PA02 — Network I/O Microbenchmarking

Graduate Systems (CSE638)

Student Name: Satatya De
Roll No: MT25084
Semester: 2
Date: February 07, 2026
Github repo link : https://github.com/satatya/GRS_PA02

AI Usage Declaration:
I have used AI in generating the shell codes of part A1,A2,A3 and B. I also used AI for giving me the commands for executing the parts A to D properly. The report is 75% written by me on my own words, the rest 25% is AI for the analysis and giving me the data, tables, etc. I used AI for the Readme file in github. AI was also used in part C and D but not completely, since I tried writing the code for python plots and all, and used AI for rectification. I used AI for fixing the errors that occurred in between, initially starting the scripts but using AI to fix it to work properly.

Table of Contents

1. Overview and objectives

This programming assignment evaluates how different Linux socket send paths affect CPU activity, cache behavior, and end-to-end throughput/latency when transferring data over a TCP connection. The work is split into four parts:

- Part A: Implement three server/client variants (baseline, one-copy, and zero-copy).
- Part B: Collect hardware/software counters using perf.
- Part C: Automate a grid of experiments (message sizes × thread counts × implementations).
- Part D: Derive metrics and generate plots, then interpret results in a short analysis.

All experiments were run locally on the lab Ubuntu machine using Linux network namespaces to create an isolated client/server topology on a single host.

2. Experimental environment and setup

Everything was done using the command line over SSH. Did not use GUI since connected remotely from local. The key idea is to create two network namespaces (ns_srv and ns_cli) connected by a veth pair, so the client and server behave like separate machines while still running on the same host.

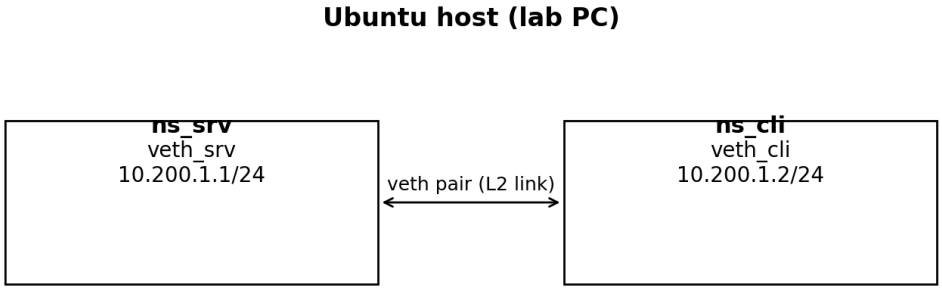


Figure 1 — Local topology using network namespaces and a veth pair.

2.1 Safety / impact on the lab machine

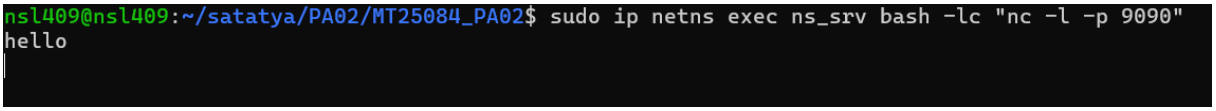
The namespace setup does NOT change the host's real network configuration. It creates virtual interfaces and virtual routing inside ns_srv and ns_cli only.

2.2 Namespace commands (CLI)

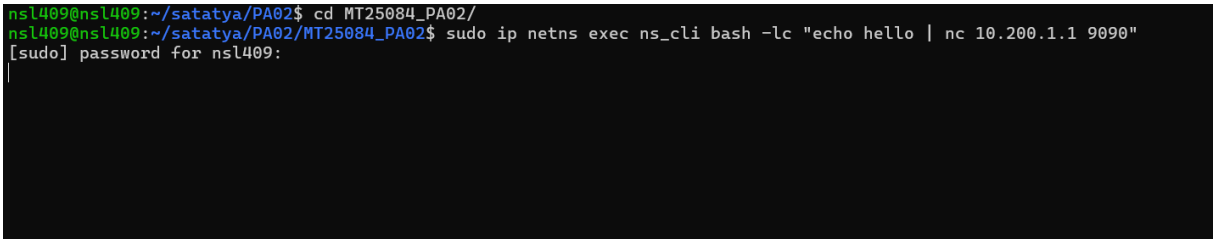
```
# Create namespaces
sudo ip netns add ns_srv
sudo ip netns add ns_cli

# Create a veth pair and move each end into a namespace
sudo ip link add veth_srv type veth peer name veth_cli
sudo ip link set veth_srv netns ns_srv
sudo ip link set veth_cli netns ns_cli

# Assign IPs inside the namespaces
sudo ip -n ns_srv addr add 10.200.1.1/24 dev veth_srv
sudo ip -n ns_cli addr add 10.200.1.2/24 dev veth_cli

# Bring interfaces up
sudo ip -n ns_srv link set lo up
```

sudo ip -n ns_cli link set lo up
sudo ip -n ns_srv link set veth_srv up
sudo ip -n ns_cli link set veth_cli up

Quick sanity check (optional): using netcat, run a listener in ns_srv and send 'hello' from ns_cli.



Screenshot — netcat listener inside ns_srv receiving 'hello'.



Screenshot — netcat client inside ns_cli sending 'hello'.

3. Implementations (Part A)

All three variants use TCP sockets. The server listens on a fixed port (9090 in the experiment scripts), accepts a specified number of client connections, and then transmits fixed-size messages for a fixed duration. Each client receives data for the same duration and prints a SUMMARY line with bytes, elapsed seconds, throughput, and an average per-message receive time (used as a latency proxy).

A1 (send/recv baseline)

User buffer → Kernel socket buffer → NIC (DMA) → Kernel rx buffer → User buffer
Copies (typical): user→kernel copy on send, kernel→user copy on recv.
Plus extra internal copies may happen (skb linearization, segmentation, etc.).

A2 (sendmsg + pre-registered/reused buffer)

Same kernel copies as A1, but avoids extra user-space copying by reusing a prepared buffer (and can use scatter/gather iovec instead of building a contiguous temp buffer).

A3 (sendmsg + MSG_ZEROCOPY)

Kernel pins user pages and can DMA directly from user pages (zero-copy send path).
Still requires bookkeeping and completion handling; small messages can be slower.

Figure 2 — Where copies happen in A1 vs A2 vs A3 (conceptual view).

3.1 A1 — Two-copy baseline (send/recv)

A1 uses the simplest primitives: the server repeatedly calls send() with a user buffer; the client repeatedly calls recv(). In the usual TCP path, the payload is copied from user space into kernel socket buffers on send, and copied from kernel buffers back into user space on recv. These are the two ‘major’ CPU copies most people refer to.
However, the kernel may perform extra internal copies in some cases like when data cannot be kept as

references, so it is safest to say: two copies are guaranteed at the user/kernel boundaries, and additional copies may occur depending on the stack.

3.2 A2 — One-copy implementation (sendmsg + reused buffer)

A2 switches to sendmsg() and uses a pre-allocated, reused transmit buffer. The practical goal is to remove any extra user-space copying that might otherwise occur when building messages (e.g., copying into a temporary contiguous buffer). With sendmsg() + iovec, scatter/gather can be used to describe the payload without first re-packing it.
Important note: unless the kernel is doing a true zero-copy send path, the user→kernel copy still exists. So the 'copy eliminated' in A2 is the application-side copy (user→user), not the mandatory user→kernel copy.

3.3 A3 — Zero-copy implementation (sendmsg + MSG_ZEROCOPY)

A3 uses sendmsg() with MSG_ZEROCOPY. When supported and enabled, the kernel can transmit data by DMA directly from user pages without copying the payload into kernel socket buffers. This reduces memory bandwidth pressure, but introduces overhead for pinning pages and for completion bookkeeping (the kernel must later notify when the user pages are safe to reuse).
In practice, zero-copy can be beneficial for larger messages, while for small messages the extra bookkeeping can outweigh copy savings.

4. Measurement methodology (Part B)

Two types of metrics are collected:
1) Application-level metrics from clients: total bytes, messages, throughput (Gb/s), and average per-message receive time.
2) System-level counters from the server process: cycles, context switches, cache misses, L1-dcache-load-misses, and LLC-load-misses.
perf stat is run around the server so that counters correspond to the workload driving the network sends.

4.1 Verifying perf events on the machine

The available perf event names were verified using:

```
perf list | grep -E 'cycles|context-switches|cache-misses|L1-dcache-load-misses|LLC-load-misses'
```

4.2 A1 example run with perf stat (manual)

Server terminal (inside ns_srv):

```
sudo ip netns exec ns_srv perf stat \
  -e cycles,context-switches,cache-misses,L1-dcache-load-misses,LLC-load-misses \
  ./MT25084_Part_A1_Server 9090 1024 10 1
```

Client terminal(s) (inside ns_cli):

```
sudo ip netns exec ns_cli ./MT25084_Part_A1_Client 10.200.1.1 9090 1024 10
```

```
[A1 Server] listening on port 9090 | msg_size=1024 | duration=10s

 Performance counter stats for './MT25084_Part_A1_Server 9090 1024 10':

     5,166,381,619      cycles
           187,671      context-switches
        88,616,316      cache-references
             5,099      cache-misses                     #    0.01% of all cache refs

      10.001343347 seconds time elapsed

       0.470136000 seconds user
       5.190304000 seconds sys

~
~
~
~
~
~
~
~
~
~
~
~
~
~
"perf_A1_m1024_t4.txt" [readonly] 15L, 627B                                   15,0-1
```

Screenshot — perf stat around A1 server showing counters after a 10s run.

```
nsl409@nsl409:~/satatya/PA02/MT25084_PA02$ sudo ip netns exec ns_srv bash -lc '
cd /home/nsl409/satatya/PA02/MT25084_PA02 &&
rm -f perf_A1_m1024_t4.txt &&
timeout -k 1s 12s perf stat -e cycles,context-switches,cache-references,cache-misses \
./MT25084_Part_A1_Server 9090 1024 10 2>&1 | tee perf_A1_m1024_t4.txt
'
[A1 Server] listening on port 9090 | msg_size=1024 | duration=10s

 Performance counter stats for './MT25084_Part_A1_Server 9090 1024 10':

     5,166,381,619      cycles
           187,671      context-switches
        88,616,316      cache-references
             5,099      cache-misses                     #    0.01% of all cache refs

      10.001343347 seconds time elapsed

       0.470136000 seconds user
       5.190304000 seconds sys


nsl409@nsl409:~/satatya/PA02/MT25084_PA02$
```

Screenshot shows the Port 9090 listening for 12s recorded the performance from the client

4.2 A1 example run with multiple clients

```
nsl409@nsl409:~/satatya/PA02/MT25084_PA02$ sudo ip netns exec ns_cli bash -lc "echo hello | nc 10.200.1.1 9090"
[sudo] password for nsl409:
nsl409@nsl409:~/satatya/PA02/MT25084_PA02$ sudo ip netns exec ns_cli ./MT25084_Part_A1_Client 10.200.1.1 9090 1024 10
[sudo] password for nsl409:
SUMMARY bytes=2292867072 seconds=10.000006 gbps=1.834293 msgs=2239128 avg_oneway_us=9.478
nsl409@nsl409:~/satatya/PA02/MT25084_PA02$ for i in 1 2 3 4; do
  sudo ip netns exec ns_cli ./MT25084_Part_A1_Client 10.200.1.1 9090 1024 10 &
done
wait
[1] 1577999
[2] 1578000
[3] 1578001
[4] 1578002
SUMMARY bytes=1953489920 seconds=10.000005 gbps=1.562791 msgs=1907705 avg_oneway_us=11.110
SUMMARY bytes=1920385024 seconds=10.000003 gbps=1.536308 msgs=1875376 avg_oneway_us=11.602
SUMMARY bytes=1926992896 seconds=10.000004 gbps=1.541594 msgs=1881829 avg_oneway_us=11.257
SUMMARY bytes=1946350592 seconds=10.000005 gbps=1.557080 msgs=1900733 avg_oneway_us=22.446
[1]   Done                    sudo ip netns exec ns_cli ./MT25084_Part_A1_Client 10.200.1.1 9090 1024 10
[2]   Done                    sudo ip netns exec ns_cli ./MT25084_Part_A1_Client 10.200.1.1 9090 1024 10
[3]-  Done                    sudo ip netns exec ns_cli ./MT25084_Part_A1_Client 10.200.1.1 9090 1024 10
[4]+  Done                    sudo ip netns exec ns_cli ./MT25084_Part_A1_Client 10.200.1.1 9090 1024 10
nsl409@nsl409:~/satatya/PA02/MT25084_PA02$ for i in 1 2 3 4; do
  sudo ip netns exec ns_cli ./MT25084_Part_A1_Client 10.200.1.1 9090 1024 10 &
done
wait
[1] 1578437
[2] 1578438
[3] 1578439
[4] 1578440
SUMMARY bytes=2292867072 seconds=10.000006 gbps=1.834293 msgs=2239128 avg_oneway_us=9.478
SUMMARY bytes=1976350720 seconds=10.000007 gbps=1.581080 msgs=1930030 avg_oneway_us=10.909
SUMMARY bytes=1959048192 seconds=10.000012 gbps=1.567237 msgs=1913133 avg_oneway_us=11.204
SUMMARY bytes=1933041664 seconds=10.000001 gbps=1.546433 msgs=1887736 avg_oneway_us=11.158
SUMMARY bytes=1984156672 seconds=10.000001 gbps=1.587325 msgs=1937653 avg_oneway_us=11.857
[1]   Done                    sudo ip netns exec ns_cli ./MT25084_Part_A1_Client 10.200.1.1 9090 1024 10
```

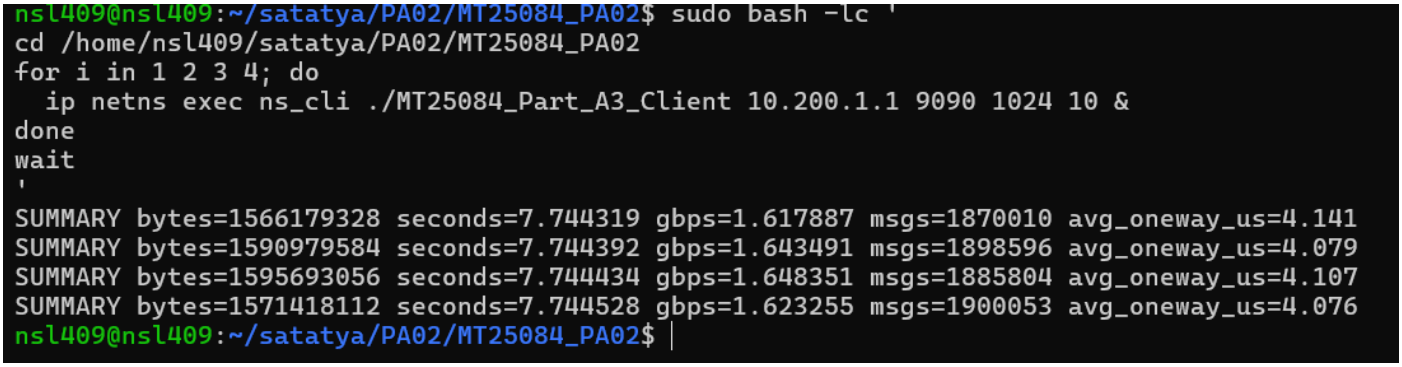
```
[A1 Server] listening on port 9090 | msg_size=1024 | duration=10s

 Performance counter stats for './MT25084_Part_A1_Server 9090 1024 10':

     5,166,381,619      cycles
           187,671      context-switches
        88,616,316      cache-references
             5,099      cache-misses                     #    0.01% of all cache refs

      10.001343347 seconds time elapsed

       0.470136000 seconds user
       5.190304000 seconds sys

"perf_A1_m1024_t4.txt" [readonly] 15L, 627B                                   15,0-1
```

4.2 A2 & A3 example run (manual)

Similarly, I executed A2 and A3 with single client and multiple clients. The goal of A2 is to reduce one avoidable copy compared to the baseline approach by using sendmsg() with a pre-allocated, reusable buffer (sometimes called a “pre-registered” buffer in the assignment). The key idea is: instead of building/assembling payloads into a temporary staging buffer and then copying them again into a send buffer, the application keeps one stable payload buffer and hands that buffer to the kernel via an iovec.

In my A2 implementation, the server still streams data to the clients for a fixed duration, but the send path uses scatter/gather (sendmsg) over a persistent payload buffer, so the application can avoid an extra user-space copy when constructing outgoing messages.

```
[A2 Server] listening on port 9090 | msg_size=1024 | duration=10s | clients=4

 Performance counter stats for './MT25084_Part_A2_Server 9090 1024 10 4':

    92,067,412,013      cycles
               132      context-switches
       497,134,141      cache-references
            13,965      cache-misses                     #    0.00% of all cache refs

      10.001656203 seconds time elapsed

       3.322813000 seconds user
      29.616341000 seconds sys

"perf_A2_m1024_t4.txt" [readonly] 15L, 641B
```



Screenshot shows the Port 9090 listening for 12s recorded the performance from the client for A2 code

In a typical socket I/O pipeline, there are always two unavoidable memory copies: data moves from user space into the kernel's socket buffer on the sender side, and then back from the kernel buffer into user space on the receiver side. That's just how standard sockets work. What A2 changes isn't those kernel-level copies—it can't remove them—but rather the extra, often accidental copies that applications introduce themselves. For example, many programs first build a payload in a temporary buffer, then copy it again into a send buffer, or they stitch headers and payload together into a new contiguous block before sending. A2 avoids this by letting you reference stable buffers directly with sendmsg and iovec, so you don't waste cycles on redundant user-space memcpy calls. In short, it streamlines the application side of the process, cutting out unnecessary duplication while still relying on the kernel's required user↔kernel transfers.

Where exactly does the "removed copy" come from?

The "removed copy" is the extra user-space memcpy that many baseline implementations introduce before calling send(). In the common pattern, developers build the payload in a temporary buffer, then copy it into another buffer designated for sending, and only then hand it off to the kernel (which itself still does the required copy into the socket buffer). A2 avoids that middle step: instead of shuffling data around in user space, it keeps the payload in one stable buffer and uses sendmsg() with an iovec pointing directly at it. That way, the kernel still performs its necessary copy, but the redundant staging-to-payload memcpy in user space disappears.

What system calls and structures are central in A2?

- sendmsg() instead of send()
- struct iovec to describe the payload buffer
- struct msghdr to wrap the iovec

How I validated that A2 is behaving correctly

- The server accepts N clients and streams for the requested duration.
- Each client prints SUMMARY bytes=… gbps=… msgs=….
- perf stat captures CPU cycles, context switches, and cache miss counters for the server workload.

A3

A3 tackles the kernel-side copy on the **send path** by using sendmsg(..., MSG_ZEROCOPY). When the kernel supports this flag, instead of copying user data into the socket buffer (skb), it can pin the user's memory pages and hand those references directly to the networking stack. The NIC then transmits the data via DMA without requiring a CPU memcpy into kernel buffers. In other words, A3 achieves "zero-copy" for transmission because the kernel no longer duplicates the payload

into its own buffer—it simply references the original user-space memory pages safely and streams them out. This cuts out the CPU overhead of that kernel copy, making the send path more efficient.

Note: In my implementation, MSG_ZEROCOPY is attempted but may fall back
My A3 server tries sendmsg(..., MSG_ZEROCOPY) and falls back to normal send() if the kernel rejects it (typical errors: EINVAL, EOPNOTSUPP, ENOTSUP, EPERM).
So the experimental behavior is:

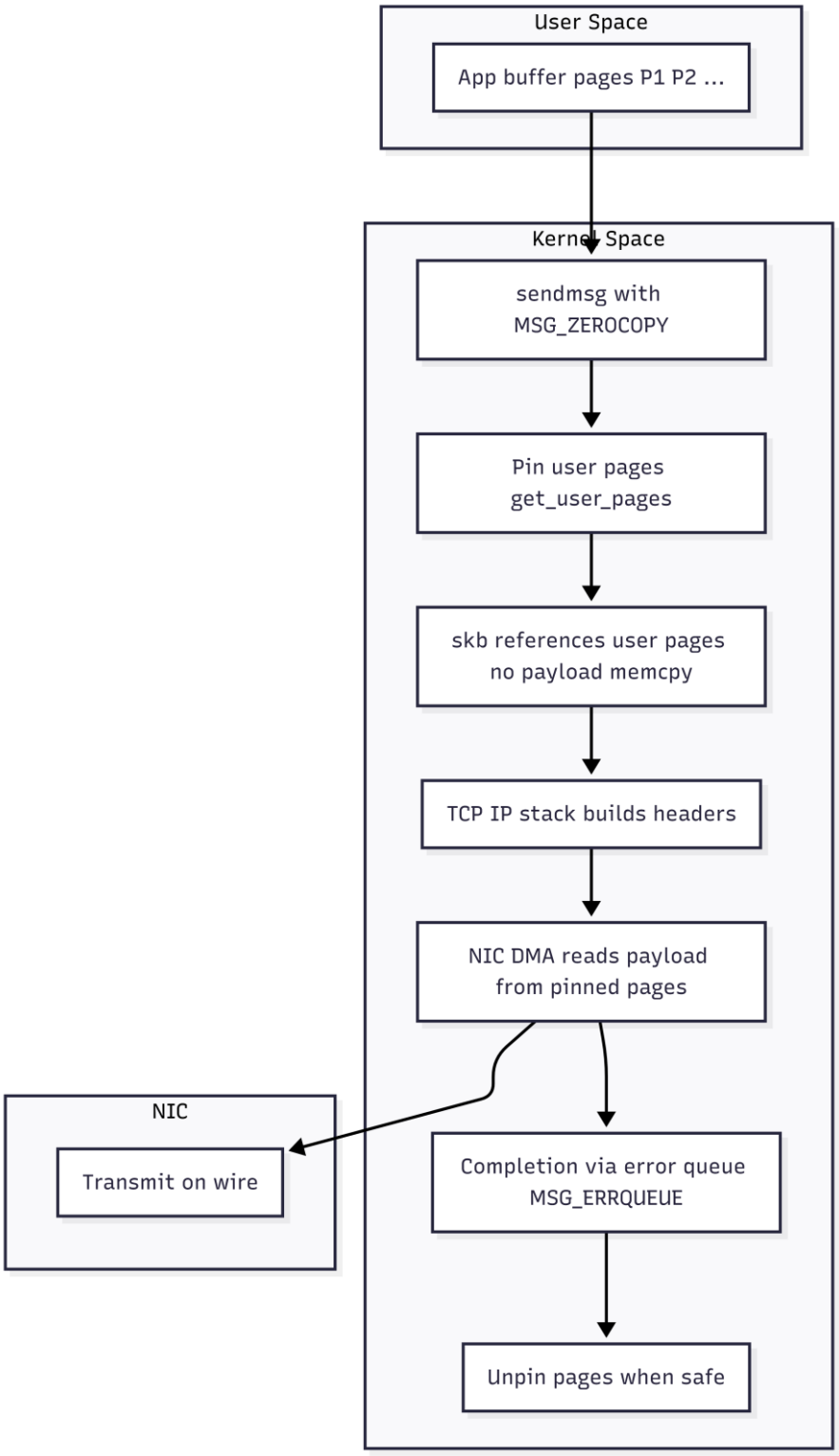
- If kernel supports / permits: “true” zero-copy transmit path is used
- Otherwise: it behaves like A1 (regular copy) for correctness

```
nsl409@nsl409:~/satatya/PA02/MT25084_PA02$ sudo ip netns exec ns_srv bash -lc '
cd /home/nsl409/satatya/PA02/MT25084_PA02 &&
rm -f perf_A3_m1024_t4.txt &&
timeout -k 1s 12s perf stat -e cycles,context-switches,cache-references,cache-misses \
./MT25084_Part_A3_Server 9090 1024 10 4 2>&1 | tee perf_A3_m1024_t4.txt
'
[A3 Server] listening on port 9090 | msg_size=1024 | duration=10s | clients=4

 Performance counter stats for './MT25084_Part_A3_Server 9090 1024 10 4':

    87,188,394,360      cycles
               97      context-switches
       506,134,249      cache-references
            19,268      cache-misses                     #    0.00% of all cache refs

      10.001640234 seconds time elapsed

       2.956972000 seconds user
      28.020741000 seconds sys
```

The Kernel behavior:

With sendmsg(MSG_ZEROCOPY), the kernel changes how it handles the payload after the application hands it off. Normally, the kernel copies user data into its own socket buffer so the user can reuse memory right away. In zero-copy mode, instead of duplicating the payload, the kernel pins the user’s memory pages, attaches those references to the skb, and lets the NIC fetch the data directly via DMA. The kernel still builds TCP/IP headers, manages segmentation, and tracks retransmissions, but the heavy CPU memcpy of the payload is skipped. To keep things safe, the kernel reports completion asynchronously through the socket’s error queue, and only then unpins the pages so the application can reuse them.

So the “extra bit” is that A3 doesn’t just cut out a memcpy—it introduces a whole mechanism of page pinning, DMA-based transmission, and completion notifications to guarantee correctness while eliminating the CPU copy on the send path.


User Space
App buffer pages P1 P2 ...
Kernel Space
sendmsg with MSG_ZEROCOPY
Pin user pages get_user_pages
skb references user pages no payload memcpy
TCP IP stack builds headers
NIC DMA reads payload from pinned pages
NIC
Transmit on wire
Completion via error queue MSG_ERRQUEUE
Unpin pages when safe

4.3 Capturing perf output to a file

perf writes its summary at the end of execution, so the output file is only filled after the server exits. For file output, the -o option is used:

```
sudo ip netns exec ns_srv perf stat \
  -e cycles,context-switches,cache-misses,L1-dcache-load-misses,LLC-load-misses \
  -o perf_A1_m1024_t1.txt \
  ./MT25084_Part_A1_Server 9090 1024 10 1
```

5. Part C — Automated experiment grid

Part C runs the full experiment grid over multiple message sizes and thread counts, executes each of A1/A2/A3, collects client SUMMARY outputs, collects perf counters, and writes a single CSV: MT25084_Part_C_results.csv.

```
nsl409@nsl409:~/satatya/PA02/MT25084_PA02$ wc -l MT25084_Part_C_results.csv
# should be 61 (60 + header)

ls -1 MT25084_Part_C_raw_*_perf.csv | wc -l
# should be 60

ls -1 MT25084_Part_C_raw_*_client*.log | wc -l
# should be 225  (5*3*(1+2+4+8)=225)
61 MT25084_Part_C_results.csv
60
225
nsl409@nsl409:~/satatya/PA02/MT25084_PA02$ grep -nE 'MSG_SIZES=|THREADS=|IMPLS=' MT25084_Part_C_Run_Experiments.sh
31:MSG_SIZES=(64 256 1024 4096 16384)
36:IMPLS=(A1 A2 A3)
nsl409@nsl409:~/satatya/PA02/MT25084_PA02$
```

```
nsl409@nsl409:~/satatya/PA02/MT25084_PA02$ cd /home/nsl409/satatya/PA02/MT25084_PA02
sudo ./MT25084_Part_C_Run_Experiments.sh
[sudo] password for nsl409:
[C] Setting up namespaces...
[C] Compiling all implementations...
[C] Running experiment grid...
[C] ==> Running A1 msg=64 threads=1 dur=10s
[C] ==> Running A2 msg=64 threads=1 dur=10s
[C] ==> Running A3 msg=64 threads=1 dur=10s
[C] ==> Running A1 msg=64 threads=2 dur=10s
[C] ==> Running A2 msg=64 threads=2 dur=10s
[C] ==> Running A3 msg=64 threads=2 dur=10s
[C] ==> Running A1 msg=64 threads=4 dur=10s
[C] ==> Running A2 msg=64 threads=4 dur=10s
[C] ==> Running A3 msg=64 threads=4 dur=10s
[C] ==> Running A1 msg=64 threads=8 dur=10s
[C] ==> Running A2 msg=64 threads=8 dur=10s
[C] ==> Running A3 msg=64 threads=8 dur=10s
[C] ==> Running A1 msg=256 threads=1 dur=10s
[C] ==> Running A2 msg=256 threads=1 dur=10s
[C] ==> Running A3 msg=256 threads=1 dur=10s
[C] ==> Running A1 msg=256 threads=2 dur=10s
[C] ==> Running A2 msg=256 threads=2 dur=10s
[C] ==> Running A3 msg=256 threads=2 dur=10s
[C] ==> Running A1 msg=256 threads=4 dur=10s
[C] ==> Running A2 msg=256 threads=4 dur=10s
[C] ==> Running A3 msg=256 threads=4 dur=10s
[C] ==> Running A1 msg=256 threads=8 dur=10s
[C] ==> Running A2 msg=256 threads=8 dur=10s
[C] ==> Running A3 msg=256 threads=8 dur=10s
[C] ==> Running A1 msg=1024 threads=1 dur=10s
[C] ==> Running A2 msg=1024 threads=1 dur=10s
[C] ==> Running A3 msg=1024 threads=1 dur=10s
[C] ==> Running A1 msg=1024 threads=2 dur=10s
[C] ==> Running A2 msg=1024 threads=2 dur=10s
[C] ==> Running A3 msg=1024 threads=2 dur=10s
[C] ==> Running A1 msg=1024 threads=4 dur=10s
[C] ==> Running A2 msg=1024 threads=4 dur=10s
```

In the final run(as shown in above ss), the following parameter sets were used:

Dimension	Values
Implementations	A1, A2, A3
Message sizes (bytes)	64, 256, 1024, 4096, 16384
Thread counts (#clients)	1, 2, 4, 8
Duration (s)	10

This produces 60 rows in the results CSV (impl × msg_size × threads). Because 3 implementations × 5 message sizes × 4 thread counts = 60, the grid meets the requirement of at least 4 message sizes and 4 thread counts.

5.1 Running Part C

cd /home/<user>/satatya/PA02/MT25084_PA02
sudo ./MT25084_Part_C_Run_Experiments.sh

The script sets up namespaces (idempotent), compiles binaries via the Makefile, runs each combination, and cleans up temporary logs. The final artifact is MT25084_Part_C_results.csv.

6. Part D — Post-processing and plots

Part D reads MT25084_Part_C_results.csv and writes a derived CSV with additional computed metrics, then generates plots as PNG and PDF files.
Derived metrics include cycles_per_byte, cache misses per GB transferred, and misses per million messages.

./MT25084_Part_D_Run.sh MT25084_Part_C_results.csv
Outputs:
- MT25084_Part_D_derived.csv
- MT25084_Part_D_plots/ (png + pdf)

6.1 Representative plots

Throughput vs message size

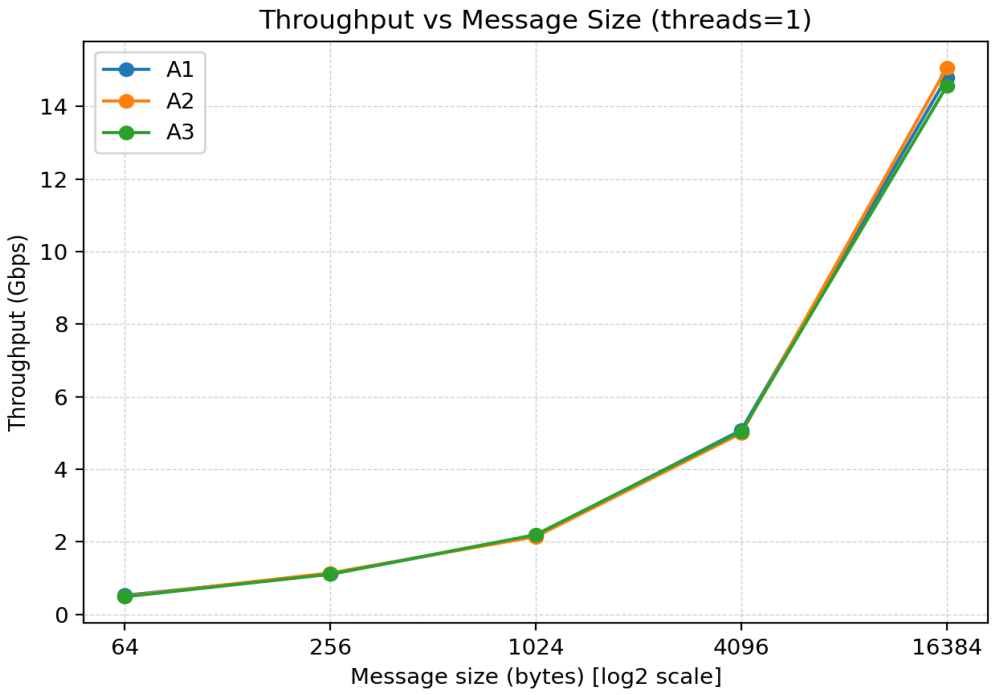


Figure — Total throughput (Gb/s) at threads=1.

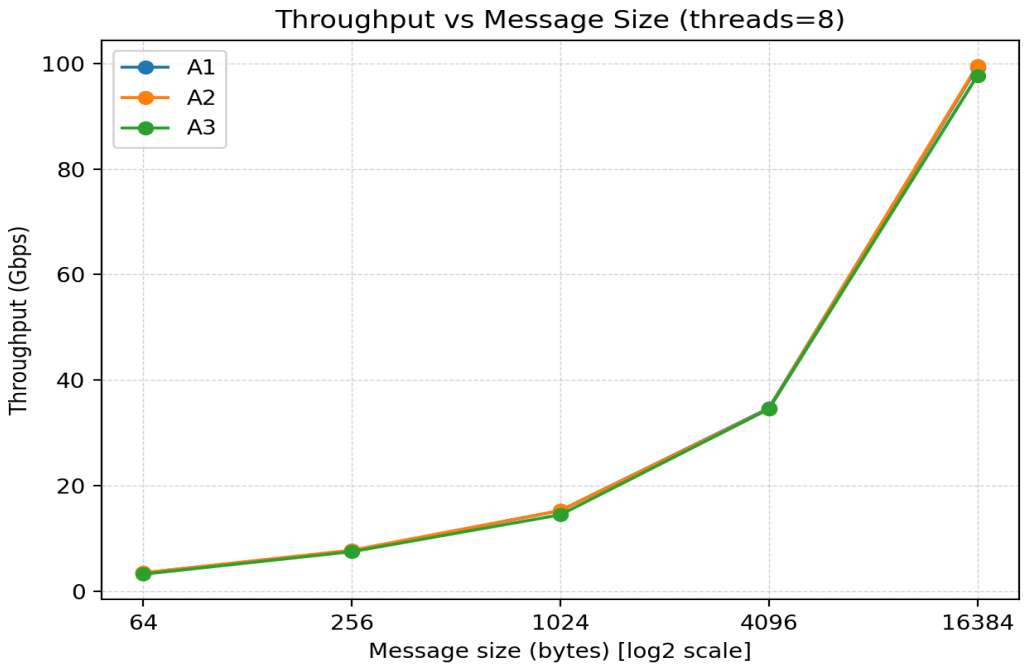


Figure — Total throughput (Gb/s) at threads=8.

Latency proxy (avg one-way µs) vs message size

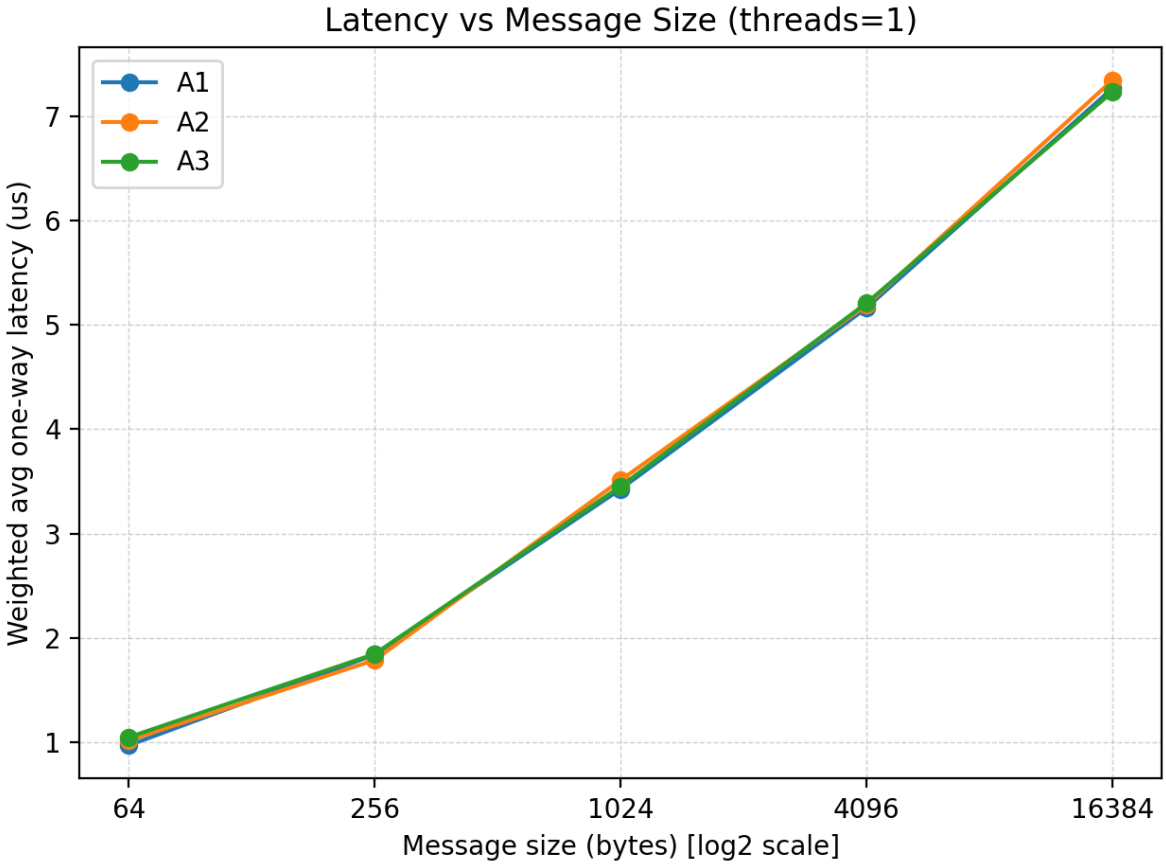


Figure — Avg per-message receive time at threads=1.

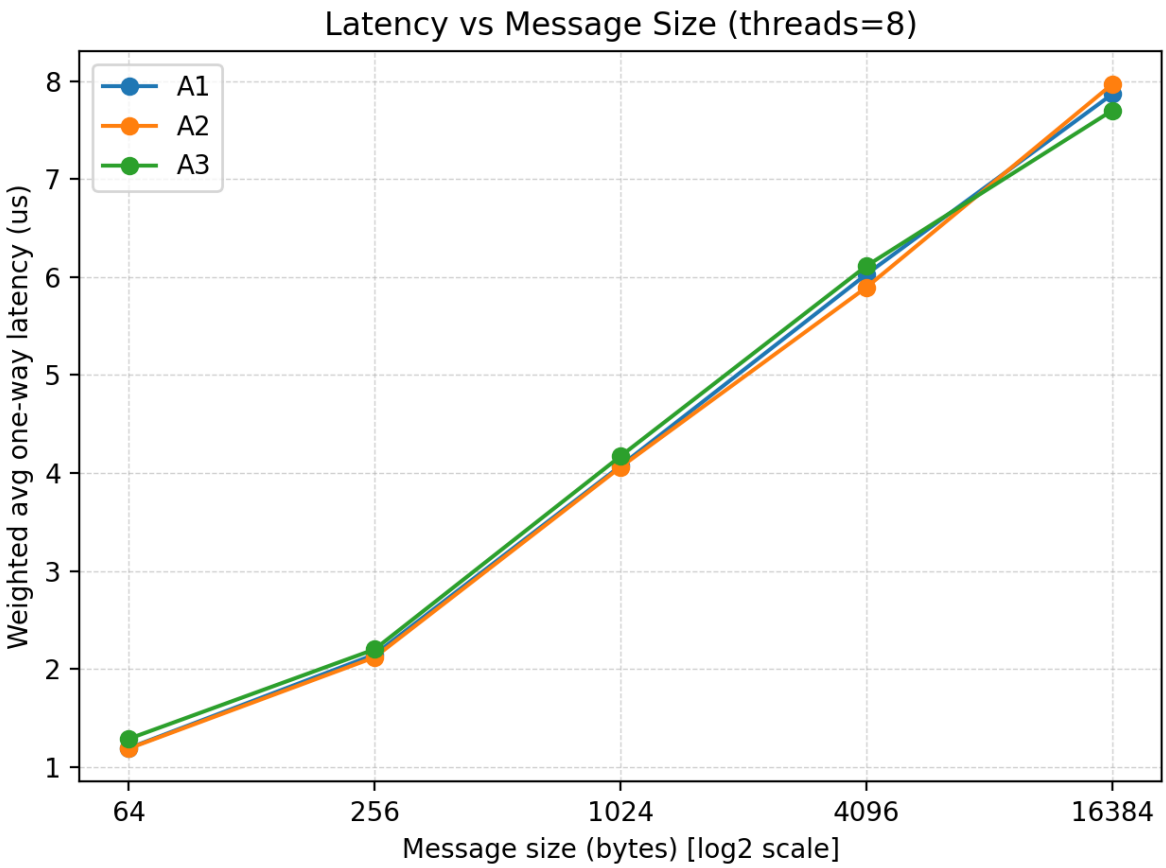


Figure — Avg per-message receive time at threads=8.

CPU efficiency (cycles/byte) vs message size

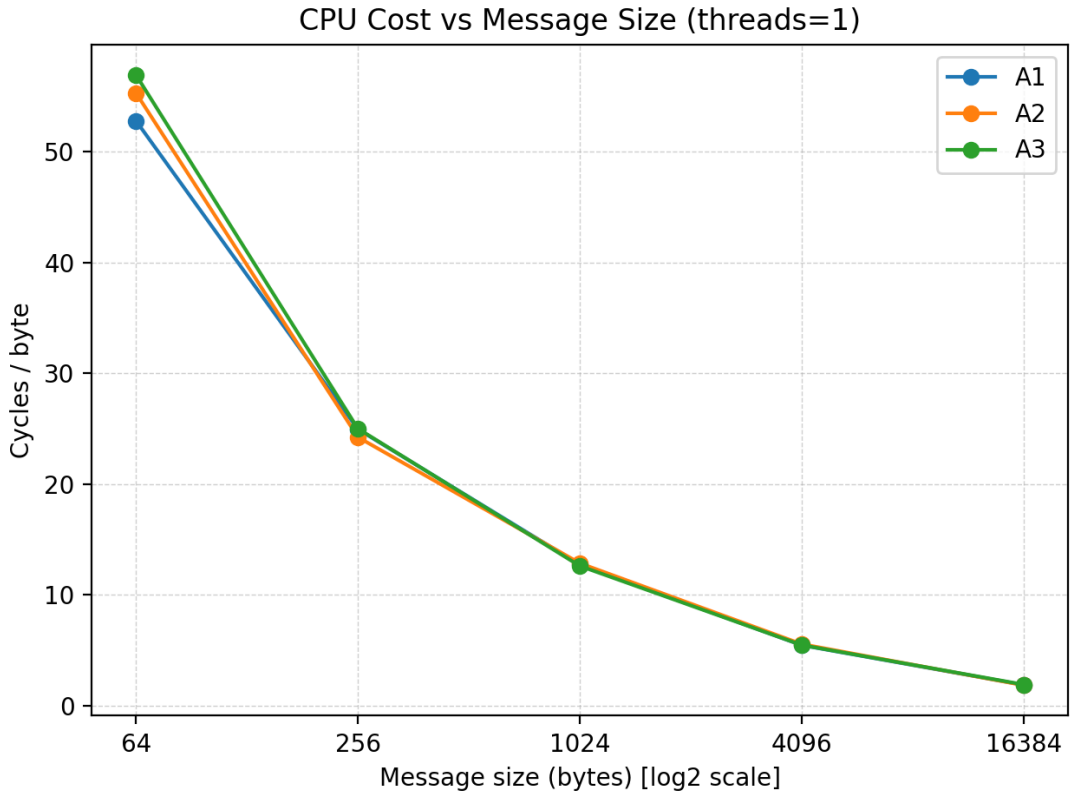


Figure — Cycles per byte at threads=1.

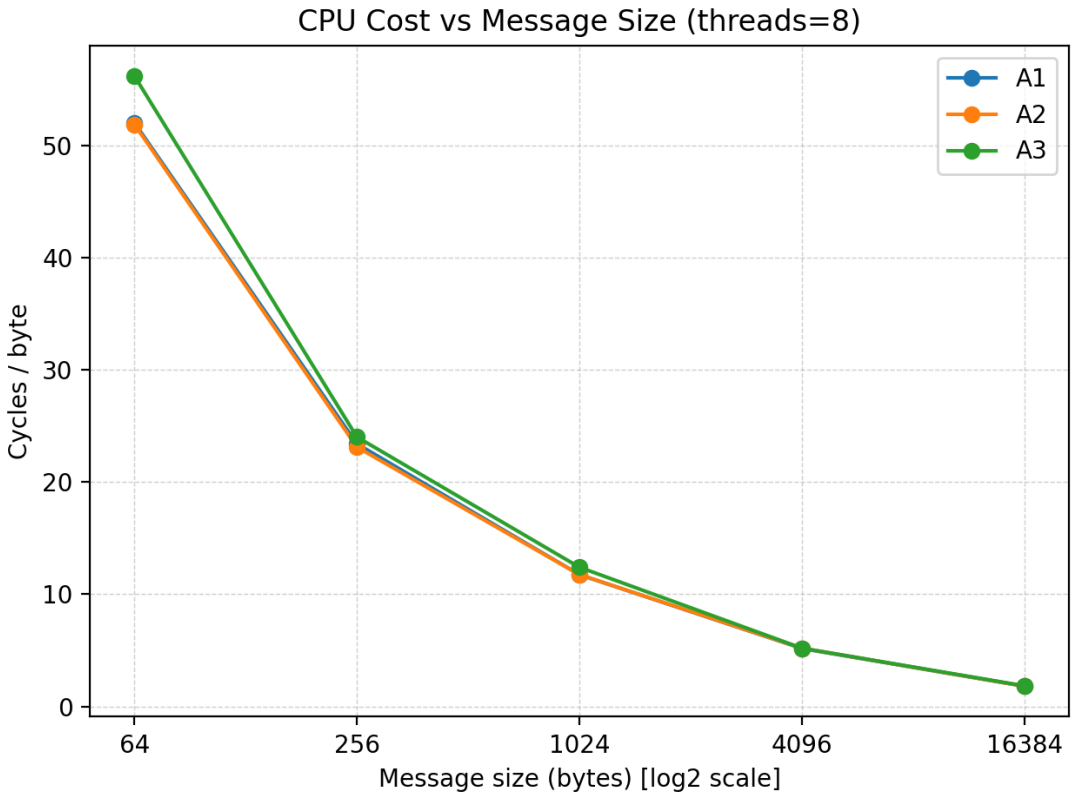


Figure — Cycles per byte at threads=8.

Last-level cache behavior (LLC misses per GB) vs message size

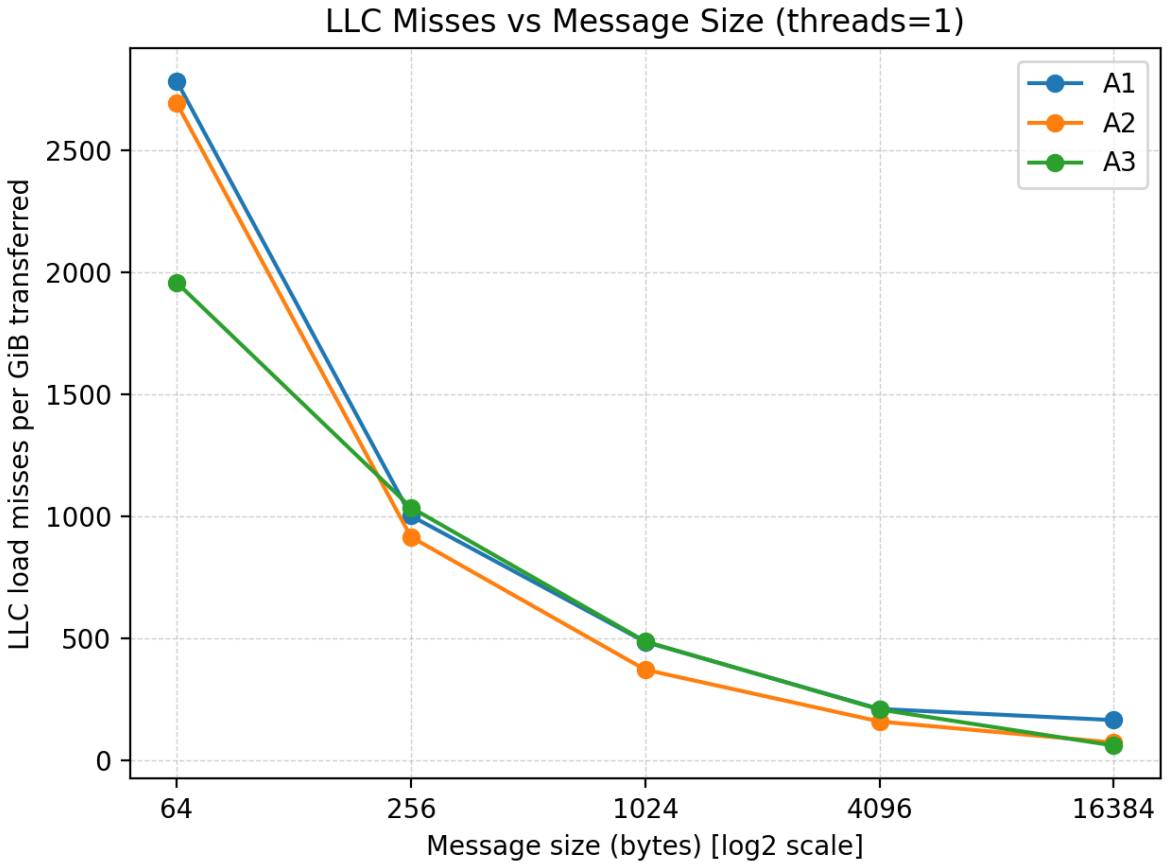


Figure — LLC misses per GB at threads=1.

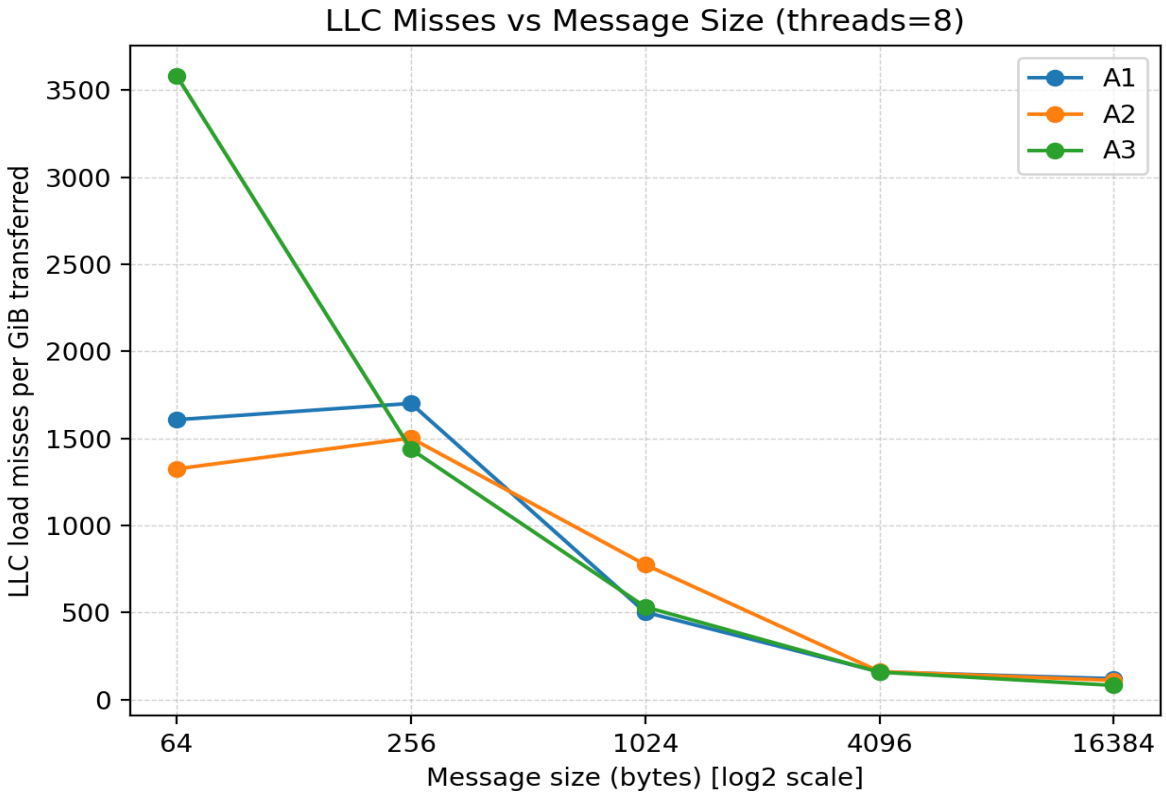


Figure — LLC misses per GB at threads=8.

7. Results and discussion

This section summarizes the observed behavior across A1/A2/A3 and answers the analysis questions for Part D.

7.1 When does A2/A3 outperform A1?

Using the measured total throughput (Gb/s) as the comparison metric, the following thresholds were observed (smallest tested message size where the implementation beats A1 at the same thread count):

threads	A2_beats_A1_at_msg>= (bytes)	A3_beats_A1_at_msg>= (bytes)
1	256	1024
2	64	4096
4	64	1024
8	64	never (in tested sizes)

Overall trend:

- A2 tends to match or slightly exceed A1 even at smaller message sizes, because it reduces extra copying/packing work in user space.
- A3 shows benefits more reliably at larger message sizes (≥ 1 KB in several thread settings). At high concurrency (threads=8), A3 did not beat A1 in the tested sizes, suggesting the zerocopy bookkeeping overhead or stack/NIC behavior dominates under contention.

7.2 Why isn't zero-copy always not always give the best throughput?

MSG_ZEROCOPY reduces payload copying, but it introduces work elsewhere:

• Page pinning and reference tracking: the kernel must ensure user pages are not freed/modified while the NIC is reading them.

• Completion notifications: the kernel maintains completion state and may deliver error-queue notifications.

• Smaller messages amplify fixed overheads: when the payload is tiny, syscall + zerocopy bookkeeping

cost can exceed the time saved by avoiding a memcpy.
• Interaction with TCP segmentation/offload: TCP may segment buffers, and depending on the NIC/driver, the path can still incur work (or even fall back).
These factors explain why A3 improves throughput for larger message sizes but can lose for small messages or high thread counts.

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

Cache-level effects (L1 vs LLC)

impl	L1_misses_per_gb	LLC_misses_per_gb
A1	1.65e+08	1.1e+03
A2	1.61e+08	762
A3	1.65e+08	867

Across the grid, the strongest improvement is visible in LLC misses per GB (A2 generally lowest, A1 highest). This matches intuition: reducing extra copying/packing reduces pressure on the shared last-level cache and memory bandwidth. L1 misses move less dramatically because the send buffer is streamed repeatedly (predictable access), while LLC reflects the larger working-set and contention effects.

7.4 How does thread count interact with cache contention?

Thread count vs cache contention

Increasing the number of clients (threads) increases total throughput up to a point, but also increases contention:
• More concurrent send loops compete for shared LLC capacity and memory bandwidth.
• More sockets increase kernel bookkeeping and can raise context-switch rates.
• At higher concurrency, the benefits of zero-copy can be offset by higher overhead and shared-resource contention.
This is visible in the plots where throughput scales with threads, but cycles/byte and LLC misses can rise for certain combinations.

7.5 Unexpected result (example)

A notable observation is that A3 did not outperform A1 at threads=8 for any tested message size. Given that A3's advantage comes from avoiding user→kernel copies, one might expect it to dominate at large payload sizes. On this system, the overhead of zerocopy bookkeeping (page pinning and completion tracking) combined with concurrency effects appears to remove that advantage. The likely interpretation is that at threads=8 the server becomes bottlenecked by kernel work and shared caches rather than by memcpy bandwidth.

7.6 Compact view at threads=4

The table below shows selected derived metrics for threads=4 across message sizes (throughput, latency proxy, cycles/byte, and miss rates). It is useful as a quick numeric cross-check against the plots.

impl	msg_size	total_gbps	weighted_avg_oneway_us	cycles_per_byte	cache_misses_per_gb	L1_misses_per_gb	LLC_misses_per_gb
A1	64	1.77708	1.15246	51.8881	21796.4	2.52388e+08	8424.94
A2	64	1.78886	1.14498	51.5878	6475.91	2.03699e+08	1528.58
A3	64	1.65243	1.23926	55.2813	7539.53	2.20424e+08	2181.47
A1	256	3.83329	2.13728	23.5137	7381.81	2.15936e+08	1588.16
A2	256	3.8000	2.15573	23.7103	3492	2.30673e+08	821.861

		4					
A3	256	3.69962	2.21439	24.3324	3326.74	2.20373e+08	1029.85
A1	1024	7.49118	4.09921	12.0176	1276.66	1.75831e+08	335.585
A2	1024	7.73207	3.98544	11.7847	1428.43	1.70156e+08	371.508
A3	1024	7.58233	4.04762	12.1869	1895.69	1.73817e+08	547.993
A1	4096	16.9678	5.87635	5.32723	454.577	1.18837e+08	127.4
A2	4096	17.3577	5.78738	5.24287	708.292	1.20647e+08	214.927
A3	4096	17.1746	6.16698	5.30238	507.35	1.21334e+08	135.492
A1	16384	50.1171	7.63688	1.80905	256.918	8.68598e+07	102.591
A2	16384	50.0942	7.56608	1.80355	263.661	8.68207e+07	92.15
A3	16384	50.4564	7.40012	1.83148	259.837	8.76405e+07	89.8049

8. Reproducibility checklist (commands)

Minimal end-to-end steps (fresh terminal session):

10. 1) Setup namespaces (once):

```
sudo ip netns add ns_srv
sudo ip netns add ns_cli
sudo ip link add veth_srv type veth peer name veth_cli
sudo ip link set veth_srv netns ns_srv
sudo ip link set veth_cli netns ns_cli
sudo ip -n ns_srv addr add 10.200.1.1/24 dev veth_srv
sudo ip -n ns_cli addr add 10.200.1.2/24 dev veth_cli
sudo ip -n ns_srv link set lo up
sudo ip -n ns_cli link set lo up
sudo ip -n ns_srv link set veth_srv up
sudo ip -n ns_cli link set veth_cli up
```

11. 2) Build:

```
make clean && make
```

12. 3) Manual run (A1 example):

```
sudo ip netns exec ns_srv ./MT25084_Part_A1_Server 9090 1024 10 4
for i in 1 2 3 4; do sudo ip netns exec ns_cli ./MT25084_Part_A1_Client 10.200.1.1 9090 1024 10 & done; wait
```

13. 4) Full grid:

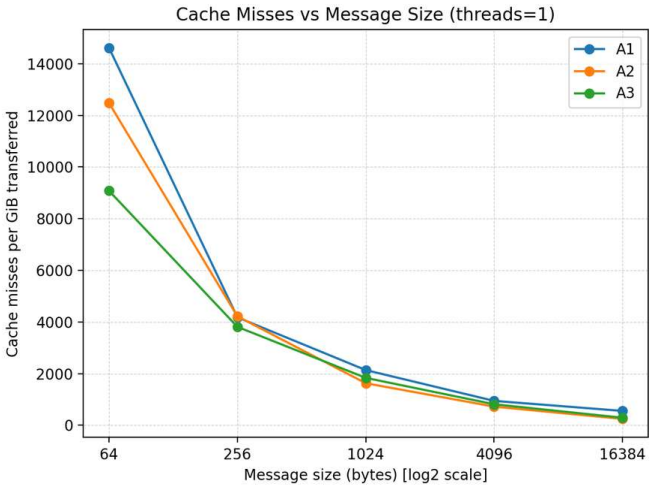
```
sudo ./MT25084_Part_C_Run_Experiments.sh
```

14. 5) Plots:

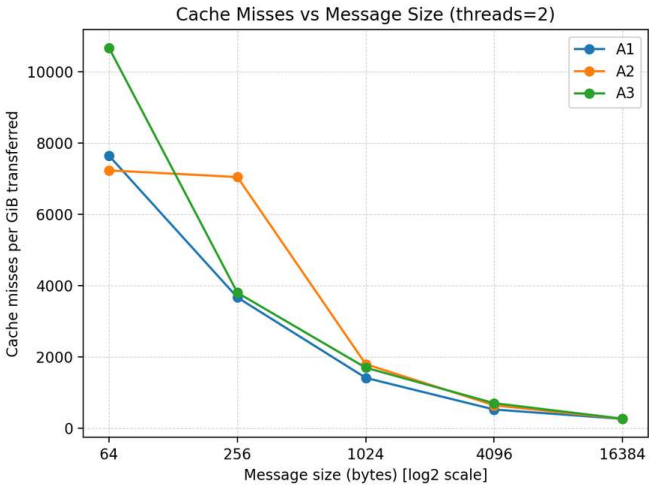
./MT25084_Part_D_Run.sh MT25084_Part_C_results.csv

Appendix — Screenshots for part D

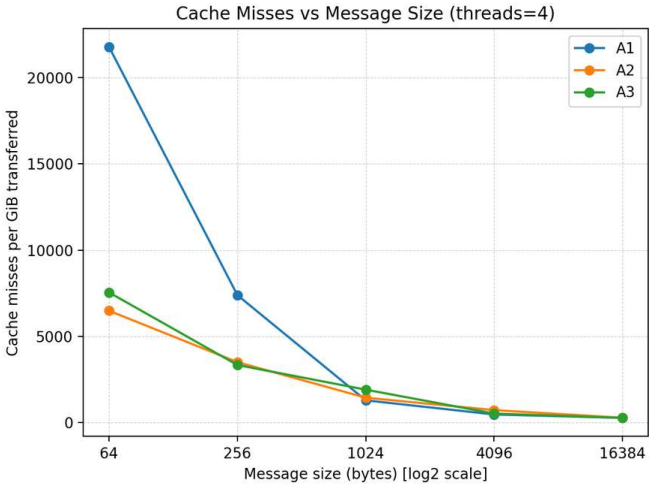
This appendix includes selected screenshots captured during execution of part D.



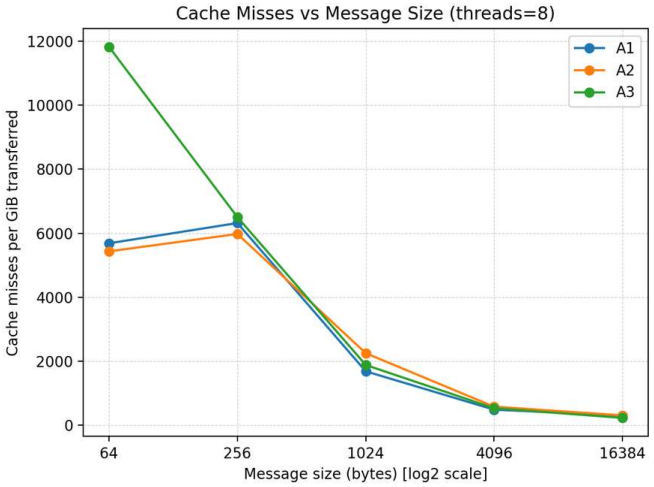
cache_misses_per_gb_t1



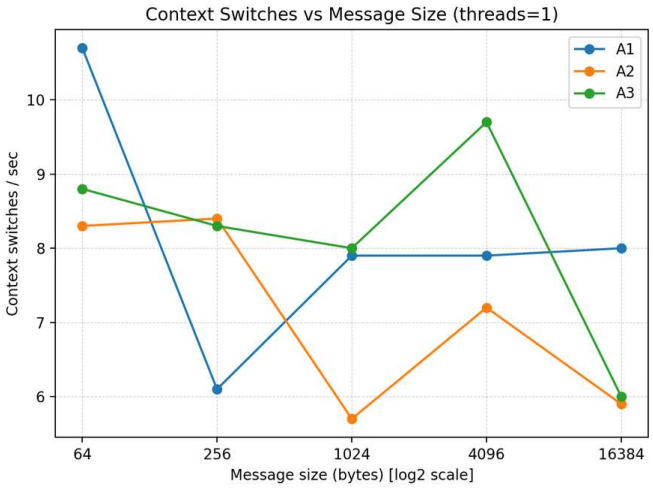
cache_misses_per_gb_t2



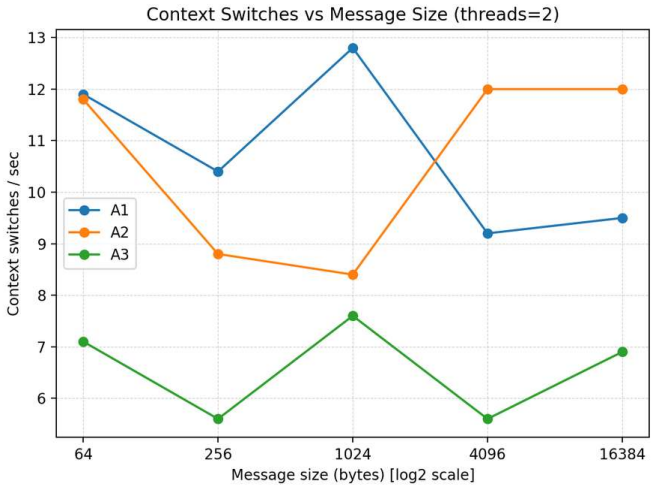
cache_misses_per_gb_t4



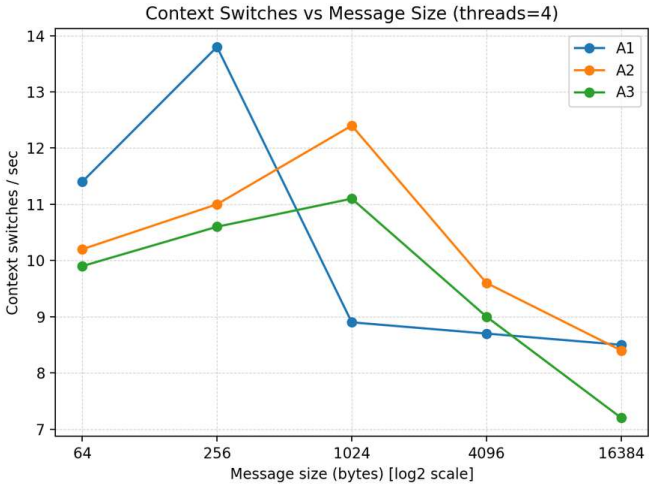
cache_misses_per_gb_t8



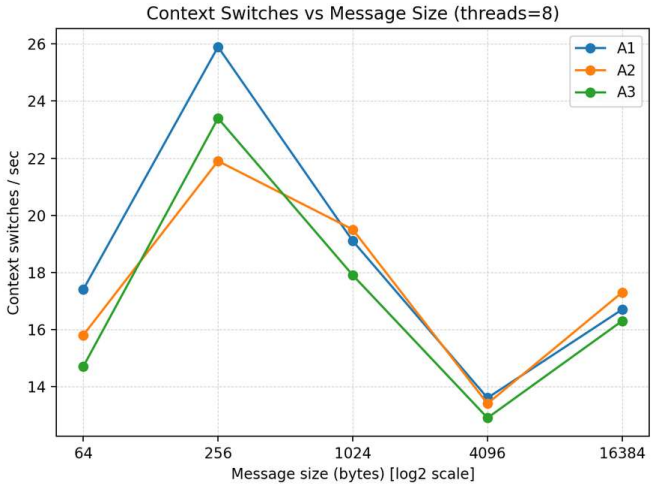
ctx_switches_per_sec_t1



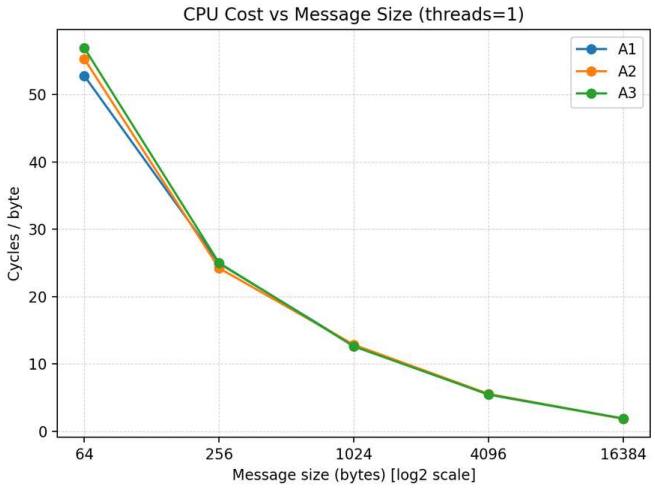
ctx_switches_per_sec_t2



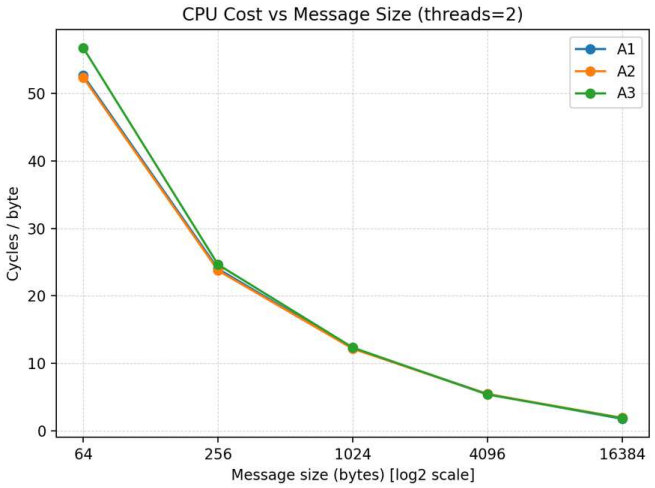
ctx_switches_per_sec_t4



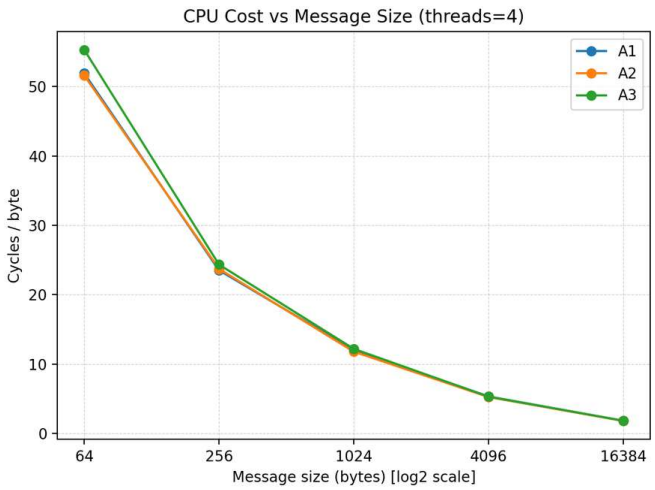
ctx_switches_per_sec_t8



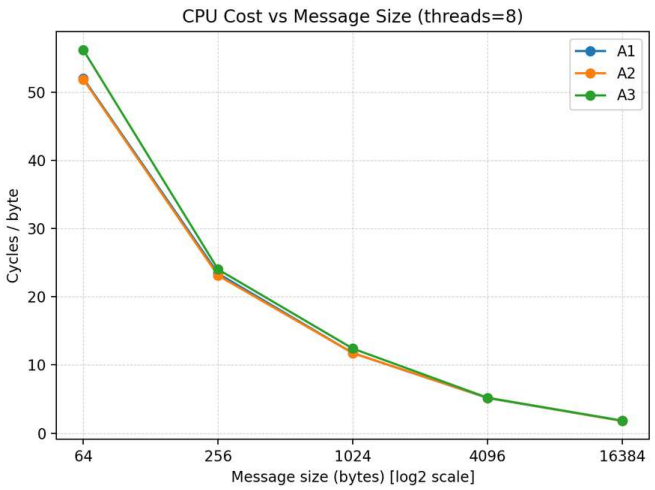
cycles_per_byte_t1



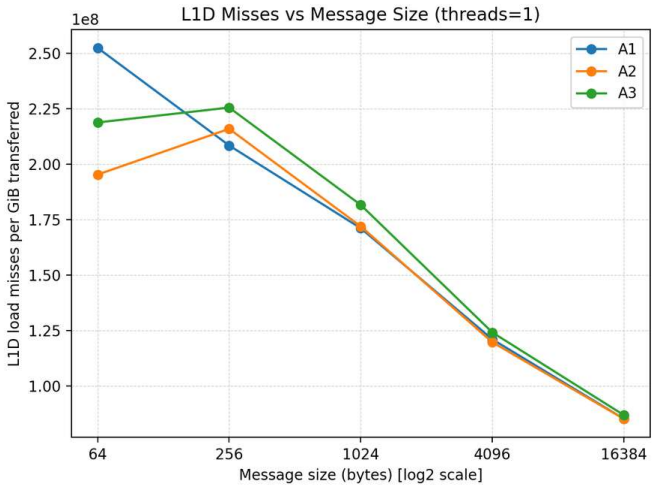
cycles_per_byte_t2



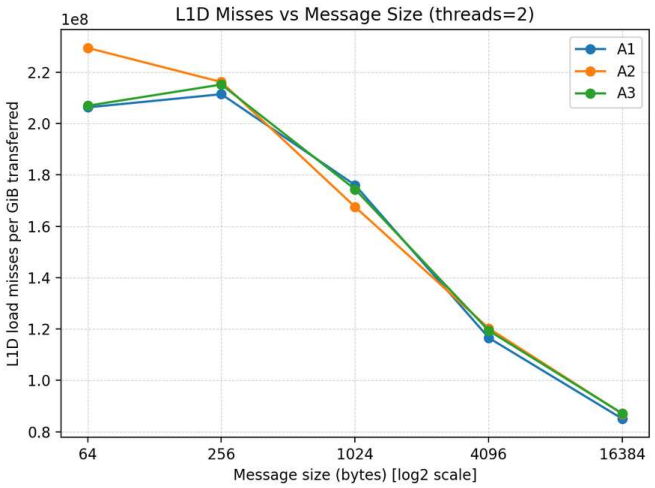
cycles_per_byte_t4



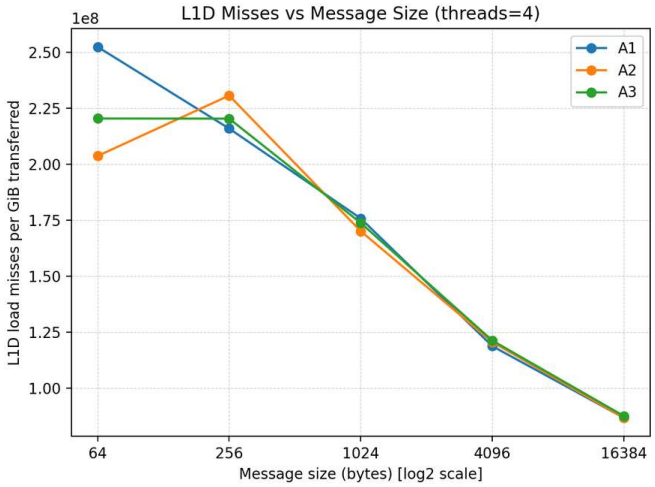
cycles_per_byte_t8



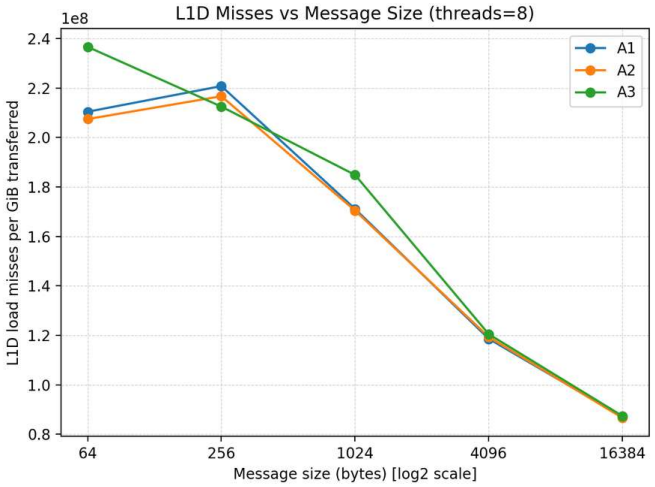
l1_misses_per_gb_t1



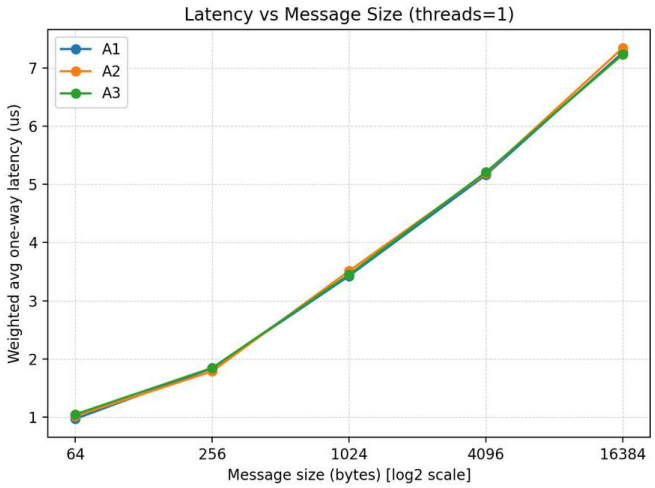
l1_misses_per_gb_t2



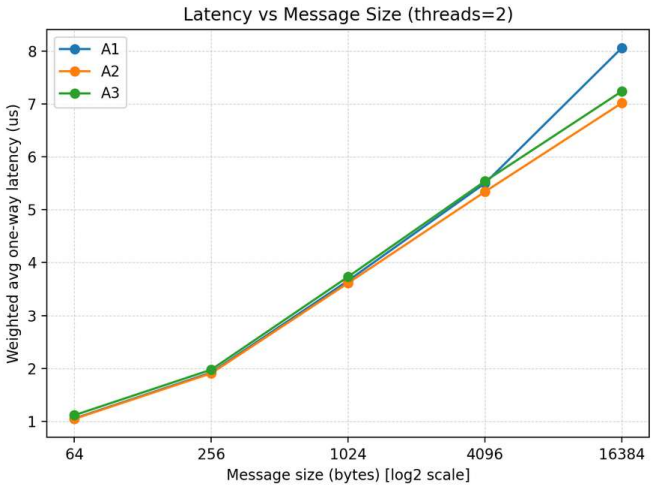
l1_misses_per_gb_t4



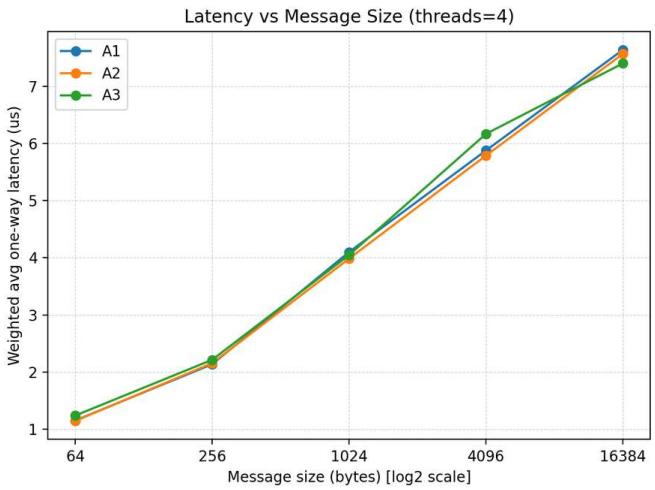
l1_misses_per_gb_t8



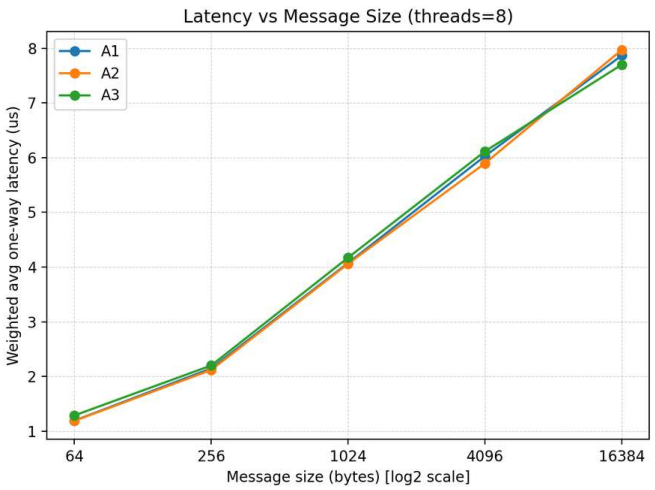
latency_us_t1



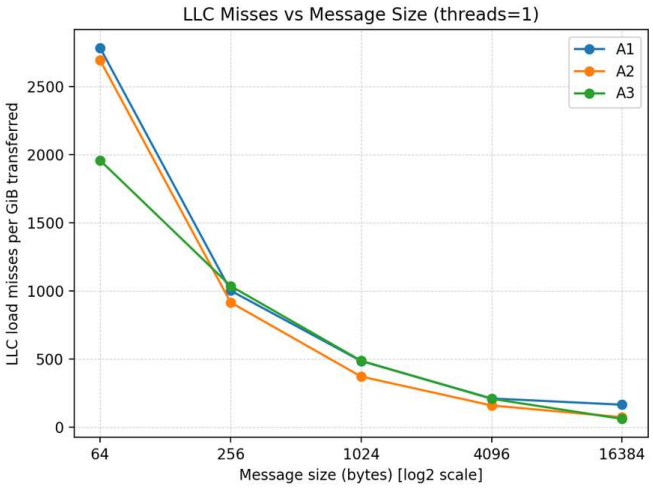
latency_us_t2



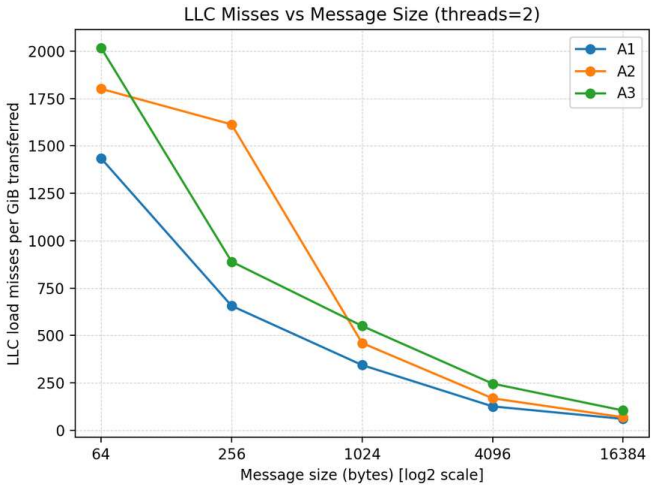
latency_us_t4



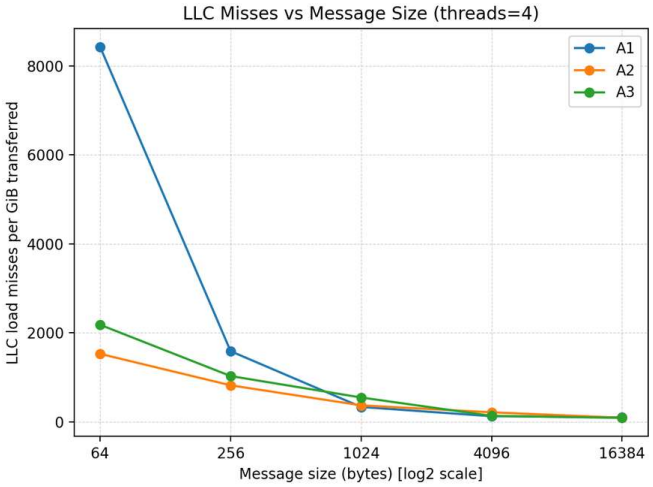
latency_us_t8



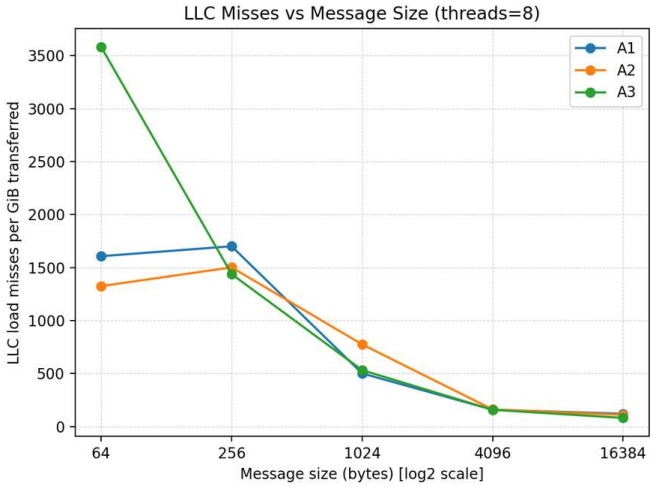
llc_misses_per_gb_t1



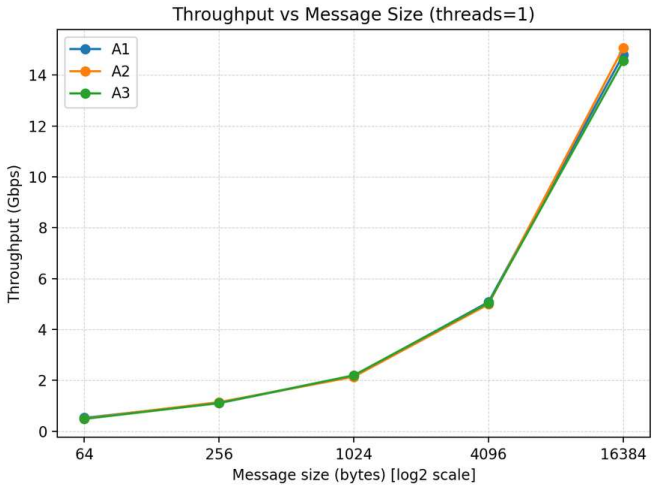
llc_misses_per_gb_t2



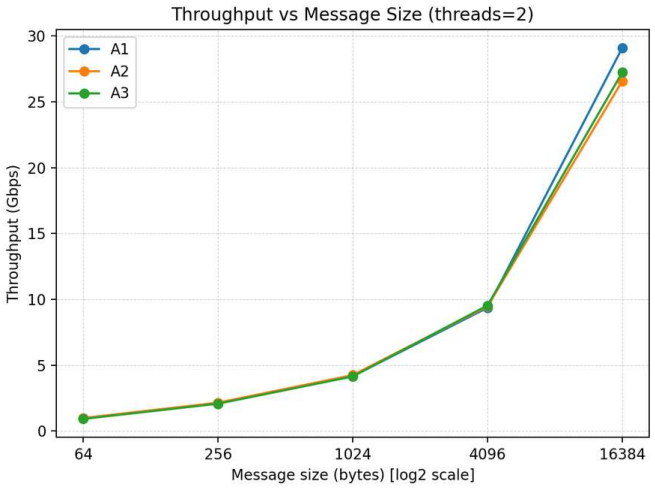
llc_misses_per_gb_t4



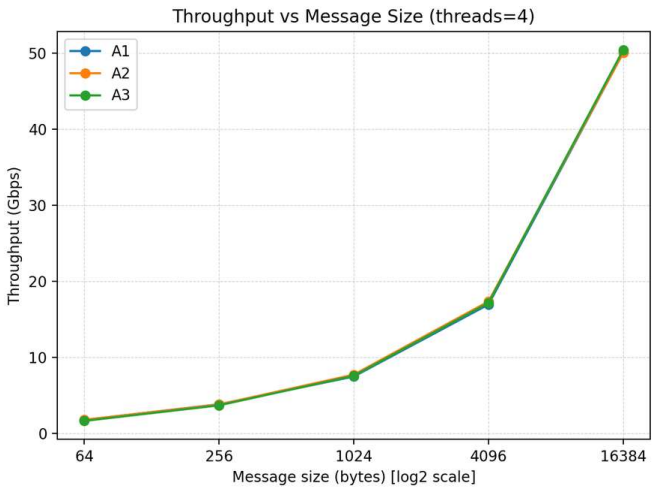
llc_misses_per_gb_t8



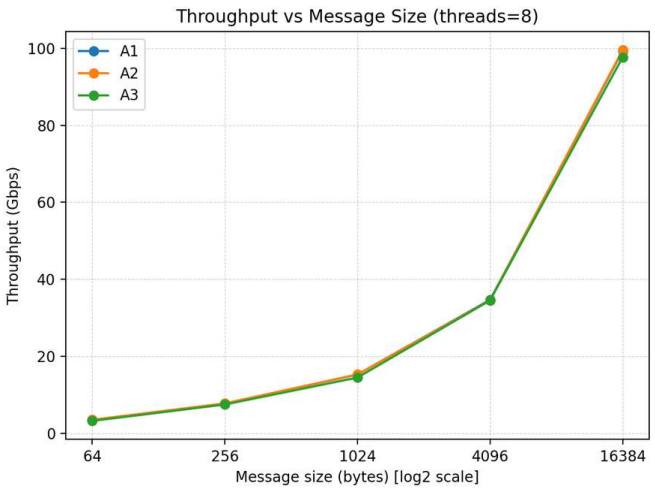
throughput_gbps_t1



throughput_gbps_t2



throughput_gbps_t4



throughput_gbps_t8