PA02: Analysis of Network I/O Primitives using *perf*

Report

Ishita Singh (MT25026)

1. Introduction

Modern high-performance systems are often bottlenecked not by computation but by **data movement**, particularly across kernel-user boundaries during network communication. Traditional socket I/O involves multiple memory copies, which increase CPU utilization and cache pressure.

This assignment experimentally studies the cost of data movement by comparing three TCP-based client-server implementations:

1. **Two-copy socket communication**
2. **One-copy optimized socket communication**
3. **Zero-copy socket communication**

All implementations are evaluated using **application-level metrics** and **micro-architectural profiling via *perf***, with experiments automated using a shell script and isolated using Linux **network namespaces**.

2. Experimental Setup

2.1 System Configuration

- **CPU:** x86_64 multi-core processor
- **OS:** Ubuntu Linux
- **Kernel:** Linux (supports *MSG_ZEROCOPY*)
- **Tooling:** *perf stat*, *pthread*, *ip netns*

2.2 Network Configuration

- Client and server are executed in **separate network namespaces** (*ns_client*, *ns_server)*
- Communication via a **veth pair**
- IP addresses:
  - Server: *10.0.0.1*
  - Client: *10.0.0.2*

This ensures isolation and avoids loopback optimizations.

3. Part A: Multithreaded Socket Implementations

3.1 Server

- Accepts multiple concurrent TCP clients
- Uses **one thread per client**
- Sends a fixed-size message repeatedly
- Message structure contains **8 heap-allocated string fields**

3.2 Client

- Connects to server
- Receives messages continuously for a **fixed duration**

3.3 Runtime Parameters

- Message size
- Thread count
- Duration

4. Part A1: Two-Copy Implementation (Baseline)

4.1 Mechanism

- Uses *send()* on server and *recv()* on client
- Each message field is sent individually

4.2 Analysis

1. Where do the two copies occur?

1. User buffer → Kernel socket buffer (server side)
2. Kernel socket buffer → NIC buffer (client side)

For every call to:

send(fd, msg->fields[i], size, 0)

the data moves as follows:

Copy 1: User space → Kernel space

- Source: *msg->fields[i]* (heap buffer in **user space**)
- Destination: TCP **socket and buffer** in **kernel space**
- Mechanism: *copy_from_user()*
- Performed by: **Kernel**

Copy 2: Kernel space → Network device (NIC)

- Source: Kernel socket buffer
- Destination: NIC transmit buffer

- Mechanism: **DMA (Direct Memory Access)**
- Initiated by: Kernel/NIC driver
- Performed by: **Hardware (DMA engine)**

2. Is it actually only two copies?

Logically: Yes, from the application's perspective
Physically: Possibly more, depending on hardware and cache behavior

- Cache line fills and evictions
- Store buffers
- NIC ring buffer staging

3. Which components (kernel/user) performs the copies?

Copy stage	From/To	Who performs it
Copy 1	User buffer → Kernel socket buffer	**Kernel (CPU)**
Copy 2	Kernel socket buffer → NIC buffer	**NIC (DMA), coordinated by kernel**

- User space never copies data explicitly
- The application only *requests* transmission via *send()*
- All actual data movement is handled by the **kernel and hardware**

5. Part A2: One-Copy Implementation

5.1 Mechanism

- Uses *sendmsg()* with a pre-constructed *iovec*
- Message fields are sent in a **single system call**
- Message buffers are allocated using *mmap()*
- Scatter-gather I/O allows multiple message fields to be sent **without user-space concatenation**

5.2 Copy Reduction

- Eliminates **intermediate user-space concatenation**
- User-space to kernel-space payload copy still occurs
- Kernel-to-NIC DMA copy is still required
- Overall data movement is reduced by **one user-space copy**

5.3 Analysis

1. Which copy has been eliminated?

In the A2 implementation, message buffers are allocated using *mmap()*, and transmitted using *sendmsg()* with scatter-gather I/O (*iovec*).

The intermediate user-space concatenation copy is eliminated.

- In the baseline implementation, multiple message fields would need to be copied into a contiguous buffer before transmission
- In A2, *sendmsg()* directly references multiple buffers via *iovec*, avoiding this extra user-space copy
- However, the kernel still performs a *copy_from_user()* operation to move data into the kernel socket buffer

Therefore, the communication path contains **one fewer copy** compared to A1.

6. Part A3: Zero-Copy Implementation

6.1 Mechanism

- Uses *sendmsg()* with the *MSG_ZEROCOPY* flag
- Zero-copy support enabled via the *SO_ZEROCOPY* socket option
- Message buffers are allocated using *mmap()*, providing **page-aligned memory** that can be pinned by the kernel during zero-copy transmission
- The kernel pins user pages and hands references directly to the NIC
- Network interface card performs DMA directly from user-space buffers

6.2 Copy Reduction

- Eliminates **user-space to kernel-space payload copy**
- Eliminates **kernel-space to NIC copy**
- Kernel handles only metadata; payload bytes are not copied
- Zero-copy operation is **best-effort** and may fall back under memory pressure

6.3 Analysis

1. Kernel Behavior using a Diagram

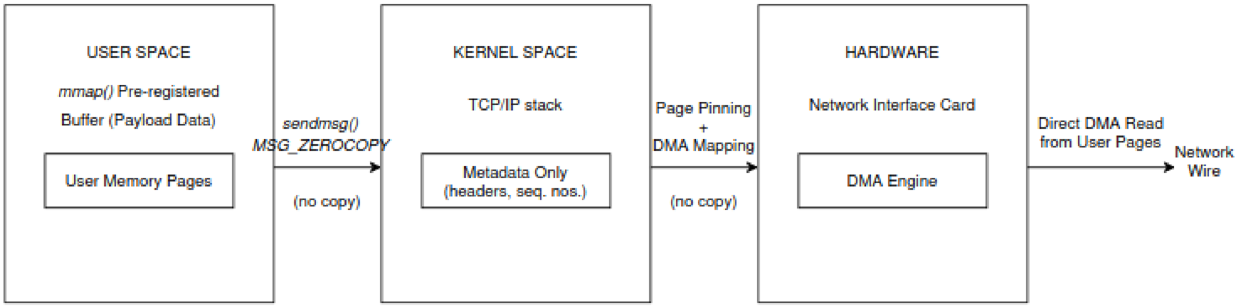


Fig. 1. Zero-copy transmission path using *MSG_ZEROCOPY*, where the NIC performs DMA directly from user-space memory without intermediate kernel buffering

1. **User Space (*mmap* buffer)**
   - Payload data resides in an *mmap()*-allocated buffer
   - Pages are page-aligned and visible to the kernel
2. ***sendmsg(MSG_ZEROCOPY)***
   - The system call passes **references**, not data
   - No *copy_from_user()* is performed
3. **Kernel Space (TCP/IP Stack)**
   - Kernel processes **only metadata** (TCP headers, ACKs)
   - User pages are **pinned** to prevent eviction
   - DMA mappings are created for the NIC
4. **NIC (DMA Engine)**
   - NIC reads payload data **directly from user memory**
   - No kernel buffer holds a copy of the payload data

7. Part B: Profiling and Measurement

In this part, we collect quantitative performance measurements for the three implementations (A1, A2, and A3). All experiments were conducted in isolated network namespaces to ensure reproducibility and minimize external interference.

7.1 Metrics Collected

The following metrics were collected for each implementation:

Metric	Description	Measurement Tool
Throughput (Gbps)	End-to-end data transfer rate	Application-level timing
Latency (µs)	Time per message transmission	Application-level timing
CPU Cycles	Total CPU cycles consumed	*perf stat*
L1 Cache Misses	L1 data cache misses	*perf stat*
LLC Cache Misses	Last-level cache misses	*perf stat*
Context Switches	Number of voluntary/involuntary context switches	*perf stat*

7.2 Experimental Dimensions

To ensure representative and scalable results, measurements were taken across multiple workload configurations.

Message Sizes (Bytes)

- 64
- 256
- 1024
- 4096

These sizes capture small, medium, and large message behaviors.

Thread Count

- 1
- 2
- 4
- 8

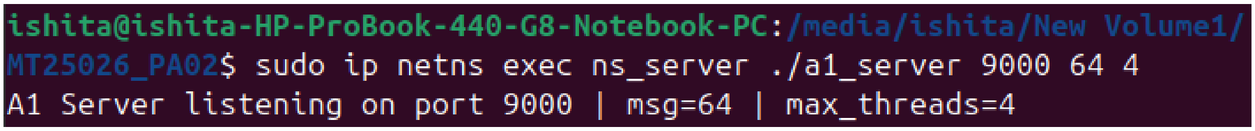
This allows analysis of scalability and contention effects.

7.3 Experimental Setup

- **Operating System:** Linux (Ubuntu)
- **Communication:** TCP sockets
- **Isolation:** Linux network namespaces
- **Server Model:** Multithreaded, one thread per client
- **Execution Control:** Client controls experiment duration
- **Measurement Scope:** Server-side profiling

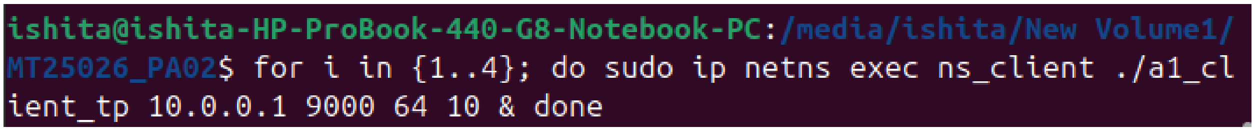
7.4 Experimental Procedure

1. Two isolated network namespaces were created to represent the server and client environments
2. A virtual Ethernet pair was set up to enable communication between the two namespaces
3. IP addresses were assigned to the virtual interfaces for inter-namespace communication
4. All network and loopback interfaces were activated inside their respective namespaces
5. The server application was launched inside the server namespace using the chosen port number, message size, and thread count
6. The client application was executed inside the client namespace to initiate data transfer for a fixed duration
7. CPU and cache-related performance metrics, including cycles, cache misses, and context switches, were collected during server execution
8. Throughput was calculated as the aggregate data rate across all concurrent client connections over the experiment duration
9. Latency was measured as the average time per message at the application level across all concurrent connections
10. The entire procedure was repeated for multiple message sizes, multiple thread counts, and all three implementations (A1, A2, and A3)



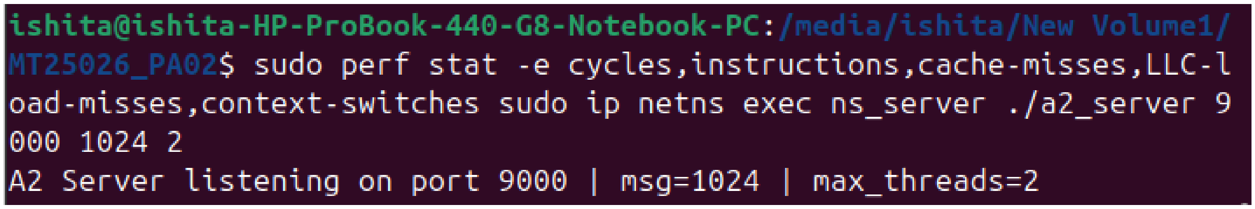
```
ishita@ishita-HP-ProBook-440-G8-Notebook-PC:/media/ishita/New Volume1/
MT25026_PA02$ sudo ip netns exec ns_server ./a1_server 9000 64 4
A1 Server listening on port 9000 | msg=64 | max_threads=4
```

Fig. 2. Server launched inside the *ns_server* server namespace with port = 9000, message size = 64 bytes, and thread count = 4



```
ishita@ishita-HP-ProBook-440-G8-Notebook-PC:/media/ishita/New Volume1/
MT25026_PA02$ for i in {1..4}; do sudo ip netns exec ns_client ./a1_cl
ient_tp 10.0.0.1 9000 64 10 & done
```

Fig. 3. Four clients executed inside the *ns_client* client namespace with port = 9000, message size = 64 bytes, and duration = 10 sec

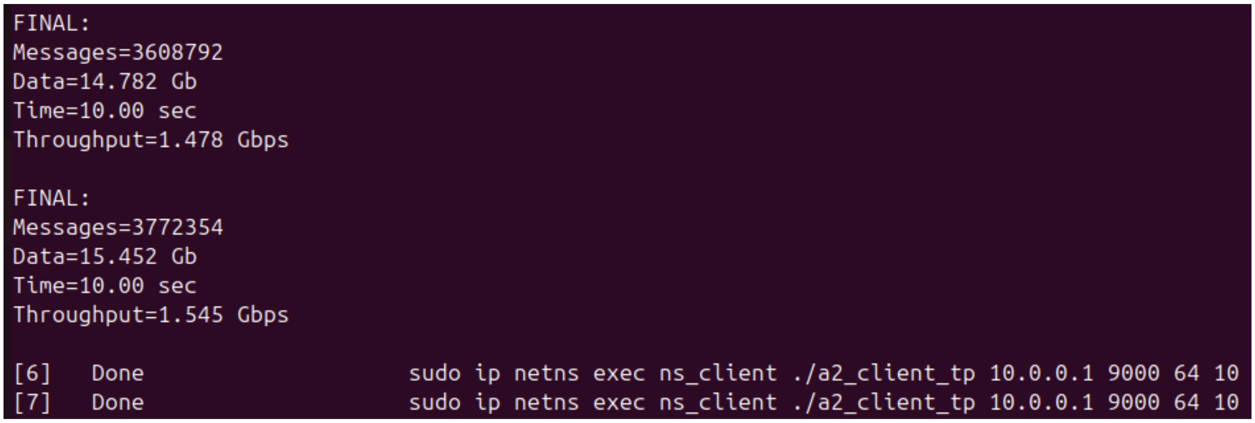


```
ishita@ishita-HP-ProBook-440-G8-Notebook-PC:/media/ishita/New Volume1/
MT25026_PA02$ sudo perf stat -e cycles,instructions,cache-misses,LLC-l
oad-misses,context-switches sudo ip netns exec ns_server ./a2_server 9
000 1024 2
A2 Server listening on port 9000 | msg=1024 | max_threads=2
```

Fig. 4. Server launched inside the *ns_server* server namespace with port = 9000, message size = 1024 bytes, and thread count = 2, with *perf stat* to collect profiling data

Sample Run: A2 Server - 2 Threads, 64-byte Message Size (10 sec)

Throughput



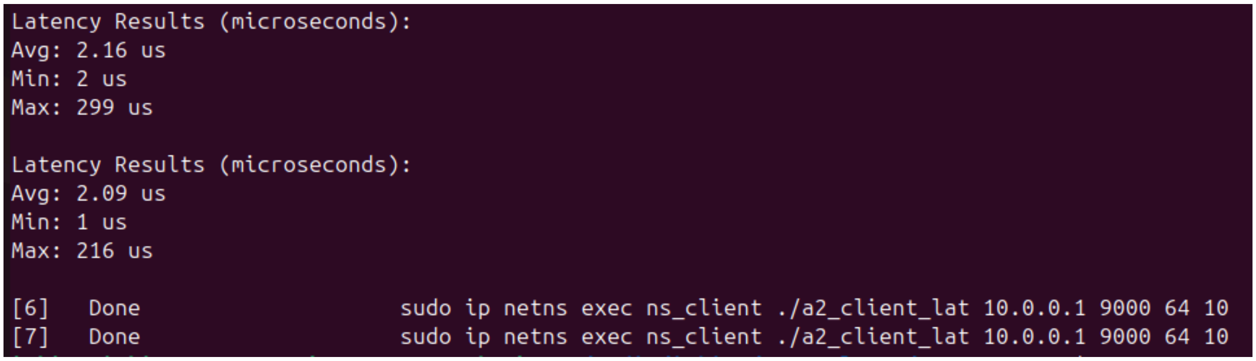
```
FINAL:
Messages=3608792
Data=14.782 Gb
Time=10.00 sec
Throughput=1.478 Gbps

FINAL:
Messages=3772354
Data=15.452 Gb
Time=10.00 sec
Throughput=1.545 Gbps

[6]   Done                    sudo ip netns exec ns_client ./a2_client_tp 10.0.0.1 9000 64 10
[7]   Done                    sudo ip netns exec ns_client ./a2_client_tp 10.0.0.1 9000 64 10
```

Fig. 5. Throughput client output for A2 server with message size = 64 bytes, and thread count = 2

Latency



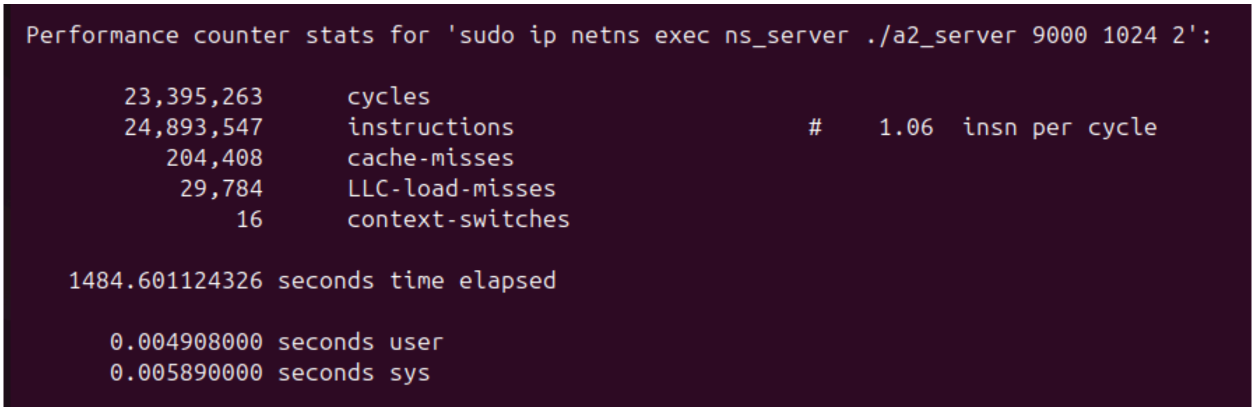
```
Latency Results (microseconds):
Avg: 2.16 us
Min: 2 us
Max: 299 us

Latency Results (microseconds):
Avg: 2.09 us
Min: 1 us
Max: 216 us

[6]   Done                    sudo ip netns exec ns_client ./a2_client_lat 10.0.0.1 9000 64 10
[7]   Done                    sudo ip netns exec ns_client ./a2_client_lat 10.0.0.1 9000 64 10
```

Fig. 6. Latency client output for A2 server with message size = 64 bytes, and thread count = 2

Perf



```
 Performance counter stats for 'sudo ip netns exec ns_server ./a2_server 9000 1024 2':

        23,395,263      cycles
        24,893,547      instructions                     #    1.06  insn per cycle
           204,408      cache-misses
            29,784      LLC-load-misses
                16      context-switches

    1484.601124326 seconds time elapsed

       0.004908000 seconds user
       0.005890000 seconds sys
```

Fig. 7. Perf client output for A2 server with message size = 64 bytes, and thread count = 2

8. Part C - Automated Experiment Script

8.1 Objective

- Automate the entire experiment process so that no manual steps are needed once the script starts
- Ensure all implementations (A1, A2, and A3) are tested under identical conditions
- Evaluate performance across different message sizes and thread counts in a consistent manner
- Collect performance metrics reliably for later comparison and analysis
- Store results in a clean, structured format suitable for plotting and reporting

8.2 Methodology (Bash Script)

1. Compile all server and client programs at the beginning of the script
2. Set up an isolated client and server network namespaces to avoid external interference
3. Run experiments by looping over predefined message sizes and thread counts
4. Start the server with the required parameters for each configuration
5. Launch multiple client instances in parallel using loop to match the server's thread count
6. Allow clients to run continuously for fixed duration to capture steady-state behavior
7. Collect CPU and cache-related metrics automatically using *perf stat*
8. Measure throughput and latency at application level during each run
9. Save all measurements in CSV files, with filenames clearly indicating experiment parameters
10. Clean up processes and temporary resources after each run to allow repeatable execution

8.3 Observations

1. Throughput Scaling

- Throughput improves with both increasing message size and thread count for all implementations, but with different scaling efficiencies
- **A2 (one-copy) implementation achieves the highest throughput across all configurations**, reaching peak values at large message sizes (4096 bytes) and high thread counts
- A3 (zero-copy) scales reasonably well but consistently delivers lower throughput than A2 (one-copy), while A1 (two-copy) shows limited scalability and saturates earlier

2. Latency Trends

- Across all message sizes and thread counts, **A2 (one-copy) consistently achieves the lowest average latency**, followed by **A3 (zero-copy)**, while **A1 (two-copy) exhibits the highest latency**, especially at higher thread counts
- For small messages (64-256 bytes), A2 (one-copy) maintains near-constant latency as thread count increases, indicating efficient synchronization and low contention
- In contrast, A1 (two-copy) shows a sharp increase in latency beyond 4 threads, with extreme maximum latency values, suggesting scheduler contention and frequent context switches
- A3 (zero-copy) lies between the two, scaling better than A1 (two-copy) but still showing noticeable latency growth under high concurrency

3. CPU and Cache Behavior

- **A1 (two-copy) implementation records extremely high context-switch counts**, especially at higher thread counts, directly relating with its poor latency and scalability
- It also experiences large last-level cache (LLC) miss rates for bigger message sizes, increasing memory access penalties
- **A2 (one-copy) implementation exhibits the lowest cycle counts, fewer cache misses, and significantly fewer context switches**, indicating better CPU utilization and cache locality
- A3 (zero-copy) shows moderate cache miss rates and context switching, explaining its intermediate performance between A1 (two-copy) and A2 (one-copy)

9. Part D - Plotting and Visualization

The collected CSV data was visualized using **matplotlib** to analyze performance trends across different configurations.

9.1 Plot throughput vs message size

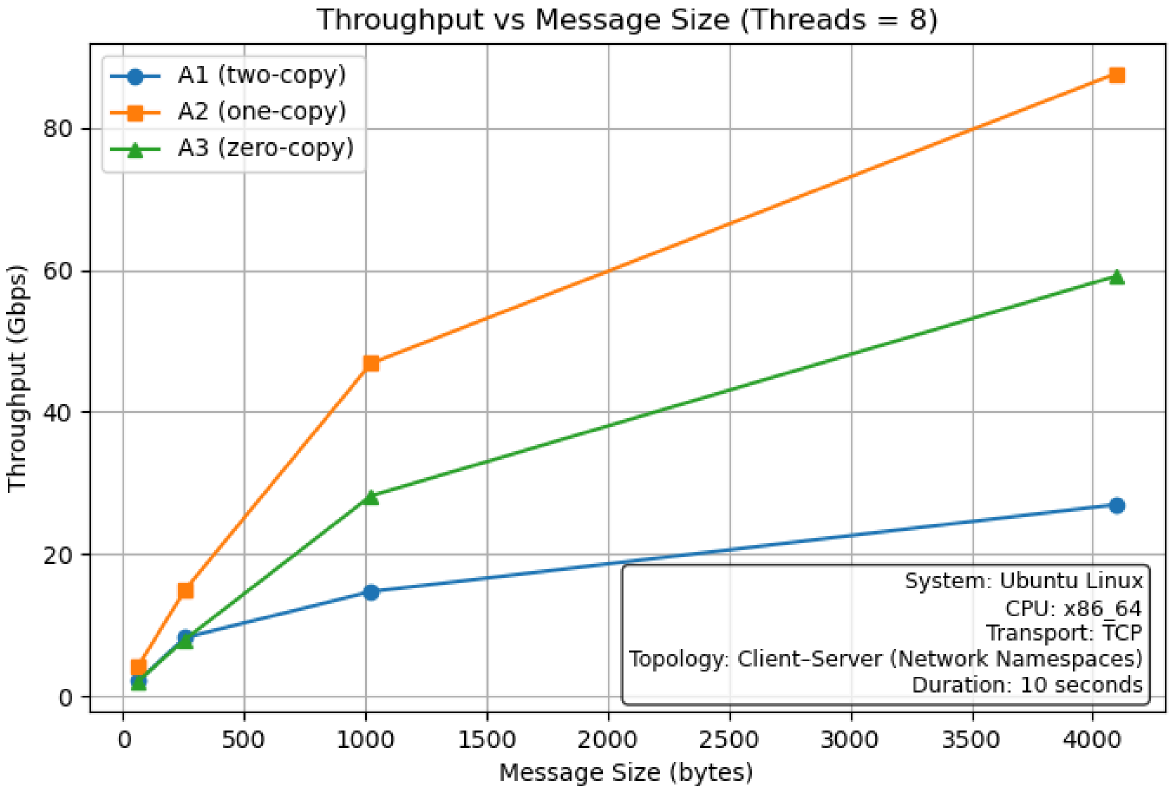


Fig. 8. Throughput vs Message Size (Threads = 8)

Observations and Analysis

- Throughput increases with message size for all implementations, as larger messages reduce the relative overhead of system calls and protocol processing
- A2 (one-copy) achieves the highest throughput across all message sizes, showing that reducing copy overhead without zero-copy setup costs is most effective on this system
- A3 (zero-copy) performs better than A1 (two-copy) but remains below A2 (one-copy), indicating that the zero-copy setup and page-pinning overhead limit its throughput at the tested message sizes

9.2 Plot latency vs thread count

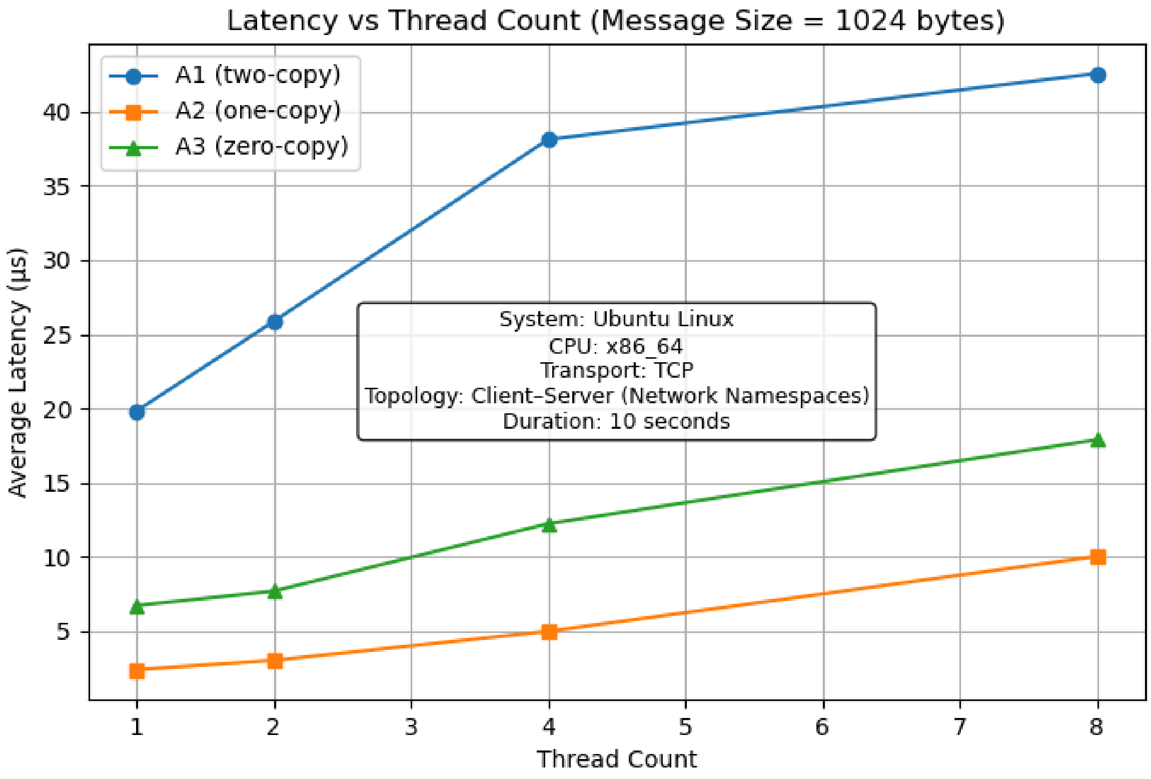


Fig. 9. Latency vs Thread Count (Message Size = 1024 bytes)

Observations and Analysis

- Latency increases with thread count for all implementations due to higher CPU contention, cache interference, and context switching as more threads run concurrently
- A1 (two-copy) shows the highest latency at all thread counts because repeated data copying and multiple system calls increase processing overhead per message
- A2 (one-copy) consistently achieves the lowest latency, while A3 (zero-copy) performs better than A1 (two-copy) but slightly worse than A2 (one-copy) due to additional zero-copy setup overhead

9.3 Plot cache misses vs message size

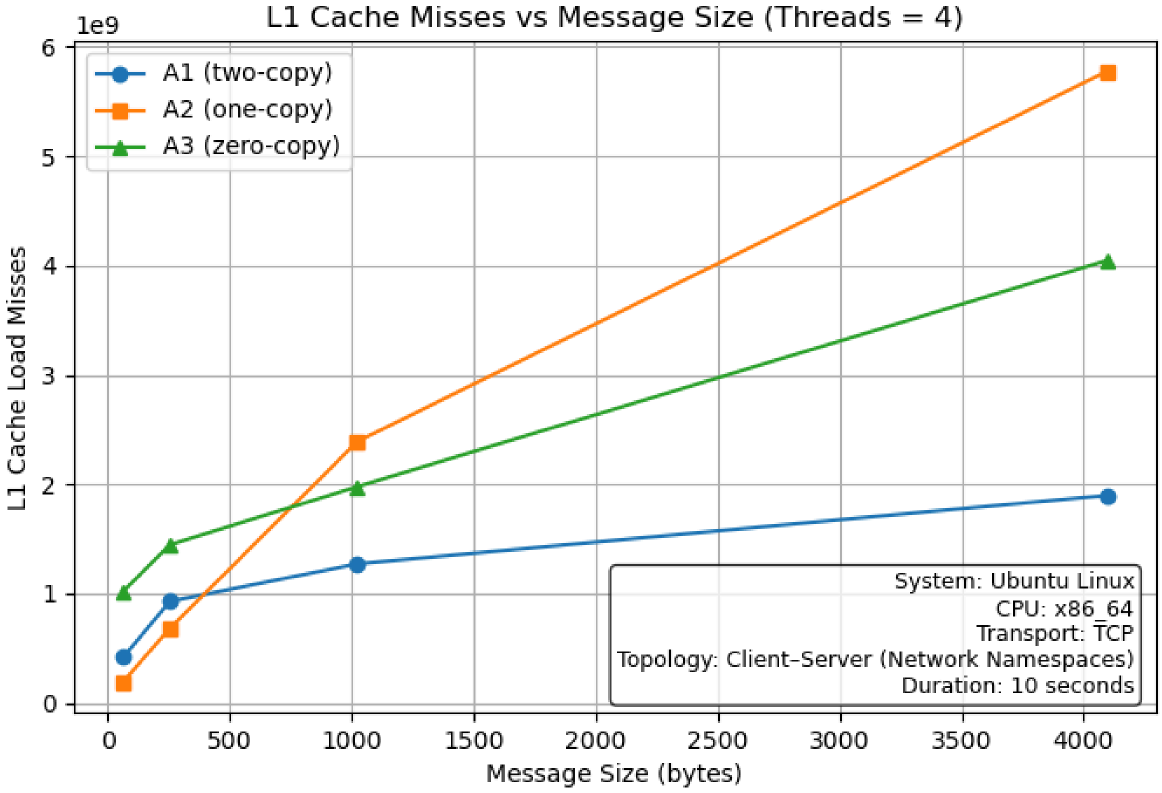


Fig. 10. L1 Cache Misses vs Message Size (Threads = 4)

Observations and Analysis

- L1 cache misses increase with message size for all implementations because larger messages touch more memory and exceed cache capacity more frequently
- A1 (two-copy) has the lowest L1 misses because repeated copying reuses hot cache lines, even though it increases CPU work
- A2 (one-copy) and A3 (zero-copy) show higher L1 misses as message size grows since reducing or eliminating copies shifts data access patterns away from cache reuse, with A3 performing better than A2 due to fewer overall memory touches

9.4 Plot CPU cycles per byte transferred

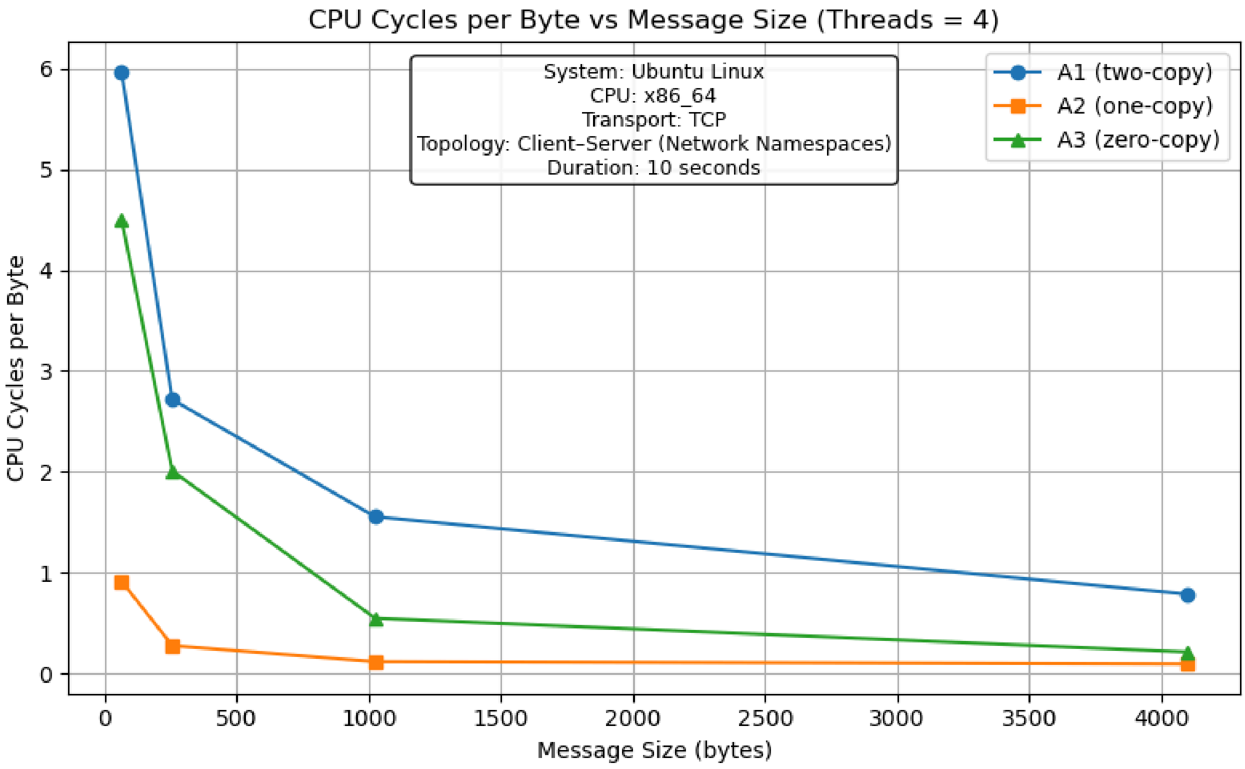


Fig. 11. CPU Cycles per Byte vs Message Size (Threads = 4)

Observations and Analysis

- CPU cycles per byte decrease as message size increases for all implementations, since fixed overheads (system calls and protocol processing) are amortized over more data
- A2 (one-copy) is the most CPU-efficient across all message sizes, showing the lowest cycles per byte due to reduced copying without zero-copy setup overhead
- A3 (zero-copy) performs better than A1 (two-copy) but slightly worse than A2 (one-copy) for smaller messages, indicating that zero-copy benefits become significant only for larger transfers

10. Part E - Analysis and Reasoning

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

Zero-copy reduces data copying, but it also has extra overhead such as page pinning and DMA setup. For small or medium message sizes, this overhead can be larger than the cost of copying itself, so throughput does not always improve. Zero-copy works best only when messages are large enough to amortize this setup cost.

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

The **L1 cache** shows the largest reduction in misses. This is because eliminating memory copies reduces repeated access to the same data, leading to less cache pollution. Since L1 cache is the most frequently accessed cache, it benefits the most from reduced data movement.

10.3 How does thread count interact with cache contention?

As the number of threads increases, multiple cores compete for shared cache resources. This causes cache lines to be evicted more frequently, increasing cache misses and memory traffic. Beyond a certain number of threads, this contention reduces performance gains from parallelism.

10.4 At what message size does one-copy outperform two-copy on *your* system?

One-copy begins to outperform two-copy even at **small message sizes (around 64 bytes)** in terms of throughput and latency, primarily due to reduced system call overhead. However, in terms of cache misses and CPU efficiency, the performance advantage becomes more pronounced at **medium message sizes (approximately a few hundred bytes to 1 KB)**. At these sizes, the reduction in data copying overhead outweighs the remaining syscall cost.

10.5 At what message size does zero-copy outperform two-copy on *your* system?

Zero-copy begins to outperform two-copy at moderate to **large message sizes**. It shows lower latency from a**round 256 bytes**, while clear benefits in CPU efficiency and cache behavior appear only at **larger message sizes (around 1 KB - 4 KB)**, especially with higher thread counts. For small messages (64-128 bytes), the overhead of zero-copy mechanisms outweighs the benefits of avoiding data copies.

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

An unexpected result was that A2 (one-copy) showed a slight drop in throughput at 8 threads for small message sizes. This happens because at high concurrency, CPU scheduling overhead, increased context switching, and cache contention dominate execution. For small messages, these OS and hardware overheads outweigh the benefits of parallelism, leading to reduced throughput.

11. Conclusion

This assignment demonstrates that **reducing data copies significantly improves efficiency**, but only when message size and workload characteristics justify the overhead. Zero-copy is powerful, but not universally optimal; understanding system behavior is crucial for performance engineering.

12. AI Usage Declaration

Generative AI tools were used to obtain conceptual hints and explanations related to socket programming. Some amount of code structure for Servers and Clients was also generated using AI. However, all code was reviewed, modified, and understood by me.

Declared AI-Assisted Components

- **Conceptual Understanding**
  - Clarification of TCP socket programming concepts, like *send()*/*recv()*, *sendmsg()*, and *MSG_ZEROCOPY*
  - High-level explanations of Linux network namespaces and how they are to be used for the purpose of this assignment
  - Interpretation of *perf* hardware counters (CPU cycles, cache misses, context switches)
- **Initial Code Scaffolding (Modified Afterwards)**
  - Code structural outlines for [Part A]
    - Multithreaded TCP servers
    - Basic client-server communication loops
  - Skeleton code ideas for: [Part A1, A2, A3]
    - A1: Two-copy server using *send()*
    - A2: One-copy server using *sendmsg()* with pre-registered buffer
    - A3: Zero-copy server using *sendmsg()* with *MSG_ZEROCOPY*
  - Overall structure for automation scripts and plotting logic [Part C, D]

Relevant Prompts:-

- Part A: "*How to design a multithreaded TCP server in C that accepts multiple concurrent clients using one thread per client"*
- Part A1: "*Give the code outline for a TCP server that repeatedly sends a message structure consisting of 8 heap-allocated buffers to a client using send()*"
- Part A2: "*How to modify the baseline two-copy server to implement one-copy using sendmsg() with pre-registered buffer*"
- Part A3: "*How to modify the one-copy server to implement zero-copy using sendmsg() with MSG_ZEROCOPY*"

- Part B: "*How to create different server and client namespaces on Linux, and what commands to run to measure throughput, latency and perf statistics of the client-server model*"
- Part C: "*Explain how to design a bash script that automatically compiles multiple C programs, runs experiments for different parameters, and stores results in CSV files without manual intervention*"
- Part D: "*Python code using matplotlib to generate line plots for performance data with labeled axes, legends, and titles*"

13. Github Repository

https://github.com/ishita282003/GRS_PA02