Observation and Adaptive Improvement of the Linux Buddy Allocator

Shuntian Yao
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
225040188

Ruichen Liu
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
225040126

***Abstract*—This project studies the Linux buddy allocator from both an observational and an adaptive optimization perspective. We first add runtime monitoring to the kernel page allocation path and export allocator statistics through a custom `/proc/buddy_monitor` interface. The exported fields include request count, success count, failure count, slowpath count, compaction count, adaptive trigger count, and recorded fragmentation percentage. We then implement an adaptive compaction policy for higher-order allocations, where compaction becomes more aggressive only when fragmentation severity exceeds a threshold. To evaluate the design, we use three workloads: general memory pressure with `stress-ng`, a custom user-space fragmentation tool, and a custom kernel module that repeatedly allocates high-order pages. The final aggressive order-10 experiment triggers 86 slowpath events, 86 compaction events, and 86 adaptive activations while preserving a 100% allocation success rate. These results show that the modified allocator is not only observable but also experimentally controllable under carefully designed high-order pressure scenarios.**

***Index Terms*—Linux kernel, buddy allocator, memory compaction, fragmentation, high-order allocation, kernel instrumentation**

I. Introduction and Motivation

Modern operating systems must manage physical memory efficiently under a wide range of workloads. In Linux, the buddy allocator is the core subsystem responsible for physical page allocation. It provides fast allocation and deallocation by organizing free memory into blocks of size 2^{order} pages and supporting recursive split and merge operations. This design is efficient in common cases, but it becomes increasingly sensitive to fragmentation when the system must satisfy higher-order allocation requests.

The main problem addressed in this project is the difficulty of observing and improving high-order page allocation behavior under fragmentation pressure. A system may still have a large amount of free memory in total, but if this free memory is distributed across many small blocks, high-order contiguous allocations may have to enter the allocator slowpath or trigger compaction. In practice, this matters because some kernel operations depend on large contiguous memory blocks, and fragmentation can make these requests more expensive or less reliable.

This problem is worth studying for two reasons. First, allocator behavior is not directly visible through standard interfaces in a way that exposes request counts, slowpath entries, compaction attempts, and the effect of a modified policy. Second, the buddy allocator is a central kernel subsystem, so even a targeted modification can reveal important tradeoffs between allocation success, compaction aggressiveness, and fragmentation awareness. Therefore, our project focuses on both runtime observation and adaptive improvement of the Linux buddy allocator.

The project has two main goals:

1) add a runtime monitoring mechanism so that allocator behavior can be observed directly from user space;
2) design and implement an adaptive compaction policy that becomes more aggressive only when high-order allocation pressure and fragmentation justify it.

In summary, this project turns the Linux buddy allocator from a relatively opaque subsystem into one that is both observable and experimentally modifiable.

II. Background and Related Mechanisms

A. Linux Buddy Allocator

The Linux buddy allocator manages physical pages in blocks whose size is expressed by an integer `order`. For example, `order 0` means 1 page, `order 1` means 2 contiguous pages, and `order 10` means 1024 contiguous pages. When a smaller block is requested but only a larger block is available, the larger block is recursively split. When two neighboring free blocks of the same order become free, they can be merged back together.

This design is efficient and elegant for contiguous page management. However, the allocator can suffer from fragmentation. Over time, frequent allocation and free operations tend to produce many lower-order free blocks while reducing the number of large contiguous blocks. As a result, high-order requests become harder to satisfy even if total free memory is still large.

B. Fast Path, Slowpath, and Compaction

The allocator first attempts to satisfy requests through a fast path. If suitable free pages are available immediately, allocation succeeds quickly. When the fast path cannot satisfy a request, the allocator enters a slower recovery path, commonly referred to as *slowpath*. In this path, the kernel may

wake reclaim threads, attempt direct reclaim, attempt direct compaction, and retry allocation under adjusted conditions.

Compaction is a kernel mechanism that rearranges movable pages in order to create larger contiguous free regions. It is especially relevant for high-order allocations. If fragmentation is the main problem, compaction can often recover enough large blocks to make an allocation succeed. However, compaction is not free. It introduces additional work, including page migration and associated bookkeeping. Therefore, compaction should ideally be used more aggressively only when the allocator is clearly under fragmentation-related pressure.

C. Existing Observation Interfaces

Linux already provides `/proc/buddyinfo`, which shows the current free block distribution for each zone and order. This interface is useful for observing allocator *state*, but it does not directly show allocator *behavior*. In particular, it does not tell us how many allocation requests were issued, how many succeeded or failed, how often slowpath was entered, how often compaction was attempted, or whether an added adaptive policy was triggered. This motivated us to implement a dedicated monitoring interface.

D. Related Mechanisms Used in This Project

Our project interacts with several related kernel mechanisms:

- the page allocation path in `mm/page_alloc.c`,
- direct compaction logic,
- zonelist scanning during allocation,
- `/proc`-based runtime monitoring via `seq_file`,
- atomic counters for concurrency-safe statistics.

These mechanisms form the technical foundation of our implementation.

III. Design and Implementation

Our implementation was concentrated in the Linux kernel source file `mm/page_alloc.c`, with additional user-space and kernel-space test programs used for evaluation.

A. Observation Design

The first part of the project added runtime observation support. We implemented per-order counters to track allocator behavior:

- `req`: number of allocation requests,
- `success`: number of successful allocations,
- `fail`: number of failed allocations,
- `slowpath`: number of requests that entered slowpath,
- `compact`: number of compaction attempts,
- `adaptive`: number of times our adaptive policy was triggered,
- `frag_pct`: last recorded fragmentation percentage for a given order.

These counters were implemented as arrays indexed by `order`, allowing us to observe allocator behavior separately for different allocation sizes. To ensure reliability under concurrency, we used `atomic_long_t` rather than ordinary integer counters.

B. /proc/buddy_monitor

We exported the collected statistics through a new `/proc` interface:

```
/proc/buddy_monitor
```

This interface was implemented using the standard `seq_file` and `proc_create()` mechanism. It prints the statistics in the following format:

```
order req success fail slowpath compact adaptive
    frag_pct
```

This design makes the monitoring data easy to access using simple commands such as `cat /proc/buddy_monitor`. Compared with `/proc/buddyinfo`, our interface exposes dynamic allocator behavior rather than only the current free-page distribution.

C. Counter Consistency Fix

During development, we observed that early versions of the monitoring logic sometimes produced inconsistent statistics where:

```
req != success + fail
```

This happened because some failure exits in the allocation path incremented the request counter but did not always increment the failure counter. We corrected these special exits so that the accounting became consistent. This fix was essential, because any later interpretation of experimental data would be unreliable if the counters themselves were inconsistent.

D. Fragmentation Metric

We designed a simple fragmentation metric for experimental use. The basic idea was to estimate fragmentation severity based on the distribution of free pages across different orders inside a zone. We defined:

- `small_free_pages`: free pages belonging to orders below a cutoff,
- `total_free_pages`: all free pages in the zone.

Then we computed

$$\texttt{frag\_pct} = \frac{\texttt{small\_free\_pages}}{\texttt{total\_free\_pages}} \times 100 \qquad (1)$$

with suitable bounds. This metric is not intended to be the only possible fragmentation metric, but it is simple, observable, and sufficient for implementing a threshold-driven adaptive policy in a course project setting.

E. Adaptive Compaction Policy

The second part of the project introduced an adaptive compaction policy. The policy logic is:

1) only consider higher-order requests, specifically `order >= 3`;
2) estimate fragmentation severity across the relevant zones in the current allocation context;
3) if the measured fragmentation level exceeds a threshold, mark this request as requiring adaptive compaction;

4) raise compaction priority to `MIN_COMPACT_PRIORITY`.

This design allows the allocator to remain conservative in ordinary cases while becoming more aggressive when high-order requests encounter fragmentation pressure. The adaptive trigger was integrated into the slowpath logic of the allocator. Therefore, the policy does not change overall allocator semantics globally; instead, it influences the decision-making path precisely when the allocator is already under pressure.

F. Compaction and Slowpath Instrumentation

To measure the effect of our design, we instrumented slowpath entry points and direct compaction invocation points. This separation is important:

- `slowpath` indicates that fast allocation was insufficient;
- `compact` indicates that compaction was attempted;
- `adaptive` indicates that our own policy was activated.

Together, these counters allow us to distinguish normal allocator activity from the behavior introduced by our modification.

G. Experimental Tools Implemented by Us

1) `fragment_stress.c`: We implemented a user-space fragmentation workload that allocates many blocks of varying sizes, frees part of them to create holes, and performs larger burst allocations after fragmentation is introduced. Its main purpose is to shape memory state and create fragmentation-like patterns.

2) `high_order_test.c`: We also implemented a custom kernel module that repeatedly requests high-order pages with `alloc_pages()`, optionally holds multiple successful allocations before freeing them, and can run in an aggressive `hold_until_fail` mode. This tool directly stresses the high-order page allocation path inside the kernel, which made it the most effective workload for validating the adaptive mechanism.

IV. Experimental Setup and Methodology

A. Platform and Software Environment

The experiments were performed in the following environment:

- Operating system: Ubuntu 20.04.3 LTS
- Kernel source version: Linux 5.15.180
- Runtime environment: VirtualBox
- Compiler: GCC
- Build tools: `make`, `bc`, `bison`, `flex`
- Kernel dependencies: `libssl-dev`, `libelf-dev`, `dwarves`, `libncurses-dev`, `cpio`, `fakeroot`, `rsync`, `wget`, `git`, `kmod`
- Optional workload tool: `stress-ng`

We compiled, installed, and booted the modified kernel successfully, and all final experiments were performed on the running custom 5.15.180 kernel.

B. Verification Procedure

After booting the modified kernel, we verified successful deployment using:

```
uname -r
cat /proc/buddy_monitor
```

This confirmed that the running kernel was the modified 5.15.180 version and that `/proc/buddy_monitor` was created correctly and readable.

Fig. 1. Verification that the system booted into the modified Linux 5.15.180 kernel.

C. Data Collection

We implemented a helper script `collect_metrics.sh` to collect:

- `/proc/buddyinfo`,
- `/proc/buddy_monitor`,
- the tail of `dmesg`.

This ensured that each experiment produced a reproducible snapshot containing memory state, allocator behavior, and recent kernel log evidence.

D. Workloads

We used three categories of workloads.

1) General Memory Pressure: We first used:

```
stress-ng --vm 4 --vm-bytes 70% --timeout 60s
```

This workload creates general virtual memory pressure. It is useful for observing large-scale allocation activity, but it does not necessarily create the exact fragmentation pattern needed for reliable high-order slowpath activation.

2) User-Space Fragmentation Workload: We then used:

```
bash scripts/run_fragment_experiment.sh 3072 2
```

This workflow builds and executes `fragment_stress.c`, which allocates and partially frees memory to create holes, then performs burst allocations. It is intended to perturb free-page distribution and observe how the allocator state changes.

3) Kernel High-Order Allocation Workload: Finally, we used:

```
bash scripts/run_high_order_module_test.sh 10 512
    5000 0 512 1
```

This is the most aggressive workload. It repeatedly requests `order 10` allocations in kernel space and uses a `hold_until_fail` mode, which keeps successful high-order allocations resident until failure or the end of the test. This is designed to consume large contiguous free blocks and force the allocator into slowpath.

E. Experimental Strategy

Our evaluation followed a staged methodology:

1) verify that observation counters work correctly in a running kernel;
2) establish baseline behavior under ordinary system pressure;
3) use a user-space fragmentation workload to study changes in memory layout;
4) use a kernel high-order workload to directly stress the allocation path;
5) compare allocator counters and free-page distributions across workloads.

This methodology was intentionally incremental. Rather than assuming that a single stress tool would be sufficient, we used progressively more targeted workloads until the adaptive path was reliably exercised.

V. Results and Analysis

A. Baseline Observation

The monitoring interface worked correctly after booting the modified kernel. At baseline, we observed nonzero request counts across multiple orders, with request and success counts matching and failure counts remaining zero. This confirmed that our observation logic was stable and internally consistent.

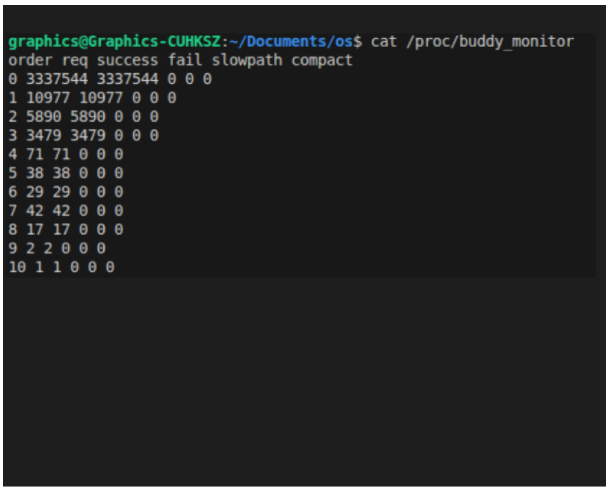

Fig. 2. Baseline `/proc/buddy_monitor` output after booting the modified kernel.

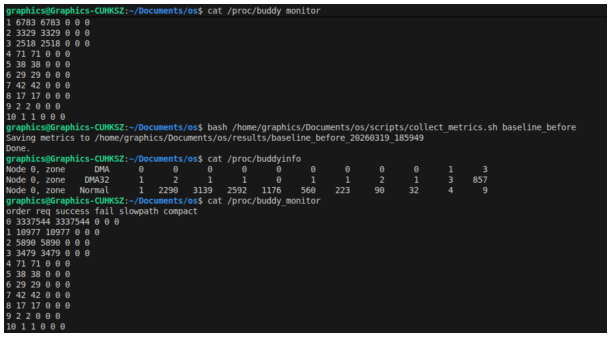

Fig. 3. Baseline `/proc/buddyinfo` showing the initial free-block distribution.

B. Stress-ng Results

Under `stress-ng`, allocator activity increased significantly, especially for lower-order requests. In one representative run, the counters changed as shown in Table I.

TABLE I
Representative `stress-ng` Before/After Counters

Order	req before	req after	slowpath	compact	adaptive
0	3,337,544	13,329,866	0	0	0
9	2	990	7	7	0

In another run with the adaptive-enabled kernel, the system showed the values in Table II.

TABLE II
Representative Adaptive-Enabled `stress-ng` Run

Order	req after	slowpath	compact	adaptive	frag_pct
9	2,199	2	2	0	2

These runs demonstrate that general pressure can trigger some high-order activity and even compaction, but it does not reliably trigger the adaptive policy.

C. Fragmentation Workload Results

The `fragment_stress` program clearly changed the allocator state, as reflected in `/proc/buddyinfo`. For example, after the experiment, the `Normal` zone showed lower counts in several lower orders, indicating that free-page distribution had shifted. However, `/proc/buddy_monitor` still showed zero `slowpath`, zero `compact`, and zero `adaptive`. This suggests that `fragment_stress` was effective at shaping memory state, but not sufficient to reliably force the allocator into the high-order slowpath in our VM environment.

This distinction is important:

- `buddyinfo` reflects allocator state;
- `buddy_monitor` reflects allocator behavior.

The workload changed state, but did not yet trigger the exact behavior we needed.

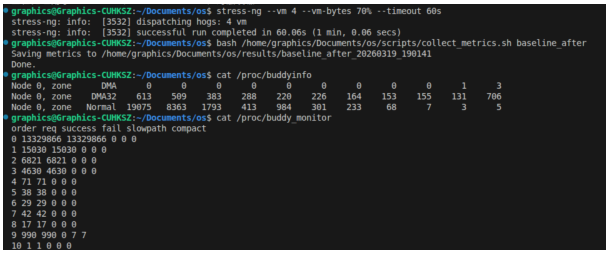

Fig. 4. Representative `/proc/buddyinfo` output after the user-space fragmentation workload.

D. Early High-Order Module Results

Before the final aggressive configuration, we used several versions of the `high_order_test` module. Early versions allocated and freed pages too quickly, or held too few pages at once. As a result, they increased high-order request counts but still showed nonzero success, zero failure, zero slowpath, zero compaction, and zero adaptive activation.

This revealed an important experimental fact: high-order allocations alone are not enough if the system still retains enough contiguous blocks. The workload must sustain pressure long enough to consume large free areas.

E. Final Aggressive High-Order Experiment

The final successful workload was:

```
bash scripts/run_high_order_module_test.sh 10 512
    5000 0 512 1
```

This configuration used `order = 10`, many repeated attempts, long holding time, and `hold_until_fail` mode. This workload finally forced the allocator into the target path. The final order-10 statistics are summarized in Table III.

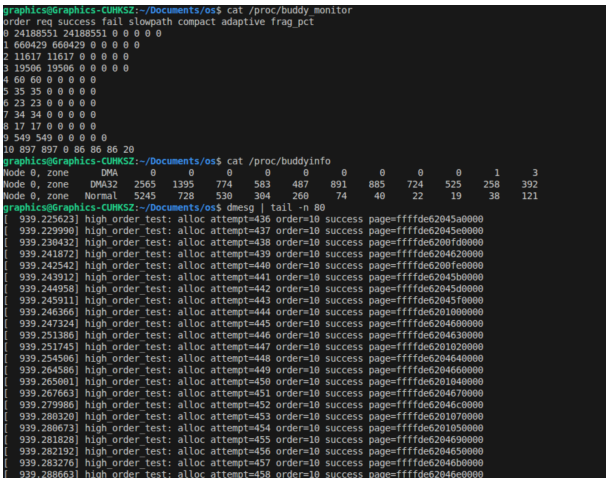

Fig. 5. Final `/proc/buddy_monitor` output from the aggressive order-10 kernel module experiment.

F. Interpretation of Final Results

These numbers show a clear behavioral chain:

1) the allocator received many `order 10` requests;
2) most requests succeeded directly, but a nontrivial subset entered slowpath;
3) every slowpath event in that final subset triggered compaction;
4) the adaptive policy was activated in exactly those compaction-related cases.

This means the fast path alone was no longer sufficient under sustained high-order pressure. The allocator had to enter the recovery path, and our modification became active there.

TABLE III
FINAL ORDER-10 RESULT

Metric	Value
Requests	897
Success	897
Fail	0
Slowpath	86
Compact	86
Adaptive	86
Fragmentation percentage	20

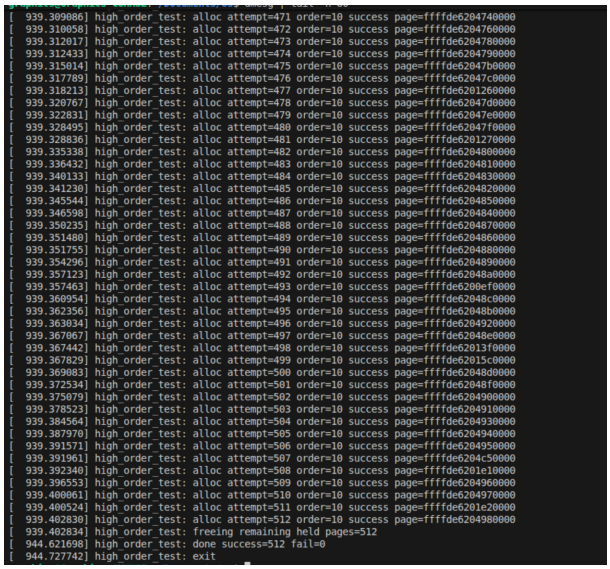

Fig. 6. Kernel log excerpt showing repeated high-order allocation activity from `high_order_test`.

G. Why `fail = 0` Is Still a Strong Result

At first glance, one might think that zero failures means the workload was not strong enough. However, this interpretation would ignore the meaning of slowpath and compaction. The correct reading is:

- the system was under real pressure, because `slowpath`, `compact`, and `adaptive` were all nonzero;
- compaction and recovery were still sufficient to keep the final allocation success rate at 100%.

This is actually a better outcome for our project. It shows not only that the allocator encountered difficulty, but also that the recovery path and adaptive policy were able to preserve high-order allocation success under pressure.

H. Performance-Oriented Analysis

Although we did not collect hardware-level profiling data such as cache misses or lock-contention traces, we can still analyze the observed behavior using kernel allocator semantics.

1) Why General Stress Often Failed to Trigger Adaptive Behavior: General-purpose workloads such as `stress-ng` primarily create broad memory pressure. They increase total allocation traffic, especially lower-order traffic, but they do not necessarily exhaust large contiguous free blocks. Therefore request volume increases and allocator state changes, but high-order fast-path success may still remain possible. As a result, compaction and adaptive triggering are inconsistent under such workloads.

2) Why `fragment_stress` Changed State but Not Behavior: The user-space fragmentation workload created holes and visibly altered free-page distribution. However, in our VM environment, this was still not enough to push the allocator into a high-order crisis. The most likely reason is that fragmentation existed, but not at a level that prevented the remaining large blocks from satisfying requests. This explains why `buddyinfo` changed but `slowpath` remained zero.

3) Why the Final Kernel Module Worked: The final kernel module worked because it targeted the exact vulnerable point of the allocator:

- repeated high-order requests,
- sustained retention of successful allocations,
- reduced opportunity for immediate reuse of the same large free blocks.

This forced the allocator to search deeper, enter slowpath, and attempt compaction. In other words, the final workload was not simply "more pressure"; it was the right kind of pressure.

4) Expected Overhead Sources: The main overhead sources introduced by our modification are:

- atomic counter updates on allocation-related paths,
- additional fragmentation checks during slowpath,
- potentially more aggressive compaction in eligible cases.

The counter updates are expected to introduce small constant overhead. The more significant cost comes from compaction itself, which already exists in the original kernel path and becomes more aggressive only under fragmentation pressure. Therefore, our policy attempts to concentrate extra work only in the cases that are already expensive.

Table IV summarizes the fields exported by `/proc/buddy_monitor`, and Table V compares the workloads used in the project.

TABLE IV
MONITORED FIELDS IN `/PROC/BUDDY_MONITOR`

Field	Meaning
req	allocation request count
success	successful allocations
fail	failed allocations
slowpath	slowpath entries
compact	compaction attempts
adaptive	adaptive policy triggers
frag_pct	recorded fragmentation percentage

TABLE V
WORKLOAD COMPARISON

Workload	Main purpose	slowpath	compact	adaptive	Main observation
`stress-ng --vm`	general pressure	low/inconsistent	low/inconsistent	usually 0	broad pressure but not targeted
`fragment_stress`	shape fragmentation	0	0	0	state changed, behavior not triggered
early `high_order_test`	high-order requests	0	0	0	still enough contiguous memory
final aggressive `high_order_test`	sustained order-10 pressure	86	86	86	target behavior successfully triggered

VI. CONCLUSION

In this project, we studied the Linux buddy allocator from both an observational and an adaptive optimization perspective.

First, we implemented a runtime observation framework by modifying `mm/page_alloc.c` and exporting allocator statistics through `/proc/buddy_monitor`. This allowed us to monitor request counts, success and failure counts, slowpath entries, compaction attempts, adaptive triggers, and recorded fragmentation percentage directly in a running system.

Second, we implemented an adaptive compaction policy that reacts to higher-order allocation pressure and measured fragmentation. Instead of making compaction universally aggressive, the policy raises compaction priority only when the allocator is already under pressure and fragmentation is severe enough.

Third, we designed a staged experimental methodology using general memory pressure with `stress-ng`, a custom user-space fragmentation workload, and a custom kernel high-order allocation module. These workloads showed that broad memory pressure alone is not sufficient to consistently trigger the target allocator path. The final aggressive `order 10` kernel module experiment produced the clearest evidence: `slowpath = 86`, `compact = 86`, and `adaptive = 86`, while all 897 requests still succeeded. This demonstrates that our modified policy was not only implemented correctly, but also executed under the intended high-order pressure scenario.

Overall, the project demonstrates successful kernel instrumentation, kernel modification, workload design, and experimental analysis. It also highlights an important systems lesson: meaningful allocator evaluation requires not just more load, but the right kind of workload that stresses the exact mechanism being studied.

REFERENCES

[1] Linux Kernel Source Tree, version 5.15.180, `mm/page_alloc.c`.
[2] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ, USA: Prentice Hall, 2004.
[3] C. Packham, "stress-ng," https://kernel.ubuntu.com/~cking/stress-ng/.
[4] Linux manual pages, `proc(5)` and `seq_file` related kernel interfaces.