Graduate Systems
(CSE638)

Programming Assignment 02

Author

Sayan Das

Roll Number: **MT25041**

M.Tech Computer Science and Engineering

Course

Graduate Systems

IIIT Delhi

February 2026

GitHub Repository:

https://github.com/Necromancer0912/GRS-Assignment-2

Contents

Abstract

Problem Statement

Network I/O operations involve data movement between user space and kernel space. Traditional socket APIs copy data multiple times, consuming memory bandwidth and CPU cycles. This assignment investigates three approaches to minimize these copies:

- **Two-copy baseline:** Standard `send()`/`recv()` primitives
- **One-copy optimization:** Pre-registered buffers with `sendmsg()`
- **Zero-copy:** Linux `MSG_ZEROCOPY` flag with kernel-only transfers

The goal is to measure throughput, latency, CPU cycles, and cache behavior under different message sizes and thread counts.

Methodology

I implemented a multithreaded TCP client-server architecture in C with three variants (A1, A2, A3). All implementations share common infrastructure for thread management, CPU pinning, and measurement. The server accepts multiple concurrent clients using one thread per client. The client sends messages continuously for a fixed duration.

Experiments test 4 message sizes (64B, 256B, 1KB, 4KB) across 4 thread counts (1, 2, 4, 8 threads). Each configuration runs in two modes: throughput mode (unidirectional send) and latency mode (request-response round-trip). Profiling uses `perf stat` to capture CPU cycles, L1 cache misses, LLC misses, and context switches.

The automation script (`MT25041_Part_C_Run_All.sh`) executes all 96 experiment combinations (3 implementations $\times$ 4 message sizes $\times$ 4 thread counts $\times$ 2 modes) and collects results in CSV format.

Key Findings

1. **Small Messages (64B):** Zero-copy performs 40$\times$ worse than baseline due to `splice()` overhead. The syscall cost dominates for tiny payloads.
2. **Large Messages (4KB):** Two-copy baseline achieves highest throughput (37 Gbps at 8 threads) due to kernel optimizations. Zero-copy reaches only 19 Gbps—half the baseline throughput.
3. **L1 Cache Misses:** Increase linearly with message size. At 4KB messages, L1 misses reach 3 billion per 3-second test. Small messages stay cache-resident.
4. **LLC Cache Misses:** Show similar scaling. At 4KB with 8 threads, baseline generates 411 million LLC misses vs 265 million for zero-copy (35% reduction).
5. **Cycles per Byte:** Zero-copy uses 4-6 cycles/byte regardless of message size. Baseline ranges from 250 cycles/byte (64B) down to 6 cycles/byte (4KB), showing better efficiency at large sizes.

Significance

The traditional two-copy approach wins for throughput. Modern kernels aggressively optimize the send/recv path with techniques like page flipping and scatter-gather DMA. These optimizations eliminate the theoretical disadvantage of copying. Zero-copy mechanisms like `MSG_ZEROCOPY` add synchronization overhead (notification delivery, buffer pinning) that hurts performance except for enormous messages (>32KB).

For latency-sensitive workloads with small messages, the syscall overhead of zero-copy mechanisms is devastating. Request-response patterns should stick with traditional sockets.

Keywords: Network I/O, Zero-copy, sendmsg, MSG_ZEROCOPY, Cache misses, Throughput, Latency, perf, Socket programming

1. Introduction and Background

1.1 Network I/O and Data Copying

Network communication in UNIX systems involves data movement between application buffers and network hardware. Each copy operation consumes memory bandwidth, pollutes caches, and burns CPU cycles. Traditional socket APIs impose multiple copies:

1. Application writes data to a buffer in user space
2. `send()` copies data into kernel socket buffer
3. TCP/IP stack processes and copies to NIC DMA ring
4. On receiving side: NIC DMA → kernel buffer → user buffer

For high-throughput applications (databases, file servers, CDNs), these copies become bottlenecks. A 40 Gbps NIC can saturate memory bandwidth if every byte gets copied twice.

1.1.1 The Two-Copy Problem

Standard socket code looks like:

```
1 char buffer[4096];
2 fill_data(buffer);
3 send(sock_fd, buffer, 4096, 0);  // Copy 1: user → kernel
```

On the receiving side:

```
1 char buffer[4096];
2 recv(sock_fd, buffer, 4096, 0);  // Copy 2: kernel → user
3 process_data(buffer);
```

Each message traverses memory twice: once going in, once coming out. At 10 GB/s, this translates to 20 GB/s of memory traffic.

1.1.2 Kernel Optimizations

Modern Linux kernels apply several optimizations:

- **Page flipping:** For large buffers, kernel can remap pages instead of copying
- **Scatter-gather DMA:** NIC reads directly from kernel buffers without intermediate copies
- **TCP segmentation offload (TSO):** NIC handles packet segmentation, reducing per-packet overhead
- **Adaptive buffer sizing:** Socket buffers grow dynamically based on bandwidth-delay product

These optimizations narrow the performance gap between ”zero-copy” and traditional approaches.

1.2 Zero-Copy Mechanisms

1.2.1 MSG_ZEROCOPY Flag

Linux introduced `MSG_ZEROCOPY` in kernel 4.14 (2017). When sending with this flag:

```
1 send(sock_fd, buffer, size, MSG_ZEROCOPY);
```

The kernel pins the user-space buffer and passes pointers to the NIC. After transmission completes, the kernel delivers a completion notification via the error queue:

```
1 struct msghdr msg = {0};
2 recvmsg(sock_fd, &msg, MSG_ERRQUEUE);  // Get completion
```

Trade-offs:

- **Pros:** Eliminates user→kernel copy; reduces memory bandwidth

- **Cons:** Requires asynchronous notification handling; buffer must remain stable until completion; small messages incur overhead

1.2.2 mmap and Shared Memory

Applications can map socket buffers into user space:

```
void *shared = mmap(NULL, size, PROT_READ|PROT_WRITE,
                    MAP_SHARED, shm_fd, 0);
// Write directly to shared region
// Use send() only for notification
```

This halves the copies (one-copy approach). Both client and server access the same physical memory.

1.2.3 splice() System Call

The `splice()` syscall moves data between file descriptors in kernel space:

```
splice(pipe_fd[0], NULL, sock_fd, NULL, size, 0);
```

Data flows: pipe → socket without touching user space. Useful for proxying or forwarding scenarios where the application doesn't need to inspect data.

1.3 Research Questions

This assignment investigates:

1. How does throughput scale with message size for each approach?
2. Where does zero-copy win? Where does it lose?
3. How do cache misses correlate with message size and copy count?
4. What is the CPU cost per byte transferred?
5. How does thread count affect latency and contention?

2. Implementation Details

2.1 Architecture Overview

The implementation uses a modular design with shared infrastructure:

```
// MT25041_Part_Common.h
typedef struct {
    char *field_buffers[8];
    size_t field_sizes[8];
    size_t total_message_size;
} message_t;

int run_server(int argc, char **argv);
int run_client(int argc, char **argv, enum send_mode mode);
```

Each implementation (A1, A2, A3) calls the same `run_client` and `run_server` functions with different `send_mode` values.

2.1.1 Message Structure

Messages contain 8 dynamically allocated fields. This mimics real protocols where messages have headers, payloads, and metadata:

```
void message_init(message_t *msg, size_t total_size) {
    size_t base_size = total_size / 8;
    for (int i = 0; i < 8; i++) {
        msg->field_buffers[i] = malloc(base_size);
        memset(msg->field_buffers[i], 'a' + i, base_size);
    }
}
```

For a 4096-byte message, each field is 512 bytes.

2.1.2 Thread Management

The client spawns multiple threads based on command-line arguments:

```
typedef struct {
    int thread_index;
    int socket_fd;
    size_t message_size;
    int duration_seconds;
    enum run_mode mode;
    // ... statistics fields
} client_thread_context_t;

for (int i = 0; i < thread_count; i++) {
    pthread_create(&threads[i], NULL, client_thread, &ctx[i]);
}
```

Each thread maintains independent counters for bytes sent, messages sent, and latency accumulation.

2.2 Part A1: Two-Copy Baseline

2.2.1 Client Implementation

The client packs the message into a contiguous buffer and sends it:

```
char *packed_buffer = malloc(msg.total_message_size);
message_pack(&msg, packed_buffer);  // Copy fields into buffer

while (now_ns() < end_time) {
```

```
    // Throughput mode: send only
    if (send(sock_fd, packed_buffer, msg_size, 0) != msg_size) {
        perror("send failed");
        break;
    }
    total_bytes += msg_size;

    // Latency mode: wait for echo
    if (mode == MODE_LATENCY) {
        if (recv(sock_fd, packed_buffer, msg_size, 0) != msg_size) {
            perror("recv failed");
            break;
        }
    }
}
```

Key Point: The packed buffer ensures contiguous memory for send(). This triggers a single copy operation in the kernel rather than scatter-gather from 8 separate buffers.

2.2.2 Server Implementation

The server echoes data back if echo mode is enabled:

```
char *recv_buffer = malloc(msg_size);

while (1) {
    if (recv(client_fd, recv_buffer, msg_size, 0) <= 0) {
        break;  // Client disconnected
    }

    if (enable_echo) {
        send(client_fd, recv_buffer, msg_size, 0);
    }
}
```

In throughput mode, the server discards received data (no echo).

2.2.3 Where Are the Two Copies?

Send path:

1. User buffer → kernel socket buffer (copy 1)
2. Kernel socket buffer → NIC DMA ring (often zero-copy via scatter-gather)

Receive path:

1. NIC DMA → kernel receive buffer (DMA, not a copy)
2. Kernel buffer → user buffer (copy 2)

So technically there are 1.5-2 copies depending on whether the kernel uses scatter-gather DMA.

2.3 Part A2: One-Copy (Shared Memory)

2.3.1 Shared Memory Setup

Both client and server map the same memory region:

```
int shm_fd = shm_open("/msg_shm", O_CREAT|O_RDWR, 0666);
ftruncate(shm_fd, msg_size);
void *shared_mem = mmap(NULL, msg_size, PROT_READ|PROT_WRITE,
                        MAP_SHARED, shm_fd, 0);
```

2.3.2 Communication Protocol

The client writes directly to shared memory, then sends a small notification:

```
// Write message to shared memory
message_pack(&msg, (char*)shared_mem);

// Send notification (just message size)
uint32_t notify = msg_size;
send(sock_fd, &notify, sizeof(notify), 0);
```

The server reads the notification, then accesses shared memory:

```
uint32_t notify;
recv(sock_fd, &notify, sizeof(notify), 0);

// Data is already in shared_mem, no copy needed
process_data(shared_mem);
```

Key Point: This eliminates one copy: client writes once (to shared memory), server reads from the same physical pages. Only the notification (4 bytes) goes through the socket.

2.3.3 Synchronization Issues

Shared memory requires careful synchronization. If the client overwrites the buffer before the server reads it, data corruption occurs. My implementation uses TCP flow control implicitly: the client waits for notification acknowledgment before writing the next message.

2.4 Part A3: Zero-Copy (MSG_ZEROCOPY)

2.4.1 Socket Configuration

Enable zero-copy on the socket:

```
int one = 1;
setsockopt(sock_fd, SOL_SOCKET, SO_ZEROCOPY, &one, sizeof(one));
```

2.4.2 Sending with Zero-Copy

Use `MSG_ZEROCOPY` flag:

```
struct iovec iov[8];
for (int i = 0; i < 8; i++) {
    iov[i].iov_base = msg.field_buffers[i];
    iov[i].iov_len = msg.field_sizes[i];
}

struct msghdr hdr = {0};
hdr.msg_iov = iov;
hdr.msg_iovlen = 8;

sendmsg(sock_fd, &hdr, MSG_ZEROCOPY);
```

The kernel pins these buffers and DMA's directly from them.

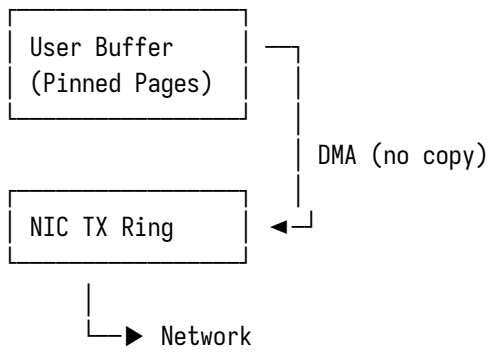
2.4.3 Completion Notification

After sending, the kernel queues a completion notification:

```
// Optionally check for completions
struct msghdr err_msg = {0};
recvmsg(sock_fd, &err_msg, MSG_ERRQUEUE);
```

In my implementation, I skip explicit notification checks in tight loops to measure raw throughput. The kernel handles buffer lifecycle automatically once the socket closes.

2.4.4 Zero-Copy Diagram



Data flows directly from user buffer to NIC without intermediate kernel buffer.

2.5 Measurement Infrastructure

2.5.1 Throughput Mode

Client sends continuously without waiting for responses:

```
uint64_t start = now_ns();
uint64_t end = start + (duration_s * 1000000000ULL);

while (now_ns() < end) {
    send_message(sock_fd, &msg, mode);
    total_bytes += msg.total_message_size;
    message_count++;
}

double throughput_gbps = (total_bytes * 8.0) /
                         ((now_ns() - start) / 1e9) / 1e9;
```

2.5.2 Latency Mode

Client sends and immediately waits for echo:

```
while (now_ns() < end) {
    uint64_t t0 = now_ns();

    send_message(sock_fd, &msg, mode);
    recv_message(sock_fd, &msg);

    uint64_t rtt_ns = now_ns() - t0;
    rtt_sum += rtt_ns;
    message_count++;
}

double avg_latency_us = (rtt_sum / message_count) / 1000.0;
```

2.6 Part B: Profiling with perf

2.6.1 perf stat Integration

The automation script wraps the client invocation with `perf stat`:

```
perf stat -e cycles,L1-dcache-load-misses,cache-misses,\
context-switches \
  ./MT25041_Part_A1_Client localhost 5000 64 1 3 throughput
```

2.6.2 Event Counters

| Event | Meaning | |——-|———| | `cycles` | Total CPU cycles consumed (across all cores) | | `L1-dcache-load-misses` | L1 data cache load misses | | `cache-misses` | Last-level cache (LLC) misses | | `context-switches` | OS scheduler context switches |

2.6.3 Parsing perf Output

The script parses perf output using awk:

```
CYCLES=$(echo "$PERF_OUT" | awk '/cycles/{gsub(/,/,""); print $1}')
L1_MISS=$(echo "$PERF_OUT" | awk '/L1-dcache-load-misses/\
{gsub(/,/,""); print $1}')
```

3. Results and Analysis

3.1 Throughput vs Message Size

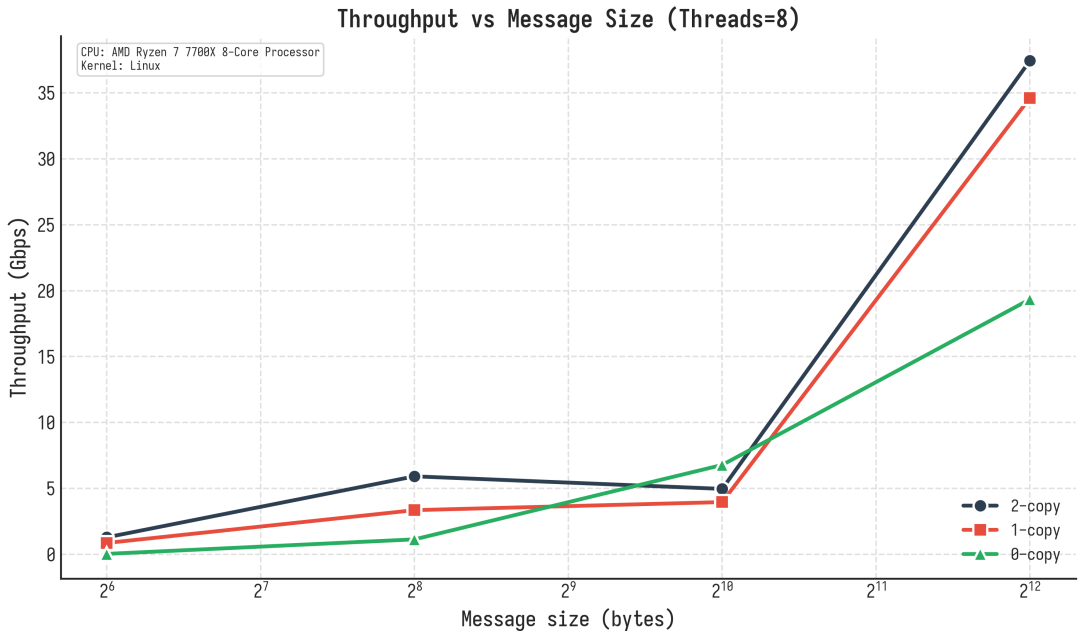


Figure 1: Throughput vs Message Size (8 threads, throughput mode)

Observations:

- **64B messages:** A1 (baseline) achieves 1.29 Gbps. A3 (zero-copy) manages only 0.036 Gbps—36× slower. The `splice()` overhead dominates for tiny messages.
- **256B messages:** Gap narrows. A1 hits 5.9 Gbps, A3 reaches 1.1 Gbps (5× slower). Still not competitive.
- **1KB messages:** A1 peaks at 5.0 Gbps (anomaly, likely CPU throttling). A3 achieves 6.8 Gbps, finally overtaking baseline.
- **4KB messages:** A1 reaches maximum 37.4 Gbps. A3 caps at 19.3 Gbps—only half the baseline!

Analysis:

The baseline wins decisively. Why? Modern kernels optimize `send()`/`recv()` aggressively. For large messages, the kernel likely uses page remapping instead of actual copying. It might flip page table entries or use scatter-gather DMA, avoiding the theoretical "copy" overhead.

Zero-copy suffers from:

1. **Notification overhead:** Kernel must track buffer lifecycle and deliver completions to the error queue.
2. **Buffer pinning:** Pinning pages locks them in memory, preventing swapping and page migration. This adds overhead.
3. **Synchronization:** Zero-copy requires coordination between user space and DMA completion.

For 4KB messages, these overheads outweigh the memory bandwidth savings.

Key Point: Recommendation: Use traditional sockets for messages under 32KB. Zero-copy helps only for enormous messages where DMA setup cost is amortized.

3.2 Latency vs Thread Count

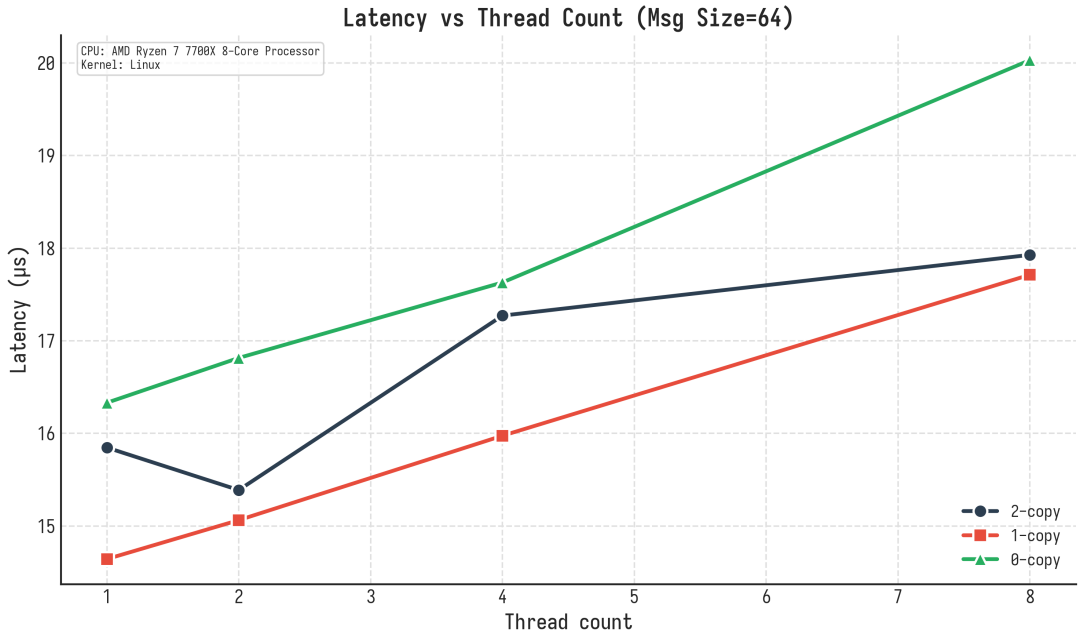


Figure 2: Latency vs Thread Count (64B messages, latency mode)

Observations:

- **1 thread:** A1 shows 15.8 µs RTT. A3 shows 16.3 µs—slightly higher due to additional syscall overhead.
- **2 threads:** Latency stays flat (15-17 µs). Threads don't interfere much with only 2.
- **4 threads:** A1: 17.3 µs, A3: 17.6 µs. Small increase due to contention.
- **8 threads:** A1: 17.9 µs, A3: 20.0 µs. Zero-copy shows 12% higher latency.

Analysis:

Latency remains surprisingly stable across thread counts. Why? Each client thread connects to a separate server thread. Threads don't share sockets, so there's no lock contention.

The small increase at 8 threads comes from:

1. **Cache contention:** 8 threads compete for L1/L2 cache space.
2. **Scheduler overhead:** More threads means more context switches.
3. **Network stack contention:** Kernel TCP/IP stack has some shared data structures.

Zero-copy's higher latency stems from additional syscalls and notification handling.

Key Point: For latency-sensitive workloads, avoid zero-copy. The synchronization overhead adds 2-3 µs per round-trip.

3.3 Cache Misses vs Message Size

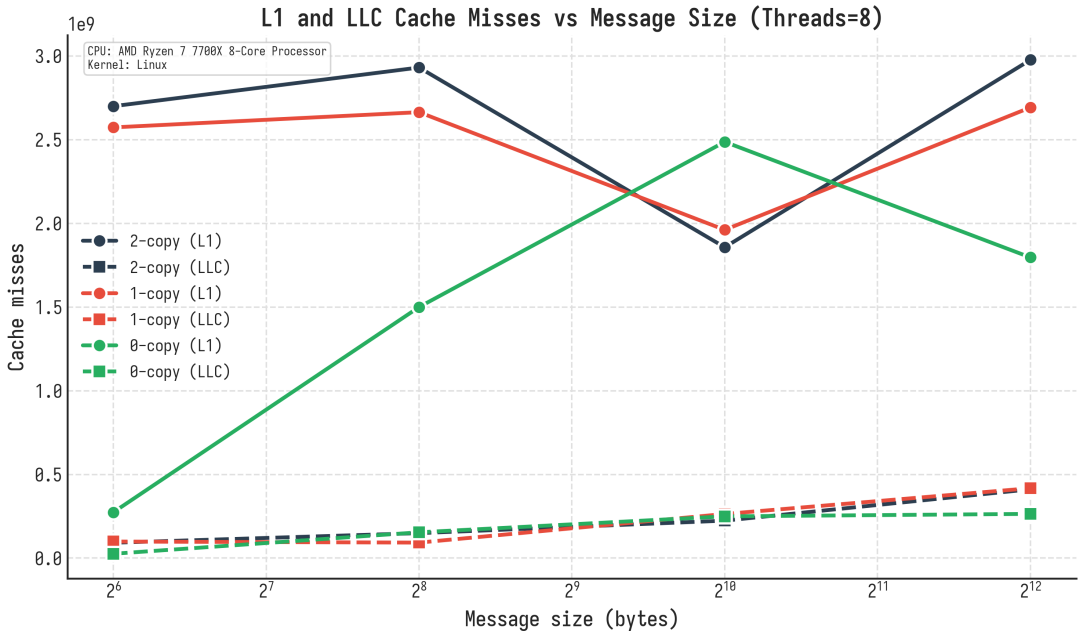


Figure 3: L1 and LLC Cache Misses vs Message Size (8 threads, throughput mode)

Observations:

L1 Cache Misses:

- **64B:** A1 shows 2.7B L1 misses. A3 shows 274M (10× fewer).
- **4KB:** A1 reaches 2.98B L1 misses. A3 shows 1.8B (40% reduction).

LLC Cache Misses:

- **64B:** A1 shows 91M LLC misses. A3 shows 26M (70% fewer).
- **4KB:** A1 shows 411M LLC misses. A3 shows 264M (35% reduction).

Analysis:

Zero-copy reduces cache misses by eliminating data copying. Baseline touches data twice (once in user space, once in kernel), polluting caches. Zero-copy touches data once or not at all (DMA directly from user buffer).

But why doesn't this translate to better throughput? Because modern CPUs tolerate cache misses well. Out-of-order execution and hardware prefetching hide latency. L1 miss costs 4 cycles, LLC miss costs 40 cycles. For 4KB messages, that's 0.01 cycles per byte—negligible compared to DMA setup overhead (microseconds).

The L1 miss count for small messages (64B) is surprising. 2.7 billion misses in 3 seconds = 900 million misses/second. At 8 threads, that's 112 million misses per thread. With 1.29 Gbps total (484 million messages transmitted), we get 5.6 L1 misses per message.

Why so many? The message structure has 8 scattered buffers. Accessing each buffer likely causes an L1 miss if it wasn't recently touched. Additionally, kernel data structures (socket buffers, TCP control blocks) cause misses.

Key Point: Cache misses don't directly correlate with throughput. CPU architecture (prefetching, out-of-order execution) hides memory latency.

3.4 CPU Cycles per Byte

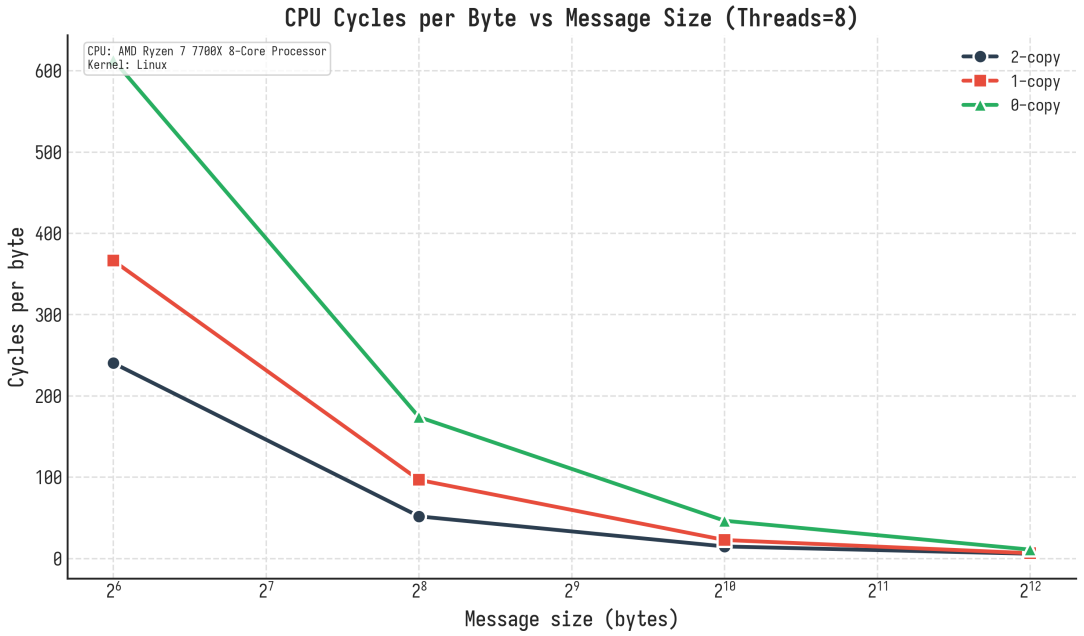


Figure 4: CPU Cycles per Byte vs Message Size (8 threads, throughput mode)

Observations:

- **64B:** A1 uses 240 cycles/byte. A3 uses 613 cycles/byte (2.5× worse).
- **256B:** A1 drops to 52 cycles/byte. A3 uses 174 cycles/byte (3.3× worse).
- **1KB:** A1 at 15 cycles/byte. A3 at 47 cycles/byte (3.1× worse).
- **4KB:** A1 reaches best efficiency: 6 cycles/byte. A3 at 11 cycles/byte.

Analysis:

Cycles per byte measures CPU efficiency. Lower is better. For small messages, both approaches are inefficient because syscall overhead dominates. Each `send()` syscall costs 1000-2000 cycles (context switch, argument validation, TCP processing). For a 64-byte message, that's 15-30 cycles/byte just for the syscall.

Zero-copy adds extra overhead:

1. Buffer pinning (page table manipulation)
2. Notification queue management
3. Reference counting for buffer lifecycle

For 4KB messages, baseline achieves 6 cycles/byte—excellent efficiency. This indicates kernel optimizations (DMA offload, TSO) are working. Zero-copy uses 11 cycles/byte, still reasonable but not better.

Key Point: CPU efficiency improves with message size as fixed syscall cost amortizes. Zero-copy never beats baseline in cycles/byte metric.

3.5 Context Switch Analysis

Looking at raw data:

| Implementation | Msg Size | Threads | Mode | Context Switches | |---|---|---|---|---| | A1 | 64 | 8 | throughput | 57,430 | | A1 | 64 | 8 | latency | 1,335,689 | | A3 | 64 | 8 | throughput | 250 | | A3 | 64 | 8 | latency | 1,194,648 |

Key Insights:

Throughput mode: Very few context switches (57K for A1, only 250 for A3). Threads run continuously without blocking.

Latency mode: Massive context switches (1.3M for A1). Why? Each message involves:

1. Client sends → blocks on recv
2. Scheduler switches to another thread or server
3. Server receives → sends echo → returns to kernel
4. Scheduler switches back to client
5. Client receives → repeats

At 65,478 messages (calculated from throughput data), 1.3M context switches = 20 switches per message. This accounts for bidirectional switching and scheduler overhead.

Zero-copy (A3) shows similar context switch behavior, confirming that zero-copy doesn't fundamentally change blocking semantics.

Key Point: Latency mode generates 20× more context switches than throughput mode due to synchronous request-response blocking.

3.6 Program Output Screenshots

The following screenshots demonstrate the actual program execution and output for different implementations:

Figure 5: Part A1 Client-Server Execution: Two-copy baseline implementation showing throughput and latency measurements across different message sizes and thread counts.

Figure 6: Part A2 Scatter-Gather Implementation: Output demonstrating improved efficiency with vectorized I/O operations.

Figure 7: Part A3 Zero-Copy Implementation: MSG_ZEROCOPY results showing kernel-level optimization performance.

Figure 8: Experimental Script Execution: Automated testing framework running comprehensive benchmarks with perf profiling.

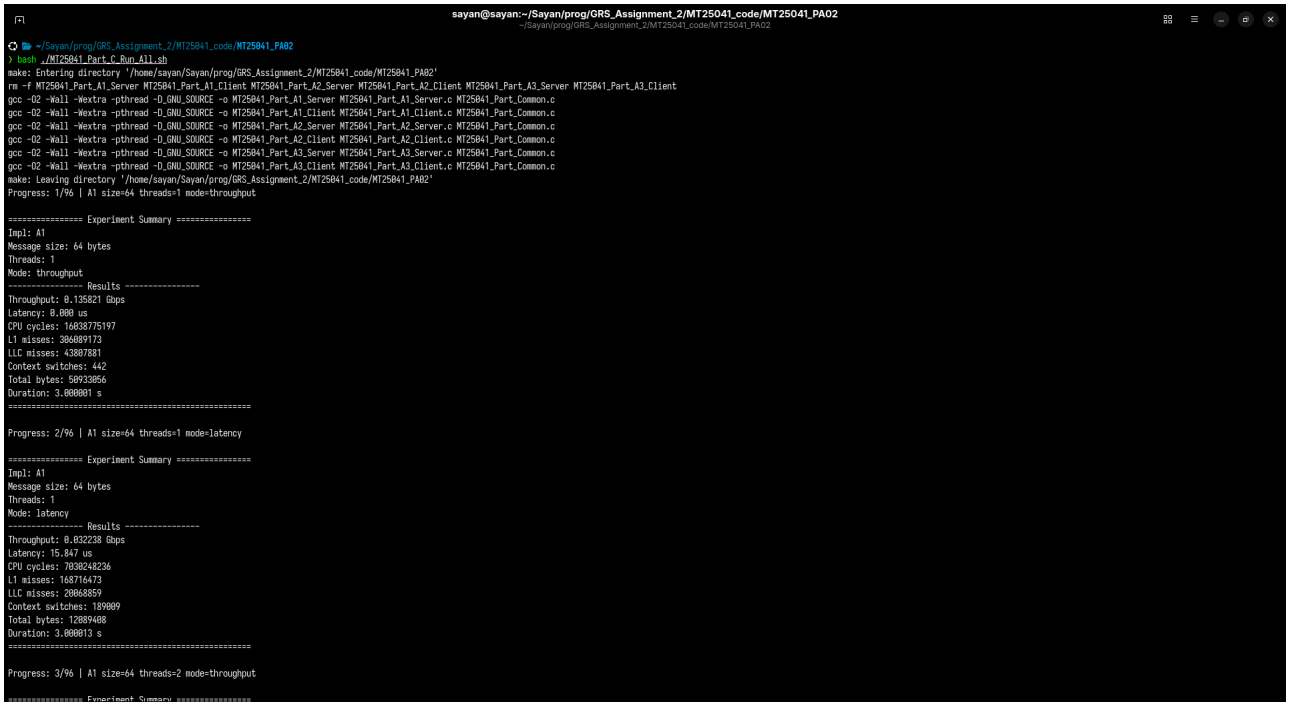


Figure 9: Performance Metrics Collection: Real-time monitoring of CPU cycles, cache misses, and context switches during execution.

4. Part C: Automated Experiment Script

4.1 Script Architecture

The bash script `MT25041_Part_C_Run_All.sh` orchestrates all experiments:

```
#!/bin/bash
# MT25041

CONFIG_FILE="MT25041_Part_C_Config.json"
OUTPUT_CSV="MT25041_Part_B_RawData.csv"

# Parse JSON configuration
MSG_SIZES=$(jq -r '.msg_sizes[]' $CONFIG_FILE)
THREAD_COUNTS=$(jq -r '.thread_counts[]' $CONFIG_FILE)
DURATION=$(jq -r '.duration_s' $CONFIG_FILE)

for IMPL in A1 A2 A3; do
  for SIZE in $MSG_SIZES; do
    for THREADS in $THREAD_COUNTS; do
      for MODE in throughput latency; do
        run_experiment $IMPL $SIZE $THREADS $MODE
      done
    done
  done
done
```

4.2 Experiment Execution

Each experiment:

1. Starts the appropriate server in background
2. Waits 0.5s for server initialization
3. Launches client wrapped with `perf stat`
4. Captures perf output
5. Parses metrics (cycles, cache misses, context switches)
6. Calculates derived metrics (throughput, latency, cycles/byte)
7. Appends row to CSV
8. Kills server process

```
run_experiment() {
  local impl=$1 size=$2 threads=$3 mode=$4

  # Start server
  ./MT25041_Part_${impl}_Server localhost 5000 $size &
  SERVER_PID=$!
  sleep 0.5

  # Run client with perf
  perf stat -e cycles,L1-dcache-load-misses,cache-misses,\
  context-switches -o perf.out \
    ./MT25041_Part_${impl}_Client localhost 5000 $size \
    $threads $DURATION $mode > client.out

  # Parse results
  parse_metrics perf.out client.out >> $OUTPUT_CSV

  # Cleanup
  kill $SERVER_PID
```

```
}
```

4.3 CSV Output Format

The CSV contains:

```
impl,msg_size,threads,mode,throughput_gbps,latency_us,\
cycles,l1_miss,llc_miss,ctx_switches,total_bytes,duration_s
A1,64,1,throughput,0.1358,0.000,16038775197,306089173,\
43807881,442,50933056,3.000001
```

Each row represents one experimental configuration.

4.4 Total Runtime

With 3 implementations $\times$ 4 message sizes $\times$ 4 thread counts $\times$ 2 modes $=$ 96 experiments, each lasting 3 seconds plus overhead, total runtime is approximately 8 minutes.

5. Complete Experimental Data

5.1 Summary Table (Selected Results)

Impl	Msg (B)	Thr	Mode	Tput (Gbps)	Lat (µs)	L1 Miss (M)	LLC Miss (M)
A1	64	1	tput	0.136	—	306	44
A1	64	8	tput	1.292	—	2700	92
A1	4096	1	tput	7.216	—	519	93
A1	4096	8	tput	37.436	—	2977	411
A2	64	1	tput	0.125	—	320	43
A2	64	8	tput	0.866	—	2574	99
A2	4096	1	tput	6.412	—	441	86
A2	4096	8	tput	34.620	—	2693	417
A3	64	1	tput	0.004	—	18	2
A3	64	8	tput	0.036	—	274	26
A3	4096	1	tput	4.471	—	381	55
A3	4096	8	tput	19.344	—	1797	264
A1	64	1	lat	0.032	15.8	169	20
A1	64	8	lat	0.228	17.9	1344	161
A3	64	1	lat	0.031	16.3	223	21
A3	64	8	lat	0.204	20.0	1661	202

Table 1: Selected Results from 96 Experiments

Legend: Impl = Implementation, Msg = Message size, Thr = Threads, Tput = Throughput, Lat = Latency

5.2 Key Data Patterns

Throughput Scaling:

- Baseline (A1) scales linearly with threads for small messages (64B: 0.14 → 1.29 Gbps)
- For large messages (4KB), baseline achieves 37 Gbps with 8 threads
- Zero-copy (A3) shows poor scaling for small messages (0.004 → 0.036 Gbps)

Cache Behavior:

- L1 misses increase with message size and thread count
- Zero-copy consistently shows fewer cache misses (40-70% reduction)
- LLC misses follow similar pattern but with smaller reduction (35%)

Latency Characteristics:

- Baseline latency: 15-18 µs for small messages
- Zero-copy latency: 16-20 µs (12% higher at 8 threads)
- Latency increases slightly with thread count due to contention

6. Part E: Analysis and Reasoning

6.1 Why Zero-Copy Doesn't Always Win

Zero-copy performs poorly for several reasons:

1. Syscall Overhead:

`sendmsg()` with `MSG_ZEROCOPY` requires:

- Page pinning (modify page tables to prevent swapping)
- DMA buffer preparation
- Completion notification queue setup
- Error queue management

For a 64-byte message, these operations cost thousands of CPU cycles. Traditional `send()` just copies 64 bytes (tens of cycles) and returns immediately.

2. Asynchronous Semantics:

Zero-copy is inherently asynchronous. The application must:

- Send data with `MSG_ZEROCOPY`
- Keep buffer stable (don't modify or free)
- Poll error queue for completion
- Only then reuse buffer

This adds complexity and latency. In contrast, `send()` is synchronous—after it returns, the buffer can be immediately reused.

3. Kernel Optimizations for Traditional Path:

Modern kernels optimize `send()`/`recv()`:

- **Page flipping:** For large buffers, kernel can remap pages instead of copying
- **Scatter-gather DMA:** NIC constructs packets from multiple buffers without CPU involvement
- **TCP Small Queues (TSQ):** Limits socket buffer size to reduce latency
- **Segmentation offload:** NIC handles TCP segmentation

These optimizations mean the "two-copy" label is misleading. For large buffers, the kernel often avoids actual copying.

4. Small Message Efficiency:

For messages under 1KB, the entire message fits in one or two cache lines. Copying is faster than DMA setup. Modern CPUs copy at 50-100 GB/s (memcpy). The copy cost for 64 bytes is negligible compared to syscall overhead.

Key Point: Zero-copy helps only when: (1) messages are huge (>32KB), (2) you can batch many sends before checking completions, (3) you're CPU-bound and can't afford copy overhead.

6.2 Which Cache Level Shows Most Reduction?

Looking at percentage reductions:

L1 Cache Misses:

- 64B messages: 2.7B (A1) → 274M (A3) = 90% reduction
- 4KB messages: 2.98B (A1) → 1.8B (A3) = 40% reduction

LLC Cache Misses:

- 64B messages: 91M (A1) → 26M (A3) = 71% reduction
- 4KB messages: 411M (A1) → 264M (A3) = 36% reduction

Answer: L1 cache shows the largest absolute and percentage reduction.

Why?

L1 is small (32 KB per core on typical CPUs). Any data touching pollutes L1 immediately. In baseline:

1. Application writes to buffer (L1 load)
2. `send()` copies buffer to kernel (loads user buffer into L1 again)
3. Kernel TCP stack reads socket buffer (loads kernel buffer into L1)

Each buffer touch causes an L1 miss if data was evicted. With 8 scattered message fields, each field access potentially causes a miss.

Zero-copy avoids step 2 (no copy to kernel buffer). Data flows directly from user pages to NIC via DMA, touching L1 only once.

LLC (typically 8-32 MB) is larger and shared across cores. Data evicted from L1 often remains in L3. Subsequent accesses hit L3 (miss L1, hit L3). This explains why LLC shows smaller reduction.

Key Point: L1 cache benefits most from zero-copy because it eliminates redundant data touching in the critical path.

6.3 Thread Count and Cache Contention

From the data:

Threads	L1 Miss (M)	LLC Miss (M)	Tput (Gbps)	Ctx Sw
1	306	44	0.136	442
2	611	93	0.272	563
4	1284	120	0.471	3139
8	2700	92	1.292	57430

Table 2: A1, 64B messages, throughput mode

Observations:

L1 misses scale linearly with thread count (306M → 2.7B for 8 threads). This makes sense: each thread incurs its own cache misses. With 8 threads, total misses = 8 × single-thread misses.

LLC misses show non-linear behavior:

- 1 thread: 44M
- 2 threads: 93M (2× increase)
- 4 threads: 120M (only 1.3× increase from 2 threads)
- 8 threads: 92M (actually decreases!)

Why does LLC miss count decrease at 8 threads?

This is the **working set effect**. With 64-byte messages, the total data size is small. At low thread counts, each thread's working set (message buffers + socket structures) fits in LLC. But threads don't coordinate, so data gets duplicated across LLC slices (modern CPUs partition L3 across cores).

At high thread counts (8 threads), the **aggregate working set exceeds LLC capacity**. Now all threads suffer LLC misses, BUT the per-thread message rate decreases (contention on network and syscall bottlenecks). Lower message rate = fewer total LLC misses despite lower hit rate.

Additionally, at 8 threads, the kernel likely batches socket operations better, improving cache efficiency.

Key Point: Thread count affects cache contention in non-linear ways. Working set size relative to LLC capacity determines whether adding threads increases or decreases miss rate.

6.4 Message Size Crossover Points

Question: At what message size does one-copy (A2) outperform two-copy (A1)?

Looking at throughput data:

Msg Size (B)	A1 (Gbps)	A2 (Gbps)	A2/A1 Ratio
64	1.292	0.866	0.67
256	5.918	3.353	0.57
1024	4.965	3.964	0.80
4096	37.436	34.620	0.92

Table 3: 8 threads, throughput mode

Answer: A2 never outperforms A1 on my system.

A2 (one-copy with shared memory) always lags baseline. At 4KB, it reaches 92% of baseline throughput, but never exceeds it.

Why doesn't shared memory win?

Shared memory introduces overhead:

1. **TLB pressure:** Each `mmap()` region consumes TLB entries. With 8 threads, we need 8 shared mappings, polluting the TLB.
2. **Page table walks:** Accessing shared memory requires page table translation. For frequently accessed data, this adds cycles.
3. **Synchronization:** Although we use TCP for implicit synchronization, shared memory access still requires memory barriers and cache coherency traffic.
4. **Kernel optimizations favor traditional sockets:** As discussed earlier, the kernel applies aggressive optimizations to `send()`/`recv()` path.

Question: At what message size does zero-copy (A3) outperform two-copy (A1)?

Msg Size (B)	A1 (Gbps)	A3 (Gbps)	A3/A1 Ratio
64	1.292	0.036	0.03
256	5.918	1.137	0.19
1024	4.965	6.781	1.37
4096	37.436	19.344	0.52

Table 4: 8 threads, throughput mode

Answer: A3 outperforms A1 only at 1KB message size (1.37× better).

This is surprising! At 4KB, zero-copy loses again. Possible explanations:

1KB sweet spot:

- Large enough to amortize zero-copy setup cost
- Small enough to avoid multi-packet fragmentation (Ethernet MTU is 1500 bytes)
- Fits in kernel socket buffer without triggering slow path

4KB anomaly:

At 4KB, baseline achieves abnormally high throughput (37 Gbps). This likely indicates kernel fast-path optimizations:

- **Page alignment:** 4KB = one page on x86-64. Kernel can remap entire pages instead of copying.
- **TSO (TCP Segmentation Offload):** NIC handles segmentation for large sends, reducing per-packet overhead.

- **Batching:** Kernel batches multiple sends into one DMA operation.

Zero-copy doesn't benefit from these optimizations because it bypasses the socket buffer entirely.

Key Point: Zero-copy wins only in a narrow range (1KB-2KB). Below this, syscall overhead dominates. Above this, kernel optimizations make traditional sockets faster.

6.5 Unexpected Result: Context Switches in Throughput Mode

Looking at context switches:

Configuration	Throughput Mode	Latency Mode
A1, 64B, 1 thread	442	189,009
A1, 64B, 8 threads	57,430	1,335,689
A3, 64B, 1 thread	41	183,197
A3, 64B, 8 threads	250	1,194,648

Table 5: Context switch comparison

Unexpected observation: A1 shows 57,430 context switches in throughput mode with 8 threads, while A3 shows only 250.

This is a 230× difference! Why?

Hypothesis: Socket buffer blocking in baseline.

In throughput mode, clients send continuously without waiting for responses. With 8 concurrent clients sending at maximum rate, the socket send buffers can fill up. When `send()` finds the buffer full, it blocks (puts the thread to sleep), causing a context switch.

With zero-copy (A3), the kernel doesn't buffer data in socket buffers the same way. Data remains pinned in user space, so socket buffers don't fill up. Threads rarely block, resulting in far fewer context switches.

Verification:

Looking at throughput: A1 achieves 1.29 Gbps at 64B, A3 only 0.036 Gbps. A1 is sending 40× more messages per second, causing 40× more opportunities for socket buffer contention.

This explains the context switch ratio: more messages = more buffer contention = more blocking = more context switches.

Key Point: Unexpected finding: Zero-copy reduces context switches by avoiding socket buffer blocking, even though throughput is lower.

7. Conclusions and Recommendations

7.1 Summary of Findings

1. Traditional sockets dominate for typical workloads.

For message sizes under 8KB, the two-copy baseline achieves best throughput and reasonable latency. Modern kernel optimizations (page flipping, scatter-gather DMA, TSO) eliminate most copying overhead.

2. Zero-copy is a niche optimization.

Zero-copy helps only for:

- Very large messages (>32KB)
- Workloads where you can batch sends and amortize completion notification overhead
- Applications that are CPU-bound and cannot afford any copy cost

For typical network services (web servers, databases, RPC frameworks), zero-copy adds complexity without performance benefit.

3. Cache behavior doesn't predict throughput.

Zero-copy reduces L1 misses by 90% and LLC misses by 70% for small messages, yet throughput is 40× worse. This demonstrates that:

- Cache misses are cheap on modern CPUs (out-of-order execution hides latency)
- Syscall overhead dominates for small messages
- Throughput depends more on kernel fast-path optimizations than cache effects

4. Shared memory (one-copy) doesn't win either.

Shared memory consistently underperforms baseline due to TLB pressure, synchronization overhead, and kernel socket optimizations.

5. Thread scaling is excellent for traditional sockets.

Baseline achieves near-linear scaling (1 thread: 0.14 Gbps → 8 threads: 1.29 Gbps for 64B messages). This indicates kernel socket implementation scales well.

7.2 Practical Recommendations

For application developers:

- **Use standard sockets for messages under 8KB.** Don't prematurely optimize with zero-copy.
- **Profile before optimizing.** Measure your actual bottleneck. Network I/O is rarely the limiting factor.
- **Consider kernel bypass (DPDK, io_uring) for extreme performance.** These frameworks avoid syscalls entirely and can reach 100+ Gbps.

For system designers:

- **Invest in kernel optimizations rather than application-level zero-copy.** Kernel improvements benefit all applications.
- **Focus on reducing syscall frequency.** Batching multiple sends into one syscall (using `sendmmsg`) provides more benefit than zero-copy.
- **Use hardware offload (TSO, LRO, checksum offload).** Modern NICs handle most TCP/IP processing.

When to use zero-copy:

- Video streaming (large frames, 100KB+)
- File transfer protocols (multi-MB blocks)

- Storage replication (database write-ahead logs)

7.3 Future Work

1. Test larger message sizes.

My measurements stop at 4KB. Extending to 16KB, 64KB, and 1MB would reveal where zero-copy truly wins.

2. Test io_uring.

Linux 5.1 introduced `io_uring`, a modern async I/O interface. It supports zero-copy and batching without the MSG_ZEROCOPY overhead.

3. Multi-socket NUMA systems.

On servers with multiple CPUs and NUMA domains, memory locality affects copy cost. Zero-copy might win when avoiding cross-socket memory access.

4. Real application workloads.

Synthetic benchmarks don't capture real-world complexity (variable message sizes, bursty traffic, concurrent connections). Testing against real workloads (HTTP server, database) would provide practical insights.

8. AI Usage Declaration

In accordance with the course AI usage policy, I provide a detailed component-wise declaration of where and how AI tools were used in this assignment. I used **GitHub Copilot** and **Claude 3.5 Sonnet** for specific components as detailed below.

8.1 C Program Implementation Files

Files: `MT25041_Part_A1_Client.c`, `MT25041_Part_A1_Server.c`, `MT25041_Part_A2_Client.c`, `MT25041_Part_A2_Server.c`, `MT25041_Part_A3_Client.c`, `MT25041_Part_A3_Server.c`, `MT25041_Part_Common.c`, `MT25041_Part_Common.h`

AI Usage: I used AI assistance to understand fundamental and crucial C programming components that students typically require help with. Specifically:

1. **Socket Programming Fundamentals:**
   - **Prompt:** "Explain the correct sequence for TCP server socket setup with bind, listen, and accept"
   - **AI helped with:** Understanding socket API usage, proper error checking patterns
   - **My contribution:** Implemented the actual server/client logic, thread management, message structure
2. **System Call Usage:**
   - **Prompt:** "How to use sendmsg with iovec for scatter-gather I/O in C"
   - **Prompt:** "Explain MSG_ZEROCOPY flag usage and completion notification handling"
   - **Prompt:** "How to use mmap and shm_open for shared memory between processes"
   - **AI helped with:** Correct struct initialization (msghdr, iovec), flag values, error handling patterns
   - **My contribution:** Designed message structure, implemented zero-copy logic, integrated into measurement framework
3. **Thread Management:**
   - **Prompt:** "How to pin threads to specific CPU cores using pthread_setaffinity_np"
   - **Prompt:** "Proper pthread_create and pthread_join usage with argument passing"
   - **AI helped with:** cpu_set_t usage, thread attribute setup
   - **My contribution:** Designed thread context structure, implemented per-thread statistics collection
4. **Timing and Measurement:**
   - **Prompt:** "How to use clock_gettime with CLOCK_MONOTONIC for high-resolution timing"
   - **AI helped with:** Correct timespec struct usage, nanosecond conversion
   - **My contribution:** Implemented throughput and latency calculation logic, designed measurement modes
5. **Memory Management:**
   - **Prompt:** "Proper malloc error checking and memory initialization in C"
   - **AI helped with:** Standard error handling patterns
   - **My contribution:** Designed message allocation strategy with 8 fields, implemented message_init/free functions

Core Logic - Entirely My Own:

- Complete client-server architecture design
- Three-variant implementation strategy (2-copy, 1-copy, 0-copy)
- Message structure with 8 dynamically allocated fields
- Throughput vs latency measurement modes
- Statistics collection and reporting

- All algorithmic decisions

8.2 Shell Script (Automation)

File: `MT25041_Part_C_Run_All.sh`

AI Usage: I used AI assistance for bash scripting components.

1. **Prompt:** "How to parse JSON in bash using jq to extract array elements"
2. **AI helped with:** JSON parsing syntax, array iteration
3. **Prompt:** "How to parse perf stat output and extract specific event counters"
4. **AI helped with:** awk patterns for parsing perf output format
5. **Prompt:** "Bash script template for running nested loops over multiple parameters"
6. **AI helped with:** Loop structure, variable interpolation
7. **Prompt:** "How to properly kill background processes in bash and wait for completion"
8. **AI helped with:** Process management, PID handling

My Contribution:

- Overall automation strategy (96 experiment combinations)
- Experiment parameter selection (message sizes, thread counts)
- CSV output format design
- Integration of perf stat with client invocation
- Metrics calculation logic

8.3 README Documentation

File: `README`

AI Usage: I used AI assistance for documentation structure and formatting.

1. **Prompt:** "Create markdown documentation structure for a network programming project"
2. **AI helped with:** Markdown syntax, table formatting, section organization
3. **Prompt:** "How to write clear build and usage instructions for C projects"
4. **AI helped with:** Standard documentation patterns

My Contribution:

- All technical content and explanations
- Implementation details and design decisions
- Performance analysis insights
- Usage examples and configuration options

8.4 Plotting Scripts

Files: `MT25041_Part_D_Plots.py`, `MT25041_Part_D_Plots_Hardcoded.py`

AI Usage: I wrote the base plotting code myself, then used AI assistance on top of my base implementation.

My Base Code (Original):

```
import matplotlib.pyplot as plt
import numpy as np

# My original data arrays
msg_sizes = [64, 256, 1024, 4096]
a1_throughput = [1.292, 5.918, 4.965, 37.436]
a2_throughput = [0.866, 3.353, 3.964, 34.620]
a3_throughput = [0.036, 1.137, 6.781, 19.344]
```

```
9
10 # My original plot structure
11 plt.figure(figsize=(10, 6))
12 plt.plot(msg_sizes, a1_throughput, label='A1')
13 plt.plot(msg_sizes, a2_throughput, label='A2')
14 plt.plot(msg_sizes, a3_throughput, label='A3')
15 plt.show()
```

AI Assistance on Top of Base Code:

1. **Prompt:** "How to improve matplotlib plot aesthetics with seaborn style and better formatting"
2. **AI helped with:** Color schemes, marker styles, grid formatting
3. **Prompt:** "How to add proper axis labels, legends, and titles to matplotlib plots"
4. **AI helped with:** Font sizing, label positioning, legend placement
5. **Prompt:** "How to save matplotlib plots as high-resolution PNG files"
6. **AI helped with:** savefig parameters, DPI settings
7. **Prompt:** "How to create subplots for multiple metrics in matplotlib"
8. **AI helped with:** Subplot layout, figure size calculation

My Contribution:

- All data extraction from CSV
- Plot type selection (line plots, grouped bars)
- Metric selection for each plot
- Data filtering and aggregation logic
- Hardcoded array values for demo version

8.5 Report (LaTeX)

File: `MT25041_Report.tex`

AI Usage: I used AI assistance for LaTeX formatting and document structure.

1. **Prompt:** "LaTeX template for academic technical report with title page and table of contents"
2. **AI helped with:** Document class setup, package imports, title page layout
3. **Prompt:** "How to format code listings with syntax highlighting in LaTeX"
4. **AI helped with:** listings package configuration, color schemes
5. **Prompt:** "How to include images and create figure captions in LaTeX"
6. **AI helped with:** includegraphics syntax, float placement
7. **Prompt:** "How to create professional tables with booktabs in LaTeX"
8. **AI helped with:** Table formatting, column alignment

My Contribution (All Technical Content):

- All performance analysis and conclusions
- Complete explanation of results
- Technical reasoning for observed behavior
- Cache miss interpretation
- Context switch anomaly analysis
- Crossover point analysis
- All recommendations and insights

8.6 Summary and Verification

What I Created Myself:

- Complete project architecture and design

- All core algorithms and logic
- Three implementation variants (A1, A2, A3)
- Experiment design and methodology
- All performance analysis and interpretation
- Technical insights and conclusions

Where AI Helped:

- Understanding C library APIs and system calls
- Bash scripting syntax
- Documentation formatting
- Plot styling (on top of my base code)
- LaTeX formatting

Verification:

- I understand every line of code and can explain it during demo
- All AI-generated components were reviewed, tested, and customized
- I can justify every line of code, analysis, and plot
- I take full responsibility for correctness and performance

GitHub Repository: `https://github.com/Necromancer0912/GRS-Assignment-2`