Flattening the Page Tables on Linux in x86_64: Design, Implementation, and Evaluation in EMT Linux

Rubin Du[*]
University of Illinois at Urbana-Champaign
rd25@illinois.edu

Shanbo Zhang[*]
University of Illinois at Urbana-Champaign
shanboz2@illinois.edu

Abstract

Virtual memory translation is a critical component of modern operating systems but often suffers from significant overhead due to deep multi-level page walks—particularly in the x86_64 architecture. This overhead becomes a bottleneck for memory-intensive applications, especially under irregular access patterns found in data analytics, graph processing, and in-memory databases. In this work, we propose and implement a novel page table structure called the Flattened Page Table (FPT), which preserves the structure of hierarchical paging while enabling selective flattening of adjacent levels (e.g., L3-L2 or L4-L3+L2-L1). We integrate FPT into EMT-Linux [1] and modify both the kernel page table logic and QEMU to emulate hardware support for flattened walks. Our implementation supports multiple folding modes via kernel configuration and preserves backward compatibility with standard radix paging. Through a comprehensive evaluation using memory-intensive macro and application benchmarks, we show that FPT reduces kernel instruction overhead, improves instruction-per-cycle (IPC), and decreases page walk latency. These results demonstrate the feasibility and benefits of flattening page tables in commodity operating systems.

1 Introduction

When the computing system was first invented, the memory was limited and expensive, and relevant logic was simple, because there were only one program running on the CPU. As time went on, the memory became larger and cheaper, and multiple users might run their programs on the same computers, but the CPU could only run one program at a time. The operating system should be able to take control of the CPU and manage the memory access of the user programs. Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory and creates the illusion to users of a very large (virtual) main memory. Paging scheme partitions the physical memory into fixed-size blocks called frames, and uses a table to map each virtual page to a physical frame. The page table contains information about each page.

Originally, the page table in x86 architecture is a two-level page table, where the address field in the first level points to the starting address of the second level page table, and the address field in the second level page table points to the starting address of the physical frame. Then, in x86_64 architecture, the page table is extended to four levels. Later, it also supports 5-level page table extension. Typically, the page table is stored in memory due to its large size. For a memory access, in the worst case, the CPU needs to walk through all the levels of the page table to find the physical address. This is inefficient, and it could be the bottleneck of the system performance. Because of the popularity of machine learning, which consists of complex memory access patterns, the page table walking could be a performance bottleneck.

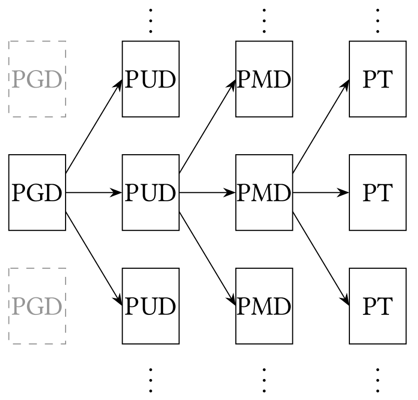

Figure 1. Traditional 4-level Radix Tree Strcuture in x86_64

In this project, we implements a new page table structure called flattened page table (FPT). FPT maintains the logical behavior and interface of the traditional multi-level tree-structured page table, while enabling the selective flattening of adjacent levels to reduce the depth of the page table walk. Specifically, assuming the 5-level paging extension is not used, we support three folding modes: L4L3, L3L2, and L4L3-L2L1 folding. In each mode, the virtual address range covered by individual levels remains unchanged, but the number of levels traversed per walk is reduced by merging multiple levels into a single larger page table. The resulting structure remains logically hierarchical, but folded levels are stored as wider tables in memory, decreasing the number of required memory accesses for translation.

2 Design

Figure 1 shows the traditional 4-level radix tree structure in x86_64. The radix tree is a tree data structure that stores the page table entries. Each node in the tree represents a

[*]Both authors contributed equally.

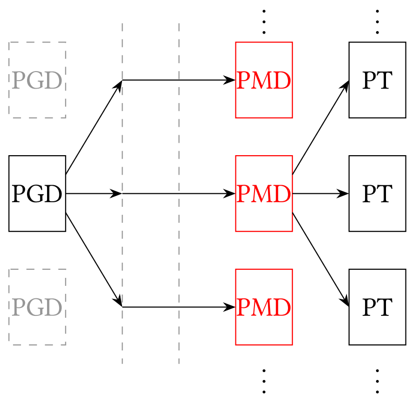

Figure 2. Flattened Page Table (FPT) in L3L2 Mode

level of the page table, and each edge represents a page table entry. The leaf nodes represent the physical frames. In the traditional 4-level radix tree structure, the uppermost level (L4) is the Page Global Directory (PGD), which points to the Page Upper Directory (PUD, L3). The PUD points to the Page Middle Directory (PMD, L2), and the PMD points to the Page Table (PT, L1). The PT points to the physical frames. In our design, we enables some level(s) of tables to be flattened. The lower level now spans the virtual address space spanned by the folded levels, but the size of the folded level(s) table is increased (from 4KB to 2MB). Take L3L2 folding as an example. An L3 page table page is 4KB and maps a 1GB address space; an L2 page table page is 4KB and maps a 2MB address space. In FPT L3L2 mode, however, the level 3 no longer exists, and L2 page table page is 2MB and maps a 1GB address space. Each entry in the L2 page table page points to an L1 page table page. Figure 2 shows the flattened page table in L3L2 mode. The PUD level no longer exists (even in hardware), and the PMD level (marked as red) is enlarged to 2MB.

3 Implementation

In this section, we will discuss the implementation of the FPT in Linux. Since x86_64 is a complex architecture, and it is not open-source, we can only emulate the behavior of the FPT mechanism in Linux by emulators. We modified QEMU with version 6.1.50 to support the FPT mechanism in "hardware". We also used Linux kernel version 5.15.0 to support the FPT mechanism. We changed the kernel to support the FPT mechanism by modifying the logic for page allocation logic, page table walk logic, and some TLB logic.

Before we start to discuss the implementation, we need to activate the FPT mechanism in the kernel. The FPT mechanism is not enabled by default in the kernel, but allowed by some new Kconfig options - one for enabling the FPT mechanism and one for selecting the folding level, as we mentioned before. The configuration becomes effective in the kernel image after the kernel is compiled with the new Kconfig options. We retain the backward compatibility with the radix structure, and the kernel could jump to suitable routines based on the enabled configuration.

3.1 EMT TEntry and Iterator Logic

The EMT Linux uses tentry struct to optimize the logic of reading, inserting, and updating a page table entry at a given address. It also comes with an iterator to reduce the overhead of traversing the page table. To accomodate for FPT, the tentry struct will ignore folded level and skip to next level, and the iterator it need to check whether the currently iterating page table is flattened or not, and align the address iterator to the same granularity as the page table when initializing the iterator from address, or finding next address to check.

3.2 Page Allocation Logic

The page allocation logic is responsible for allocating physical pages to the virtual address space of a process. The Linux kernel uses a buddy allocator to allocate physical pages. We do not need to modify the buddy allocator, but we need to modify the page allocation logic to support the FPT mechanism. Traditionally, the page table pages in each level have size 4KB. The kernel uses `struct page` to represent a 4KB page frame, and contiguous `struct page` maps to physically contiguous address space. In FPT, however, the page table pages of the flattened level could have size 2MB. Hence, we need to modify the page allocation logic to check the level we are allocating the page table. If the level is the flattened level, we need to allocate a contiguous address space of 2MB, which corresponds to 512 (2^9) `struct page` objects. To achieve that, the generic EMT interface of `create_context` and `insert_tentry` is overridden with fpt specific version, which upon allocating a page, checks if the respective level is folded into its next level, if not, it will attempt to allocate a 2MB page using `get_free_pages` with order of 9, if that was not possible due to that flattening this level was disabled by kernel config flags, or allocation of a order 9 page failed, it will allocate a 4KB page.After that, it will set the folded bit of the entry of that page to reflect whether it is a flattened page or not, to be used in translation and deallocation.

3.3 Page Table Walk Logic

The page table lookup logic is responsible for translating a virtual address to a physical address. The Linux kernel uses a multi-level page table structure to perform this translation. In FPT, the page table structure is flattened, which means that some levels of the page table are folded into the next level. This requires modifying the page table lookup logic to handle the flattened levels correctly. The translation happening on the hardware path mainly utilizes the flattened bit to determine whether to use 9 bits or 18 bits to index the next level. On the software path, the generic EMT interface of `p*d_offset_map_with_mm` is overriden with `fpt_p*d_offset_map_with_mm`, which checks kernel config and the respective flatten bit of the PT entry, to correctly compute the pointer to the next level of the page table. For example, if the system calls `pmd_offset_map_with_mm` via EMT, it will check the kernel config flags and the pud entry to see if pmd is folded into pte, if so it will simply return original pud entry, which can be used as a pmd entry to get pte via its next level translation interface. If not, it will then check if pud is folded into pmd, if so it will use 18 bits in the address to traverse the pmd table to find the correct entry, else it will do a normal pmd lookup.

3.4 Page Table Free Logic

The page table deallocation logic is responsible for freeing the physical pages that are no longer needed. In FPT, the page table deallocation logic needs to be modified to handle the flattened levels, as well as the use of mixed flattened and normal page tables correctly. The deallocation logic is similar to the reverse of the allocation logic, first it uses `unmap_vmas` to unmap and deallocate leaf pages, which uses the EMT tentry iterators to iterate over page tables to free all leaf pages. After that it will call `free_pgtables` to deallocate all page tables, which recursively scans from the top to lowest pagetable levels and deallocate all page tables fully covered from the vma's start to end address. In FPT this is modified to check if its the scanning level is folded, normal, or flattened. If folded it will simply pass all parameters to its next level, if normal it will follow radix deallocation logic, and if flattened it will scan over 2MB range and call next level deallocation logic for each entry, and if all is freed and within the vma's range, it will delete its entry from its parent's parent level. When a page table is throughly freed, also it need to invalidate itself from TLB using `p*d_free_tlb` and release the page back to the buddy allocator, if the page is flattened this will invalidate the range its parent cover from TLB, and free a order 9 page.

3.5 TLB Logic

TLB, or Translation Lookaside Buffer, is a hardware cache that stores the most recently used page table entries. The TLB is used to speed up the translation of virtual addresses to physical addresses. Flushing the TLB is a process of invalidating the TLB entries to ensure that the TLB does not contain stale entries. In Linux kernel, the TLB flush is associated with `struct page`, too. Since flushing the TLB is an expensive operation, the Linux kernel batches page frames to be flushed, until the batched list reaches some certain threshold. Nevertheless, in FPT, everytime we release a flattened page table page, there will be 512 (2^9) pages to be flushed, which is higher than all possible existing thresholds. Hence, we need to modify the TLB flush logic to support the FPT mechanism. Currently, our design is to enforce a TLB flush if some flattened page table pages are being released, which also helps release the pages existing in the batched list. This might not be a best design. Some improvements could be made in the future. For example, we could reprogram the existing TLB interfances to accommondate the changing page sizes (because we found some routines accept the `struct page` only, but not the size or the order). We could also change the policy to trigger a TLB flush, design a specialized TLB interface for the FPT mechanism, or even further polish the hardware structure of TLB.

3.6 Page Fault Handling Logic

The page fault handling logic is essentially same to radix, the only changes are the modifications to EMT iterators to align to the flattened page size when iterating over vmfs.

4 Evaluation

In this section, we evaluate the performance of our implementation of the flattened page table in Linux. We first present the experimental setup, including the hardware and software configurations. Then, we compare the performance of our implementation with the original 4-level radix paging mechanism in Linux using various benchmarks. Finally, we analyze the results and discuss the implications of our findings.

4.1 Methodology and Metrics

To evaluate the performance of our Flattened Page Table (FPT) implementation, we designed a comprehensive evaluation pipeline that executes benchmarks inside QEMU virtual machines, collects low-level memory traces, and performs offline analysis using instrumentation and custom tooling. Our methodology ensures consistency and fairness when comparing FPT against the baseline 4-level Radix Tree paging and the ECPT mechanism.

We run various benchmarks inside QEMU virtual machines with different configurations, including the original 4-level radix paging, the ECPT mechanism, and two configurations, L3L2 mode and L4L3+L2L1 mode, of our FPT implementation. The benchmarks include a mix of workloads that stress different aspects of the memory management system, such as memory allocation, page table lookups, and memory-intensive applications. For each benchmark, we split the execution into loading and running phases. The loading phase is used to load the benchmark into memory, while the running phase is used to execute the benchmark. We measure the execution time of each phase separately to understand the impact of our FPT implementation on both loading and running times.

During executing the benchmarks, we collect low-level memory traces using DynamoRIO, a dynamic binary instrumentation framework. We collect information about memory accesses, page walk latencies, TLB misses, instruction counts, and some other performance metrics. We then generate some visualization graphs to help partition the memory access of

Application	Working Set	# Records	Read:Write	# Requests
Redis	128 GB	536 M	50:50	60 M
Memcached	69 GB	56 M	80:20	10 M
PostgreSQL	64 GB	21 M	100:0	25 M

Table 1. Application workloads used in the evaluation.

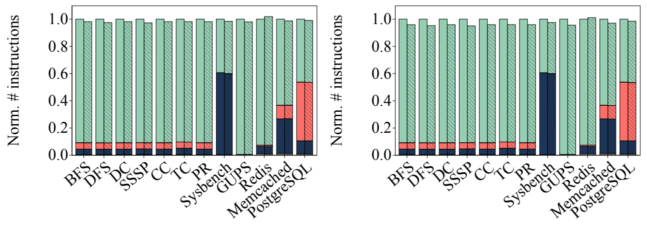

Figure 3. Distribution of kernel instructions of EMT-Linux with the Radix and FPT L3L2 mode (left) and L4L3+L2L1 mode (right).

each execution of benchmarks and compare the performance of the different configurations.

4.2 Benchmarks

We selected a diverse set of benchmarks to evaluate the performance of our FPT implementation.

We use seven memory-intensive macro benchmarks. Some benchmarks are selected from the GraphBIG benchmark suite, which is a collection of graph processing workloads. The benchmarks include: Breadth-First Search (BFS), Depth-First Search (DFS), Degree Centruality (DC), Single-Source Shortest Path (SSSP), Connected Components (CC), Triangle Counting (TC), PageRank (PR) [5].

We also run the GUPS benchmark for random memory access, and a memory test from SysBench [2, 3]. Both benchmarks have 64GB working sets.

We also keep track of three memory-intensive real-world applications: Redis, Memcached, PostgreSQL. [4, 7, 8]. Table 1 summarizes the workloads of these applications.

4.3 Results

4.3.1 OS Overhead of FPT

Figure 3 shows the distribution of kernel instructions under the Radix paging and two Flattened Page Table (FPT) configurations: L3L2 mode (left) and L4L3+L2L1 mode (right). The shaded areas represent instructions executed under FPT modes, while the blue, red, green, and gray segments correspond to instructions related to timers, system calls, page faults, and other kernel activities, respectively. Compared to the Radix baseline, both FPT modes show a slight reduction in the total number of kernel instructions, with a more noticeable decrease in the L4L3+L2L1 configuration.

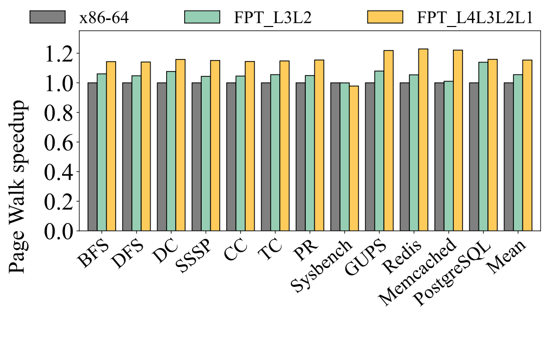

Figure 4. Page Walk Latencies of Radix and FPT modes

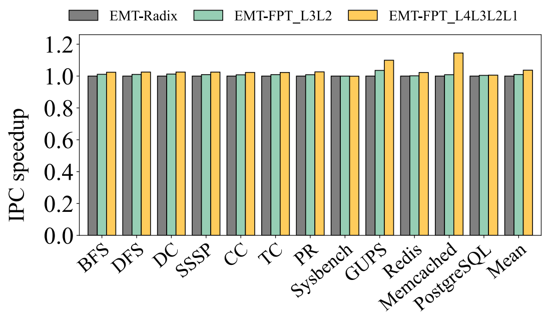

Figure 5. IPC Speedup of Radix and FPT modes

We believe that the reduction in kernel instructions is primarily due to the decreased number of page table walks and TLB misses. The kernel instruction reduction in L4L3+L2L1 mode could be because the L4L3+L2L1 mode reduces one more page table level that need to be traversed during address translation, leading to fewer kernel instructions related to page table lookups. Overall, these results demonstrate that FPT, especially the more aggressive L4L3+L2L1 mode, can alleviate kernel instruction overhead and improve execution efficiency for memory-intensive applications.

4.3.2 Performance Comparison

We compare the performance of Radix and FPT modes across several dimensions, including page table walk counts, instructions per cycle (IPC), and end-to-end execution latency. Across all metrics, the Flattened Page Table (FPT) configurations show consistent improvements over the traditional 4-level Radix page table.

Page Table Walks. As shown in Figure 4, the average page walk speed for both L3L2 and L4L3+L2L1 modes are accelerated, and the L4L3+L2L1 mode is mostly improved (up to about 1.2 times faster) than the Radix mode. The L3L2 mode shows a more modest improvement, but it still outperforms the Radix mode in most benchmarks. The reduction in page walk latency is particularly pronounced in workloads with irregular memory access patterns, especially in GUPS and

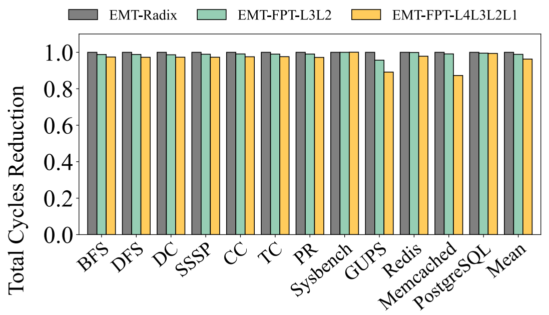

Figure 6. End-to-End Latency of Radix and FPT modes

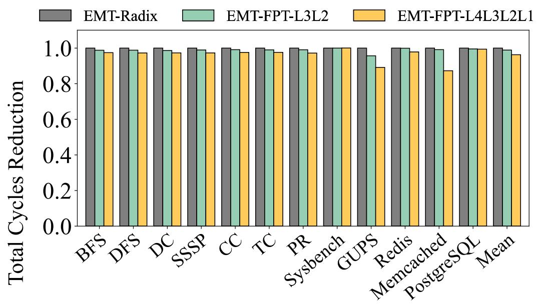

Figure 7. Running Phase Latency of Radix and FPT modes

Redis in L4L3+L2L1 mode. This suggests that the FPT optimizations are effective in reducing the overhead associated with deep page table walks.

Instructions Per Cycle (IPC). Figure 5 illustrates the IPC speedup across different benchmarks. Both FPT configurations achieve modest but consistent IPC improvements. The L4L3+L2L1 mode shows the most significant IPC improvement, particularly in GUPS and Memcached. The reduced number of page walks allows the processor to spend less time stalled on memory operations, thus improving execution throughput.

End-to-End Latency. We further evaluate the impact of FPT on total execution time in Figure 6, which includes both the loading and running phases, and Figure 7, which focuses on the running phase only. The L4L3+L2L1 mode consistently outperforms Radix, with reduction of up to 0.85 time in some benchmarks.

Overall, these results demonstrate that FPT reduces memory translation overhead, improves CPU efficiency, and shortens execution time. The L4L3+L2L1 mode provides the best performance across all benchmarks, validating the effectiveness of a deeper flattening approach.

5 Related Work

Address translation has long been a fundamental challenge in virtual memory systems, especially for memory-intensive workloads with large working sets. Traditional x86_64 systems use multi-level radix page tables, which, while space-efficient, suffer from translation overhead due to deep page walks. Numerous alternative designs have been proposed to reduce this overhead, improve scalability, or exploit memory-level parallelism.

To mitigate the inefficiencies of deep page walks, several works have also proposed flattened page table structures. Park et al advocate flattening to improve TLB reach and page table cacheability, reducing page walk latency [6]. Our Flattened Page Table (FPT) design extends this idea by selectively merging multiple page table levels (e.g., L3 and L2) to reduce walk depth while preserving the logical hierarchy.

Beyond flattening, researchers have explored alternative structures for page tables. Other works have explored alternative page table structures, like the clustered page table by Talluri and Hill [10], which groups multiple contiguous virtual pages into a single page table entry. By doing so, it reduces memory accesses for translations, making it particularly effective in systems with sparse or fragmented address spaces.

More recently, Skarlatos et al. proposed Elastic Cuckoo Page Tables (ECPT) [9], a novel design that replaces sequential pointer-chasing with fully parallel lookups using cuckoo hashing. ECPT supports dynamic resizing, process-private page tables, and multiple page sizes. It exploits memory-level parallelism during address translation and achieves substantial performance gains by avoiding the serialized nature of radix tree walks.

These alternative designs collectively reflect a growing interest in rethinking traditional paging mechanisms to meet the demands of modern high-performance and data-intensive applications.

6 Conclusion

This work presented the Flattened Page Table (FPT), a practical modification to the x86_64 Linux memory management subsystem that reduces virtual-to-physical address translation overhead by selectively collapsing levels of the page table hierarchy. By merging adjacent levels—such as L3L2 or L4L3+L2L1—into wider tables, FPT shortens the translation path while maintaining compatibility with existing software and hardware expectations. We implemented FPT in EMT-Linux and extended QEMU to emulate the required hardware behavior, enabling end-to-end support for flattened page walks.

Our evaluation demonstrates that FPT leads to measurable reductions in kernel instruction counts, page walk latency, and overall execution time on memory-intensive workloads, outperforming both standard radix paging and Elastic Cuckoo Page Tables (ECPT) in several cases. These results suggest that even modest changes to the page table layout can bring substantial benefits. FPT serves as a promising direction for future systems that seek to optimize address translation with minimal disruption to existing software ecosystems.

7 Metadata

The presentation of the project can be found at:

https://zoom/cloud/link/

The code/data of the project can be found at:

https://github.com/xlab-uiuc/linux_gen

https://github.com/xlab-uiuc/dynamorio

https://github.com/xlab-uiuc/VM-Bench

References

[1] Chai, S., Zhang, J., Kim, J., Wang, A., Chung, F., Stojkovic, J., Jia, W., Skarlatos, D., Torrellas, J., and Xu, T. Emt: An os framework for new memory translation architectures. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)* (2025).

[2] HPC Challenge Benchmark. RandomAccess: GUPS (Giga Updates Per Second). https://hpcchallenge.org/projectsfiles/hpcc/RandomAccess.html, Aug. 2022.

[3] Kopytov, A. SysBench: Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench/tree/1.0.20, Apr. 2020.

[4] Memcached. memcached - a distributed memory object caching system. https://memcached.org, Apr. 2024.

[5] Nai, L., Xia, Y., Tanase, I. G., Kim, H., and Lin, C.-Y. Graphbig: understanding graph computing in the context of industrial solutions. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), pp. 1–12.

[6] Park, C. H., Vougioukas, I., Sandberg, A., and Black-Schaffer, D. Every walk's a hit: making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2022), ASPLOS '22, Association for Computing Machinery, pp. 128–141.

[7] PostgresSQL. https://www.postgresql.org/, 2024.

[8] Redis. Redis. http://redis.io/, Apr. 2024.

[9] Skarlatos, D., Kokolis, A., Xu, T., and Torrellas, J. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, pp. 1093–1108.

[10] Talluri, M., Hill, M. D., and Khalidi, Y. A. A new page table for 64-bit address spaces. *SIGOPS Oper. Syst. Rev. 29*, 5 (Dec. 1995), 184–200.