Software Requirements Specification (SRS) for the High-Performance NUMA-Aware Memory Allocator

- **Team:** Devesh Mirchandani, Kartik Sharma, Padmnabh Tewari

1. Introduction

1.1 Purpose

The purpose of this document is to provide a comprehensive and unambiguous Software Requirements Specification for our project: a custom, NUMA-aware memory allocator. This document will serve as the foundational blueprint for our team's development, testing, and validation efforts. It details the functional and non-functional requirements, architectural design, and a detailed plan for proving the allocator's effectiveness.

1.2 Scope

Our project's scope is strictly defined: we will design and implement a dynamic memory allocator to serve as a high-performance replacement for the standard C library's `malloc()` and `free()`. The allocator will intelligently use **Non-Uniform Memory Access (NUMA)** features on multi-socket Linux servers to improve application performance. The project will **not** include a full-fledged memory management system for advanced features like thread caching, garbage collection, or memory pooling; our focus is solely on **local NUMA node allocation**.

1.3 Target Audience

This document is intended for our development team members, project stakeholders, and any future contributors. A reader should possess a foundational understanding of C programming, system architecture, and basic NUMA concepts.

2. System Context and Architectural Design

2.1 System Overview

The allocator will exist as a shared library (`.so`) that is external to the target application. This non-intrusive design is critical as it allows for performance optimization without requiring any modification to the application's source code. The core interaction is facilitated by the Linux dynamic linker.

2.2 Architectural Model: The Dynamic Interception Layer

Our architecture is a classic **dynamic linker interception model** using the `LD_PRELOAD` environment variable. When an application is executed, `LD_PRELOAD` forces the dynamic linker to load our shared library before the standard C library. Because our library exports functions with the same names (`malloc`, `free`), they are prioritized, effectively overriding the default system functions. This model allows our custom logic to be transparently injected into any application.

3. Functional Requirements (FRs)

3.1 FR-1: High-Fidelity `malloc` and `free` Interface

Requirement: The allocator shall expose two functions, `malloc(size_t size)` and `free(void *ptr)`, that are fully compatible with the C standard library's functions.

- **FR-1.1:** The `malloc` function shall accept a `size_t` argument and return a pointer to the newly allocated memory block.
- **FR-1.2:** The `free` function shall accept a pointer to a previously allocated memory block and release it. Passing a `NULL` pointer to `free` shall have no effect.
- **FR-1.3:** `malloc` shall return a `NULL` pointer if the allocation request cannot be fulfilled due to insufficient memory.

3.2 FR-2: NUMA Node Identification

Requirement: For every `malloc` call, the allocator must programmatically identify the NUMA node of the CPU on which the requesting thread is currently executing.

- **FR-2.1:** The primary mechanism shall be a combination of `sched_getcpu()` to get the current CPU and `numa_node_of_cpu()` to map that CPU to a NUMA node ID.
- **FR-2.2:** The identification logic shall be executed with minimal overhead to ensure it does not negate the performance gains.

3.3 FR-3: Explicit First-Touch Allocation

Requirement: Once the local NUMA node ID is identified, the allocator shall use `libnuma` to explicitly request memory from that specific node.

- **FR-3.1:** The core allocation function shall be `numa_alloc_onnode(size, node_id)`.
- **FR-3.2:** This strategy enforces a "first-touch" policy at the allocation phase, ensuring that the memory pages are physically located on the local node where the thread is first using them.

3.4 FR-4: Robust Fallback Mechanism

Requirement: The allocator shall include a robust and safe fallback mechanism to the standard C library's `malloc` and `free` functions.

- **FR-4.1:** The fallback shall be triggered if `libnuma` is not available, the system is not NUMA-enabled, or if `numa_alloc_onnode` returns an error.
- **FR-4.2:** The fallback mechanism shall be implemented using `dlsym` to dynamically resolve the addresses of the original `malloc` and `free` functions, thereby avoiding infinite recursion.

3.5 FR-5: Thread Safety

Requirement: The allocator must be completely **thread-safe** and reentrant.

- **FR-5.1:** It shall be capable of handling simultaneous `malloc` and `free` calls from multiple threads without introducing race conditions, deadlocks, or memory corruption.
- **FR-5.2:** We will rely on `libnuma`'s inherent thread-safety for the core allocation, and use explicit locking mechanisms (e.g., mutexes) if any custom, internal data structures are required.

4. Non-Functional Requirements (NFRs)

4.1 NFR-1: Performance

Requirement: Our primary objective is to demonstrably improve application performance.

- **NFR-1.1:** We will target a **minimum of 20% reduction in execution time** for a synthetic, memory-intensive, multi-threaded workload on a dual-socket server.
- **NFR-1.2:** We expect a significant reduction in cache and memory access latency, specifically for remote NUMA accesses.

4.2 NFR-2: Portability

Requirement: The allocator must function correctly on any Linux system with `libnuma` and a kernel that supports NUMA.

- **NFR-2.1:** On non-NUMA systems, the allocator must seamlessly fall back to the standard `malloc` without any user intervention or configuration.

4.3 NFR-3: Efficiency

Requirement: The overhead introduced by our custom allocation logic must be negligible.

- **NFR-3.1:** On single-socket systems or when the fallback is used, our allocator's performance overhead relative to the standard `malloc` shall be less than 5%.

5. Test Plan and Validation Strategy

We will conduct rigorous testing to validate both the correctness and the performance of our allocator.

5.1 Test Environment

- **Hardware:** A dual-socket server with at least two NUMA nodes (e.g., Intel Xeon E5 or higher).
- **Software:** Linux kernel 3.19 or later, `libnuma` library, and standard C compiler (`gcc`).

5.2 Test Cases

Test ID	Description	Expected Result	Rationale
TC-1	Single-threaded allocation on a NUMA-enabled system.	Memory allocated is correctly returned, and `numastat` shows a high `numa_hit` count for the corresponding node.	Validates the core "first-touch" policy.
TC-2	Multi-threaded allocation with CPU affinity pinning.	Each thread, pinned to a specific NUMA node, successfully allocates memory on that node. `numastat` shows high `numa_hit` counts on multiple distinct nodes.	Verifies correct multi-threaded, multi-node allocation.
TC-3	Deallocation and memory leak check.	A large allocation followed by deallocation should not show any memory leaks when using a tool like Valgrind.	Ensures our `free` wrapper is correct and safe.
TC-4	Execution on a non-NUMA system.	Our allocator transparently falls back to `malloc`, and the program executes correctly without errors or performance degradation.	Validates the robustness of the fallback mechanism.

5.3 Performance Analysis and Metrics

We will use a variety of tools to gather a complete performance profile. The following **conceptual graphs** illustrate our expected results.

- **Graph 1: Execution Time Comparison.** This bar chart will compare the total execution time of our test application when using the standard `malloc` versus our NUMA-aware allocator. We will plot the normalized time, with the standard allocator set to 100%. We expect our solution to demonstrate a significant reduction in execution time, providing quantifiable evidence of our NFR-1.
- **Graph 2: NUMA Statistics Over Time.** This line graph will show the `numa_hit` and `numa_miss` counts collected using `numastat` over the execution time of our test application. We expect to see an almost flat line for `numa_miss` and a steeply rising line for `numa_hit` when using our allocator, visually proving that we are successfully allocating on the local node. The standard `malloc`'s graph would show a higher proportion of `numa_miss` and `numa_foreign` pages.

6. Conclusion and Future Work

Our team is confident that this detailed SRS provides a clear and professional roadmap for delivering a high-quality, high-performance NUMA-aware memory allocator. We believe our solution will provide a substantial performance boost for memory-intensive applications on modern server hardware.

Upon successful completion and validation of the core project, we will explore the following advanced features:

- **Adaptive NUMA Policies:** Developing dynamic allocation policies that can adapt to changing application behavior at runtime, moving beyond a simple "first-touch" strategy.
- **Hardware Topology Awareness:** Extending the allocator to be aware of the underlying hardware topology, including L1/L2/L3 cache hierarchies and socket interconnects.
- **Advanced Benchmarking:** A formal, large-scale benchmarking effort against industry-standard memory allocators like **jemalloc** and **tcmalloc** to publish a comparative analysis.