An Evolutionary Study of Linux Memory Management for Fun and Profit

Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan, *Georgia Institute of Technology*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/huang

This paper is included in the Proceedings of the
2016 USENIX Annual Technical Conference (USENIX ATC '16).
June 22–24, 2016 • Denver, CO, USA
978-1-931971-30-0

An Evolutionary Study of Linux Memory Management for Fun and Profit

Jian Huang, Moinuddin K. Qureshi, Karsten Schwan

Georgia Institute of Technology

Abstract

We present a comprehensive and quantitative study on the development of the Linux memory manager. The study examines 4587 committed patches over the last five years (2009-2015) since Linux version 2.6.32. Insights derived from this study concern the development process of the virtual memory system, including its patch distribution and patterns, and techniques for memory optimizations and semantics. Specifically, we find that the changes to memory manager are highly centralized around the key functionalities, such as memory allocator, page fault handler and memory resource controller. The well-developed memory manager still suffers from increasing number of bugs unexpectedly. And the memory optimizations mainly focus on data structures, memory policies and fast path. To the best of our knowledge, this is the first such study on the virtual memory system.

1 Introduction

The virtual memory system has a long history. It was first proposed and implemented in face of memory scarcity in 1950s [16, 27, 28, 81, 86]. With this technique, the main memory seen by programs can be extended beyond its physical constraints, and the memory can be multiplexed for multiple programs. Over the past several decades, the virtual memory has been developing into a mature and core kernel subsystem, the components and features it has today are far more than the basic functionalities (i.e., page mapping, memory protection and sharing) when it was developed [22].

However, today's virtual memory system still suffers from faults, suboptimal and unpredictable performance, and increasing complexity for development [10, 41, 62, 70, 82]. On the other hand, the in-memory and big memory systems are becoming pervasive today [57, 91], which drives developers to re-examine the design and implementation of the virtual memory system. A quantitative study of the virtual memory system's development process is necessary as developers move forward to next steps. The insights derived from the study can help developers build more reliable and efficient memory management systems and associated debugging tools.

In this paper, we perform a comprehensive study of the open-source Linux memory manager (*mm*). We examine the patches committed over the last five years from 2009 to 2015. The study covers 4587 patches across Linux versions from 2.6.32.1 to 4.0-rc4. We manually label each patch after carefully checking the patch, its descriptions, and follow-up discussions posted by developers. To further understand patch distribution over memory semantics, we build a tool called *MChecker* to identify the changes to the key functions in *mm*. *MChecker* matches the patches with the source code to track the hot functions that have been updated intensively.

We first investigate the overall patterns of the examined patches. We observe that the code base of Linux *mm* has increased by 1.6x over the last five years, and these code changes are mainly caused by bug fixes (33.8%), code maintenance (27.8%), system optimizations (27.4%) and new features (11.0%). More interestingly, we find that 80% of the *mm* patches are committed to 25% of the source code, indicating that its updates are highly concentrated. Such an observation discloses the targeted code regions for our study and future development on virtual memory system.

Furthermore, we examine the bugs in Linux *mm*. We identify five types of bugs: *memory error*, *checking*, *concurrency*, *logic* and *programming*. These bugs are mainly located in the functional components of *memory allocation*, *virtual memory management* and *garbage collection*. Specifically, *mm* is suffering from more concurrency and logic bugs due to its complicated page state management. For example, the memory leaks are mainly caused by the incorrect settings of page states rather than non-freed pages; a significant number of logical incorrectnesses are caused by missing checks on page states.

We further investigate the system optimization patches in *mm*. We identify three major sources: *data structure*, *memory policy* and *fast path*. (1) For *data structure*, we find that 76.2% of patches are committed for software overhead reduction, and 23.8% of them contributed to scalability improvement, across the four popular data structures: *radix tree*, *red-black tree*, *bitmap* and *list*, and their derived structures. (2) For *policy* patches, we find that most of them are concentrated around five design trade-offs: *lazy vs. non-lazy*, *local vs. global*, *sync vs. async*, *latency vs. throughput* and *fairness vs. performance*. For example, OS developers can alleviate overhead caused by expensive operations (e.g., memory com-

Table 1: A brief summary of the Linux *mm* patch study.

	Summary	Insights/Implications
Overview	4 types of patches (i.e., *bug*, *optimization*, *new feature*, *code maintenance*) were committed to 8 major *mm* components (e.g., memory allocation, resource controller and virtual memory management). The patch distribution is highly centralized (§ 3).	(1) the identified 13 hot files from the massive *mm* source code (about 90 files) unveil the focus of the recent *mm* development; (2) with these knowledge, developers can narrow their focus to pinpoint *mm* problems more effectively.
Bug	5 types of bugs (i.e., *checking*, *concurrency*, *logic*, *memory error* and *programming*) have various patterns: *null pointer* and *page alignment* are the popular memory errors; *checking* and *logic* bugs are pervasive due to the complicated page state management (§ 4).	(1) a set of unified and fine-grained page states can be defined to reduce the effort on page tracking for kernel developers; (2) the page state machine should be combined with lock schemes to avoid unnecessary locks; (3) a formal, machine-checked verification framework for *mm* is needed.
Optimization	4 types of *data structure* (i.e., *radix tree*, *red-black tree*, *bitmap* and *list*) optimizations on software overhead reduction and scalability improvement (§ 5.1).	(1) careful examination on nested data structures is necessary to avoid the consequential side effects as we adjust data structures; (2) the internal scalability inside system calls is not well exploited yet.
	Memory *policies* are tackling 5 design trade-offs: *lazy vs. non-lazy*, *local vs. global*, *sync vs. async*, *latency vs. throughput* and *fairness vs. performance* (§ 5.2).	(1) *lazy* policy is preferred as *mm* interacts with fast devices like processor cache, while *async* policy is mostly used for the interaction with slow devices like disk; (2) a large amount of latency-related patches suggest that *mm* profilers are desired to identify more latency-critical operations;
	Fast path has 8 types of approaches: *code reduction*, *lockless optimization*, *new function*, *state caching*, *inline*, *code shifting*, *group execution* and *optimistic barrier*. (§ 5.3).	(1) alleviating redundant functions and reducing lock contentions are the two major contributors for reducing software overhead; (2) these techniques can be generalized and applied in other software systems.
Semantic	35 key functionalities are identified in 13 hot files in Linux *mm*. A majority (75.6%) of them absorb much more patches on *bug fix* and *optimization*. Certain patch pattern is seen for each functionality (§ 6).	(1) the well-developed memory allocators still have tremendous *checking* and *lock* issues due to the increasing complexity of page state management; (2) the fault handler in *mm* is buggy, especially for the cases of out of memory and allocation failures; (3) the patch patterns on memory policy suggest that a policy verification and validation framework is in pressing need;

paction, TLB flush) with lazy policies, but associated checking mechanism has to be implemented to guarantee the program logic is not violated (more patches on this part are committed than the optimization patch itself). (3) We identify eight types of approaches (Table 6) for *fast path* optimization in Linux *mm*, such as *code reduction*, *lockless optimization* and *state caching*.

With the *MChecker* tool, we study the patch distribution over the core *mm* functions to understand the various patterns on memory semantics. Taking the memory policy as example, we categorize it in two types: *policy definition* and *enforcement*. We find that *policy definition* has more issues than *enforcement*. And 30% of the patches were addressing the issues caused by missing checks (e.g., whether page is dirty), missing one check fails the policy enforcement.

We briefly summarize the key findings and present the outline of the paper in Table 1. We discuss the related work in § 7 and conclude the paper in § 8. In the following, we describe the methodologies used in our study.

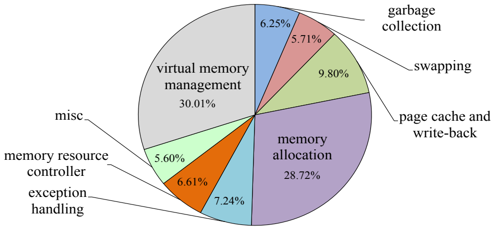


Figure 1: Component breakdown of memory manager in Linux version 4.0-rc4, in terms of lines of codes.

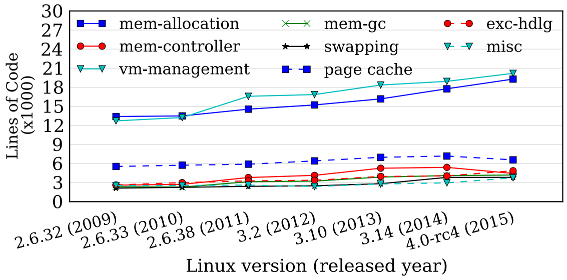


Figure 2: The change in *mm* code in terms of LoC.

2 Methodology

In our study, we target at open-source Linux memory managers, as they provide much more resources (e.g., source code, patches, online discussions) for such a study compared to commercial operating systems. We only select the stable versions that are still supported by the open-source community. The selected Linux kernel versions range from 2.6.32 (released on December 2, 2009) to 4.0-rc4 (released on March 15, 2015), and the time difference between the release dates of two successive versions is 12 months on average. It is noted that Linux 2.6.32 is the oldest version that is still supported. Thus, we believe our study over the past five years represents the latest development trend of the Linux *mm*.

In order to have a comprehensive study of the selected virtual memory system, we manually examine most of the committed patches to Linux memory manager (*root/mm*) following the approaches described in

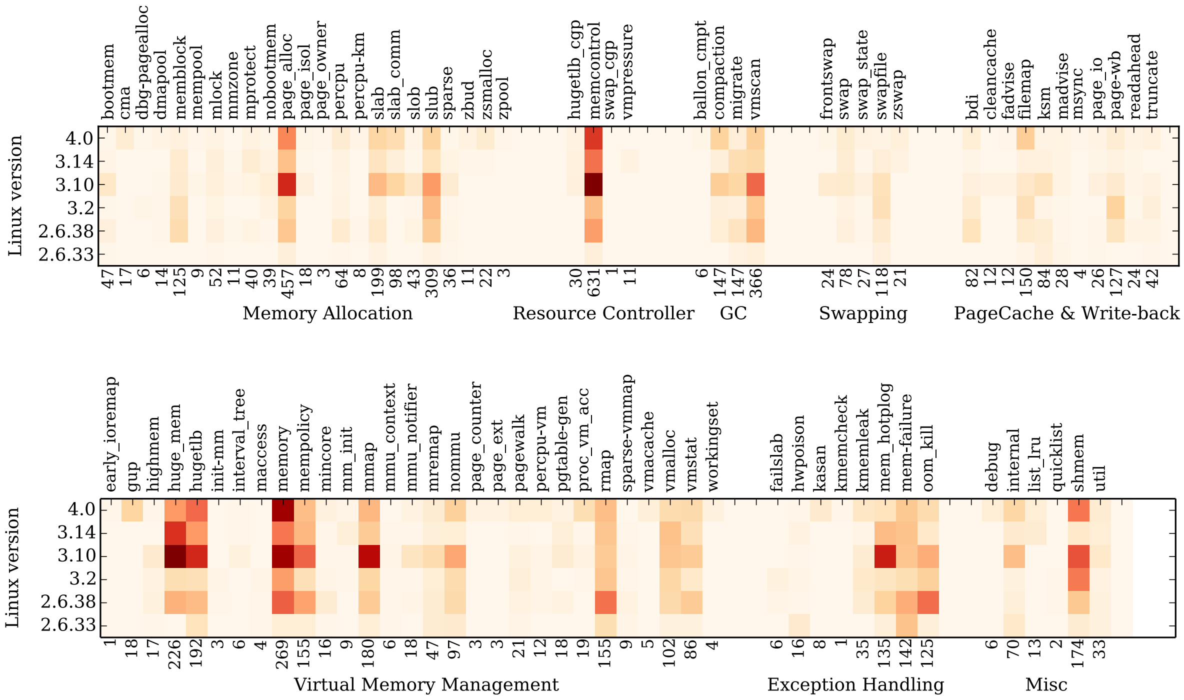


Figure 3: The heat map of patch distribution in each component of Linux memory management. Each block represents the patches committed to the current version since the last stable version. The darker the color, the more patches applied. The number below the bar indicates the total number of committed patches applied to the given file.

[23, 26, 37]. Since December 2, 2009, there are totally 5358 patches relevant to Linux *mm* reported in the patch repositories of Linux kernel. After excluding the duplicated and invalid patches, we examine 4587 patches (85.6% of the total patches). To precisely analyze and categorize each examined patch, we manually tag each patch with appropriate labels after checking the patch, its descriptions, corresponding source code changes, and follow-up discussions posted by developers. The labels include *LinuxVersion*, *CommitTime*, *SrcFile*, *MMComponent*, *PatchType*, *Consequence*, *Keywords*, *Causes*, *Note* and etc. For the patch that belongs to several categories, it will be classified into all the respective categories and studied from different viewpoints. We place all the examined patches into our patch database *MPatch* for patch classification and statistical analysis.

To facilitate our analysis, we break down the Linux *mm* into 8 components according to the functionalities (see Figure 1). We match the examined patches with each component. Taking the Linux version 4.0-rc4 for example, we use the SLOCCount tool [74] to count the line of codes (LoC) in each component. Figure 1 shows the fraction of code serving to accomplish specific functionalities in *mm*. The two largest contributors to Linux *mm* code are memory allocation (28.7%) and virtual memory management (30.0%). This is expected with considering their core functions in virtual memory system. We will discuss how the patches are distributed among these eight components in detail in the following sections.

To further analyze the examined patches, we build a patch analysis tool called *MChecker* to understand the memory semantics by mining the relationships between patches and the key functions in the 'hot' files of Linux *mm*. This will be discussed in detail in § 6.

3 Virtual Memory Evolution

Linux *mm* is constantly updated like other subsystems (e.g., file systems, device drivers) in the Linux kernel. However, few quantitative studies have been done on the Linux *mm*. In our study, we conduct the virtual memory study from the oldest stable version 2.6.32 until the version 4.0, demonstrating what *mm* developers concentrated on over the past five years.

3.1 How is the *mm* code changed?

Taking the Linux 2.6.32 version as the baseline, we examine the source lines of code changes in different Linux *mm* components.We obtain the LoC across different *mm* component in total 7 versions using SLOCCout.

As shown in Figure 2, the LoC is increased in all the *mm* components across successive years compared with the baseline 2.6.32. Overall, Linux *mm* code base has increased by 1.6x over the last five years. *Memory allocation* and *virtual memory management* are the two major components in *mm*, the updates to the two components constantly occupy a large portion of the overall patches.

Understanding the code changes is important for us to pinpoint how the Linux *mm* is evolved. More detailed analysis is given on where and why the *mm* code has been changing in the following.

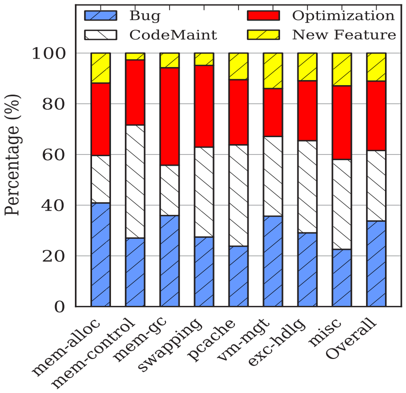


Figure 4: Patch overview. It shows the patch distribution according to general types including *bug*, *code maintenance*, *improvement* and *new feature*.

3.2 Where is the *mm* code changing?

The patches applied to Linux *mm* record all the changes to its code base and provide the evidences showing how one version transforms to the next. Figure 3 demonstrates the patch distribution among all the components in Linux *mm*. One patch may be applied to several files in *mm*, we count it to all the involved files. The average source LoC changed in each patch is 62, it is much less than the source LoC in feature patches. For example, the compressed swap caching (*zswap*) was introduced in 2013 [84], and a new file named `zswap.c` with 943 LoC was added in the code base of Linux *mm*.

We identify several interesting findings via the heat map. First, the patches are concentrated around only a few files in each component (see the darker column and blocks in the heat map of Figure 3). About 80% of the patches were applied to the 25% of the source code. These hot files generally represent the core functions of the corresponding component. Second, the patches applied to these hot files are much more than other files. For instance, the number of patches relevant to *huge_mem* in *virtual memory management* component is about 12x more than that of 'cold' files. Third, for these hot files, most of them are constantly updated along the Linux evolution from one version to the next. Typical examples include the *memcontrol* in memory resource controller, the *memory* in virtual memory management.

It is understandable that more patches are committed between Linux 3.2 and 3.10 compared to other intervals, as the time between the two versions is 19 months which is longer than the average time interval (12 months).

3.3 Why is the *mm* code changed?

We identify that the *mm* source code changes come from four sources: *new feature*, *bug fixes*, *optimization* and *code maintenance*. We classify the patches into these four categories, and examine how each category contributes to the evolution of Linux *mm*.

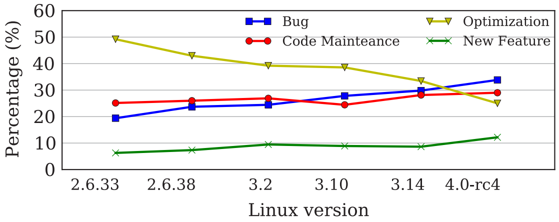


Figure 5: The changes of patch distribution along the Linux *mm* evolution, taking Linux 2.6.32 as the baseline.

Figure 4 shows the patch distributions among the 8 components. Overall, 33.8% of the patches are applied for bug fixes, 27.8% of the patches are relevant to code maintenance, 27.4% are for system optimizations, and 11.0% are new features. Common sense suggests that virtual memory system has been developed into a mature system, our findings reveal that the bug fixes are still the main thread of patch contributions.

Furthermore, we examine how the four types of patches changed over time. As shown in Figure 5, we find that *bug* patches are increasing steadily, indicating more bugs are expected in Linux *mm* as the complexity of its code base is increasing (see Figure 2). The percentage of *code maintenance* and *new feature* patches keep at a constant level in general, but a slightly increase in *new feature* patches is seen recently. Perhaps most interestingly, *optimization* patches are decreasing over time, which can be expected as Linux *mm* becomes more adapted to current systems (e.g., multi-core processors).

Summary: Linux *mm* is being actively updated, The code changes are highly concentrated around its key functionalities: 80% of the patches were committed to the 25% of the source code.

4 Memory System Bugs

In this section, we examine the *bug* patches in detail to understand their patterns and consequences.

4.1 What are *mm* bugs?

With the tagging of these patches, we classify the *bug* patches into 5 general types: *memory error (MErr)*, *checking*, *concurrency*, *logic* and *programming*. Each general type is further broken down into multiple subtypes according to their causes, as shown in Table 2. Like the systems such as *file systems* [37] and *device drivers* [29], many bugs are general software bugs (e.g., *programming* bugs). In this paper, we are more interested in memory-related bugs, for example the *alignment* bugs in *MErr*, and the *logic* bugs (§ 4.2).

4.2 How *mm* bugs are distributed?

The heat map of Linux *mm* bug distribution among its eight components is shown in Figure 6. More bugs lie in

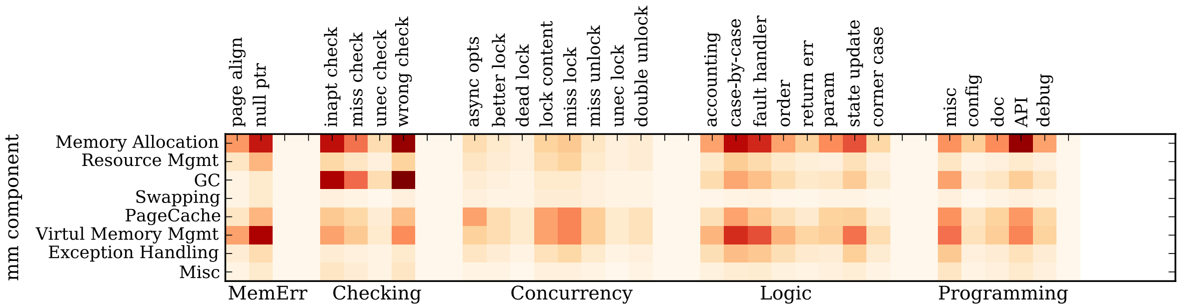


Figure 6: Bug distribution among Linux *mm* components.

Table 2: Classification of *bug* patches.

	Sub-type	**Description**
MErr	alignment	data/page alignment and padding.
	null pointer	refer to an invalid object.
Checking	inapt check	inappropriate check.
	miss check	check is required.
	unec check	unnecessary check.
	wrong check	check conditions are incorrect.
Concurrency	async opts	faults due to async operations
	better lock	lock is implemented in better way.
	dead lock	two or more threads are blocked.
	lock contention	concurrently access to shared locks or data structures.
	miss lock	lock is required.
	miss unlock	the unlock is missed.
	unec lock	unnecessary lock.
	double unlock	unlock twice.
Logic	accounting	error in collecting statistics.
	case-by-case	bug fix requires specific knowledge.
	fault handler	error and exception handling.
	order	the order of execution is violated.
	return err	return code is not correct.
	parameter	misuse of parameters.
	state update	issues in updating state and data structures.
	corner case	uncovered cases in implementations.
Programming	configuration	wrong/missed configuration.
	document	comments & docs for functions.
	API	issues caused by interface changes.
	debug	issues happened in debugging.
	misc	any other programming issues

the three major components *memory allocation*, *virtual memory management*, and *GC*, which matches with the patch distribution as shown in Figure 3. More specifically, we identify several interesting findings in *mm*:

Memory Error (MErr): We find that *null pointer* dereferences (e.g., [45, 67, 76]) are the most common bugs because of the missing validations of pointers before using them in *mm*. These bugs happened even in *mm*'s core functions such as slub, which is unexpected. The alignment of data structures and pages are important factors in *mm* optimizations, however bugs frequently happen at boundary checking and calculations for padding (e.g., [5, 6]). As they usually involve many shift operations, validating the correctness of the bit manipulations is necessary.

Checking: As *mm* involves many state checking operations, especially in *memory allocation* and *garbage collection* (GC) components. The *checking* bugs appear frequently due to inappropriate and incorrect checking, for instance, the GC has to check if a page is used or not before the page migration is issued; *free_bootmem_core* may free wrong pages from other nodes in NUMA without correct boundary checking [47].

Concurrency: We find that more *miss lock* and *lock contention* bugs appeared in *virtual memory management* due to the complicated page states, and more efforts are required for kernel developers to track the page states. In addition, the page state machine can be combined with lock schemes to avoid unnecessary locks, for instance, when kernel pages are charged or uncharged, the *page_cgroup* lock is unnecessary as the procedure has been serialized (e.g., [40]).

Logic: We identify three important *logic* bugs: *case-by-case*, *state update* and *fault handler*. For the first two types, they may not stall the system or generate exceptions immediately, but they make the system execute in unexpected workflow or states, resulting in incorrect states or *runtime error* eventually. Fixing these bugs often require domain specific knowledge. For example, when *shmem* intends to replace a swap cache page, the original implementation calls *cgroup* migration without *lrucare* based on the incorrect assumption that the page is not on the LRU list. As for *fault handler* bugs, many of them were caused by lack of or inappropriate implementation of exception handling (e.g., [56, 77, 85]).

There is still a long way to have a bug-free virtual memory system. It is much hard for formal proof to verify the correctness of *concurrency* events [31], and few previous work has the formal, machine-checked verification for virtual memory system specifically.

4.3 What are the *mm* bug consequences?

We further examine the consequences of *mm* bugs to understand how serious they are. We classify the bug consequences into 7 types with reference to the classification in [37]. Figure 7 shows that *logic* bugs lead to wrong behaviors and runtime errors with higher chances. *Concurrency* bugs are more likely to make system crash or hang, since they often produce *null pointers* and *deadlocks* if

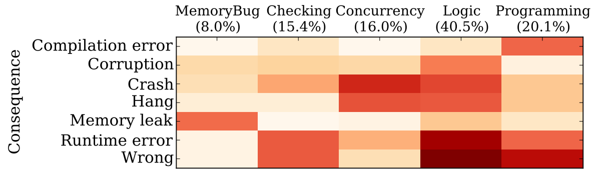


Figure 7: Bug consequence.

the shared data structures are not maintained properly.

Moreover, we find that *checking* bugs are also the major contributors to the wrong behaviors and runtime errors in Linux *mm*. More interestingly, we find that the memory leaks are mainly caused by *MErr* and *logic* bugs. It is noted that most of the memory leaks in *mm* are not caused by not-freed memory, they are mostly caused by the *accounting* (e.g., the unused page is not counted as free page) and *fault handler* bugs (e.g., pages are not reclaimed when fault happens). For *programming* bugs that mainly cause compilation errors, runtime error and wrong behaviors, they are easier to be fixed compared with other types of bugs.

> **Summary:** Memory leaks in *mm* were mainly caused by the *accounting* bugs and inappropriate implementation of *fault handler* (e.g., page fault), instead of non-freed memory. The complex states of pages complicate the implementation of the *checking* and *locking* mechanism, which requires large effort for kernel developers to track the correct states.

5 Memory Optimizations

As discussed in § 3, *optimization* (27.4% of total patches) is one of the major contributors to the code changes in Linux *mm*. We identify several sources that contributed to the optimizations in *mm*: *data structure*, *policy* and *fast path*. In this section, we will explain how these optimizations were performed in improving Linux *mm*.

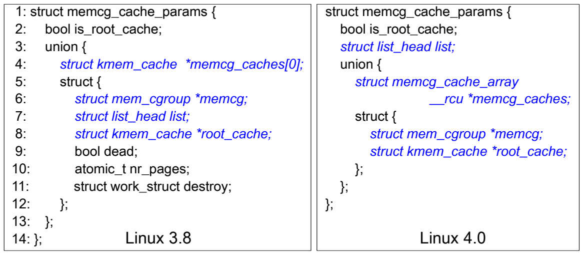
5.1 Memory Data Structures

In virtual memory system, data structure is one the critical factors for its efficiency [10, 15, 20]. Likewise, the data structures in Linux *mm* are constantly tuned to reduce software overheads, and specialized data structures are leveraged for special purposes such as page lookup and memory allocation.

5.1.1 What are the common data structures?

We identify four popular data structures in Linux *mm*: *radix tree*, *red-black tree*, *bitmap* and *list*, according to their relevant patch distributions.

Radix tree [65, 66] is typically used within *address_space* structure in Linux *mm* for tracking in-core pages for page caches, because of its storage efficiency for sparse trees. **Red-black tree** [69] such as the one in *vm_area_struct* can perform lookups in logarithmic time, and its insert and delete operation can be finished in bounded time. It is used to track *VMA*s. **Bitmap** is usually used to index pages in RAM, which involves bit manipulation frequently. Besides these specific data structures, other data structures such as **list** are widely used in Linux *mm*. Most of the recent patches are related to their derived data structures, such as LRU list which is used by multiple *mm* components to track page access frequency.



```
struct memcg_cache_params {
  bool is_root_cache;
  union {
    struct kmem_cache  *memcg_caches[0];
    struct {
      struct mem_cgroup *memcg;
      struct list_head list;
      struct kmem_cache *root_cache;
      bool dead;
      atomic_t nr_pages;
      struct work_struct destroy;
    };
  };
};
```

```
struct memcg_cache_params {
  bool is_root_cache;
  struct list_head list;
  union {
    struct memcg_cache_array
            __rcu *memcg_caches;
    struct {
      struct mem_cgroup *memcg;
      struct kmem_cache *root_cache;
    };
  };
};
```

Figure 8: Comparison of *memcg_cache_params* structure in Linux version 3.8 and 4.0.

Table 3: Typical examples of approaches to reduce software overhead of different data structures.

Type	**Overhead Source**	**Optimization Example**
radix tree	Tree walking	Provide hints, cache intermediate states [48]
	Linear search	Add bit-optimized iterator [64]
rb tree	Tree walking	Optimized tree walking [1, 44]
	Lock contention	Batch lookup [11]
	Balancing tree	Reduce lazy operations [52]
list	List search	Limit list length [43]
	Lock contention	Add per-node LRU list [35]
	Storage overhead	Dynamic allocation

5.1.2 How are data structures optimized?

We identify that the optimization of *mm* data structures mostly focuses on two aspects: software overhead reduction (76.2%) and scalability improvement (23.8%).

Reducing software overhead. In Linux *mm*, we find that the software overhead on these data structures mainly come from the following sources: *tree walk*, *tree balance*, *lock contention* and *storage cost*. A variety of approaches have been applied to address these issues as shown in Table 3. For instance, to reduce lock contentions, multiple lookups can be performed in batch once a lock is acquired.

Reorganizing data structures is another approach that usually adopted to improve memory efficiency. However, this approach may introduce additional overhead that offsets its benefits. Beyond our expectation, a significant portion of patches were applied to avoid the extra overhead caused by nested data structures. An interesting example is shown in Figure 8. The structure *memcg_cache_param* in Linux version 4.0 shrinks compared to its initial design in version 3.8. However, the saved memory does not justify, as the pointer dereference in its correlated structure *kmem_cache* may cause extra cache line access. Thus, the pointer was replaced with embedded variable [73]. As we adjust data structures in virtual memory system, their referenced data structures are often overlooked, producing side effects.

Improving scalability for data structures. Scalability issue is another major aspect of data structure optimizations. Most of the scalability issues are caused by locking for atomic access to shared data structures. We find that a main thread of the applied patches is to decentralize the data structures and replace the shared counterparts with per-core, per-node and per-device approaches.

With the increasing memory size and core counts, the scalability issues become more prominent [10, 90], appropriate usage of data structures is critical for both performance and memory efficiency, such as recent work [11] suggested that replacing red-black tree with radix tree to track non-overlapping virtual memory regions for *mmap* and *munmap* provides better scalability. Furthermore, the system support for new memory technologies like persistent memory bring new scalability challenges, the dramatically increased physical memory capacity generates large pressure on memory management, e.g., a 1 TB of persistent memory with 4 KB page size requires 256 million page structures [14, 61].

To scale OS kernels, Clements et al. [12] proposed a commutativity tool to guide the implementation of high-level APIs, however it cannot expose the internal scalability issues (e.g., global vs. local data structure) inside the system calls. Our findings on *mm* data structures suggest that it is necessary to build tools to check the bottlenecks introduced by global and shared data structures.

Summary: The software overhead and scalability issues caused by data structures remain big concerns for OS developers: more efforts on system-wide optimization for nested data structures, and the internal scalability inside system calls are required.

5.2 Policy Design: Tackling Trade-offs

Memory is one of the most desired yet constrained resource in computer systems, multiple design trade-offs have to be made to fully utilize the resource and to improve performance. We find that a majority of the patches relevant to the policy design are concentrated on tackling these trade-offs. Through the patch study, we expect to learn from the lessons with policy designs and implementations conducted by OS developers.

5.2.1 What are the trade-offs?

Based on our patch study, we summarize the trade-offs that OS designers have frequently tackled in Table 4, and also present a case study for each of them. The software overhead caused by expensive operations, such as memory compaction, page migration and TLB flush, can be significantly alleviated using asynchronous or lazy policies. However, such benefit is not free because they complicate the program logic, leading into serious runtime errors like data inconsistency. We will present how the policy design decisions were made in Linux *mm*, with a focus on the new class of *mm* optimizations.

Table 4: Classification of typical design choices in Linux *mm* based on the analysis of *optimization* patches.

Trade-off	%	Case Study
Latency Vs. Throughput	10.9	disk access upon page fault and swapping.
Synchronous Vs. Asynchronous	22.3	With asynchronous method, *mm* can avoid delays while executing expensive operations like swapping, compaction.
Lazy Vs. Non-lazy	15.6	Expensive operations (e.g., TLB flush, page migration) can be executed in batch.
Local Vs. Global	33.1	Maintaining per-process variables improves scalability, but it increases storage overhead, e.g., slub vs. slab allocator.
Fairness Vs. Performance	18.1	Fairness guarantee when memory is shared among multiple processes.

Table 5: Examples of applying lazy policy in Linux *mm*.

Functionality	Example
vmalloc	lazy TLB flush, lazy unmapping
mempolicy	lazy page migration between nodes
huge_memory	lazy huge zero page allocation
frontswap	lazy backend initialization
cleancache	lazy backend registration
backing-dev	lazy inode update on disk

5.2.2 How are the policy decisions made in *mm*?

Latency matters in memory manager. This trade-off of latency vs. throughput centers around the *page cache and write-back* component in *mm*. The I/O requests are issued in batch and served in a disk-friendly order to exploit the full bandwidth of disks for high throughput, but it may increase I/O latency. For instance, the original design of *readahead* component favors sequential access for higher I/O throughput, making the average latency of random reads disproportionately penalized [68]. A general approach on decision-making for this trade-off is to prioritize the dominate workloads patterns, so systems pay little or acceptable cost on its downsides.

More interestingly, we identify 137 patches committed specially for reducing the latencies of *mm* operations (e.g., page allocation, vma lookup, page table scanning). As in-memory systems are becoming pervasive and latency matters to today's application services, it is worthwhile to build *mm* profilers or test framework to identify more latency-critical operations.

Async is popular, but be careful to its faults. Asynchronous operations are widely used to hide expensive operations in *mm*. For example, async compaction was

Table 6: Classification of approaches for fast path optimization in Linux *mm*.

Type	%	Description	Example
Code Reduction	34.1	Simplify the fast path logic and reduce redundant codes.	Avoid redundant `get/put_page` in `munlock_vma_range` as pages will not be referred anymore [50].
Lockless Optimization	27.3	Reduce the usage of lock and atomic operations.	Lockless memory allocator in SLUB [34, 36].
New Function	11.6	Improve with new *mm* functionality.	New SLUB fast paths are implemented for `slab_alloc/free` with `cmpxchg_local` [75].
State Caching	8.2	Cache the states to avoid expensive functions.	Pre-calculate the number of online nodes instead of always calling expensive `num_online_nodes` [59].
Inline	6.4	Inline simple functions in fast path.	Inline `buffered_rmqueue` [58].
Code Shifting	4.7	Move unfrequently executed code from fast path to slow path.	In SLUB allocator, slow path executes the irq enable/disable handlers, fast path will execute them only at fallback [78].
Group Execution	4.1	Avoid calling the same function repeatedly.	Using `pte_walk` to avoid the repeated full page table translation and locks in `munlock_vma_pages_range` [49].
Optimistic Barrier	3.6	Optimistically skip or reduce the number of barriers, and re-execute the logic once false failure is detected.	Using only read barriers in `get/put_mems_allowed` to accelerate page allocation [13].

introduced to reduce the overhead caused by expensive memory compaction; the expensive *trim* command [80] in SSDs should be issued in parallel with other I/O operations before actual writes happened due to its long latency. We find that the common issues in implementing async mechanisms located in their fault handlers for exceptions (e.g., early termination [46]).

Trying lazy policy to alleviate expensive operations. The key idea of lazy policy is to delay several expensive operations, and batch them into a single operation or system call if semantics are allowed. Table 5 shows a set of cases that have leveraged lazy policy to reduce the frequency of expensive operations. In contrast to *async* policy used usually as *mm* interacts with slow devices, *lazy* policy is more beneficial when *mm* components interact with fast devices (e.g., CPU, processor cache, TLB) according to our patch studies.

Since lazy policy may change the execution order of subsequent functions, which would make systems in unexpected states temporarily, careful examination should be conducted as we decide whether a specific function should be delayed or not. For instance, the operation of flushing virtual cache on *vunmap* in *pcpu_unmap* cannot be deferred as the corresponding page will be returned to page allocator, while TLB flushing can be delayed as the corresponding *vmalloc* function can handle it lazily [18].

Decentralizing global structures for better scalability. As seen in our patch study, more per-node, per-cpu variables are replacing their global counterparts to improve the system scalability. For example, new dynamic per-cpu allocator was introduced to avoid the lock contention involved in memory allocations. This approach has also been widely adopted in other subsystems such as device drivers, CPU scheduler and file systems.

Memory resource scheduling for fairness and performance. The trade-off between fairness and performance is a well-known yet hard problem. In Linux *mm*, we find that this type of patches mainly concentrated on the *memory allocation and reclamation*. In page allocation, round-robin algorithm is used to guarantee *zone fairness*. However, this algorithm did not consider the latency disparity across zones, resulting in remote memory reference and performance regression. During page reclamation, the allocator reclaimed page in LRU order which can only provide the fairness for low order pages but not for pages at higher order, which could penalize the performance of the applications (e.g., network adapter) that desire high-order pages [38].

Summary: Most of the *policy* patches are tackling five types of trade-offs. The experiences with *mm* development provide us the hints on how and where each policy would be leveraged in practice.

5.3 Fast Path

To further reduce the software overhead, another optimization approach is to accelerate the commonly executed codes, which is named as *fast path*. We identify that Linux *mm* maintains fast paths in most of its key components, such as *memory allocation*, *memory resource controller*, and *virtual memory management*. These fast paths are carefully and frequently re-examined in every version of Linux kernels, contributing many patches to the code base of Linux *mm*. We study these patches to understand what are the common techniques leveraged in fast path optimizations.

Based on our patch study, we categorize the techniques used for fast path in Linux *mm* into eight types as described in Table 6. We find that *code reduction* and *lockless optimization* are the most commonly used techniques for fast path, which contributed 61.4% of the fast path related patches. As case studies, we list a set of patches for these types in Table 6. For instance, on the fast path for the allocation and deallocation of page table pages, there are two costly operations: finding a zeroed page and maintaining states of a slab buffer. To reduce these overheads, new component called *quicklist* was introduced to replace the allocation logic as it touches less cache lines and has less overhead of slab management. Another interesting approach is *optimistic barrier*, which is proposed to reduce the synchronous overheads caused by system call like *barrier* and *fence*, e.g., the full memory barriers on both reads and writes are replaced with only read barriers on the fast path, while re-executing the code with slow path when false failure is detected.

Summary: There are 8 types of optimizations for fast path. *Code reduction* and *lockless optimization* are the two most widely used techniques to alleviate redundant functions and reduce lock contention.

6 Memory Semantics

In order to better understand memory semantics, we build a tool named *MChecker* to pinpoint the modified functions in source code by leveraging the information (e.g., which lines of code are changed) provided in patches. *MChecker* will place the located functions under the record of the corresponding patch in *MPatch*. By analyzing the call graphs of specific memory operations, we can identify their core functions. And with the information provided by *MPatch*, we can easily understand what are the common bugs lying in these functions, how these functions were optimized and etc.

In this paper, we analyze the hot files which are evolved with more committed patches (see Figure 3). Due to the space limitation of the paper, we only present the hot functions that were updated intensively in Table 7. Across the 35 major functionalities (the 3rd column in Table 7), 75.6% of them have more patches for *bugs* and *optimization* than those for *code maintenance* and *new feature*, which demonstrates the main thread of contributions from the open-source community.

6.1 Memory Allocation

The memory allocation and deallocation functions in the kernel space are mostly implemented in `page_alloc`, `slab`, `slub` and `slob` files which are the cores of the well-known buddy system. As the default memory allocator, `slub` absorbs 1.6-7.2x more patches than other two allocators. The functions in these hot files can be categorized into *allocation/free*, *create/destroy* and *page/cache management* to fulfill the implementation of the user-space `malloc` and `free` functions. As memory allocators become mature, about half of the relevant patches are concentrated on the page and cache management (e.g., cache initialization within `kmem_cache_init`, object state maintenance within `show_slab_objects`).

We find that the mature memory allocation and deallocation are still suffering from bugs. The *allocation* and *create* functions have more bugs than *free* and *destroy* functions, and these bugs are usually relevant to *checking* and *lock contentions*. The cumbersome checks and lock protections (due to complicated page states) not only increase the possibility of bug occurrence, but also incur increasing software overhead. To reduce such software overhead, an increasing number of improved versions of allocation functions with *fast path* are implemented. For instance, a lockless allocation algorithm based on the atomic operation `this_cpu_compxchg_double` improves the allocation performance significantly [83].

Summary: The mature memory allocators still suffer from serious *checking* and *lock* issues due to the complicated states maintained for memory pages during their life cycles.

6.2 Memory Resource Controller

In memory resource controller, the majority (93.8%) of its patches are committed to the `memcontrol` file. As more resource is available on today's machines, the OS-level resource control named *memory cgroup* was proposed to support resource management and isolation. It is motivated to isolate the memory behavior of a group of tasks from the rest of the system.

To control the usage of memory, *charge/uncharge* functions are used to account the number of pages involved along with the running tasks. We find that 26.2% of the committed patches relate to *concurrency* issues, because of the complicated intrinsic operations in these functionalities. For instance, for a uncharging page, it may involve actions of truncation, reclaim, swapout and migration. Interestingly, many of these issues are caused by missing locks. However, detecting missing locks is more than a software engineering issue, as it requires program semantics (e.g., page states) provided by the virtual memory system. Such an observation may give us the hint that the decoupled memory resource controller should be integrated into the *mm* framework to avoid redundant data structures and software overheads, e.g., *memcontrol* can rely on the existing LRU lists to obtain page information and schedule pages dynamically.

Moreover, we find that fault handlers suffer from a significant portion of bugs, the involved cases include out of memory (OOM) and allocation failure. Similar trends are seen in the *exception handler* component which has two hot files: `memory-failure` and `oom_kill`. Most of them are caused by the inappropriate or wrong handling of memory errors and exceptions (e.g., [8, 19]). We expect these findings would reveal the weakness aspect of Linux *mm* and supply useful test cases to the memory testing and debugging tools like mmtests [42, 53].

Summary: *Concurrency* issues (e.g., missing lock) are the major concern for the memory resource controller development. Our study discloses that the fault handler is still a weak aspect in *mm*.

Table 7: The heat map of patches for the hot functions in *mm*. Functions in each hot file are categorized into sub-types (the 3rd column). The proportion of each functionality is illustrated in the 4th column. The distribution of *bugs* (BUG), *code maintenance* (CODE), *new feature* (FTR) and *optimization* (OPT) is shown across 5-8th columns.

	File	Functionality	%	BUG	CODE	FTR	OPT	Representative Hot Functions
Memory Allocation	page_alloc	Allocation	24.6	8.1	6.8	2.2	7.4	alloc_pages, alloc_pages_slowpath, _rmqueue
		Free	19.4	4.8	8.3	1.3	5.0	free_one_page, free_pages_bulk, free_zone_pagesets
		Page management	56.0	14.1	22.0	4.4	15.5	build_zonelists, zone_init_free_lists, read_page_state
	slab	Create/destroy	11.6	4.1	4.9	1.5	1.2	kmem_cache_create, kmem_cache_destroy
		Allocation/free	29.8	11.6	12.4	2.0	3.8	cache_alloc, kmalloc, kmem_cache_free
		Shrink/grow	6.9	1.7	3.2	0.6	1.4	cache_grow, cache_reap
		Cache management	51.7	8.7	22.2	1.8	19.1	kmem_cache_init, kmem_getpages
	slub	Create/destroy	14.9	5.4	4.5	0.4	4.5	kmem_cache_create, kmem_cache_destroy
		Alloction/free	33.8	9.5	8.1	0.9	15.3	slab_alloc, kmalloc_large_node, kfree, slab_free
		Slub management	51.4	22.1	9.5	4.1	15.8	kmem_cache_init, show_slab_objects
Controller	mmcntr	Charge/uncharge	26.5	9.3	1.0	2.4	13.8	mem_cgroup_do_charge, mem_cgroup_try_charge, mem_kmem_commit_charge
		Cgroup management	73.5	35.4	7.9	7.2	23.0	mem_cgroup_alloc, mem_cgroup_handle_oom
Expt Hndler	failure	Fault handler	75.9	38.9	13.0	9.3	14.8	memory_failure, collect_procs_file
		Hwpoison	24.1	13.0	1.9	3.7	5.6	hwpoison_user_mappings, hwpoison_filter_task
	OOM	Candidate task	53.7	14.8	16.6	7.4	14.8	oom_badness, select_bad_process, oom_unkillable_task
		OOM handler	46.3	24.1	5.6	3.7	13.0	out_of_memory, oom_kill_process
Virtual Memory Management	memory	Page table	26.3	11.6	10.0	1.6	3.1	do_set_pte, pte_alloc, copy_one_pte, zap_pte_range
		Page fault	23.4	8.8	7.3	1.5	5.9	do_fault, handle_mm_fault, do_shared_fault
		Paging	25.5	7.3	9.5	5.1	3.6	vm_normal_page, do_anonymous_page, do_swap_page
		NUMA support	18.2	6.3	2.1	1.4	8.4	do_numa_page, access_remote_vm, numa_migrate_prep
		TLB	6.6	2.2	0.5	1.1	2.8	tlb_flush_mmu, tlb_gather_mmu
	mpol	Policy definition	47.7	17.5	15.1	4.7	10.5	mpol_new, mpol_dup, mpol_shared_policy_lookup
		Policy enforcement	52.3	17.4	19.8	9.3	5.8	do_mbind, do_set_mempolicy, vma_replace_policy
	hugemm	Page table support for hugepage	68.5	33.7	19.1	2.3	13.5	change_huge_pmd, do_huge_pmd_numa_page, copy_huge_pmd
		Hugepage alloc	31.5	9.0	13.5	1.1	7.9	hugepage_init, alloc_hugepage, khugepaged_alloc_page
	hugetlb	Hugepage management	33.7	20.5	4.8	2.4	6.0	alloc_huge_page, free_huge_page, hugetlb_cow
		Hugepage fault	8.4	3.6	1.2	2.4	1.2	hugetlb_fault
		VM for hugepage	57.8	19.2	27.7	7.2	3.6	hugetlb_change_protection, vma_has_reserves
	mmap	Mmap operations	31.7	13.3	10.0	5.0	3.3	do_mmap_pgoff, do_munmap, exit_mmap
		VM for mmap	68.3	23.3	21.7	8.3	15.0	mmap_pgoff, mmap_region, vma_adjust
	alloc	Vmalloc	48.9	17.8	22.2	2.2	6.7	vmalloc_node_range, vmalloc_area_node, vmalloc_open
		Vmap	51.1	13.3	26.7	2.2	8.9	alloc_vmap_area, vunmap, free_vmap_area
GC	vmscan	Kswapd	20.0	5.5	7.3	1.8	5.5	kswapd, wakeup_kswapd
		Shrinker	52.7	12.7	16.3	7.3	16.3	shrink_inactive_list, shrink_page_list, shrink_zones
		GC helper	27.3	7.3	3.6	1.8	14.6	get_scan_count, pageout, scan_control

6.3 Virtual Memory Management

The virtual memory management is the largest component in Linux *mm*, it has six hot files: `memory`, `memorypolicy`, `huge_memory`, `hugetlb`, `mmap` and `vmalloc`. These hot files which contain the essential elements in virtual memory (e.g., page table, page fault handler, paging and TLB) has the largest number of committed patches (see Figure 3). These essential functions which have been developed for decades are still being updated to support new hardwares (e.g., NVDIMM and persistent memory [61]), and new usage of memory (e.g., huge page). And they are still buggy, e.g., even in the well-developed `do_set_pte` function, missing setting the *soft dirty* bit [79] could cause data inconsistency if a user space program is tracking memory changes [17].

A core component of memory management is the design and implementation of memory policies. We generally categorize the functions for memory policies into two categories: *policy definition* and *policy enforcement*. We find that *policy definition* has more *optimization* patches than *policy enforcement*, for instance, new policy is defined for choosing preferred NUMA node based on the number of private page faults; the kernel should avoid immediate memory migrations after switching nodes. As for *bugs* in memory policies, we find that 30% of the patches were applied to address the issues caused by missing checks, since memory policies usually count on many states (e.g., whether page is dirty) and statistics (e.g., cache hit/miss rate, number of page fault), missing one check would fail the policy enforcement.

Summary: The well-developed virtual memory management is still buggy. Specifically, a large portion of issues in *policy definition* and *enforcement* are caused by missing checks.

6.4 Garbage Collection

To preserve a certain amount of available memory pages for future allocation, Linux *mm* will garbage collect unused pages across memory zones. In `vmscan`, its functions can be categorized into two major categories: *kswapd* (the kernel swap daemon) and *shrinker*. For the remaining functions, we define them as *GC helper*, as they either provide the statistics of page usage, or handle the page out for *shrinker*. The *kswapd* will try to free pages if the number of free pages in the system runs low, while *shrinker*-relevant functions will be called to scan the corresponding components (e.g., slab cache, zones), and locate the candidate pages to be freed. Most interestingly, we find that more patches (52.7%) were applied to *shrinker* functions, and 84.1% of them relate to memory policies, focusing on how to scan memory regions with low overhead, and which pages should be reclaimed. However, the congestion events (22 related patches, e.g., [39, 51, 87]) during garbage collection make the memory performance unpredictable, it is essential to scale the GC process (e.g., per-zone scanning) and use page statistics to coordinate GC activities.

Summary: The *shrinker* is the hot spot in GC, it causes unpredictability in memory performance. A scalable and coordinated GC is desirable.

7 Related Work

The key components of virtual memory were studied back to 70's when DRAM first appeared [16]. Over the past decades, the functionalities of the core components have been enriched markedly, such as the buddy system [33] was proposed in 90's to remove the internal fragmentation, and later, general OS support for huge pages was proposed [21]. Today, the virtual memory system is still actively improved to support new features [3, 30, 72] and hardware [25, 32, 54, 71, 88]. There is no doubt that the memory manager has become one of the core subsystems in today's OS kernel. But few work has done any studies on the development of virtual memory system recently (over the past decade). Gorman et al. [22] analyzed the *mm* source code in Linux version 2.6.0-test4 (2003), while our work focuses on the study of patches which were committed to the latest Linux versions (from 2.6.32, 2009 to 4.0-rc4, 2015).

Patch and bug studies provide us insights on issue patterns in specific system software, and the experiences along its development. A number of such studies in various systems have been conducted recently. Lu et al. [37] studied the Linux file system patches, Chou et al. [9] investigated the operating system errors in Linux kernels, Kadav et al. [29] examined the code base of Linux device drivers, Palix et al. [60] studied the Linux kernel to version 2.6.33. We share the same purposes with these studies, but with a focus on Linux virtual memory system. To the best of our knowledge, our work is the first to conduct such a comprehensive study, which examines 4587 committed patches to Linux memory manager.

Memory issues can lead to serious problems, such as system crash, data corruption, suboptimal performance, etc. Many tools [2, 4, 7, 55] were built to address memory-related errors such as buffer overflows, dangling pointers, memory leaks in the programming and library space. Few of the related work is targeting at addressing the bugs in kernel space. Yang et al. [89] built checkers to find the bugs in storage and file systems, Prabhakaran et al. [63] developed a file system with better failure policies. Such efforts are also required as we build more reliable and efficient virtual memory systems. And also, a specific formal verification [24, 31] for virtual memory system is needed to verify its policies and implementation. The insights and bug patterns disclosed in this work would facilitate the development of these tools.

8 Conclusion

In this paper, we present a comprehensive study of 4587 committed patches in the Linux memory manager over the last five years. These patches reflect the development of the virtual memory system. We expect our findings to benefit the development of existing and future virtual memory systems, and their associated bug-finding and debugging tools. Our study would also shed light on the development of memory manager in relevant OSes as they share the same principles as Linux kernel.

Acknowledgments

This paper is dedicated to the memory of our colleague and mentor Karsten Schwan. Karsten was a prolific researcher, a great advisor, and a faculty leader in the School of Computer Science at Georgia Institute of Technology. The lead author greatly thanks him for his mentorship and guidance. We all miss you Karsten.

We thank our shepherd Zhen Xiao as well as the anonymous reviewers. We also thank Xuechen Zhang for the initial help and discussions on this work. This work was supported in part by the Center for Future Architectures Research (CFAR), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] Add rbtree postorder iteration functions, runtime tests, and update zswap to use. https://lwn.net/Articles/561124/.

[2] Akritidis, P. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security'10* (2010).

[3] Basu, A., Gandhi, J., Chang, J., Hill, M. D., and Swift, M. M. Efficient Virtual Memory for Big Memory Servers. In *ISCA'13* (Tel-Aviv, Israel, June 2013).

[4] Berger, E. D., and Zorn, B. G. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *PLDI'06* (Ottawa, Ontario, Canada, June 2006).

[5] bootmem: fix checking the bitmap when finally freeing bootmem. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/bootmem.c?id=6dccdcbe2c3ebe152847ac8507e7bded4e3f4546.

[6] bootmem: fix free_all_bootmem_core() with odd bitmap alignment. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/bootmem.c?id=10d73e655cef6e86ea8589dca3df4e495e4900b0.

[7] Bruening, D., and Zhao, Q. Practical memory checking with Dr. Memory. In *CGO'11* (Apr. 2011).

[8] call shake_page() when error hits thp tail page. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/memory-failure.c?id=09789e5de18e4e442870b2d700831f5cb802eb05.

[9] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An Empirical Study of Operating Systems Errors. In *SOSP'01* (Dec. 2001).

[10] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Scalable Address Spaces Using RCU Balanced Trees. In *ASPLOS'12* (London, England, UK, Mar. 2012).

[11] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Eurosys'13* (Prague, Czech Republic, 2013).

[12] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., and Kohler, E. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *SOSP'13* (Farmington, Pennsylvania, USA, Nov. 2013).

[13] cpuset: mm: reduce large amounts of memory barrier related damage v3. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=cc9a6c8776615f9c194ccf0b63a0aa5628235545.

[14] Cramming more into struct page. https://lwn.net/Articles/565097/.

[15] Cranor, C. D., and Parulkar, G. M. The UVM Virtual Memory System. In *USENIX ATC'99* (1999).

[16] Denning, P. J. Virtual memory. *ACM Computing Survey 2*, 3 (Sept. 1970).

[17] Don't forget to set softdirty on file mapped fault. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/?id=9aed8614af5a05cdaa32a0b78b0f1a424754a958&context=40&ignorews=0&dt=0.

[18] Fix Too Lazy Vunmap Cache Flushing. https://lkml.org/lkml/2009/6/16/722.

[19] fix wrong num_poisoned_pages in handling memory error on thp. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/memory-failure.c?id=4db0e950c5b78586bea9e1b027be849631f89a17.

[20] Fragkoulis, M., Spinellis, D., Louridas, P., and Bilas, A. Ralational access to Unix kernel data structures. In *EuroSys'14* (Amsterdam, The Netherlands, Apr. 2014).

[21] Ganapathy, N., and Schimmel, C. General purpose operating system support for multiple page sizes. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (1998), USENXI ATC '98.

[22] Gorman, M. Understanding the Linux Virtual Memory Manager. *ISBN 0-13-145348-3* (2004).

[23] Gunawi, H. S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K. J., Laksono, A., Lukman, J. F., Martin, V., and Satria, A. D. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *SOCC'14* (Seattle, WA, Nov. 2014).

[24] Hawblitzel, C., Howell, J., Lorch, J. R., Narayan, A., Parno, B., Zhang, D., and Zill, B. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *OSDI'14* (Broomfield, CO, Oct. 2014).

[25] Huang, J., Badam, A., Qureshi, M. K., and Schwan, K. Unified Address Translation for Memory-Mapped SSDs with FlashMap. In *ISCA'15* (Portland, OR, June 2015).

[26] Huang, J., Zhang, X., and Schwan, K. Understanding Issue Correlations: A Case Study of the Hadoop System. In *SOCC'15* (Kohala Coast, HI, Aug. 2015).

[27] Improving the Future by Examing the Past. http://isca2010.inria.fr/media/slides/Turing-Improving_the_future_by_examining_the_past.pdf.

[28] Jones, R. M. Factors affecting the efficiency of a virtual memory. *IEEE Transactions on Computers C-18*, 11 (Nov. 1969).

[29] Kadav, A., and Swift, M. M. Understanding Modern Device Drivers. In *ASPLOS'12* (London, England, UK, Mar. 2012).

[30] Karakostas, V., Gandhi, J., Ayar, F., Cristal, A., Hill, M. D., McKinley, K. S., Nemirovsky, M., Swift, M. M., and Unsal, O. Redundant Memory Mappings for Fast Access to Large Memories. In *ISCA'15* (Portland, OR, June 2015).

[31] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Norrish, M., Kolanski, R., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal Verification of an OS Kernel. In *SOSP'09* (Big Sky, Montana, Oct. 2009).

[32] Lee, E., Bahn, H., and Noh, S. H. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *FAST'13* (San Jose, CA, Feb. 2013).

[33] Li, K., and Cheng, K. H. A two dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system. In *Proceedings of the 1990 ACM Annual Conference on Cooperation* (1990), CSC '90.

[34] Linux 2.6.39 merge window. https://lwn.net/Articles/434637/.

[35] list_lru: per-node list infrastructure. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=3b1d58a4c96799eb4c92039e1b851b86f853548a.

[36] Lockless (and preemptless) fastpaths for slub. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=8a5ec0ba42c4919e2d8f4c3138cc8b987fdb0b79.

[37] Lu, L., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Lu, S. A Study of Linux File System Evolution. In *FAST'13* (Feb. 2013).

[38] Lumpy Reclaim. `https://lwn.net/Articles/211199`.

[39] memcg: fix zone congestion. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/vmscan.c?id=d6c438b6cd733834a3cec55af8577a8fc3548016`.

[40] memcontrol: do not acquire page_cgroup lock for kmem pages. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/memcontrol.c?id=a840cda63e543d41270698525542a82b7a8a18d7`.

[41] Memory Management in Memcached. `https://code.google.com/p/memcached/wiki/MemoryManagement`.

[42] Memory-management testing and debugging. `https://lwn.net/Articles/636549/`.

[43] mm anon rmap: replace smae_anon_vma linked list with an interval tree. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=bf181b9f9d8dfbba58b23441ad60d0bc33806d64`.

[44] mm: augment vma rbtree with rb_subtree_gap. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=d37371870ceb1d2165397dc36114725b6dca946c`.

[45] mm: avoid null-pointer dereference in sync_mm_rss(). `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/memory.c?id=a3a2e76c77fa22b114e421ac11dec0c56c3503fb`.

[46] mm, compaction: terminate async compaction when rescheduling. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=aeef4b83806f49a0c454b7d4578671b71045bee2`.

[47] mm: fix boundary checking in free_bootmem_core. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/bootmem.c?id=5a982cbc7b3fe6cf72266f319286f29963c71b9e`.

[48] mm: keep page cache radix tree nodes in check. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/lib/radix-tree.c?id=449dd6984d0e47643c04c807f609dd56d48d5bcc`.

[49] mm: munlock: manual pte walk in fast path instead of follow_page_mask(). `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=7a8010cd36273ff5f6fea5201ef9232f30cebbd9`.

[50] mm: munlock: remove redundant get_page/put_page pair on the fast path. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=5b40998ae35cf64561868370e6c9f3d3e94b6bf7`.

[51] mm: vmscan: fix inappropriate zone congestion clearing. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/vmscan.c?id=ed23ec4f0a510528e0ffe415f9394107418ae854`.

[52] mm, x86: saving vmcore with non-lazy freeing of vmas. `://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=3ee48b6af49cf534ca2f481ecc484b156a41451d`.

[53] MMTests. `https://github.com/gormanm/mmtests`.

[54] Moraru, I., Andersen, D. G., Kaminsky, M., Tolia, N., Ranganathan, P., and Binkert, N. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *TRIOS'13* (Farmington, USA, Nov. 2013).

[55] Nguyen, H. H., and Rinard, M. Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation. In *ISMM'07* (Montreal, Quebec, Canada, Oct. 2007).

[56] NUMA: do not trap faults on the huge zero page. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/huge_memory.c?id=e944fd67b625c02bda4a78ddf85e413c5e401474`.

[57] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast Crash Recovery in RAMCloud. In *SOSP'11* (Cascais, Portugal, Oct. 2011).

[58] page allocator: inline buffered_rmqueue(). `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=0a15c3e9f649f71464ac39e6378f1fde6f995322`.

[59] page allocator: use a pre-calculated value instead of num_online_nodes() in fast paths. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=62bc62a873116805774ffd37d7f86aa4faa832b1`.

[60] Palix, N., Thomas, G., Saha, S., Calves, C., Lawall, J., and Muller, G. Faults in Linux: Ten Years Later. In *ASPLOS'11* (Newport Bench, California, Mar. 2011).

[61] Persistent Memory Support Progress. `https://lwn.net/Articles/640113/`.

[62] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The Operating System is the Control Plane. In *OSDI'14* (Broomfield, CO, Oct. 2014).

[63] Prabhakaran, V., Bairavasundaram, L. N., Agrawal, N., Gunawi, H. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. IRON File Systems. In *SOSP'05* (Brighton, United Kingdom, Oct. 2005).

[64] Radix-Tree: Introduce bit-optimized iterator. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/lib/radix-tree.c?id=78c1d78488a3c45685d993130c9f17102dc79a54`.

[65] Radix Trees. `https://lwn.net/Articles/175432`.

[66] Rao, S., Heger, D., and Pratt, S. Examining Linux 2.6 Page Cache Performance. In *Ottawa Linux Symposium (OLS'05)* (2005).

[67] Readahead: fix NULL flip dereference. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/readahead.c?id=70655c06bd3f25111312d63985888112aed15ac5`.

[68] readahead: fix sequential read cache miss detection. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/readahead.c?id=af248a0c67457e5c6d2bcf288f07b4b2ed064f1f`.

[69] RED-BLACK TREES.
https://lwn.net/Articles/184495.

[70] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. Log-structured Memory for DRAM-based Storage. In *FAST'14* (Santa Clara, CA, Feb. 2014).

[71] SAXENA, M., AND SWIFT, M. M. Flashvm: Virtual memory management on flash. In *USENIX Annual Technical Conference'10* (2010).

[72] SESHADRI, V., PEKHIMENKO, G., RUWASE, O., MUTLU, O., GIBBONS, P. B., KOZUCH, M. A., MOWRY, T. C., AND CHILIMBI, T. Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management. In *ISCA'15* (Portland, OR, June 2015).

[73] SLAB: EMBED MEMCG_CACHE_PARAMS TO KMEM_CACHE.
https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=f7ce3190c4a35bf887adb7a1aa1ba899b679872d.

[74] SLOCCOUNT.
http://www.dwheeler.com/sloccount/.

[75] SLUB: ALTERNATE FAST PATHS USING CMPXCHG_LOCAL.
https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/slub.c?id=1f84260c8ce3b1ce26d4c1d6dedc2f33a3a29c0c.

[76] SLUB: CHECK FOR PAGE NULL BEFORE DOING THE NODE_MATCH CHECK.
https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=c25f195e828f847735c7626b5693ddc3b853d245.

[77] SLUB: FAILSLAB SUPPORT.
https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/failslab.c?id=773ff60e841461cb1f9374a713ffcda029b8c317.

[78] SLUB PAGE ALLOC FALLBACK: ENABLE INTERRUPTS FOR GFP_WAIT.
http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=caeab084deb61cd2d51cb8facc0e894a5b406aa4.

[79] SOFT DIRTY PTES.
https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt.

[80] SSD DISCARD COMMAND.
https://wiki.archlinux.org/index.php/Solid_State_Drives.

[81] STEINHORST, G. C., AND BATEMAN, B. L. Evaluation of efficiency in a virtual memory environment. In *SICOSIM4* (Oct. 1973).

[82] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. In *SOSP'03* (Bolton Landing, New York, Oct. 2003).

[83] THE LINUX KERNEL ARCHIVES.
https://www.kernel.org.

[84] THE ZSWAP COMPRESSED SWAP CACHE.
https://lwn.net/Articles/537422/.

[85] THP: ABORT COMPACTION IF MIGRATION PAGE CANNOT BE CHARGED TO MEMCG.
https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/compaction.c?id=4bf2bba3750f10aa9e62e6949bc7e8329990f01b.

[86] VIRTUAL MEMORY HISTORY.
http://en.wikipedia.org/wiki/Virtual_memory.

[87] VMSCAN: COUNT ONLY DIRTY PAGES AS CONGESTED.
https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=1da58ee2a0279a1b0afd3248396de5659b8cf95b.

[88] WEISS, Z., SUBRAMANIAN, S., SUNDARARAMAN, S., TALAGALA, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *FAST'15* (Santa Clara, CA, Feb. 2015).

[89] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI'06* (Seattle, WA, Nov. 2006).

[90] YOSHII, K., ISKRA, K., NAIK, H., BECKMAN, P., AND BROEKEMA, P. C. Performance and Scalability Evaluation of 'Big Memory' on Blue Gene Linux. *International Journal High Performance Applications 25*, 2 (May 2011).

[91] ZHANG, H., CHEN, G., OOI, B. C., TAN, K.-L., AND ZHANG, M. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering 27*, 7 (Apr. 2015).