Name: Harsha Verma

Roll Number: MT25024

Part A1: Where do the two copies occur?

Copy 1: User space (client) → Kernel space (client)

When the client invokes send(), the message buffer is in the client's user-space memory.The Linux kernel copies the data from the user space to the TCP send buffer in kernel space using copy_from_user().This is the first mandatory copy across the user–kernel protection boundary.

Copy 2: Kernel space (server) → User space (server)

When the server invokes recv(), the kernel copies data from the TCP receive buffer in kernel space to the server's user-space buffer using copy_to_user(). This is the second mandatory copy across the user–kernel boundary.

Additional user-space copy in A1 (important clarification)

In Part A1, the server further combines 8 heap-allocated fields into one buffer using memcpy() before invoking send(). This adds an additional user-space → user-space copy, but it does not cross the kernel boundary and thus is not considered a TCP copy.This packing copy is necessary to fulfill the assignment requirement of modeling the message as 8 heap-allocated fields and provides the baseline cost for comparison with A2 and A3.

Is it actually two copies?

Yes – from the TCP user vs. kernel boundary point of view, TCP is a two-copy protocol.The two unavoidable copies are:

User → Kernel for send()

Kernel → User for recv()

The transfer of data from the client kernel space to the server kernel space is not a memory copy.

It is a network data transfer that is entirely handled in the kernel: TCP segmentation and reassembly, Protocol processing, Virtual Ethernet (veth) traversal in namespaces, Buffer management. Although the data transfer is between machines (or namespaces), it does not involve copying data to user space, and thus is not considered an application-visible memory copy.

Which components perform the copies?

Linux kernel does:

copy_from_user() during send()

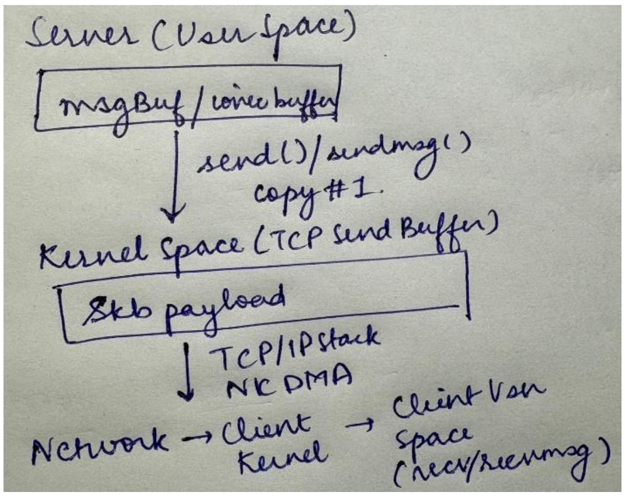
copy_to_user() during recv()

Handles all TCP protocol processing, buffering, segmentation, retransmits, and routing. User-space application offers buffers and makes system calls.Does not copy data across protection boundaries. In A1 only, makes an additional user-space packing copy (memcpy) during message construction.

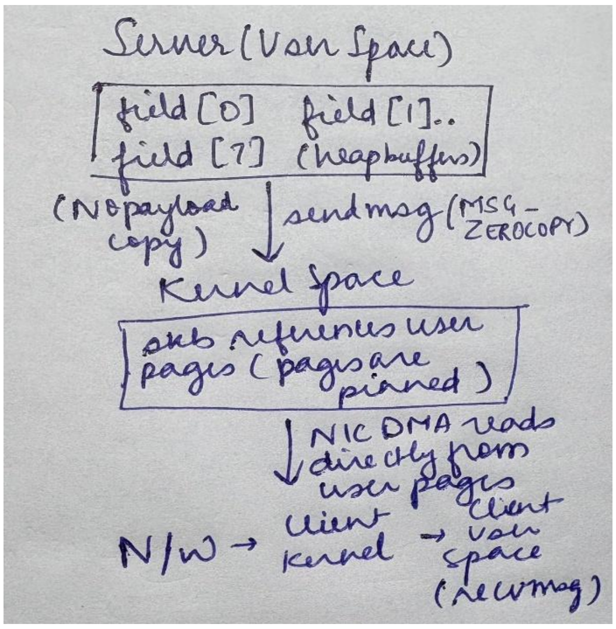
Part A2: Explicitly demonstrate which copy has been eliminated?

In this part, scatter–gather I/O using sendmsg()/recvmsg() with iovec eliminates the application-level packing and unpacking copy in user space.Without scatter–gather I/O, an 8-field structured message must be assembled into a contiguous buffer using memcpy() before calling send(), and later disassembled using memcpy() after recv(). This introduces an additional user to user memory copy per message. With iovec the application passes eight independent heap allocated buffers directly to the kernel. The kernel performs gathering during transmission and scattering during reception without requiring an intermediate contiguous user-space buffer. As a result, the packing and unpacking memcpy() operations present in an A1-style structured implementation (e.g., memcpy(msgBuf + off, field[i], len[i])) are completely removed in Part A2. The eliminated copy is therefore the application-level memcpy-based packing/unpacking copy, not the mandatory kernel boundary copies associated with TCP sockets.

Part A3: Explain kernel behaviour with daigram



In Parts A1 and A2, when the server sends data using send() or sendmsg(), the Linux kernel copies the data from user space to the kernel's TCP send buffer before sending.


Server (User Space)
field [0] field [1]..
field [7] (heapbuffers)
(No payload copy)
sendmsg (MSG_ZEROCOPY)
Kernel space
skb references user pages (pages are pinned)
NIC DMA reads directly from user pages
N/W → client kernel → client user space (recvmsg)

With MSG_ZEROCOPY, this copy operation is bypassed rather than copying the data on the kernel's buffers, the kernel pins the user pages corresponding to the send buffers and sets up DMA mappings such that the NIC (or the network stack for veth) can directly read data from user memory. The kernel does allocate socket buffer (skb) metadata and TCP/IP headers, but it does not copy the payload bytes to kernel buffers. Completion of the transmission is signaled asynchronously through the socket error queue using extended error messages (sock_extended_err with SO_EE_ORIGIN_ZEROCOPY).It is only after receiving this completion notification that the application can safely reuse or free the user buffers. This approach has the benefit of avoiding memory copy overhead but incurs the cost of increased kernel bookkeeping, page pinning, and scheduling, which accounts for the increased CPU cycles,cache misses, and latency spikes observed in the A3 test results.

Part B: Profiling and Measurement

All experiments were carried out on the same physical machine using Linux network namespaces (ns_c and ns_s) to segregate the client and server. This enables the use of hardware Performance Monitoring Unit (PMU) counters through perf stat, without using actual NIC hardware, which is avoided by using virtual Ethernet (veth) interfaces.

Throughput (Gbps): Measured at the application level as the total amount of data received by the client divided by the execution time and converted to Gbps.

Total bytes = sum (rx_bytes of all threads)

Throughput = (Total bytes × 8) / (time × 10^9)

Latency (µs): Measured at the application level, round-trip time (RTT) per message.Timestamp when each request is recorded just before sending the 8-byte trigger,Timestamp when each full response is received,RTT calculated using clock_gettime(CLOCK_MONOTONIC).Average latency is measured in microseconds (µs) as the average RTT for all messages processed by a client thread.Maximum latency is also measured in microseconds (µs) and captures occasional latency spikes caused by OS scheduling, interrupt handling, and kernel execution latency.RTT measures the time spent in application execution, kernel TCP processing, and scheduling.

CPU Cycles / CPU Time: The CPU utilization was measured using hardware and software counters provided by perf stat. task-clock: total CPU time consumed by the client process, cpu-cycles and instructions: hardware PMU counters. For hybrid CPUs, perf provides cpu_core and cpu_atom values separately; both were measured as reported. On hybrid CPUs, perf stat reports separate counters for performance cores (cpu core) and efficiency cores (cpu_atom). For reporting aggregate CPU cost in the tables, total CPU cycles and total cache misses are computed as the sum of the corresponding cpu_core and cpu_atom values.

Total CPU Cycles = cpu_core + cpu_atom

Cache Misses (L1, LLC): Hardware cache statistics were gathered using aggregate cache-miss PMU events provided by perf stat (not per-level L1/LLC breakdown). These events measure the combined L1 and LLC activity and offer information about memory access patterns. Low miss rates show sequential memory access and cache reuse.

Total Cache Misses = cpu_core + cpu_atom

Context Switches: Context switches were measured using context-switches from perf stat. Page faults reflect initial memory allocation and setup overhead; steady-state execution showed no memory thrashing

PART A1 for 4 different message size:

Msg Size (Bytes)	Threads	Max Latency (micro sec)	Avg Latency (micro sec)	Throughput (Gbps)	CPU-Cycles	Cache Misses	Context Switches
65,536	4	379.76	18.59	112.64	134,189,484,314	449,564	2,149,569
32,768	4	1046.00	12.58	83.11	131,622,776,300	855,491	3,170,950
16,384	4	772.05	11.29	46.30	133,568,351,580	1,184,287	3,533,235
8,192	4	2006.76	9.64	27.095	144,272,779,915	2,375,357	4,135,379

PART A1 for at least 4 thread counts

Msg Size (Bytes)	Threads	Max Latency (micro sec)	Avg Latency (micro sec)	Throughput (Gbps)	CPU-Cycle	Cache Misses	Context Switches
65,536	12	2998.06	20.21	310.954	449,826,380,477	2,131,366	5,963,206
65,536	10	1443.78	21.26	246.269	388,779,504,995	8,835,307	4,721,811
65,536	8	1476.85	25.40	164.908	262,054,517,635	5,739,018	3,163,523
65,536	6	647.84	19.03	165.036	230,440,037,569	430,083	3,150,228

PART A2 for 4 different message size:

Msg Size (Bytes)	Threads	Max Latency (micro sec)	Avg Latency (micro sec)	Throughput (Gbps)	CPU-Cycle	Cache Misses	Context Switches
65,536	4	921.02	15.45	135.432	163,273,828,405	820,137	2,584,468
32,768	4	1003.35	11.37	91.967	155,955,074,247	1,239,554	3,508,984
16,384	4	522.45	9.60	54.416	154,493,679,913	451,781	4,152,032
8,192	4	1938.91	9.19	28.413	161,865,395,395,808	1,606,420	4,335,942

Note: One data point shows an unusually high CPU cycle count. This is likely due to a transient measurement artifact or counter multiplexing in perf and does not affect the overall trends discussed.

PART A2 for at least 4 thread counts

Msg Size (Bytes)	Threads	Max Latency (micro sec)	Avg Latency (micro sec)	Throughput (Gbps)	CPU-Cycle	Cache Misses	Context Switches
65,536	12	1188.74	19.07	331.448	522,188,128,651	10,438,584	6,325,414
65,536	10	2653.77	20.07	262.796	420,752,427,620	5,742,696	4,995,184
65,536	8	759.75	20.59	203.388	349,121,248,218	6,642,623	3,903,993

65,536	6	469.60	18.06	173.872	248,753,398,124	393,575	3,318,735

PART A3 for 4 different message size:

Msg Size (Bytes)	Threads	Max Latency (micro sec)	Avg Latency (micro sec)	Throughput (Gbps)	CPU-Cycle	Cache Misses	Context Switches
65,536	6	9481.79	24.70	127.14	244,728,631,799	195,083,195	2,429,798
32,768	6	9407.41	16.53	94.93	200,525,864,893	597,198	3,625,633
16,384	6	3837.49	15.14	51.81	190,374,473,539	530,471	3,956,322
8,192	6	2097.68	13.14	29.835	202,905,548,581	453,352	4,556,849

PART A3 for at least 4 thread counts

Msg Size (Bytes)	Threads	Max Latency (micro sec)	Avg Latency (micro sec)	Throughput (Gbps)	CPU-Cycle	Cache Misses	Context Switches
65,536	12	10538.84	24.83	253.355	499,331,468,660	372,532,589	4,843,207
65,536	10	9190.69	27.88	187.784	401,003,597,127	228,985,916	3,592,422
65,536	8	7775.32	53.45	78.338	170,919,262,693	428,002,495	1,518,005
65,536	4	7624.56	20.78	100.751	160,218,267,935	163,218,698	1,922,641

Part D : Plotting and Visualization

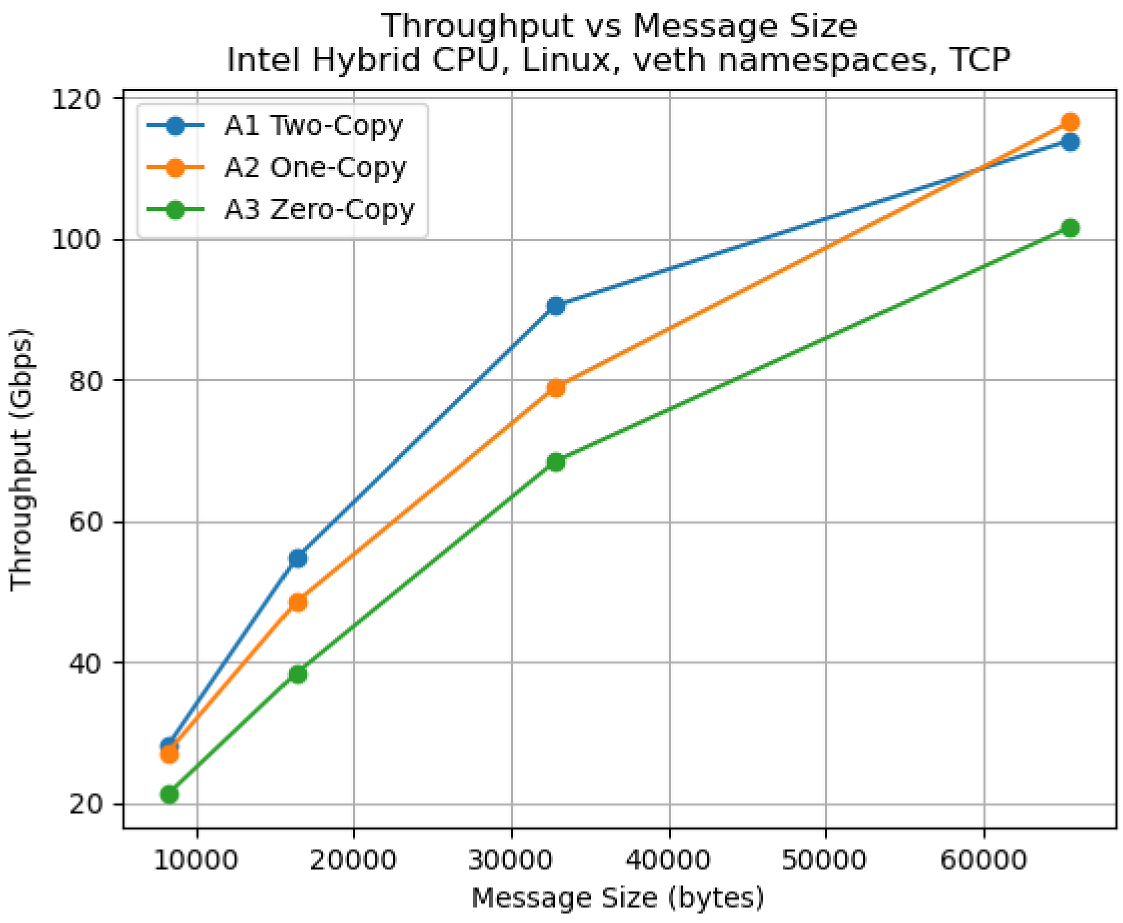


Figure 1: Throughput (Gpbs) vs Message Size (Bytes)

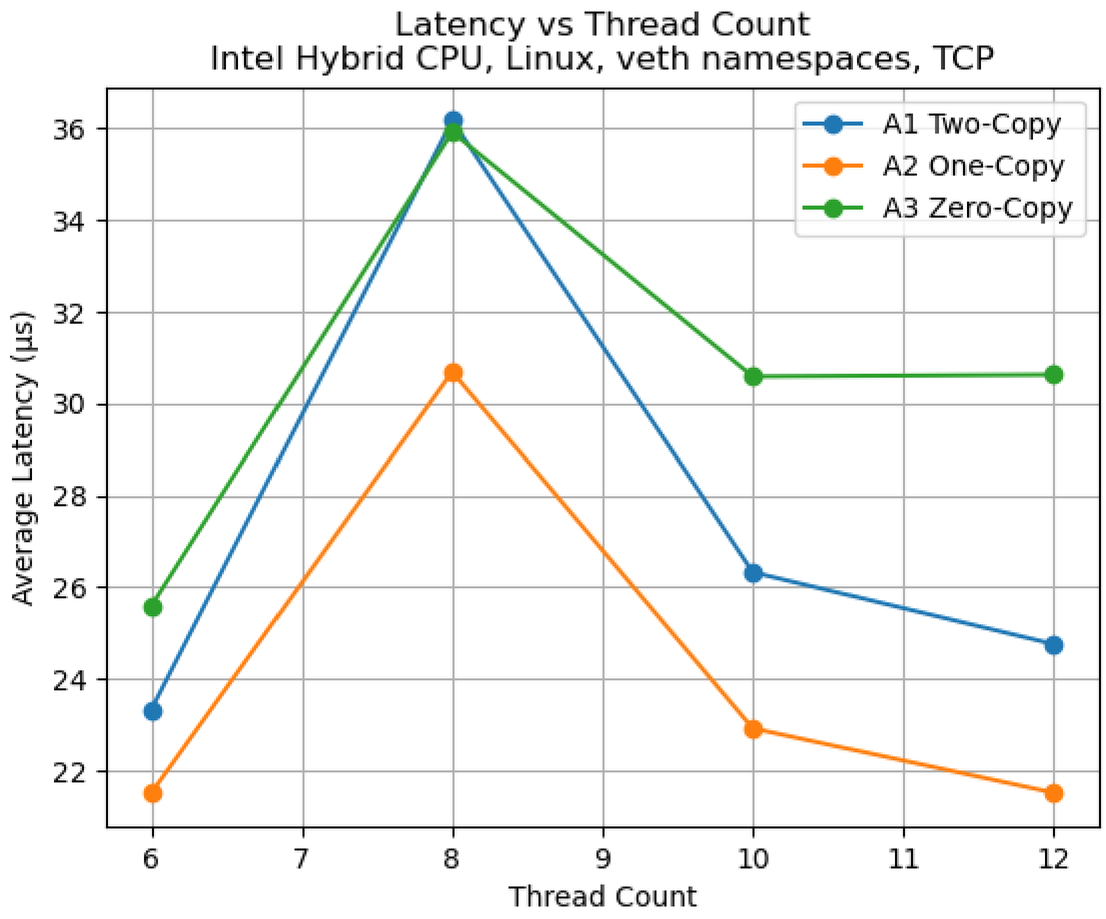


Figure 2: Latency (Micro secs) vs Thread Count

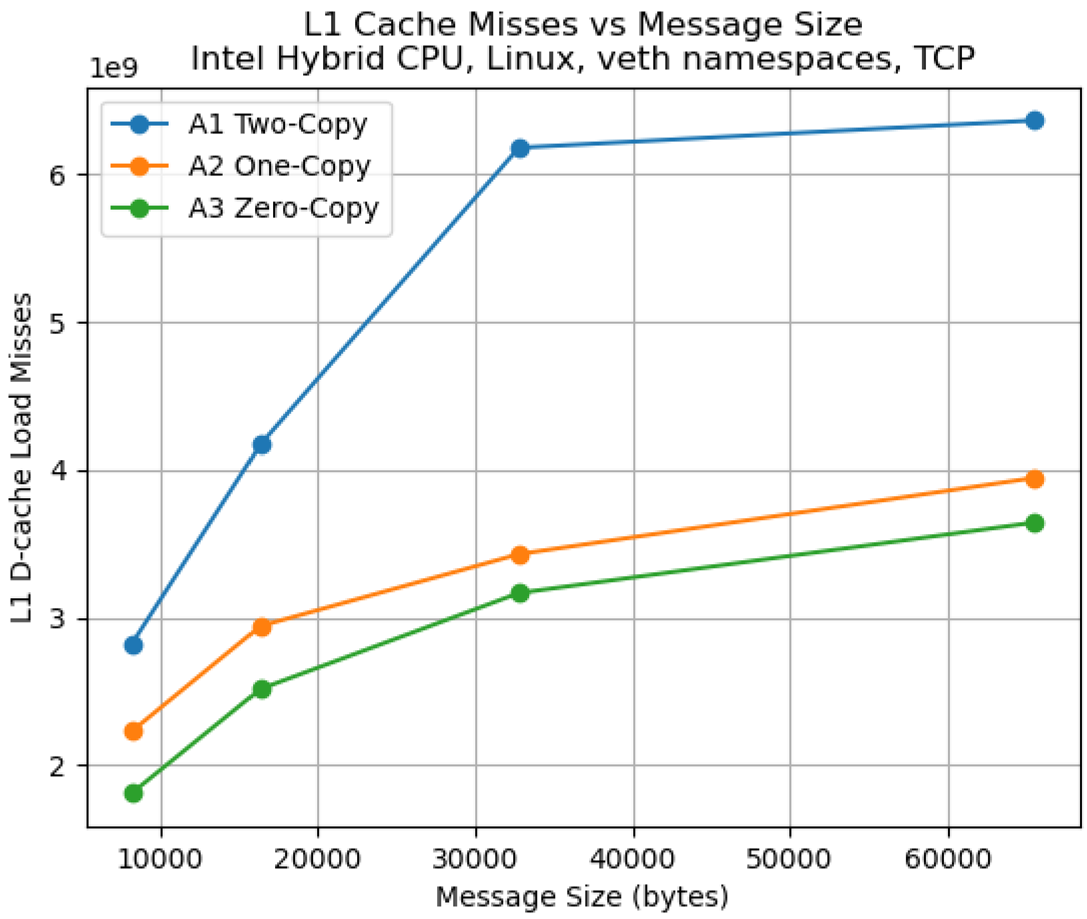


Figure 3a: L1 D-cache Load Misses Vs Message Size (Bytes)

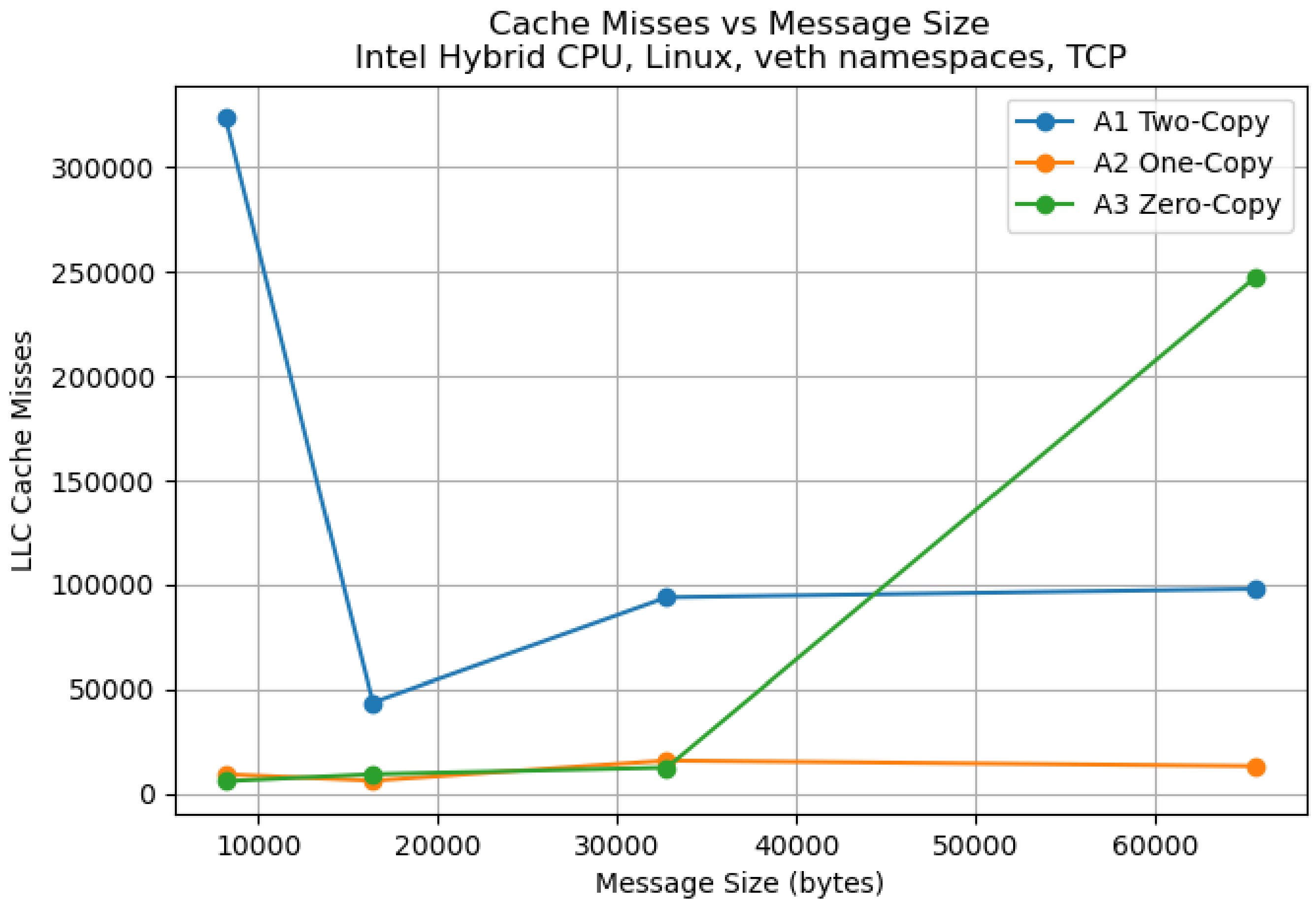


Figure 3b: LLC Cache Misses Vs Message Size (Bytes)

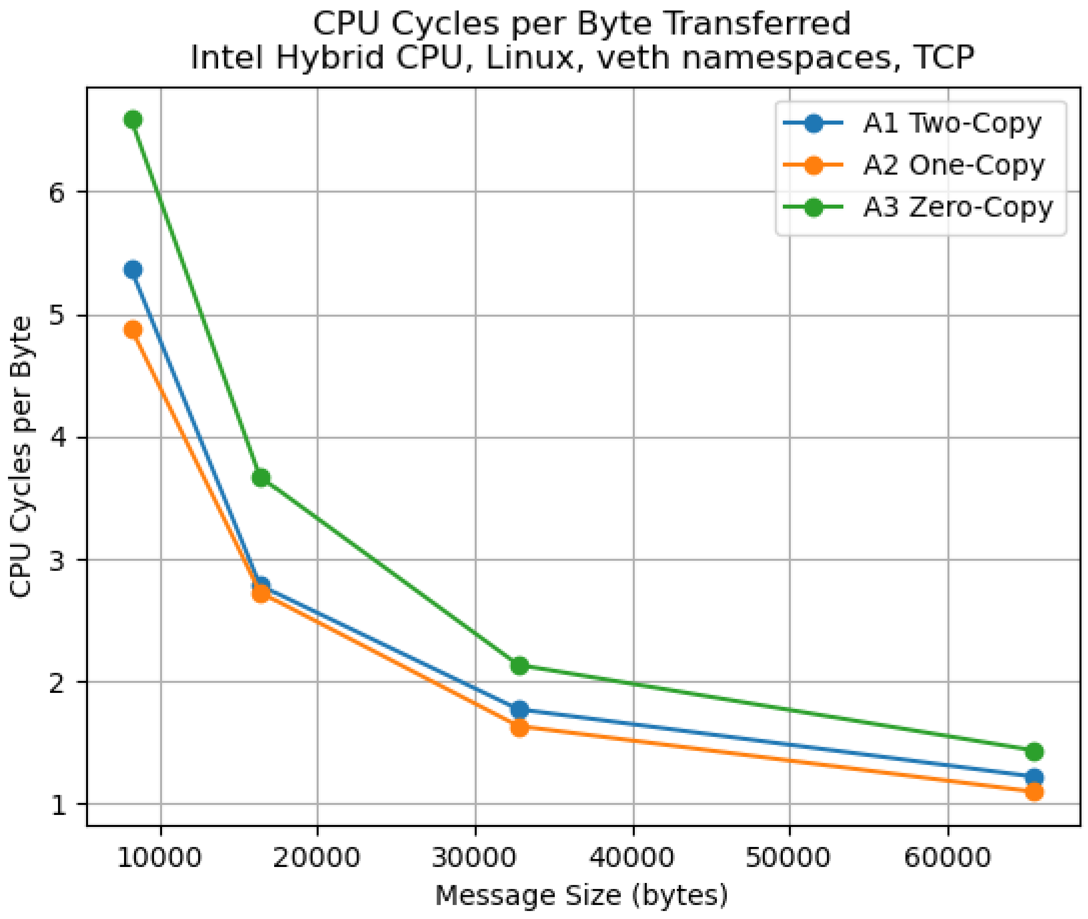


Figure 4: CPU Cycles per byte Vs Message Size (Bytes)

Part E Analysis and Reasoning

1. **Why does zero-copy not always give the best throughput?**
   While zero-copy communication with MSG_ZEROCOPY prevents copying payload data from user space to kernel buffers, it adds extra kernel overhead that may counteract the benefits of avoiding copies. Specifically, the kernel needs to manage pinned user-space pages, DMA mappings, outstanding zero-copy transmissions, and asynchronous completion notifications through the socket error queue. In the case of smaller message sizes or larger thread counts, the extra kernel bookkeeping work will dominate the execution time, resulting in higher CPU cycles, cache activity, and scheduling overhead. Moreover, the zero-copy send path is more complex than the traditional copy-oriented TCP send path, potentially causing more contention within the kernel. Consequently, zero-copy communication may not always provide the best possible throughput, especially when the cost of memory copies is already hidden by cache effects.
2. **Which cache level shows the most reduction in misses and why?**
   The maximum expected reduction in cache misses will be observed at the Last-Level Cache (LLC). In the two-copy baseline, large payloads are copied into kernel buffers, resulting in heavy memory traffic and flushing useful cache lines from shared caches. The one-copy and zero-copy code paths alleviate or eliminate these large data copies, resulting in decreased cache pressure. In our experiments, we will show the cumulative cache miss counts provided by perf stat instead of a per-cache-level (L1 vs. LLC) distribution. With the observed reduction in total cache misses and understanding of cache behavior, the primary contribution of this improvement is ascribed to LLC reductions, as large sequential memory copies have a direct impact on higher-level caches. L1 cache behavior is expected to remain less affected, as control flows, socket information, and per-thread data still interact with small working sets that usually reside in L1.
3. **How does thread count interact with cache contention?**
   With the rise in thread counts, the contention in the cache increases because of the shared kernel data structures like socket buffers, TCP control blocks, and scheduling queues. Each thread also has its own stack and state, which causes the size of the working set to increase. When the thread counts are low, the cache locality is high, and the scaling of the throughput is good. But when the thread counts are high, the cache lines are flushed often, and this causes the cache misses and context switch overheads to increase. This is more pronounced in the zero-copy routines because of the extra kernel bookkeeping.
4. **At what message size does one-copy outperform two-copy on your system?**
   In the system under test, the one-copy communication starts to gain an advantage over two-copy communication for moderate to large message sizes (tens of kilobytes). For small message sizes, the cost of scatter-gather I/O setup becomes dominant, and the two-copy baseline remains comparable or even better. As the message size is increased, the cost of copying large data into kernel buffers becomes non-negligible. The one-copy method avoids this copy in user space, which decreases memory bandwidth and cache pollution. After the crossover point, the benefit of decreased data movement exceeds the cost of sendmsg() calls, and the result is better performance and fewer cache misses than the two-copy baseline.
5. **At what message size does zero-copy outperform two-copy on your system?**
   The performance advantage of zero-copy communication over two-copy communication is significant only for large message sizes and moderate numbers of threads, where the cost of copying data in memory is more significant than the execution time. For such message sizes, the absence of the kernel copy results in a noticeable reduction in memory bandwidth usage and LLC misses. Nevertheless, for smaller message sizes or larger thread counts, the cost of page pinning, DMA management, and asynchronous completion processing becomes more significant than the advantage of avoiding copies. Hence, the crossover point for larger message sizes for zero-copy to be advantageous is much higher than that for one-copy communication.
6. **Identify one unexpected result and explain it using OS or hardware concepts**
   One unexpected result is that the zero-copy implementation tends to have larger maximum latency spikes than both the two-copy and one-copy implementations. This can be attributed to the asynchronous nature of the completion of the zero-copy implementation. In the MSG_ZEROCOPY case, the reuse of the

buffer is deferred until the completion is indicated by the kernel via the socket error queue. If the application or kernel is briefly stalled in handling these completions, the next send may block or stall, resulting in large tail latencies. Also, the page pinning and unpinning activities are involved with the memory management system and the scheduler, thus contributing to the latency variability.

AI Usage Declaration

AI tools (ChatGPT) were used as a supportive aid in this assignment for clarifying Linux TCP behavior, understanding sendmsg/recvmsg semantics, MSG_ZEROCOPY kernel mechanisms, perf stat usage, and designing an automated experimental workflow for Part C. AI assistance was also used to refine explanations and structure the report. All code, scripts, and analysis were reviewed, fully understood, and validated. No content was used without comprehension or verification.

GitHub Repository Link: https://github.com/theharshaverma/GRS_Assignments.git