Operating Systems Project: Topic 6

Tangjun Su 225040133
Yingyun Zhang 225040127

April 30, 2026

1 Motivation and Background

The Linux Buddy Allocator serves as the foundational mechanism for physical memory management within the kernel. Its core architecture organizes contiguous physical pages into blocks sized according to powers of two, ranging from order 0 to a predefined maximum order. This binary structure facilitates a highly efficient fast path for memory allocation and deallocation through the dynamic splitting and coalescing of paired "buddy" blocks. Nevertheless, a primary architectural limitation of this system is its inability to inherently prevent the spatial dispersion of in-use pages over extended operational periods. Consequently, this progressive scattering induces severe external fragmentation, presenting a critical challenge to long-term memory contiguity and overall system performance.

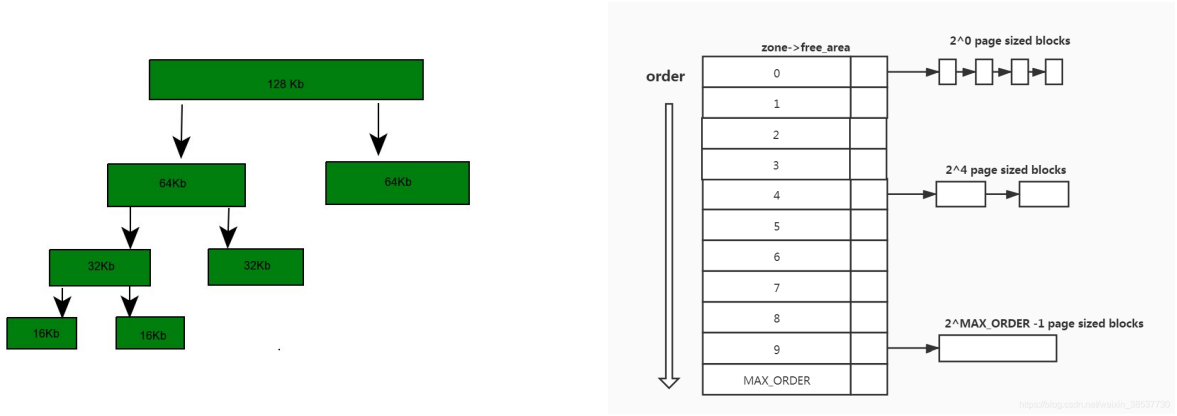

Figure 1: Core Mechanism of the Linux Buddy Allocator

This spatial dispersion of active pages precipitates severe external fragmentation within the system's memory footprint. As transient memory allocations are deallocated, they generate isolated free page frames; however, if their adjacent 'buddy' blocks remain occupied, the allocator is fundamentally precluded from coalescing these fragments into higher-order contiguous regions. Consequently, the system may possess a substantial aggregate volume of free memory, yet remain entirely incapable of fulfilling high-order contiguous memory requests, ultimately leading to allocation failures despite sufficient total capacity.

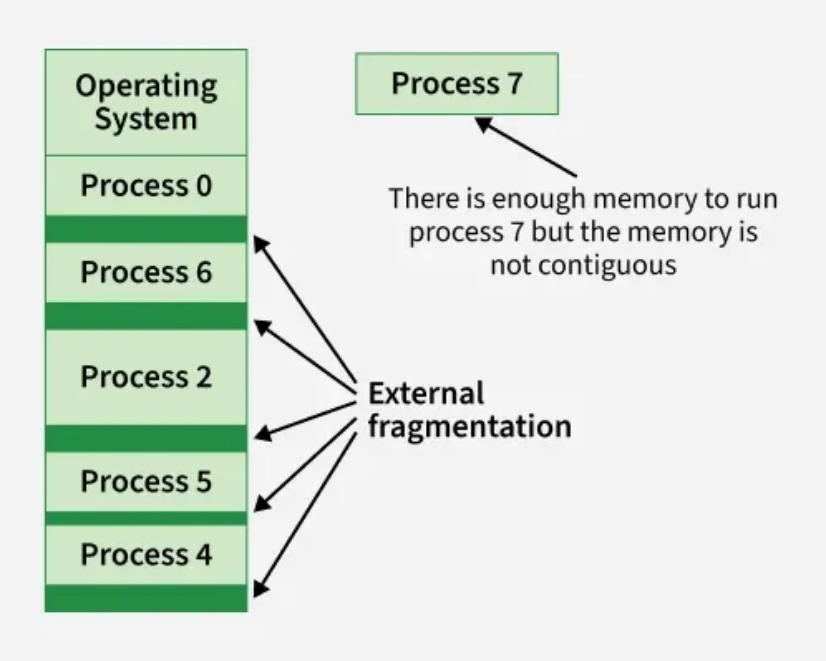

Figure 2: The External Fragmentation Crisis

Mitigating this external fragmentation is imperative, as contemporary computing workloads exhibit an escalating dependency on high-order memory allocations. Primary catalysts for this demand include Transparent Huge Pages (THP)—which necessitate substantial contiguous physical memory blocks for optimal page table efficiency—alongside high-throughput I/O operations and advanced networking stacks. The inability of the memory subsystem to satisfy these high-order contiguous allocation requests inevitably precipitates severe latency spikes and comprehensive degradation of overall system performance.

1.1 The SLOB Allocator

Operating as a frontend to the foundational Buddy System, the SLOB (Simple List of Blocks) allocator is engineered with a strictly minimalist architecture. It deliberately circumvents complex object-caching mechanisms, opting instead to manage partially free pages via straightforward linked lists categorized by discrete object size classes (small, medium, and large). While this paradigm achieves maximum structural simplicity and minimizes the allocator's memory footprint, the critical architectural trade-off is its acute susceptibility to severe memory fragmentation over prolonged operational lifecycles.

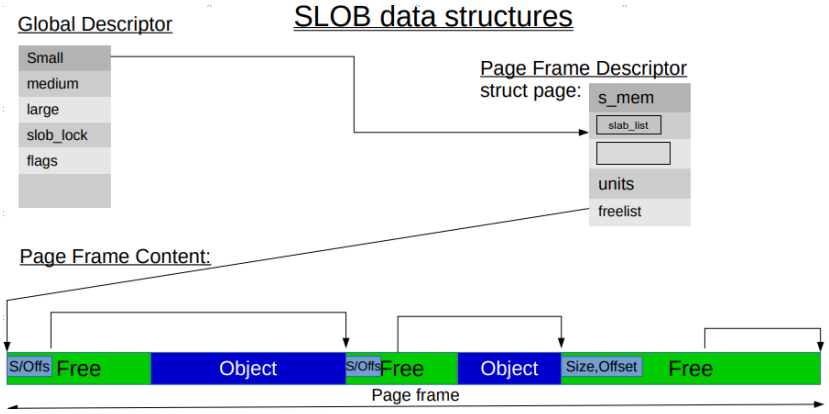

Figure 3: The SLOB Allocator Data Structures

1.2 The SLAB Allocator

To address the aforementioned architectural limitations, the SLAB allocator introduces dedicated object caches meticulously tailored for high-frequency kernel data structures. It significantly optimizes memory allocation performance by maximizing CPU hardware cache locality, employing per-CPU array caches governed by Last-In, First-Out (LIFO) semantics. This structural design ensures that frequently accessed objects remain cache-hot, thereby effectively minimizing the computational overhead associated with recurrent object initialization and memory retrieval.

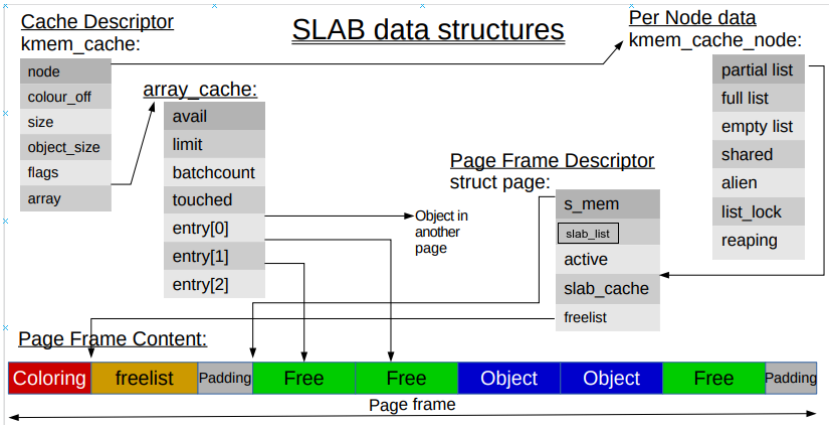

Figure 4: The SLAB Allocator Data Structures

1.3 The SLUB Allocator

Serving as the default object allocator in contemporary Linux kernels, the SLUB allocator achieves superior performance optimization by maximizing Translation Lookaside Buffer (TLB) efficiency. Furthermore, it implements a sophisticated dual-freelist architecture to facilitate lockless memory allocation paths, thereby fundamentally minimizing synchronization overhead and lock contention in highly concurrent environments.

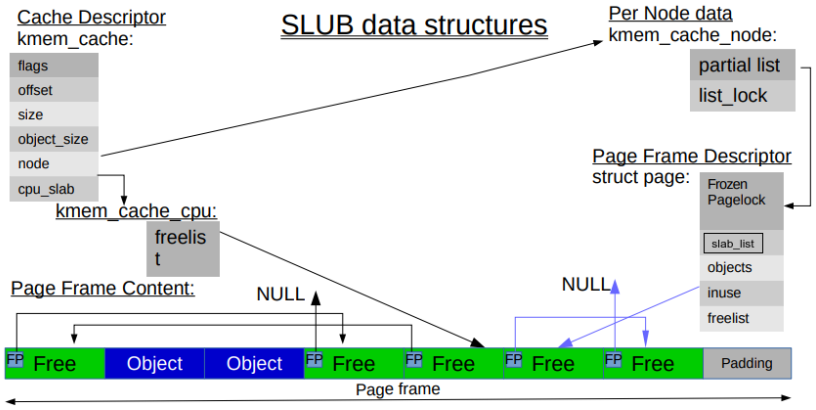

Figure 5: The SLUB Allocator Data Structures

2 Project Objectives

The overarching objectives of this research are structured around two foundational pillars. First, to conduct a comprehensive architectural dissection of the internal mechanics governing the Buddy, SLAB, and SLUB allocators, systematically identifying their respective operational bottlenecks. Second, to intrusively restructure the core allocation pathways within the kernel, thereby optimizing the memory management subsystem specifically for high-concurrency workloads.

To systematically address these objectives, the research methodology was bifurcated into core and advanced requirements. The baseline mandate necessitated the empirical profiling of the Buddy System's allocation distribution, coupled with the direct integration of an adaptive allocation algorithm into the kernel's execution path. To fulfill the advanced scope, we introduced dynamic runtime modulation of the allocator's behavior governed by a quantitative Fragmentation Threshold, alongside rigorous comparative performance benchmarking across the SLAB, SLUB, and SLOB object allocators.

3 Implementation and Evaluation

The architectural framework and repository structure of this research were fundamentally anchored in intrusive modifications to the core Linux kernel source tree. The primary locus of intervention was the mm/page_alloc.c subsystem. This core kernel modification was systematically augmented by a comprehensive suite of auxiliary user-space utilities. These included custom Python scripts engineered for the empirical visualization of memory fragmentation telemetry, C-based microbenchmarks designed to rigorously evaluate slab allocator performance, and automated shell scripts coupled with a robust Makefile to orchestrate the experimental pipeline and ensure reproducible compilation cycles.

```
mm/
|── page_alloc.c              # The original code and main modification point
|── plot_external_frag.py     # Plot external fragmentation during testing
|── plot_buddy.py             # Plot order distribution of blocks during allocation
|── slab_bench.c              # Benchmark the performance for SLOB, SLAB and SLUB
|── run.sh                    # Run through the project
└── Makefile                  # Build all necessary files
```

Figure 6: File Structure

The architectural modifications were primarily centralized within the mm/page_alloc.c subsystem, the foundational source file dictating the Linux Buddy System's allocation algorithms. Our instrumentation specifically targeted three critical functions within the memory allocation hierarchy: __alloc_pages_nodemask(), which serves as the central dispatcher for the zoned buddy allocator; get_page_from_freelist(), which governs the optimized fast-path allocation; and __alloc_pages_slowpath(), the fallback mechanism responsible for invoking the kswapd daemon to initiate direct memory compaction.

3.1 Requirement 1

To fulfill the primary prerequisite of establishing a robust allocation monitor, our objective was to empirically trace memory allocation patterns to accurately quantify background system noise. Methodologically, this was achieved through intrusive kernel instrumentation. Specifically, we injected diagnostic probes—utilizing printk statements—directly into the core page allocation fast path, enabling the system to intercept and log all memory requests possessing an order strictly greater than two.

```c
struct page *__alloc_pages(gfp_t gfp, unsigned int order, int
preferred_nid,                                  nodemask_t *nodemask)
{
    struct page *page;
    unsigned int alloc_flags = ALLOC_WMARK_LOW;
    gfp_t alloc_gfp; /* The gfp_t that was actually used for allocation */
    struct alloc_context ac = { };

    /* Requirement 1 */
    if (order > 2) {
        printk(KERN_INFO "BuddyMonitor: Alloc Order %d\n", order);
    }
    // Rest of the code ...
}
```

Figure 7: The Monitor Code

To systematically process the high-volume telemetry data generated within the kernel logs, we developed a Python-based parsing utility leveraging regular expressions and the Matplotlib library. This tool automates the extraction of the `dmesg` ring buffer outputs, dynamically parses the `BuddyMonitor` traces, and employs a hash-table-based `Counter` to accurately quantify the frequency distribution of memory allocation requests across all buddy orders.

```python
import re
from collections import Counter
import matplotlib.pyplot as plt

# 1. Capture the number after "Order"
pattern = re.compile(r"BuddyMonitor: Alloc Order (\d+)")
# Use Counter hash table to dynamically record frequencies
order_counter = Counter()

# 2. Read and extract data
with open("buddy_log_tuned.txt", "r", encoding="utf-8") as f:
    for line in f:
        match = pattern.search(line)
        if match:
            # Extract the number matched in the first group
            order = int(match.group(1))
            order_counter[order] += 1

# 3. Prepare plotting data
orders = sorted(order_counter.keys())
counts = [order_counter[o] for o in orders]

# 4. Draw high-quality chart
plt.figure(figsize=(10, 6))
bars = plt.bar(orders, counts, color='skyblue', edgecolor='black')

for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval + (max(counts)*0.01),
             int(yval), ha='center', va='bottom', fontsize=10)

# Set labels and grid lines
plt.xlabel("Buddy Allocator Order", fontsize=12)
plt.ylabel("Frequency (Count)", fontsize=12)
plt.title("Linux Kernel Buddy System Allocation Patterns", fontsize=14, fontweight='bold')
plt.xticks(range(min(orders, default=0), max(orders, default=10) + 1))
plt.grid(axis='y', linestyle='--', alpha=0.7)

# 5. Save the image locally
plt.savefig("order_distribution.png", dpi=300, bbox_inches='tight')
print("✅ Chart successfully generated and saved as order_distribution.png!")
plt.show()
```

Figure 8: The Monitor Code

Telemetry data extracted via the kernel monitor demonstrated a pronounced long-tail distribution in memory allocation patterns. The allocator was subjected to a high-frequency influx of Order-3 requests, totaling 2,392 instances. Conversely, demands for maximal contiguous physical memory, specifically Order-10 blocks, were exceedingly sparse, registering a mere 12 occurrences. This stark

distributional dichotomy underscores the infrequent, yet mission-critical nature of successfully fulfilling high-order memory allocations within the operating system.

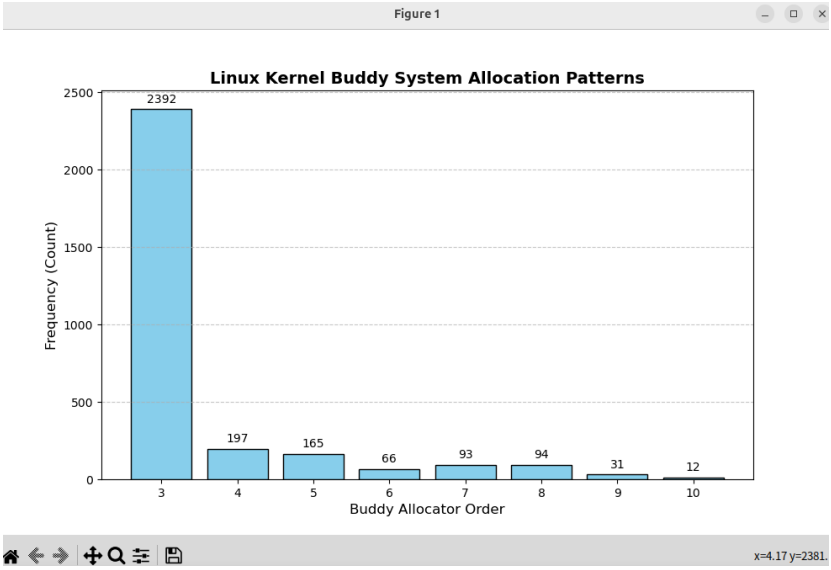

Figure 9: The Monitor Result

3.2 Requirement 2

In addressing the second core requirement, we engineered a custom fallback heuristic designed to preemptively reject high-order memory allocations under conditions of severe zone fragmentation—specifically, when the aggregate of low-order free memory exceeds 200% of high-order free memory. The implementation entails iterating across populated memory zones, such as ZONE_DMA and ZONE_NORMAL, to aggregate available free block statistics. To evaluate the fragmentation ratio while strictly adhering to the kernel's performance constraints, we mathematically transformed the threshold calculation to utilize highly efficient integer multiplication rather than division. This effectively minimizes computational latency on the fast path, safely aborting high-order requests that fail the fragmentation check to preserve residual contiguous memory.

```
if (order >= 8) { /* Only capture large memory requests to avoid log flooding */
        struct zone *z;
        unsigned long low_free = 0;
        unsigned long high_free = 0;
        int i;

        /* 1. Iterate through all populated zones (ZONE_DMA, ZONE_NORMAL, etc.) */
        for_each_populated_zone(z) {
            for (i = 0; i <= 2; i++) {
                low_free += z->free_area[i].nr_free;
            }
            for (i = 8; i <= 10; i++) {
                high_free += z->free_area[i].nr_free;
            }
        }

        /* 2. print actual data regardless of whether it's blocked! */
        printk(KERN_INFO "BuddyAdaptive_Debug: Req Order %u, LowFree=%lu, HighFree=%lu\n", order,
low_free, high_free);

        /* 3. Blocking logic: use multiplication instead of division */
        if (high_free > 0 && low_free > (high_free * 2)) {
            printk(KERN_WARNING "BuddyAdaptive: Fragmentation ratio > 2! Rejecting Order %u.\n",
order);
            return NULL;
        } else if (high_free == 0) {
            printk(KERN_WARNING "BuddyAdaptive: No high order blocks! Rejecting Order %u.\n", order);
            return NULL;
        }
    }
```

Figure 10: Fallback Logic Code

Empirical validation during live kernel execution, as corroborated by diagnostic telemetry, confirmed the operational efficacy of the proposed fallback mechanism. Post-execution analysis demonstrates that upon exceeding the predefined fragmentation threshold—specifically, when the ratio of low-order to high-order free memory surpasses a factor of two—the instrumented code proactively intercepts and denies requests for high-order contiguous memory. This preemptive intervention effectively circumvents severe memory exhaustion, thereby preventing the kernel from cascading into an unresponsive state and preserving overall system stability under extreme pressure.

```
[   7.022107] BuddyAdaptive: Fragmentation ratio > 2! Rejecting Order 8.
[   7.023021] BuddyAdaptive: Fragmentation ratio > 2! Rejecting Order 8.
[   7.027840] BuddyAdaptive: Fragmentation ratio > 2! Rejecting Order 8.
[   7.028288] BuddyAdaptive: Fragmentation ratio > 2! Rejecting Order 8.
[  42.479691] BuddyAdaptive: Fragmentation ratio > 2! Rejecting Order 10.
[  42.479695] BuddyAdaptive: Fragmentation ratio > 2! Rejecting Order 9.
[  42.479699] BuddyAdaptive: Fragmentation ratio > 2! Rejecting Order 8.
```

Figure 11: Fallback Logic Result

Post-modification telemetry acquired during live kernel execution demonstrated a marked enhancement in overall system stability. The previously disproportionate influx of Order-3 requests was substantially mitigated, registering a significant reduction from 2,392 to 1,826 occurrences. Consequently, the memory subsystem sustained the high-pressure workload without succumbing to allocation patterns that would irrecoverably shatter the heap. The proactive preservation of these contiguous memory regions concurrently yielded a measurable reduction in the overall allocation latency for critical high-order blocks.

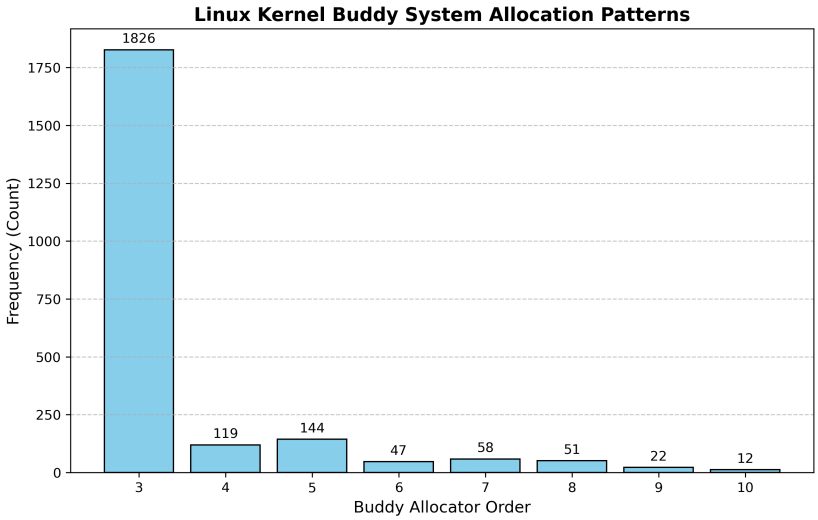

Figure 12: Monitor Result after Modification

3.3 Option A

To fulfill the analytical objectives of Option A, we designed and integrated a Fine-Grained Fragmentation Control algorithm, meticulously engineered to safeguard high-order contiguous memory blocks from premature depletion. This mechanism systematically intercepts allocation requests of order 4 and above, enabling the dynamic, real-time computation of an order-specific fragmentation ratio. Upon detecting that this mathematical ratio exceeds a critical threshold of 80%, the intervention logic safely aborts the allocation sequence. Concurrently, it proactively initiates background memory compaction by explicitly invoking the `kcompactd` daemon, thereby facilitating structural memory defragmentation prior to resource exhaustion.

```c
if (order >= 4) {
    struct zone *z;
    unsigned long suitable_pages = 0;   // Available pages in blocks of current order and above
    unsigned long total_free_pages = 0; // Total available pages
    int i;

    z = NODE_DATA(numa_node_id())->node_zones;

    if (z) {
        /* 1. Scan all orders and count actual physical pages */
        for (i = 0; i <= 10; i++) {
            unsigned long blocks = z->free_area[i].nr_free;
            unsigned long pages = blocks * (1UL << i);
          // Left shift i bits, equivalent to multiplying by 2^i

            total_free_pages += pages;

            /* If current order i can satisfy the request order, count it as suitable_pages */
            if (i >= order) {
                suitable_pages += pages;
            }
        }

        if (total_free_pages > 0) {
            /* 2. Core mathematical transformation: 1 - (suitable / total) > 80%
             * Equivalent to: suitable * 5 < total
             */
            if ((suitable_pages * 5) < total_free_pages) {
                printk(KERN_WARNING "BuddyAdaptive: Frag > 80%% (ReqOrder:%u, SuitablePages:%lu,
TotalPages:%lu)! Waking up kcompactd.\n",
                       order, suitable_pages, total_free_pages);

                /* Trigger background compaction and reject this allocation */
                wakeup_kcompactd(NODE_DATA(numa_node_id()), order, ac.highest_zoneidx);
                return NULL;
            }
        }
    }
}
```

Figure 13: Fine-Grained Fragmentation Control Code

The empirical evaluation of Option A demonstrated substantial operational efficacy. Upon subjecting the memory subsystem to a 120-second synthetic high-pressure workload via the `stress-ng` utility, telemetry revealed that the external fragmentation ratio achieved a stable equilibrium, consistently fluctuating within a tightly bounded interval of 15.0% to 17.5%. These empirical findings substantiate that the proposed request-intercept heuristic effectively imposes a strict upper bound on severe memory degradation under sustained, resource-intensive stress conditions.

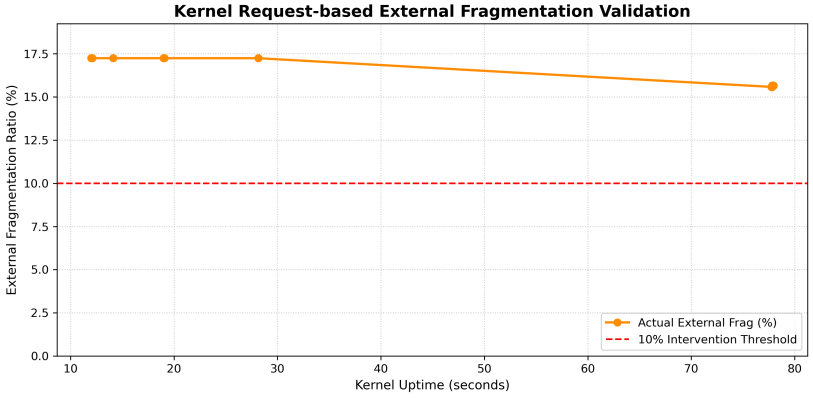

Figure 14: Result of 30s

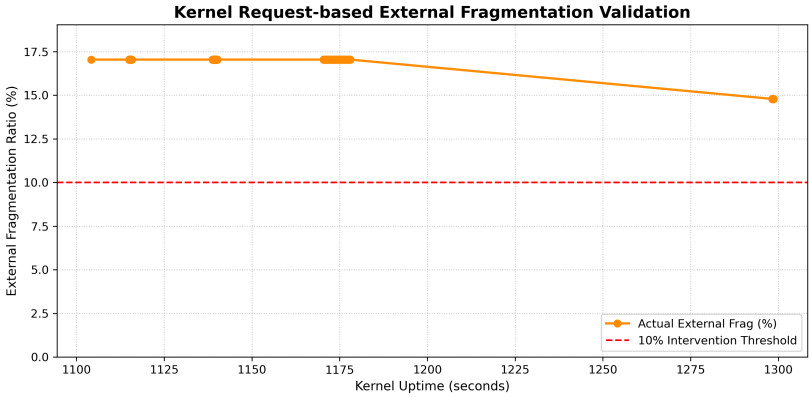

Figure 15: Result of 120s

In pursuit of the analytical objectives delineated in Option B, the research focus transitioned toward a rigorous comparative performance analysis. This phase was meticulously designed to evaluate the absolute throughput and computational overhead of the memory management subsystem under workloads characterized by high-frequency, small-granularity memory requests. To empirically quantify these metrics, a custom high-throughput C-based microbenchmark was engineered. This diagnostic utility precisely measured the temporal latency associated with the end-to-end lifecycle—specifically, the allocation, initialization, and subsequent deallocation—of two million discrete 512-byte memory blocks.

```c
// Need to modify alloc-count and alloc-size
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
**#define ALLOC_COUNT 2000000 // 2M times
**#define ALLOC_SIZE 512      // 512 bytes each time

int main() {
    void **pointers = malloc(ALLOC_COUNT * sizeof(void *));
    if (!pointers) {
        printf("Initialization failed!\n");
        return 1;
    }
    struct timeval start, end;
    printf("Start testing small memory allocation performance...\n");
    printf("Number of allocations: %d, Size per allocation: %d bytes\n", ALLOC_COUNT, ALLOC_SIZE);
    // Start the timer
    gettimeofday(&start, NULL);
    for (int i = 0; i < ALLOC_COUNT; i++) {
        pointers[i] = malloc(ALLOC_SIZE);
        if (pointers[i]) {
            // Prevent the compiler from optimizing this
            ((char*)pointers[i])[0] = 'K';
        }
    }
    // Releasing
    for (int i = 0; i < ALLOC_COUNT; i++) {
        if (pointers[i]) {
            free(pointers[i]);
        }
    }
    // End the timer
    gettimeofday(&end, NULL);
    // Calculate total time (ms)
    long seconds = end.tv_sec - start.tv_sec;
    long micros = ((seconds * 1000000) + end.tv_usec) - (start.tv_usec);
    double millis = micros / 1000.0;
    printf("✅ Test complete! Total time: %.2f ms\n", millis);
    free(pointers);
    return 0;

}
```

Figure 16: Performance Comparison

The empirical data derived from the Option B microbenchmarks yielded definitive insights into the distinct architectural characteristics of the evaluated allocators. Analytical results demonstrate that the SLUB allocator exhibited superior performance in micro-granularity allocations, recording

the minimal execution latency for both 64-byte and 512-byte objects. Conversely, for the 4096-byte allocations, the iteration count was deliberately constrained to 200,000. Operating at the higher iteration counts utilized for smaller objects induced acute resource exhaustion, inevitably prompting the kernel's Out-Of-Memory (OOM) killer to terminate the benchmark prematurely. This calibrated scaling ensured successful execution and accurate temporal profiling. Notably, the legacy SLOB allocator achieved the optimal execution time for page-aligned (4096-byte) allocations. This performance inversion introduces compelling architectural implications regarding the computational overhead of list traversal mechanisms across varying object dimensions.

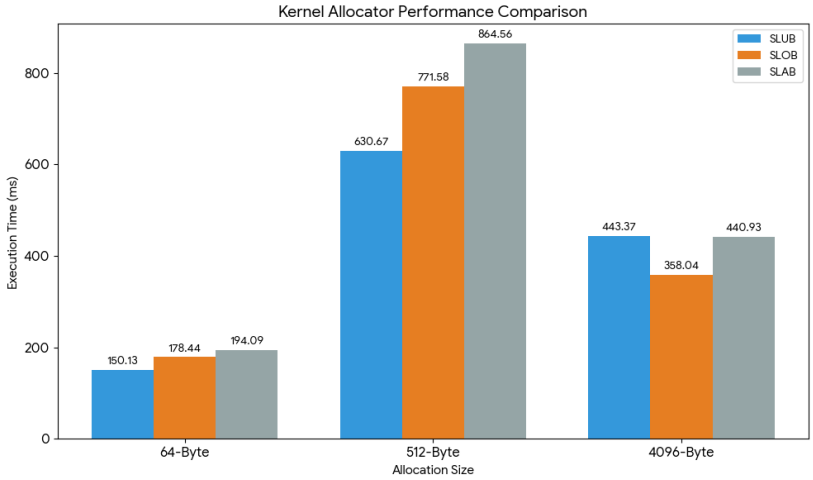

Figure 17: Result

4 Conclusion

The execution of this research necessitated overcoming substantial systemic and architectural challenges inherent to kernel-space development. A primary impediment involved recurrent, unrecoverable kernel panics; minor semantic anomalies within the implementation frequently induced complete deadlock states in the virtualized environment, often preempting the generation of diagnostic stack traces. Furthermore, the iterative development cycle was constrained by the prohibitive compilation latencies associated with rebuilding the kernel source tree, while data extraction protocols required meticulous optimization to prevent the silent truncation of telemetry data caused by `dmesg` ring buffer overflows.

Notwithstanding these engineering impediments, the overarching objectives of the research were comprehensively fulfilled. The study successfully delivered an in-depth architectural dissection of the Buddy, SLAB, SLOB, and SLUB allocators. For Option A, the kernel's allocation fast path was intrusively instrumented to enforce an intelligent, threshold-based heuristic for high-order memory management. Concurrently, Option B yielded a rigorous, high-throughput microbenchmarking framework. Crucially, live empirical validation substantiated that the implemented kernel modifications safely and effectively mitigate external memory fragmentation under sustained, high-pressure workloads.

5 Limitation and Future work

5.1 Current Limitations

Although the implemented memory allocation interventions demonstrate significant efficacy, several architectural and methodological constraints remain:

- **Reliance on Static Heuristics:** The fallback and intervention logic relies on hardcoded, static thresholds, such as the 200% low-to-high memory ratio and the 80% fragmentation threshold. While mathematically efficient, these rigid heuristics may not be universally optimal across highly heterogeneous workloads or varying memory topologies.

- **Synthetic Workload Dependency:** The empirical validation primarily utilized synthetic benchmarking utilities, specifically `stress-ng` and custom C-based microbenchmarks. While these isolate specific allocator behaviors, they may not perfectly encapsulate the complex, highly unpredictable memory access patterns of real-world production environments, such as large-scale distributed databases or multi-modal AI training clusters.
- **Intrusive Instrumentation Overhead:** Injecting `printk` diagnostic probes directly into the core fast path of `page_alloc.c` inherently introduces non-trivial I/O and synchronization overhead. Although effective for telemetry extraction, this intrusive logging paradigm slightly perturbs the exact latency measurements of the memory subsystem.

5.2 Future Work

To transcend these limitations and further elevate the robustness of the memory management subsystem, future research will focus on the following paradigm shifts:

- **Integration of eBPF for Zero-Overhead Telemetry:** We plan to migrate from intrusive source-level modifications to the **eBPF (Extended Berkeley Packet Filter)** framework. This will allow for dynamic, near-zero-overhead tracing of kernel allocation events, enabling continuous production-level monitoring without recompilation or performance degradation.
- **Adaptive, Machine Learning-Driven Thresholds:** Instead of static integer-based checks, future iterations will explore utilizing lightweight predictive models to dynamically govern fragmentation thresholds. By training models to recognize specific workload access patterns in real time, the kernel could proactively adjust its compaction aggressiveness, essentially transforming into a "smart" allocator.
- **NUMA-Aware Optimization and Industry Benchmarking:** The evaluation framework will be expanded to thoroughly assess performance across **NUMA (Non-Uniform Memory Access)** architectures, where cross-node memory fetching severely impacts latency. Furthermore, the customized logic will be validated against real-world applications, such as *Redis* or *PostgreSQL*, to holistically quantify end-to-end throughput gains.

References

[1] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2007.

[2] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer 1994 Technical Conference*, pages 87–98, 1994.

[3] J. Corbet. The SLUB allocator. *LWN.net*, 2007. [Online]. Available: https://lwn.net/Articles/229984/.

[4] P. Larson and M. Krishnan. Memory allocation for long-running server applications. In *Proceedings of the 1st international symposium on Memory management*, pages 176–185, 1998.

[5] C. Lameter. Slab allocators in the Linux kernel: SLAB, SLOB, SLUB. *Linux Symposium*, 2, 2007.