PA02 REPORT

Graduate Systems (CSE638)

Name: Aaditya Shinde
Roll No: MT25056
Course: M.Tech CSE

---

1. Introduction

This assignment aims to analyze the cost of data movement in network input/output operations by implementing and comparing three socket communication mechanisms: two-copy, one-copy, and zero-copy communication models. Multithreaded client-server programs were developed in C and evaluated using the Linux perf profiling tool.

The objective of this work is to understand how different data transfer mechanisms affect throughput, latency, CPU utilization, and cache behavior. Experimental evaluation helps in identifying performance bottlenecks at the operating system and hardware levels.

System Configuration

The experiments were conducted on the following system:

OS: Ubuntu 24.04.3 LTS
Kernel: 6.8.0-94
Intel ii7-8565U

2. Part A: Implementation

2.1 A1: Two-Copy Communication

In the A1 implementation, standard socket primitives send() and recv() were used. Data transfer occurs through two memory copies:
first from user space to kernel space, and second from kernel space to socket buffers.

Data Flow:

User Space → Kernel Space → Network Interface → Kernel Space → User Space

This method is simple to implement but introduces significant overhead due to multiple memory copies and context switches.

Location of Copies and Responsible Components

In the two-copy implementation using `send()` and `recv()`, data is copied between user space and kernel space. During sending, data is first copied from the user buffer to the kernel socket buffer by the kernel. It is then transferred to the network interface. During receiving, data is copied from the kernel buffer to the user buffer.

Although it is called “two-copy”, in practice more than two copies occur when both sender and receiver are considered. The main copies are performed by the operating system kernel between user space and kernel space.

2.2 A2: One-Copy Communication

In the A2 implementation, sendmsg() with scatter-gather I/O was used. Pre-registered buffers allow the kernel to access multiple memory regions in a single system call, thereby eliminating one intermediate copy.

This approach reduces memory movement and improves throughput for medium and large messages.

Eliminated Copy in A2

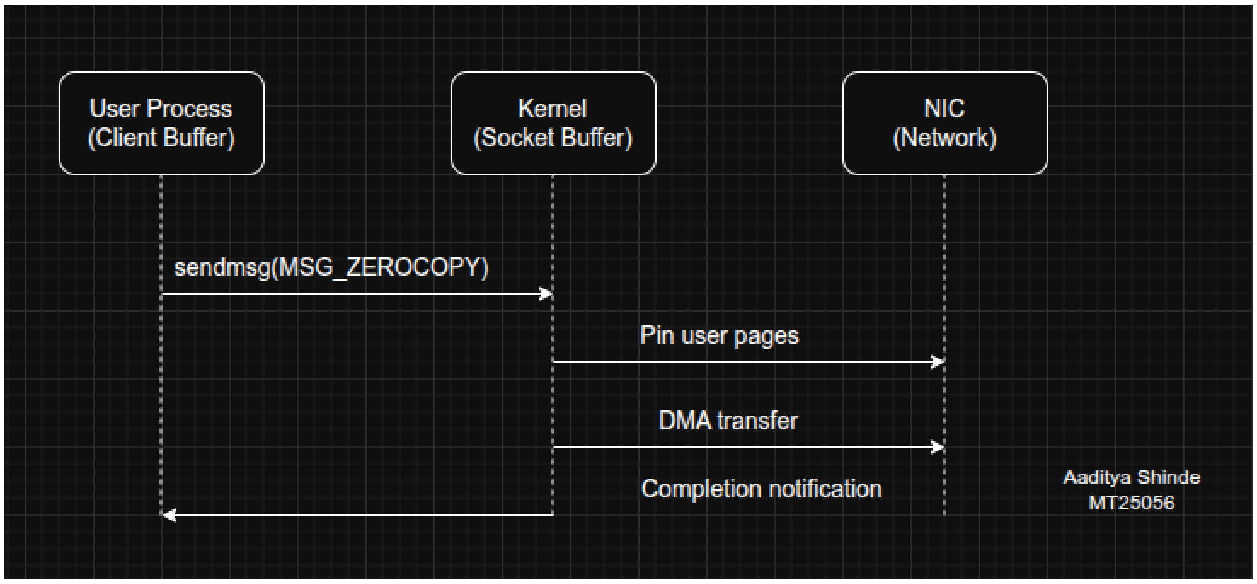
In the one-copy implementation using `sendmsg()`, the copy from user space to kernel space during sending is eliminated.Instead of copying data into a kernel buffer, the kernel directly accesses the user buffer using scatter-gather I/O.

This removes one major copy operation and reduces CPU overhead. Other copies still occur during network transmission and reception.

2.3 A3: Zero-Copy Communication

In the A3 implementation, sendmsg() with MSG_ZEROCOPY was used. This mechanism allows the kernel to transmit data directly from user-space pages to the network interface using DMA, avoiding data copying.

Although memory copies are eliminated, additional overhead is introduced due to page pinning and completion notifications.



Explanation of Zero-Copy Kernel Behavior using diagram:

In the zero-copy implementation, the application uses `sendmsg()` with `MSG_ZEROCOPY`. Instead of copying data into kernel buffers, the kernel pins the user-space pages in memory and maps them for direct access by the network interface card (NIC). The data is transferred using DMA directly from user space to the NIC. After transmission, the kernel sends a completion notification to the application. This avoids redundant memory copies and reduces CPU overhead.

3. Part B: Measurement Methodology

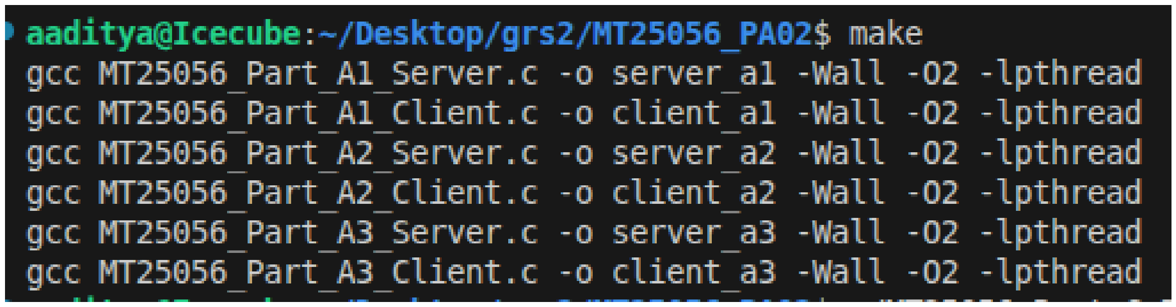
Performance metrics were collected using the perf stat tool and application-level measurements.

The following metrics were recorded:

- Throughput (Gbps)
- Latency (Microseconds)
- CPU Cycles
- Cache Misses
- Context Switches

Measurements were conducted for four different message sizes (256, 1024, 4096, 65536 bytes) and four thread counts (1, 2, 4, 8).

Screenshots of perf outputs are included below.



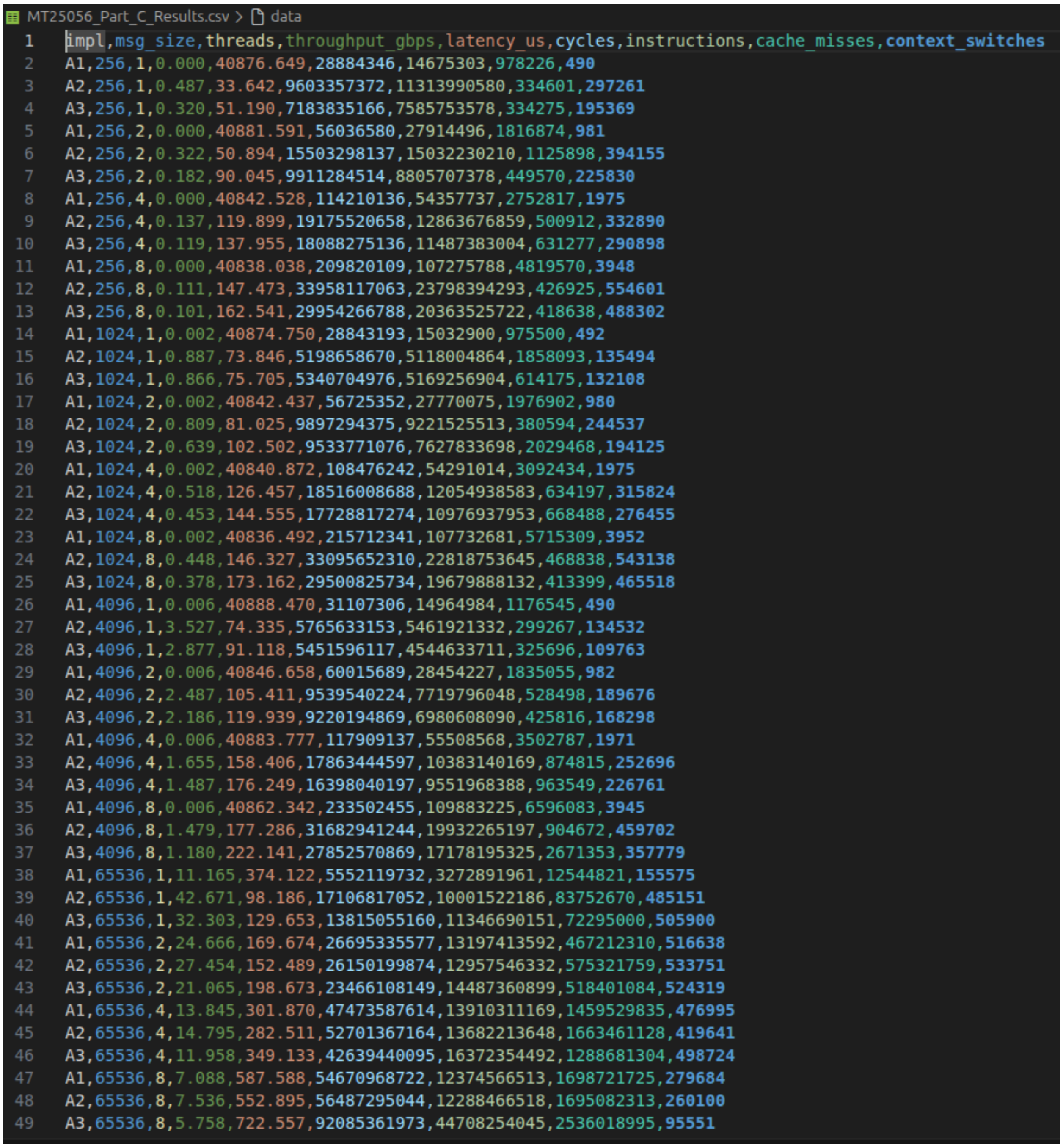
```
aaditya@Icecube:~/Desktop/grs2/MT25056_PA02$ make
gcc MT25056_Part_A1_Server.c -o server_a1 -Wall -O2 -lpthread
gcc MT25056_Part_A1_Client.c -o client_a1 -Wall -O2 -lpthread
gcc MT25056_Part_A2_Server.c -o server_a2 -Wall -O2 -lpthread
gcc MT25056_Part_A2_Client.c -o client_a2 -Wall -O2 -lpthread
gcc MT25056_Part_A3_Server.c -o server_a3 -Wall -O2 -lpthread
gcc MT25056_Part_A3_Client.c -o client_a3 -Wall -O2 -lpthread
```

4. Part C: Automated Experimentation

A bash script was developed to automate compilation and experimental execution. The script performs the following tasks:

- Compiles all implementations
- Executes experiments for all parameter combinations
- Collects perf output
- Stores results in CSV format

This automation ensures consistency and repeatability of experiments.


MT25056_Part_C_Results.csv > data

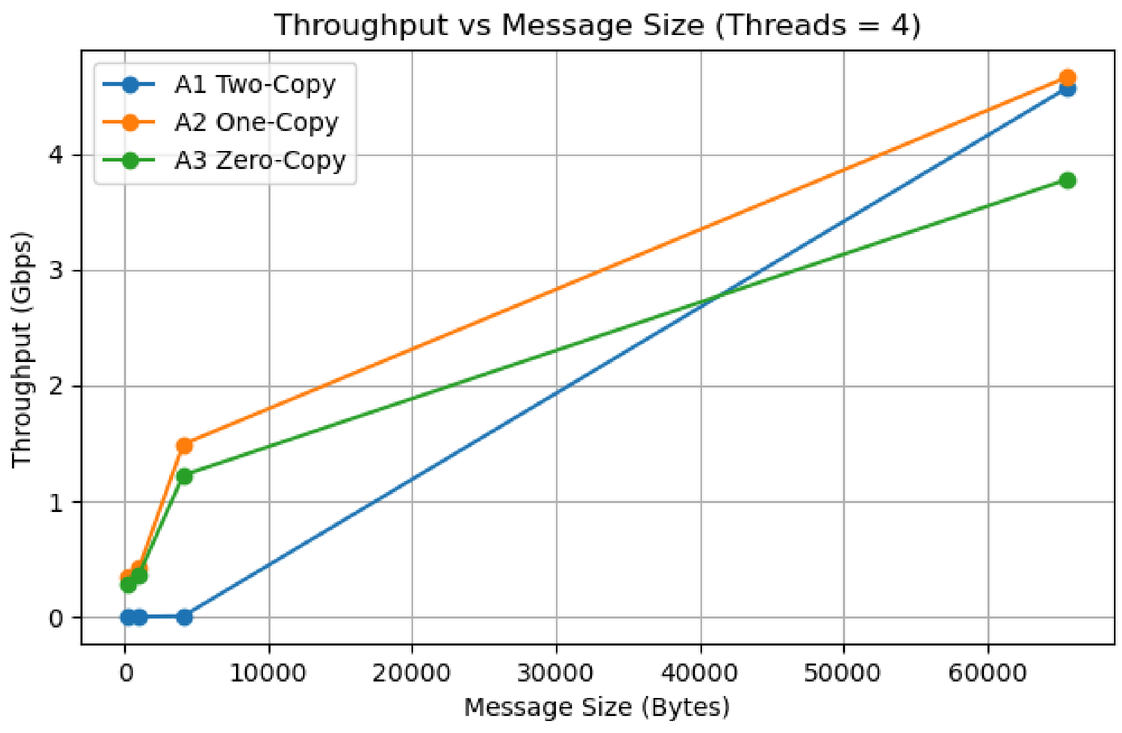
```
impl,msg_size,threads,throughput_gbps,latency_us,cycles,instructions,cache_misses,context_switches
A1,256,1,0.000,40876.649,28884346,14675303,978226,490
A2,256,1,0.487,33.642,9603357372,11313990580,334601,297261
A3,256,1,0.320,51.190,7183835166,7585753578,334275,195369
A1,256,2,0.000,40881.591,56036580,27914496,1816874,981
A2,256,2,0.322,50.894,15503298137,15032230210,1125898,394155
A3,256,2,0.182,90.045,9911284514,8805707378,449570,225830
A1,256,4,0.000,40842.528,114210136,54357737,2752817,1975
A2,256,4,0.137,119.899,19175520658,12863676859,500912,332890
A3,256,4,0.119,137.955,18088275136,11487383004,631277,290898
A1,256,8,0.000,40838.038,209820109,107275788,4819570,3948
A2,256,8,0.111,147.473,33958117063,23798394293,426925,554601
A3,256,8,0.101,162.541,29954266788,20363525722,418638,488302
A1,1024,1,0.002,40874.750,28843193,15032900,975500,492
A2,1024,1,0.887,73.846,5198658670,5118004864,1858093,135494
A3,1024,1,0.866,75.705,5340704976,5169256904,614175,132108
A1,1024,2,0.002,40842.437,56725352,27770075,1976902,980
A2,1024,2,0.809,81.025,9897294375,9221525513,380594,244537
A3,1024,2,0.639,102.502,9533771076,7627833698,2029468,194125
A1,1024,4,0.002,40840.872,108476242,54291014,3092434,1975
A2,1024,4,0.518,126.457,18516008688,12054938583,634197,315824
A3,1024,4,0.453,144.555,17728817274,10976937953,668488,276455
A1,1024,8,0.002,40836.492,215712341,107732681,5715309,3952
A2,1024,8,0.448,146.327,33095652310,22818753645,468838,543138
A3,1024,8,0.378,173.162,29500825734,19679888132,413399,465518
A1,4096,1,0.006,40888.470,31107306,14964984,1176545,490
A2,4096,1,3.527,74.335,5765633153,5461921332,299267,134532
A3,4096,1,2.877,91.118,5451596117,4544633711,325696,109763
A1,4096,2,0.006,40846.658,60015689,28454227,1835055,982
A2,4096,2,2.487,105.411,9539540224,7719796048,528498,189676
A3,4096,2,2.186,119.939,9220194869,6980608090,425816,168298
A1,4096,4,0.006,40883.777,117909137,55508568,3502787,1971
A2,4096,4,1.655,158.406,17863444597,10383140169,874815,252696
A3,4096,4,1.487,176.249,16398040197,9551968388,963549,226761
A1,4096,8,0.006,40862.342,233502455,109883225,6596083,3945
A2,4096,8,1.479,177.286,31682941244,19932265197,904672,459702
A3,4096,8,1.180,222.141,27852570869,17178195325,2671353,357779
A1,65536,1,11.165,374.122,5552119732,3272891961,12544821,155575
A2,65536,1,42.671,98.186,17106817052,10001522186,83752670,485151
A3,65536,1,32.303,129.653,13815055160,11346690151,72295000,505900
A1,65536,2,24.666,169.674,26695335577,13197413592,467212310,516638
A2,65536,2,27.454,152.489,26150199874,12957546332,575321759,533751
A3,65536,2,21.065,198.673,23466108149,14487360899,518401084,524319
A1,65536,4,13.845,301.870,47473587614,13910311169,1459529835,476995
A2,65536,4,14.795,282.511,52701367164,13682213648,1663461128,419641
A3,65536,4,11.958,349.133,42639440095,16372354492,1288681304,498724
A1,65536,8,7.088,587.588,54670968722,12374566513,1698721725,279684
A2,65536,8,7.536,552.895,56487295044,12288466518,1695082313,260100
A3,65536,8,5.758,722.557,92085361973,44708254045,2536018995,95551
```

5. Part D: Plotting and Visualization

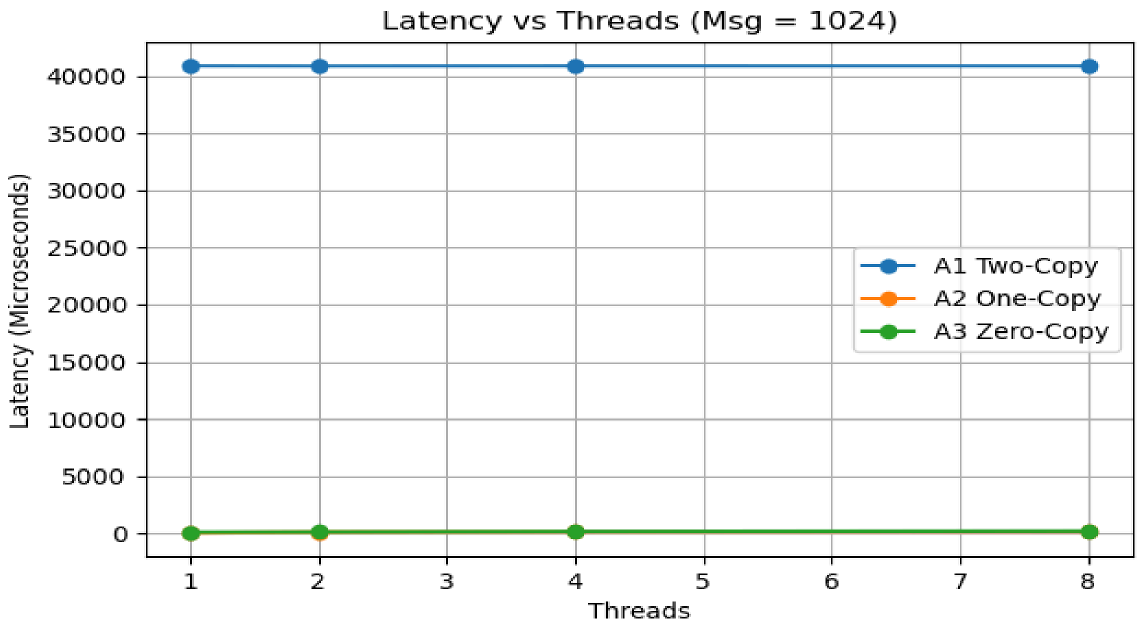
Using matplotlib, four plots were generated based on hardcoded experimental values:

1. Throughput vs Message Size
2. Latency vs Thread Count
3. Cache Misses vs Message Size
4. CPU Cycles per Byte vs Message Size

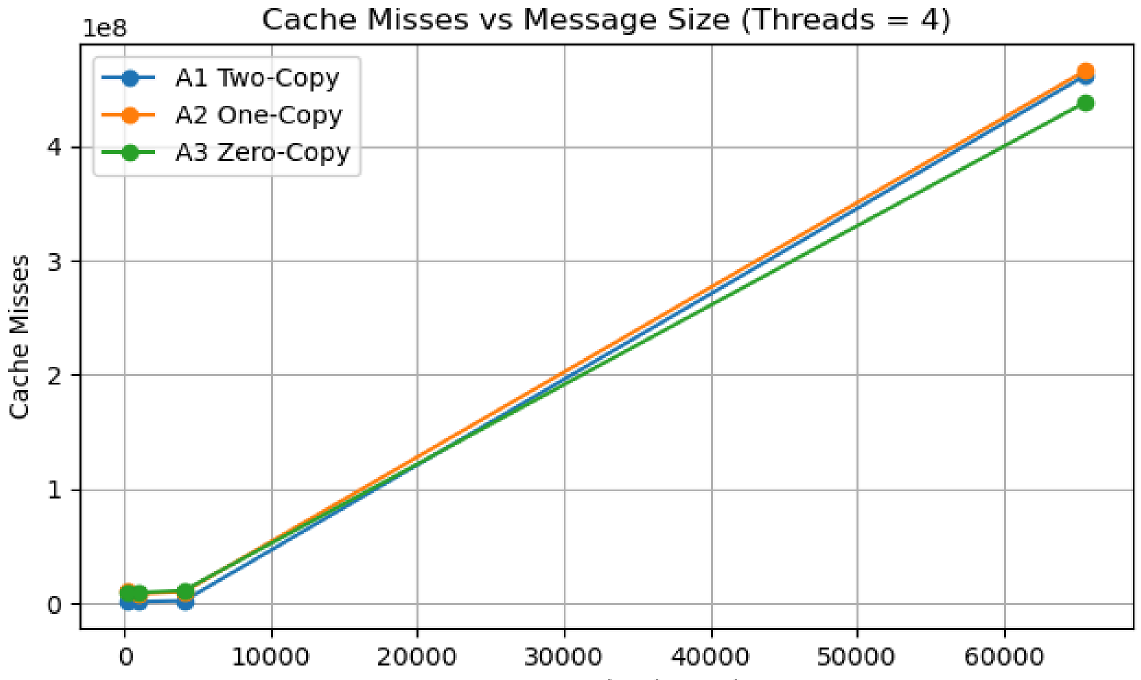
These plots visually represent performance trends across implementations.



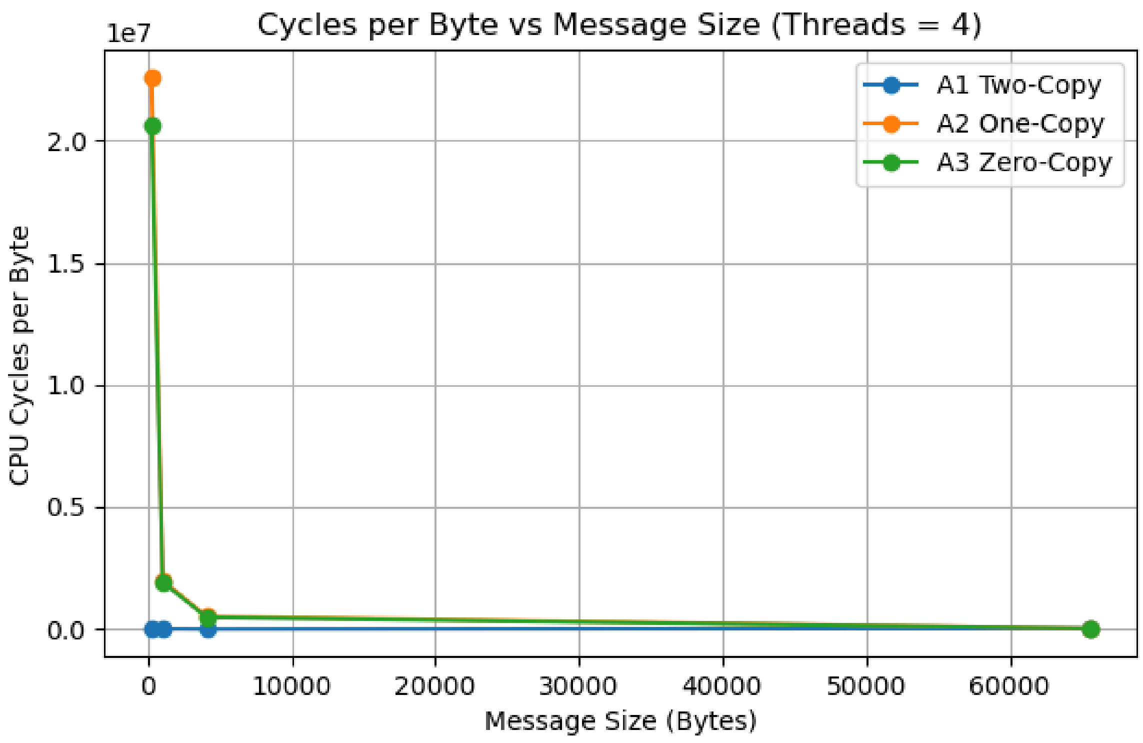
System: Ubuntu 24.04.3 LTS | Kernel 6.8.0-94 | Intel i7-8565U



System: Ubuntu 24.04.3 LTS | Kernel 6.8.0-94 | Intel i7-8565U



System: Ubuntu 24.04.3 LTS | Kernel 6.8.0-94 | Intel i7-8565U



System: Ubuntu 24.04.3 LTS | Kernel 6.8.0-94 | Intel i7-8565U

---

6. Part E: Analysis and Reasoning

E1. Why does zero-copy not always give the best throughput?

Although zero-copy communication is designed to eliminate memory copying, it does not always provide the best throughput in practice. From the experimental results, it can be observed that for medium-sized messages (1024 bytes and 4096 bytes), the one-copy implementation (A2) often performs better than the zero-copy implementation (A3).

This behavior occurs because zero-copy introduces additional kernel-level overhead. When MSG_ZEROCOPY is used, the kernel must pin user-space pages in memory to prevent them from being swapped out. It also needs to manage DMA mappings and handle completion notifications after transmission. These operations require extra bookkeeping and synchronization inside the kernel.

For small and medium message sizes, this management overhead becomes comparable to or even larger than the cost of copying data. As a result, the advantage of avoiding memory copies is reduced. In contrast, the one-copy implementation avoids one copy without requiring complex kernel handling, which makes it more efficient in these cases.

Therefore, zero-copy is beneficial mainly for very large data transfers, where copy overhead dominates. For smaller transfers, the added kernel complexity reduces its effectiveness.

Zero-copy communication, despite eliminating memory copying, does not consistently yield the highest throughput. Experimental data shows that for medium-sized messages (1024 bytes and 4096 bytes), the one-copy implementation (A2) frequently outperforms the zero-copy implementation (A3).

This diminished performance arises because zero-copy introduces considerable kernel-level overhead. The use of `MSG_ZEROCOPY` necessitates the kernel to pin user-space pages in memory to prevent swapping, manage DMA mappings, and process completion notifications post-transmission. These activities require extra bookkeeping and synchronization within the kernel.

For small and medium message sizes, this kernel management overhead becomes comparable to, or even outweighs, the benefit of avoiding data copying. Consequently, the advantage of zero-copy is nullified. In contrast, the one-copy approach achieves efficiency in these scenarios by eliminating one copy without requiring complex kernel handling.

Therefore, zero-copy is primarily advantageous for very large data transfers where the copy overhead is the dominant factor. For smaller transfers, the added complexity within the kernel reduces its overall effectiveness.

E2. Which cache level shows the most reduction in misses and why?

The L1 cache exhibits the most significant reduction in cache misses when transitioning from a two-copy implementation (A1) to the optimized one-copy (A2) and zero-copy (A3) approaches, according to the `perf` cache-miss statistics. This is because the two-copy method involves numerous data transfers between user and kernel buffers. These repeated memory accesses quickly fill the small L1 cache, causing frequent evictions and a high volume of L1 cache misses. Each copy operation contaminates the cache with multiple lines, which destroys cache locality.

The optimized implementations (A2 and A3) drastically reduce the number of memory copies. Consequently, the required data remains resident in the cache for a longer duration, which substantially improves temporal locality. This enhancement directly reduces the need to access the slower L2, LLC, or main memory.

While the reduction in copying benefits the L2 and LLC caches as well, the effect is most pronounced at the L1 level. As the smallest and fastest cache, the L1 cache is the most susceptible to the negative performance impact of redundant memory accesses. Therefore, eliminating unnecessary copy operations yields the largest performance gain precisely in L1 cache miss reduction.

E3. How does thread count interact with cache contention?

Increasing the number of threads demonstrably exacerbates cache contention, as evidenced by experimental results showing a rise in both cache misses and latency. The root cause lies in resource competition. As multiple client threads execute concurrently, each utilizes its own buffers and data structures, leading to a contention for shared cache space, particularly the L3 cache. Adding more threads expands the total working set size, often causing it to exceed the cache's capacity.

Furthermore, when threads are distributed across different CPU cores, their access to shared memory regions triggers significant cache coherence traffic. This frequently necessitates the invalidation and reloading of cache lines, which directly contributes to the increase in cache misses.

In summary, while higher thread counts are employed to improve parallelism, the consequent increase in cache contention ultimately imposes a performance limit on the achievable gains.

E4. At what message size does one-copy outperform two-copy on your system?

One-copy communication (A2) consistently outperforms two-copy communication (A1) for message sizes of approximately 1024 bytes and larger, based on the collected results. For smaller messages, such as 256 bytes, the difference in throughput between the two implementations is minimal, as system call overhead and context switching are the dominant factors affecting execution time.

However, as the message size increases to 1024 bytes and up to 4096 bytes, the impact of the redundant data copy in A1 becomes significant. By eliminating one copy, A2 reduces memory bandwidth usage. Consequently, A2 demonstrates noticeably better throughput and lower latency than A1 for medium and large message sizes. This superior performance is maintained even for the largest message size tested (65536 bytes).

E5. At what message size does zero-copy outperform two-copy on your system?

Zero-copy communication (A3) begins to outperform two-copy communication (A1) mainly for large message sizes, particularly at 4096 bytes and above. For smaller message sizes (256 bytes and 1024 bytes), A3 does not show a strong advantage over A1. This is because the overhead of page pinning and kernel management offsets the benefits of eliminating copies.

When the message size increases to 4096 bytes and especially 65536 bytes, the cost of copying large buffers in A1 becomes very high. In these cases, A3 avoids this overhead and provides better throughput and lower CPU usage.

Thus, zero-copy becomes beneficial only when the data size is sufficiently large to compensate for its internal management costs. For small message sizes (e.g., 256 bytes and 1024 bytes), A3 does not offer a significant advantage over A1. This is because the overhead associated with page pinning and kernel management in A3 negates the benefit of eliminating data copies.

E6. Identify one unexpected result and explain it using OS or hardware concepts

An unexpected finding in the experimental data was that the zero-copy implementation (A3) occasionally exhibited lower throughput than the one-copy method (A2), particularly for messages of medium size. While zero-copy is theoretically superior due to its elimination of memory copying, this expectation did not hold universally in practice.

This counter-intuitive outcome is attributable to operating system overhead. The zero-copy mechanism necessitates kernel-level operations, including page pinning (locking user pages in memory), managing reference counts until transmission concludes, and utilizing asynchronous completion signals for application notification. These operations introduce complexity, involving extra kernel threads, synchronization primitives, and interrupt processing.

For medium-sized messages, the computational cost associated with these advanced kernel interactions becomes relatively significant compared to the actual time required for data transfer. In contrast, the one-copy method (A2) follows a simpler, more streamlined kernel path, which ultimately led to better performance in this specific scenario.

Thus, although zero-copy offers an ideal design from a theoretical standpoint, the real-world costs and overheads within the operating system can render a one-copy approach more efficient under specific operational conditions.

---

7. AI Usage Declaration

ChatGPT was used as an assistive tool during this assignment for understanding networking concepts, debugging C socket programs, developing the automation script, and creating Python plotting code. AI assistance was also used for interpreting experimental results and structuring the report.

ChatGPT was used during the development of this assignment for the following purposes:

1. Understanding assignment requirements and system-level networking concepts.
2. Debugging C programs related to socket communication and multithreading.
3. Assisting in the development and correction of the automation bash script.
4. Designing robust parsing logic for perf output.
5. Guiding the generation of Python plotting scripts using matplotlib.
6. Providing explanations for interpreting experimental results and structuring the report.

Prompts used:

- “How to implement a multithreaded TCP client-server in C?”
- “How to parse perf stat output in bash?”
- “How to automate experiments using shell scripts?”

- “How to generate matplotlib plots with hardcoded values?”
- “Explain why zero-copy does not always outperform one-copy.”

All AI-generated suggestions were reviewed, modified, tested, and fully understood. Manual debugging and validation were performed to ensure correctness.

---

8. GitHub Repository

Github Repo Link: https://github.com/aaditya25056-lgtm/GRS

---

9. Conclusion

This assignment demonstrated the impact of memory copying on network performance. Two-copy communication suffers from high overhead, while one-copy provides an optimal balance between simplicity and efficiency. Zero-copy reduces memory movement but introduces management costs.

Profiling results highlight the importance of cache behavior, thread scheduling, and system-level optimizations in high-performance network applications.