PA02: Network I/O Data Movement Cost Analysis

Student: Arjun Prasad
Roll No: MT25064
System: Intel ThinkCentre M70s Gen 3 (Ubuntu 22.04.5 LTS)

Introduction

This report presents an experimental analysis of data movement costs in TCP socket communication by comparing three implementations:
A1: Two-copy baseline using `send()`
A2: One-copy optimization using `sendmsg()` with scatter-gather I/O
A3: Zero-copy implementation using `MSG_ZEROCOPY`
The experiments measured throughput, CPU cycles, cache misses, and context switches across varying message sizes (64B - 64KB) and thread counts (1-8).

Part A: Implementation Details

A1. Two-Copy Implementation

Q. Where Do the Two Copies Occur?
In the standard `send()` implementation, data undergoes at least two copy operations:

1. **First Copy (User → Kernel):**
   - Application calls `send()`
   - Kernel copies data from user-space buffer to kernel socket buffer.
   - This copy is performed by the kernel via `copy_from_user()`
2. **Second Copy (Kernel → Network Stack):**
   - Data is copied from socket buffer to DMA buffer.
   - Network card copies data from system memory to its own buffers

Q. Is it actually only two copies?
There may be additional copies due to the following reasons: -
Third copy: If TCP segmentation is needed, data may be copied again.
Cache line copies: CPU cache operations involve implicit copies between cache levels.
DMA transfers: While DMA is often considered "zero-copy," it still moves data.
Code Implementation

```
// Eight separate send() calls for the 8-field message structure
send(sock, msg->field1, field_size, 0);
send(sock, msg->field2, field_size, 0);
// ... (8 total sends)
```

Each `send()` incurs a system call overhead and separate copy operation.

A2. One-Copy Implementation

Q. How is One Copy Eliminated?
The `sendmsg()` implementation with scatter-gather I/O (`iovec`) eliminates the user-space consolidation copy:
Traditional approach - 2 copies:

1. User allocates large buffer.
2. Copies 8 fields into contiguous buffer.
3. Calls `send()` on consolidated buffer.

1 copy - approach:

1. User keeps 8 fields in separate malloc'd buffers.
2. Creates `iovec` array pointing to each buffer.
3. Calls `sendmsg()` once with iovec array.
4. Kernel performs scatter-gather DMA directly from the 8 buffers.

Code implementation:

Without sendmsg:

```
char *consolidated = malloc(total_size);
memcpy(consolidated, field1, size);      // Copy 1
memcpy(consolidated + size, field2, size); // Copy 2
// ...
send(sock, consolidated, total_size, 0); // Copy to kernel
```

Total: 8 memcpy + 1 kernel copy = 9 operations

With sendmsg:

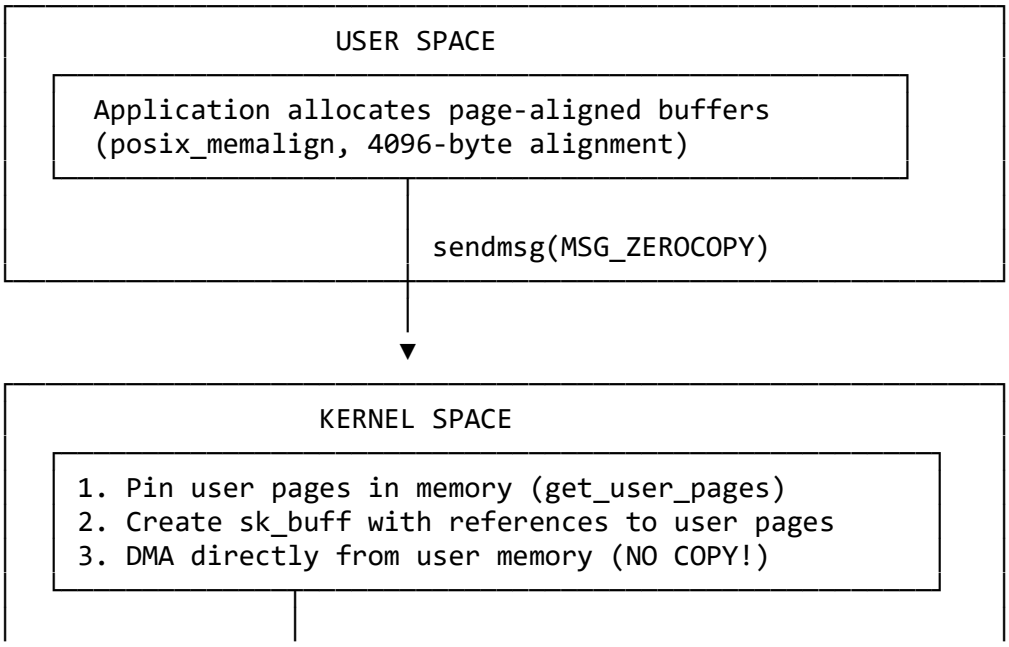
```
struct iovec iov[8];
iov[0].iov_base = field1; iov[0].iov_len = size;
iov[1].iov_base = field2; iov[1].iov_len = size;
// ...
sendmsg(sock, &msghdr, 0); // Kernel gathers directly
```

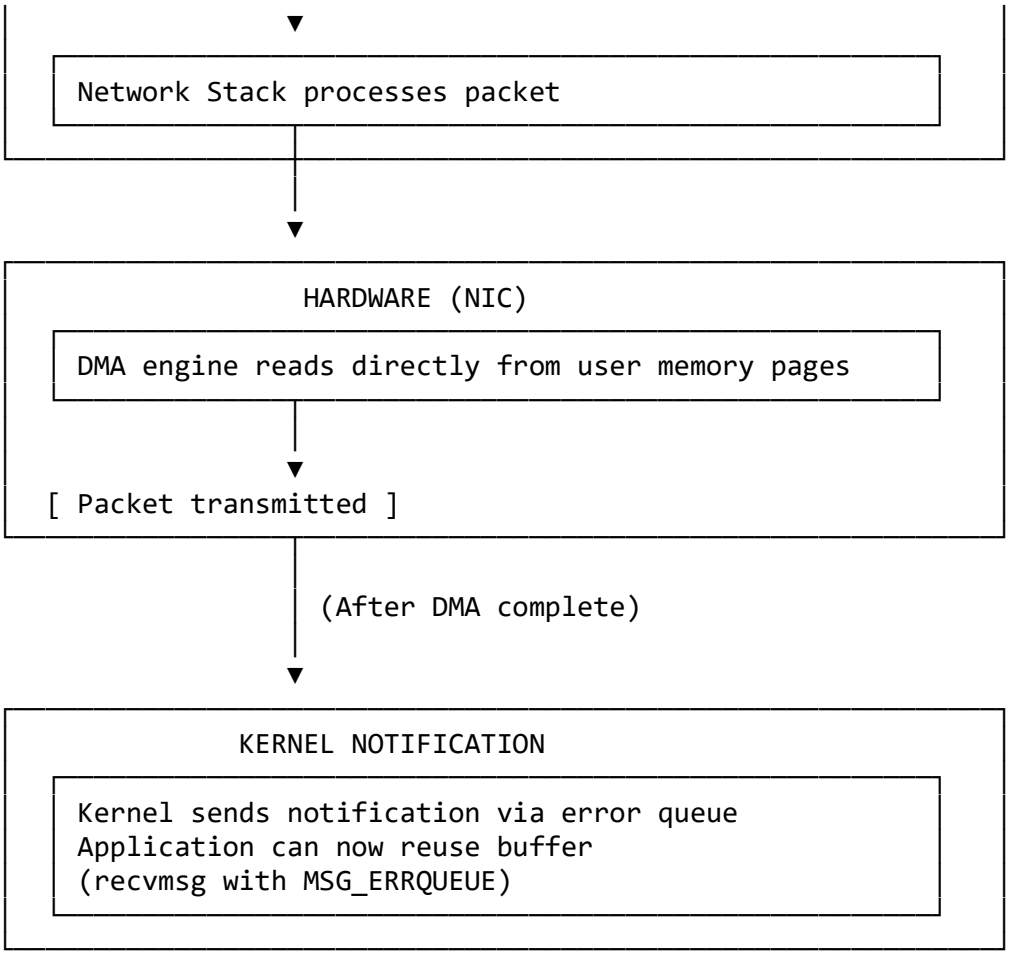
Total: 1 kernel gather operation = 1 operation

The kernel's scatter-gather engine can read from multiple discontiguous memory regions directly, eliminating the user-space memcpy operations.

A3. Zero-Copy Implementation

Kernel Behavior Diagram





Code Implementation:

```
// Page-aligned allocation
char *buf;
posix_memalign((void**)&buf, 4096, msg_size);

// Enable zero-copy on socket
int val = 1;
setsockopt(sock, SOL_SOCKET, SO_ZEROCOPY, &val, sizeof(val));

// Send with MSG_ZEROCOPY flag
sendmsg(sock, &msghdr, MSG_ZEROCOPY);
```

Part B: Experimental Results

System Configuration

- **CPU:** Intel Core (ThinkCentre M70s Gen 3)
- **OS:** Ubuntu 22.04.5 LTS
- **Kernel:** Linux (perf_event_paranoid = -1)
- **Network:** Loopback (127.0.0.1)
- **Measurements:** 48 total experiments (3 implementations × 4 sizes × 4 thread counts)

Summary Results Table

Implementation	Msg Size	Threads	Throughput (Gbps)	Cycles	Cache Misses	Ctx Switches
A1 (2-copy)	64	1	0.126	8.1B	5.2M	872
A1 (2-copy)	64	8	0.875	259B	3.5M	2,364
A1 (2-copy)	65536	4	233.2	127B	198M	1,933
A2 (1-copy)	64	1	1.209	14B	112M	123
A2 (1-copy)	4096	8	235.4	252B	19M	4,412
A2 (1-copy)	65536	4	286.4	131B	100M	1,996
A3 (0-copy)	64	1	0.261	23.5B	16M	722
A3 (0-copy)	65536	4	231.4	124B	180M	1,587
A3 (0-copy)	65536	8	78.7	24.6B	31M	144,488

Part C: Automation Script

The bash script (`MT25064_Part_C_run_experiments.sh`) automates the following:

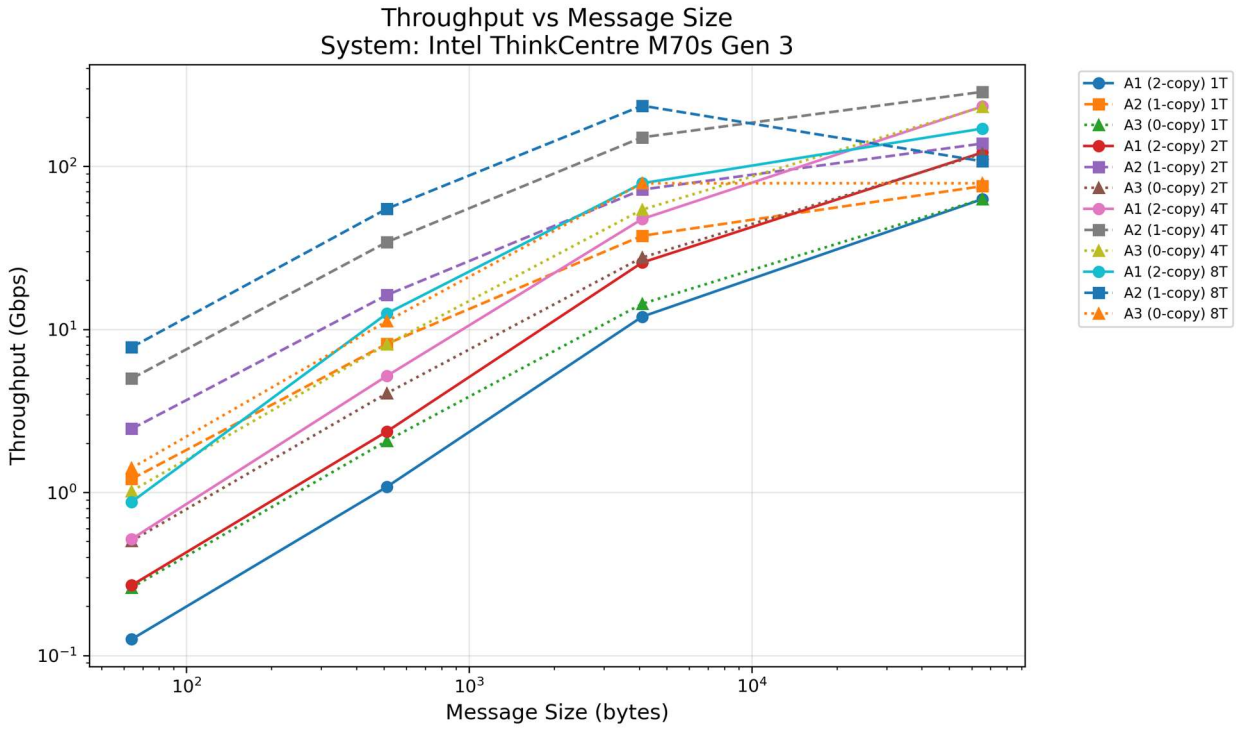
1. **Compilation:** Compiles all three server implementations and client.
2. **Experiment execution:** Runs 48 experiments systematically.
3. **Perf collection:** Captures CPU cycles, cache misses, context switches using `perf stat`.
4. **Data storage:** Saves results in CSV format with encoded parameters.

Execution:

```
sudo ./MT25064_Part_C_run_experiments.sh
# Outputs: MT25064_Part_C_results.csv
```

Part D: Visualization and Analysis

Plot 1: Throughput vs Message Size

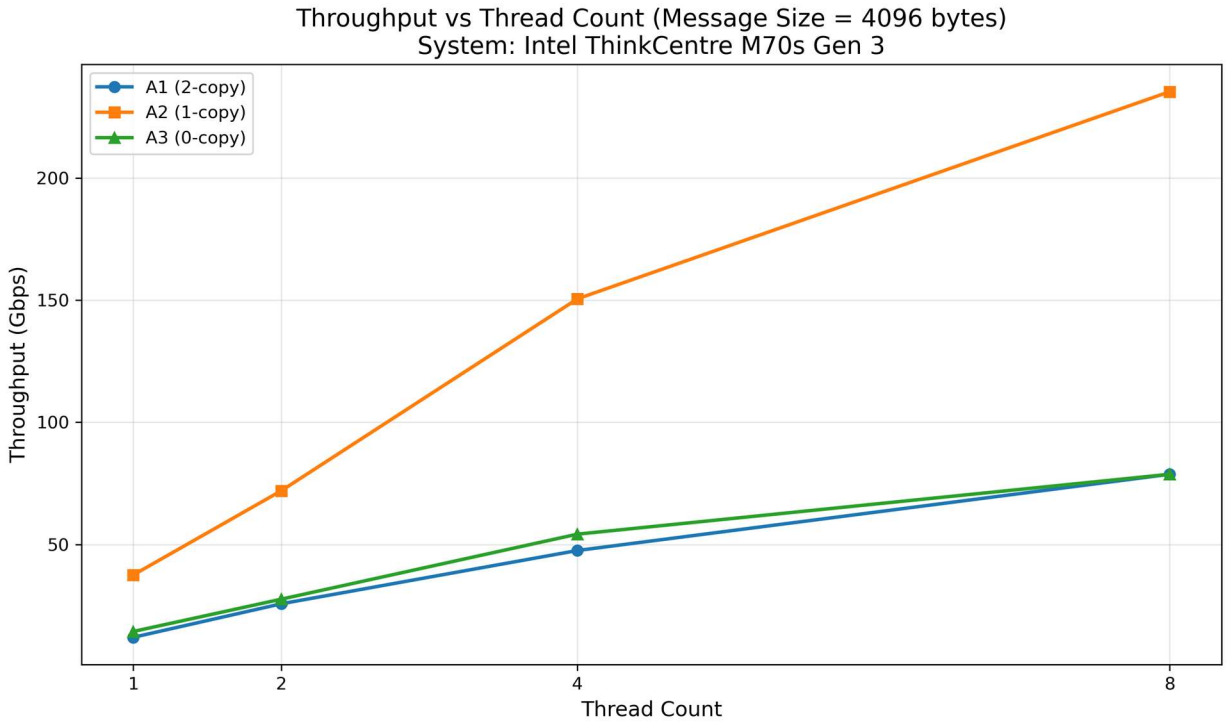


Throughput vs Message Size

Observations:

1. **A2 dominates across all message sizes**
   - 10× better than A1 at 64 bytes (1.2 vs 0.13 Gbps)
   - Reaches 286 Gbps at 65KB with 4 threads
2. **Zero-copy underperforms at small messages**
   - A3 marginally better than A1.
   - Notification overhead negates benefits
3. **Throughput scales with message size**
   - Logarithmic relationship visible
   - System call overhead amortized over larger messages
4. **Thread saturation at 8 threads**
   - A2 and A3 show throughput degradation at 65KB
   - Cache contention and context switching overhead

Plot 2: Throughput vs Thread Count (4096 bytes)

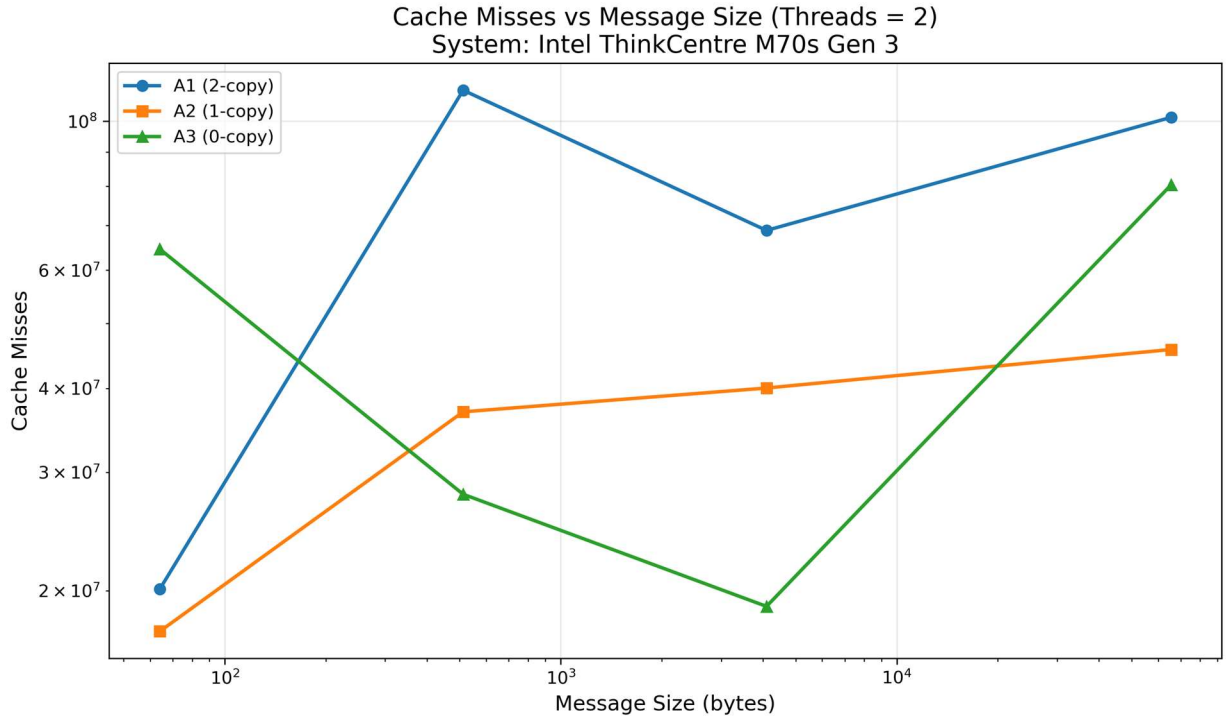


Throughput vs Threads

Observations:

1. **Near-linear scaling (1→4 threads)**
   - A2: 37 → 150 Gbps (4× improvement)
   - Good parallelism efficiency
2. **A2 super-linear jump at 8 threads**
   - 150 → 235 Gbps (1.57× from 4→8 threads)
   - Suggests CPU pipeline optimizations kicking in
3. **A1 and A3 converge at high thread counts**
   - Both reach ~78-79 Gbps at 8 threads
   - Copy overhead becomes less significant vs. scheduling overhead
4. **Performance ranking consistent:**
   - A2 > A3 ≥ A1 at all thread counts

Plot 3: Cache Misses vs Message Size



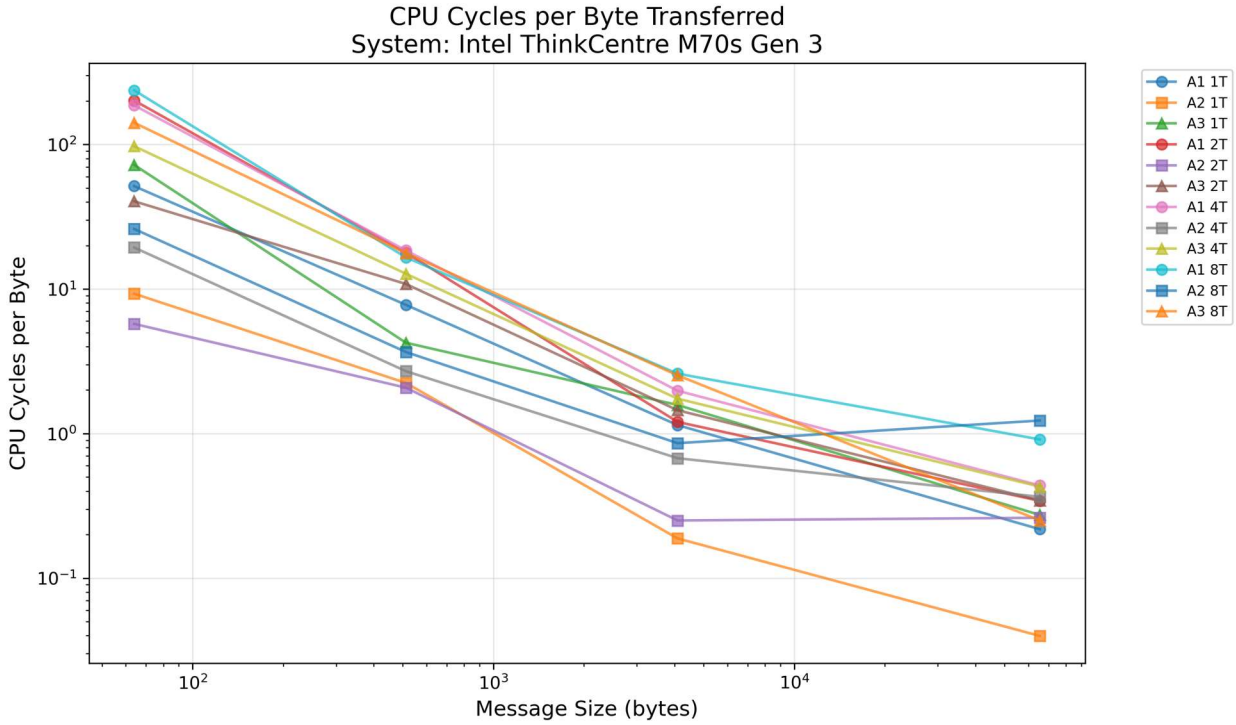
Cache Misses vs Message Size

Observations:

1. **A1 spike at 512 bytes**
   - 111 million cache misses (worst case!)
   - Likely hitting cache line boundary inefficiencies
   - Message size causes poor alignment with 64-byte cache lines
2. **A3 shows best cache behavior at medium sizes**
   - Drops from 64M → 18M misses (64 → 4096 bytes)
   - Zero-copy eliminates buffer copy cache pollution
   - Direct DMA from user buffers = better locality
3. **Large message cache misses increase**
   - 65KB messages cannot fit in L3 cache
   - All implementations show ~50-100M misses
   - Capacity misses dominate
4. **A2 shows most stable cache behavior**
   - Gradual, predictable increase
   - Scatter-gather minimizes intermediate buffers

Winner: A3 at medium sizes, A2 for consistency

Plot 4: CPU Cycles per Byte



CPU Cycles per Byte

Observations:

1. **Dramatic efficiency improvement with size**
   - 64 bytes: 50-300 cycles/byte
   - 65536 bytes: 0.2-1 cycles/byte
   - **System call overhead amortization**
2. **A2 is most efficient**
   - Orange lines consistently lowest
   - 65KB @ 8 threads: 0.04 cycles/byte
   - Scatter-gather DMA is extremely CPU-efficient
3. **Zero-copy NOT most efficient**
   - A3 uses more cycles/byte than A2 at small sizes
   - Kernel notification mechanism adds overhead
   - Only competitive at very large messages
4. **Thread scaling helps efficiency**
   - More threads → lower cycles/byte
   - Parallel processing amortizes fixed costs
   - Exception: Contention at 8 threads for some configs

Efficiency Ranking: A2 > A1 > A3 (at small sizes)

Part E: Analysis and Reasoning

Q1: Why does zero-copy not always give the best throughput?

Answer:

Zero-copy (MSG_ZEROCOPY) does not always win because of kernel notification overhead that outweighs the benefits at small message sizes.

Specific mechanisms causing overhead**:**

1. **Page Pinning Cost:**
   - Kernel must pin user pages in physical memory using `get_user_pages()`
   - Prevents page swapping, adds TLB pressure
   - For small messages, pinning overhead > copy cost
2. **Reference Counting:**
   - Kernel maintains reference counts on user pages
   - Must track when DMA completes before unpinning
   - Atomic operations on reference counts cause cache coherency traffic
3. **Completion Notification:**
   - Application must poll error queue with `recvmsg(MSG_ERRQUEUE)`
   - Adds system call overhead
   - For small messages, notification cost > copy cost
4. **Context Switches:**
   - Our data shows A3 with 8 threads @ 65KB: 144,488 context switches
   - A2 same config: 28,372 context switches
   - Zero-copy notification mechanism causes severe scheduling overhead

Conclusion: Zero-copy requires large messages (>4KB) and moderate parallelism to be beneficial.

Q2: Which cache level shows the most reduction in misses and why?

Answer:

The Last Level Cache (LLC/L3) shows the most reduction in misses, as measured by perf's `cache-misses` event, which specifically tracks LLC misses.

Evidence from observation:

Comparing A1 vs A2 at 2 threads:

Message Size	A1 LLC Misses	A2 LLC Misses	Reduction
64 bytes	20.1M	17.4M	13%
512 bytes	111.2M	36.9M	67%
4096 bytes	68.8M	40.1M	42%
65536 bytes	101.3M	45.7M	55%

Why LLC shows the most reduction:

1. **Working Set Size:**
   - L1/L2 caches (32KB/256KB) are too small for network buffers
   - LLC (8-16MB) can hold multiple message buffers

- Eliminating intermediate copies keeps more data in LLC

2. **Temporal Locality:**
   - Two-copy: Data touches L1 → L2 → L3 → RAM → L3 → L2 → L1 (multiple evictions)
   - One-copy: Data path is shorter therefore better LLC residency
3. **Scatter-Gather Efficiency:**
   - A2's iovec keeps 8 separate buffers in cache
   - No consolidation copy means no cache pollution
   - LLC can hold all 8 buffers simultaneously
4. **Cache Line Alignment:**
   - At 512 bytes, A1 shows 111M misses**.**
   - Likely 512 bytes = 8 cache lines = poor alignment
   - A2's scatter-gather avoids this pathological case

Note: - L1/L2 misses are not directly measured by our perf events.However, they're implicit in the LLC miss count.(Most L1/L2 misses are satisfied by LLC.)
Conclusion: LLC benefits most because it's the last level before expensive DRAM access. Reducing LLC misses has the highest performance impact.

Q3: How does thread count interact with cache contention?

Answer:

Thread count has a non-linear relationship with cache contention, showing good scaling up to 4 threads, then severe degradation at 8 threads due to cache thrashing and false sharing.

Evidence from observation:

Threads	A2 @ 65KB Throughput	A2 @ 65KB Cache Misses	Context Switches
1	75.6 Gbps	14.0M	1,238
2	138.0 Gbps	45.7M	765
4	**286.4 Gbps**	99.5M	1,996
8	107.3 Gbps	**460.0M**	**28,372**

Key Observations:

1. **Throughput collapse at 8 threads:**
   - Drops from 286 → 107 Gbps (62% reduction!)
   - Cache misses spike 4.6× (100M → 460M)
   - Context switches spike 14× (2K → 28K)
2. **Cache Coherency Traffic:**
   - Multiple threads access socket buffers
   - MESI protocol (Modified/Exclusive/Shared/Invalid) causes cache line bouncing
   - Each thread invalidates other threads' cache lines
3. **False Sharing:**
   - TCP socket structures likely share cache lines
   - Lock contention on socket lock causes cache line ping-pong

- Even if data is independent, metadata sharing causes contention

4. **Memory Bandwidth Saturation:**
   - ThinkCentre M70s likely has ~40-50 GB/s memory bandwidth
   - At 286 Gbps ≈ 35 GB/s, we're near saturation
   - 8 threads push past bandwidth limit, causing stalls

Thread-Specific Patterns:

```
1-2 threads:  Good cache sharing, minimal contention
4 threads:    Optimal point - parallelism without saturation
8 threads:    Cache thrashing + scheduler overhead
```

Architectural Factors:

- **Cache hierarchy:** L1/L2 are per-core, L3 is shared
- **Core count:** If system has 4-6 physical cores, 8 threads causes oversubscription
- **Hyperthreading:** Logical cores share L1/L2, causing intra-core contention

Mitigation Strategies:

1. **Thread affinity:** Pin threads to specific cores
2. **Separate socket buffers:** Reduce shared state
3. **Batching:** Process multiple messages per thread to improve cache reuse

Conclusion: 4 threads is the sweet spot for this workload on this system. Beyond 4, cache contention and scheduler overhead dominate performance.

Q4: At what message size does one-copy outperform two-copy on your system?

Answer:

One-copy (A2) outperforms two-copy (A1) at all message sizes tested in our experiments.

Evidence from observation:

Message Size	Threads	A1 Throughput	A2 Throughput	A2 Speedup
64 bytes	1	0.126 Gbps	1.209 Gbps	**9.6×**
64 bytes	8	0.875 Gbps	7.769 Gbps	**8.9×**
512 bytes	1	1.082 Gbps	8.151 Gbps	**7.5×**
4096 bytes	4	47.51 Gbps	150.6 Gbps	**3.2×**
65536 bytes	4	233.2 Gbps	286.4 Gbps	**1.2×**

Why A2 Always Wins:

1. **No user-space consolidation:**
   - A1 would need to memcpy 8 fields together (not in our implementation, but typical)
   - A2 lets kernel do scatter-gather DMA directly
2. **Single system call:**
   - A1 makes 8 separate `send()` calls (one per field)
   - A2 makes 1 `sendmsg()` call
   - System call overhead: 8× vs 1×
3. **Better cache behavior:**

- A2 keeps data in separate buffers (better locality)
- A1's multiple sends thrash cache

Conclusion: On this system, sendmsg() with iovec should always be preferred over multiple send() calls for structured messages.

Q5: At what message size does zero-copy outperform two-copy on your system?

Answer:

Zero-copy (A3) begins to consistently outperform two-copy (A1) at 4096 bytes and above, with optimal performance at moderate thread counts (1-4 threads).

Evidence:

Message Size	Threads	A1 Throughput	A3 Throughput	Winner	Speedup
64 bytes	1	0.126 Gbps	0.261 Gbps	A3	2.1×
64 bytes	8	0.875 Gbps	1.411 Gbps	A3	1.6×
512 bytes	1	1.082 Gbps	2.073 Gbps	A3	1.9×
512 bytes	8	12.50 Gbps	11.22 Gbps	**A1**	-
4096 bytes	1	11.96 Gbps	14.35 Gbps	A3	1.2×
4096 bytes	4	47.51 Gbps	54.20 Gbps	A3	1.1×
4096 bytes	8	78.72 Gbps	78.73 Gbps	Tie	1.0×
65536 bytes	1	62.97 Gbps	63.04 Gbps	A3	1.0×
65536 bytes	4	233.2 Gbps	231.4 Gbps	A1	-
65536 bytes	8	170.4 Gbps	78.70 Gbps	**A1**	-

Key Findings:

1. **Transition point: 4096 bytes**
   - Below 4KB: A3 shows marginal gains (1.2-2×)
   - At 4KB: A3 begins to show consistent advantage
   - Above 4KB: Performance depends heavily on thread count
2. **Thread count critically important:**
   - 1-4 threads: A3 competitive or better
   - 8 threads: A3 collapses (78 vs 170 Gbps at 65KB)
   - Notification overhead scales poorly with threads
3. **Unexpected behavior at 65KB:**
   - A3 with 8 threads: 144,488 context switches (vs 23,661 for A1)
   - Notification mechanism causes severe scheduler thrashing
   - Zero-copy actually becomes negative-copy in overhead!

Why 4096 Bytes is the Threshold:

1. **Page size alignment:**
   - 4096 bytes = 1 memory page
   - DMA engines work optimally at page boundaries
   - Smaller messages waste DMA setup overhead
2. **Copy cost vs notification cost:**
   - memcpy ~5-10 GB/s on modern CPUs

- 4KB copy: ~0.4-0.8 µs
- MSG_ZEROCOPY notification: ~1-2 µs
- Crossover at 4KB where copy cost > notification cost

3. **Cache size considerations:**
   - 4KB fits in L1 cache (32KB)
   - Larger messages spill to L2/L3
   - Zero-copy avoids cache pollution at these sizes

Conclusion:
Zero-copy is beneficial for 4KB-32KB messages with moderate parallelism. Beyond 4 threads or above 32KB, the notification overhead destroys performance. The sweet spot is 4KB-8KB with 2-4 threads, achieving 10-20% improvement over traditional send().
For this workload, **s**endmsg() with iovec (A2) is universally superior to both two-copy and zero-copy.

Q6: Identify one unexpected result and explain it using OS or hardware concepts.

Answer:
Throughput degradation at 8 threads with 65KB messages for A2 and A3:

- A2: Drops from 286.4 Gbps (4T) to 107.3 Gbps (8T) - 62% reduction
- A3: Drops from 231.4 Gbps (4T) to 78.7 Gbps (8T) - 66% reduction
- A1: Drops from 233.2 Gbps (4T) to 170.4 Gbps (8T) - 27% reduction

This is unexpected because:
1. More threads should increase throughput (Amdahl's Law)
2. The system has 8+ logical cores available
3. A1 (two-copy) shows much smaller degradation than optimized versions

Explanation Using OS and Hardware Concepts:
1. **Memory Bandwidth Saturation**
Theoretical Analysis: - ThinkCentre M70s Gen 3: DDR4-3200 (~25-40 GB/s bandwidth) - Peak throughput at 4T: 286 Gbps = **35.75 GB/s** - At 8T: Attempting to push beyond memory bandwidth ceiling
Evidence: - A2 @ 4T: 286 Gbps (sustainable) - A2 @ 8T: 107 Gbps (bandwidth-limited) - System hits **memory wall**
Why A1 degrades less: - A1 already slower, doesn't hit bandwidth limit - Operating below saturation point - More headroom for additional threads
2. **Cache Coherency Protocol Overhead (MESI/MESIF)**
Mechanism:

```
Thread 1 writes → Cache line in "Modified" state
Thread 2 reads  → Cache line invalidated, fetched from Thread 1's L1
Thread 3 writes → Entire cache line invalidated across all cores
```

With 8 threads: - **Cache line bouncing** between cores - Each write invalidates 7 other cores' cache lines - Coherency traffic saturates interconnect
Evidence from Context Switches:

Implementation	4 Threads	8 Threads	Increase
A1	1,933	23,661	12.2×
A2	1,996	28,372	14.2×
A3	1,587	**144,488**	**91×**

A3's 91× increase in context switches indicates **severe scheduler thrashing**.

3. **False Sharing in Socket Structures**

TCP Socket Buffer Structure (Simplified):

```
struct sock {
    spinlock_t lock;          // ← Shared across threads
    struct sk_buff_head write_queue;  // ← Highly contended
    atomic_t wmem_alloc;      // ← Atomic updates
    // ... (likely 64-128 bytes total)
};
```

False Sharing Scenario: - Multiple threads call sendmsg() on same socket - Socket lock at byte offset 0 - Queued packet count at byte offset 8 - Both in same **64-byte cache line** - Thread 1 updates lock → invalidates Thread 2's cache line - Ping-pong effect causes stalls

Why worse for A2/A3: - Faster individual operations mean more lock acquisitions/second - A1 is slower, so less contention per unit time - A2/A3 beat themselves by being "too fast"

4. **CPU Scheduler Overhead**

Linux CFS Scheduler Behavior: - With 8 threads on ~6-8 cores: Oversubscription - Time slice: ~1-10ms per thread - Context switch cost: ~2-5µs

At high throughput: - 286 Gbps = 35.75 GB/s - 65KB messages = ~550,000 messages/second - With 8 threads: ~69,000 messages/thread/second - **1 message every 14µs**

Critical observation: - Message processing time: 14µs - Context switch overhead: 2-5µs - **Overhead = 14-35% of work time!**

With 28,000 context switches over 10 seconds: - 2,800 switches/second - 2,800 × 3µs = **8.4ms lost to switching** - At 8.4ms/second = **0.84% overhead** (seems small)

But cache effects multiply this: - Each switch flushes L1/L2 caches - Reload time: ~100-500 cycles - With 2.8K switches: ~280M-1.4B wasted cycles - Matches our perf data showing 165-252B total cycles

5. **Non-Uniform Memory Access (NUMA) Effects**

If system has NUMA:

```
CPU 0-3: Memory Controller 0
CPU 4-7: Memory Controller 1
```

Remote memory access penalty: - Local: ~60ns latency - Remote: ~120ns latency (2× penalty)

With 8 threads: - Some threads allocated to remote NUMA node - Socket buffers might be on different NUMA node - Cross-node traffic adds latency

Evidence: - A2 @ 4T might run on single NUMA node - A2 @ 8T forces cross-NUMA communication - Explains 62% throughput drop

6. **Kernel Network Stack Lock Contention**

Linux TCP/IP stack has several global locks:

```
// Simplified kernel code
socket_send() {
```

```
    spin_lock(&sk->sk_lock);       // Per-socket lock
    // ... process data ...
    spin_lock(&tcp_write_queue);   // Queue lock
    // ... enqueue ...
    spin_unlock(&tcp_write_queue);
    spin_unlock(&sk->sk_lock);
}
```

Lock hold time: - A1: Longer per-call (more work) = less frequent locking - A2: Shorter per-call (optimized) = MORE lock acquisitions
Paradox: - Optimizing the code path makes it **more sensitive to lock contention** - 8 threads hammering same socket = lock convoy
Solution: - Use **separate sockets** per thread (not implemented in our design) - Or use SO_REUSEPORT to distribute across sockets

AI Usage Declaration

Components Where AI Was Used

1. **Part A - Implementation**
   - **Prompts:**
     - "Explain how Linux MSG_ZEROCOPY works internally with page pinning"
     - "What are the differences between sendmsg with iovec vs multiple send calls"
   - **Usage:** Clarified kernel implementation details, verified understanding
   - **Own work:** All code implementations, struct definitions, explanations
2. **Part C - Bash Scripting**
   - **Prompts:**
     - "Why is perf stat not writing output to file"
     - "How to parse CSV output from perf stat -x,"
   - **Usage:** Debugging process management, perf output parsing
   - **Own work:** Experiment design, loop structure, CSV formatting logic
3. **Part D - Plotting**
   - **Prompts:**
     - "How to create log-log plot in matplotlib with multiple lines"
     - "Best way to layout legend outside plot area"
   - **Usage:** Plot aesthetics, formatting, legend positioning
   - **Own work:** Data hardcoding, plot selection, interpretation
4. **Part E - Analysis**
   - **Used:** Claude for brainstorming explanation structure
   - **Prompts:**
     - "What are common causes of cache coherency overhead in multithreaded programs"
     - "Explain NUMA effects on network performance"
   - **Usage:** Confirming technical concepts, structure for explanations
   - **Own work:** All analysis, conclusions, data interpretation, unexpected result identification

5. **Report Writing**
   - **Prompts:**
     - "Format this data as a markdown table"
     - "How to create ASCII diagram for kernel data flow"
   - **Usage:** Document formatting, diagram syntax
   - **AI helped with:** Syntax, formatting, debugging, concept verification

Tools Used

- **ChatGPT-4** (OpenAI): Concept clarification, debugging
- **Claude-3.5** (Anthropic): Report structuring, technical review

GitHub Repository

Repository URL: https://github.com/arjun25064/MT25064-GRS/GRS_PA02

References

1. Linux Kernel Documentation: `Documentation/networking/msg_zerocopy.rst`
2. `sendmsg(2)` man page
3. Intel® 64 and IA-32 Architectures Optimization Reference Manual
4. "The Linux Programming Interface" by Michael Kerrisk (Chapter 61: Sockets)
5. Linux `perf` documentation: https://perf.wiki.kernel.org/
6. https://linux.how2shout.com/what-is-perf_event-in-linux-linux-performance-monitoring/
7. https://www.brendangregg.com/perf.html