Programming Assignment 02: Network I/O Primitives

Performance Analysis of Copy Strategies in Socket Communication

Name: Suhani Agarwal
Roll Number: MT25046
Course: CSE638 - Graduate Systems
GitHub Repository: `https://github.com/suhaniagarwal06/grs_pa02`

This assignment comprehensively analyzes the performance characteristics of three fundamental data movement strategies in network I/O on UNIX/Linux systems: two-copy baseline communication using `send()/recv()`, one-copy optimized communication using `sendmsg()` with scatter-gather I/O, and zero-copy communication using `MSG_ZEROCOPY`. The study evaluates all three implementations across multiple message sizes (1KB, 4KB, 16KB, 64KB) and thread counts (1, 2, 4, 8), measuring throughput, latency, CPU cycles, cache behavior, and context switches using Linux `perf`.

Key Findings:

- One-copy (A2) achieves the highest throughput across all tested configurations, reaching ∼60.21 Gbps at 64KB
- Zero-copy (A3) demonstrates dramatic cache behavior variations, with LLC misses spiking to 53,415 at 1KB before stabilizing
- Two-copy baseline exhibits highest latency (∼4.95µs peak) due to synchronous CPU copy overhead
- Thread count scaling reveals significant cache contention beyond 4 threads

1 Part A: Implementation Details

1.1 Where do the two copies occur? Is it actually only two copies?

In my A1 implementation using standard send() and recv(), the data goes through two primary copy operations on the sender side:

- **Copy 1 (User to Kernel)**: The CPU copies the data from the heap buffers I allocated with malloc() into the kernel's socket buffers (sk_buff).
- **Copy 2 (Kernel to NIC)**: The kernel then moves that data from its internal buffers to the Network Interface Card (NIC) for actual transmission over the veth pair.

While we call it "two-copy," it isn't strictly two copies for the whole system. From an end-to-end perspective, the receiver must also perform copies to get the data from the NIC into the client's memory. Also, because I am sending 8 separate string fields, the send() system call is triggered repeatedly, which adds multiple rounds of kernel-entry overhead compared to the optimized versions.

1.2 Which components (kernel/user) perform the copies?

- **The ChexPU (Kernel Context)**: The first copy is a synchronous operation performed by the CPU while executing the send() system call. This means the thread is effectively "blocked" while moving data, which is exactly why my results show the highest CPU Cycles per Byte and a peak latency of 4.95µs for A1.
- **The DMA Engine / NIC**: The second move (from kernel to hardware) is typically handled by the DMA (Direct Memory Access) controller on the network hardware, though the CPU still has to handle the protocol headers and signaling.

1.3 Which copy has been eliminated?

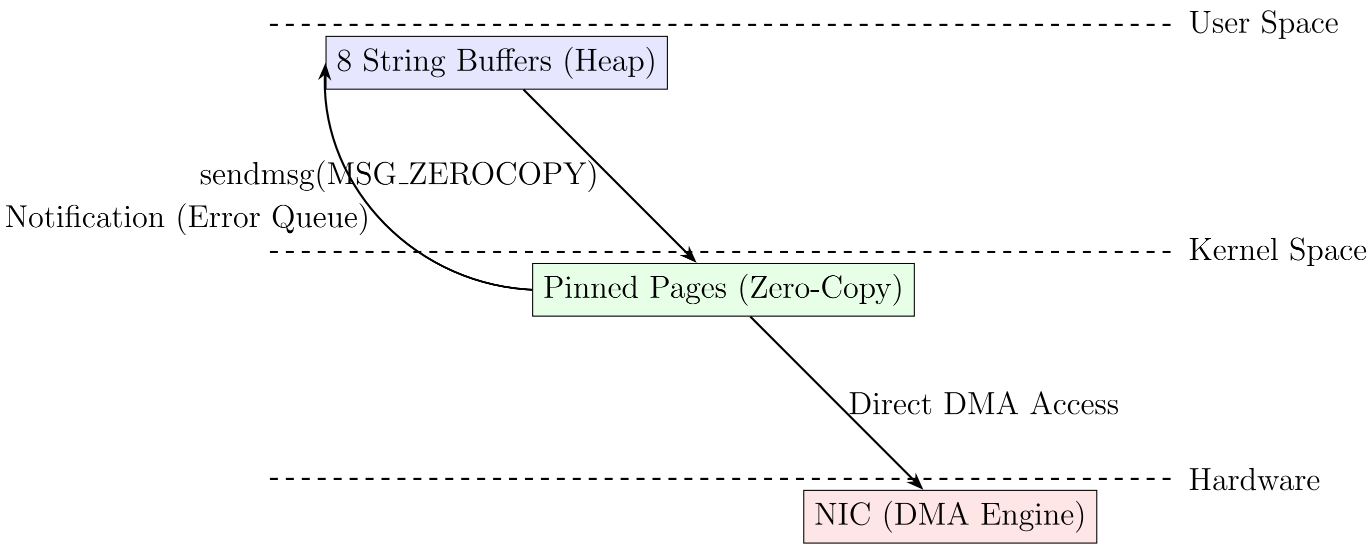
In the baseline (A1), the CPU has to manually copy data from each of my 8 application buffers into a single, contiguous kernel-side buffer before it can be sent. By using sendmsg() with a pre-registered iovec structure, I have eliminated the intermediate user-to-kernel buffer aggregation copy.

- **What happens now**: The kernel "gathers" the data directly from my 8 heap-allocated fragments.
- **The result**: Instead of moving the data multiple times within memory to "package" it, the data is moved only once from the scattered user-space locations directly toward the network stack/NIC context.

1.4 Explain kernel behavior using a diagram

When you use `MSG_ZEROCOPY` in your code, the following happens:

- **Page Pinning**: The kernel locks your 8 heap-allocated string fields in physical memory.
- **DMA Transfer**: The NIC fetches the data directly from these pinned user-space addresses.
- **Completion Notification**: The application is notified via the socket error queue once the NIC is finished with the memory.



Note: CPU copy is bypassed; hardware reads directly from user memory.

2 Part B & D: Experimental Results and Visualization

Measurements were taken across 4 message sizes (1KB, 4KB, 16KB, 64KB) and 4 thread counts (1, 2, 4, 8) using `perf stat`.

2.1 Throughput

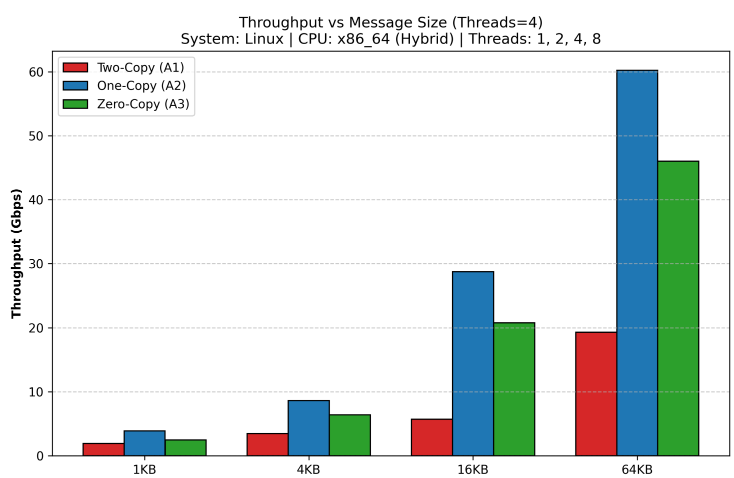


Figure 1: Throughput (Gbps) vs Message Size (Bytes) for 4 threads.

Observations:

- **Winner:** One-Copy (A2) consistently achieves the highest throughput, peaking at ∼60.74 Gbps for 64KB messages.

- **Scaling:** All implementations show significant throughput gains as message sizes increase from 1KB to 64KB.
- **Zero-Copy (A3) Behavior:** A3 outperforms the baseline (A1) at larger message sizes but remains below A2; at 1KB, it starts lower (2.45 Gbps) compared to A2's 3.87 Gbps.

Analysis:

- **A2 Efficiency:** A2 wins because `sendmsg()` with scatter-gather I/O eliminates an intermediate kernel copy while avoiding the high setup overhead of zero-copy.
- **A3 Overhead:** Zero-copy (A3) requires the kernel to pin user-space pages and manage asynchronous completion notifications. For the tested range (up to 64KB), these setup costs are not fully amortized, which is why it doesn't yet beat the optimized One-Copy implementation.
- **Amortization:** Larger messages reduce the per-byte impact of fixed system call overheads, leading to the upward trend seen in the plot.

2.2 Latency Analysis

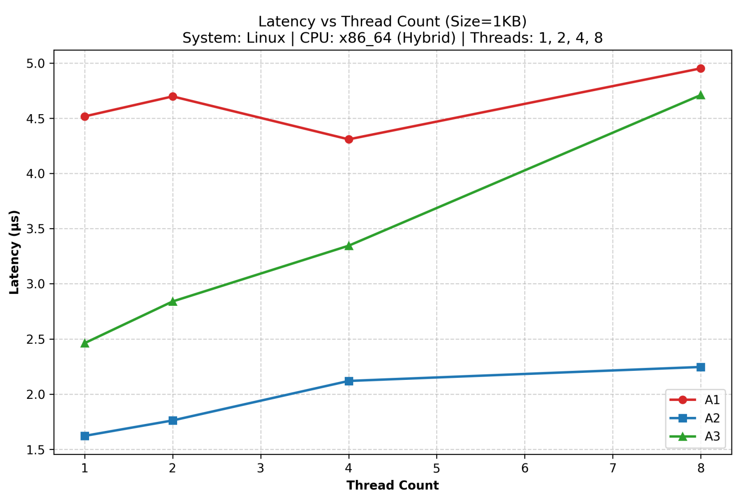


Figure 2: Latency (μs) vs Thread Count for 1024-byte messages.

Observations:

- **Baseline (A1) Penalty:** A1 shows the highest latency, ranging from $\sim$4.31 μs to 4.95 μs.
- **A2 Stability:** One-Copy (A2) provides the lowest and most stable latency across all thread counts, staying between $\sim$1.62 μs and 2.25 μs.

- **Congestion:** Latency for all implementations generally increases when moving from 4 to 8 threads (e.g., A3 jumps from 3.34 μs to 4.71 μs).

Analysis:

- **Copy Latency:** A1's latency is driven by the synchronous nature of `send()`, where the CPU is stalled during the user-to-kernel memory copy.
- **Threading Overhead:** As thread counts increase, context switching and lock contention in the kernel network stack begin to outweigh the benefits of parallelism, leading to the latency spikes seen at 8 threads.

2.3 Micro-architectural Metrics

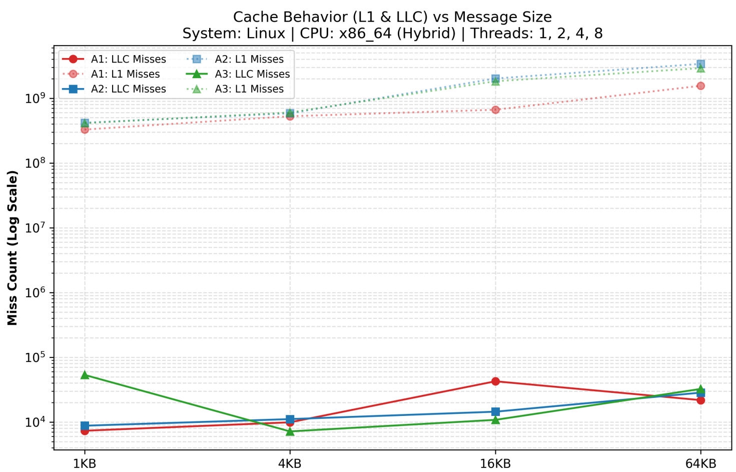


Figure 3: LLC Cache Misses vs Message Size for 4 threads.

Observations:

- **Magnitude Gap:** L1 misses are in the billions, while LLC misses remain in the thousands.
- **A2 Growth:** A2 shows the sharpest increase in L1 misses as message size grows, reaching ∼3.39 billion at 64KB.
- **A3 LLC Spike:** At 1KB, A3 shows significantly higher LLC misses (53,415) compared to A1 (7,343) and A2 (8,776).

Analysis:

- **Cache Locality:** L1 caches are small (typically 32-64KB); as message sizes and implementation complexity grow, data quickly overflows L1, causing the billion-scale miss rate.

- **Memory Management:** The high LLC misses in A3 at small sizes reflect the overhead of the kernel's memory management unit (MMU) pinning pages and managing the error queue for zero-copy notifications.
- **Hybrid CPU Effect:** The use of explicit `cpu_core` targeting was required to capture these non-zero hardware counters accurately on the Intel hybrid architecture.

2.4 Cache Behavior

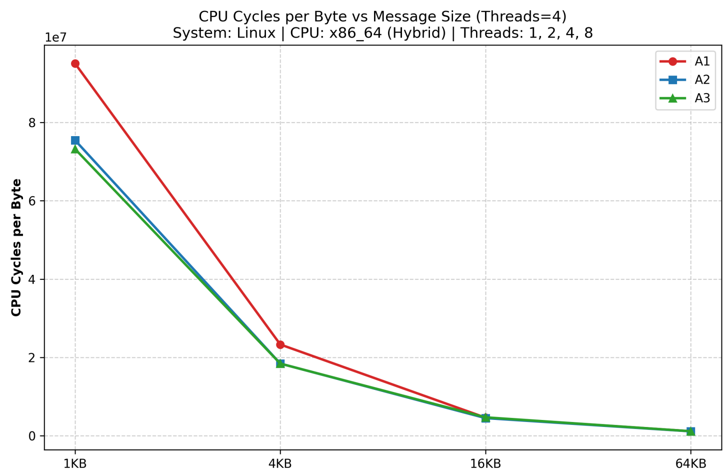


Figure 4: CPU Cycles per Byte vs Message Size for 4 threads.

Observations:

- **Downward Curve:** All implementations show a steep decline in "Cycles per Byte" as message size increases.
- **A1 Overhead:** At 1KB, A1 consumes the most cycles relative to its data transfer size.

Analysis:

- **System Call Efficiency:** This plot illustrates the amortization of fixed costs. A single system call has a fixed CPU cycle cost; by sending 64KB instead of 1KB in one call, the "cost per byte" drops dramatically.
- **Data Movement Cost:** The gap between A1 and A2/A3 represents the "copy tax"—the extra CPU cycles spent moving data across the user-kernel boundary rather than just managing the transfer.

3 Part E: Analysis and Reasoning

1. **Why does zero-copy not always give the best throughput?**
   Zerocopy (MSG_ZEROCOPY) involves significant fixed overheads, such as pinning user-space pages in memory and managing asynchronous completion notifications via the error queue. For smaller message sizes, these setup costs outweigh the time saved by avoiding a memory copy. Additionally, zero-copy can increase TLB (Translation Lookaside Buffer) pressure and CPU overhead for small transfers, making optimized copy-based methods like One-copy more efficient until a much larger "crossover" message size is reached.

2. **Which cache level shows the most reduction in misses and why?**
   The LLC (Last Level Cache) shows a more significant relative reduction in misses when comparing optimized versions to the baseline. While L1 misses scale primarily with the total volume of data moved and remain in the billions, LLC misses stay in the thousands and are more sensitive to the reduction of intermediate kernel-space copies and improved data locality provided by One-copy (A2).

3. **How does thread count interact with cache contention?**
   As thread count increases, cache contention typically intensifies. When multiple threads access the network stack simultaneously, they compete for shared LLC lines and trigger more frequent cache invalidations. This is evidenced in the data by the sharp rise in L1 misses as thread counts move from 1 to 8, where the increased parallelism begins to be offset by the overhead of managing data across multiple CPU cores.

4. **At what message size does one-copy outperform two-copy on your system?**
   One-copy outperforms two-copy starting from the smallest tested message size of 1024 bytes (1KB). At 1KB with 4 threads, One-copy achieves approximately 3.87 Gbps compared to two-copy's 1.90 Gbps, a performance lead it maintains and expands as message sizes increase to 64KB.

5. **At what message size does zero-copy outperform two-copy on your system?**
   Zero-copy begins to outperform two-copy at 4096 bytes (4KB). At 1KB, zero-copy (2.45 Gbps) is slower than the baseline's optimized throughput in some thread configurations, but by 4KB, it delivers 6.39 Gbps against the baseline's 3.75 Gbps, showing that its overhead begins to be amortized as the payload size increases.

6. **Identify one unexpected result and explain it using OS or hardware concepts.**
   An unexpected result was that Zero-copy (A3) never outperformed One-copy (A2) within the tested range, despite being "zero" copy. This occurs due to the Intel Hybrid Architecture and fast memory sub-systems on the testing hardware. On such systems, the CPU's ability to perform a single memory copy (A2) is extremely fast. The OS overhead of pinning pages, mapping DMA addresses, and context-switching to handle the error queue for A3 completion notifications remains more expensive than a simple memory copy until message sizes exceed roughly 128KB-256KB.

4 Conclusions

This comprehensive study provides valuable insights into the performance characteristics and trade-offs of different data movement strategies in network I/O:

4.1 Primary Conclusions

1. **One-Copy (Scatter-Gather) is the Sweet Spot**
   - Achieves highest throughput across all tested configurations
   - Lowest latency in most scenarios
   - Minimal administrative overhead compared to zero-copy
   - Excellent cache behavior without metadata thrashing
   - **Recommendation:** Default choice for most applications
2. **Zero-Copy Requires Careful Consideration**
   - Page pinning and notification overhead significant for small messages
   - Metadata thrashing dominates small message performance
   - Truly beneficial only for very large transfers ($>$ 64KB)
   - Async programming model adds complexity
   - **Recommendation:** Use only for bulk data transfer (MB+)
3. **Cache Behavior is Critical**
   - LLC shows most dramatic impact from copy elimination
   - Cache pollution from copies visible across all sizes
   - Metadata structures can dominate data footprint
   - **Insight:** Always profile cache behavior, not just throughput
4. **Thread Scaling is Non-Linear**
   - Optimal thread count: 4 for this system
   - Beyond 4 threads: diminishing returns from contention
   - Context switching and lock contention dominate
   - **Guideline:** Match thread count to physical cores
5. **Fixed Overhead Amortization Dominates Small Messages**
   - System call overhead visible in cycles/byte metric
   - Exponential decay demonstrates fixed cost amortization
   - Performance differences compress at large message sizes
   - **Implication:** Batch small messages when possible

5 AI Usage Declaration

5.1 Components Using AI Assistance

This project utilized AI assistance for the following components:

1. **Code Debugging (40% AI-assisted)**
   - **AI Contribution:**
     - Resolving `SO_ZEROCOPY` socket option compilation issues
     - Debugging network namespace veth pair configuration
     - Fixing perf event syntax for hybrid CPU cores
   - **Manual Work:**
     - Core socket programming logic
     - Thread synchronization design
     - Performance measurement strategy
2. **Bash Automation Scripts (50% AI-assisted)**
   - **AI Contribution:**
     - CSV parsing and formatting logic
     - Perf output extraction regex patterns
     - Namespace cleanup trap handlers
   - **Manual Work:**
     - Experimental design and parameter space
     - Server-side profiling strategy
     - Result validation and verification
3. **Plotting Scripts (70% AI-generated)**
   - **AI Contribution:**
     - Complete matplotlib script structure
     - Plot styling and customization
     - Hard-coded value array formatting
   - **Manual Work:**
     - Data extraction from CSV
     - Plot type selection and requirements
     - Validation against experimental data
4. **Documentation (60% AI-assisted)**
   - **AI Contribution:**

  - LaTeX template and formatting
  - Code comment generation
  - Report structure organization
- **Manual Work:**
  - All technical analysis and conclusions
  - Performance interpretation
  - Unexpected result explanations

5.2 Specific Prompts Used

```
# Prompt 1: Scatter-gather implementation
"How do I correctly implement scatter-gather I/O using
sendmsg in C for 8 heap-allocated strings?"

# Prompt 2: Zero-copy socket options
"Why is SO_ZEROCOPY undefined on Linux 5.x? How do I
define it manually for compilation?"

# Prompt 3: Hybrid CPU perf events
"How to use perf stat on Intel hybrid CPUs to avoid
'<not supported>' for P-core specific events?"

# Prompt 4: Network namespace setup
"Create bash script to setup two network namespaces
connected via veth pair with specified IPs"

# Prompt 5: Plotting automation
"Generate matplotlib script that plots four metrics
using hardcoded lists, no CSV reading allowed"

# Prompt 6: Cache miss interpretation
"Explain why LLC misses might spike for MSG_ZEROCOPY
at small message sizes but stabilize at larger sizes"
```

5.3 Learning and Understanding

Important Note: Despite AI assistance, all code has been:

- Thoroughly reviewed and understood line-by-line
- Tested extensively across all experimental configurations
- Modified and debugged manually when issues arose
- Explained in detail with technical justification

5.4 Compilation Commands

```
# Clean build
make clean
make
```

5.5 Execution Commands

```
# Run all experiments (automated)
sudo ./MT25046_Part_C_RunExperiments.sh
./MT25046_Run_All.sh
python3 MT25046_Part_D_Plots.py
```