MT25033 Report

Palak Bafna

January 2026

Contents

1 Introduction

1.1 Goal:

- Measure and compare the cost of data movement between the user space and kernel space for the three different network I/O:
  - Two-Copy I/O using recv()\send()
  - One-Copy I/O using recvmsg()\sendmsg() with scatter-gather I/O
  - Zero-Copy I/O using sendmsg() with `MSG_ZEROCOPY`
- Profiled all 3 with ‘perf stat‘ and export CSV
  Collected: throughput, latency, CPU cycles, cache misses, cache references, context switches
- Automated the entire experiment pipeline, Loop over message sizes x thread counts x implementations
- Generated performance comparison 6 matplotlib plots from CSV data

1.2 File Map and Responsibilities:

```
MT25033_PA02-main/
|
|-- MT25033_Part_A_Common.h
|-- MT25033_Part_A1_Server.c
|-- MT25033_Part_A1_Client.c
|-- MT25033_Part_A2_Server.c
|-- MT25033_Part_A2_Client.c
|-- MT25033_Part_A3_Server.c
|-- MT25033_Part_A3_Client.c
|-- results/  (temporary folder created for output)
    |-- MT25033_Part_B_Combined.csv
    |-- MT25033_Part_B_ZeroCopy.csv
    |-- MT25033_Part_B_oneCopy.csv
    |-- MT25033_Part_B_twoCopy.csv
```

2 Experimental Setup

2.1 System Configuration:

- Compiler: GCC with -O2 -pthread
- OS: Linux (Ubuntu)
- Kernel: $\geq$ 4.14 (required for `MSG_ZEROCOPY`)
- Profiler: perf stat
- Network: Linux network namespaces with veth pair

2.2 Test Parameters

MessageSi	Threads	Throughpu	Latency_us	TotalBytes	CPUCycles	CyclesPerByte	CacheMisses	CacheRefs	ContextSwitches
1024	1	3.5323	1.88	4415393792	13026955089	2.9503	13733334	46525366	6343
1024	2	6.8617	1.95	8577098704	43553453594	5.0778	15916826	113621833	6744
1024	4	13.1355	2.02	16419398560	1.27431E+11	7.7609	6382050	245380594	195127
1024	8	18.7468	2.82	23433527104	2.78712E+11	11.8937	3474524	528537487	451722
4096	1	8.6851	3.56	10856402944	89561878	0.0082	12730	113264	68
4096	2	17.21	3.73	21512552448	20657102821	0.9602	7319779	37733383	136
4096	4	33.1476	3.64	41434447872	94983442861	2.2923	2870251	146354455	261
4096	8	49.0967	4.14	61370933744	2.73996E+11	4.4645	1504788	558703760	1436
65536	1	62.1994	6.69	77749288560	93383551	0.0012	32828	113266	77
65536	2	106.7423	7.83	1.33428E+11	4896662472	0.0366	17420574	39667714	137
65536	4	225.3312	7.46	2.81664E+11	1.012E+11	0.3592	19515724	130680943	300
65536	8	316.4052	11.17	3.95507E+11	2.69994E+11	0.6826	21526914	1351440094	15336
1048576	1	89.8767	7.23	1.12346E+11	30141970884	0.2682	27176207	104556144	4978
1048576	2	178.2624	6.85	2.22828E+11	10860020703	0.0487	101899908	224886710	5020
1048576	4	322.1586	9.27	4.02699E+11	1.25834E+11	0.3124	72894665	668499039	38810
1048576	8	178.7788	101.49	2.23482E+11	44551263438	0.1993	63129130	395725474	167944
4194304	1	81.3105	8.05	1.01638E+11	18257183543	0.1796	55603131	316112492	8983
4194304	2	168.9639	8.06	2.11205E+11	55957839287	0.2649	78950674	1146973982	12452
4194304	4	288.2761	9.86	3.6035E+11	1.10996E+11	0.308	214536871	2178907256	30621
4194304	8	195.9469	100.04	2.44946E+11	74085088705	0.3024	297966329	1089269367	94752
16777216	1	73.523	7.82	91903862936	32547214269	0.3541	332159191	675626506	3276
16777216	2	114.2395	11.88	1.42802E+11	52563218642	0.368	655994998	1123484064	19711
16777216	4	192.4292	13.99	2.40538E+11	1.13339E+11	0.4711	1337172518	2193790540	27829
16777216	8	203.0113	37.11	2.53794E+11	1.8819E+11	0.7415	1725191478	2526095002	48704

Figure 1: Part B ZeroCopy

MessageSize	Threads	Throughput_Gbps	Latency_us	TotalBytes	CPUCycles	CyclesPerByte	CacheMisses	CacheRefs	ContextSwitches
1024	1	5.3215	1.5	6651833344	41252434	0.0062	33066	138691	49
1024	2	10.5513	1.52	13189111808	2.3745E+10	1.8003	3049594	52172068	85
1024	4	19.6607	1.63	24575822848	2.173E+10	0.8841	5339764	51749529	109
1024	8	42.6367	1.51	53295856640	2.773E+11	5.203	39047269	1198003755	4290
4096	1	11.1706	2.31	13963243520	60724268	0.0043	41442	189479	43
4096	2	22.2771	2.39	27846383264	2.3549E+10	0.8456	111355302	266546978	107
4096	4	46.1647	2.37	57705876304	7.2336E+10	1.2535	6970268	179677766	168
4096	8	73.3163	2.84	91645386048	2.6509E+11	2.8925	4022978	785952197	2697
65536	1	79.8057	6.54	99757195264	29214114	0.0002	33320	90412	37
65536	2	161.0965	6.47	2.01371E+11	3.1762E+10	0.1577	19631692	79096578	97
65536	4	302.7634	6.89	3.78454E+11	2.8023E+10	0.074	6802022	38688239	118
65536	8	393.9304	11.76	4.92414E+11	2.6004E+11	0.528	6593761	2444128792	7058
1048576	1	100.7074	5.12	1.25884E+11	8139132	0	108766	206269	42
1048576	2	227.0832	4.93	2.83854E+11	179554476	0.0006	119330	2713744	77
1048576	4	419.4043	5.12	5.24256E+11	6.5192E+10	0.1243	58242729	1611655294	182
1048576	8	486.5475	9.53	6.08188E+11	2.7701E+11	0.4554	46435809	7121261844	8450
4194304	1	107.1344	5.33	1.33918E+11	35602422	0.0002	146349	1069220	45
4194304	2	196.3474	5.52	2.45434E+11	6967423163	0.0283	65730446	230147161	163
4194304	4	375.9845	5.7	4.69981E+11	1.2893E+11	0.2743	298611039	5149330332	657
4194304	8	376.6603	12.25	4.70826E+11	2.7337E+11	0.5806	2624518980	7200473211	6796
16777216	1	109.418	4.98	1.36772E+11	14461750	0.0001	163204	215777	55
16777216	2	181.5211	5.78	2.26901E+11	3944525023	0.0173	50965050	127576403	118
16777216	4	298.1605	6.99	3.72701E+11	1.3754E+11	0.369	1964000847	5051390745	385
16777216	8	300.4881	15.77	3.7561E+11	2.9443E+11	0.7838	3531875510	6971802066	9073

Figure 2: Part B oneCopy

MessageSi	Threads	Throughpu	Latency_us	TotalBytes	CPUCycles	CyclesPerByte	CacheMisses	CacheRefs	ContextSwitches
1024	1	4.9369	1.53	6171155456	103224897	0.0167	30877	281009	67
1024	2	11.1838	1.36	13979807744	18293978	0.0013	39505	93846	62
1024	4	20.4805	1.43	25600596992	25492818886	0.9957	1572902	54717836	115
1024	8	41.0125	1.52	51265582928	2.83111E+11	5.5224	14555763	1048768660	2002
4096	1	11.1564	2.12	13945458688	91550118	0.0065	44402	262139	42
4096	2	23.977	2.07	29971270480	158376211	0.0052	45623	379467	75
4096	4	45.4998	2.16	56874774528	47572617161	0.8364	10124636	123282974	128
4096	8	81.0574	2.7	1.01322E+11	2.79767E+11	2.7611	5800177	1188726688	2612
65536	1	76.6513	6.8	95814156288	94003065	0.0009	28891	143153	47
65536	2	151.4676	6.89	1.89335E+11	148256851	0.0007	56808	270768	83
65536	4	321.3809	6.49	4.01726E+11	61588614541	0.1533	11078762	124921123	310
65536	8	400.0821	10.96	5.00101E+11	2.67846E+11	0.5355	6417832	2382859458	7692
1048576	1	103.8437	5.15	1.29805E+11	49592876	0.0003	104142	244061	42
1048576	2	200.1731	5.18	2.50216E+11	11243176846	0.0449	17705337	173711299	90
1048576	4	403.4949	5.2	5.04369E+11	3286107075	0.0065	4628599	71700448	172
1048576	8	482.638	9.78	6.03298E+11	2.77603E+11	0.4601	25916237	7196258037	4113
4194304	1	105.8799	5.64	1.3235E+11	28041440	0.0002	270960	752420	56
4194304	2	203.4314	5.51	2.5429E+11	48081216856	0.189	130664297	1866252574	171
4194304	4	368.6199	5.81	4.60775E+11	1.43173E+11	0.3107	188481574	6184996702	869
4194304	8	378.3084	12.16	4.72886E+11	2.80143E+11	0.5924	2814066066	7272697138	10025
16777216	1	111.1669	5.47	1.38959E+11	88653526	0.0006	1079976	1968923	55
16777216	2	168.9028	6.23	2.11129E+11	34513265851	0.1634	518845475	1063305312	127
16777216	4	293.3486	7.11	3.66686E+11	1.21349E+11	0.3309	1783740936	4623701381	484
16777216	8	299.7336	15.84	3.74668E+11	2.97037E+11	0.7928	3368756680	6622757819	10282

Figure 3: Part B twoCopy

- Message Sizes: 1KB, 4KB, 64KB, 1MB, 4MB, 16MB
- Thread Counts: 1, 2, 4, 8
- Duration: 10 seconds per run
- Total Runs: 72

3 Overview of Implementation

3.1 Message Structure:

All implementations use a Message structure, which has eight individually allocated members to provide a non-contiguous memory arrangement. This increases the cost of serialization in Two-Copy and demonstrates the advantage of the Scatter-Gather I/O technique.

3.2 Two-Copy (A1):

Data Path: User buffers → serialized buffer → kernel socket buffer → NIC

3.3 One-Copy (A2):

Data Path: User buffers (iovec) → kernel socket buffer → NIC

3.4 Zero-Copy (A3):

Data Path: User buffers → page pinning → NIC DMA

4 Automation and Interaction Menu

4.1 User Interactive Menu

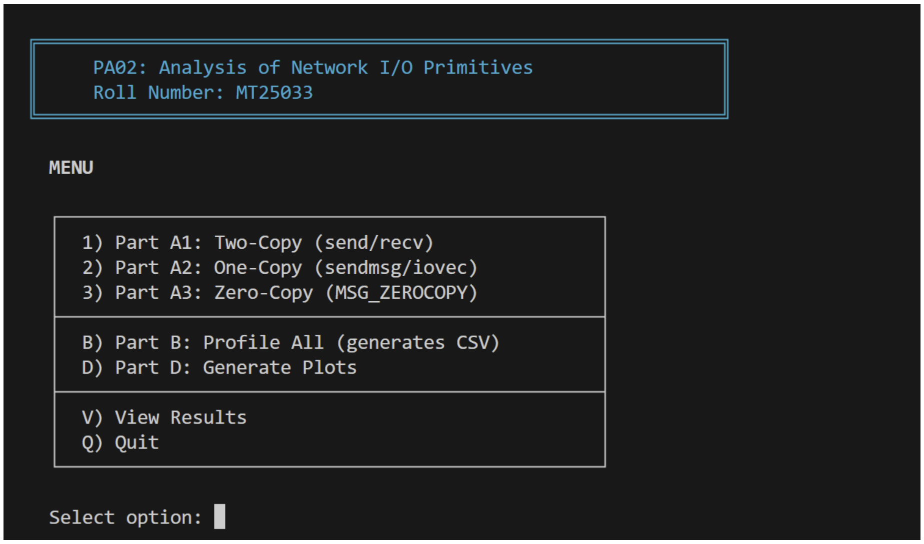

Figure 4: Interactive Menu Interface

4.2 Automation Workflow:

The automated script runs all combinations of message sizes and thread counts, and it aggregates the perf stats into CSV files.

5 Analysis of results

5.1 Analysis of throughput:

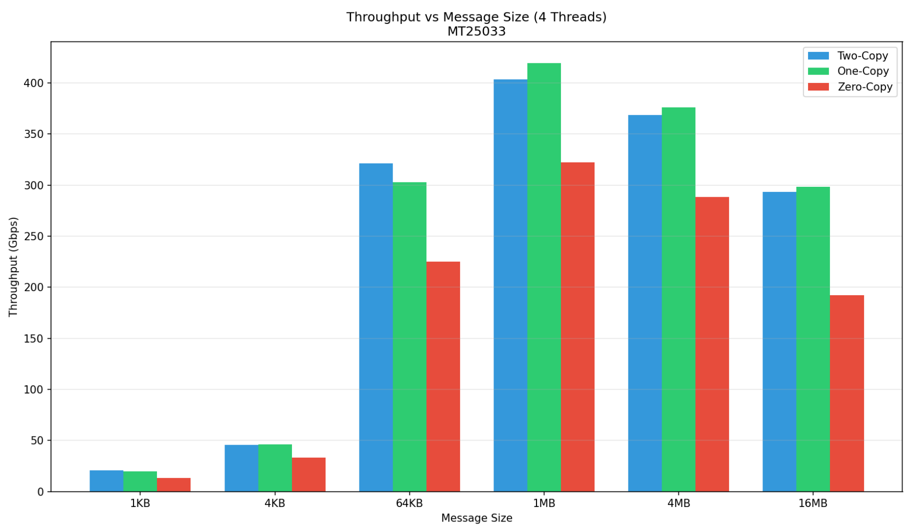

Figure 5: Throughput vs Message Size (4 Threads)

Observations:

1. In the experiment, I observed that throughput is increases as the message size increases in all three implementations till the message size reaches to 1MB. After 1MB throughput is saturated or dropped. This is a clear sign that the system changes from a CPU-bound to a memory/kernel-path-bound system.

2. Two-Copy (A1) performs competitively at small message sizes (1–4 KB). At these sizes, the cost of setting up iovec structures in A2 outweighs the serialization cost in A1.

3. One-Copy (A2) achieves the highest peak throughput (approx 420 Gbps at 1 MB). Eliminating user-space serialisation removes an extra memcpy, reducing CPU cycles and cache pollution.

4. Zero-Copy (A3) consistently underperforms across all message sizes. The advantage of using DMA is outweighed by page pinning costs and kernel accounting.

5. However, the decline in the throughput rate beyond 1 MB indicates cache pressure and memory bandwidth saturation, seen in A1 and A2.

5.2 Analysis of Latency:

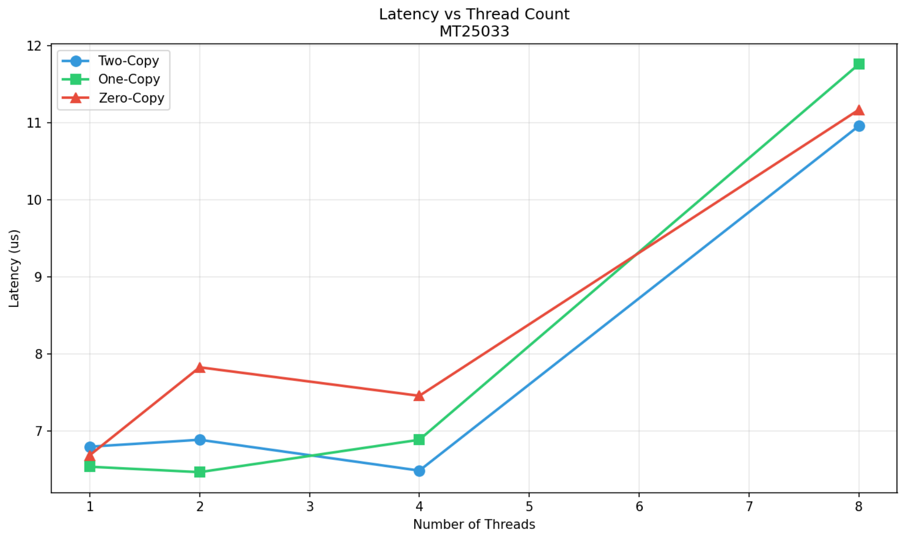

Figure 6: Latency vs Thread Count (64 KB)

Observations:

- From 1 to 4 threads, there is nearly the same value of delay (6.5–7 microseconds). This means that the system scales well with low thread counts when threads don't block one another much.
- At 8 threads, the delays suddenly rise again (11 or 12 microsec).This occurs because the CPU begins switching between threads more and shared data continues moving in and out of the caches, which slows performance.
- One-Copy (A2) is the fastest at 1–2 threads because it avoids extra steps, so each message is handled quicker.
- Zero-Copy (A3) is a bit slower even with few threads because it has extra overhead like locking memory pages and handling completion signals.

5.3 CPU Overhead:

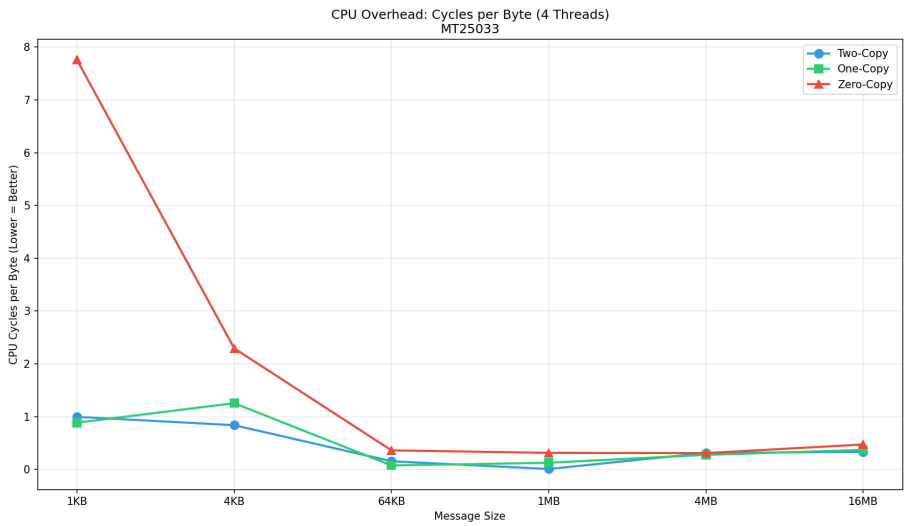

Figure 7: CPU Cycles per Byte vs Message Size

Observations:

- Zero-Copy (A3) uses a lot of CPU (about 7.7 cycles per byte), almost 8 times worse than A1 and A2 with small messages (1 KB).
- This happens because Zero-Copy has fixed setup costs (locking memory pages, setting up DMA, kernel work) that don't get smaller when the message is small.
- For medium-sized messages (64 KB), one-Copy (A2) is the most efficient because it sends data smartly using scatter-gather, reducing extra work.
- All approaches perform similarly (around 0.3–0.5 cycles per byte) because the setup cost gets spread over a lot of data for very large messages (4 MB and above).

5.4 Cache Behavior:

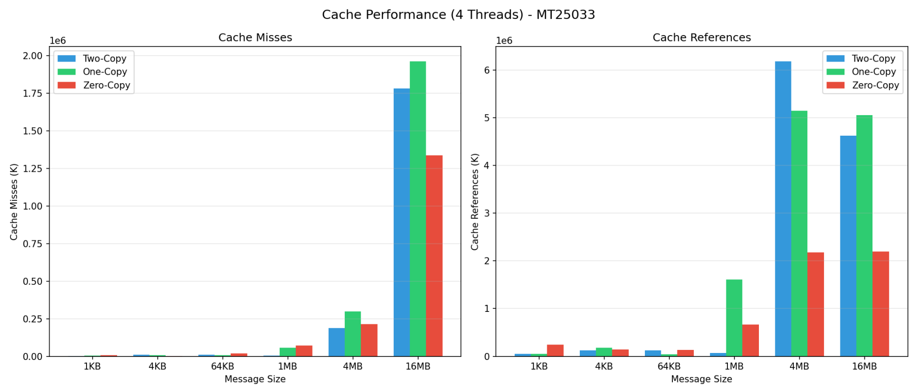

Figure 8: Cache Misses and Cache References vs Message Size

Observations:

- Two-Copy (A1) shows higher cache misses at medium sizes due to serialization buffers polluting the cache hierarchy.
- One-Copy (A2) consistently reduces cache misses compared to A1 by eliminating the intermediate contiguous buffer.
- Zero-Copy (A3) significantly reduces cache references at large sizes (16 MB), confirming that bypassing kernel buffers lowers cache traffic.
- Despite fewer cache references, A3 does not outperform others because cache efficiency alone does not compensate for scheduling and pinning overheads.

5.5 Context Switching:

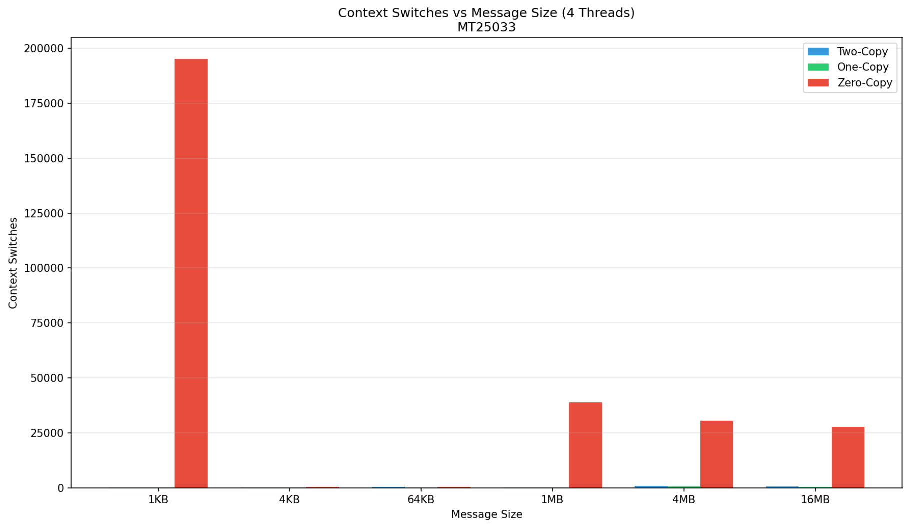

Figure 9: Context Switches vs Message Size

Observations:

- Zero-Copy (A3) triggers an extreme number of context switches, especially at 1 KB (approx 195K), which is over 1000× higher than A1/A2.
- These context switches arise from asynchronous zero-copy completion notifications handled via the socket error queue.
- Even at large message sizes (1–16 MB), A3 consistently shows tens of thousands of context switches, while A1 and A2 remain below 1K.
- High context switching leads to TLB flushes, cache invalidation, and scheduler overhead, directly harming throughput and latency.

5.6 Summary Plot:

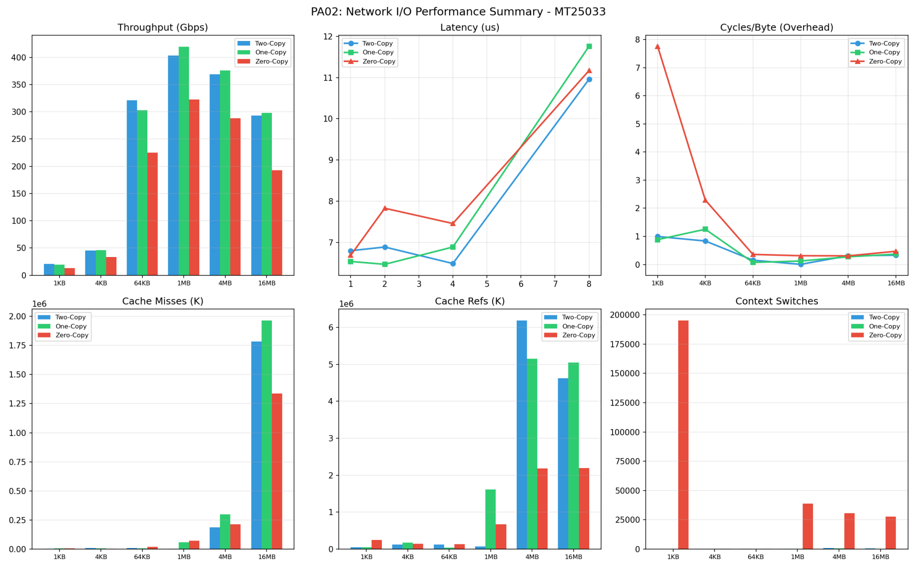

Figure 10: Combined Performance Summary

This figure provides a consolidated view of throughput, latency, CPU overhead, cache behavior, and context switching.

6 Conclusion

- One-Copy is the most balanced approach, offering high throughput, low latency, and reasonable CPU/cache behavior.
- Two-Copy is simple but inefficient, especially as message sizes grow.
- Zero-Copy is theoretically optimal, but in practice performs poorly in a virtualized testbed due to pinning and scheduling overheads.

7 AI Usage Declaration

The following components used AI assistance:

- Socket implementation structure and boilerplate code
- perf event configuration for hybrid CPUs
- Bash script for automated experiments

- Matplotlib plotting script structure
- Report structure and data analysis formatting

8 References

- Linux kernel documentation: `MSG_ZEROCOPY` – `https://www.kernel.org/doc/html/latest/networking/msg_zerocopy.html`
- Solarflare, "Understanding Zero-Copy Networking"
- perf wiki – `https://perf.wiki.kernel.org/`
- Stevens, W.R. "Unix Network Programming, Vol. 1" – Socket APIs and advanced I/O
- Linux man pages: `sendmsg(2)`, `recvmsg(2)`, `setsockopt(2)`, `ip-netns(8)`

9 GitHub

`https://github.com/PalakBafna/MT25033_PA02.git`