guest_memfd HugeTLB support, continued

For 2025-11-13 guest_memfd bi-weekly upstream call

Contact ackerleytng@google.com if you have questions/suggestions!

Recap from 2 weeks ago

- guest_memfd with HugeTLB
  - == taking hugepages from HugeTLB, managing them in guest_memfd
- Phased introduction in 3 patch series + 1 prerequisite fix series

Proposal: Phased introduction in [stages]

- [st_blocks] Track allocations in guest_memfd, have guest_memfd update `st_blocks` in `fstat()`
- [HugeTLB support] Add HugeTLB support for either private-only or shared-only use
- [HugeTLB restructuring] Let HugeTLB be used with conversions
  - Restructuring: split directly from `PUD_SIZE` to `PAGE_SIZE` and merge directly back
- [Optimizing restructuring] Restructuring includes `PMD_SIZE`

[st_blocks]	[HugeTLB support]	[HugeTLB restructuring]	[Optimizing restructuring]

[HugeTLB support]

What does this stage give guest_memfd?

- Huge pages from HugeTLB
- Mappings in stage 2 page tables up to 1G level
- Mappings in host page tables up to 1G level
  - To be explored after comments last week
- No conversions within guest_memfd
  - All the above only if `vm_mem_attributes = true`
  - Only legacy dual-backing - shared memory to be provided in memslots' `userspace_addr`
  - Conversions at the VM level (legacy)

[st_blocks]	[HugeTLB support]	[HugeTLB restructuring]	[Optimizing restructuring]

[HugeTLB restructuring]

What does restructuring give us?

Last stage	With restructuring
• Huge pages from HugeTLB • Mappings in stage 2 page tables up to 1G level	
• Mappings in host page tables up to 1G level • No conversions within guest_memfd	• Mappings in host page tables up to 4K level • Conversions within guest_memfd ○ At `PAGE_SIZE` granularity

How is page sharing/conversion supported?

- Sharing of a page => host/rest of the kernel can use the page
- When converting back to private, guest_memfd must ensure that there are no other users
  - By checking refcounts
- Conversion is supported by splitting pages
- Split pages allow us to have per-page refcounts

When do we restructure?

- Allocation time: split if any page ranges are shared
- Conversion time:
  - From private to shared: split pages to `PAGE_SIZE`
  - From shared to private: merge pages to original size if entire range is private
- Truncation time: merge
  - Removal of huge page from guest_memfd ownership

How do we restructure?

- Use `__split_folio()` from kernel
  - Add on undoing and re-application of HugeTLB Vmemmap Optimization (HVO)
- Currently experimenting with keeping folio in filemap during restructuring, using `xa_split_order()`
  - Extend this to split xarrays of higher shifts (for 1G => 4K split)

Conversion flow (TBD)

- On a per-HugeTLB page basis
  - Lock out all (other) allocations, faults, truncations, conversions
  - For shared to private conversions
    - Unmap range from host userspace
    - Fail with `EAGAIN` if there are elevated refcounts
  - Allocate memory for updating shared/private state in maple tree => may fail with `ENOMEM`
  - Restructure folios - splitting needs allocations => may fail with `ENOMEM`
  - — point of no return —
  - Set KVM's invalidation range up to the huge page (expanded from conversion range)
  - Split boundary leaves (TDX)
  - Unmap from stage 2 page tables
  - Commit updates to shared/private state in maple tree
  - Clean up invalidation range

[st_blocks]	[HugeTLB support]	[HugeTLB restructuring]	[Optimizing restructuring]

Truncation quirks (hence custom truncation from [st_blocks])

- Truncation requires merging, merging requires safe refcounts
  - Different reason from conversions
    - Conversion requires safe refcounts because we don't want the host (holding the elevated refcount) to have access to private memory
    - Merging requires safe refcounts because the holder of the refcount is expecting a split page, might cause problems if the page were suddenly larger than expected
- `fallocate(PUNCH_HOLE)` will fail (`EAGAIN`) if there are elevated refcounts
  - Because we can fail `fallocate()`, and it is easier to handle
- But we can't fail truncations due to inode release, hence hook folio_put() to do the merging
  - Merging is complex, hence defer to kernel worker thread

Truncation on inode release

- Inode release: always try to merge in process context
- If some split page is still pinned
  - Mark page as requiring merge (with some page type)
  - Truncate it
  - Let folio_put() merge it
  - => folio outlives inode
- folio_put() will be called on each split page (eventually)
- Track pages yet to be merged in global data structure
  - When pages yet to be merged == original folio nr_pages, do the merge
- Do the merge in a kernel worker thread (deferred)
  - Because `folio_put()` can be called from atomic context

Why track allocated HugeTLB folio metadata?

- Know the original size of the folio, when folio outlives inode
- Fewer fields to save/restore during folio restructuring
  - `__split_folio()` doesn't care about HugeTLB fields on the third `struct page`
  - Track metadata at allocation, restore on free
    - While folio is owned by guest_memfd these fields are static anyway
- Memory failure will use this to answer the question "Does this pfn belong to guest_memfd HugeTLB?" => handle guest_memfd HugeTLB failure separately from other folio types
- When offlining HugeTLB cgroups, `hugetlb_cgroup_css_offline()` will iterate `h->hugepage_activelist` to move HugeTLB charges (nr_pages) to parent
  - Split folio => `folio_nr_pages()` is less than it should be
  - => charge for full number of pages will not be moved
  - => look up this tracking to move charges, update charged cgroup

[st_blocks]	[HugeTLB support]	[HugeTLB restructuring]	[Optimizing restructuring]

How to track allocated HugeTLB folio metadata?

- Multi-index XArray nicely lends itself to tracking folios by order
- Has to be tracked in the kernel itself, outside of KVM, since folio can outlive inode (and KVM)

Code organization

- Have tracking and restructuring code in `mm/hugetlb_restructuring.c`
  - (Name suggestions?)
- Built into the kernel, not in KVM module

[st_blocks]	[HugeTLB support]	[HugeTLB restructuring]	[Optimizing restructuring]

Advertising support

- `vm_memory_attributes` are supported
  - `KVM_CAP_GUEST_MEMFD_FLAGS` will include `GUEST_MEMFD_FLAG_HUGETLB`
- `vm_memory_attributes = false`
  - `KVM_CAP_GUEST_MEMFD_FLAGS` will now include `GUEST_MEMFD_FLAG_HUGETLB`

Issues/Questions

- Should mmap() return a `PUD_SIZE` aligned address for guest_memfd `PUD_SIZE`-d HugeTLB?
- Deferred merge vs in process context: Yan's suggestion to merge if in process context is great, does anyone know how best to check if code is called atomic/process context? [1]

[1] https://lore.kernel.org/all/diqzcy7d60e2.fsf@google.com/

[Optimizing restructuring]

Why optimize restructuring?

- Splitting to `PAGE_SIZE` upon sharing loses too much HVO
  - Based on usage patterns of shared pages, optimizing could save ~160MB per VM
    - => up to 40 VMs => 6GB savings
  - Some more savings from fewer DPAMT entries (TDX)
- Splitting to `PMD_SIZE` is faster than splitting to `PAGE_SIZE`
  - Performance numbers TBD

How to optimize restructuring?

- Upon conversion to shared, keep as many pages at `PMD_SIZE`
  - Split only the ones converted to shared to `PAGE_SIZE`
- Same for merge, merge to as large a page size as possible, based on shared/private status

Complexities

- HVO code today always removes optimization by splitting from `PUD_SIZE` to `PAGE_SIZE` and optimizes back directly
- Can achieve what we want (but inefficient)
  - `PUD_SIZE` => `PAGE_SIZE` => `PMD_SIZE`
- Vishal has done some investigation on direct `PUD_SIZE` to `PMD_SIZE` HVO re-optimization