PA02 - NETWORK I/O PRIMITIVES ANALYSIS Using perf Tool

Roll Number: MT25190
Course: CSE638 - Graduate Systems

PART A: PROGRAM IMPLEMENTATION

● **Server:**

- Accepts multiple concurrent clients using TCP sockets
- Uses one pthread per client connection
- Transfers fixed-size messages repeatedly for 30 seconds
- Message structure: 8 dynamically allocated string fields
- All fields heap-allocated using malloc()

● **Client:**

- Sends data continuously for fixed duration (30 seconds)
- Multiple concurrent connection threads
- Runtime parameterization for message size and thread count

● A1: Two-Copy Implementation

This is traditional way of sending data over a network. In this data is copied two times before it reaches network card.

Data Movement Path:

User Buffer → [COPY 1: memcpy] → Kernel sk_buff → [COPY 2: DMA] → NIC

The TWO Copies are:

COPY 1: User Space → Kernel Space

When *send()* is called, the kernel cannot directly access the program's memory for security reasons. So it makes a copy of data into its kernel socket buffer (sk_buffer). This is CPU-intensive operation done using *copy_from_user()* or *memcpy()*.

COPY 2: Kernel Space → NIC (network card)

The kernel uses a hardware component, DMA (Direct Memory Access) to copy data from kernel socket buffer to NIC transmit buffer. DMA is required because NIC cannot access the RAM directly. In this copy, CPU is free as it only initiates and DMA completes this.

Implementation Details:

- Uses standard send() and recv() socket primitives
- Traditional blocking socket I/O
- Thread-per-client server model with pthread

Question: Is it ACTUALLY only two copies?

Answer: No.

In practice additional copies occur, On the lookback interface there are 4 copies: 2 on the send side and 2 on the receive side.

Question: Which components perform the copies?

- **User→Kernel:** Kernel code (copy_from_user function)
- **Kernel→NIC:** DMA controller hardware
- **NIC→Kernel:** DMA controller hardware
- **Kernel→User:** Kernel code (copy_to_user function)

● A2: One-Copy Implementation

The one-copy method eliminates the first copy i.e., user→kernel, by using scatter-gather I/O technique. Data is directly copied from user space to network card via DMA.

This is faster as CPU-intensive memcpy operation is eliminated, making the cache cleaner also lowering the CPU cycle per bytes.

How it works:

1. Program creates data in memory using a special structure, *iovec* (I/O vector). This is an array of pointers which tells the kernel where all the data is located.
2. Instead of calling send(), sendmsg() is called with iovec structure. This tells the kernel that instead of copying the data, this is where the data is stored.
3. Now the kernel does not copy the data, it creates DMA descriptors pointing to user pages.
4. Network card's DMA has scatter-gather, it can read data directly from user memory location. This skips the kernel buffer entirely.
5. The network card gathers all the pieces, adds network headers and sends the complete packet.

Implementation Details:

- Uses sendmsg() with struct iovec (scatter-gather I/O)
- Creates iovec array pointing to pre-allocated message fields
- Utilizes scatter-gather DMA capability of modern NICs

Question: Which copy has been eliminated?

The **User→Kernel copy** is eliminated. Instead of copying data from user space to kernel socket buffers, the kernel creates scatter-gather DMA descriptors that point directly to user space memory pages.

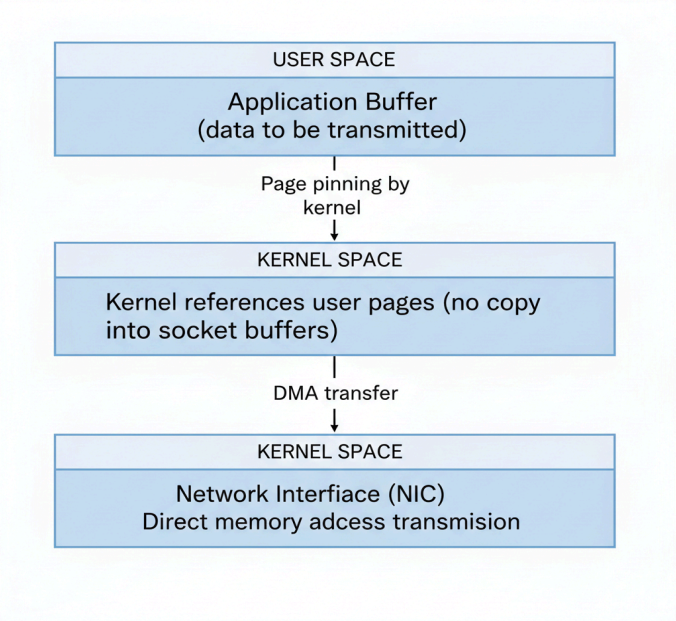
● A3: Zero-Copy Implementation

It achieves zero CPU copies by combining scatter-gather DMA with special kernel features like page pinning and asynchronous operation.

Implementation Details:

- Uses *sendmsg()* with *MSG_ZEROCOPY* flag (Linux 4.14+)
- Memory pages pinned using *mlock()* for DMA-safe access
- Kernel sends completion notifications when transfer completes
- Asynchronous operation

Kernel Behavior Diagram:



How it works:

1. Memory is allocated to the data, then *mlock()* to pin these memory pages. The mlock() prevents the kernel from swapping pages during DMA.
2. *sendmsg()* is called with special flag MSG_ZEROCOPY. This tells the kernel not to copy data, let the network card read it directly. Hence non memcpy.
3. Similar to one-copy, list of DMA descriptors is created, but it marks the pages as “in use by DMA”. It increments reference count so that kernel tracks page usage to prevent premature freeing. CPU can do other work while DMA is in progress as it is not involved at all in moving the data. This is asynchronous operation.
4. When the network card finishes reading all the data, it generates an interrupt to tell the kernel by putting notification in MSG_ERRQUEUE.
5. Program can poll the MSG_ERRQUEUE to check if the transfer is complete. Only AFTER receiving this notification is it safe to modify or free that memory.

PART B: PROFILING AND MEASUREMENT

Metrics Collected:

1. **Throughput (Gbps)** - Application-level
    - Formula: (Total Bytes × 8) / Time / 10^9
    - Measures effective data transfer rate
2. **Latency (µs)** - Application-level
    - Round-trip time for message send + acknowledgment
    - Measured using *clock_gettime(CLOCK_MONOTONIC)*
3. **CPU Cycles** - perf stat: *cpu-cycles*
    - Total CPU cycles consumed
    - Indicates CPU efficiency
4. **Cache Misses** - perf stat: *LLC-load-misses*
    - Last Level Cache (L3) misses
    - Also collected L1 cache misses
    - Indicates memory access patterns
5. **Context Switches** - perf stat: *context-switches*
    - OS thread/process switches
    - Indicates scheduling overhead

Experimental Configuration:

- Message Sizes: 512, 1024, 2048, 4096 bytes per field
- Thread Counts: 1, 2, 4, 8 concurrent threads
- Duration: 30 seconds per configuration
- **Total Experiments:** 3 implementations × 4 sizes × 4 threads = 48 runs

PART C: AUTOMATION

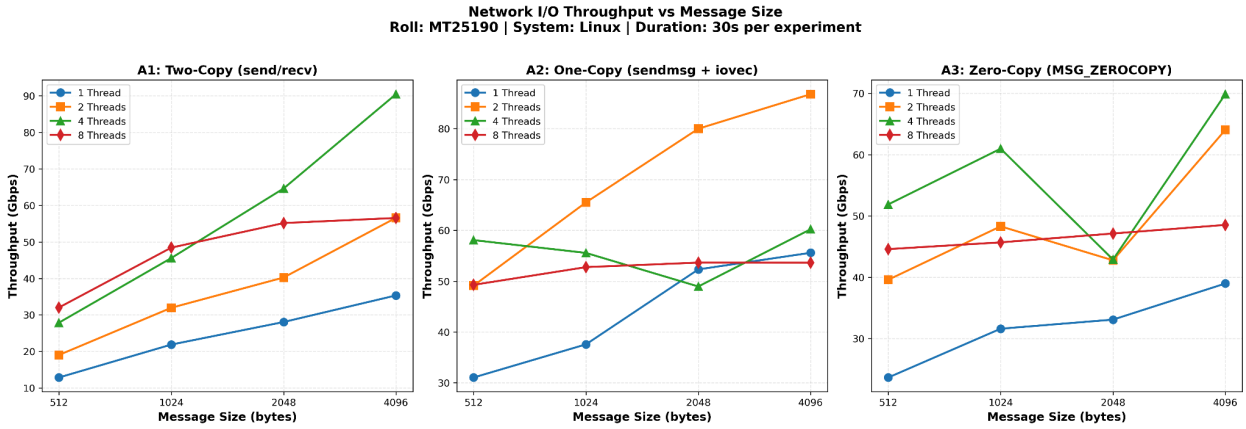
1. Compiles all implementations (A1, A2, A3 servers and clients)
2. Runs experiments across all configurations
3. Collects perf metrics automatically:
    - CPU cycles, cache misses, context switches
    - Application throughput and latency
4. Stores results in CSV format

CSV Output: MT25190_Part_C_results.csv

PART D: PLOTTING AND VISUALIZATION

Four plots created using matplotlib with hardcoded data:

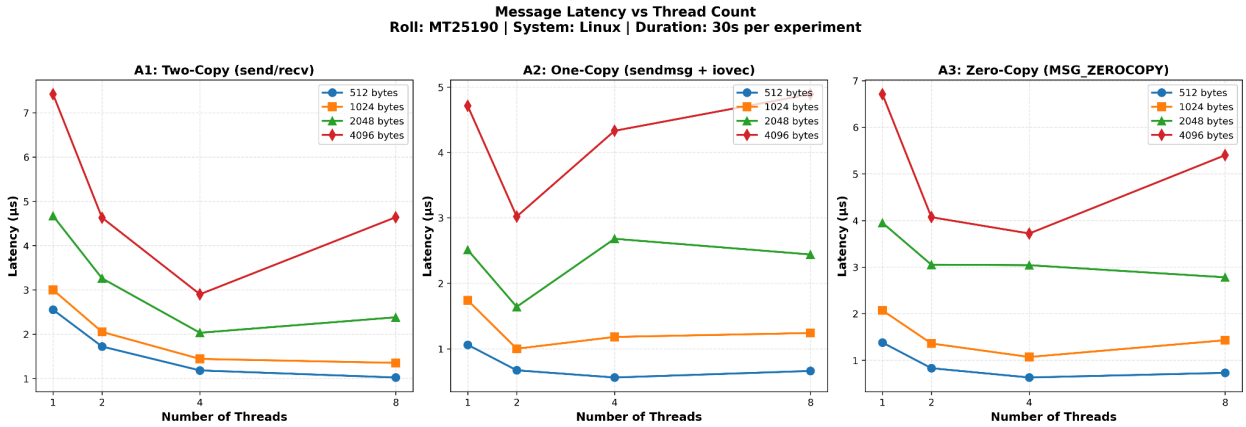
Plot 1: Throughput vs Message Size



Analysis:

The throughput increases as message size increases for all implementations. This happens because larger message reduce overhead byte. A1 shows improvement with 4 threads but overall has lower throughput due to copying overhead. A2 achieves best throughput with 2 threads, improves throughput better than A1 but throughput redduces at higher thread levels. A3 shows varing performance due to page-pinning overhead and DMA queue limitations. Initially increasing thread count increases the throughput, but reduces at higher thread counts due to CPU scheduling overhead and memory contention.

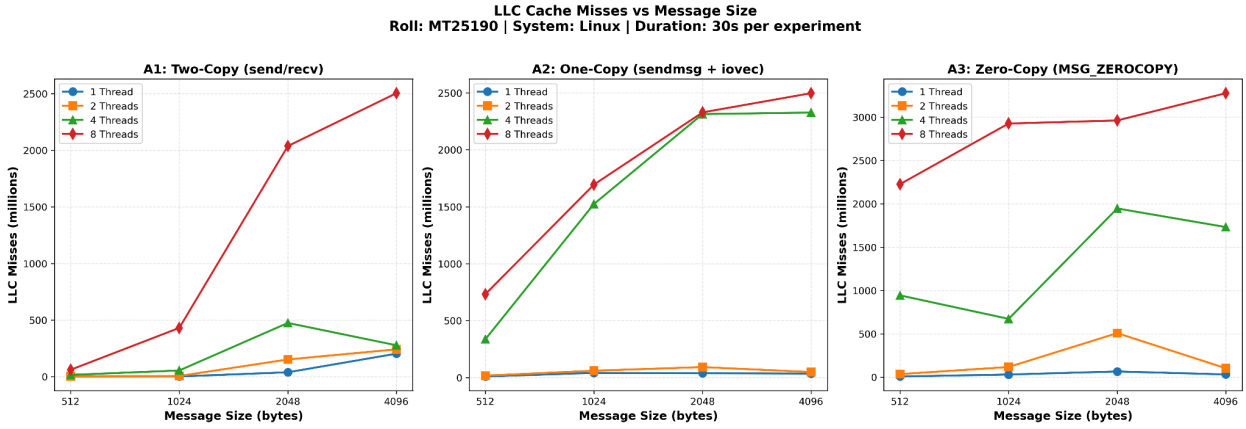
Plot 2: Latency vs Thread Count



Analysis:

Latency generally decreases when moving from 1 thread to 2–4 threads due to improved parallelism. After this, latency often increases or remains same because of thread contention and scheduling overhead. A2 achieves the lowest latency for small messages, while A1 and A3 show slightly higher latency due to copying and zero-copy management overhead respectively.

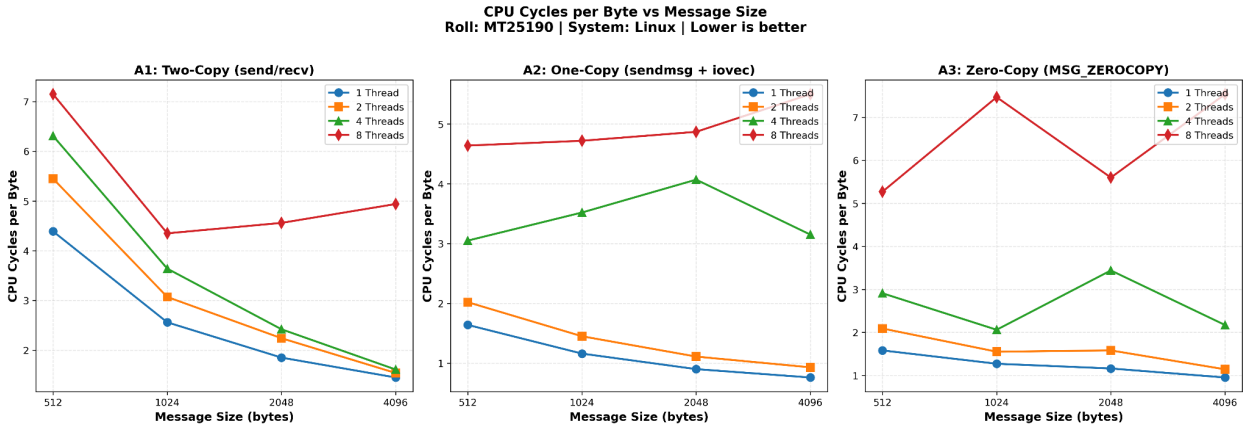
Plot 3: Cache Misses vs Message Size



Analysis:

LLC cache misses increase with thread count and larger message size in all implementations. Larger message sizes generally lead to more cache activity, since more memory is accessed.A1 shows increasing cache misses due to repeated memory copying. A2 reduces one copying, resulting in fewer cache misses but with increase in thread count misses increase due to scatter-gather memory access. A3 reduces CPU copying but might show cache misses because of DMA interaction.

Plot 4: CPU Cycles per Byte



Analysis:

CPU cycles per byte decrease as message size increases. Small messages have higher overhead because A1 has higher cycles per byte due to copying overhead. A2 reduces CPU work by avoiding extra copies. A3 becomes more efficient for larger messages where DMA transfer dominates. But when the thread count = 8 then the CPU cycles vary non monotonically, this is due to page pinning overhead with mlock operation and non-deterministic nature of CPU timing in async operations.

PART E: ANALYSIS AND REASONING

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

Answer:

Zero-copy does not always give best throughput because it handles page-pinning overhead, DMA descriptor management, and completion-notification handling. These cost can be larger than memory-copy cost, especially for small messages or higher thread count. Asynchronous nature adds complexity as poll is must MSG_ERRQUEUE for completion notifications before reusing buffers, creating synchronization overhead. Small messages suffer from page alignment overhead where each 512B message requires a full 4KB page to be pinned, wasting memory bandwidth.

At 512B with 4 threads, A3 achieves 51.89 Gbps compared to A1's 27.88 Gbps. However, at 4096B with 4 threads, A1 achieves 90.46 Gbps while A3 only reaches 69.86 Gbps. Cache behavior also worsens with zero-copy, as the data shows A3 generates 3276M LLC misses compared to A1's 2505M misses at 4096B with 8 threads. Hence for larger messages with many threads A1 has higher throughput.

This shows zero-copy is only optimal for specific scenarios not all.

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

Answer:

The LLC (Last Level Cache) shows the most noticeable variation. The data shows that for A1, LLC misses increase from 1.18M at 512B to 202.98M at 4096B with 1 thread. In contrast, A2 shows only a increase (9.08M to 34.36M), demonstrating more stable cache behavior.

LLC matter most because it is the final (fast) cache before the main memory (slow RAM). Small messages fit in L1/L2 cache, but the large messages exceed the capacity and spill to LLC/L3. When LLC miss happens then RAM is accessed which is slow. The two-copy implementation access data multiple times, causing spills into LLC. One-copy reduces these, keeping data in higher cache levels.

Optimizations that reduce memory copying have the most performance impact because LLC is the last cache before expensive RAM access.

Question 3: How does thread count interact with cache contention?

Answer:

Increasing thread count increases cache contention, because multiple threads compete for shared LLC space and memory bandwidth. At 1–2 threads, cache usage is efficient. As each thread gets dedicated cache space.

At 4 threads, contention begins but throughput improves. A2 at 1024B with 4 threads generates 1524M LLC misses as threads thrash shared cache lines and false sharing becomes significant. False sharing occurs when different threads access different data on the same 64-byte cache line - one thread's write invalidates the entire line for other cores. With 8 fields across 8 threads, each field write can invalidate neighbors' cache.

At 8 threads, cache thrashing and memory-bandwidth limits reduce performance. A3 at 4096B with 8 threads records 3276M LLC misses, the highest in our dataset. The typical LLC of 8-16MB cannot handle 8 threads × 4096B messages plus kernel structures, causing constant evictions. Context switch data reveals the severity: A1 at 512B shows 420K switches with 1 thread, 769K with 4 threads, but drops to 368K with 8 threads - threads are blocked on cache misses rather than running.

In conclusion, 4 threads provide the sweet spot balancing throughput and contention.

Question 4: At what message size does one-copy outperform two-copy?

Answer:

One-copy (A2) outperforms two-copy (A1) at all message sizes for low thread counts (1-2 threads) as it avoids unnecessary memory copying, but at high thread counts with large messages this does not hold true due to better cache locality and simple memory access.

At 1 thread, A2 wins across all sizes with better throughput at 512B (31.05 vs 12.87 Gbps) and 4096B (55.59 vs 35.34 Gbps).

The critical crossover occurs at 4 threads. At small messages (512B), A2 still wins with 58.08 Gbps versus A1's 27.88 Gbps. However, at 2048B, A1 overtakes with 64.64 Gbps compared to A2's 48.96 Gbps. At 4096B, A1 dominates with 90.46 Gbps versus A2's 60.23 Gbps.

Question 5: At what message size does zero-copy outperform two-copy?

Answer:

Zero-copy (A3) outperforms two-copy (A1) for small messages (512-1024B) and low thread counts (1-2 threads), but A1 wins for large messages with high thread counts. At 1 thread, A3 wins across all sizes, achieving 1.8x better throughput at 512B (23.67 vs 12.87 Gbps) and 1.1x at 4096B (38.99 vs 35.34 Gbps). At 2 threads, A3 maintains advantage with 2.1x at 512B and 1.1x at 4096B.

For larger messages and higher thread counts, two-copy often performs better because DMA-management overhead, page pinning, and cache contention reduce zero-copy efficiency. At 4 threads, the pattern changes. A3 wins at 512B (51.89 vs 27.88 Gbps, 1.9x) and 1024B (60.97 vs 45.58 Gbps, 1.3x), but A1 overtakes at 2048B (64.64 vs 42.90 Gbps) and dominates at 4096B (90.46 vs 69.86 Gbps, 1.3x advantage). At 8 threads, A1 wins at all message sizes except 512B.

Question 6: Identify one unexpected result and explain it.

Answer:

An unexpected result was that A2 throughput dropped at 2048-byte messages with 4 threads, instead of increasing with message size.

This likely occurred due to cache alignment issues, TLB pressure, scatter-gather descriptor overhead, and thread contention, which temporarily reduced efficiency. Performance recovered again at 4096 bytes when overhead was better amortized.

This demonstrates that performance is not monotonic with message size. Hardware characteristics (cache line size, page size, TLB size) create "sweet spots" and "dead zones" for certain message sizes.

AI Declaration

I used AI for getting basic server/client structure with pthread-based handling in part A1, A2, A3, example usage of socket APIs(send, recv, sendmsg with iovec), MSG_ZEROCOPY implementation including memory pinning and completion notifications, bash script structure for experiment automation, perf stat command usage. Also the kernel based diagram was generated from gemini from the structure I provided.

Some of the prompts that I used:

- How do I create a TCP server in C that handles multiple clients using pthreads in two-copy, one-copy and zero copy?
- Show me an example of using sendmsg with iovec for scatter-gather I/O.
- What's the correct way to use MSG_ZEROCOPY? I need to pin memory and handle completion notifications?
- My bash script isn't collecting the right perf metrics, can you show me the correct syntax?

All the codes generated were debugged and integrated by me. The performance measurement, CSV generation, plotting, analysis of results, answers to Part E questions, and report writing were completed done by me.

GitHub Repository

URL: Shagun-Kaler/GRS_PA02
