The New Costs of Physical Memory Fragmentation

Alexander Halbuer*
Leibniz Universität Hannover

Illia Ostapyshyn
Leibniz Universität Hannover

Lukas Steiner
Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau

Lars Wrenger
Leibniz Universität Hannover

Matthias Jung
Universität Würzburg and Fraunhofer IESE

Christian Dietrich
Technische Universität Braunschweig

Daniel Lohmann
Leibniz Universität Hannover

Abstract

External fragmentation is becoming a serious problem again after paging temporarily solved it with its one-size-fits-all 4 KiB approach. The increasing adoption of mixed base, huge, and giant page sizes, DRAM energy-saving techniques, and memory disaggregation, necessitates a memory management system capable of handling larger entities in the range of multiple megabytes up to several gigabytes.

A case study in Linux reveals that the operating system reasonably minimizes fragmentation up to huge page size, but falls short when it comes to larger granularities. Therefore, it requires much effort to entirely free a memory block for powering down or returning it to the memory provider; in some cases, this may be entirely impossible due to immovable kernel memory.

Additionally, our analysis highlights that the page cache is responsible for a large share of memory usage, as it keeps all cached pages until memory pressure rises. This behavior originates from the outdated assumption that utilizing memory comes at no cost and, therefore, requires further investigation.

CCS Concepts: • **Hardware** → *Power estimation and optimization*; **Memory and dense storage**; • **Software and its engineering** → **Memory management**.

Keywords: Operating Systems, Memory Management, Fragmentation, Physical Memory, Linux, Energy Savings, DRAM, Distributed Memory

ACM Reference Format:
Alexander Halbuer, Illia Ostapyshyn, Lukas Steiner, Lars Wrenger, Matthias Jung, Christian Dietrich, and Daniel Lohmann. 2024. The New Costs of Physical Memory Fragmentation. In *2nd Workshop on Disruptive Memory Systems (DIMES '24), November 3, 2024, Austin, TX, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3698783.3699378

*Corresponding author (halbuer@sra.uni-hannover.de)

DIMES '24, November 3, 2024, Austin, TX, USA

1 Introduction

Since its introduction in ATLAS [10, 18], page-based memory management has become the de facto standard for access, sharing, and virtualization of main memory. Even access to block-oriented storage (i.e., disk files) is implemented by the kernel via means of memory objects [29] and the unified page cache [5]. The 4 KiB page frame has become the de facto entity for everything: user memory, kernel memory, page cache, and IO buffers. The OS relies on it for virtualization techniques, such as demand paging and COW [30], to hand out memory as late as possible. Also, the uniform size allows the page cache to expand into all gaps, aggressively utilizing the remaining memory to speed up file accesses.

The underlying assumptions here are that *"All page frames are equal"* and *"Only used memory is good memory"* (because unused memory is basically a *stranded asset* [21]). These assumptions were fundamental to the elegance of the original Mach memory model [29], whose ideas and concepts can still be found in the memory subsystems of all modern operating systems. However, on the hardware side, some things have changed over the last 35 years:

Fragmentation is back While the immunity to external fragmentation of the physical memory was one of the original motivations for paging [3], its extension to simultaneously support multiple frame sizes to mitigate page-table overhead and TLB pressure brought back the problem. The OS now has to deal not only with 4 KiB *base*, but also with, for example, 2 MiB *huge* and 1 GiB *giant* pages and has to ensure that frames of all sizes are available.[1] To further mitigate virtual-memory overhead in big-memory workloads Basu et al. propose *direct segments* [2], one arbitrary large page per address space, which would make external fragmentation even worse. Moreover, given that some frames have to be considered as non-movable or restricted regarding DMA accessibility, framentation seems to be a notoriously hard

[1] Without loss of generality, we stick to the x86-64 page sizes within this paper. Even if some architectures use different or support multiple base, huge, and giant page sizes, the general fragmentation problem remains.

problem, as a large body of research around pooling and placement strategies [6, 7, 9, 28] and (pro)active compaction and reservation [19, 20, 26] shows.
The bottom line is that all page frames are *not* equal.

Using memory comes at a cost Utilizing free memory no longer comes for free: The page cache is (as we show in this paper) a major contributor to fragmentation, leading to secondary costs regarding huge-page reclamation. Also, due to the much improved access times and bandwidth of modern SSDs, the benefit of caching is declining.

Besides contributing to physical-memory fragmentation, aggressively using all memory also results in missed opportunities: DRAM already accounts for over 30 percent of a racks power consumption [24], and the average utilization in cloud environments is around 75 percent [33], so it might be worthwhile to consider powering off the unused memory. While current memory controllers provide only limited possibilities in this respect, we expect this to become more prevalent in the future. This is certainly true with the emerging CXL [4] technology, which allows building disaggregated memory pools [21], making physical memory a redistributable commodity within a rack. For the cloud, this creates the opportunity for much more flexible pricing models, giving guest machines an incentive to return memory to the host [11, 12]. However, all this will be possible only at much higher granularities than individual 4 KiB base frames. And only a few occupied page frames may hinder the shutdown of a DRAM bank or reassigning a memory area to another server. For example, with CXL.mem the minimum meaningful granularity is 64 MiB as protections can not be applied on a finer level, and due to the limited number of shared memory regions per client the block size may be even larger in practice [31]. The bottom line is that it actually can (even economically) pay off to *not* use memory.

In this paper, we analyze and discuss the issue of physical memory fragmentation in Linux, without questioning the established paging memory-virtualization technique itself. Instead, we approach the problem from a software perspective. We look at the causes and costs of fragmentation as well as possible mitigations and benefits of reduced fragmentation. The underlying research is still at an early stage, so we do not claim soundness nor completeness of our first results and ideas, but instead want to foster the discussion.

2 Case Study: Fragmentation Patterns

To understand the current state of Linux, we conduct a case study analyzing the physical-memory fragmentation and occupation. Our study aims to answer the following questions: (1) How does Linux manage the physical memory space? (2) How is physical memory fragmented at different granularities? (3) How much can we improve fragmentation with (limited) active memory compaction?

Test Setup We use a QEMU virtual machine (VM) equipped with 16 GiB of memory and 12 CPU cores running Debian 12 with Linux kernel version 6.1. This setup allows us to pause execution, enabling accurate sampling of memory usage. Also, the measurements do not interfere with the benchmark execution because they run on the hypervisor side and do not utilize memory inside the VM.

The additional abstraction layer may affect the timing behavior compared to running directly on physical hardware but does not alter the memory usage and allocation as they depend on the application demands and the OS's allocation strategy, not the underlying machine. Notably, in large cloud environments where savings and optimizations have the greatest impact, applications often run within virtual machines. Due to nested paging, fragmentation can be handled independently at both the VM and host levels. Although more sophisticated techniques would be possible, where memory management decisions within the VM could directly benefit host-level fragmentation, these would come at the expense of increased complexity.

While we assert that memory usage in virtual and physical machines should be comparable, this assumption needs further investigation to confirm its validity.

Benchmark Scenario Simulating fragmentation is challenging, especially for long-running systems, as there is no single representative workload [22]. Consequently, we do not claim that our results, derived from a single application, are universally applicable to other applications, and additional research is necessary to get a holistic view.

We have chosen the build process of the Clang compiler from the LLVM project as a suitable starting point for our analysis. This workload involves frequent (de-)allocations of anonymous application memory, as well as intense file system usage resulting in high page cache activity.

To ensure reproducibility, we start the build on a freshly booted system. After the build completes, we remove all intermediate build files with `make clean`. Finally, we drop[2] the page cache and other reclaimable kernel objects, such as file-system metadata, to return to a clean system state [32].

Methodology During this scenario, we take snapshots of the physical-memory state at five points in time:

(A) Right after boot of the virtual machine,
(B) during the build process, at about minute 5 of 30,
(C) when the build has finished,
(D) after running `make clean`,
(E) and after dropping the caches.

Using the kernel's page frame descriptors (`struct page`), we categorize all page frames into one of four classes:

- FREE (not allocated),
- IMMOVABLE (kernel memory),
- MOVABLE (conventional application memory), and

[2] `echo 3 > /proc/sys/vm/drop_caches`

Snapshot		FREE	IMMOVABLE	MOVABLE	DROPPABLE
(A)	After boot	4 072 099 (97.09%)	110 019 (2.62%)	10 463 (0.25%)	1 723 (0.04%)
(B)	Building Clang	2 360 497 (56.28%)	128 879 (3.07%)	1 425 173 (33.98%)	279 755 (6.67%)
(C)	Finished build	2 454 473 (58.52%)	171 111 (4.08%)	10 516 (0.25%)	1 558 204 (37.15%)
(D)	After clean	3 914 103 (93.32%)	132 176 (3.15%)	10 516 (0.25%)	137 509 (3.28%)
(E)	Dropped caches	4 070 843 (97.06%)	111 571 (2.66%)	10 516 (0.25%)	1 374 (0.03%)

Table 1. Number of page frames per type at each sample point (4 194 304 page frames in total).

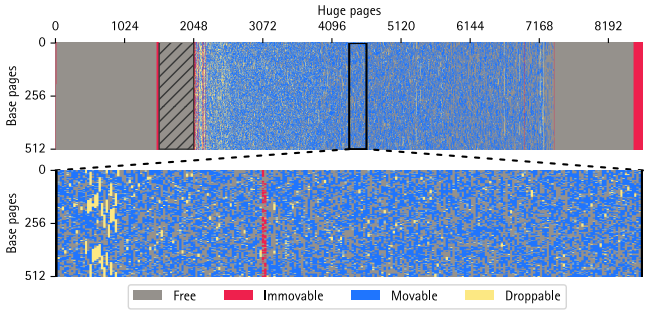

Figure 1. Visualization of the physical memory usage during the build process (B): The range 1536-2048 is an 1 GiB wide memory hole on the evaluation system. The zoom magnifies one cutout to show the distribution on page-frame level.

- DROPPABLE (page cache memory).

Although dirty DROPPABLE frames have to be written out to disk before eviction, we do not further distinguish between dirty and clean pages as Linux' automatic write-back mechanism keeps the ratio of dirty frames low.

2.1 Physical-Memory Occupation

Tab. 1 shows the amount of different frame types for each snapshot: Initially (A), nearly all memory is FREE on the freshly booted system. This amount decreases during the build process (B) where about one third of all memory is application memory (MOVABLE). After the build (C), the number of free frames stays slightly above the previous level. Most memory is used by the page cache (DROPPABLE) as it keeps recently accessed file data in memory even if the corresponding process has already exited. In contrast, the number of MOVABLE frames nearly matches its initial value. Removing the intermediate build files (D) releases most page cache entries. The clean operation also releases 38 935 IMMOVABLE page frames, which likely have been used for managing the deleted files and their page cache entries. Dropping the caches (E) almost brings the physical memory back to the initial state; at least from this summary perspective. However, there are still some DROPPABLE frames left because they are either dirty and, therefore, have not been dropped [32], or they have been requested between the drop and the snapshot.

Allocation Patterns While the summary of (A) and (E) are quite close, both actually differ significantly in their physical-memory states. To approach this difference, we first look at the distribution of frame types at the height of the build process at (B). As we can see in Fig. 1, the used memory is not as compact as it could be. The areas with used memory contain many holes in the size of a single or up to a few frames, as well as 2 MiB naturally aligned clusters of IMMOVABLE memory. The latter are due to the Linux page frame allocator's policy of favoring 2 MiB areas (`pageblocks`) containing pages of the same type when allocating. On the other hand, MOVABLE and DROPPABLE frames are completely mixed.

2.2 Memory Fragmentation

With the premise that DRAM can be turned off at many different granularities (Sec. 3.2), we quantify the memory fragmentation of our physical-memory states using various block sizes: Starting with the smallest possible block size of 4 KiB, which is a single page frame, we double the block size repeatedly. This process continues up to a size of 8 GiB, which is half of the available memory. If a (naturally-aligned) block only contains free memory, the whole block accounts as usable (for the current block size), otherwise it counts as unusable. In Fig. 2, the red baseline shows the free blocks for different sizes at all five sample points. By comparing the 4 KiB data point with the following decline, we get an impression of the memory state: the later the curve declines, the lesser is the fragmentation.

Active Memory Compaction Next, we want to quantify how much better the fragmentation could get if we would perform active memory compaction. As compaction is costly, an OS only performs it incrementally and with a limited budget of dropped and moved frames. For example, Linux triggers such a compaction only under memory pressure, assuming that "all pages are equal". To capture this, we define a *cost function* as the number of touched page frames: #DROPs+2×#MOVEs. Because moving a page is more costly and touches two page frames (source and destination), we make it twice as costly as dropping a frame from the page cache.

On each recorded state and targeted at a specific block size, we virtually perform compaction: From the block with the least costs to free, we drop all frames or move them to the block with the highest costs that still can accommodate it. We repeat this iteratively until the budget runs out. This algorithm is optimal and results in the highest reduction in fragmentation for a given budget and block size.

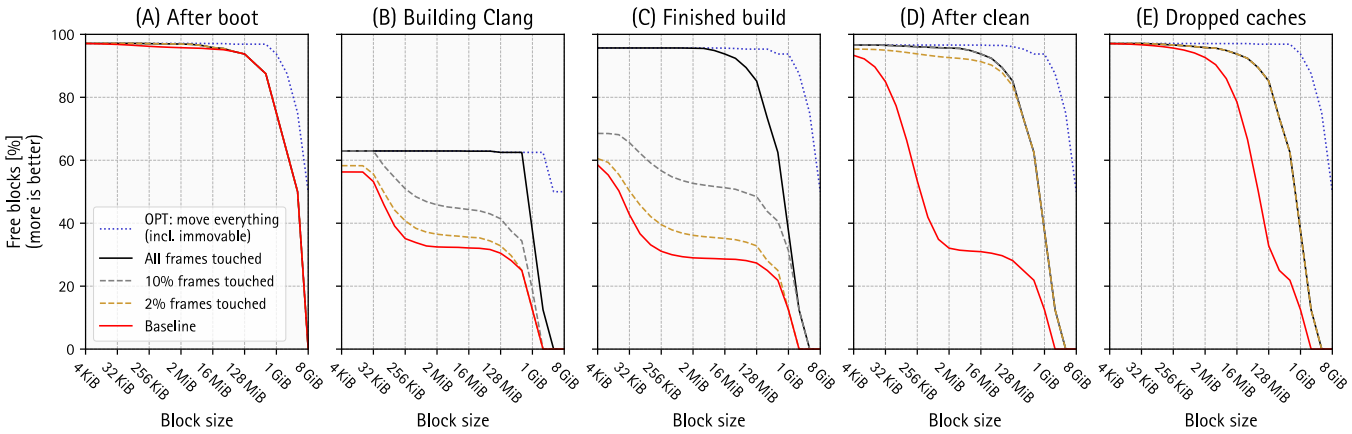

Figure 2. Potential memory savings in a compilation scenario for different block sizes with the number of touched page frames (drop: 1x, move: 2x) as cost function for active compaction. The black line marks the achievable maximum, the dotted blue line the theoretical optimum (if we could move IMMOVABLE frames).

In Fig. 2, we look at the fragmentation after compacting with three different budgets: With an unlimited budget (black solid), we can drop or move all MOVABLE and DROPPABLE frames resulting in the upper limit that active compaction can reach in Linux. The dashed lines show the fragmentation if we compact with a budget of touching up to 2, respectively, 10 percent of all physical memory. We consider this a realistic budget for limited active defragmentation that would not disrupt user applications. The blue dotted line shows the theoretical optimum if we were able to move IMMOVABLE frames.

Initially (A), the baseline and the achievable maximum are nearly overlapping for all block sizes, indicating that active compaction is not required as there is no fragmentation.

During the build process (B), the theoretical optimum decreases as there is non-droppable application memory (Tab. 1). For medium block sizes (2 MiB – 16 MiB) the baseline of free blocks falls to 32 percent and the achievable average improvement is quite low with 3.7 (12.7) percent of additional free blocks for 2 (10) percent of all frames touched. This is to be expected as anonymous memory requires expensive migration.

After the build process (C) the system is in idle state, but the baseline is similar to the previous snapshot. On the other hand, the theoretical optimum returns to above 90 percent for low and medium block sizes. Also, the achievable gain increases to 6.8 percent (23.1%) of additional free blocks for medium block sizes, because most memory is now used by DROPPABLE page cache frames.

Executing clean (D) significantly improves the baseline for small block sizes but barely affects medium and high block sizes. The maximum and theoretical optimum lines are marginally higher, but now the 10-percent line matches the achievable maximum and even the 2-percent line is very close. As deleting the intermediate files also removes the related page cache entries the physical memory now is only sparsely used. Due to the scattered distribution of non-free page frames only low effort is required to reclaim the sparsely filled blocks.

The final cache drop (E) nearly empties the page cache completely and thereby drastically improves the baseline for medium page sizes. Even less effort is required to reclaim the remaining blocks so that also the 2-percent line matches the achievable maximum. Important to note is, that even if the quantitative view (Tab. 1) indicates (A) and (E) being mostly equal, the physical-memory state does not return to its initial unfragmented state.

Immovable Kernel Memory In Linux, compaction has to leave IMMOVABLE memory in place, also limiting the maximal effectiveness of the approach. Therefore, we finally look at the theoretical optimum (Fig. 2, dotted line) if all memory pages were movable and we compact with an unlimited budget, resulting in a maximal compact physical-memory occupation. From the gap to the maximal achievable number of free blocks (solid black), we can see that even a low number of IMMOVABLE pages (see Tab. 1) has a significant effect on fragmentation for large block sizes.

2.3 Discussion

The cases study confirms our claims that a modern operating system, like Linux, is based on the design principle that free memory is wasted memory. The cached file data remains in main memory after the build process, until it is explicitly dropped or forced out by new allocations causing memory pressure. As one would expect, the share of free blocks reduces with an increasing block size because a single page frame is enough to render a block unusable.

With limited effort (2% or 10% of all page frames touched) it is possible to reduce fragmentation considerably. The actual improvement depends on the system state and the block

Power Saving Mode	Granularity	Power Saving [mW/GiB]		Add. Latency	Data Retention
1. Full Power Off	Channel (64 GiB)	82.8	(100%)	> 25 ms	no
2. Self-Refresh [14]	Channel (64 GiB)	0.53	(0.6%)	640 ns	yes
3. MPSM Deep Power Down [14]	Channel (64 GiB)	22.95	(27.7%)	640 ns	no
4. MPSM Power Down [14]	Rank (16 GiB)	15.79	(19.1%)	21.5 ns	no
5. Power Down [14]	Rank (16 GiB)	14.6	(17.6%)	7.5 ns	yes
6. Partial Array Self-Refresh (PASR) [13]	1/8 Rank (2 GiB)	0.53 - 22.95	(0.6% - 27.7%)	640 ns	no
7. Partial Array Refresh Control (PARC) [13]	1/8 Rank (2 GiB)	0 - 9.19	(0% - 11.1%)	14 ns	no
8. Row-Granular Refresh [25]	Row (8 KiB)	0 - 9.19	(0% - 11.1%)	0	no

Table 2. Potential power savings estimated with the equations of DRAMPower [17].

size, where larger blocks are harder to reclaim. Generally, active defragmentation techniques should be the last option as they induce overhead and can degrade application performance [23]. Passive allocation and page cache policies could be more effective and could further reduce the residual work for active techniques.

3 Costs of Fragmentation

Fragmentation incurs primarily indirect costs, from execution time to energy consumption. In this section, we provide a brief overview of the implications of fragmentation in three areas that are already problematic, or we expect to become challenging in the future.

3.1 Availability of Huge Pages

The observation that the TLB reach can be significantly extended through the use of huge pages [8] led to the introduction of *Transparent Huge Pages (THP)* in Linux 2.6.38 [15]. This optimization eagerly maps huge pages instead of base pages and periodically promotes pages with a background task. However, the effectiveness and performance gains of THP highly depend on the availability of huge pages, directly reflecting physical memory fragmentation [28]. Notably, the costs associated with active memory compaction may significantly outweigh any potential performance benefits [23, 27].

3.2 Energy Consumption

DRAM power consumption could be reduced if the memory is not fully occupied or not accessed for an extended period of time. Current DRAM standards offer several energy saving modes, that either turn off refresh for unused regions or disable unnecessary logic during idle intervals.

To estimate potential savings, we consider a specific DDR5 configuration, where we apply different existing or conceptually adapted (from another DRAM technology) power saving modes. Foundation of our survey is the Micron MT60B4G4 2 GiB DDR5 device. The complete DRAM subsystem consists of eight memory channels, providing a total storage capacity of 512 GiB (64 GiB per channel). Each channel is composed of four 16 GiB ranks, comprising eight MT60B4G4 chips each. We assume 32 banks and 65 536 rows per bank, resulting in a typical 8 KiB row figure.

Table 2 presents the potential power savings in different modes for an idle DRAM subsystem, with a conventionally, cyclically refreshed system as baseline. The table also specifies the granularity of each mode, the additional latency required to return to normal operation, and whether each mode retains data. It should be noted that the operating currents listed in the datasheet are specified for worst-case conditions (manufacturing process, temperature, and voltage). In practice, measured currents can be significantly lower, which may affect the results presented here. On the other hand, our analysis does not account for secondary energy savings that arise from reduced power supply losses and lower cooling requirements. For more accurate estimations of achievable savings in real-world settings, measurements of physical hardware would be necessary.

As ranks are addressed independently, all per-device power-saving techniques theoretically could be applied at the rank granularity. However, some techniques (1–3) disable on-die termination (ODT) and thereby violate impedance matching for the whole channel, making them incompatible with high data rates when applied to a single rank. For instance, completely powering off (1) an unused memory channel (64 GiB) saves 5.3 W or 82.8 mW/GiB. On the finer granularity, switching a rank (16 GiB) into *Power Down* (5) mode saves 233.65 mW or 14.6 mW/GiB. Here, the data could be retained by periodically interrupting the power down mode and issuing a refresh command.

In the *Self-Refresh* (2) mode the data is retained by an internal refresh mechanism, but no memory accesses are possible. PASR (6) introduces a mask register to disable data retention for 1/8 rank segments (2 GiB) during Self-Refresh, enabling potential energy savings of up to 22.95 mW/GiB. The LPDDR5 standard introduces *PARC* (7), adopting the PASR segment mask for refresh control during normal operation. With PARC, in contrast to PASR, the memory in unmasked segments can be accessed normally.

Mathew M. et al. (8) bring the granularity to the level of OS paging by disabling refresh and issuing activation commands on used rows manually [25]. They optimize the mechanism by relaxing DRAM timing requirements and observe better energy efficiency than during conventional

(a) Varying Instance Memory **(b)** Varying CPU Count

Figure 3. Price of Amazon AWS EC2 Instances. Dots are all instance types. The lines denote the cheapest type for a CPU/memory combiantion.

full refresh. However, the success of this technique may vary from device to device.

Although these values may appear minor compared to the total power consumption of a server system, they can accumulate to substantial savings when scaled up to an entire rack or data center. Moreover, this analysis underlines the importance of performing physical memory management at much larger granularities than huge page size.

3.3 Cloud Memory Pricing

Due to economies of scale, cloud computing is probably the area where uni- and bilateral memory reclamation has the highest potential for economic impact. Currently, providers offer CPU and memory only as a *composite good* (virtual machine), limiting the efficiency and price transparency of the cloud market. Still, by looking at current offerings, we can estimate whether memory or CPU is the more scarce resource.

We take the price matrix for all on-demand Amazons AWS EC2 instance types (2024/08/07, all regions) without distinguishing between different CPU or memory types, and compare the per-hour cost with the included resources (dots in Fig. 3). For each CPU/memory combination, we select the cheapest instance type and draw *iso-resource lines* (lines in Fig. 3). Please note the double-logarithmic scale that we use to cover a large range of the matrix.

First, we see that cloud providers do not offer the full range of VM types. You simply cannot rent a 2 vCPU machine with 2048 GiB of DRAM, even if this would make sense for a low-latency caching service. Second, doubling instance memory results in a consistent price jump (see Fig. 3a), while doubling the CPU count of machines with large memories has diminishing effect. Fig. 3a even reveals pricing anomalies where increasing the CPU count results in a lower price (e.g., 32 vs. 64 vCPUs at 1024 GiB). Memory seems to be the more scarce resource.

We believe that cloud providers will start to offer discounts for customers that are (1) able and (2) willing to give back memory as soon as redistribution is technologically viable (via CXL [4]). In this future, the system software has to (1) better manage physical-memory fragmentation and (2) to assure that the utility of using memory outweighs its cost. Page cache pages need to have a per-hour price tag.

4 Towards Less Fragmentation

Based on the results from the case study, we identified the following causes for fragmentation in Linux especially for large block sizes.

4.1 Page-Frame Allocation

The kernel's page-frame allocator is our first citizen, as it ultimately decides which memory to hand out. While the Linux buddy allocator is mostly successful in grouping immovable page frames within 2 MiB areas [8, 28], it has also shown an unfortunate tendency to scatter all other allocations across the available memory (Sec. 2.2). The latter is mostly caused by core-local frame caches, which significantly reduce contention on the allocator lock, but lead to blindness regarding the global allocation and fragmentation state [34]. Also, the maximum block size of the buddy allocator is 4 MiB, which is not enough for most DRAM power saving techniques (Tab. 2).

Alternative allocator designs [34] have demonstrated much better scalability and fragmentation avoidance. Furthermore, the lock-free access to the global allocation state makes them more accessible for active defragmentation, as the respective algorithms can efficiently search for almost-free blocks of higher granularities.

4.2 Movable Kernel Design

Additionally, reducing the number of immovable allocations would be highly advantageous. This could be achieved by making (most) kernel allocations movable. If the kernel would not be designed around an identity mapping, allocations could be moved and remapped, similar to user space. However, DMA buffers are more challenging. Although they can be remapped via the IOMMU, the handling of page faults (needed by active compaction mechanisms) is not yet supported by all devices and TLB invalidations are even more expensive on an IOMMU compared to the MMU [1].

Nevertheless, a kernel design in which *only* some DMA buffers need to be immovable would already be a big step forward.

4.3 Clever File Caching

The page cache is a major contributor to both, memory usage and fragmentation. Given that unused memory is no longer wasted and employing it for caching comes at a cost, we

should find a balance between costs and benefits of a page to being held in the cache.

For user-memory, the idea of a cost-benefit model is not new: Mansi et al. employ a system-specific cost model and an application-specific benefit profile to weight up memory-management decision, like huge page promotion, with the goal of improving page-fault tail latency and end-to-end performance [23]. They use units of time for their quantification of costs and benefits. Energy and memory usage fees would add other dimensions to the optimization problem, which may be contrary to performance.

However, the page cache is fundamentally different in that it is (a) a global resource, which means that cache entries are currently not accountable onto individual tenants, and (b) the actual costs and benefits are much more dynamic, depending on the overall system state.

On the cost side, we expect having quantifiable costs per memory block of a given size. These costs may be uniformly distributed among the used frames within each block. The last frame that prevents an entire block from becoming free incurs a higher cost compared to a frame residing in an almost-full memory block. The cost model could have multiple granularity levels. For instance, these could be huge and giant page sizes, where a free giant page is more valuable than a free huge page, or technical levels regarding possible energy reductions, such as rows, ranks, or even an entire CXL memory node. Costs may also be dynamic, influenced by parameters such as memory pressure on the (guest) system, the availability of huge/giant pages, the current electricity price or current memory usage fees in a cloud environment. If we could integrate all these aspects into a single cost metric, we could begin to weigh a page's costs against its benefits.

The primary advantage of caching a page is the time saved on subsequent accesses. This benefit depends on two factors: access frequency and secondary storage latency. The currently employed LRU/second chance algorithms already weight by access frequency. Similarly, data stored on remote servers or HDDs would benefit more from caching than data stored on local SSDs [35]. For fast and directly accessible storage, using the page cache may not pay off at all [16].

Quantified costs and benefits would provide for an informed decision-making. Ultimately, they require, however, additional per-frame metadata that needs to be efficiently stored and managed. This is a topic of further research.

5 Conclusion

Current Linux memory management tends to fragmentation when considering blocks larger than huge pages and utilizes nearly all unused memory to cache file data. Historically, there has been little incentive to mitigate such fragmentation or drop cached data prematurely. However, we anticipate a shift in this paradigm with the increasing adoption of giant pages, new power-saving features, and flexible cloud storage pricing models.

Hurdles towards a better fragmentation avoidance are the allocator design, unmovable kernel memory and the page cache. While possible solutions for the first two points exist, the latter has – as far as we know – not been thoroughly studied in this regard.

We propose future research looking at a cost-aware redesign of the page cache that does not hold unconditionally all elements forever until they are forced out by memory pressure. Quantified costs and benefits would enable informed decision-making for both memory management and page cache policies.

Acknowledgments

We thank our reviewers for their valuable feedback. This work was partly supported by the German Research Foundation (DFG) under grant no. 501887536.

References

[1] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2012. IOMMU: Strategies for Mitigating the IOTLB Bottleneck. In *Computer Architecture*, Ana Lucia Varbanescu, Anca Molnos, and Rob van Nieuwpoort (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–274.

[2] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers. *SIGARCH Comput. Archit. News* 41, 3 (jun 2013), 237–248. https://doi.org/10.1145/2508148.2485943

[3] A. Bensoussan, C. T. Clingen, and R. C. Daley. 1969. The Multics Virtual Memory. In *Proceedings of the 2nd ACM Symposium on Operating Systems Principles (SOSP '69)*. ACM Press, New York, NY, USA, 30–42. https://doi.org/10.1145/961053.961069

[4] Compute Express Link Consortium, Inc. 2020. *CXL Specification, Revision 2.0*.

[5] Robert A. Gingell, J. Moran, and William Shannon. 1987. Virtual Memory Architecture in SunOS. In *Proceedings of Summer '87 USENIX Conference*.

[6] Mel Gorman. 2007. The performance and behaviour of the anti-fragmentation related patches. Linux Kernel Mailing List. https://lkml.org/lkml/2007/3/1/92 https://lkml.org/lkml/2007/3/1/92.

[7] Mel Gorman and Patrick Healy. 2008. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management - ISMM '08*. ACM Press, Tucson, AZ, USA, 41. https://doi.org/10.1145/1375634.1375641

[8] Mel Gorman and Patrick Healy. 2012. Performance characteristics of explicit superpage support. In *Computer Architecture: ISCA 2010 International Workshops A4MMC, AMAS-BT, EAMA, WEED, WIOSCA, Saint-Malo, France, June 19-23, 2010, Revised Selected Papers 37*. Springer, 293–310.

[9] Mel Gorman and Andy Whitcroft. 2006. The What, The Why and the Where To of Anti-Fragmentation. In *Proceedings of the Linux Symposium*, Vol. Volume 1. Ottawa, Ontario, Canada, 370–384.

[10] Fritz Rudolf Güntsch. 1957. Logischer Entwurf eines digitalen Rechengerätes mit mehreren asynchron laufenden Trommeln und automatischem Schnellspeicherbetrieb.

[11] David Hildenbrand and Martin Schulz. 2021. virtio-mem: paravirtualized memory hot(un)plug. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Virtual, USA) *(VEE 2021)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3453933.3454010

[12] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2018. HUB: hugepage ballooning in kernel-based virtual machines. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, Virginia, USA) *(MEMSYS '18)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/3240302.3240420

[13] JEDEC 2023. *Low Power Double Data Rate (LPDDR) 5/5X*. JEDEC.

[14] JEDEC 2024. *DDR5 SDRAM*. JEDEC.

[15] Jonathan Corbet. 2011. Transparent huge pages in 2.6.38. https://lwn.net/Articles/423584/ Accessed: 2024-08-07.

[16] Jonathan Corbet. 2017. The future of the page cache. https://lwn.net/Articles/712467/ Accessed: 2024-08-08.

[17] Matthias Jung, Deepak M. Mathew, Éder F. Zulian, Christian Weis, and Norbert Wehn. 2016. A New Bank Sensitive DRAMPower Model for Efficient Design Space Exploration. In *International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS 2016)*.

[18] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner. 1962. One-Level Storage System. *IRE Transactions on Electronic Computers* EC-11, 2 (April 1962), 223–235. https://doi.org/10.1109/TEC.1962.5219356

[19] Sang-Hoon Kim, Sejun Kwon, Jin-Soo Kim, and Jinkyu Jeong. 2015. Controlling Physical Memory Fragmentation in Mobile Systems. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)* (Portland, OR, USA). Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2754169.2754179

[20] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *12th Symposium on Operating Systems Design and Implementation (OSDI '16)* (Savannah, GA, USA). USENIX Association, USA, 705–721.

[21] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23), Volume 2* (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3575693.3578835

[22] Mark Mansi and Michael M. Swift. 2024. Characterizing Physical Memory Fragmentation. arXiv:2401.03523 [cs.OS] https://arxiv.org/abs/2401.03523

[23] Mark Mansi, Bijan Tabatabai, and Michael M. Swift. 2022. CBMM: Financial Advice for Kernel Memory Managers. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 593–608. https://www.usenix.org/conference/atc22/presentation/mansi

[24] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23), Volume 3* (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[25] Deepak Mathew M., Matthias Jung, Christian Weis, and Norbert Wehn. 2017. Using Runtime Reverse Engineering to Optimize DRAM Refresh.

[26] Juan Navarro, Sitaram Iyer, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. USENIX Association, Boston, MA.

[27] Ashish Panwar, Naman Patel, and K. Gopinath. 2016. A Case for Protecting Huge Pages from the Kernel. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, Hong Kong) *(APSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 15, 8 pages. https://doi.org/10.1145/2967360.2967371

[28] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[29] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. 1987. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS '87)* (Palo Alto, California, USA) *(ASPLOS '87)*. IEEE Computer Society Press, Washington, DC, USA, 31–39. https://doi.org/10.1145/36206.36181

[30] Richard F. Rashid and George G. Robertson. 1981. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*. ACM Press, New York, NY, USA, 64–75. https://doi.org/10.1145/800216.806593

[31] Samuel W. Stark, A. Theodore Markettos, and Simon W. Moore. 2023. How Flexible is CXL's Memory Protection? Replacing a sledgehammer with a scalpel. *Queue* 21, 3 (July 2023), 54–64. https://doi.org/10.1145/3606014

[32] The kernel development community. 2008. Documentation for /proc/sys/vm/ - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html Accessed: 2024-08-06.

[33] Yaohui Wang, Ben Luo, and Yibin Shen. 2023. Efficient Memory Overcommitment for I/O Passthrough Enabled VMs via Fine-grained Page Meta-data Management. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 769–783. https://www.usenix.org/conference/atc23/presentation/wang-yaohui

[34] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. 2023. LLFree: Scalable and Optionally-Persistent Page-Frame Allocation. In *2023 USENIX Annual Technical Conference (USENIX '23)*. USENIX Association, Boston, MA, 897–914. https://www.usenix.org/conference/atc23/presentation/wrenger

[35] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '15)*. Association for Computing Machinery, New York, NY, USA, Article 6, 11 pages. https://doi.org/10.1145/2757667.2757684