Every Mapping Counts in Large Amounts: Folio Accounting

David Hildenbrand, *Technical University of Munich and Red Hat GmbH;*
Martin Schulz, *Technical University of Munich;*
Nadav Amit, *Technion, Israel Institute of Technology*

https://www.usenix.org/conference/atc24/presentation/hildenbrand

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Every Mapping Counts in Large Amounts: Folio Accounting

David Hildenbrand
Technical University of Munich and Red Hat GmbH

Martin Schulz
Technical University of Munich

Nadav Amit
Technion, Israel Institute of Technology

Abstract

Operating systems can significantly enhance performance by utilizing large contiguous memory regions, even when the memory is not mapped using huge pages, by streamlining memory management. To harness these advantages, Linux has introduced "folios," representing multiple contiguous pages. Unlike traditional huge pages, folios can be partially mapped, which complicates folio accounting and hinders both performance and memory savings.

Accurate and efficient folio accounting is crucial for optimizing memory management operations, enforcing various memory management policies, and performing Unique Set Size accounting in the operating system. In particular, determining whether a folio is exclusively mapped in a single address space is essential for avoiding unnecessary Copy-On-Write operations when memory is no longer shared.

We introduce a novel tracking scheme to determine, with negligible overhead, whether a folio is exclusively mapped in a single address space. Our solution achieves a memory overhead that grows sublinearly with the number of pages per folio. By implementing our method in Linux, we demonstrate a notable improvement in fork and unmap operations by 1.9x and 4.2x respectively, and in the performance of fork-intensive workloads, such as Redis, achieving up to a 2.2x speedup.

1 Introduction

Until recently, the adoption of huge pages in virtual memory systems has primarily been driven by the goal of utilizing architectural huge-page mappings for address translation, thereby reducing virtual address translation overheads by lowering the number of Translation Lookaside Buffer (TLB) misses and enhancing the efficiency of page table walks [8, 16, 23, 24, 30, 38]. These architectural benefits, crucial for system performance, have been the focus of extensive research. However, this emphasis on TLB advantages has often overshadowed another vital area: the broader potential benefit on Operating System (OS) memory management, particularly as system memory scales [15] to accommodate growing workloads, and these are expected to increase with the rising trend of memory disaggregation [17, 36, 40].

In Linux, for instance, the overheads of memory management tasks, such as paging operations, become more noticeable with increased memory sizes. The page reclaiming process involves traversing a page-linked list, the length of which is inversely proportional to the page size. To mitigate this and similar overheads, Linux introduced “folios”-aggregations [11] of multiple aligned contiguous pages. Managing memory in folio-sized units, potentially larger than the architectural base page size, rather than individual pages, substantially reduces many memory management overheads.

However, although folio state is mostly tracked as a single unit, the OS still tracks the number of mappings for each page, as folios might be partially mapped into page tables. Consequently, memory mappings and unmappings, as occur during `fork` or `munmap` system calls (syscalls), might necessitate updating the map count for each page within the folio, typically through resource-intensive atomic operations. A unified map count for the entire folio, while appearing efficient, is insufficient. It hinders the OS’s capability to ascertain if a folio is exclusively mapped to a single address space, a determination critical for various policy decisions. For example, it is essential to prevent unnecessary Copy-On-Write (COW) operations [18]. Aggregating map counts at the folio level inaccurately reflects the number of address spaces to which each page is mapped.

Accounting folios instead of individual pages has been identified by Linux maintainers as one of the critical future work items [25]. Past research neglected this challenge or deferred its solution. Resolving this issue is complex, as traversing reverse mappings entails significant overheads, and maintaining a per-folio counter of the number of mapping address spaces requires frequent page table scanning. Alternative OS designs like FreeBSD’s “VM Objects” [14] are known for their inefficiency, and the practicality of adapting them to folios remains questionable.

To eliminate the need for per-page map counts and the asso-

ciated overheads, we introduce a novel tracking scheme that effectively and accurately determines if a folio is exclusively mapped to a single address space. Utilizing the Pigeonhole Principle when the number of mappings exceeds the pages in a folio, we concentrate on cases where the map count is equal to or less than the folio's page count. For these instances, we have developed an aggregated per-folio value tracking scheme that allows efficient determination of whether all of the folio's pages are mapped to a single address space.

This solution significantly reduces folio map count update overheads and cuts down on redundant COW operations for folios that consist of multiple pages. As it eliminates the need for per-page map counts, it reduces the number of atomic operations needed to update the map count in the current Linux implementation—for a folio of 2^9 pages (typically 2MB)—from up to 1024 to just 1, and the memory required to hold mapping tracking data from over 2KB to a mere 40 bytes. Such efficiency sets the stage for future OS optimizations, including the potential reclaiming of individual page metadata that currently holds the map count to reduce memory overhead [26].

We implemented our solution in Linux and evaluated its performance. Microbenchmark tests demonstrate a speedup in fork and unmap operations for 1GB of memory by up to 1.9x and 4.2x, respectively. It also enhances write-fault performance by avoiding unnecessary COW operations. In macrobenchmark tests, we noted a speedup in Redis IOPS of up to 2.2x, a fork-intensive workload, following fork operations. The performance of a multi-process Python program also improved by up to 60% due to our solution.

Our contributions are threefold:

- We analyze the challenge of efficiently determining whether folios are exclusively mapped to a single address space when using folios or similar constructs.
- We present a solution for precisely determining whether folios are exclusively mapped with a memory overhead that only grows sublinearly relative to the number of pages per folio.
- We evaluate our solution on Linux showcasing effective performance enhancement in fork-intensive workloads of up to 2.2x.

2 Background

Folios represent a new concept in Linux, conceived as an extension of the "huge page" (or compound page) metadata to represent a contiguous range of base pages. To fully understand their importance and the challenge of tracking mapping count only in the folio level, we first delve into huge pages and their advantages.

Huge Pages. A huge page is a large contiguous memory region, whose size is a multiple of the base page size. CPU architectures have mechanisms to map these pages in virtual memory as a singular unit. This allows for the efficient caching of address translations for larger memory regions in the Translation Lookaside Buffer (TLB), which holds the translations from virtual to physical memory addresses, thereby reducing the TLB miss rate [38]. In radix-tree-based page table architectures, such as x86-64, huge pages are mapped at a higher level within the page table, reducing the time needed to translate a virtual address to a physical one during a TLB miss.

Traditionally, CPUs supported only a few huge page sizes, and these huge pages were mapped as single units. Modifying permissions or mappings within huge pages necessitated, besides updating the page table entries, splitting the huge-page data structure, as saved and tracked by the OS into smaller ones. In addition, automatic promotion of memory to huge pages has proven to be hard [23, 29, 42]. These limitations restricted the flexibility and applicability of huge pages.

Nonetheless, huge pages present advantages beyond just accelerated address translation. Allocation of physical memory of fewer larger blocks can be more efficient [22], as well as other memory management operations in larger page granularity [12, 41]. Moreover, by using huge pages, which are contiguous in physical memory, sequential access benefits from improved cache efficiency, and the interleaving of Dynamic Random-Access Memory (DRAM) pages allows for more effective prefetching, thus reducing the access times for DRAM by minimizing row conflicts [21].

In addition, recent hardware developments have broadened the capabilities for virtual memory mapping of memory chunks in diverse sizes. Notably, ARM architectures have implemented the "contiguous bit" feature in their page table entries [7]. This bit indicates physical contiguity of a series of pages, allowing the system to treat them as a larger memory block. Recent AMD CPUs can now transparently cache aligned contiguous translations of 16KB blocks in a single TLB entry [6], similar to previous work by Pham et al. [33]. This advancement significantly improves performance, mitigating many of the limitations previously associated with the use of huge pages [37]. Other work proposed further Memory Management Unit (MMU) extensions for the OS to communicate contiguous translations in the page tables to the hardware [20, 31, 32].

Folios. To streamline the management of huge pages, capitalize on new hardware features, and reduce the performance and memory overheads associated with memory management operations, Linux introduced the concept of folios. A folio represents a contiguous range of physical pages, the size of which is a power of 2 [11]. This approach effectively decouples the sizes of physical memory allocation units from their corresponding virtual memory mappings. Consequently, folios can be partially mapped and their splitting upon partial

unmapping can be deferred, pending memory pressure in the system. This decoupling is particularly advantageous, for example, in Arm64 architectures, where it is possible to allocate and map 64KB folios by default, even with a base page size of 4KB, leading to notable speedups [12].

The introduction of folios offers the possibility of reducing the volume of metadata that the OS must track for memory management. By shifting to per-folio metadata instead of the traditional per-page metadata approach, this change has the potential to not only lower the memory overhead—presently about 1.6% in systems with 4KB base page size [26]—but also to decrease the number of memory accesses required for reading and updating metadata. This reduction could lead to lower latency in memory management operations, enhancing system efficiency.

While managing information at the folio level is typically straightforward for most of the page's metadata, such as reference counters and flags, complexities arise in tracking the *map count*. The map count reflects the number of times a memory segment is mapped into address spaces. During a process fork—a standard operation in Unix systems where a child process is created as a copy of the parent and its address space is replicated—memory is not copied but mapped into the child's address space [10, 27]. The map count of the involved memory pages is therefore increased.

In the legacy approach of tracking the map count at the page level, this count for anonymous pages—Unix memory not backed by a file—serves a dual purpose. It indicates both the total number of mappings set in page tables for a page and the number of distinct address spaces in which the page is mapped. These values align since anonymous pages are mapped only once in each address space. However, aggregating this count at the folio level does not indicate the number of address spaces in which the folio is mapped, as the folio might be partially mapped in multiple address spaces. This presents a challenge: although the OS does not require the exact count of address spaces where a folio is mapped, it is essential to determine if the folio is exclusively mapped within a single address space.

Folio Exclusivity. Determining whether a folio is exclusively mapped currently affects various policies and operations of the OS. For example, Linux allows userspace to modify the memory policy of a folio through the `madvise`, `set_mempolicy`, and `mbind` syscalls only if the folio is mapped exclusively in a single process address space to avoid unintended interactions between processes.

Furthermore, when a process writes to memory following a fork, it triggers a Copy-On-Write (COW) page fault. The OS then needs to determine whether the folio containing the faulting page is shared with other processes. If the folio is not shared, the OS can allow the process to write directly to the folio, i.e., *reuse* the folio, which is much more efficient than copying. Losing this optimization can lead to significant performance degradation [18].

Similarly, when calculating the Unique Set Size (USS) of a process, the OS needs to determine whether a folio is exclusively mapped into a single process address space. The USS is the memory guaranteed to be private to a process, and accounting mapped folios that are exclusively mapped towards the USS maintains that guarantee.

File-backed folios. Although pages of a file-backed folio can be mapped multiple times in a single address space, it is uncommon, and optimizing for this use case is not deemed important. Linux considers file-backed pages as exclusively mapped when they are mapped only once, but considers them as shared when they are mapped multiple times, even within the same address space. Consequently, Linux applies the same logic for detecting exclusively mapped file-backed folios that it uses for detecting anonymous folio exclusivity.

Related Works. While various studies have delved into issues somewhat related to these challenges, none have directly described it or offered a solution. The use of fork, despite criticisms that it is an "outdated hack" [9], remains prevalent due to its simplicity and effectiveness in many application implementations. Sharing page-tables, which has been proposed as a substitute to huge pages for reducing fork overheads [41], presents practical implementation difficulties and negates the benefits of huge pages, such as reduced address translation and OS memory management overheads. Other studies have either overlooked these challenges [39] or postponed addressing them to future research endeavors [18].

Examining industrial solutions, we observe that OnePlus employs a customized Linux kernel that tracks the mapping count at both the page level, when the folio is partially mapped, and at the folio level when fully mapped [28]. Although this approach reduces overheads for fully mapped folios, it still encounters the same overheads when the folio is partially mapped, and cannot reduce the memory overhead. Moreover, this method does not allow determining if the folio is exclusively mapped in a single address space. As for other OSes than Linux, FreeBSD adopts a distinct approach to memory management, tracking the number of mappings at the "VM object" level, which represents a memory layer referenced by a page [14]. Although this design can address the issue, managing VM objects in FreeBSD has been noted for its inefficiency [13], casting doubt on the feasibility of adapting them for use with folios.

3 Design

Our goal is to efficiently determine whether a folio is exclusively mapped to a single address space. In other words, if any page of the folio is mapped in one address space and another page (or the same page) is mapped in a different address space, the folio is not exclusive, and our scheme must correctly identify this. Conversely, if all mapped pages of the folio are in the same address space, our scheme must report it as exclusively

mapped. Our solution should perform these checks quickly and with minimal memory overhead. Concretely, we aim for the number of operations and the size of the tracking data to grow sublinearly with the number of pages per folio.

We therefore rule out naive and inefficient solutions such as traversing the folio's reverse mappings, which would require multiple memory accesses to disparate locations and acquiring multiple locks, or tracking for each folio the *number* of times its pages are mapped in each address space.

Eliminating individual page map_count values. Currently, each page within the folio maintains its own separate *map_count* for individual mappings through page-table entries. In addition, folios that consist of multiple pages have an *entire_map_count* that reflects the number of times the folio is mapped as a huge page, using a single large entry in the page tables. We eliminate the per-page *map_count* values and instead introduce a new folio-level *map_count* to track the cumulative number of times all of the folio's pages are mapped across all address spaces, regardless of the specific page table mapping type used.

Pigeonhole Principle-Based Decisions. Based on anonymous folio behavior, we can assume that while pages within a single folio can be mapped in multiple address spaces, each page is restricted to one mapping per address space. This limits each folio to a maximum of $nr\_pages = 2^{order}$ mappings in each address space. Determining if a folio is exclusively mapped is therefore straightforward when a folio's *map_count* is greater than its *nr_pages* value. In this case, we know it cannot be exclusive since each page is only mapped once per address space. This simplifies our task to considering cases where $folio.map\_count \leq folio.nr\_pages$.

Determining Exclusivity with Low Mapcount. For determining whether folios with a low mapcount are exclusively mapped in a particular address space, we extend the *map_count*-based tracking scheme.

First, we assign a unique identifier to each address space, called *as.id*. Additionally, each folio maintains a multi-word cumulative value, termed *folio.as_aggregator*. Changes to *folio.as_aggregator* happen atomically with changes to *folio.map_count* as pages are mapped or unmapped. We derive an adjustment value *as_id_unit* from *folio.nr_pages* and the respective *as.id*. Whenever we map or unmap a page, we add or subtract *as_id_unit* to *folio.as_aggregator*, respectively. As the addition and subtraction are inverse operations, we disregard any overflows or underflows during these operations as they do not affect correctness. We can then simply compare *folio.as_aggregator* against the product of *as_id_unit* and the folio's *map_count*, to determine if the folio is exclusively mapped in a particular address space:

$$folio.as\_aggregator = folio.map\_count \cdot as\_id\_unit \quad (1)$$

If and only if the values match, the folio is exclusively mapped to this address space; otherwise, it is not.

```
def as_id_unit(as, folio):
    binary_unit = bin(as.id)[2:]
    return int(binary_unit, base=folio.nr_pages + 1)

def add_folio_mapping(as, folio):
    folio.map_count += 1
    folio.as_aggregator += as_id_unit(as, folio)

def remove_folio_mapping(as, folio):
    folio.map_count -= 1
    folio.as_aggregator -= as_id_unit(as, folio)

def is_folio_mapped_exclusively(as, folio):
    if folio.map_count > folio.nr_pages:
        return false
    return (as_id_unit(as, folio) *
            folio.map_count == folio.as_aggregator)
```

Figure 1: Pseudo-code for folio accounting.

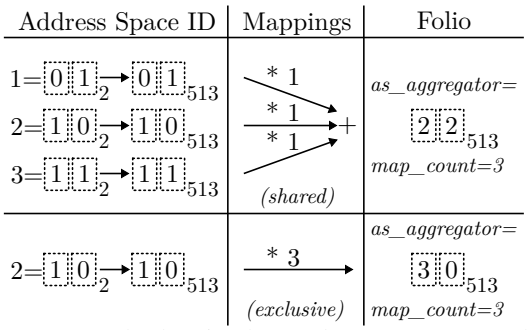

Figure 2: Example showing how *folio.as_aggregator* differs when three address spaces each map a page of a folio that consists of 512 pages, compared to only a single address space mapping 3 pages. For instance, whether ID 2 exclusively maps the folio is checked by $10_{513} \cdot 3 = 30_{513}$.

However, how we derive *as_id_unit from folio.nr_pages* and the *as.id* of a particular address space is crucial for correctness. To prevent errors, our derived adjustments must ensure that the sum of multiple address-space *as_id_unit* values never equals to the product of a single *as_id_unit* and the *folio's map_count*, provided that $folio.map\_count \leq folio.nr\_pages$.

Our solution is to derive *as_id_unit* from the binary representation of *as.id*, but interpreted in a base of $folio.nr\_pages + 1$. For instance, if *folio.nr_pages* is 512 and the as.id is 5 (binary '101'), the adjustment would be $101_{512+1} = 101_{513} = 263,170$. The entire algorithm pseudo-code is detailed in Figure 1.

Example. Consider three address spaces with respective IDs of 1, 2 and 3. Figure 2 shows how *folio.as_aggregator* differ for a folio that consists of 512 pages, when all three address spaces each map a page of the folio, compared to when only the address space with ID 2 maps three pages of the folio. Our solution can successfully distinguish the first, non-exclusive case from the second, exclusive case even though the binary representations of the address space IDs overlap.

Correctness. Considering addition and subtraction are inverse operations, when a folio is exclusively mapped to one address space, its *as_aggregator* should exactly match the product $as\_id\_unit \cdot folio.map\_count$. We must ensure that *as_aggregator* equals this product only in that case. Note that since *folio.as_aggregator* is the sum of *folio.nr_pages* numbers or fewer, and in the base of $folio.nr\_pages + 1$, these numbers are composed of digits 0 or 1, the addition of each digit would not result in a carry. If a page of the folio is mapped to a different address space, this address space would have a different ID, and at least one digit in the sum will be incremented or decremented, leading to a different *as_aggregator* value. This ensures that *as_aggregator* uniquely represents exclusive mapping to a single address space when $folio.map\_count \leq folio.nr\_pages$.

Memory Consumption. The number of bits necessary to store the *as_aggregator* is defined by:

$$n\_bits = \lceil log_2((folio.nr\_pages + 1)^{\lceil log_2(max\_as) \rceil} - 1) \rceil \quad (2)$$

where *max_as* is the maximum number of address spaces. For 2MB folios (512 base pages) and 22-bit maximum address space, 199 bits are needed, which in practice require $4 \cdot 8$ byte.

Accounting Optimizations. To accelerate the computation of the *folio.as_aggregator* value, we can precompute and store the *as_id_unit* values for common folio sizes and address spaces. Another approach is to use a larger base value that is a power of 2, which simplifies the calculations at the cost of a slight increase in the number of bits required to store a folio's *as_aggregator*.

4 Evaluation

We implemented our solution on Linux 6.7, incorporating the upcoming multi-size THP (mTHP) patches, designed to allow control over the enabled folio sizes [34]. Our implementation, consisting of 1519 lines of code, builds upon the existing functionalities of mTHP.

For evaluation, we use the mTHP-enabled Linux 6.7 as a baseline (*Baseline*) and compare it with our modified kernel (*FolioMap*). We measure the performance of all available folio sizes. Benchmarking is conducted on a dual 10-core Intel Silver 4210R CPU server with 32GB memory, running Fedora 38. The compilers and software used are supplied by Fedora 38. To ensure accuracy, we disable Hyper-Threading, fix the CPU frequency, and repeat each test at least 10 times. The standard deviation in all experiments is below 3%.

4.1 Microbenchmarks

OS Primitives. We wish to evaluate the effect of our solution on memory management operations that modify folio mappings, particularly when the folio is mapped using base pages instead of huge pages. This scenario occurs when the folio size does not match the supported huge-page sizes or when different pages within a folio require different protections. To assess the performance impact, we run microbenchmarks that allocate 1GB of memory using folios of varying sizes. For the case where 2MB folios are used, the benchmarks execute `madvise` syscalls to force the folios to be mapped using 4KB base page table entries. We measure the execution time of two common syscalls that require updates to folio accounting: `fork`, which duplicates both the process and its memory mappings, and `munmap`, which unmaps the allocated memory.

The results, depicted in Figure 3, show that *FolioMap* significantly reduces the execution time of both `fork` and `munmap` syscalls. *FolioMap* executes `fork` and `munmap` up to 1.9 and 4.2 times faster, respectively, due to fewer metadata updates being required. By dividing the time difference between *Baseline* and *FolioMap* by the number of pages in 2048KB folios, we find that *FolioMap* shortens the execution time by 26ns per page for `fork` and 68ns per page for `munmap`.

Although *FolioMap* significantly improves the performance of both `fork` and `munmap`, the benefit is more pronounced for `munmap`. This is mainly because `munmap` updates both the per-page mapping counters and the counter of the number of mapped pages in a folio, while `fork` only updates the per-page mapping counters. By removing both counters, *FolioMap* eliminates the overhead of updating them, resulting in a more significant performance improvement for `munmap` compared to `fork`

Write Fault Latency with COW. *FolioMap* enables memory reuse in scenarios where a write-protected folio is no longer shared, in contrast to *Baseline*, which must resort to a COW operation due to its inability to determine if a page is no longer shared. We run a microbenchmark, which allocates 2MB of memory using folios of varying sizes. When 2MB folios are used, the benchmark triggers a remapping so they are mapped using 4KB base page table entries instead of a single huge-page entry. The benchmark then forks, exits the child process, and then triggers write faults across different folio sizes. *FolioMap* handles these write faults in 826ns for 16KB folios and 830ns for 2MB folios. These times are up to 2.1x faster than *Baseline*, which handles write faults in 1538ns for 16KB folios and 1762ns for 2MB folios.

However, in a variant of the microbenchmark where the child process remains active and folios continue to be shared, both *Baseline* and *FolioMap* are required to perform COW to maintain correctness. Under these conditions, *FolioMap* exhibits a slight slowdown of up to 20ns (1%), likely due to more operations for managing the folio's *as_aggregator* when only replacing a single page of the shared folio that consists of multiple pages in the page tables (data not shown).

Concurrent Accounting Operations. One concern is whether the additional synchronization operations during accounting might negatively impact the performance of our so-

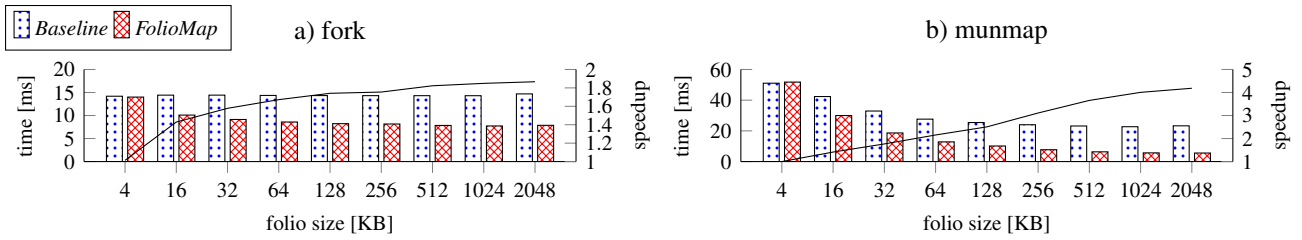

Figure 3: Average execution time of `fork` and `munmap` with 1GB memory backed by varying folio sizes, with the lines representing *FolioMap*'s speedup over *Baseline*. 2MB folios are mapped using 4KB base page table entries instead of a single huge-page entry.

lution. To assess this impact, we conducted a stress test using the vm-scalability benchmarks `case-anon-cow-rand` and `case-anon-cow-seq` [5], specifically chosen for their ability to simulate extreme contention scenarios. These benchmarks run multiple processes, which share memory, and continuously write to it, thus repeatedly triggering write faults that necessitate COW operations. The test runs 20 processes simultaneously, each on a separate core, writing to a total of 1GB of read-only shared memory.

In the `case-anon-cow-rand` benchmark, *FolioMap*'s performance slowdown compared to *Baseline* is minimal, less than 0.3% across all tested folio size. The `case-anon-cow-seq` benchmark represents an even more rigorous test, with all processes synchronously triggering page faults on the same pages in the same order—a scenario that is rather unrealistic. Despite this, *FolioMap* exhibits a slowdown of less than 2%, demonstrating its robustness even under highly stressful and uncommon conditions.

4.2 Macrobenchmarks

In our macrobenchmark tests, we focus on workloads involving short-lived subprocesses, which are notably affected by inefficiencies in per-page mapping accounting. It is important to note that on certain CPUs, such as recent AMD models capable of caching multiple page table entries in a single TLB entry, the performance advantages of large folios over base 4KB pages are likely to be even more pronounced.

Redis with Snapshotting. We run Redis [2], an in-memory database that uses the `fork` syscall to efficiently create background snapshots [3]. By invoking fork, Redis spawns a temporary child process that shares the parent's memory, which the OS write-protects, allowing the child to save the snapshot to persistent storage.

For our experiment, we populate Redis with 10^8 keys in a "parallel (sequential)" pattern, consuming 10GB of memory. After the snapshot is completed, we run the *memtier_benchmark* [4] v2.0.0 for 60 seconds with three concurrent connections and a pipeline depth of 2000, using the same key pattern. We conduct this with 64KB folios using both *FolioMap* and *Baseline*, and compare it to a 4KB baseline, expecting similar or enhanced improvements with larger folios.

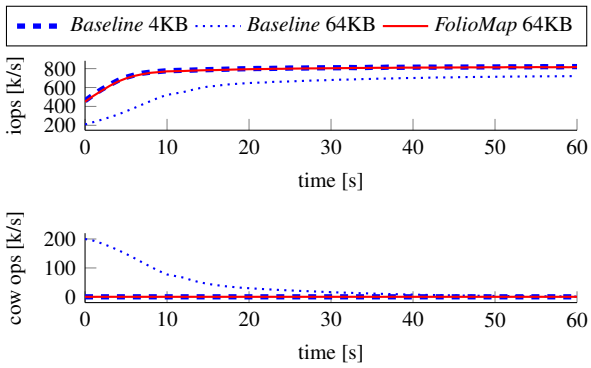

Figure 4: IOPS and number of COW operations during memtier benchmark after taking a background snapshot.

We first measure the duration of the `fork` syscall, a period when no requests are processed and should therefore be shortened. *FolioMap*, using 64KB folios, completes `fork` in 111ms, outperforming *Baseline*, which takes 167ms for both 4KB and 64KB folios ($\sigma < 0.5\%$, data not shown).

Subsequently, we monitor average IOPS and COW operations each second. Figure 4 presents these results. Initially, *FolioMap* demonstrates a throughput (IOPS) that is 2.2 times higher than *Baseline* with 64KB folios. This advantage decreases over time as more memory becomes writable. Notably, the IOPS achieved by *FolioMap* are on par with *Baseline* using 4KB folios ($\sigma < 1.3\%$). The results highlight that without our solution, the use of large folios by the OS leads to numerous redundant COW operations, resulting in significant performance degradation.

While *FolioMap* delivers IOPS on par with *Baseline* using 4KB folios on our system, the advantage of 64KB folios lies in speeding up memory management operations, such as the measured `fork`, with potential for even greater benefits on systems supporting multiple page table entries per TLB entry.

Python Sharing Memory with Subprocesses. Sharing large in-memory data structures with subprocesses is a common practice in scenarios such as parallel computations [1] or content-based filtering recommendations [35]. These subprocesses typically access the shared data without modifying it. Python, frequently used in such contexts, defaults to using the `fork` syscall for spawning subprocesses through its

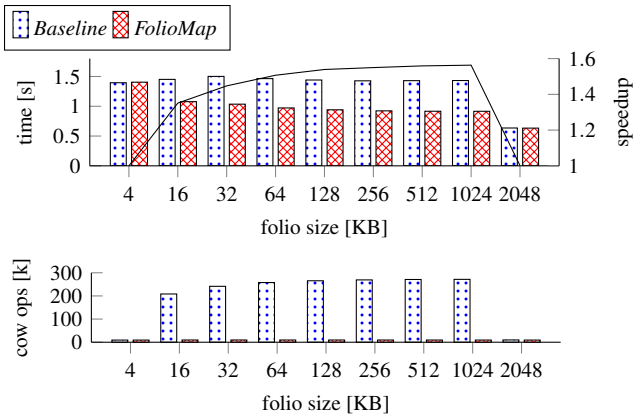

Figure 5: Execution time and number of COW operations during a Python program execution, with the line representing *FolioMap*'s speedup over *Baseline*. Throughout the experiment, 2 MB folios are consistently mapped by a single huge-page entry.

"multiprocessing" API.

We run a Python script that creates a 1GB numpy array, subsequently spawning eight subprocesses to compute the sum of its elements (detailed in Appendix A). As shown in Figure 5, *FolioMap* demonstrates up to a 1.6x speedup, progressively increasing with folio size, except for 2MB folios. This improvement stems from minimizing unnecessary page copies during write faults, combined with faster subprocess creation and teardown with large folios. For 4KB and 2MB folios, however, *FolioMap*'s performance aligns with *Baseline*, as these folios are mapped with a single huge-page entry and therefore existing mapping accounting mechanisms are capable of accurately distinguishing them as exclusive during write faults.

This benchmark highlights the benefit of using large folios with our solution, as it lowers memory management overhead without negatively inducing unnecessary COW operations. Using medium-sized folios, such as 64KB, can be a compelling alternative to using very large folios (e.g., 2MB), as large folios are often unavailable due to memory fragmentation.

5 Conclusions

In this paper, we introduced an innovative tracking scheme that efficiently determines if a folio is exclusively mapped to a single address space. By combining a specialized tracking approach with the Pigeonhole Principle, our solution ensures scalable and precise tracking, achieving speedups of up to 2.2x.

Our work effectively addresses an outstanding challenge in folio accounting, making medium-sized folios a viable option for Linux. Medium-sized folios offer an appealing alternative to huge pages, as they are less susceptible to fragmentation and can benefit from emerging TLB enhancements. By enabling more efficient utilization of medium-sized folios, our solution paves the way for enhanced system performance in Linux memory management.

Acknowledgments

We thank our anonymous reviewers, Dr. David Alan Gilbert, and the Linux community, particularly Hugh Dickins, Mike Rapoport, Ryan Roberts, Yang Shi, Linus Torvalds, Peter Xu, Zi Yan, and Fengwei Yin, for their valuable feedback and contributions to Linux's folio accounting. Special thanks to Matthew Wilcox for driving the folio abstraction in the Linux kernel and to Ryan Roberts for developing mTHP support.

Availability

After sharing an early implementation with the Linux community for feedback, we have incrementally upstreamed our code. So far, 69 preparatory patches have been merged into the Linux kernel, up to version 6.9. These patches optimize accounting during fork and unmap operations when processing multiple pages of the same folio and prepare for further folio accounting changes. In Linux 6.10-rc1, we have added a single folio *map_count* counter to folios that consist of more than a single page and are preparing for the removal of the individual per-page *map_count* values. We will continue collaborating with the community to improve the implementation and fully integrate our solution into future Linux kernel releases.

A Python Benchmark Program

Figure 6 shows the Python program we use for our evaluation in § 4.2.

```
import multiprocessing as mp
import numpy

size = pow(512, 3)
arr = numpy.ones(size)

def fn(range):
    return numpy.sum(arr[range[0]:range[1]])

def multi_process_sum(arr):
    c = int(size / 8)
    ranges = [(i,i + c) for i in range(0, size, c)]

    pool = mp.Pool(processes = 8)
    return int(sum(pool.map(fn, ranges)))

assert(multi_process_sum(arr) == size)
arr[0:size] = 0
assert(multi_process_sum(arr) == 0)
```

Figure 6: Python program that uses numpy and multiprocessing to perform parallel computations on read-only shared memory.

B Artifact Appendix

Abstract

FolioMap is our implementation of the folio accounting design presented in Section 3. The artifact comprises patches for two custom Linux kernels—*FolioMap* and *Baseline*—based on Linux 6.7, three microbenchmarks, two macrobenchmarks and helper scripts to install and run the artifact.

Scope

The artifact contains all necessary software components to run the experiments used throughout the evaluation in Section 4.

Contents

The included `README.md` file contains further information on installation, configuration and evaluation.

Hosting

The artifact is available on Gitlab (https://gitlab.com/foliomap_paper/ae/, "v1" tag) and was archived [19] on Zenodo.

Requirements

FolioMap has been tested only on the x86-64 architecture, and the artifact is designed to run on x86-64 hardware with 20 cores, 32GB of RAM, and 30GB of free disk space. Our test system had 2 CPU sockets, each with a 10-core CPU and 16GB of RAM, although we do not expect this specific configuration to significantly impact the results.

Running the artifact and reproducing the results requires a Fedora 38 installation with root and internet access.

References

[1] Multiprocessing with NumPy arrays. https://www.geeksforgeeks.org/multiprocessing-with-numpy-arrays/. Accessed: 2024-01-10.

[2] Redis. https://redis.io/. Accessed: 2024-01-10.

[3] Redis persistence. https://redis.io/docs/management/persistence/. Accessed: 2024-01-10.

[4] Source code of the memtier benchmark. https://github.com/RedisLabs/memtier_benchmark. Accessed: 2023-12-14.

[5] Source code of the vm-scalability benchmark. https://git.kernel.org/pub/scm/linux/kernel/git/wfg/vm-scalability.git. Accessed: 2023-12-14.

[6] Advanced Micro Devices (AMD). *Software Optimization Guide for the AMD Zen4 Microarchitecture*, revision 1.00 edition, 2023. Document No. 57647, Accessed 2024.

[7] ARM. Armv8-A address translation. https://documentation-service.arm.com/static/5efa1d23dbdee951c1ccdec5, 2019. Accessed: 2024-01-07.

[8] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, page 237–248, New York, NY, USA, 2013. Association for Computing Machinery.

[9] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *ACM Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 14–22, 2019.

[10] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson. TENEX, a Paged Time Sharing System for the PDP-10. *Communications of the ACM (CACM)*, 15(3):135–143, 1972.

[11] Jonathan Corbet. Clarifying memory management with page folios. https://lwn.net/Articles/849538/, 2021. Accessed: 2023-12-03.

[12] Jonathan Corbet. Large folios for anonymous memory. *LWN.net*, 2023. Accessed: 2024-01-09.

[13] Charles D Cranor and Gurudatta M Parulkar. The UVM virtual memory system. In *USENIX Annual Technical Conference (ATC)*, 1999.

[14] Matthew Dillon. Design elements of the FreeBSD VM system. https://docs.freebsd.org/en/articles/vm-design/, 2023. Accessed: 2024-01-10.

[15] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS), page 37–48, New York, NY, USA, 2012. Association for Computing Machinery.

[16] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, page 178–189, USA, 2014. IEEE Computer Society.

[17] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with Infiniswap. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 649–667, Boston, MA, March 2017. USENIX Association.

[18] David Hildenbrand, Martin Schulz, and Nadav Amit. Copy-on-pin: The missing piece for correct copy-on-write. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2023.

[19] David Hildenbrand, Martin Schulz, and Nadav Amit. Software artifacts for the paper "Every Mapping Counts in Large Amounts: Folio Accounting". https://doi.org/10.5281/zenodo.11401995, May 2024.

[20] Jae Young Hur. Contiguity representation in page table for memory management units. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(1):147–158, 2019.

[21] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *IEEE/ACM International Symposium on Microarchitecture*, page 24–35, New York, NY, USA, 2011. Association for Computing Machinery.

[22] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, volume 1. Addison-Wesley, 1968.

[23] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with Ingens. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 705–721, 2016.

[24] Taowei Luo, Xiaolin Wang, Jingyuan Hu, Yingwei Luo, and Zhenlin Wang. Improving tlb performance by increasing hugepage ratio. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015.

[25] Matthew Wilcox. Memory folios in the linux kernel. https://www.infradead.org/~willy/linux/2022-06_LCNA_Folios.pdf, 2022. Accessed: 2024-05-28.

[26] Matthew Wilcox. Shrinking memmap: Saving memory for fun and profit. https://www.infradead.org/~willy/linux/2022_11_Shrinking_Memmap.pdf, 2022. Accessed: 2024-05-28.

[27] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley Longman Publishing Co., Inc., USA, 1996.

[28] OnePlusOSS. Android kernel for OnePlus SM8550. https://github.com/OnePlusOSS/android_kernel_oneplus_sm8550.git, 2024. Accessed: 2024-01-10.

[29] Ashish Panwar, Sorav Bansal, and K Gopinath. Hawkeye: Efficient fine-grained OS support for huge pages. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 347–360, 2019.

[30] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, page 679–692, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. Perforated page: Supporting fragmented memory allocation for large pages. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, page 913–925. IEEE Press, 2020.

[32] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. pages 444–456, 2017.

[33] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced large-reach TLBs. In *IEEE/ACM International Symposium on Microarchitecture*, pages 258–269. IEEE, 2012.

[34] Ryan Roberts. Multi-size THP for anonymous memory. Linux Kernel Mailing List, https://lore.kernel.org/linux-mm/20231214160251.3574571-1-ryan.roberts@arm.com/T/, 2023.

[35] Luis Sena. Sharing big NumPy arrays across python processes. https://luis-sena.medium.com/sharing-big-numpy-arrays-across-python-processes-abf0dc2a0ab2, 2021. Accessed: 2024-01-10.

[36] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 69–87, Carlsbad, CA, 2018. USENIX Association.

[37] Eliot H. Solomon, Yufeng Zhou, and Alan L. Cox. An empirical evaluation of PTE coalescing. In *International Symposium on Memory Systems (MEMSYS)*, 2023.

[38] Madhusudhan Talluri and Mark D. Hill. Surpassing the tlb performance of superpages with less operating

system support. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, ASPLOS VI, page 171–182, 1994.

[39] Zi Yan, David Nellans, Daniel Lustig, and Abhishek Bhattacharjee. Translation ranger: Operating system support for contiguity-aware TLBs. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 698–710, 2019.

[40] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*, page 266–282, New York, NY, USA, 2023. Association for Computing Machinery.

[41] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 540–555, 2021.

[42] Weixi Zhu, Alan L Cox, and Scott Rixner. A comprehensive analysis of superpage management mechanisms and policies. In *USENIX Annual Technical Conference (ATC)*, pages 829–842, 2020.