Physical memory anti-fragmentation mechanisms in the FreeBSD kernel

Bojan Novković
Zagreb, Croatia
bnovkov@freebsd.org

***Abstract*—The use of virtual memory practically eliminated the need for the contiguity of physical memory allocation as physically discontiguous memory can be contiguous in the virtual address space. Unfortunately, avoiding performance degradation for memory-intensive workloads on modern CPUs requires a steady supply of contiguous physical memory, making external physical memory fragmentation in modern operating systems a serious issue once again.**

This paper presents the design and implementation of several anti-fragmentation mechanisms for the FreeBSD kernel. Parts of this work were sponsored by the *Google Summer of Code '23* program.

I. Introduction

Memory fragmentation is a long-standing problem in computer science that can often have dire consequences for the performance or memory capacity of a running computer system. Historically, the introduction of virtual memory and scatter-gather DMA transfers made computer systems virtually insensitive to physical memory fragmentation. Unfortunately, the same cannot be said for modern computer systems where new page sizes were introduced to reduce the performance hit from the virtual address translation process. The performance of memory-intense workloads has once again become very sensitive to the rate of external fragmentation in physical memory.

This paper gives an overview of recent work done to reduce the rate of external fragmentation in the FreeBSD operating system. Section II gives the necessary theoretical background and presents the problem at hand. Section III describes the details related to the design of several new anti-fragmentation mechanisms in the FreeBSD kernel. We describe our experimental setup in Section IV, analyze and discuss the evaluation results in Section V, and conclude the paper in Section VII.

II. Background

A. Physical memory allocation

All physical memory allocation in the FreeBSD kernel is done by the page allocator (`vm/vm_phys.c`), which uses the well-known *buddy allocator* algorithm to partition the memory and fulfil allocation requests. A buddy allocator is a power-of-two allocator, meaning that it breaks free memory up into power-of-two-sized blocks of contiguous pages which are subsequently used to serve allocation requests. The *order* of each block is the exponent of that block size. A buddy allocator freelist maintains a list of free blocks for each order. If a block of a requested size is not available, higher order blocks are repeatedly split in half until the requested block size is available. When a block is freed (i.e. returned to its respective list of free blocks), the allocator first inspects the state of its *buddy block* and coalesces the two blocks if the *buddy block* is free. Figure 1 depicts the layout of a single buddy allocator freelist.

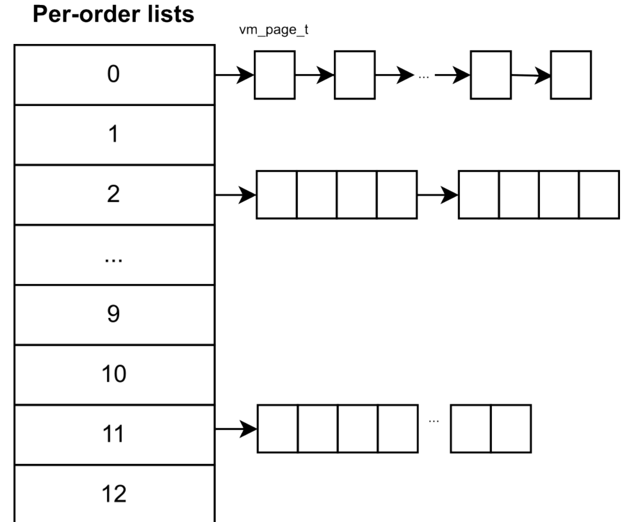

Figure 1: High-level overview of a buddy allocator freelist.

The FreeBSD physical memory allocator maintains queues for orders $[0, 12]$.

B. Superpages

Virtual address translation is a costly operation which can account for up to 10-30% of total system runtime for some workloads [1]. To make matters worse, the constantly increasing memory footprint of modern workloads puts even more pressure on the virtual memory system [2]. Modern computer systems offer a way of decreasing pressure on the virtual address translation caches using *superpages* - i.e. pages of larger size than a standard page. The size of these *superpages* is a multiple of the base page size and ranges from 2MB up to 1GB. Their use brings several important benefits. First off, superpages greatly increase the maximum amount of memory mapped by the

Translation Lookaside Buffer (TLB) (i.e. its *reach*) since they occupy only one TLB entry. For example, modern Intel processors can store a maximum of 1536 entries in their shared TLB [3]. Using 2 MB superpages instead of standard 4K pages increases the TLB's reach from 6 MB to 3072 MB. [4]. Furthermore, they reduce the total number of memory accesses required for the address translation process, as depicted in Figure 2.

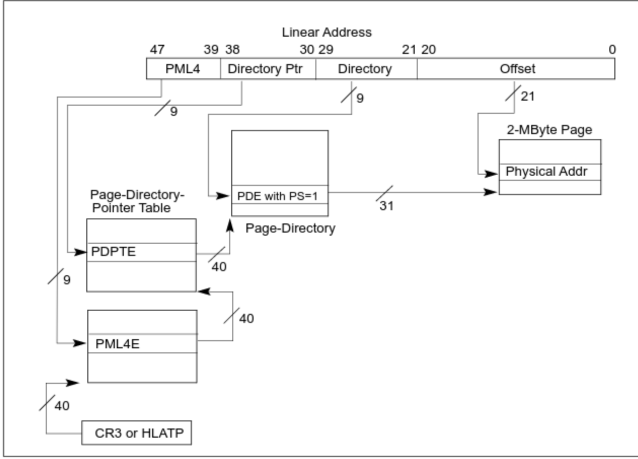

Figure 2: Overview of the address translation process for 2MB pages on Intel CPUs [5].

Superpage mappings require a contiguous and properly aligned physical memory region, which in turn means that the operating system must be able to handle mixed-size page allocations and maintain a steady source of contiguous physical memory regions to maintain the performance benefits provided by superpages. Unfortunately, the addition of another page size class (re)introduced a previously largely irrelevant problem - external memory fragmentation.

C. External fragmentation

External fragmentation occurs when the patterns of allocating and releasing memory leave ”holes” between allocated blocks. Allocating a contiguous chunk of memory in this situation may fail even though there is enough discontiguous memory to satisfy the request.

The previous section briefly mentioned superpages as an example of a feature that is very sensitive to external fragmentation. However, this isn't the only situation where performance suffers in highly fragmented environments. Modern network devices offer performance boosts that require a contiguous physical memory spanning multiple pages. A great example of this are *Ethernet Jumbo frames* - Ethernet frames of sizes between 1500 to 9000 bytes. When in use, jumbo frames offer increased throughput and lower per-packet processing overhead [6]. Leveraging this feature requires a steady supply of contiguous memory chunks to allocate the frames, making it very sensitive to highly fragmented physical memory.

A machine with a long uptime will have a harder time finding spans of contiguous pages to fulfil certain allocation requests. This effect can be easily observed even on freshly booted machines. Table I shows the number of pages in the buddy allocator queues before and after a single `buildkernel` workload on a `bhyve` virtual machine with 6 GB of RAM. The overall state of the physical memory space can be indicated by the number of blocks stored on buddy allocator freelists. For instance, if we observe Table I, we can see that the free physical memory before the `buildkernel` workload was less fragmented since it was mostly comprised of higher-order pages. The situation changes drastically in the second column where most of the free memory is now mostly comprised of 0-order pages. This type of workload is especially effective at fragmenting physical memory, mostly due to the nature of the build workloads where a large number of short-lived processes are spawned in a short period.

Page order	No. pages before	No. pages after
12 (16384K)	337	11
11 (8192K)	1	3
10 (4096K)	2	23
9 (2048K)	1	68
2 (16K)	9	1139
1 (8K)	1	1712
0 (4K)	1	2156

Table I: State of a buddy allocator freelist before and after a `buildkernel` workload. Some orders were left out for the sake of brevity.

III. Design and implementation

This chapter gives a brief overview of each anti-fragmentation mechanism that was developed as a part of this work. The mechanisms listed below can be grouped into two categories: *passive* and *active* anti-fragmentation mechanisms. Passive mechanisms refer to a set of allocation policies and fragmentation-aware usage of page allocations. Their primary goal is to lower the rate of fragmentation as a part of the regular page allocation process. On the other hand, active mechanisms seek to improve free memory contiguity by rearranging individual pages.

A. Memory compaction

Memory compaction is an active anti-fragmentation mechanism that tries to rearrange scattered free pages into a single contiguous block. The `vm_phys` subsystem was extended to support system-wide, per-domain physical memory compaction. It uses a compaction function based on the 'two-finger' mark-compact algorithm [7] to rearrange 0-order pages inside a given physical memory region. Since system-wide compaction is an expensive operation the `vm_phys` subsystem uses a special data structure to quickly identify heavily fragmented regions of a memory domain, increasing the overall efficiency of the compaction.

1) Quantifying fragmentation: The compaction subsystem implements a well-known metric for tracking external fragmentation - the *'Free Memory Fragmentation Index* (FMFI)' [8]. The 'FMFI' metric measures the degree of physical memory fragmentation for a given order by using metadata from the buddy allocator freelists. Figure 3 shows the formula for calculating FMFI, where o is the page order to be allocated, $NoPagesFree$ is the total number of free 0-order pages in the system, and $BlocksFree$ is the total number of contiguous pages stored on the buddy allocator freelists.

$$F_i(o) = 1 - \frac{NoPagesFree/2^o}{BlocksFree}$$

Figure 3: The Free Memory Fragmentation Index.

Its values range from arbitrary negative values up to 1000. A negative value implies that there is ample memory to serve an allocation request of the given order. A value between 0 (no fragmentation) and 1000 (highly fragmented) indicates the degree of physical memory fragmentation. The value of the FMFI metric for each memory domain can be retrieved using the `vm.phys_frag_idx` sysctl.

2) Metadata structures: System-wide memory compaction is an expensive process since it involves scanning large regions of memory and lots of copying. Compaction might not even be effective (or even necessary) in some memory regions, so blindly running a compaction algorithm will result in a lot of wasted CPU time. To improve the accuracy and efficiency of the whole process, the compaction subsystem maintains a set of metadata about the state of individual memory regions. It does so by tracking fixed-size, aligned chunks of physical memory. The core idea behind these data structures is to "divide" the physical memory space into power-of-two-sized chunks and track various metrics for each of these chunks. Listing 1 shows the data structures used by the compaction subsystem.

```
struct vm_phys_search_chunk {
  int holecnt; /* no. available 0-order pages */
  int score;   /* chunk compaction 'score' */
  int skipidx; /* index of next valid chunk. */
  struct vm_phys_subseg_head *shp; /* list of valid
     regions inside this chunk */
};

struct vm_phys_search_index {
  struct vm_phys_search_chunk *chunks;
  int nchunks;
  vm_paddr_t domain_start;
  vm_paddr_t domain_end;
};
```

Listing 1: Compaction metadata.

Page allocation is a very critical operation used by all parts of the system, which is why the guiding principle for this design was to minimize the overhead related to compaction metadata management in the buddy allocator. The search index currently tracks the amount of available memory and the number of free 0-order pages in each chunk. This information is updated each time a page gets added or removed from the buddy allocator freelists. Since the metadata is tracked for power-of-two-sized chunks, finding the corresponding metadata slot is cheap since we only have to mask the page's physical address to index into the metadata array. Since the physical memory space may contain holes, the search index also tracks a list of valid memory regions for each chunk to prevent the compaction algorithm from operating on invalid or non-existent regions of physical memory. Figure 4 depicts the way the search index maps to chunks of physical memory. The search index is also used by the `vm_phys_compact_search` function to identify memory regions suitable for compaction.

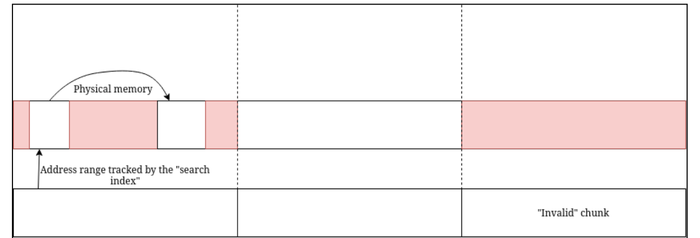

Figure 4: Tracking physical memory metadata in the compaction subsystem.

3) Proactive background compaction: All of the previously described components come together in the form of the *'compaction daemon'*. This daemon is started during boot and spawns a kernel thread for each NUMA domain present in the system. The `vm_phys_compact_thread` function monitors the state of each NUMA domain and periodically performs compaction on its given domain. The compaction daemon also relies on the FMFI metric to reduce its CPU time and to track the impact of each compaction run. The main goal of the compaction daemon is to reduce the value of the FMFI metric for a given order, in this case, the `VM_LEVEL_0_ORDER`. Compaction will not be started if the value of the FMFI metric falls below a certain threshold. This threshold is exposed as a sysctl (`vm.phys_compact_thresh`). Furthermore, if the compaction daemon was unable to relocate any pages or reduce the fragmentation after several runs, it sleeps for a longer period before trying again to avoid wasting CPU time.

B. Kernel stack allocation

The first implemented passive anti-fragmentation mechanism deals with contiguity issues caused by kernel stack allocations. Each kernel stack has at least one "guard" page (i.e. an unmapped page) that acts as a debugging tool

and a defensive mechanism. Assuming that the total size of the kernel stack is N physical pages, the kernel stack ends up using $N+1$ virtual pages in the kernel's virtual address space. On `amd64` this defaults to 4 physical pages and 1 extra guard page. Mapping the stack is done by grabbing N consecutive physical pages from the `kstack vm_object` structure, starting at an index derived from the virtual address (KVA) of the stack. Since stack guard pages aren't backed by physical pages, the index corresponding to the $N+1$ -th physical page is simply skipped, leaving a single unallocated 0-order page for each kernel stack. The new mapping scheme is used to effectively "skip" guard pages and assign pindices for non-guard pages in a contiguous fashion, resulting in fewer wasted 0-order pages. Figure 5 depicts the state of a physical memory region used to back two kernel stacks on the `amd64` architecture with and without the new mapping scheme. Allocating kernel stacks with the new scheme "packs" all physical pages together.

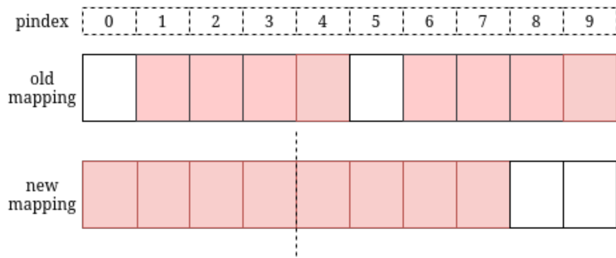

Figure 5: Illustration of the guard page-aware kernel stack allocation.

This scheme requires that all default-sized kstack KVA allocations come from a separate, specially aligned region of the KVA space. For this to work, the work also introduced a new, dedicated kstack KVA arena used to allocate kernel stacks of default size. Aside from fulfilling the requirements imposed by the new scheme, a separate kstack KVA arena has additional performance benefits (keeping guard pages in a single KVA region facilitates superpage promotion in the rest of the KVA space) and security benefits (packing kstacks together results in most kernel stacks having guard pages at both ends).

This mechanism runs the risk of increasing fragmentation in the kernel's virtual address space, which is especially problematic on 32-bit machines. However, this may not be an issue since future releases of FreeBSD are not expected to have support for 32-bit machines [cite].

C. Contiguity-aware allocations

1) Batched page allocations: The main goal of this mechanism was to reduce the rate of fragmentation by improving the contiguity of common page allocation operations in the kernel. Currently, allocating multiple pages boils down to allocating individual 0-order pages in a `for` loop. Before each allocation, the code must also use a "domain iterator" to determine from which NUMA domain the page is going to be allocated from.

This work introduced a new routine for batched allocation of pages, `vm_page_alloc_pages`, along with a new domain iterator (Listing 2) to ensure that the batched allocations are done according to the active NUMA policy. Aside from reducing the rate of fragmentation, this routine offers a performance boost when allocating pages since it will attempt to allocate higher-order pages instead of one 0-order page at a time.

```
static void
vm_domainset_batch_iter_npages_first(struct
    vm_domainset_batch_iter *dbi, struct vm_object*
    obj,  vm_pindex_t pindex, int *npages)
{
  struct vm_domainset_iter *di = &dbi->di;

  switch (di->di_policy) {
  case DOMAINSET_POLICY_FIRSTTOUCH:
    /* FALLTHROUGH */
  case DOMAINSET_POLICY_PREFER:
    *npages = dbi->dbi_npages;
      break;
  case DOMAINSET_POLICY_ROUNDROBIN:
    *npages = dbi->dbi_npages / vm_ndomains;
    break;
  case DOMAINSET_POLICY_INTERLEAVE:
    vm_domainset_batch_iter_npages_interleave(dbi,
    obj, pindex, npages);
    break;
  default:
    panic("%s: Unknown policy %d", __func__,
        di->di_policy);
  }
}
```

Listing 2: Batch page domain iterator

2) Speeding up `mlock`: This mechanism is another example of contiguity-aware allocation. The primary motivation behind this change was to speed up wiring large amounts of memory. This is especially important for hypervisors that wire guest memory to avoid the performance penalty from on-demand paging.

The `vm_map_wire` routine currently uses `vm_fault` to allocate and map one 0-order page at a time. This work attempts to speed up that process by preallocating and inserting higher order pages into an entry's object in order to avoid the excessive overhead of `vm_fault`'s slow path. Aside from speeding up the whole process, allocating in a contiguity-aware manner also serves as a passive anti-fragmentation mechanism.

IV. Experimental setup

Prototype implementations of all previously listed mechanisms were evaluated on a system featuring an AMD Ryzen 5 5600X CPU and 48 GB of DDR4 RAM.

The memory compaction mechanism was evaluated by repeatedly building the FreeBSD kernel. The directories containing the source files and the build output were mounted as `tmpfs` filesystems, mimicking a real-life build server. Each measurement involved running the build

process 10 times and tracking the number promotions and superpage mapping in the system.

Evaluation of passive anti-fragmentation mechanisms was focused on smaller microbenchmarks. Batched page allocation was evaluated by measuring the amount of time it takes to allocate **N** pages at a time, repeating the measurements 30 times for each batch size. The `mlock` changes were evaluated by measuring the amount time spent in the `mlock` system call when booting a `bhyve` virtual machine with 10GB of (wired) memory. The timing was repeated 10 times for the patched and the baseline kernel.

V. Results

Table II lists the `mlock` benchmark results. The results show a significantly shorter duration of the `mlock` system call when compared to the baseline kernel, 9.26x faster on average.

	patched	**baseline**
Avg (ms)	92.49	875.02
Median (ms)	79.98	883.77
Stddev	18.56	80.79
Min	76.76	761.68
Max	115.02	992.12

Table II: `mlock` benchmark results.

Figure 7 illustrates the total number of superpage promotions gained when compared to the baseline kernel. The number of additional superpages that were able to be reused by the operating system grows steadily throughout the benchmark.

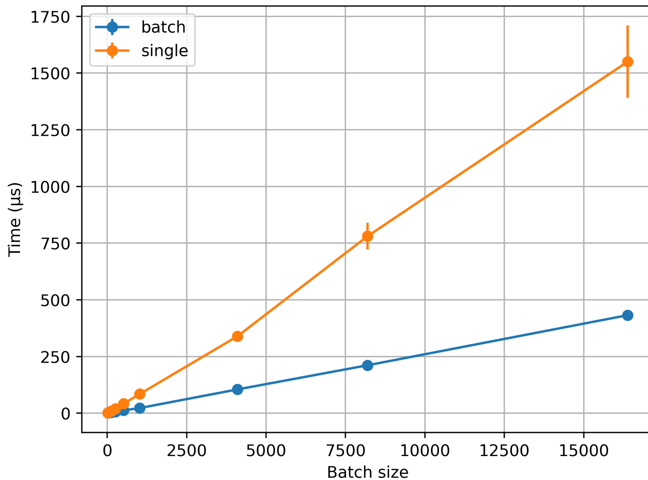

Figure 6: Batched page allocation benchmark results.

Figure 6 represents the average amount of time it took to allocate a number of pages, along with the standard deviation for each data point. The results show a significant speedup when using the proposed batched page allocation approach and more tightly bound execution times due to a smaller standard deviation. This is due to better overall CPU cache usage as all individual subsystems involved in the allocation process get invoked batched.

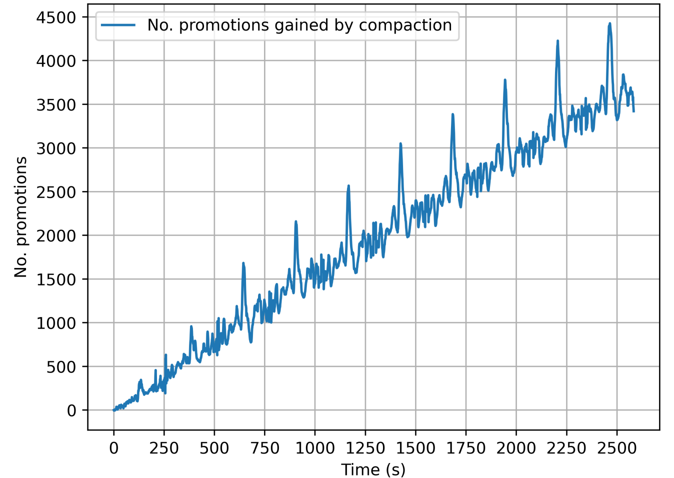

Figure 7: Compaction benchmark results.

VI. Future work

The compaction subsystem currently uses a naive heuristic to select compaction candidates. Future work will focus on designing and implementing more suitable heuristics to increase compaction efficiency.

Another issue that affected the effectiveness of the compaction subsystem was related to the placement of zones for UMA NOFREE objects and wired pages with a long lifetime. Revision D16620 attempted to solve the first issue by segregating UMA NOFREE zones but still hasn't been merged. Furthermore, there is currently no way of signalling the lifetime of wired pages, which leads to poor placement of wired pages used for various caches in the kernel (e.g. buffer cache, ARC).

VII. Conclusion

This paper presented the design and implementation details of several physical memory anti-fragmentation mechanisms for the FreeBSD kernel. Parts of this work are available at D40772 and are under review.

Acknowledgment

Parts of this work have been done as a part of the Google Summer Of Code 2023. program, funded by Google LLC. I'd like to thank Mark Johnston (markj@) for his mentorship and support during this work.

References

[1] S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer, "Rebooting virtual memory with midgard," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 512–525.

[2] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 223–234.

[3] CPU-World, "Cpuid for intel core i5-8265u." [Online]. Available: https://web.archive.org/web/20231121205358/https://www.cpu-world.com/cgi-bin/CPUID.pl?CPUID=66505

[4] W. Zhu, A. L. Cox, and S. Rixner, "A comprehensive analysis of superpage management mechanisms and policies," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 829–842.

[5] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3: system programming guide*, vol. 3, 2023.

[6] P. Prakash, M. Lee, Y. C. Hu, R. R. Kompella *et al.*, "Jumbo frames or not: That is the question!" 2013.

[7] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2023.

[8] M. Gorman and A. Whitcroft, "The what, the why and the where to of anti-fragmentation," in *Ottawa Linux Symposium*, vol. 1. Citeseer, 2006, pp. 369–384.