Google

KVM x86 MMU: Improvements to the TDP Direct Case

Ben Gardon (bgardon@google.com)

Background

- VMs are getting larger
- Live migration needed for host maintenance, software upgrades
- Latency sensitive workloads must migrate under load
- Many vCPUs + memory = hard to migrate

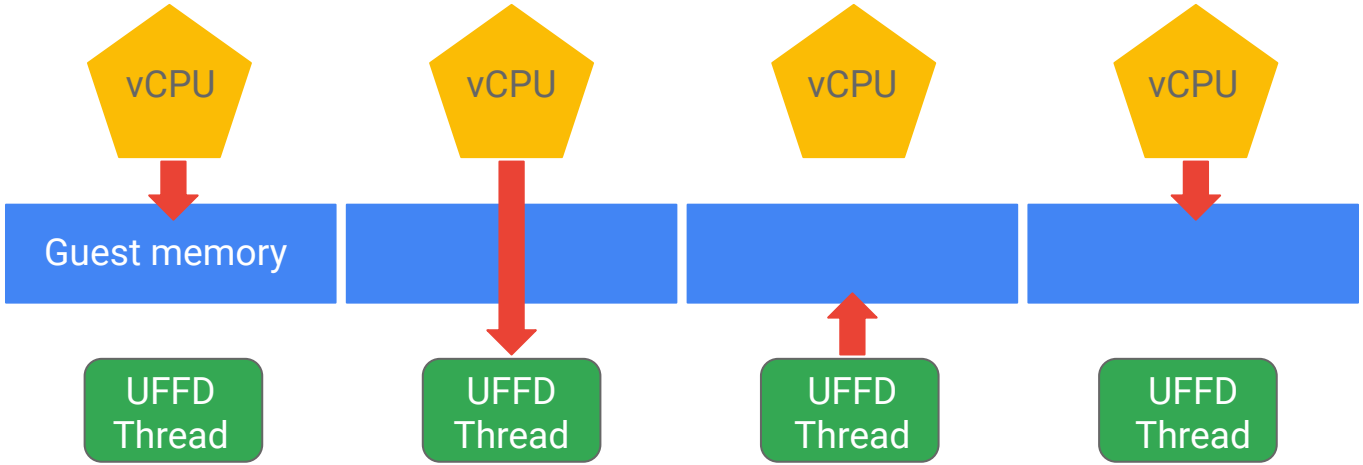
GCE Machine Type	vCPUs	RAM
m1-ultramem-160	160	3.8 TiB
m2-ultramem-416	416	11.5 TiB

Demand paging

- Last stage of live migration
- Memory transfer initiated by vCPU page faults
- User Fault FD
  - Guest page fault -> Host page fault
  - User fault FD thread copies memory
  - Page fault(s) fixed
  - Guest resumes

Sizing up the problem

- KVM selftests `demand_paging_test`
- User fault FD demand paging micro benchmark
- Simulates minimal overheads + perfect userspace tuning



Performance Improvements - Results

```
./demand_paging_test -v 416 -b $(( 4 << 30 )) # 4GiB per vCPU

perf kvm --host --guest record --all-cpus -g -- sleep 1

- 98.72% vmx_handle_exit
   - 98.72% handle_ept_violation
      - 98.72% kvm_mmu_page_fault
         - 98.72% tdp_page_fault
            - 98.11% queued_spin_lock_slowpath
```

Eliminating MMU lock contention will speed demand paging 90%

MMU lock contention

- EPT violations acquire the KVM MMU Lock
- Concurrent page faults cause lock contention
- Contention increases exit latency
- Applications become unresponsive
- Long page faults induce soft lockups
- **Must parallelize page faults**

Why parallel page faults?

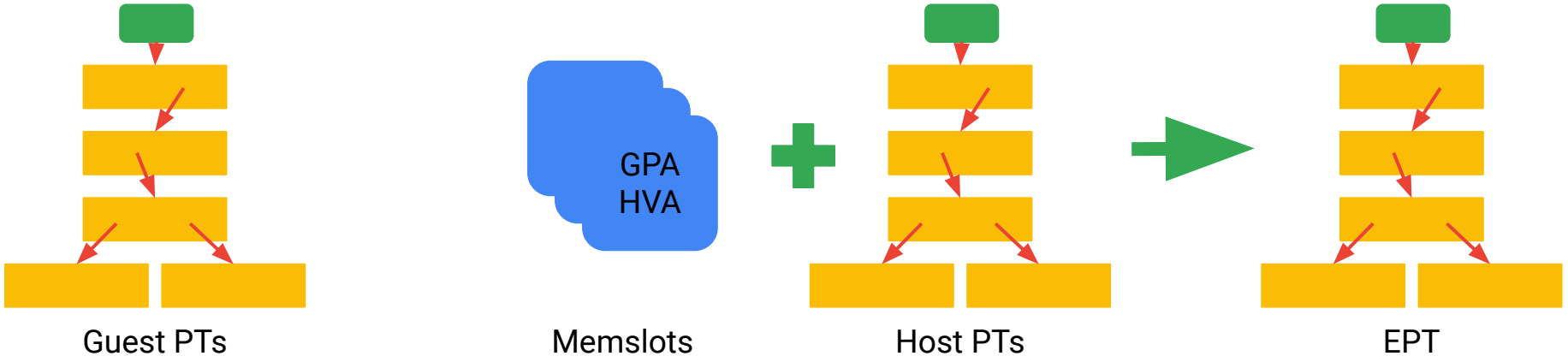
- Not all users of KVM use demand paging
- Other operations cause high PF rate (e.g. disabling dirty logging)
- Lock contention a fundamental MMU scaling issue
- Parallel PF handling -> other parallel operations
- Existing shadow paging difficult to parallelize
- Build a scalable MMU for non-nested TDP
- Data structure problem + hardware concerns

KVM MMU Background

- Data structure problem + hardware concerns
- Translate guest addresses to host addresses
  - Must map GPA -> HPA
- Two major features:
  - Interoperation w/ memory management subsystem
  - Handle vCPU page faults

Guest address translation - TDP direct

- Non-nested guest with Two Dimensional Paging (direct)
- Guest controls CR3 (GVA->GPA), Host controls EPT (GPA->HPA)
- Propagate memslot / host PT changes to EPT



MMU notifiers

Communication channel between **MM subsystem and secondary MMUs (KVM)**

Secondary MMUs program hardware to map physical memory

`invalidate_range_start` **`invalidate_range_end`**	**Free backing physical memory, Must sync w/ PF handler**
`change_pte`	Host PTE changed

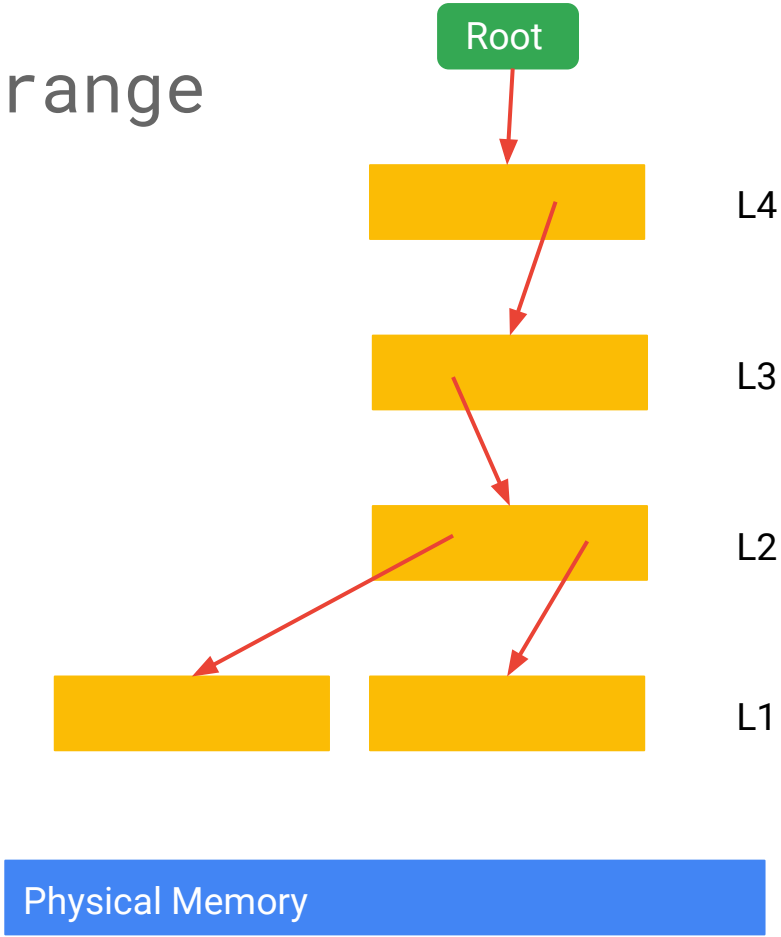
`clear_young`
`clear_flush_young`
`test_young`
	Check if a page has been accessed, optionally clear accessed status / flush TLBS

mmu_notifier_invalidate_range

- Main MM must remap some memory

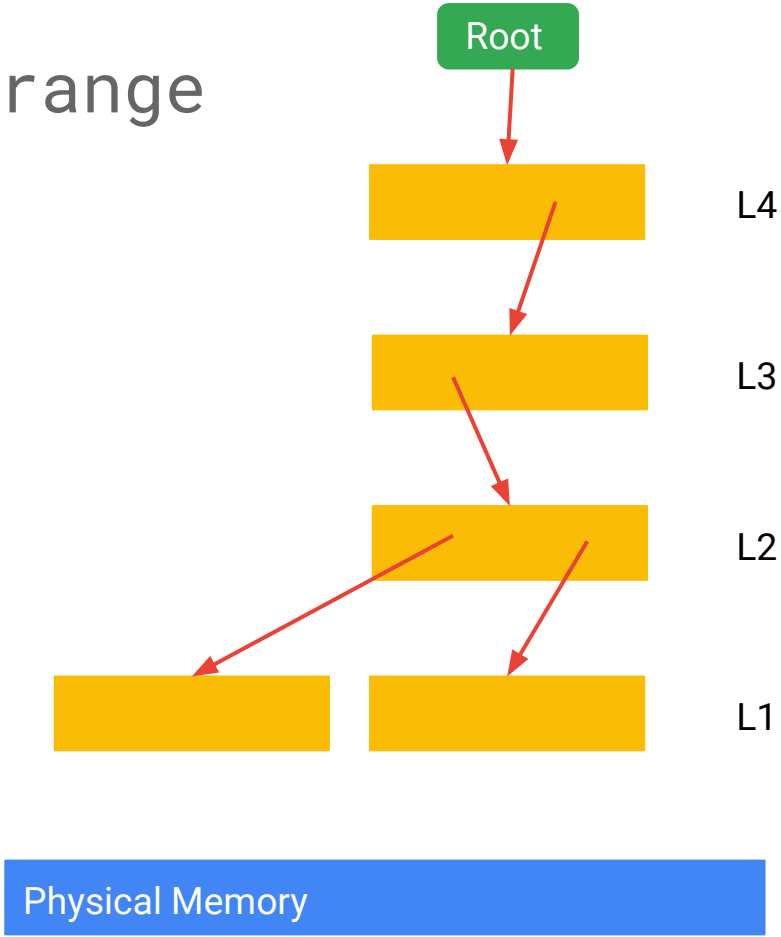
mmu_notifier_invalidate_range

- Main MM must remap some memory
- `invalidate_range_start`
  - Prevent access to a range of memory



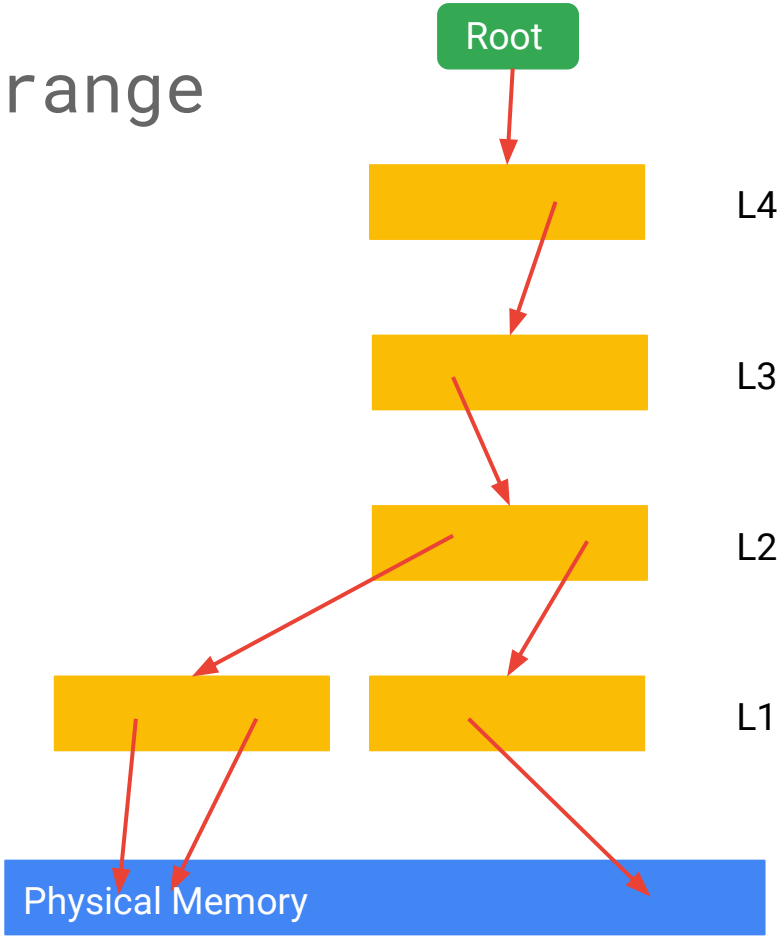
mmu_notifier_invalidate_range

- Main MM must remap some memory
- `invalidate_range_start`
  - Prevent access to a range of memory
- Memory is remapped



mmu_notifier_invalidate_range

- Main MM must remap some memory
- `invalidate_range_start`
  - Prevent access to a range of memory
- Memory is remapped
- `invalidate_range_end`
  - Allow mapping the range
- **Page faults must not run concurrently w/ notifier**



Parallel Page Fault Handling - Requirements

- Concurrently access + safely free page table memory
- Concurrent PTE modifications + track changes
- Interoperate with MMU notifiers
- Prevent vCPU access to remapped memory
- Prevent vCPU address translation through freed memory

Parallel Page Fault Handling - Requirements

- Concurrently access + safely free page table memory
- Concurrent PTE modifications + track changes
- Interoperate with MMU notifiers
- Prevent vCPU access to remapped memory
- Prevent vCPU address translation through freed memory

Walking paging structures

- `direct_walk_iterator`
- Preorder traversal of paging structure
- Common pattern for MMU operations
  - Setup traversal range
  - Modify PTEs
  - Cleanup, sync caches
- Transparently handles synchronization

Why use `direct_walk_iterator`

- TDP direct -> one paging structure
- Direct paging structure traversal parallelizes well
- No reverse map
  - Only allocated for nested virtualization
  - ~8 bytes / 4k guest memory
- No `struct kvm_mmu_page`
  - `direct_walk_iterator` tracks metadata
  - ~170 bytes / page of PTEs
- 0.2% of guest memory saved = 24GiB for 12T VM

Freeing and Accessing Page Table Memory

- Must prevent KVM access to freed memory
- `direct_walk_iterator` holds RCU read lock
- Free page table memory after:
  - Disconnect
  - RCU grace period / callback
- `direct_walk_iterator` transparently handles:
  - Acquiring RCU lock, RCU derefences
  - Restarting traversal after yielding

Parallel Page Fault Handling - Requirements

- Concurrently access + safely free page table memory
- Concurrent PTE modifications + track changes
- Interoperate with MMU notifiers
- Prevent vCPU access to remapped memory
- Prevent vCPU address translation through freed memory

Atomic Operations

- Atomic `cmpxchg` for all PTE modifications
- On success:
  - Bookkeeping handled by `direct_walk_iterator`
  - Based on level, guest address, previous value, new value
- On failure:
  - Other thread makes progress
  - Transparent retry handled by `direct_walk_iterator`

Parallel Page Fault Handling - Requirements

- Concurrently access + safely free page table memory
- Concurrent PTE modifications + track changes
- Interoperate with MMU notifiers
- Prevent bad vCPU cache state
- Prevent vCPU address translation through freed memory

Interoperation - MMU lock

- Need parallel page faults
- Other operations need exclusive access
  - e.g. MMU notifiers
  - e.g. Interop w/ shadow paging for nested
- MMU spinlock -> MMU reader / writer lock

MMU notifiers,
other operations

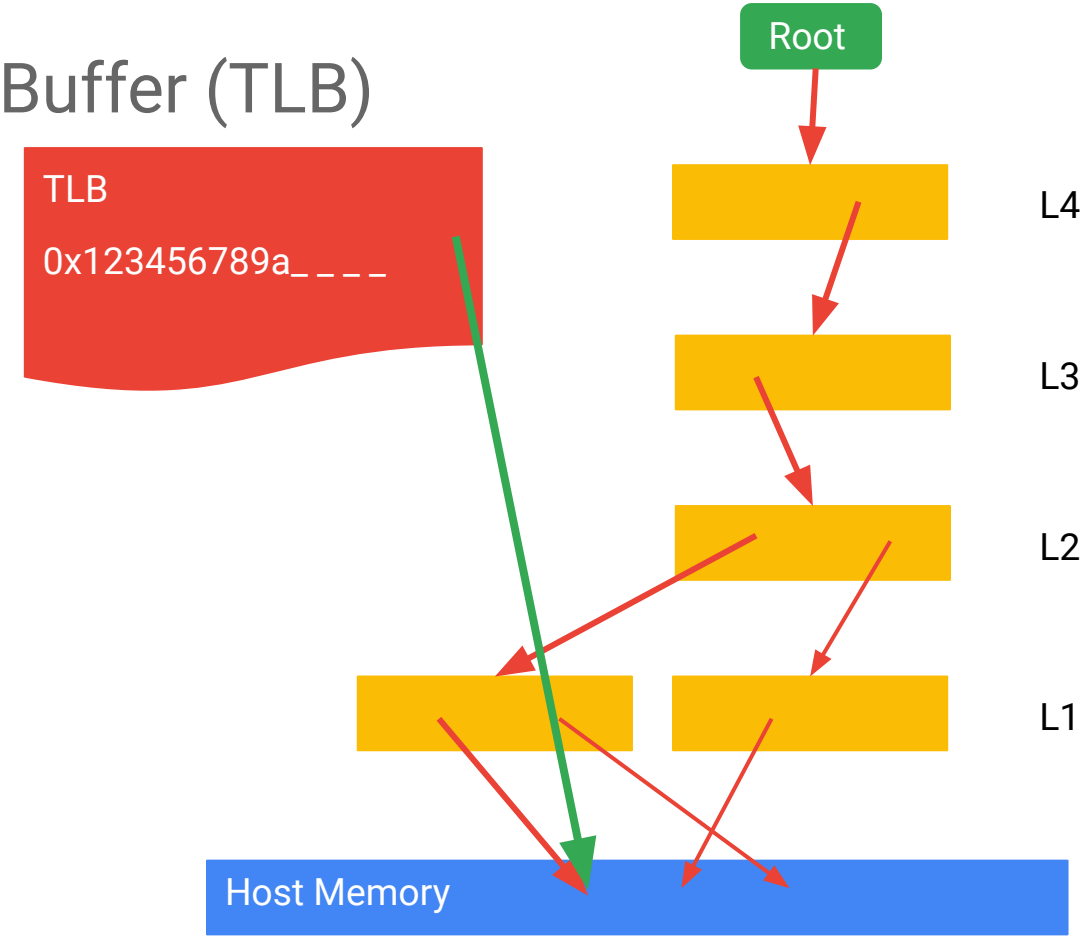
Page Faults

Parallel Page Fault Handling - Requirements

- Concurrently access + safely free page table memory
- Concurrent PTE modifications + track changes
- Interoperate with MMU notifiers
- Prevent vCPU access to remapped memory
- Prevent vCPU address translation through freed memory

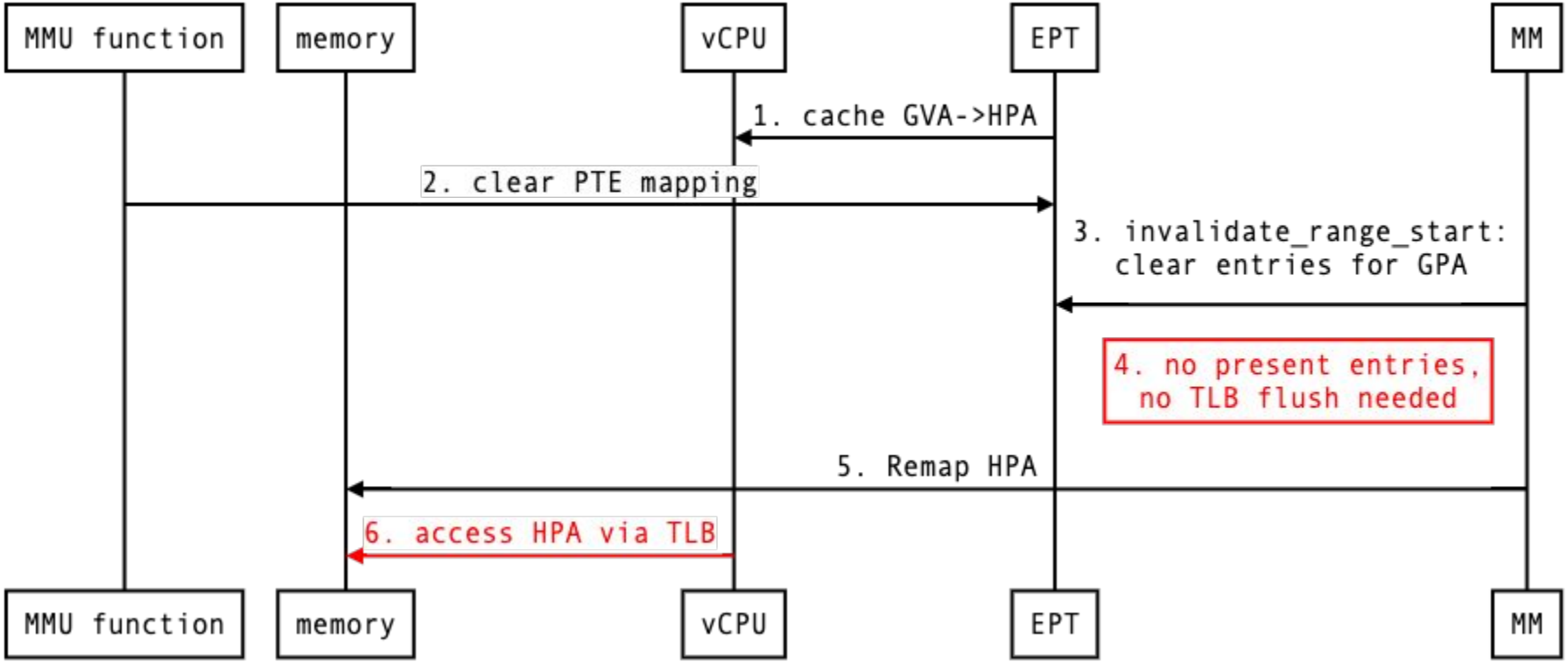
Translation Lookaside Buffer (TLB)

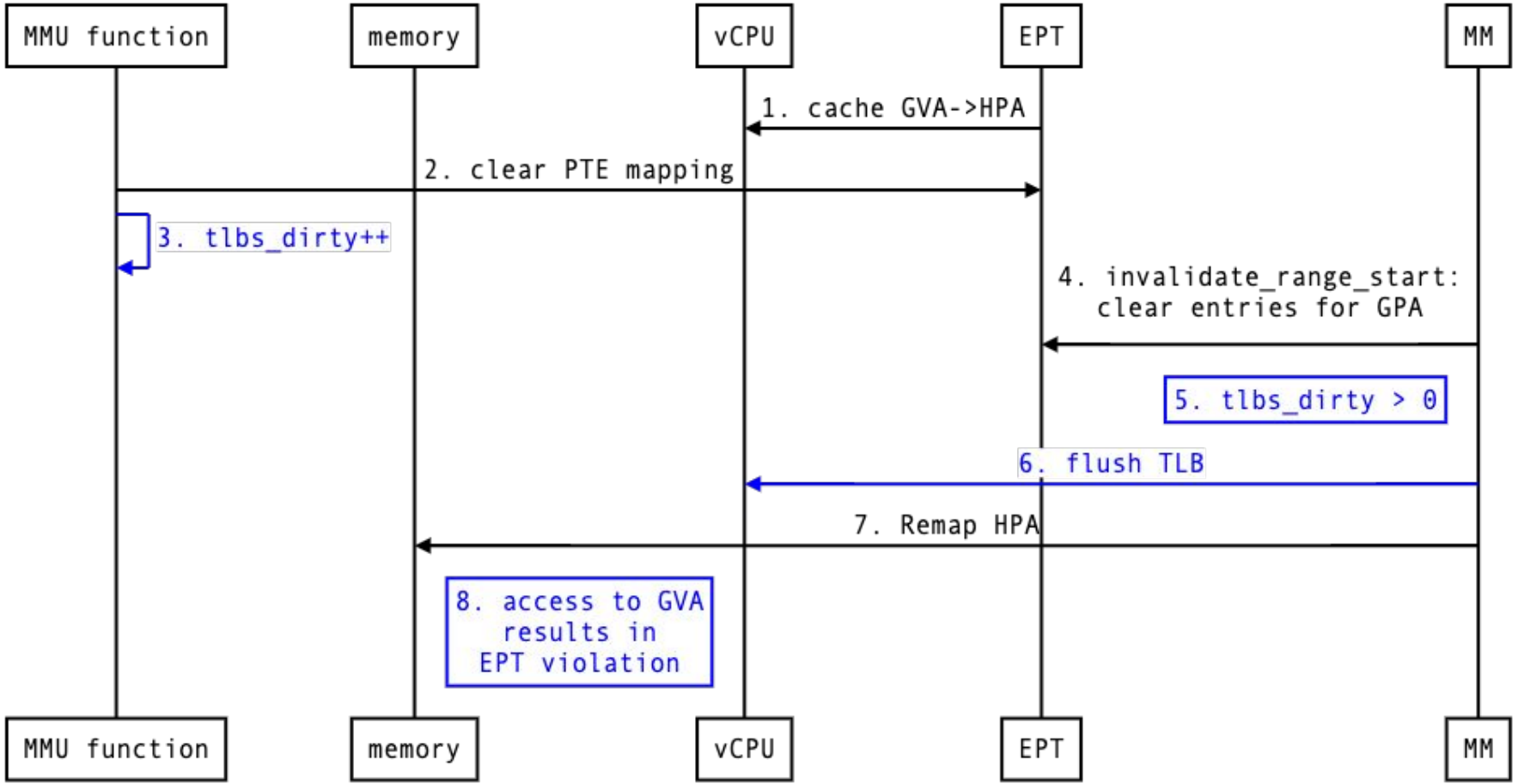
- Cache complete translation
  - e.g. GVA->HPA, HVA->HPA
- TLB Flush
  - `kvm_flush_remote_tlbs`
  - Clears TLBs
  - Synchronizes caches with in-memory state



Preventing improper guest access

- Problem
  - TLB flushes are slow
  - Frequent TLB flushes degrade guest TLB performance
- Solution
  - Flush TLBs after clearing PTEs
- Avoid flushing TLBs in page fault handler and other operations


MMU function
memory
vCPU
EPT
MM
1. cache GVA->HPA
2. clear PTE mapping
3. invalidate_range_start:
clear entries for GPA
4. no present entries,
no TLB flush needed
5. Remap HPA
6. access HPA via TLB
MMU function
memory
vCPU
EPT
MM


MMU function
memory
vCPU
EPT
MM
1. cache GVA->HPA
2. clear PTE mapping
3. tlbs_dirty++
4. invalidate_range_start:
clear entries for GPA
5. tlbs_dirty > 0
6. flush TLB
7. Remap HPA
8. access to GVA
results in
EPT violation
MMU function
memory
vCPU
EPT
MM

Deferring TLB Flushes

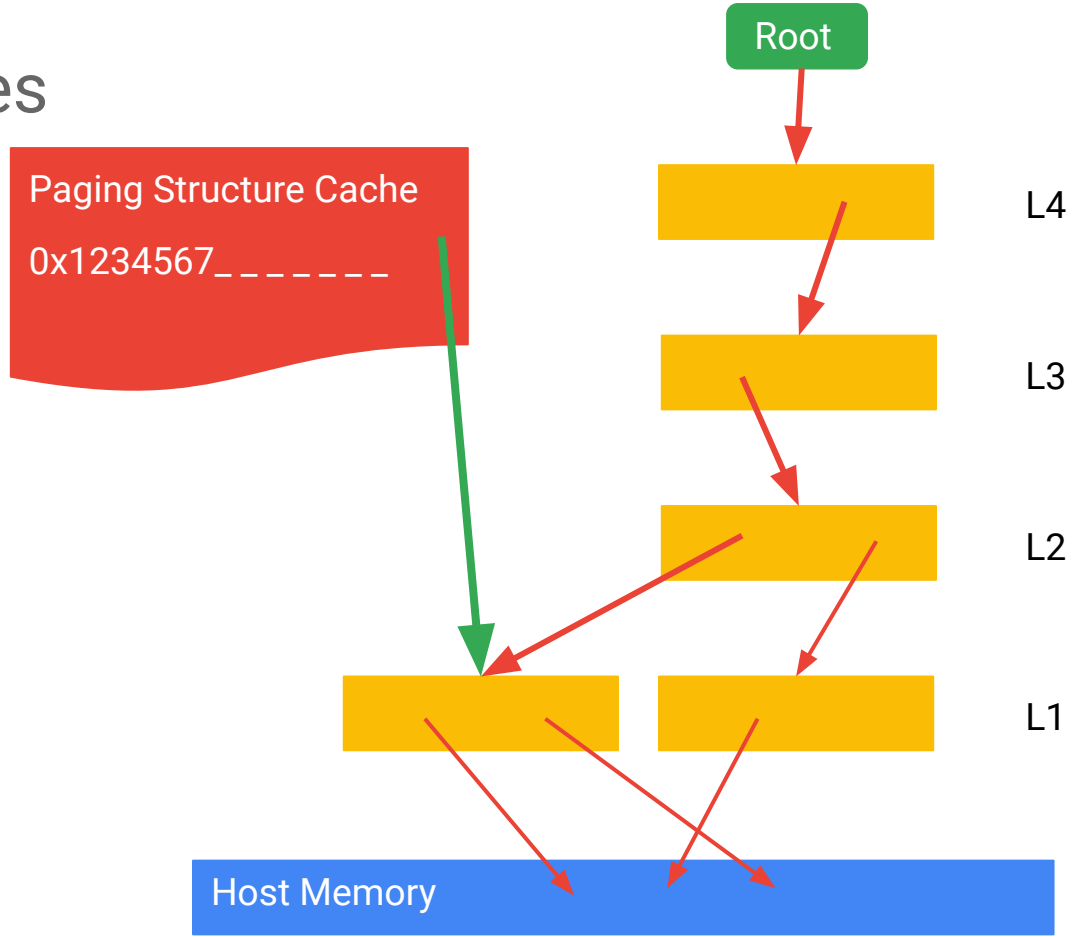
- Increment TLBs dirty counter when changing PTEs
  - Handled by `direct_walk_iterator`
- Skip flush if guest sync not needed
  - E.g. page fault handler
- If sync needed, flush on PTEs or TLBs dirty
  - E.g. MMU notifiers

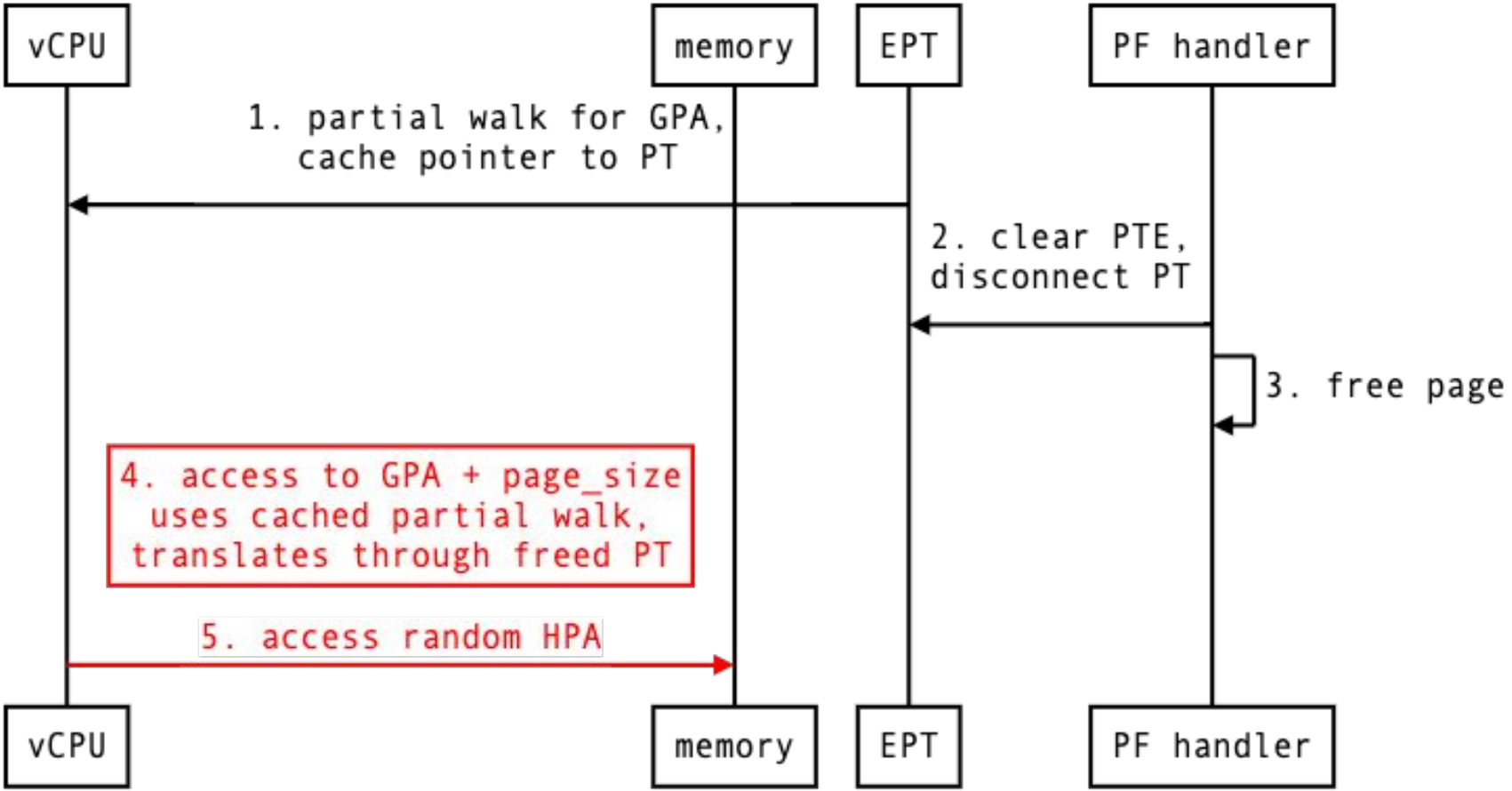
Parallel Page Fault Handling - Requirements

- Concurrently access + safely free page table memory
- Concurrent PTE modifications + track changes
- Interoperate with MMU notifiers
- Prevent vCPU access to remapped memory
- Prevent vCPU address translation through freed memory

Paging Structure Caches

- Cache partial walks
  - e.g. High 27 bits of GPA -> EPT level 1 page
- vCPUs can access disconnected PT memory
- TLB Flush
  - `kvm_flush_remote_tlbs`
  - Clears TLBs
  - Clears paging structure caches
  - Synchronizes caches w/ in-memory state



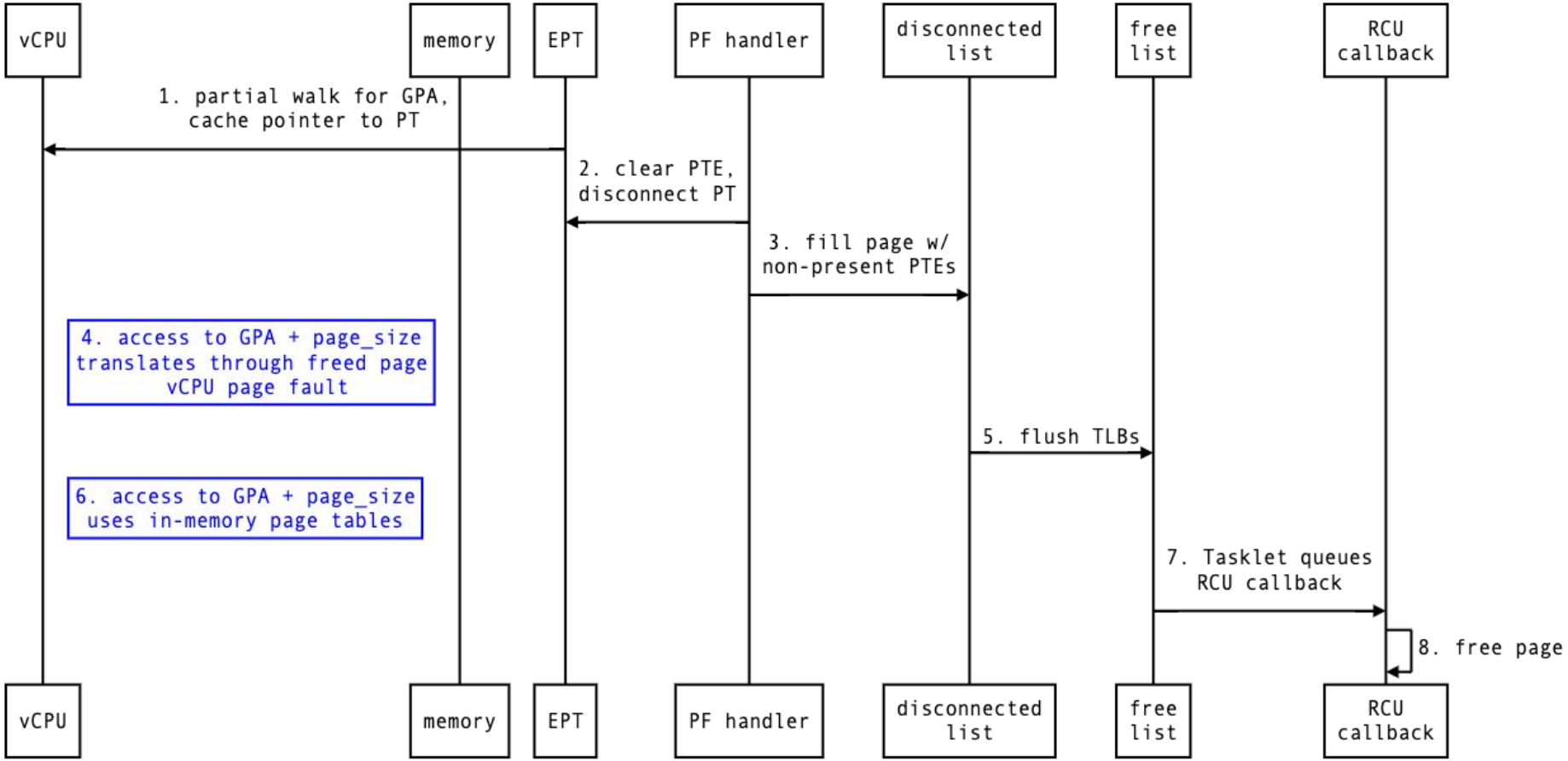

vCPU
memory
EPT
PF handler
1. partial walk for GPA,
cache pointer to PT
2. clear PTE,
disconnect PT
3. free page
4. access to GPA + page_size
uses cached partial walk,
translates through freed PT
5. access random HPA
vCPU
memory
EPT
PF handler

TLB Flushes and Freeing Page Table Memory

Two prerequisites to free page table memory:

RCU grace period

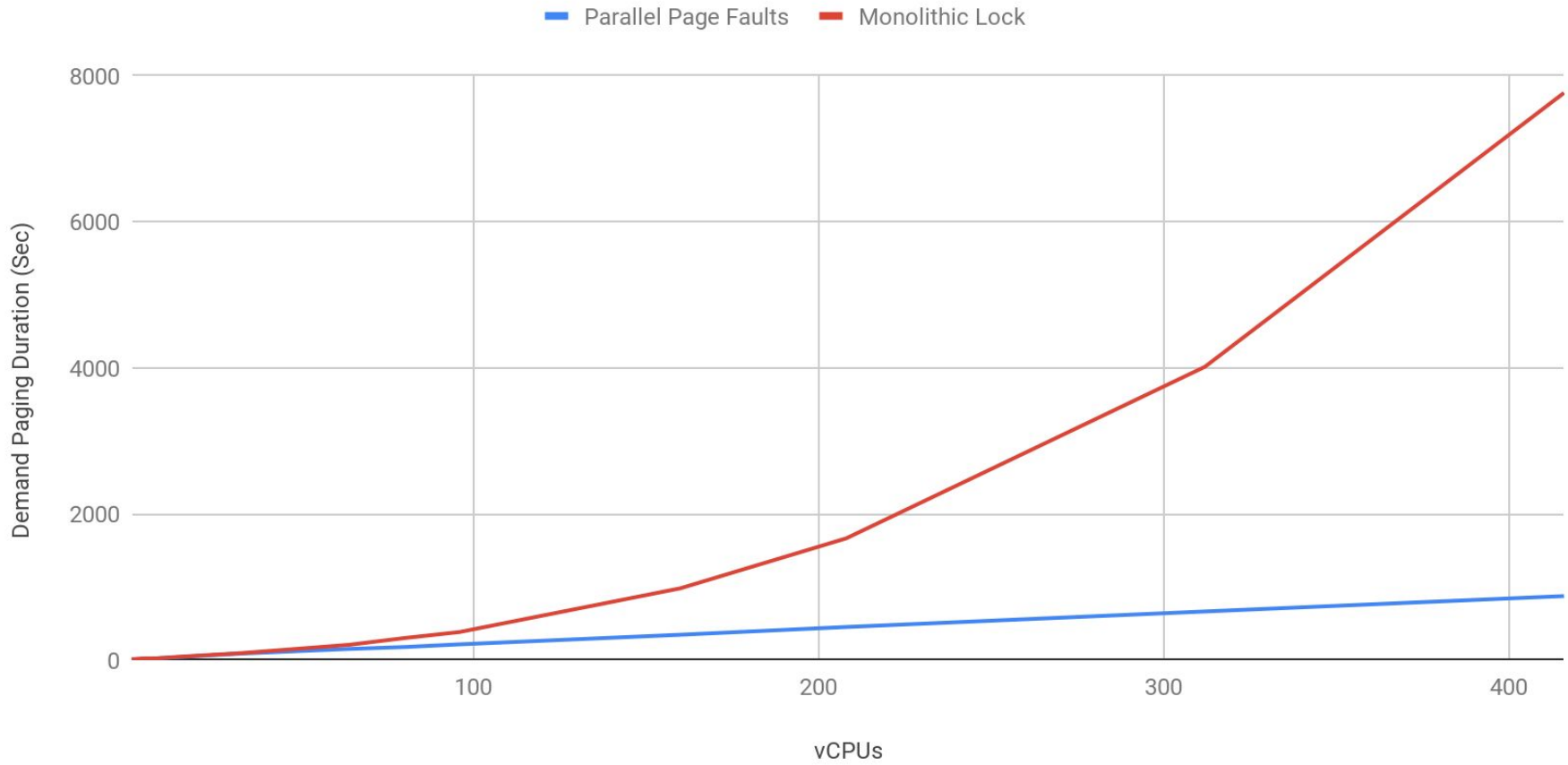
TLB flush


vCPU
memory
EPT
PF handler
disconnected list
free list
RCU callback
1. partial walk for GPA, cache pointer to PT
2. clear PTE, disconnect PT
3. fill page w/ non-present PTEs
4. access to GPA + page_size translates through freed page vCPU page fault
5. flush TLBs
6. access to GPA + page_size uses in-memory page tables
7. Tasklet queues RCU callback
8. free page
vCPU
memory
EPT
PF handler
disconnected list
free list
RCU callback

Improvements

- Simplified access pattern
  - Reduced maintenance burden, harder to forget to free/flush/update
- Memory savings
  - Improves efficiency, reduces overheads
- Parallel page faults
  - Improves live migration performance, reduces guest jitter and soft lockups
- Reduced / deferred TLB flushes
  - Improves MMU performance and guest TLB performance

Demand paging duration with N vCPUs (4GiB/vCPU)



Integration

- These are a lot of changes to the way the MMU works
- The best way to integrate is an open question

- Replacement
  - Extend the iterator pattern for x86 and nested shadow paging
  - Replace the whole KVM MMU
- Modularization
  - Formalize the MMU interface
  - Split the TDP MMU and x86 shadow MMU

Viewing the RFC

KVM mailing list

https://www.spinics.net/lists/kvm/msg196464.html

Gerrit

https://linux-review.googlesource.com/c/virt/kvm/kvm/+/1416

Parallel Page Fault Handling - Where are the costs now?

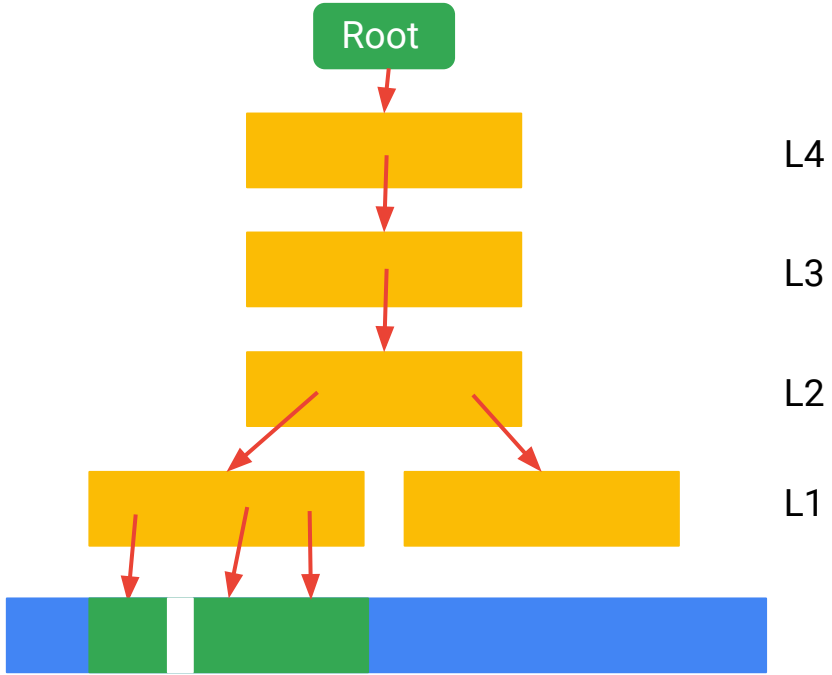
```
- 78.74% kvm_vcpu_ioctl
  - 78.74% kvm_arch_vcpu_ioctl_run
    - 78.72% vmx_handle_exit
      - handle_ept_violation
        - 78.72% kvm_mmu_page_fault
          - 78.72% tdp_page_fault
            - 64.44% try_async_pf
              - 64.44% __gfn_to_pfn_memslot
                - 49.76% get_user_pages_unlocked
                  + 41.43% down_read
                  - 8.32% __get_user_pages
                    - 8.29% handle_mm_fault
                      - 8.29% __handle_mm_fault
                        - 8.29% handle_userfault
                          + 8.27% schedule
                          + 14.67% down_read
            - 14.25% mapping_level
              - 14.25% kvm_host_page_size
                + 14.24% down_read
```

```
- 13.53% userfaultfd_ioctl
  - 13.36% mcopy_atomic
    - 13.33% down_read
      - 13.32% call_rwsem_down_read_failed
        - 13.31% rwsem_down_read_failed
          - 13.31% _raw_spin_lock_irq
              queued_spin_lock_slowpath
```

TL;DR There's still contention, but less, in more places, and it's not in KVM anymore

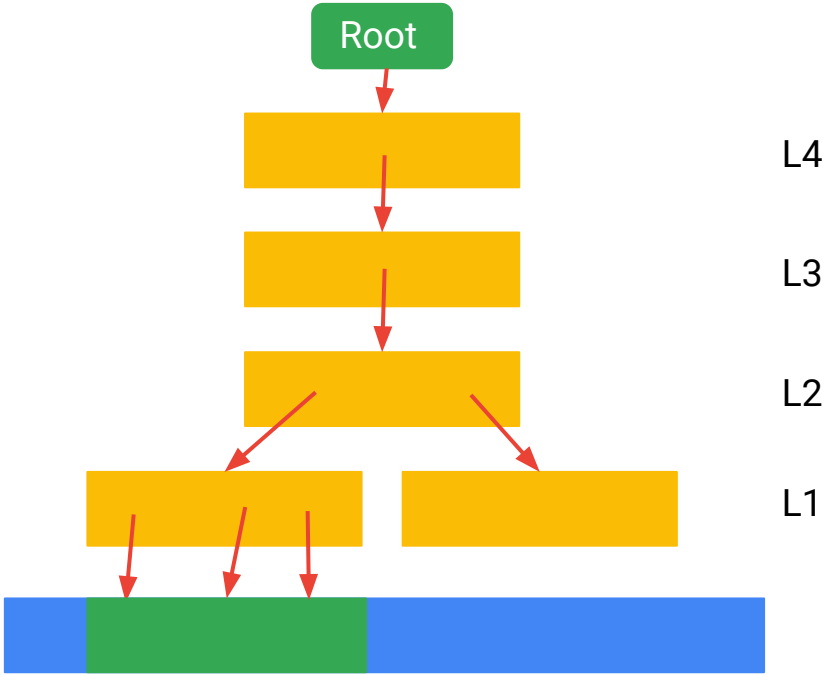
Reducing TLB Flushes - Why are we clearing PTEs?

- Optimization: demand page at 4k, back memory w/ large pages



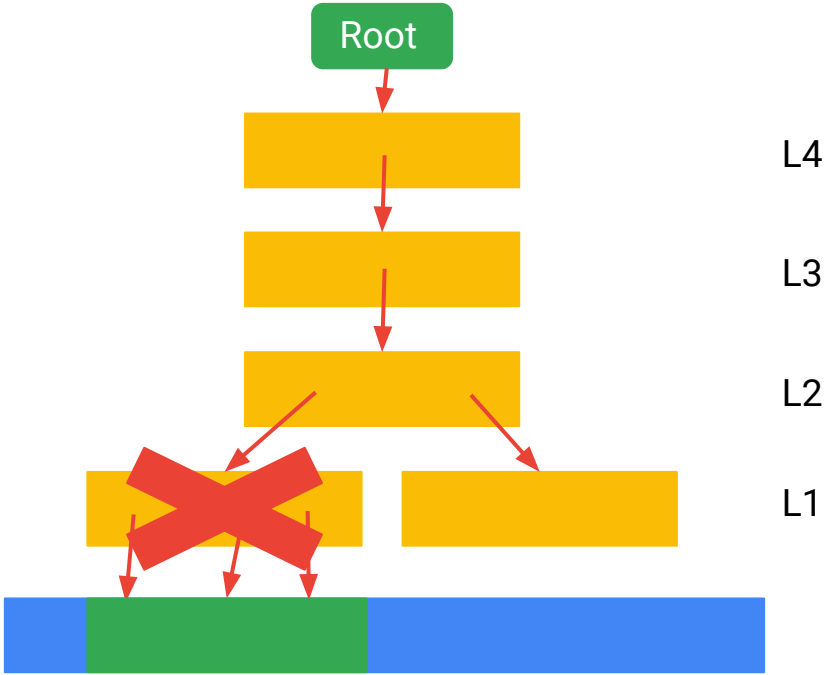
Reducing TLB Flushes - Why are we clearing PTEs?

- Optimization: demand page at 4k, back memory w/ large pages



Reducing TLB Flushes - Why are we clearing PTEs?

- Optimization: demand page at 4k, back memory w/ large pages



Reducing TLB Flushes - Why are we clearing PTEs?

- Optimization: demand page at 4k, back memory w/ large pages