Transparent Hugepage Support

Objective

Performance critical computing applications dealing with large memory working sets are already running on top of libhugetlbfs and in turn hugetlbfs. Transparent HugePage Support (THP) is an alternative mean of using huge pages for the backing of virtual memory with huge pages that supports the automatic promotion and demotion of page sizes and without the shortcomings of hugetlbfs.

Currently THP only works for anonymous memory mappings and tmpfs/shmem. But in the future it can expand to other filesystems.

Note

in the examples below we presume that the basic page size is 4K and the huge page size is 2M, although the actual numbers may vary depending on the CPU architecture.

The reason applications are running faster is because of two factors. The first factor is almost completely irrelevant and it's not of significant interest because it'll also have the downside of requiring larger clear-page copy-page in page faults which is a potentially negative effect. The first factor consists in taking a single page fault for each 2M virtual region touched by userland (so reducing the enter/exit kernel frequency by a 512 times factor). This only matters the first time the memory is accessed for the lifetime of a memory mapping. The second long lasting and much more important factor will affect all subsequent accesses to the memory for the whole runtime of the application. The second factor consist of two components:

1. the TLB miss will run faster (especially with virtualization using nested pagetables but almost always also on bare metal without virtualization)
2. a single TLB entry will be mapping a much larger amount of virtual memory in turn reducing the number of TLB misses. With virtualization and nested pagetables the TLB can be mapped of larger size only if both KVM and the Linux guest are using hugepages but a significant speedup already happens if only one of the two is using hugepages just because of the fact the TLB miss is going to run faster.

THP can be enabled system wide or restricted to certain tasks or even memory ranges inside task's address space. Unless THP is completely disabled, there is `khugepaged` daemon that scans memory and collapses sequences of basic pages into huge pages.

The THP behaviour is controlled via :ref:`sysfs <thp_sysfs>` interface and using madvise(2) and prctl(2) system calls.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\admin-guide\mm\(linux-master) (Documentation) (admin-guide) (mm) transhuge.rst`, line 55); *backlink*

Unknown interpreted text role "ref".

Transparent Hugepage Support maximizes the usefulness of free memory if compared to the reservation approach of hugetlbfs by allowing all unused memory to be used as cache or other movable (or even unmovable entities). It doesn't require reservation to prevent hugepage allocation failures to be noticeable from userland. It allows paging and all other advanced VM features to be available on the hugepages. It requires no modifications for applications to take advantage of it.

Applications however can be further optimized to take advantage of this feature, like for example they've been optimized before to avoid a flood of mmap system calls for every malloc(4k). Optimizing userland is by far not mandatory and khugepaged already can take care of long lived page allocations even for hugepage unaware applications that deals with large amounts of memory.

In certain cases when hugepages are enabled system wide, application may end up allocating more memory resources. An application may mmap a large region but only touch 1 byte of it, in that case a 2M page might be allocated instead of a 4k page for no good. This is why it's possible to disable hugepages system-wide and to only have them inside MADV_HUGEPAGE madvise regions.

Embedded systems should enable hugepages only inside madvise regions to eliminate any risk of wasting any precious byte of memory and to only run faster.

Applications that gets a lot of benefit from hugepages and that don't risk to lose memory by using hugepages, should use madvise(MADV_HUGEPAGE) on their critical mmapped regions.

sysfs

Global THP controls

Transparent Hugepage Support for anonymous memory can be entirely disabled (mostly for debugging purposes) or only enabled inside MADV_HUGEPAGE regions (to avoid the risk of consuming more memory resources) or enabled system wide. This can be achieved with one of:

```
echo always >/sys/kernel/mm/transparent_hugepage/enabled
```

```
    echo madvise >/sys/kernel/mm/transparent_hugepage/enabled
    echo never >/sys/kernel/mm/transparent_hugepage/enabled
```

It's also possible to limit defrag efforts in the VM to generate anonymous hugepages in case they're not immediately free to madvise regions or to never try to defrag memory and simply fallback to regular pages unless hugepages are immediately available. Clearly if we spend CPU time to defrag memory, we would expect to gain even more by the fact we use hugepages later instead of regular pages. This isn't always guaranteed, but it may be more likely in case the allocation is for a MADV_HUGEPAGE region.

```
    echo always >/sys/kernel/mm/transparent_hugepage/defrag
    echo defer >/sys/kernel/mm/transparent_hugepage/defrag
    echo defer+madvise >/sys/kernel/mm/transparent_hugepage/defrag
    echo madvise >/sys/kernel/mm/transparent_hugepage/defrag
    echo never >/sys/kernel/mm/transparent_hugepage/defrag
```

always
: means that an application requesting THP will stall on allocation failure and directly reclaim pages and compact memory in an effort to allocate a THP immediately. This may be desirable for virtual machines that benefit heavily from THP use and are willing to delay the VM start to utilise them.

defer
: means that an application will wake kswapd in the background to reclaim pages and wake kcompactd to compact memory so that THP is available in the near future. It's the responsibility of khugepaged to then install the THP pages later.

defer+madvise
: will enter direct reclaim and compaction like `always`, but only for regions that have used madvise(MADV_HUGEPAGE); all other regions will wake kswapd in the background to reclaim pages and wake kcompactd to compact memory so that THP is available in the near future.

madvise
: will enter direct reclaim like `always` but only for regions that are have used madvise(MADV_HUGEPAGE). This is the default behaviour.

never
: should be self-explanatory.

By default kernel tries to use huge zero page on read page fault to anonymous mapping. It's possible to disable huge zero page by writing 0 or enable it back by writing 1:

```
    echo 0 >/sys/kernel/mm/transparent_hugepage/use_zero_page
    echo 1 >/sys/kernel/mm/transparent_hugepage/use_zero_page
```

Some userspace (such as a test program, or an optimized memory allocation library) may want to know the size (in bytes) of a transparent hugepage:

```
    cat /sys/kernel/mm/transparent_hugepage/hpage_pmd_size
```

khugepaged will be automatically started when transparent_hugepage/enabled is set to "always" or "madvise, and it'll be automatically shutdown if it's set to "never".

Khugepaged controls

khugepaged runs usually at low frequency so while one may not want to invoke defrag algorithms synchronously during the page faults, it should be worth invoking defrag at least in khugepaged. However it's also possible to disable defrag in khugepaged by writing 0 or enable defrag in khugepaged by writing 1:

```
    echo 0 >/sys/kernel/mm/transparent_hugepage/khugepaged/defrag
    echo 1 >/sys/kernel/mm/transparent_hugepage/khugepaged/defrag
```

You can also control how many pages khugepaged should scan at each pass:

```
    /sys/kernel/mm/transparent_hugepage/khugepaged/pages_to_scan
```

and how many milliseconds to wait in khugepaged between each pass (you can set this to 0 to run khugepaged at 100% utilization of one core):

```
    /sys/kernel/mm/transparent_hugepage/khugepaged/scan_sleep_millisecs
```

and how many milliseconds to wait in khugepaged if there's an hugepage allocation failure to throttle the next allocation attempt:

```
    /sys/kernel/mm/transparent_hugepage/khugepaged/alloc_sleep_millisecs
```

The khugepaged progress can be seen in the number of pages collapsed:

```
    /sys/kernel/mm/transparent_hugepage/khugepaged/pages_collapsed
```

for each pass:

```
    /sys/kernel/mm/transparent_hugepage/khugepaged/full_scans
```

`max_ptes_none` specifies how many extra small pages (that are not already mapped) can be allocated when collapsing a group of small pages into one large page:

```
/sys/kernel/mm/transparent_hugepage/khugepaged/max_ptes_none
```

A higher value leads to use additional memory for programs. A lower value leads to gain less thp performance. Value of max_ptes_none can waste cpu time very little, you can ignore it.

`max_ptes_swap` specifies how many pages can be brought in from swap when collapsing a group of pages into a transparent huge page:

```
/sys/kernel/mm/transparent_hugepage/khugepaged/max_ptes_swap
```

A higher value can cause excessive swap IO and waste memory. A lower value can prevent THPs from being collapsed, resulting fewer pages being collapsed into THPs, and lower memory access performance.

`max_ptes_shared` specifies how many pages can be shared across multiple processes. Exceeding the number would block the collapse:

```
/sys/kernel/mm/transparent_hugepage/khugepaged/max_ptes_shared
```

A higher value may increase memory footprint for some workloads.

Boot parameter

You can change the sysfs boot time defaults of Transparent Hugepage Support by passing the parameter `transparent_hugepage=always` or `transparent_hugepage=madvise` or `transparent_hugepage=never` to the kernel command line.

Hugepages in tmpfs/shmem

You can control hugepage allocation policy in tmpfs with mount option `huge=`. It can have following values:

always
: Attempt to allocate huge pages every time we need a new page;

never
: Do not allocate huge pages;

within_size
: Only allocate huge page if it will be fully within i_size. Also respect fadvise()/madvise() hints;

advise
: Only allocate huge pages if requested with fadvise()/madvise();

The default policy is `never`.

`mount -o remount,huge= /mountpoint` works fine after mount: remounting `huge=never` will not attempt to break up huge pages at all, just stop more from being allocated.

There's also sysfs knob to control hugepage allocation policy for internal shmem mount: /sys/kernel/mm/transparent_hugepage/shmem_enabled. The mount is used for SysV SHM, memfds, shared anonymous mmaps (of /dev/zero or MAP_ANONYMOUS), GPU drivers' DRM objects, Ashmem.

In addition to policies listed above, shmem_enabled allows two further values:

deny
: For use in emergencies, to force the huge option off from all mounts;

force
: Force the huge option on for all - very useful for testing;

Need of application restart

The transparent_hugepage/enabled values and tmpfs mount option only affect future behavior. So to make them effective you need to restart any application that could have been using hugepages. This also applies to the regions registered in khugepaged.

Monitoring usage

The number of anonymous transparent huge pages currently used by the system is available by reading the AnonHugePages field in `/proc/meminfo`. To identify what applications are using anonymous transparent huge pages, it is necessary to read `/proc/PID/smaps` and count the AnonHugePages fields for each mapping.

The number of file transparent huge pages mapped to userspace is available by reading ShmemPmdMapped and ShmemHugePages fields in `/proc/meminfo`. To identify what applications are mapping file transparent huge pages, it is necessary to read `/proc/PID/smaps` and count the FileHugeMapped fields for each mapping.

Note that reading the smaps file is expensive and reading it frequently will incur overhead.

There are a number of counters in `/proc/vmstat` that may be used to monitor how successfully the system is providing huge pages for use.

thp_fault_alloc
: is incremented every time a huge page is successfully allocated to handle a page fault.

thp_collapse_alloc
: is incremented by khugepaged when it has found a range of pages to collapse into one huge page and has successfully allocated a new huge page to store the data.

thp_fault_fallback
: is incremented if a page fault fails to allocate a huge page and instead falls back to using small pages.

thp_fault_fallback_charge
: is incremented if a page fault fails to charge a huge page and instead falls back to using small pages even though the allocation was successful.

thp_collapse_alloc_failed
: is incremented if khugepaged found a range of pages that should be collapsed into one huge page but failed the allocation.

thp_file_alloc
: is incremented every time a file huge page is successfully allocated.

thp_file_fallback
: is incremented if a file huge page is attempted to be allocated but fails and instead falls back to using small pages.

thp_file_fallback_charge
: is incremented if a file huge page cannot be charged and instead falls back to using small pages even though the allocation was successful.

thp_file_mapped
: is incremented every time a file huge page is mapped into user address space.

thp_split_page
: is incremented every time a huge page is split into base pages. This can happen for a variety of reasons but a common reason is that a huge page is old and is being reclaimed. This action implies splitting all PMD the page mapped with.

thp_split_page_failed
: is incremented if kernel fails to split huge page. This can happen if the page was pinned by somebody.

thp_deferred_split_page
: is incremented when a huge page is put onto split queue. This happens when a huge page is partially unmapped and splitting it would free up some memory. Pages on split queue are going to be split under memory pressure.

thp_split_pmd
: is incremented every time a PMD split into table of PTEs. This can happen, for instance, when application calls mprotect() or munmap() on part of huge page. It doesn't split huge page, only page table entry.

thp_zero_page_alloc
: is incremented every time a huge zero page is successfully allocated. It includes allocations which where dropped due race with other allocation. Note, it doesn't count every map of the huge zero page, only its allocation.

thp_zero_page_alloc_failed
: is incremented if kernel fails to allocate huge zero page and falls back to using small pages.

thp_swpout
: is incremented every time a huge page is swapout in one piece without splitting.

thp_swpout_fallback
: is incremented if a huge page has to be split before swapout. Usually because failed to allocate some continuous swap space for the huge page.

As the system ages, allocating huge pages may be expensive as the system uses memory compaction to copy data around memory to free a huge page for use. There are some counters in `/proc/vmstat` to help monitor this overhead.

compact_stall
: is incremented every time a process stalls to run memory compaction so that a huge page is free for use.

compact_success
: is incremented if the system compacted memory and freed a huge page for use.

compact_fail
: is incremented if the system tries to compact memory but failed.

It is possible to establish how long the stalls were using the function tracer to record how long was spent in __alloc_pages() and using the mm_page_alloc tracepoint to identify which allocations were for huge pages.

Optimizing the applications

To be guaranteed that the kernel will map a 2M page immediately in any memory region, the mmap region has to be hugepage naturally aligned. posix_memalign() can provide that guarantee.

Hugetlbfs

You can use hugetlbfs on a kernel that has transparent hugepage support enabled just fine as always. No difference can be noted in hugetlbfs other than there will be less overall fragmentation. All usual features belonging to hugetlbfs are preserved and unaffected. libhugetlbfs will also work fine as usual.