Coordinated and efficient huge page support(Ingens)-Scribe

- On many workloads, the TLB is not able to capture the working set of the pages and that leads to large latency for memory lookup. Paging is an attractive solution only when the hit rate for the TLB is very high(about 99.99%). If we use large pages (2MB in size) in place of regular 4KB pages, we can increase the TLB coverage and solve this problem.

- In current operating systems, there exists no principled way of huge page allocation. There is a need to manage memory contiguity as a resource and allocate it across processes.

- There is a latency for page faults because the page needs to be allocated from a contiguous memory region(which might require some compaction) and the entire memory region also needs to be wiped(for security reasons) before it is allocated to a process. Tail latency is the value such that the latency encountered by 90% of the operations is lesser than this particular value. Tail latency is an important factor to take into consideration because it guarantees that a large percentage of operations would take lesser time than the tail latency.

- There is a tradeoff between memory savings and performance. When we use only large pages greedily(current Linux strategy), we can get good performance because of fewer TLB misses however this can lead to memory bloat. This tradeoff needs to be taken into consideration while designing a solution for this problem.

- Another incentive for moving to huge pages is that DRAM sizes are constantly increasing leading to an increase in the address space and so it makes sense to use larger pages so that fewer TLB entries are required. In modern 64 bit systems, we use a 4 level page table and this has exacerbated the problem of latency due to TLB misses.

- For a setup where we are running the application on a virtual OS running on a hypervisor, the page fault latency is even more because there can be as many as 24 memory lookups for a single memory dereferencing. This is because in a virtualization setup there are two translations involved. The guest virtual address is translated to the guest physical address using the guest page table. This guest physical address(host virtual address) is in turn translated to the host physical address using the host page table. For a 64 bit machine that there are 4 lookups required for both the host page table and the guest page table. In the worst case this means that there can be a maximum of 24 lookups.

- Modern microprocessor manufacturers have interesting 2 level TLB designs with static translation and dedicated space for large pages. This is mostly just wasted if we dont use huge pages properly.

- Why is the anonymous memory named this way?
    - Anonymous memory is independent of the the user to which it is allocated. Linux currently supports huge pages only for the anonymous memory. Huge pages for page cache pages are not supported.

- In all the experiments performed on Ingens, they use virtualized workloads. This means that every application runs on a guest OS which runs on a host OS which in this case are both Linux. We can see that this setup gives us the best case speedup because it benefits from using huge pages during both page table translations. Also, some experiments are performed on memory intensive workloads which makes the use of huge pages more attractive. For a virtualized setup we can see that there can be latencies as high as 53%, and in this case paging is a failure. Base pages do not suffice in this case and huge pages can help improve performance in this case. If we compare the performance benefits of using huge pages in the host OS and guest OS, then advantages are comparable in both levels. There is no significant advantage in using huge pages at either of these levels in particular.

- The advantages of greedy huge page allocation that is done in Linux are:
    - Since pages allocated are of a larger size, this means that there are fewer page faults when huge pages are greedily allocated.
    - More huge pages implies lesser pressure on TLB leading to better performance.

  However when we are allocating a huge page to a process, the entire 2MB memory region needs to be cleared before being given to a process because of security reasons. For huge page allocation the OS might also have to perform memory compaction before it finds a suitable 2MB block that can be allocated. These factors increase latency of page faults when huge pages are allocated and hence it is not a suitable design choice to synchronously perform memory compaction.

- Fragmentation of memory is measured using the FMFI index:
  The extent of fragmentation depends on the number of free blocks1 in the system, their size and the size of the requested allocation. In this section, we define two metrics that are used to measure the ability of a system to satisfy an allocation and the degree of fragmentation. We measure the fraction of available free memory that can be used to satisfy allocations of a specific size using an unusable free space index, F_u. The fragmentation index is defined as the percentage of free space in memory that is not usable because pages cannot be allocated in that region.

- Asynchronous promotion to reduce page fault latency :
    - Lower huge page promotion speed means under-utilization of the huge page support in hardware and limited performance improvements.
    - High huge page promotion rates increase the CPU cycles required by the promoting thread (memory scanning and compaction), thus increasing CPU usage and affecting user application performance.
    - In a multiprocessor setting, TLB invalidation due to compaction needs to be communicated to the other processors as well in the form of TLB shootdowns which hugely increases overhead.
    - Linux asynchronous promotion speed is chosen to be 1.6 MB/s (i.e. a maximum of 1 huge page per 1.25 seconds).

- Fragmentation variation over time with huge page support enabled vs disabled.
    - With base page support, FMFI quickly falls from 0.3 to almost zero. The reason in that new base pages are allocated by the OS closely in the memory. Additionally, pre-existing pages that fragment memory get unmapped or compacted by the OS periodically.
    - Huge page allocation on the other hand aggravates the FMFI because of less effective compaction and the inability to closely allocate huge pages in an already fragmented memory considering their 512x larger size.
- FreeBSD approach to huge page support -> Extremely conservative and strict. Kills the possible performance improvements out of huge pages in order to keep strict hold on memory utilization. Also supports file cached pages but again is conservative about writing huge pages to avoid I/O traffic.
    - Canneal : An application that allocates memory gradually, the conservative huge page allocation clearly falls back in performance in comparison to the greedy huge page allocation by Linux.

- Fairness : Some issues like fair huge page allocation across processes and especially VMs is important to maintain consistent performance across them. Linux simply has no support or monitoring for fair distribution of huge pages across VMs/processes. In a situation where someone purchases VM from a service provider, one expects an equivalent / comparable performance for VM instances of the same price. This is not guaranteed with the Linux huge page support.

DESIGN

- Tracking the util bitvector(512 bits) for a huge page aligned region with each bit storing utilization of a base page in the aligned region.

- Proportional Promotion
  S # of huge pages f : % huge pages that are idle

P1	10000	100	40%
P2	10000	100	10%
P3	10000	10	10%

Here, all the three processes have the same priority for huge pages (S) and process P3 has the lowest number of allocated huge pages. Intuitively, P3 must be prioritized over P1 and P2 for new huge page allocation.
P1 and P2 have the same priority and same number of allocated huge pages, P1 however has far more huge pages that are actually idle. P1 therefore should be penalized and not be allocated as many huge pages as P2.

- NUMA considerations with huge pages : More likely to create load imbalance across NUMA domains.
    - Page level false sharing : Unrelated data addresses that could be present in different base pages, when present in a single huge page can lead to false sharing.
    - Hot page Effect : If the memory region is frequently accessed (hot) across different cores, it aggravates the load imbalance especially if the hot page is a huge page. It makes sense to split such huge pages in situations where there is false sharing or the huge page is too hot.

- Limitations
    - The Ingens approach pivots from the Linux huge page management at some fundamental level, but it still not completely principled. Some decisions and parameters seem arbitrary unless backed up by tests on workloads.
    - Ingens opts to fall back to Linux huge page management in circumstances when it starts to fail on certain applications.
    - Scan-kth scans for 2 s but periodically repeats only after every 10 s. For the 8 s period in between no access information is captured. -> Not very principled.
    - Scan-kth has a lot of CPU overhead for access bit tracking which can be as high as 11% for 10 GB memory. This memory is not very large, which makes its performance with larger memory worrisome.