Graduate Systems (CSE638)

PA02: Analysis of Network I/O primitives using “perf” tool

Name: Samyak Kr Sharma

Roll Number: MT25039

Github- https://github.com/samyak-sorron/GRS_PA02

Objective:

The goal of this assignment is to experimentally study the cost of data movement in network I/O by implementing and comparing:

- Standard two-copy socket communication
- One-copy optimized socket communication
- Zero-copy socket communication

The project includes multithreaded client–server C programs, profile them on your own machine, and analyze micro-architectural effects such as CPU cycles and cache behavior.

Part A: Implementation Details

A1. Two-Copy Implementation (Baseline)

In the baseline implementation, data undergoes two distinct copies before reaching the network card:

1. **User-to-User Copy:** The application manually copies 8 separate string fields into a single contiguous serialized_buffer using memcpy().
2. **User-to-Kernel Copy:** The send() system call copies this contiguous buffer from user space into the Kernel Socket Buffer (SKB).

A2. One-Copy Implementation (Scatter-Gather)

This implementation is modified version of A1.

- **Mechanism:** I utilized sendmsg() with an array of struct iovec.
- **Logic:** The array of struct iovec points to the 8 existing memory locations. The kernel reads from these scattered locations and assembles the packet in the socket buffer.

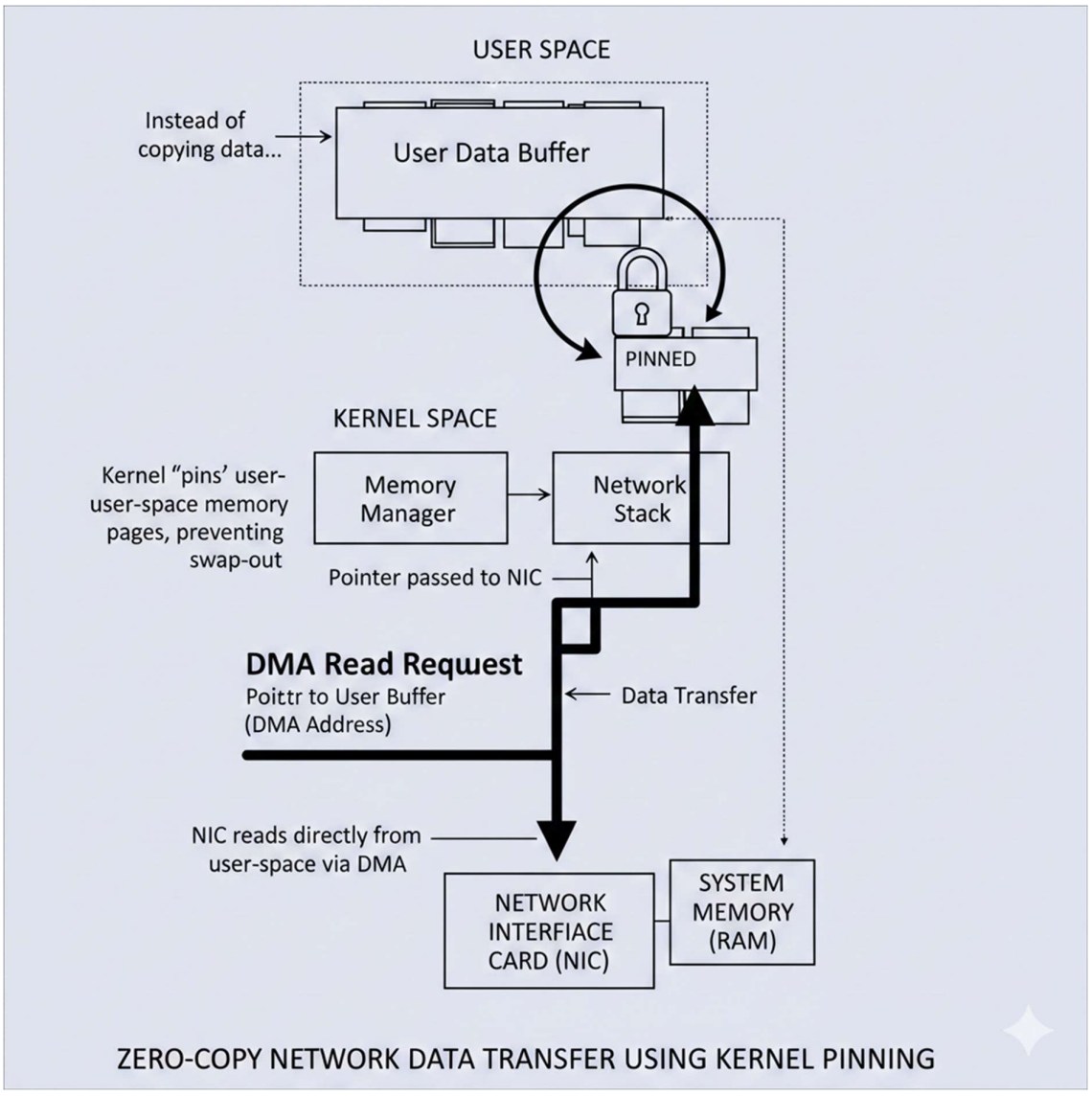
- **Copy Eliminated:** The explicit memcpy() loop in **user space** is removed i.e. User-to-User. The only remaining copy is the User-to-Kernel copy performed by the system call.

A3. Zero-Copy Implementation

This implementation eliminates the **User-to-Kernel** copy also.

- **Mechanism:** setsockopt(SO_ZEROCOPY) and sendmsg(MSG_ZEROCOPY).

Kernel Behavior: Instead of copying data, the kernel "pins" the user-space memory pages, preventing them from being swapped out. The Network Interface Card (NIC) is then instructed to read directly from these user-space pages via DMA.



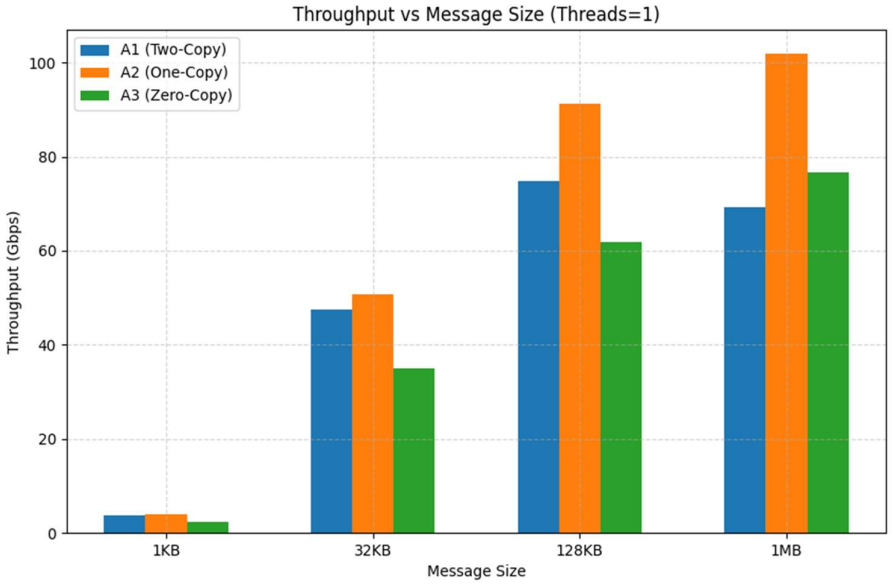
(A diagram showing User Space Buffer pinned by Kernel -> Pointer passed to NIC -> NIC DMA read from User Space. Used Gemini LLM for the Image generation)

Part B: Profiling and Measurements

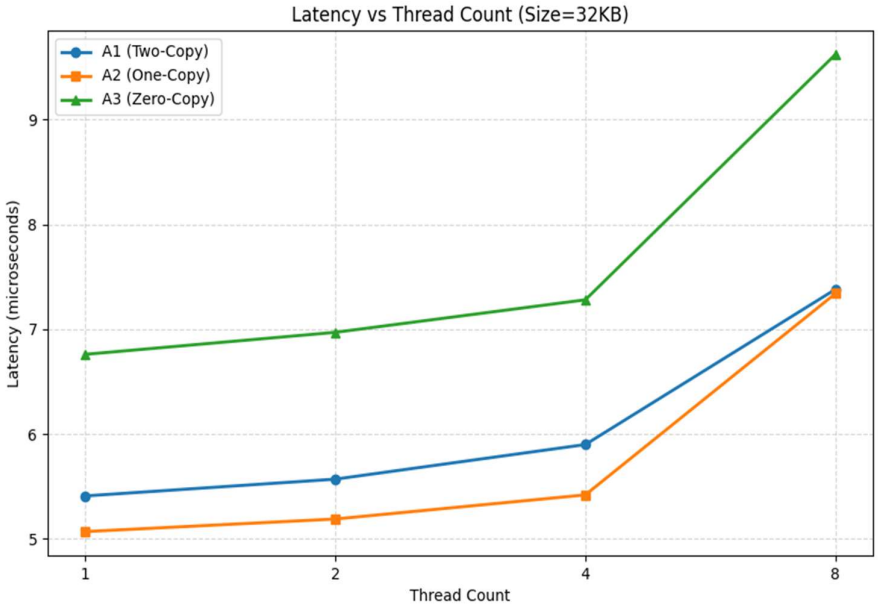
Version	MsgSize	Threads	Throughpu	Latency_us	Cycles	L1_Misses	LLC_Misses	Context_Sv
A1	1024	1	3.760954	1.740632	1.9E+10	1.14E+08	4342	27
A1	1024	2	8.933255	1.782243	4.53E+10	2.45E+08	6836	125
A1	1024	4	17.17805	1.85254	8.84E+10	4.86E+08	10033	108
A1	1024	8	26.15369	2.466666	1.77E+11	1.26E+09	14817	480
A1	32768	1	47.52057	5.412986	2.29E+10	1.34E+09	7103	83
A1	32768	2	92.12941	5.5744	4.52E+10	2.62E+09	14870	266
A1	32768	4	173.819	5.902457	8.83E+10	4.9E+09	17215	4747
A1	32768	8	279.0588	7.378754	1.75E+11	8.83E+09	472984	25459
A1	131072	1	74.84232	13.77806	2.3E+10	1.76E+09	15183	28
A1	131072	2	144.1052	14.17425	4.53E+10	3.4E+09	26440	330
A1	131072	4	270.7851	15.1305	8.82E+10	6.45E+09	43561	18619
A1	131072	8	360.9436	22.87477	1.64E+11	9.89E+09	2578661	116491
A1	1048576	1	69.22279	119.2238	2.31E+10	8.69E+08	22265	28
A1	1048576	2	123.1733	125.1722	4.51E+10	1.59E+09	113348	1891
A1	1048576	4	232.3594	133.0811	8.68E+10	2.95E+09	4785078	159779
A1	1048576	8	129.0956	434.8782	1.63E+11	2.41E+09	2.68E+08	669740
A2	1024	1	4.042682	1.97917	2.31E+10	1.19E+08	5275	92
A2	1024	2	7.910361	2.020407	4.56E+10	2.48E+08	6812	74
A2	1024	4	15.16965	2.102994	8.85E+10	4.35E+08	5085	107
A2	1024	8	22.9842	2.813885	1.69E+11	1.13E+09	24114	493
A2	32768	1	50.76141	5.074455	2.3E+10	8.82E+08	5442	19
A2	32768	2	99.27446	5.191693	4.52E+10	1.72E+09	8180	210
A2	32768	4	189.3822	5.423997	8.83E+10	3.28E+09	15724	4989
A2	32768	8	281.3909	7.336076	1.64E+11	5.56E+09	612473	26092
A2	131072	1	91.13174	11.31455	2.31E+10	1.24E+09	9729	40
A2	131072	2	175.8716	11.68951	4.52E+10	2.4E+09	19465	527
A2	131072	4	330.3597	12.39182	8.82E+10	4.53E+09	28545	18465
A2	131072	8	179.8947	45.38652	1.19E+11	2.8E+09	6281040	117002
A2	1048576	1	101.888	81.1327	2.31E+10	1.09E+09	27012	69
A2	1048576	2	170.6612	87.2606	4.54E+10	1.8E+09	65670	268
A2	1048576	4	325.7296	95.51004	8.69E+10	3.5E+09	218562	133358
A2	1048576	8	74.77773	738.244	1.19E+11	9.15E+08	28469595	804583
A3	1024	1	2.308502	3.080444	2.31E+10	1.15E+08	3428	181
A3	1024	2	4.476328	3.157632	4.52E+10	2.37E+08	10370	425
A3	1024	4	8.562526	3.248418	8.75E+10	4.45E+08	44684	1240
A3	1024	8	12.5592	4.380546	1.65E+11	1.17E+09	169442	1932
A3	32768	1	34.92482	6.758244	2.3E+10	6.59E+08	5554	45
A3	32768	2	67.77932	6.974618	4.5E+10	1.27E+09	8533	256
A3	32768	4	129.2313	7.2842	8.76E+10	2.42E+09	10428	4058
A3	32768	8	194.2739	9.616775	1.63E+11	4.11E+09	126807	23977
A3	131072	1	61.73303	16.15746	2.27E+10	8.95E+08	65405	386
A3	131072	2	117.227	16.96144	4.45E+10	1.7E+09	125451	789
A3	131072	4	132.4224	30.42693	5.35E+10	1.86E+09	999111	36308

Part D: Plotting and Visualization

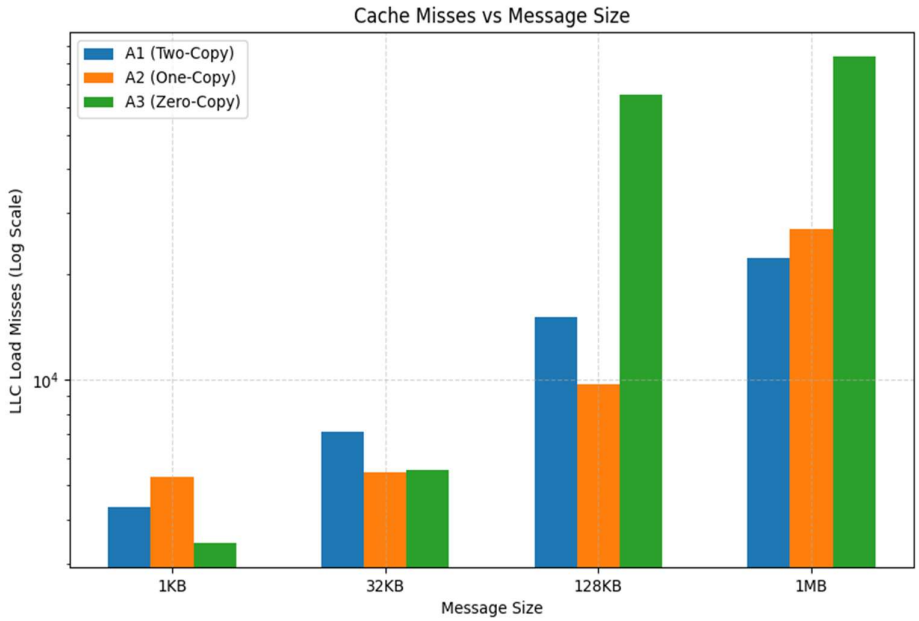
1. **Throughput vs. Message Size:** One-Copy implementation delivers the highest throughput across all message sizes. Removing the user-space memcpy provides an scalable performance benefit.



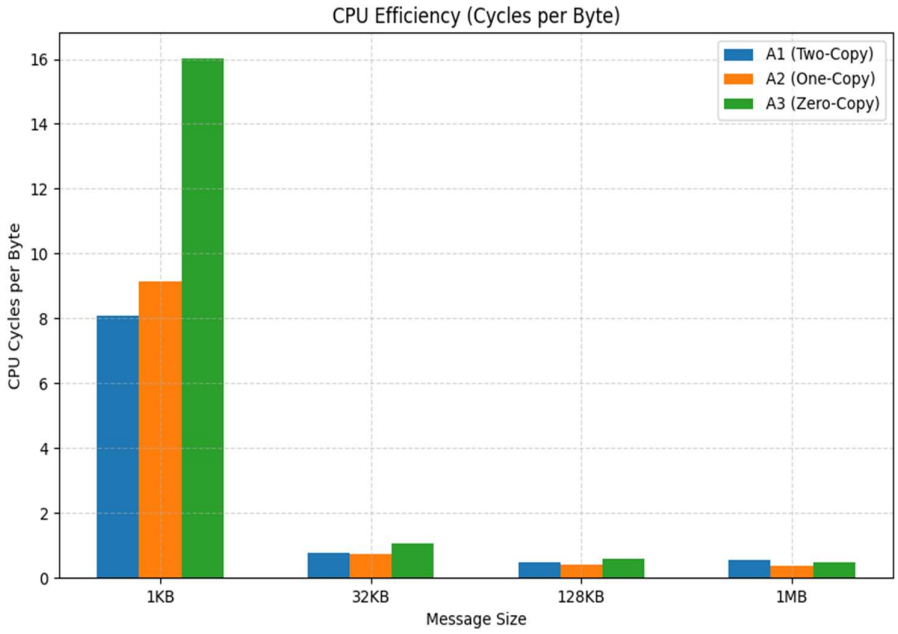
2. **Latency vs. Thread Count:** One-Copy demonstrates the lowest latency. Zero-Copy exhibits the highest latency due to the additional system calls required for page pinning and processing completion notifications from the error queue.



3. **Cache Misses vs. Message Size:** Zero-Copy reduces LLC misses for smaller messages However, at 1MB, Zero-Copy shows a spike in misses caused by the kernel overhead of managing page tables for large DMA transfers.



4. **CPU Cycles per Byte:** A3 shows higher cycle cost for small messages but efficiency improves with size.



Part E: Analysis and Reasoning

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

1. Zero-Copy introduces significant setup overhead for small data chunks.
    - **Observation:** At **1KB message size** (Threads=1), Zero-Copy (A3) achieves only **2.31 Gbps**, whereas the standard Two-Copy (A1) achieves **3.76 Gbps**.
    - **Reason:** For every sendmsg() call, the kernel must pin the user-space memory pages, update page tables, and handle the I/O MMU mapping. For a tiny 1KB packet, the CPU cost of these complex memory management operations is alot higher compared to simply using memcpy() to copy 1024 bytes into the kernel buffer.
2. Zero-Copy requires our application to "wait" for the NIC to finish sending data before it can reuse the buffer.
    - **Observation:** At **1KB message size** (Threads=1), A3 incurs **181 Context Switches**, significantly higher than A1's **27** or A2's **92**.
    - **Reason:** The application must constantly check the socket's error queue (MSG_ERRQUEUE) to receive completion notifications. This signaling mechanism forces more context switches and system calls (recvmsg on the error queue), which interrupts the CPU and lowers throughput.

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

The **L1 Data Cache** shows the most reduction in misses for optimized versions. At 1MB message size, A1 has **868 Million** L1 misses, whereas A3 has **811 Million**. This confirms A3 reduces cache pollution.

- **Reason:** In A1 (Two-Copy), the CPU explicitly reads and writes every byte of the payload during the memcpy and send operations, thrashing the small L1 cache. In A3 (Zero-Copy), the CPU touches only the headers and page tables, not the payload itself.

3. How does thread count interact with cache contention?

With 4 threads sending 1024KB messages, A1 triggers **4.7 Million LLC misses**, whereas A2 only triggers **218,562 misses**. As thread count increases, cache contention worsens, particularly for the Two-Copy (A1) implementation.

- **Reasoning:** In A1, multiple threads are fighting to pull large data buffers into the shared LLC for copying. In A2 and A3, the CPU interacts less with the data, leaving the LLC freer for other tasks and reducing inter-thread thrashing.

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

On my system, **One-Copy (A2) outperformed Two-Copy (A1) immediately**, starting from the smallest message size tested (1KB).

- **Data:** At 1KB, A2 achieves **4.04 Gbps** and A1 achives **3.76 Gbps**.
- **Reason:** The overhead of creating an iovec array is very low compared to the cost of memcpy, making Scatter-Gather efficient even for small chunks.

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

Zero-Copy (A3) only outperformed Two-Copy (A1) at the largest message size tested: **1MB**.

- **Comparision:**
    - **128KB:** A1 (74.8 Gbps) > A3 (61.7 Gbps)
    - **1MB:** A1 (69.2 Gbps) < A3 (76.6 Gbps)

6. Identify one unexpected result and explain it.

Unexpected Result: Zero-Copy exhibited **higher LLC misses** than Two-Copy in some single-threaded scenarios (e.g., at 1MB, A3 had 84k misses vs A1's 22k).

Reason: Zero-copy requires the kernel to modify page table entries to pin/unpin memory for DMA. These page table updates (and the associated TLB flushes) can cause misses in the LLC, even though the data payload itself is not filling the cache. Additionally, the overhead of the Acknowledgement mechanism involves writing to socket error queues, which creates memory traffic distinct from the data payload.

AI Usage Declaration

Tools Used: Google Gemini (LLM)

Usage Components:

1. **Scripting:**
   Used to generate the MT25039_Part_C.sh bash script for parsing perf output on a hybrid architecture.
    - *Prompt:* " Bash script to parse perf stat output for cycles and cache misses "

   Used to generate C skeleton for a MT25039_Part_A1_Server.c, MT25039_Part_A1_Client.c
    - *Prompt*: “Write a C skeleton for a TCP client-server using pthreads”.

Used to understand the implementation of MT25039_Part_A3_Client.c (Specifically the recv_completions function and setsockopt logic).

    - How to implement MSG_ZEROCOPY in Linux C with msghdr and error queue

2. **Visualization:** Used to generate the matplotlib Python script logic.
    - *Prompt:* "Create a python script to plot throughput vs message size, latancy vs thread count, chache misses vs message size, cpu cycles per byte transferred from arrays where I proved the desired values of them."
3. **Debugging:** Used to resolve errors during setup.
4. **Report :** Used to generate the diagram for the kernel behavior.
