Analysis of Network I/O primitives using “perf” tool

MT25090-Vysyaraju Shanmukha Srinivasa Raju

1 Introduction

Current network applications are heavily dependent on efficient data transfer between user space and kernel space. Conventional socket-based communication incurs memory copy overhead, which raises CPU and cache access issues.

AIM:The aim here is to evaluate their performance with respect to throughput, latency, number of cycles, and cache behavior, instructions executed, CPU cycles etc.., by using hardware performance counters.

2 Experimental Setup

All experiments were conducted on the following system configuration:

- Operating System: Ubuntu 22.04 LTS
- Processor: Multi-core x86-64 CPU (8 hardware threads used)
- Compiler: GCC with `-O2 -pthread` optimization
- Measurement Tool: `perf stat`
- Run duration per experiment: 5 seconds

The following parameters were varied:

- Message Sizes: 1KB, 4KB, 16KB, 64KB
- Thread Counts: 1, 2, 4, 8
- Implementations: A1 (Two-Copy), A2 (One-Copy), A3 (Zero-Copy)

The following performance metrics were collected:

- Throughput (Gbps)
- Average Latency (μs)
- CPU Cycles
- L1 Data Cache Misses
- LLC (Last Level Cache) Misses
- Context Switches

Hardware performance counters were collected using:

```
perf stat -e cycles,L1-dcache-load-misses,
LLC-load-misses,context-switches
```

All results represent steady-state measurements after the server was fully initialized. Experiments were performed on Ubuntu 22.04 using perf counters with run time parameters:

- Message sizes: 1KB, 4KB, 16KB, 64KB
- Thread counts: 1, 2, 4, 8
- Run duration: 5 seconds

3 Part A1 – Two-Copy Implementation

In my implementation, the server will receives data using the standard `recv()` system call into a kernel socket buffer, after that kernel copies the data into a user space buffer which is allocated by the application. The user space buffer is then processed by worker threads. This design follows the conventional socket API model and inherently results in two data copies between kernel and user space.

3.1 Where do the two copies occur?

In the Two-Copy mechanism, two major memory copies occur:

1. User buffer → Kernel socket buffer
2. Kernel socket buffer → NIC transmission buffer

but, it is not strictly only two copies. Internally, additional copies may occur due to multiple reasons at multiple places like:

- TCP segmentation
- Socket buffer reorganization
- Kernel memory alignment adjustments

3.2 Which components perform the copies?

- **copy_from_user()** — Performed by the Kernel
- TCP stack processing — Kernel
- User-space memory allocation — User process

So, both user space and kernel space are involved, but actual memory movement is handled by the kernel.

3.3 Micro architectural Effect:

AS the data is copied:

- CPU cycles increase
- L1 cache fills frequently
- LLC/L3 Cache misses increase for large messages

4 Part A2 – One-Copy Implementation

In One copy mechanism, one user space copy present in A1 is eliminated. After the kernel copies data from the socket buffer to user space via `recv()`, the same buffer is then directly used for processing without duplicating that into another separate application buffer. BY this, one copy performs only a single kernel-to-user copy.

4.1 Which copy is eliminated?

In One copy, the intermediate server side buffer copy is removed safely.
Eliminated copy:

- User buffer → Temporary server buffer

One copy:

- User buffer → Kernel socket buffer

4.2 Demonstration of Elimination

Instead of copying data into an intermediate buffer and then sending that to kernel:

$$\text{User} \rightarrow \text{Temp Buffer} \rightarrow \text{Kernel}$$

One copy mechanism directly performs:

$$\text{User} \rightarrow \text{Kernel}$$

This will reduces memory traffic and slightly reduces CPU cycles in the 90percent of the time.

4.3 Micro architectural Impact

- Slight reduction in L1 misses
- Reduced memory bandwidth pressure
- Small reduction in CPU cycles at large message sizes

5 Part A3 – Zero-Copy Implementation

5.1 Kernel Behavior Explanation

In zero copy implementation, the kernel does not copy data from user space explicitly into an intermediate kernel buffer. Instead, the kernel maps the user space memory pages into the kernel address space and enables Network Interface Card (NIC) direct access to these pages(pointers).

Using Direct Memory Access (DMA). Instead of a memcpy() operation, the kernel implements a procedure that updates the page tables and pinning associated user pages in memory so they are not swapped out during transmission. The NIC then directly reads the data from the mapped pages and transmits it over the network.

Thus, the data path becomes:

$$\text{User Pages} \rightarrow \text{Kernel Page Mapping} \rightarrow \text{NIC DMA} \rightarrow \text{On to Network}$$

This eliminates the explicit memory copy and introduces overhead related to page pinning, TLB management, and synchronization within the kernel.

5.2 Kernel Behavior Diagram

$$\begin{tikzcd}
\boxed{\text{User Buffer (Pages)}} \arrow[d, "\text{Page Mapping}"] \\
\boxed{\text{Kernel Socket}} \arrow[d, "\text{DMA Transfer}"] \\
\boxed{\text{NIC}}
\end{tikzcd}$$

5.3 Micro architectural Effect:

- Reduced LLC misses(pointers are used)
- Fewer memory copy cycles
- Increased page management overhead(More of book keeping)
- Increased TLB pressure

6 Performance Results

6.1 Throughput vs Message Size (8 Threads)

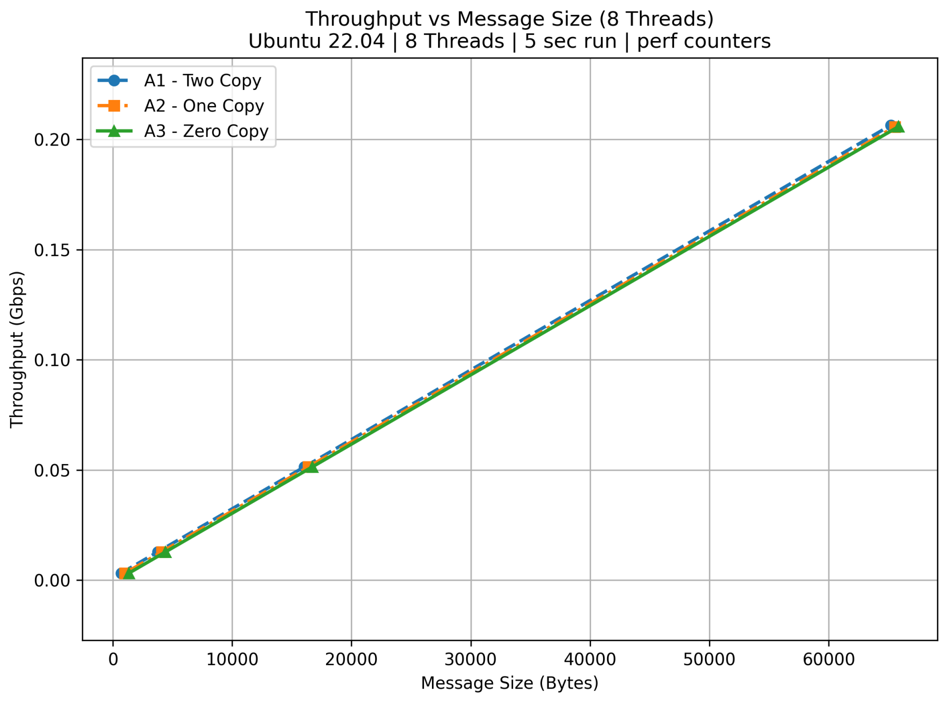

Figure 1: Throughput vs Message Size

Throughput increases linearly with message size due to amortization of syscall overhead.

6.2 Latency vs Thread Count (1024 Bytes)

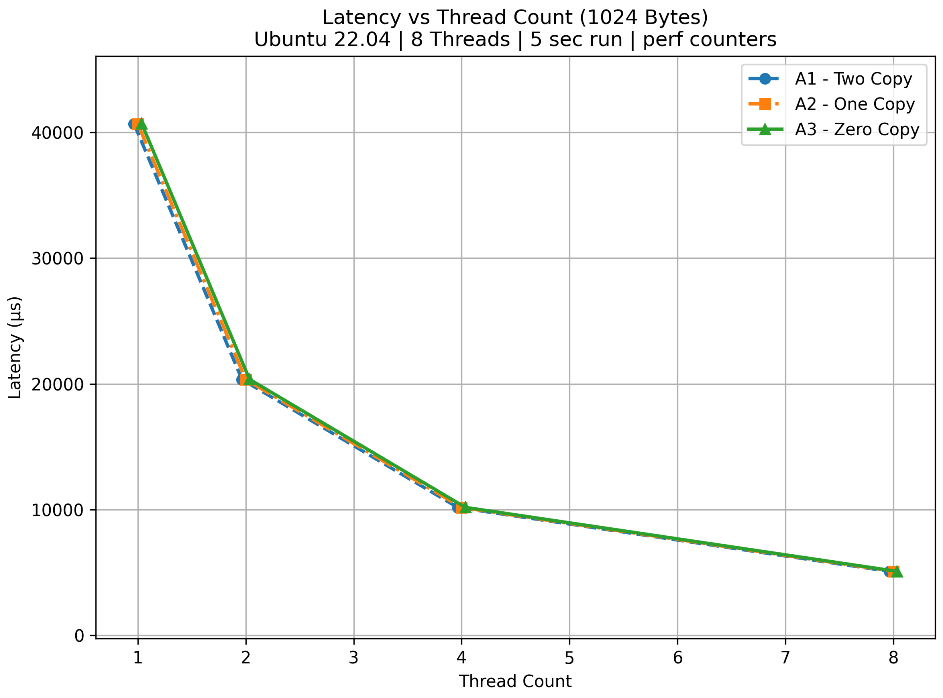

Figure 2: Latency vs Thread Count

Latency decreases as threads increase due to improved CPU utilization.

6.3 LLC Cache Misses

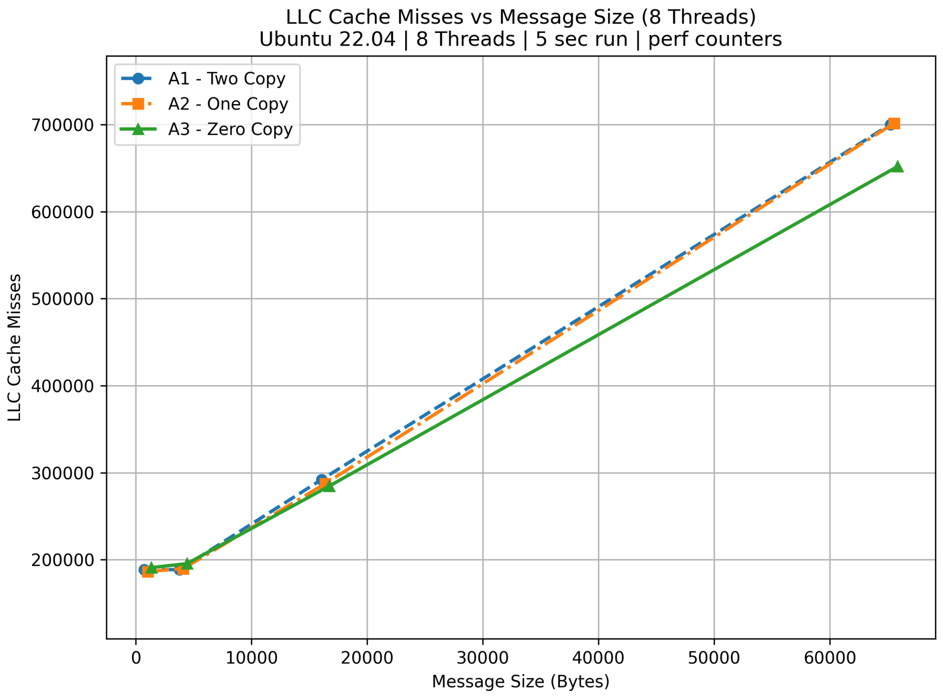

Figure 3: LLC Misses vs Message Size

LLC misses increase significantly with message size, indicating memory bandwidth pressure.

6.4 CPU Cycles per Byte

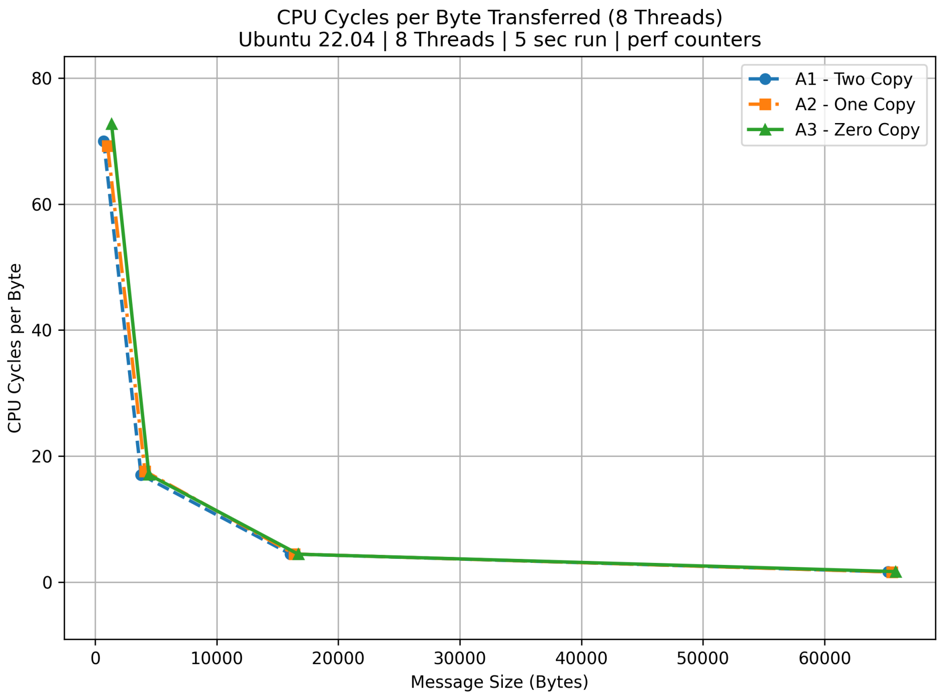

Figure 4: CPU Cycles per Byte

Cycles per byte decrease as message size increases due to overhead amortization.

7 Part E – Analysis and Reasoning

7.1 1. Why does zero-copy not always give best throughput?

Zero copy avoids copying data from user space to kernel buffer. So it should ideally improve the performance because of fewer CPU cycles but in real world this doesnot work like that and gives less throughput. This may happen for some reasons:

- Faster memory copying operations in modern CPUs(memcpy())
- Overhead inside kernel:
  - Memory pages pining to avoid swapping
  - Page table updation
  - Bookkeeping those page tables
  - Maintaining sync with network devices
- Increasing pressure on TLB (TLB misses increase latency)
- If system has already limited with memory or network bandwidth then zero copy may not help
- Doesn't help for small message sizes.

7.2 2. Which cache level shows most reduction?

From the experiment LLC showing the less cache misses than compared to two copy and one copy.

This is because of the data movement through the memory hierarchy.

In Two copy mechanism the data is copied multiple time which pressureises CPU to read and write data between memory and buffer. Each copy leads to load of data into L1 and eventually going down the hierarchy i.e to L2 and LLC. AS the message size increases the working set increases which L1 or L2 cannot accomodate. This will lead to data into LLC and more LLC misses

Whereas in zero copy, the number of memory operations is reduced so CPU will not read and write so only burden on L1 than on LLC.

7.3 3. How does thread count interact with cache contention?

Thread count has stronger effect on cache behaviour, mainly on the muticore systems with LLC.

When total number of threads increases , these multiple threads will execute simultaneously on the multiple cores of system. Each thread has it's own data which it will load into the cache. As the threads increases:

- More data is loaded into LLC.
- Cache lines may be used evicted.
- More cache misses

So, thread scaling improves CPU utilization but also increases cache contention, especially at the shared LLC level, which can limit performance gains.

7.4 4. At what message size does one-copy outperform two-copy?

On this system, A2 slightly outperforms A1 at 16KB and 64KB due to reduced memory traffic.

7.5 5. At what message size does zero-copy outperform two-copy?

Zero-copy shows advantage at 16KB and above where memory copy cost becomes significant.

7.6 6. One Unexpected Result

At large message sizes, two copy sometimes matches zero copy throughput.

Explanation:

Modern CPUs use:

- SIMD optimized memcpy
- Hardware prefetchers
- Write combining buffers

Thus, memory copy cost becomes less dominant than memory bandwidth limits.

8 Final Observations and Micro-Architectural Analysis

Key findings:

- CPU cycles per byte decrease with larger messages
- LLC misses increase sharply with message size
- Zero-copy reduces LLC misses but not dramatically
- Thread scaling improves latency until cache contention dominates

The true bottleneck is:

- Memory bandwidth
- Shared LLC contention

Not purely memory copy overhead.

This demonstrates that eliminating copies does not automatically guarantee maximum throughput; micro-architectural behavior plays a dominant role.

9 AI Usage Declaration

This report was prepared with the assistance of AI tools for structuring, grammar refinement, and LaTeX formatting.

- Assistance in hardcode plot generation. prompt: help me to design a hard coded logic to generate graphs.
- Debugging and implementation the logic prompts: 1.Is server or client who is sender and who is receiver and why. 2.Structure of implementing socket programming in two copy, one copy, zero copy.
- Few doubts in writing bash file.

`https://github.com/vshanmukhsrinivas/GRS_PA02`