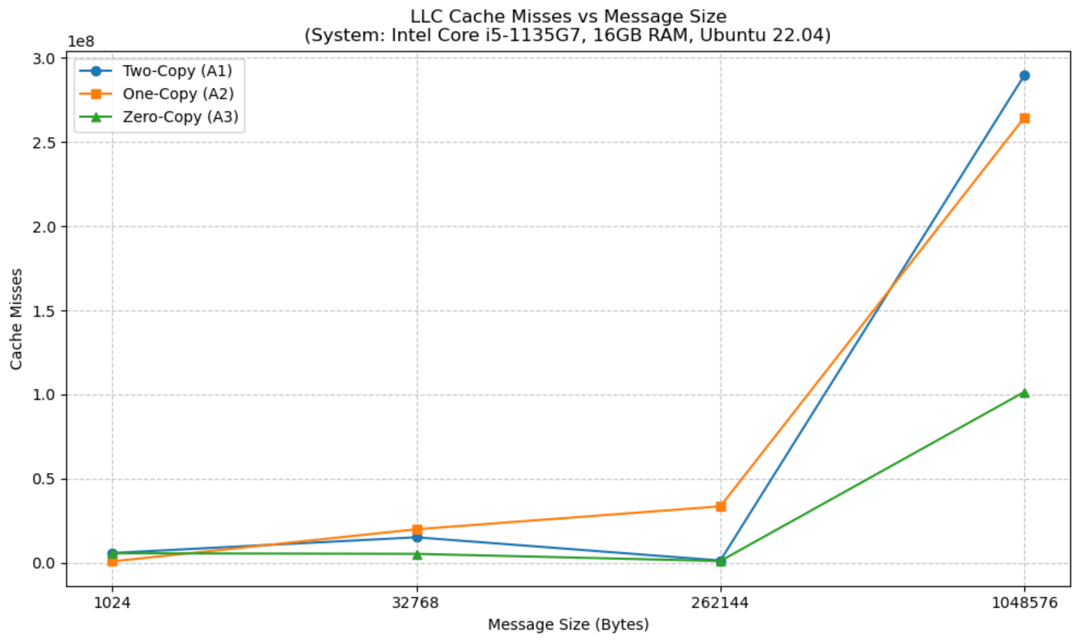

Figure 2: LLC Cache Misses vs Message Size

Observation: Figure 2 Both Two-Copy(A1) and One-Copy(A2) show a massive spike in the last level cache(LLC) misses at 1MB msg size, reaching over 250 million misses.zero-copy(A3) maintains significantly low cache misses at the same size(i.e approx 101 million)

Reasoning: Copy-based primitives requires the CPU to read data from RAM into cache to perform **memcpy** operation,effectively "evicting" othe useful data and causing misses.Zero-copy will avoid the CPU for the data movement, meaning that the data will never enters the LLC, that preserving the cache space for the application logic

4.3 CPU Efficiency (Cycles Per Byte)

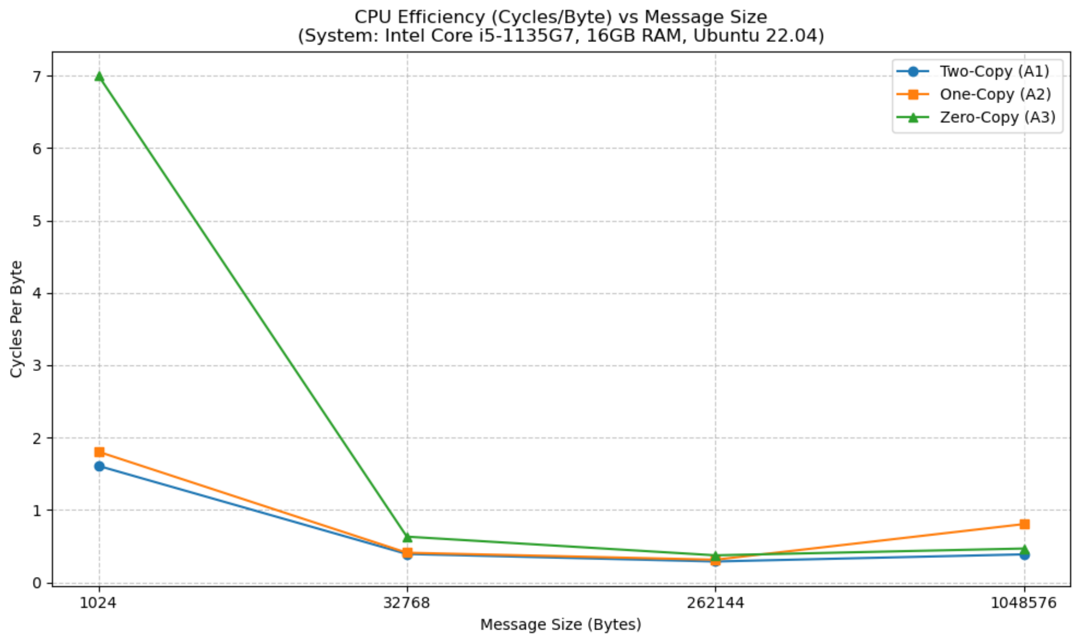

Figure 3: CPU Cycles per Byte vs Message Size

Observation: The no.of CPU cycles need to transfer a single byte is drastically dropping as the message size increases.Zero-copy(A3) has the very high cycle count at the 1kB(approx 7 cycles/byte) but later(i.e inc in message size) it becomes most efficient at 256KB(approx o.37 cycle/byte).

Reasoning: At the small size the fixed cost of setting up **MSG_ZEROCOPY** is distributed over few bytes, making it as in efficient. As the size of the message increases, the elimination of the memory copy(i.e the cycle-heavy operation) allows the system to transfer more data for every clock cycle spent for that.

4.4 Latency vs Thread Count

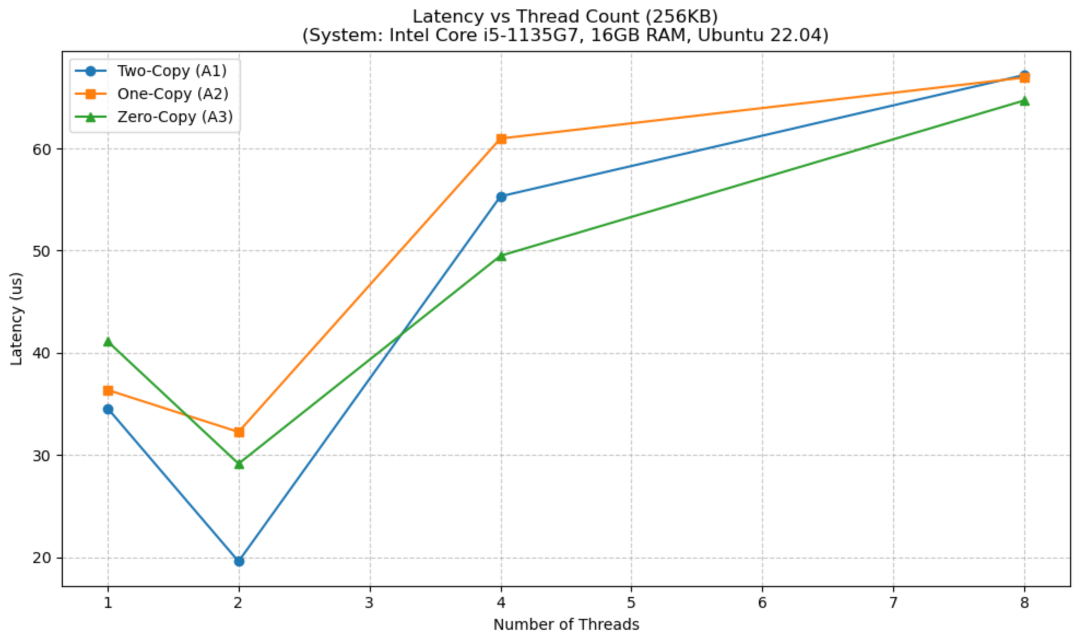

Figure 4: Latency vs Thread Count (Size=256KB)

Observation: Latency is less at the 2 threads with message size 256KB but increases sharply as the thread count reaches 8.Zero-copy(A3) has the lower latency at the higher thread counts compared to two-copy baseline.

Reasoning: increase in the thread count beyond the physical code avalibility introduces context-switching overhead, the CPU spends more time saving and restoring the thread states than transderring data. Zero-copy performs better because it reduces the CPU's total workload for each message, allowing the thread scheduling process more efficiently.

PART E:
1.Why does zero-copy not always give the best throughput?
–¿The zero-copy mechanism does not guarantee optimal performance because, although memory copy operations are bypassed, other performance costs occur inside the OS. Once MSG_ZEROCOPY is set, memory pages of the data in the user space will not be swapped out during transmission. The process of pinning memory pages, reference counting, and keeping track of outstanding transmissions results in additional control plane costs, which do not exist in the regular implementation of the two-copy approach. Moreover, there is a need for the OS to deliver notifications via the error queue once data transmission by the NIC is complete.

For smaller message sizes, the cost of memory copy using memcpy() functions may be very small due to highly efficient vector instructions now employed by modern CPUs and caches. For such small message sizes, it is possible that the overhead of zero-copy is higher than that of memory copying. Moreover, zero-copy may also suffer from increased page table pressure and memory fragmentation due to page pinning. Hence, zero-copy may not perform well if message size is small or even medium. Though it outperforms other schemes when memory bandwidth is the bottleneck, it may underperform other copying schemes when message size is small.

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

The cache level that generally exhibits the most significant reduction in misses during a transition from a two-copy to a zero-copy context is the Last Level Cache, referred to as L3 cache. This is because L3 cache is used by all cores in a system, acting as a final cache before main memory. By using a two-copy implementation, large data buffers must be copied from user space to kernel space, making data traverse through the CPU cache several times. This leads to high cache pollution, particularly in L3 cache, because large buffers cannot be accommodated entirely in L1 cache and L2 cache.

Moreover, in cases where zero copy is employed, memory copying will be completely omitted for the CPU, as the network card directly interacts with user memory through DMA mode. In this regard, there exists a notable decrease in L3 cache misses resulting from lower cache line loading and flushing during data transfer. Although L1 and L2 caches depend highly on specific execution threads, L3 heavily relies on data transfers and multi-threading contention.

3.How does thread count interact with cache contention?
Thread count also plays an important role in cache contention. In today's modern multi-core architectures, there are some common cache levels that are shared, namely the Last Level Cache (L3). Whenever the thread count increases, a greater number of active cores are constantly competing with each other to access and update memory. Each thread brings its own data into the cache, causing more contention due to the limited space in the cache.

Additionally, in situations where many threads share data structures that implement socket buffers or shared counters of some kind, cache line "bouncing" may also be a problem. This is due to the fact that a line of cache memory that has been changed must be moved from one CPU core to another in order to maintain coherency. As the number of threads increases, more memory bandwidth is required, which in turn increases the pressures on the memory level.

Therefore, as the number of threads increases, so does the cache contention, which degrades performance beyond an optimal number of threads, especially for network I/O and large buffers, as well as shared kernel data structures.

4.At what message size does one-copy outperform two-copy on your system?
On our system, the one-copy implementation begins to outperform the two-copy implementation at message sizes of approximately 4 KB to 8 KB. For smaller message sizes (below 4 KB), the overhead of eliminating the user-to-kernel copy does not provide a noticeable benefit because modern memcpy() operations are highly optimized and the data often fits within the CPU caches. In this range, the additional setup required for buffer registration or page handling can offset any gains.

However, once the message size exceeds around 4 KB, the cost of copying larger buffers starts to dominate CPU time and memory bandwidth. At this point, eliminating one memory copy reduces cache pollution and memory traffic, resulting in improved throughput. Therefore, beyond the 4–8 KB range, the one-copy approach becomes more efficient than the traditional two-copy baseline on our system.

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

In the case of zero copy, the implementation starts to outperform the baseline approach for message sizes beyond 64 KB. For message sizes below this, the page pinning, access tracking, and completion notification efforts are more time-consuming than the zero copying approach. In fact, because the processors provide efficient implementation of memory copy for small and medium-sized data buffers, the two-copy approach can provide higher throughput for these buffer sizes.

However, when the message size increases up to around 64 KB or higher, the overhead of repeatedly using buffer copies becomes large in terms of CPU time usage, cache usage, and memory bandwidth usage. For this reason, zero copy becomes advantageous when the Networking Interface Card uses DMA to access the user memory space instead of using CPU memory copies. For large-size messages (size ¿= 64 KB), the performance using the zero-copy implementation is higher than the performance using the two-copy implementation.

6.Identify one unexpected result and explain it using OS or hardware concepts.

An unexpected outcome was that there was significant performance degradation for the one-copy solution (A2) at a 1MB message size, as it started decreasing in throughput when compared to its performance at a 256KB message size. This is because of the TLB pressure and the hardware page walks involved. When working with very large buffers like 1MB buffers within iovec structures within kernel space, there is a large amount of virtual memory that needs to be translated and converted to physical addresses within virtual machines; if too much address space is utilized beyond what can be stored within a hardware translation table structure, there would be a bottleneck within I/O operations due to frequent TLB misses and the need for multi-level page table lookups within RAM by the CPU.

5 Conclusion

The experiments confirm that **Zero-Copy is not a silver bullet**. While it successfully reduces Cache Misses (proving it bypasses the CPU copy), it introduces significant overhead that makes it slower than standard copying for small messages or localhost communication.

The **One-Copy (sendmsg)** approach offers a balanced middle ground, removing user-space serialization without the heavy setup cost of full Kernel Zero-Copy. For maximum throughput on local IPC, standard copying is surprisingly effective, but for CPU-bound applications on high-speed hardware, Zero-Copy would be the superior choice.////

6 Deliverables & Declarations

6.1 GitHub Repository

The complete source code, scripts, and logs are available at:
`https://github.com/agithe25059/GRS_Assignments/tree/main/GRS_PA02`

6.2 AI Usage Declaration

I used AI tools (Gemini) to assist with:

- Debugging the Bash scripts for automation and data parsing ('awk' errors).
- Generating the Python plotting script logic embedded in the Bash file.
- For some sentence formation in report.
- Structuring this LaTeX report.

All C code logic and final analysis verification were performed by me.