A NUMA-Aware Memory Allocator

Devesh Mirchandani Padmnabh Tewari Kartik Sharma

August 30, 2025

1 Introduction

This document supports the presentation of our project: a NUMA-Aware Memory Allocator. The primary motivation behind this work is to solve a critical performance bottleneck that occurs in modern multi-socket servers due to the non-uniform distribution of memory access times.

The following sections expand upon the problem, our proposed solution, the testing methodology, as well as the tools and technologies that supported our implementation.

2 Problem Statement

2.1 The Hardware Context

Modern high-performance servers are often built as multi-socket systems, meaning they contain multiple physical CPUs. These CPUs follow the **NUMA (Non-Uniform Memory Access)** architecture.

In NUMA systems, each CPU owns a bank of "local" memory that it can access very quickly. Accessing memory belonging to another CPU ("remote" memory) introduces higher latency due to the signal having to cross a slower interconnect between the CPU sockets.

Analogy: Consider an office building with two towers:

- Fetching coffee from the kitchen in your own tower is fast (local memory access).
- Walking to the other tower for coffee takes much longer (remote memory access).

This delay is the "non-uniform" nature of the memory access.

2.2 The OS Allocation Policy

Operating systems like Linux typically follow a simple rule known as the **"first-touch" policy**. A physical page of RAM is not assigned to a program's virtual memory address until a CPU core first *writes* to it. At that moment, the memory is physically allocated on the NUMA node of that specific CPU.

This creates a problem in multi-threaded applications:

1. A thread running on a CPU in **Node 0** might initialize a large dataset.
2. Because of the "first-touch" rule, all that data is placed in Node 0's physical RAM.
3. Later, a different thread running on a CPU in **Node 1** needs to process that data. It now incurs constant, slow, remote memory accesses for its entire workload.

Standard memory allocation is therefore "blind" to the NUMA topology, leading to suboptimal performance.

3 Proposed Solution: The Smart Allocator

3.1 Design Philosophy

We developed a user-space library that acts as a NUMA-aware allocator. By intercepting an application's memory requests, our solution explicitly instructs the Linux kernel on where memory should be physically allocated, thereby overcoming the limitations of the default "first-touch" policy.

3.2 Implementation Methodology

The allocator is implemented as a complete replacement for the `malloc` family (`malloc`, `free`, `calloc`, `realloc`) and is injected into programs using the `LD_PRELOAD` mechanism, which allows the overriding of system functions without modifying the application's source code.

Our allocation process follows a **Direct Kernel Instruction Strategy**:

1. **Thread-Safe Initialization:** The first time any allocation function is called, a one-time, thread-safe initialization routine is triggered via `pthread_once`. This routine detects the system's NUMA topology.
2. **Select Target Node:** The allocator applies a round-robin policy to select which NUMA node should host the new memory block, ensuring an even distribution of memory across the system.
3. **Reserve Virtual Memory:** We use the `mmap` system call to request a raw, unassigned block of virtual address space directly from the kernel. This memory is zero-initialized, which efficiently satisfies the requirements for `calloc`.
4. **Set NUMA Policy via `mbind`:** This is the core of our solution. We manually create a bitmask on the stack and use the `mbind` system call with the `MPOL_BIND` policy. This is a direct instruction to the kernel: *"For this specific block of virtual memory, when you do eventually assign physical pages, they MUST come from the local memory of the target NUMA node we've selected."*
5. **First Touch:** We then iterate through the allocated region, writing to each memory page. This forces the kernel to immediately honor the `mbind` policy and allocate the physical RAM on the correct node.
6. **Return Pointer to Application:** A pointer to the usable memory region is returned. Our `realloc` implementation uses a standard `malloc-memcpy-free` approach to handle resizing.

4 Testing and Verification

4.1 Benchmark Setup

To validate our approach, we created a single-threaded benchmark program that allocates a large block of memory (256 MB) and performs a series of read/write operations on it.

4.2 Verification Methodology

The primary goal of testing was to verify correct memory placement. We used the `/proc` filesystem, which provides a window into the kernel's state.

1. **Run the Benchmark:** The test program is executed with our allocator injected via `LD_PRELOAD`.
2. **Pause Execution:** The program is paused immediately after the memory is allocated.
3. **Inspect Kernel Memory Maps:** We use the command `cat /proc/PID/numa_maps` to ask the kernel for the exact NUMA placement of the program's memory.

The output of `numa_maps` provides definitive proof that our allocator successfully instructed the kernel to bind and place the memory on the correct NUMA node.

5 Tools and Technologies

The implementation of our allocator relied on the following tools and technologies:

- **Linux OS:** Provides NUMA system calls and `LD_PRELOAD` support.
- **C Programming Language:** For implementing low-level memory management logic.
- `pthread` **Library:** Used for creating mutex locks and for the `pthread_once` thread-safe initialization mechanism.

- `string.h`**:** Required for `memcpy` in our `realloc` implementation.
- `mmap` **&** `munmap` **System Calls:** For reserving and releasing virtual memory regions from the kernel.
- `mbind` **System Call (**`numaif.h`**):** For issuing direct NUMA placement policy instructions to the kernel.
- `/proc` **Filesystem (**`numa_maps`**):** The primary tool for verifying correct memory placement.

6 Conclusion

This project successfully tackles the inefficiency of the default "first-touch" policy in NUMA systems. By using the `mbind` system call, our solution moves beyond influencing the kernel's behavior and instead provides direct, explicit instructions for memory placement. The implementation of the complete `malloc` family with thread-safe initialization results in a robust and transparent tool for optimizing application performance on modern server hardware.