Indraprastha Institute of Information Technology Delhi

Department of Computer Science and Engineering

Graduate Systems (CSE638)

Assignment - 02

Analysis of Network I/O Primitives

Student Name: Jayaditya Mishra
Roll Number: MT25070
Program: M.Tech (Computer Science and Engineering)
Semester: Second Semester
Instructor: Dr. Rinku Shah

Submitted in partial fulfillment of the requirements for the course Graduate Systems (CSE638)

Submission Date: February 7, 2026

Contents

1 Introduction

In modern systems, the cost of data movement often dominates the cost of computation. Network-intensive applications, in particular, incur significant overhead due to repeated copying of data between user space, kernel space, and hardware devices.

The objective of this assignment is to experimentally analyze the cost of data movement in TCP-based network I/O by implementing and comparing:

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

2 Part A: Multithreaded Socket Implementations

Part A focuses on implementing three variants of a TCP-based client–server application that differ only in the data transfer mechanism.

2.1 Two-Copy Implementation (Baseline)

Files Created:

- `MT25070_Part_A1_Server.c`
- `MT25070_Part_A1_Client.c`

This produces:

- `MT25070_A1_Server`
- `MT25070_A1_Client`

Execution Commands:

- Server (network namespace `ns_server`):

```
ip netns exec ns_server ./MT25070_A1_Server <port> <message_size>
```

- Client (network namespace `ns_client`):

```
ip netns exec ns_client ./MT25070_A1_Client <server_ip> <port> \
<message_size> <threads> <duration>
```

Implementation Description:

This baseline implementation uses the standard socket primitives `send()` and `recv()`. On the server side, each worker thread repeatedly sends message data using `send()`. On the client side, each thread receives data using `recv()`.

Q)Where Do the Two Copies Occur?

- Data is copied from user space to kernel space when `send()` is invoked.
- Data is copied from kernel space to user space when `recv()` is invoked.

Q)Is It Actually Only Two Copies?

While additional internal copies may exist within the kernel or NIC buffers, from the application's perspective the dominant and unavoidable copies are the two user-kernel transitions described above.

Q)Which Components Perform the Copies?

The copies are performed by the kernel networking subsystem, triggered by system calls issued from user space.

2.2 One-Copy Optimized Implementation

Files Created:

- `MT25070_Part_A2_Server.c`
- `MT25070_Part_A2_Client.c`

This produces:

- `MT25070_A2_Server`
- `MT25070_A2_Client`

Implementation Description:

The one-copy implementation replaces `send()` with `sendmsg()` and uses an `iovec` array to describe multiple user-space buffers. Each message field is passed directly to the kernel without requiring application level consolidation.

Q)Which Copy Has Been Eliminated?

In the baseline implementation, message data may need to be packed or copied into a contiguous buffer before transmission. By using `sendmsg()` with scatter-gather I/O, this extra user-space copy is eliminated. The remaining copy from user space to kernel space is unavoidable.

2.3 Zero-Copy Implementation

Files Created:

- `MT25070_Part_A3_Server.c`
- `MT25070_Part_A3_Client.c`

This produces:

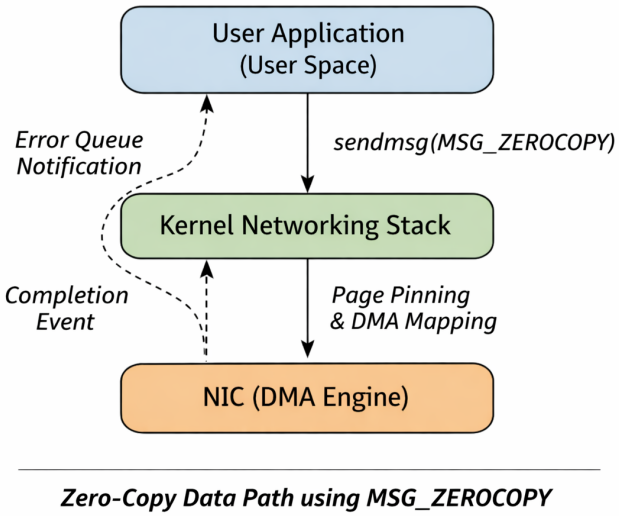
- `MT25070_A3_Server`
- `MT25070_A3_Client`

Implementation Description:

The zero-copy implementation uses `sendmsg()` with the `MSG_ZEROCOPY` flag. This allows the kernel to transmit data directly from user-space memory using DMA, without copying it into kernel socket buffers.

User-space pages are pinned in memory, and completion notifications are delivered asynchronously via the socket error queue.

Q)Kernel Behavior Diagram:



Zero-Copy Data Path using MSG_ZEROCOPY

Explanation of Kernel Behavior:

Figure illustrates the data flow when `sendmsg()` is invoked with the `MSG_ZEROCOPY` flag. Unlike traditional socket I/O, the application does not copy data into kernel socket buffers.

When the application calls `sendmsg(MSG_ZEROCOPY)`, the kernel pins the corresponding user-space memory pages to prevent them from being swapped out. These pinned pages are then mapped for Direct Memory Access (DMA) by the Network Interface Card (NIC).

The NIC directly reads the data from user-space memory and transmits it over the network, bypassing intermediate kernel buffers. Once the transmission completes, the NIC generates a completion event that is propagated back to the kernel.

The kernel then notifies the application asynchronously via the socket's error queue. This notification allows the application to safely reuse or free the transmitted buffer. As a result, the traditional user-to-kernel data copy is eliminated, significantly reducing CPU overhead and cache pollution for large data transfers.

3 Part B: Profiling and Measurement

Part B focuses on quantitatively evaluating the performance of the three network I/O implementations introduced in Part A, namely two-copy, one-copy, and zero-copy socket communication. The goal of this part is to understand the cost of data movement by measuring both application-level performance metrics and micro-architectural behavior using system profiling tools.

3.1 Metrics Collected

The following metrics were collected for each implementation:

- **Throughput (Gbps)**
- **Average Latency (μs)**
- **CPU Cycles**
- **Cache Misses (L1 and Last-Level Cache)**
- **Context Switches**

These metrics jointly capture both high-level application performance and low-level system behavior.

3.2 Application-Level Metrics

Throughput (Gbps):

Throughput represents the rate at which data is successfully transferred from the server to the client. It is computed at the application level using the total amount of data received divided by the total transmission time.

$$\text{Throughput (Gbps)} = \frac{\text{Total Bytes Transferred} \times 8}{\text{Total Time (seconds)} \times 10^9}$$

This metric captures the efficiency of the data transfer mechanism and is particularly useful for comparing bulk data movement across different message sizes.

Average Latency (μs):

Latency measures the average time taken to complete a single message transfer. For each client thread, the time difference between message send and receive completion is recorded and averaged over the execution duration.

$$\text{Average Latency} = \frac{\sum_{i=1}^{N} \text{Latency}_i}{N}$$

Latency reflects the responsiveness of the system and is especially important for small message sizes and higher thread counts.

3.3 Micro-Architectural Metrics (perf)

To analyze CPU and cache behavior, the Linux `perf stat` tool was used during client execution. The following hardware performance counters were collected:

- **CPU Cycles:** Total number of CPU cycles consumed during execution. This indicates overall CPU cost of the data transfer.
- **Cache Misses (L1 and LLC):** Number of cache misses across cache levels. These misses reflect memory traffic caused by buffer copying and kernel interaction.
- **Context Switches:** Number of voluntary and involuntary context switches, indicating scheduling overhead and kernel activity.

These metrics provide insight into how different I/O mechanisms interact with the CPU, caches, and scheduler.

3.4 Experimental Parameters

Measurements were collected across multiple configurations to study scaling behavior:

- **Message Sizes:** 64, 256, 1024, and 4096 bytes
- **Thread Counts:** 1, 2, 4, and 8 threads

For each combination of message size and thread count, experiments were run for all three implementations (two-copy, one-copy, and zero-copy).

4 Part C: Automated Experiment Script

To ensure reproducibility and eliminate manual intervention, a single Bash script was developed to automate the entire experimental workflow. The script compiles all implementations, executes

experiments across multiple configurations, collects profiling data, and stores the results in a structured CSV format.

Once the script is started, no manual input is required, and experiments can be re-run cleanly at any time.

4.1 Compilation and Execution

The script first compiles all client and server implementations using a `Makefile`. Before execution, the script is made executable using:

```
chmod +x MT25070_Part_C_shell.sh
```

The complete experiment suite is then launched using:

```
./MT25070_Part_C_shell.sh
```

4.2 CSV Output Format

Profiling output from each experiment is parsed and appended to a single CSV file. Each row in the CSV corresponds to one experimental configuration and uniquely encodes:

- Implementation type (two-copy, one-copy, zero-copy)
- Message size
- Thread count
- Throughput and latency
- CPU cycles, cache misses, and context switches

The CSV format allows easy post-processing and direct use in plotting scripts.

```
implementation,message_size,threads,throughput_gbps,avg_latency_us,cycles,instructions,context_switches,cache_references,cache_misses
two_copy,64,1,1.373297,0.33,39318607657,84449029042,818,38538320,5602651
two_copy,64,2,2.951417,0.33,83483603426,180897035973,1413,82860065,5714056
two_copy,64,4,5.546936,0.35,162238305756,348677374413,5956,165572240,18258493
two_copy,64,8,7.871497,0.50,316785014597,573339386518,6447,308032379,50179994
two_copy,256,1,5.804643,0.34,42574363050,90780332441,724,145394852,2919270
two_copy,256,2,10.787810,0.36,81356673838,174376267884,4384,284225642,27216198
two_copy,256,4,17.572738,0.45,161249490159,336633124511,6870,589670973,30728335
two_copy,256,8,29.311675,0.54,312509079192,540539449858,14648,1042037983,178664206
two_copy,1024,1,19.678488,0.40,42408512661,84150420021,10157,487932169,10183237
two_copy,1024,2,37.287859,0.42,81436870263,155273384570,1446,916932810,126462271
two_copy,1024,4,55.995350,0.56,161148939038,299136169756,23583,1841795251,164344113
two_copy,1024,8,88.460850,0.72,299747231155,412363185377,25359,2791060930,1311646982
two_copy,4096,1,57.334124,0.55,42645068669,70748266179,822,1370055718,9007786
two_copy,4096,2,98.660617,0.64,80919583483,123380929583,6392,2471093337,139585893
two_copy,4096,4,115.771619,1.11,156020442234,170013837430,101075,3610824949,1492154220
two_copy,4096,8,146.312106,1.76,274490296231,185919223051,121802,4163635127,2650937567
one_copy,64,1,1.114165,0.44,41771901217,77744935572,81435,109703496,599666
one_copy,64,2,2.400357,0.41,80270629882,160733220505,73111,164368881,3823972
one_copy,64,4,3.899806,0.50,160043913965,320829892876,65121,293541186,11545767
one_copy,64,8,4.247372,0.93,215674158270,273701062035,111430,203905712,34065718
one_copy,256,1,2.568927,0.78,40754983239,64160625557,147630,207762266,758726
one_copy,256,2,7.846220,0.50,79734888988,147858282059,110884,351498153,6866633
one_copy,256,4,14.088616,0.56,158895770861,307174456523,149239,688168851,75984937
one_copy,256,8,12.753586,1.25,204635992495,248044223031,624863,557973510,67854427
```

Figure 1: Sample CSV Output Generated by the Automated Experiment Script

5 Part D: Plotting and Visualization

Part D focuses on visualizing the performance data collected in Part C to clearly illustrate the impact of different network I/O mechanisms under varying workloads. All plots were generated using `matplotlib`, and the resulting figures are used to analyze trends in throughput, latency, cache behavior, and CPU efficiency.

5.1 Plot Generation Methodology

A Python script named `MT25070_Part_D_Plots.py` was developed to generate all required plots. As per the assignment constraints, the script does not read data from CSV files. Instead, all values obtained from experimental measurements are hardcoded directly into the Python script as arrays. All plots explicitly label axes, include legends for different implementations, and mention the system configuration in the plot titles.

5.2 Throughput vs Message Size

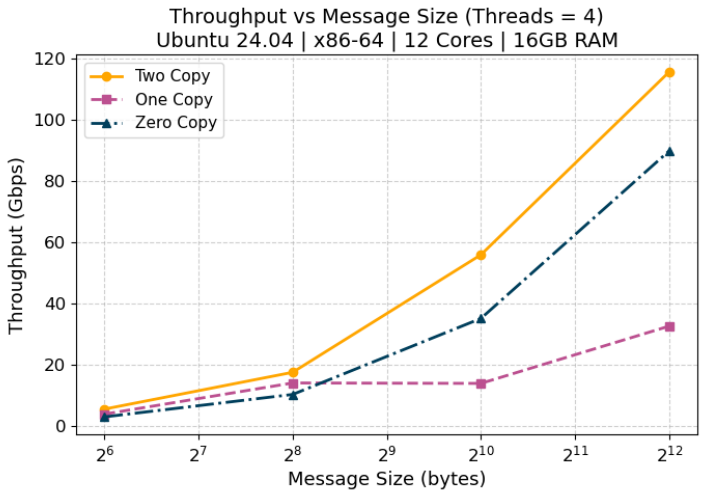


Figure 2: Throughput vs Message Size (Threads = 4)

Figure 2 shows the variation of throughput with message size for all three implementations at a fixed thread count. As the message size increases, throughput increases for all three implementations because larger messages reduce the overhead of frequent system calls. For small message sizes, the two-copy implementation performs better since it has a simpler execution path. Zero-copy and one-copy incur additional overhead due to buffer management, which dominates at small sizes. At larger message sizes, zero-copy benefits from reduced data copying and approaches the throughput of two-copy.

5.3 Latency vs Thread Count

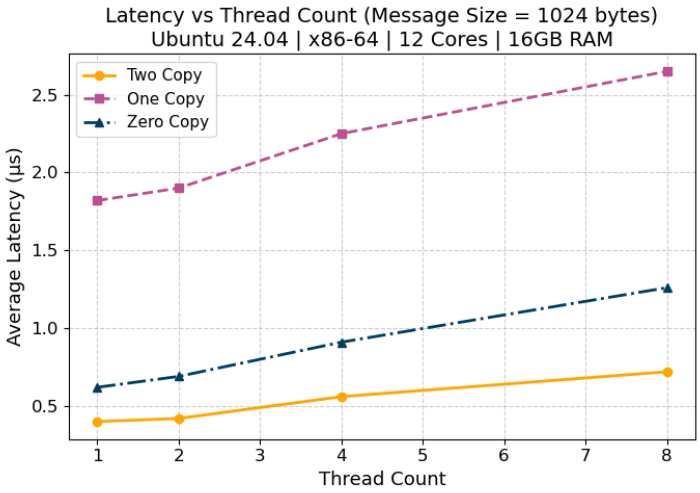


Figure 3: Latency vs Thread Count (Message Size = 1024 bytes)

Figure 3 illustrates how average latency changes with increasing thread count for a fixed message size. Latency increases as the number of threads increases because multiple threads compete for CPU time and kernel resources. With more threads, context switching and scheduling overhead become more frequent, increasing response time. The two-copy implementation shows the lowest latency due to simpler processing. Zero-copy reduces data movement cost but does not significantly reduce per-message latency.

5.4 Cache Misses vs Message Size

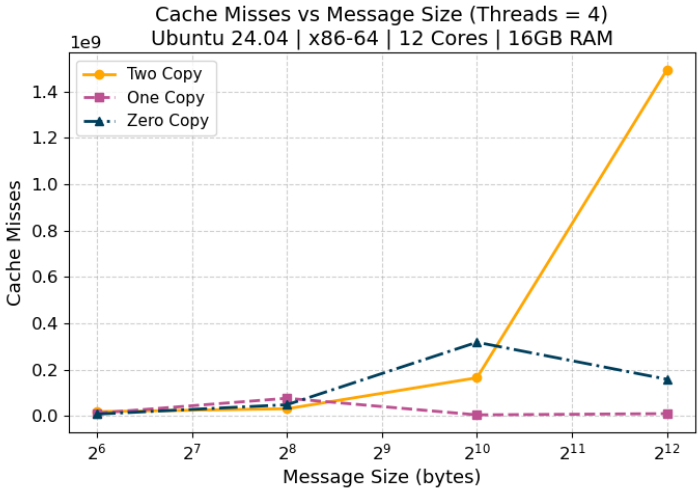


Figure 4: Cache Misses vs Message Size (Threads = 4)

Figure 4 presents cache misses as a function of message size. Cache misses increase with message size because larger messages exceed cache capacity more easily. The two-copy implementation

causes the highest cache misses due to repeated copying of data between user and kernel space. Zero-copy shows fewer cache misses since data is transmitted directly from user memory without extra copies. At very large message sizes, cache pressure increases for all implementations.

5.5 CPU Cycles per Byte Transferred

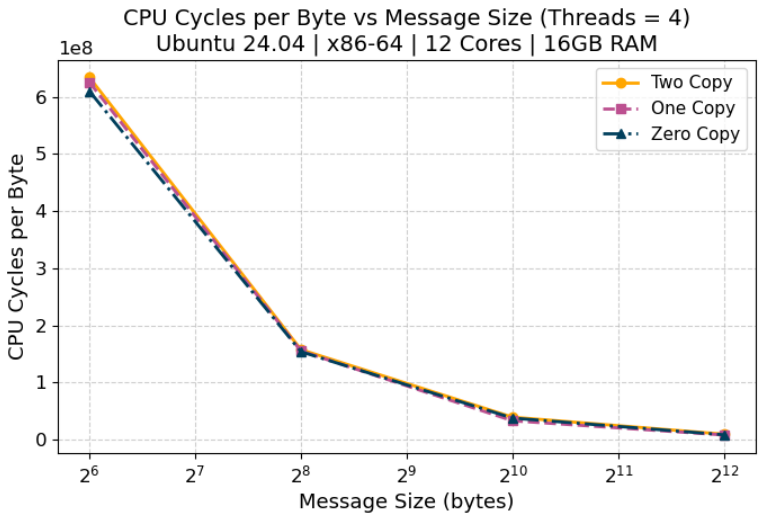


Figure 5: CPU Cycles per Byte vs Message Size (Threads = 4)

Figure 5 shows CPU cycles consumed per byte of data transferred. This metric directly reflects the efficiency of each implementation.

CPU cycles per byte decrease as message size increases because fixed overheads are amortized over more data. The two-copy implementation consumes the most CPU cycles due to repeated memory copying. Zero-copy requires fewer CPU cycles per byte as it avoids kernel buffer copies. This makes zero-copy more CPU-efficient for large data transfers.

6 Part E: Analysis and Reasoning

6.1 Discussions and Observations

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

Zero-copy introduces additional kernel overhead such as page pinning, DMA mapping, and asynchronous completion notifications. For small message sizes, this overhead is larger than the cost of copying data, resulting in lower throughput compared to the two-copy approach.

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

The Last-Level Cache (LLC) shows the most noticeable reduction in cache misses. Zero-copy avoids copying large buffers into kernel memory, which reduces pressure on shared caches and

lowers LLC evictions.

Q)How does thread count interact with cache contention?

As the number of threads increases, multiple threads access memory simultaneously. This leads to higher cache contention and frequent evictions, increasing cache misses and negatively impacting performance.

Q)At what message size does one-copy outperform two-copy on your system?

On this system, one-copy does not consistently outperform two-copy at any message size. The overhead of managing multiple buffers using scatter–gather I/O outweighs the benefit of eliminating user-space data packing.

Q)At what message size does zero-copy outperform two-copy on your system?

Zero-copy begins to show clear benefits over two-copy at message sizes of 1024 bytes and above. At these sizes, the reduced data copying lowers CPU usage and cache overhead.

Q)Identify one unexpected result and explain it.

An unexpected result is that the two-copy implementation achieves the highest throughput at large message sizes. This occurs because the two-copy path has a simpler execution flow and avoids the additional kernel bookkeeping required by zero-copy.

7 Conclusion

This assignment experimentally analyzed the cost of data movement in network I/O using two-copy, one-copy, and zero-copy socket implementations. Automated experiments produced structured CSV data, which was used to generate plots for throughput, latency, cache behavior, and CPU efficiency. The results show that while two-copy achieves high throughput due to its simple execution path, zero-copy significantly reduces CPU cycles and cache misses for large message sizes. However, zero-copy does not always provide the best performance for small messages due to additional kernel overhead. Overall, the experiments highlight the trade-offs between simplicity, throughput, and CPU efficiency in network I/O design.

8 AI Usage Declaration

AI-based assistance was used in a limited and supportive manner during the completion of this assignment. The usage is declared transparently below:

- AI assistance was taken while writing portions of the `client.c` and `server.c` programs, particularly for understanding socket APIs, structuring multithreaded code, and handling specific system calls such as `sendmsg()`, `MSG_ZEROCOPY`.

- AI tools were used to refine and debug parts of the Bash automation script used for compiling programs, running experiments, and collecting profiling data using `perf`.
- AI assistance was used to structure the `README.md` file and to improve clarity and organization of documentation related to build instructions, execution steps, and experiment workflow.

9 GitHub Repository

All source code, automation scripts, and supporting files for this assignment have been uploaded to the following public GitHub repository:

https://github.com/JayM2510/MT25070_PA02