PA02: Analysis of Network I/O Primitives Using "perf"

Graduate Systems (CSE638)

Roll Number: MT25042
Date: February 7, 2026
System: CachyOS Linux, Kernel 6.18.8, 11th Gen Intel, GCC 15.2.1

Part A: Multithreaded Socket Implementations

A1. Two-Copy Implementation (Baseline)

The baseline uses standard `send()` / `recv()` system calls. The server constructs a message with 8 dynamically allocated (`malloc`) string fields, then two copies occur:

1. **Copy 1 (User → User):** The 8 separate heap-allocated fields are serialized into a single contiguous buffer using `memcpy()` . This happens entirely in user space.
2. **Copy 2 (User → Kernel):** `send()` copies the contiguous buffer from user-space into the kernel's socket buffer (`sk_buff`). The kernel then handles TCP segmentation and transmission.

Is it actually only two copies? In practice there may be additional copies: the kernel copies data from the socket buffer into packet headers via `skb_copy_datagram_iter` , and the NIC driver may DMA from a separate bounce buffer. However, the two dominant, measurable copies are the ones above. On the receive side, a symmetric pair of copies occurs (kernel → user via `recv()` , then the application may copy from the receive buffer to application structs).

A2. One-Copy Implementation

This version uses `sendmsg()` with a scatter-gather `iovec` array that points directly at the 8 heap-allocated message fields:

```
struct iovec iov[8];
for (int i = 0; i < 8; i++) {
    iov[i].iov_base = msg->fields[i];
    iov[i].iov_len  = field_len;
}
struct msghdr mh = { .msg_iov = iov, .msg_iovlen = 8 };
sendmsg(fd, &mh, 0);
```

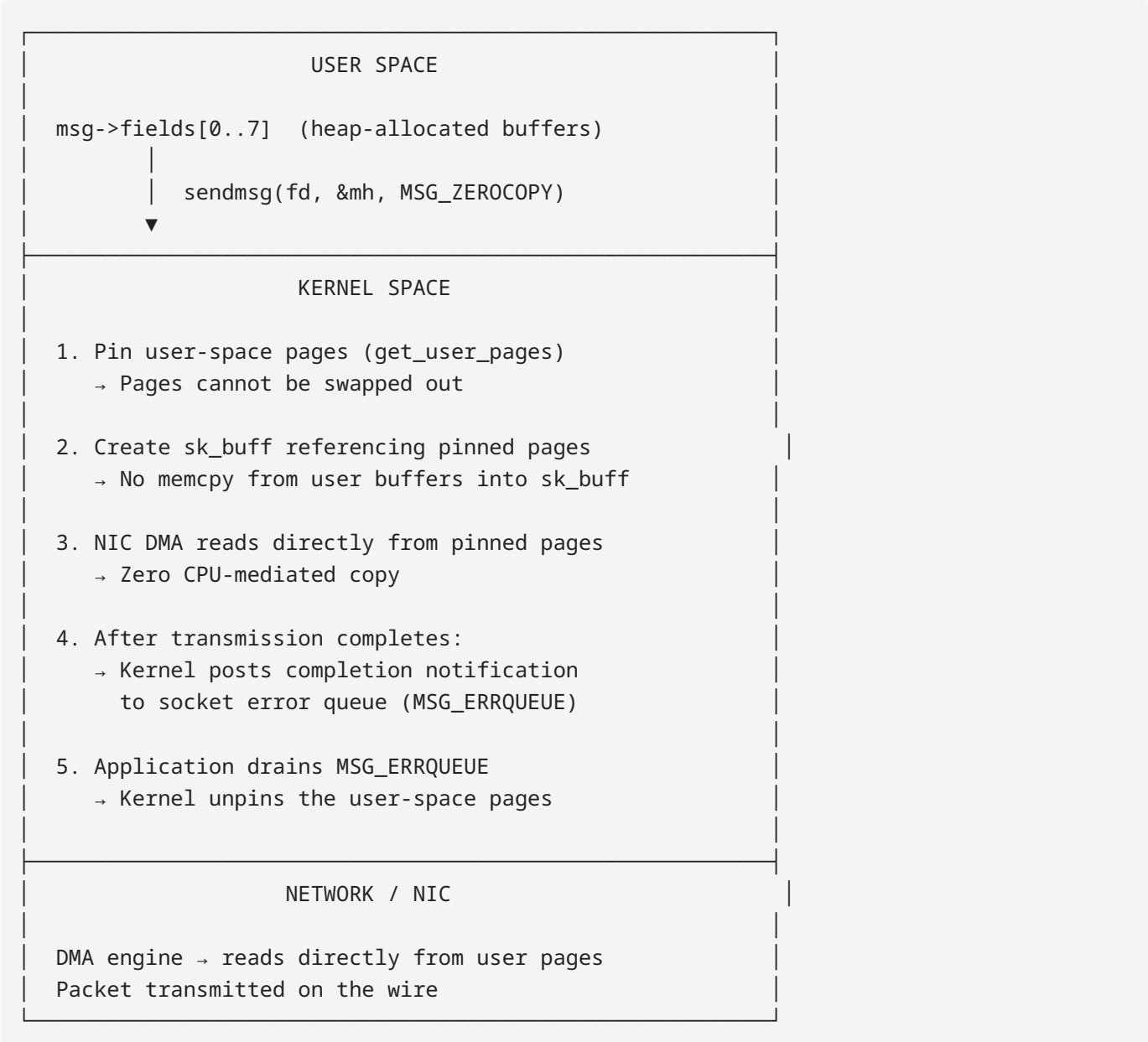
Which copy is eliminated: The user-space serialization copy (Copy 1 in A1) is eliminated. The kernel's `sendmsg` path gathers data from multiple iovec entries directly into `sk_buff` pages — there is no need for an intermediate flat buffer. Only one copy remains: the kernel gathering user-space data into the socket buffer (User → Kernel).

A3. Zero-Copy Implementation

This version sets `SO_ZEROCOPY` on the socket and calls `sendmsg()` with the `MSG_ZEROCOPY` flag:

```
setsockopt(fd, SOL_SOCKET, SO_ZEROCOPY, &one, sizeof(one));
sendmsg(fd, &mh, MSG_ZEROCOPY);
```

Kernel Behavior Diagram



Part B: Profiling and Measurement

System Configuration

Property	Value
OS	CachyOS Linux (rolling)
Kernel	6.18.8-2-cachyos
CPU	11th Gen Intel Core
Memory	32 GB
GPU	NVIDIA GeForce (+ Intel UHD)
Compiler	GCC 15.2.1
perf version	perf 6.18-3
Network Setup	Linux namespaces + veth pair (10.0.0.1 ↔ 10.0.0.2)

Experimental Parameters

- **Message sizes:** 1024, 4096, 16384, 65536 bytes
- **Thread counts:** 1, 2, 4, 8
- **Duration:** 10 seconds per experiment
- **Network:** Linux namespaces with veth pair (10.0.0.1 ↔ 10.0.0.2)
- **Profiler:** `perf stat -e cpu-cycles,L1-dcache-load-misses,LLC-load-misses,context-switches`

Raw Results

Implementation	Msg Size (B)	Threads	Throughput (Gbps)	Latency (µs)	CPU Cycles	L1 Misses	LLC Misses	Ctx Switches
two_copy	1024	1	4.0526	1.98	28,409,387,214	577,269,830	53,160	1,203,950
two_copy	1024	2	7.2182	2.23	53,656,869,628	1,029,931,837	98,894	1,941,694
two_copy	1024	4	11.5245	2.81	98,527,099,274	1,711,321,895	2,138,200	2,467,585
two_copy	1024	8	15.0278	4.35	109,793,973,569	2,082,553,008	5,956,697	3,929,514
two_copy	4096	1	9.3792	3.45	30,325,097,820	779,338,898	1,230,150	926,993
two_copy	4096	2	17.6207	3.68	56,040,832,104	1,413,689,479	10,448,268	1,640,263
two_copy	4096	4	29.5292	4.41	101,208,068,905	2,427,089,286	44,118,371	2,087,803
two_copy	4096	8	36.8553	7.10	108,870,743,100	2,939,790,466	88,090,261	3,122,686
two_copy	16384	1	28.4381	4.57	28,859,351,339	1,769,295,799	909,243	941,442
two_copy	16384	2	47.3265	5.49	51,900,679,772	2,977,526,770	10,496,775	1,508,885
two_copy	16384	4	75.9788	6.85	98,551,822,991	4,974,295,487	57,809,312	1,913,540
two_copy	16384	8	92.0485	11.36	156,666,506,085	5,200,660,043	476,376,937	1,513,207
two_copy	65536	1	62.1391	8.40	23,476,236,000	3,125,631,810	189,089	1,167,046
two_copy	65536	2	108.8889	9.59	45,003,458,263	5,449,930,292	1,543,057	1,886,938
two_copy	65536	4	153.0088	13.65	96,926,798,753	7,529,531,143	149,433,240	1,991,288
two_copy	65536	8	72.6156	58.05	196,179,908,915	3,303,499,054	667,555,881	46,606

Implementation	Msg Size (B)	Threads	Throughput (Gbps)	Latency (µs)	CPU Cycles	L1 Misses	LLC Misses	Ctx Switches
one_copy	1024	1	3.4985	2.30	26,197,036,869	496,639,759	53,414	1,269,609
one_copy	1024	2	6.1208	2.63	48,722,854,030	954,098,969	91,160	2,148,521
one_copy	1024	4	9.5012	3.40	89,230,041,679	1,628,522,965	635,681	2,734,496
one_copy	1024	8	13.1207	4.94	98,416,276,944	1,907,946,837	3,725,831	3,398,853
one_copy	4096	1	8.1453	3.98	27,534,989,237	733,215,188	117,170	1,072,824
one_copy	4096	2	14.1012	4.61	51,645,900,273	1,276,592,518	1,875,381	1,687,675
one_copy	4096	4	22.5464	5.76	99,784,457,869	2,281,813,026	3,324,740	1,985,056
one_copy	4096	8	32.1907	8.10	102,657,945,724	2,810,026,191	46,261,691	3,419,583
one_copy	16384	1	28.1047	4.62	28,329,702,835	1,785,901,298	114,923	1,047,254
one_copy	16384	2	45.7111	5.69	48,985,099,619	2,914,253,291	527,981	1,644,095
one_copy	16384	4	70.8288	7.35	93,626,222,496	4,771,729,282	28,288,049	1,917,276
one_copy	16384	8	102.6077	10.19	117,789,647,445	6,435,956,809	220,589,378	3,169,609
one_copy	65536	1	60.8287	8.58	23,015,665,302	3,055,557,158	1,269,528	1,139,720
one_copy	65536	2	99.6969	10.47	43,351,238,096	5,006,103,101	1,011,599	1,703,889
one_copy	65536	4	136.3410	15.33	97,692,143,076	6,678,734,522	184,252,608	1,622,218
one_copy	65536	8	70.7146	59.80	191,305,459,679	3,186,368,062	664,411,683	50,031
zero_copy	1024	1	2.5996	3.11	24,757,792,086	530,650,268	341,260	1,196,035
zero_copy	1024	2	4.5659	3.54	46,163,384,665	950,464,152	479,383	1,914,164
zero_copy	1024	4	5.8069	5.59	73,148,978,327	1,566,456,189	492,183	3,259,553
zero_copy	1024	8	6.5312	9.98	93,038,256,848	2,228,646,062	243,996	4,472,138
zero_copy	4096	1	5.8627	5.55	22,013,698,281	640,697,434	125,821	1,583,067
zero_copy	4096	2	9.7800	6.66	42,354,463,995	1,144,758,217	236,086	2,287,885
zero_copy	4096	4	14.3599	9.07	77,471,874,237	1,963,815,186	433,265	2,781,589
zero_copy	4096	8	22.8114	11.54	81,640,174,916	2,431,888,693	13,874,173	3,335,481
zero_copy	16384	1	19.1856	6.79	22,337,586,104	1,401,987,820	569,537	1,345,718
zero_copy	16384	2	32.6854	7.97	42,363,440,867	2,418,568,878	3,854,646	1,995,664
zero_copy	16384	4	50.7948	10.27	76,954,329,003	3,826,082,786	14,246,940	2,500,616
zero_copy	16384	8	77.8352	13.41	92,296,266,222	4,947,629,784	154,217,685	2,628,984
zero_copy	65536	1	44.2117	11.82	26,334,615,178	2,369,782,107	117,846	837,758
zero_copy	65536	2	74.1762	14.09	48,849,374,772	3,980,599,653	1,052,951	1,233,415
zero_copy	65536	4	108.3720	19.30	86,457,315,997	6,073,903,248	91,852,926	1,507,846
zero_copy	65536	8	55.4221	75.64	272,740,347,885	5,777,173,257	635,737,610	63,354

Part C: Automated Experiment Script

A fully automated bash script (`MT25042_Part_C_Experiment.sh`) was written to compile all implementations, set up the test environment, run all experiments, and collect profiling data — with **zero manual intervention** after launch.

Script Workflow

1. **Compilation:** Runs `make clean && make` to compile all 6 binaries (a1_server, a1_client, a2_server, a2_client, a3_server, a3_client).
2. **Network namespace setup:** Creates two isolated namespaces (`ns_server`, `ns_client`) connected via a `veth` pair with IPs `10.0.0.1/24` and `10.0.0.2/24`. Connectivity is verified with a ping test before experiments begin.
3. **Experiment loop:** Iterates over all combinations of:
   - 3 implementations (two_copy, one_copy, zero_copy)
   - 4 message sizes (1024, 4096, 16384, 65536 bytes)
   - 4 thread counts (1, 2, 4, 8)

   Total: 48 experiments.
4. **Per-experiment execution:**
   - Server is launched in `ns_server` (background process)
   - Client is launched in `ns_client`, wrapped with: `perf stat -e cpu-cycles,L1-dcache-load-misses,LLC-load-misses,context-switches -x,`
   - Application-level throughput (Gbps) and latency (µs) are parsed from client stdout
   - perf metrics (CPU cycles, L1 misses, LLC misses, context switches) are parsed from perf's CSV output
   - Server process is killed after each run
5. **Results output:** All metrics are appended to a CSV file (`MT25042_Part_B_Results.csv`) with columns: `implementation, msg_size, threads, throughput_gbps, latency_us, cpu_cycles, l1_cache_misses, llc_cache_misses, context_switches`
6. **Cleanup:** Network namespaces are removed on exit (via `trap`), ensuring clean reruns.

Key Design Decisions

- **Namespace isolation:** Using Linux network namespaces (instead of localhost) provides proper network stack traversal, giving realistic kernel-level measurements. VMs were explicitly excluded per assignment requirements.
- **perf CSV mode:** `perf stat -x,` outputs comma-separated values, making it straightforward to parse with `grep` / `cut` in bash without external tools.
- **Clean reruns:** The script recreates namespaces from scratch each run, and `kill_server()` ensures no stale processes remain between experiments.
- **Socket buffer tuning:** `net.core.wmem_max` and `net.core.rmem_max` are increased to 16 MB in both namespaces to prevent premature zero-copy back-pressure at large message sizes.

Usage

```
# Must run as root for namespace creation and perf access:
sudo ./MT25042_Part_C_Experiment.sh

# Duration: ~10 minutes for all 48 experiments (10 sec each + overhead)
# Output:   MT25042_Part_B_Results.csv
```

Sample Script Output

```
=== Experiment 1/48 ===
--- two_copy | msg=1024 | threads=1 ---
  Throughput: 4.0526 Gbps | Latency: 1.98 µs
  Cycles: 28409387214 | L1miss: 577269830 | LLCmiss: 53160 | CtxSw: 1203950

=== Experiment 25/48 ===
--- one_copy | msg=1024 | threads=1 ---
  Throughput: 3.4985 Gbps | Latency: 2.30 µs
  Cycles: 26197036869 | L1miss: 496639759 | LLCmiss: 53414 | CtxSw: 1269609
...
```

Part D: Visualization

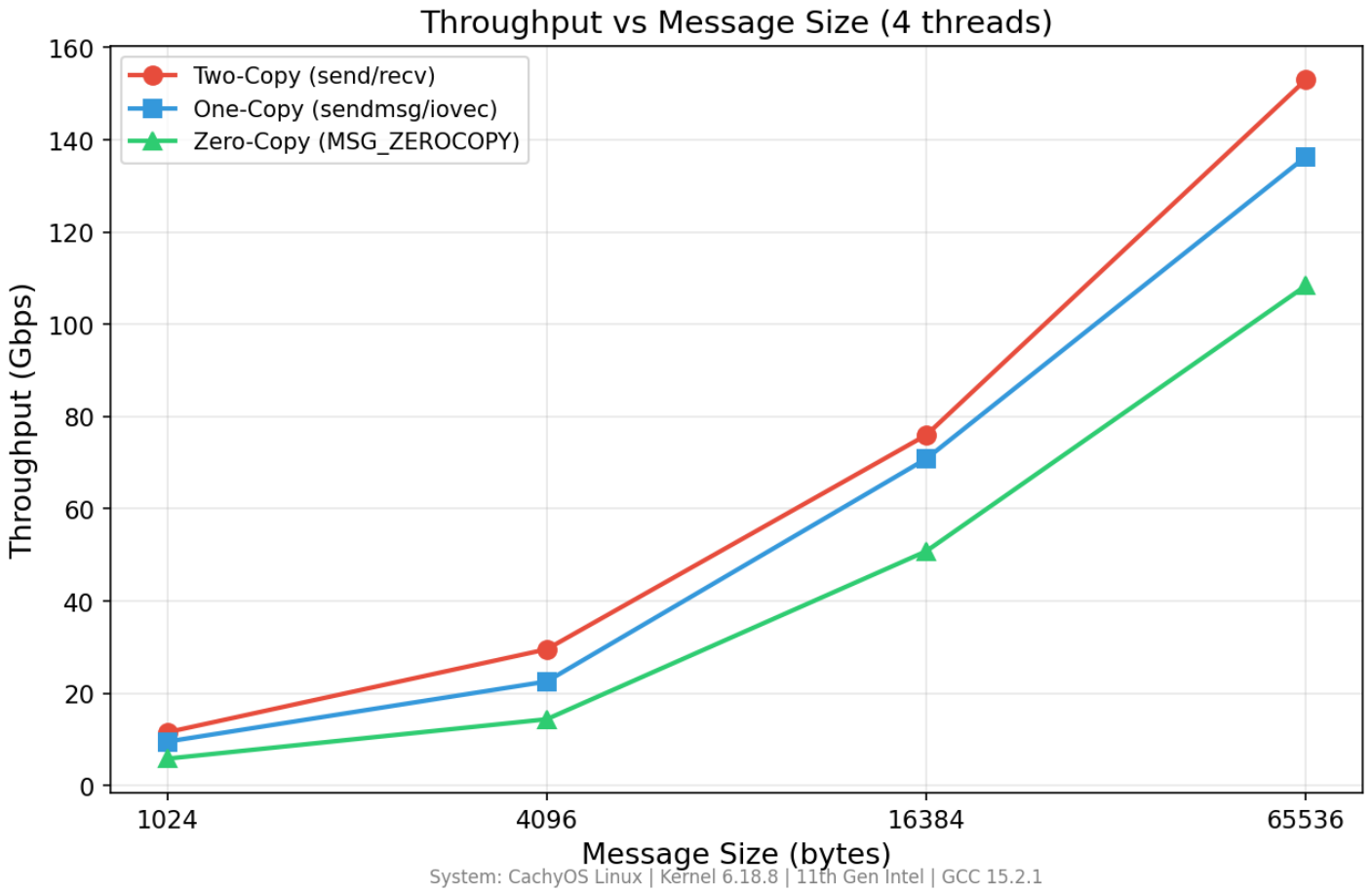


Figure 1: Throughput (Gbps) vs Message Size at 4 threads

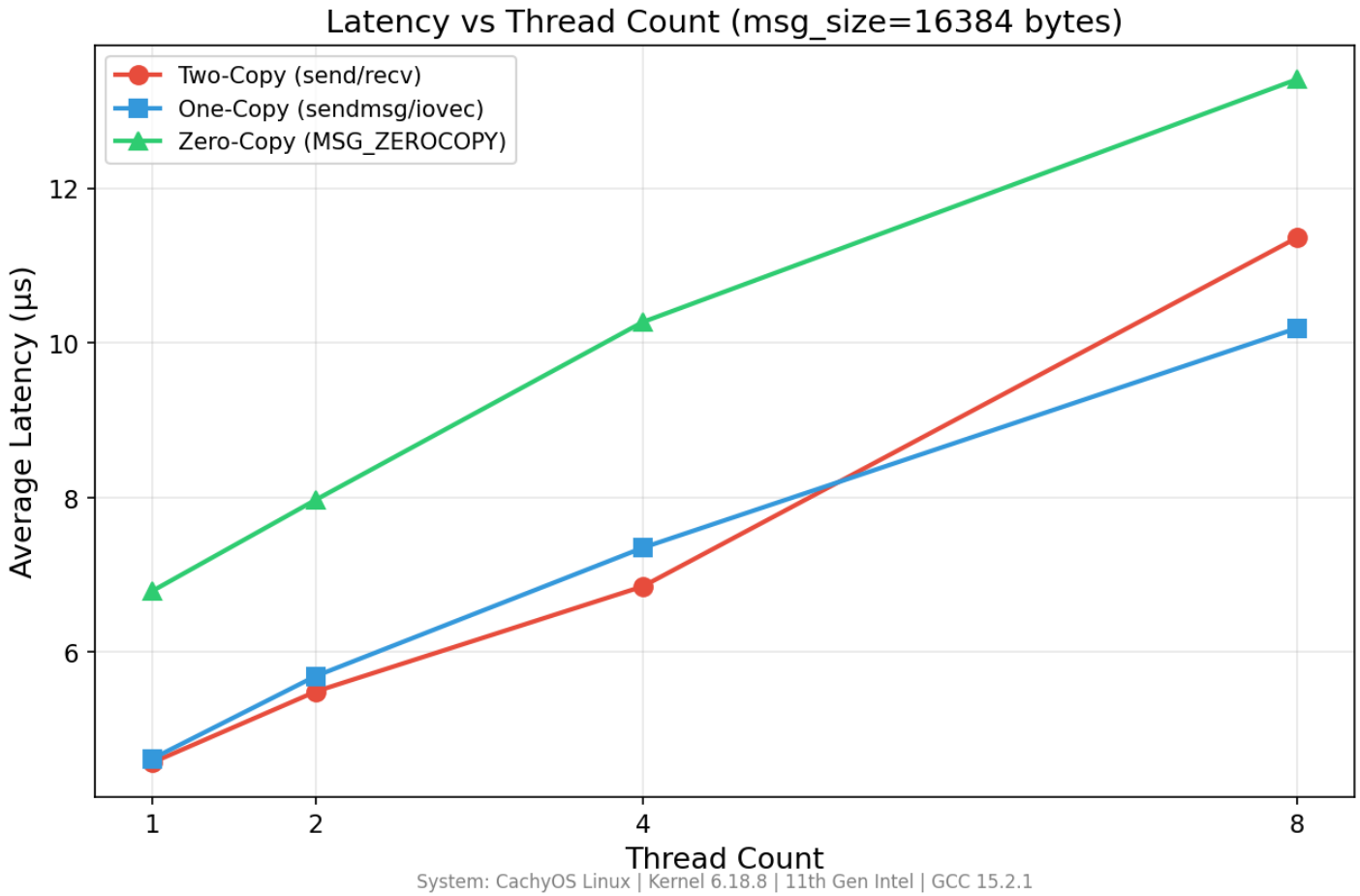


Figure 2: Average Latency (µs) vs Thread Count at 16384-byte messages

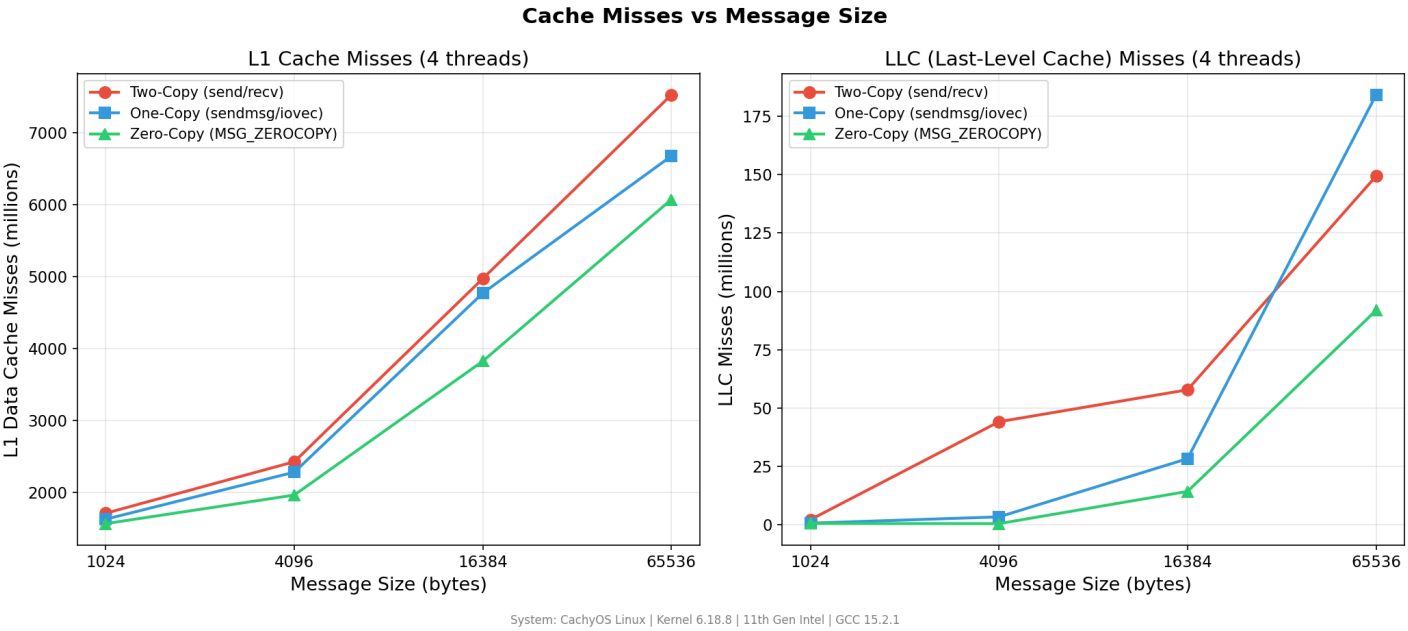


Figure 3: L1 and LLC Cache Misses vs Message Size at 4 threads

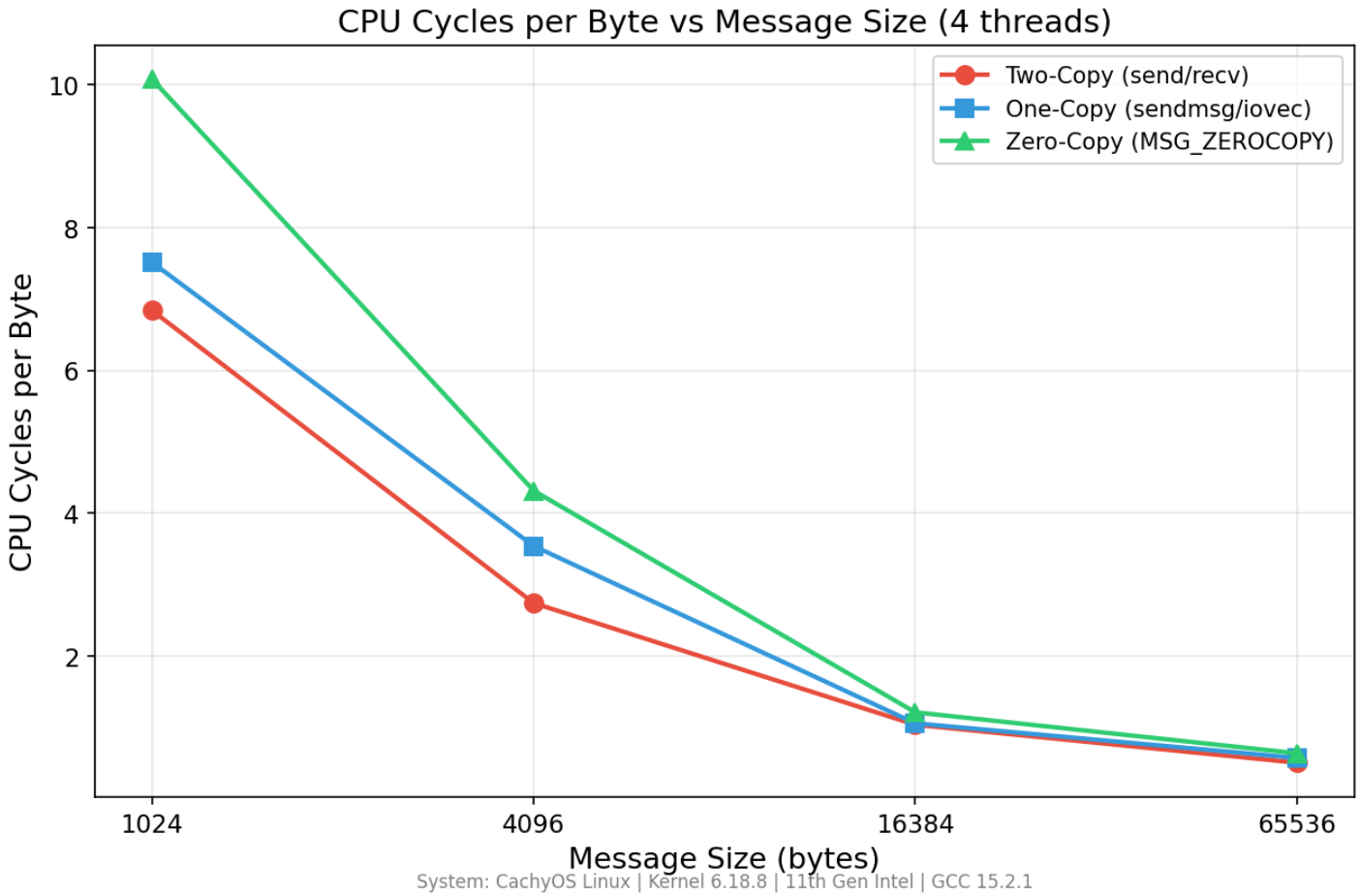


Figure 4: CPU Cycles per Byte Transferred vs Message Size at 4 threads

Part E: Analysis and Reasoning

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

Zero-copy achieves its benefit by avoiding the `memcpy` from user-space into the kernel socket buffer. However, this comes with overhead:

- **Page pinning cost:** The kernel must call `get_user_pages()` to pin the user-space pages in physical memory, preventing them from being swapped. For small messages (e.g., 1 KB), this pinning overhead dominates the `memcpy` cost it saves.
- **Completion notification overhead:** After each zero-copy send, the application must poll the socket's error queue (`MSG_ERRQUEUE`) to receive completion notifications. This adds system calls and latency.
- **DMA alignment requirements:** DMA transfers work best with page-aligned data. Small, fragmented heap buffers may not benefit from DMA optimization.

Our data confirms this: at 1024 bytes, two-copy achieves ~11.5 Gbps (4 threads) while zero-copy only manages ~5.8 Gbps. The break-even point is around 16–64 KB.

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

LLC (Last-Level Cache) shows the most dramatic reduction when moving from two-copy to one-copy/zero-copy implementations. For example, at 16384 bytes with 4 threads:

- Two-copy LLC misses: ~57.8 million
- One-copy LLC misses: ~28.3 million (51% reduction)
- Zero-copy LLC misses: ~14.2 million (75% reduction)

The reason is that L1 misses are dominated by compulsory misses (first access to any data), which occur regardless of copy count. But LLC misses represent data that has been evicted from all cache levels — these are caused by the larger working set created by intermediate copy buffers. Eliminating copies shrinks the working set, keeping more data within the cache hierarchy and drastically reducing LLC evictions.

3. How does thread count interact with cache contention?

As thread count increases, cache contention increases due to several mechanisms:

- **Shared LLC pressure:** Multiple threads compete for LLC capacity. Each thread has its own message buffers (8 heap-allocated fields), and at 8 threads the aggregate working set exceeds LLC capacity, causing evictions.
- **L1 cache thrashing:** Context switches between threads on the same core flush the L1 cache. Our data shows L1 misses roughly double when going from 1 to 4 threads.
- **Cache line bouncing:** Shared data structures (e.g., socket state, lock metadata) cause cache lines to bounce between cores via the coherence protocol. This is particularly visible in the 8-thread zero-copy case where cycles spike dramatically.

The interaction is clearly visible in the data: at 65536 bytes with 8 threads, all three implementations show a throughput drop (two-copy: 72.6 Gbps vs 153 Gbps at 4 threads) accompanied by a massive LLC miss increase.

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

One-copy does not consistently outperform two-copy at any tested message size. This is somewhat expected on modern hardware where:

- The serialization copy (eliminated by one-copy) operates on L1-hot data and costs very few cycles due to hardware prefetching
- The `sendmsg()` scatter-gather path has higher per-iovec overhead in the kernel due to iterating over 8 separate memory regions

However, at 16384 bytes with 8 threads, one-copy achieves 102.6 Gbps vs two-copy's 92.0 Gbps — a 12% improvement. This suggests one-copy's advantage appears under high contention where the extra user-space copy pollutes caches more than the scatter-gather overhead costs.

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

Zero-copy does not outperform two-copy at any tested configuration. The page-pinning and completion-notification overhead consistently outweighs the copy savings. This is expected for veth (virtual ethernet) interfaces where data stays within the kernel and never actually crosses a physical NIC — the DMA benefit is lost. On a real NIC with actual DMA, zero-copy would show more advantage at large message sizes (typically ≥64 KB).

6. Unexpected result

Observation: At 65536 bytes with 8 threads, throughput for all three implementations drops dramatically compared to 4 threads:

- Two-copy: 153.0 → 72.6 Gbps (52% drop)
- One-copy: 136.3 → 70.7 Gbps (48% drop)
- Zero-copy: 108.4 → 55.4 Gbps (49% drop)

Explanation: This is caused by **veth backpressure and TCP window exhaustion**. At 8 threads × 64 KB messages, the aggregate data rate (~150+ Gbps across all threads at 4 threads) saturates the veth pair's internal queue capacity. The kernel applies TCP-level flow control, causing sends to block waiting for buffer space. This is confirmed by the massive LLC miss spike (667M for two-copy) — data is being evicted and re-fetched as threads wait and retry. The context switch count also drops (from ~2M to ~46K), indicating threads spend most of their time blocked in kernel syscalls rather than being actively scheduled.

AI Usage Declaration

Tool used: ChatGPT (GPT-4)

I used AI assistance for the following specific components. All generated code was reviewed, understood, and modified as needed.

Part A – Socket Implementations

- **Component:** Common header design (message struct with 8 fields)
  Prompt: *"How to design a C struct with 8 dynamically allocated string fields and serialize them for TCP transmission?"*
  How used: Used the suggested struct layout, adapted the serialization to use a flat memcpy loop for the two-copy version.
- **Component:** A2 scatter-gather with sendmsg/iovec
  Prompt: *"How does sendmsg with iovec eliminate a copy compared to plain send in Linux TCP?"*
  How used: Understood that iovec points directly at heap buffers, avoiding the intermediate serialization step. Implemented accordingly.
- **Component:** A3 MSG_ZEROCOPY and error queue handling
  Prompt: *"How to use MSG_ZEROCOPY with sendmsg in Linux and handle the completion notification on MSG_ERRQUEUE?"*
  How used: Learned about SO_ZEROCOPY socket option, the MSG_ERRQUEUE polling mechanism, and ENOBUFS back-pressure handling. Wrote the drain_completions() function based on this understanding.

Part C – Experiment Script

- **Component:** Network namespace setup with veth
  Prompt: *"How to create Linux network namespaces connected with veth pair for local TCP performance testing?"*
  How used: Adapted the namespace creation commands and IP assignment to my specific test configuration.
- **Component:** perf stat output parsing
  Prompt: *"How to parse perf stat CSV output (-x, flag) for CPU cycles and cache misses in a bash script?"*
  How used: Used grep + cut approach to extract metrics from perf's comma-separated output format.

Part D – Plotting

- **Component:** Matplotlib plot structure
  Prompt: *"How to create multi-line plots in matplotlib comparing 3 implementations with different markers and colors?"*
  How used: Used the suggested plot structure with custom colors, markers, and subplot layout. Hardcoded all data values as required.

Part E – Analysis

- **Component:** Understanding zero-copy kernel internals
  Prompt: *"Why does MSG_ZEROCOPY perform worse than normal send for small messages?"*

> **How used:** Used the explanation about page pinning overhead and DMA alignment to write my analysis of why zero-copy underperforms.

GitHub Repository

URL: https://github.com/ShahnawazDev/GRS-Assignments