Memory Externalization With userfaultfd

Red Hat, Inc.

Andrea Arcangeli
aarcange at redhat.com

Germany, Düsseldorf

15 Oct 2014
Memory Externalization

- Memory externalization is about running a program with part (or all) of its memory residing on a remote node.
- Memory is transferred from the memory node to the compute node on access.
- Memory can be transferred from the compute node to the memory node if it's not frequently used during memory pressure.
- The Kernel needs new VM (as in Virtual Memory) features to allow this kind of memory externalization.
Postcopy Memory Externalization

- Postcopy live migration is also some form of one-way memory externalization

- The compute node is running the qemu live migration destination

- The memory node is running the qemu live migration source

- If we solve the memory externalization problem in a generic way that can work for all linux applications, it will also allow qemu to implement postcopy live migration
  - Without requiring any KVM/virt specific patch
Initial Postcopy Live Migration

- The initial KVM postcopy live migration prototype from Isaku Yamahata was very inspiring
- Great prototype to demonstrate it, but in production environments its kernel backend would have disabled:
  - Overcommit and swap
  - THP
  - KSM
  - NUMA balancing
  - NUMA hard bindings (mbind/set_mempolicy etc..)
- A special device driver would have required special privileges similar to mlock()
- It could have been hardly adopted by non-virt users
  - i.e. volatile pages on tmpfs
First problem: userfault

- qemu destination running in the compute node must be notified the first time a page fault happens if a page is still missing

  Destination guest virtual memory (kernel side is a vma)

  Unmapped virtual addresses (pages) must trigger userfault on access

- To get the notification through SIGBUS (info->si_addr) we introduced:
  - madvise(MADV_USERFAULT)
  - madvise(MADV_NOUSERFAULT)
SIGBUS userfault not enough

- SIGBUS is ok to trap userland accesses (like \textit{volatile pages})
- SIGBUS generates \textit{failures} when kernel code tries to access the unmapped virtual addresses:
  - `get_user_pages` would return -EFAULT
    - KVM page fault
    - O\_DIRECT I/O
  - syscalls using `copy\_from\_user/copy\_to\_user`
    - `write()`
    - `read()`
    - ...
- In qemu we might handle a special error from the `/dev/kvm` ioctl, but we don't want to handle errors for \textbf{all} syscalls
Userfault ideal behavior

What should happen when an userfault trigger is:

- The page fault of the thread that touched the unmapped page is blocked
- One thread of the application is notified by the kernel about an userfault having triggered at a certain address
- The thread transfers the missing page from the (remote) memory node to the (local) compute node
- The thread maps the missing page at the userfault address atomically
- The thread tells the kernel to wakeup any blocked page fault for a certain virtual address range that was just mapped
- The waken up page fault retries the fault and finds the virtual page mapped
Problem in blocking the page fault

- We want the userfault feature not to require special privilege
- Page faults runs while holding the mmap_sem for reading
- We cannot indefinitely block a page fault while holding a core kernel lock
- The page fault flag “FAULT_FLAG_ALLOW_RETRY” if set allows us to drop the mmap_sem (it was written to drop the mmap_sem before I/O)
- Problem: many get_user_pages users don't set FAULT_FLAG_ALLOW_RETRY when simulating the page fault
- get_user_pages_locked/unlocked fixes get_user_pages users to always use FAULT_FLAG_ALLOW_RETRY
ufd = userfaultfd() - syscall

- The userfaultfd syscall provides userland a protocol to control the userfaults in a way that is transparent to all syscalls and get_user_pages kernel users.
- An userland thread responsible to manage the userfaults can listen to the userfaultfd to know the virtual addresses where any userfault triggered.
- After resolving the userfaults the thread just need to notify the kernel about it, to wakeup any page fault that was blocked.
- There can be an unlimited number of userfaultfd per process:
  - Shared libs can use userfaultfd independently of each other and the main program.
  - Each userfaultfd must register its own userfault range.
    - MADV_USERFAULT must be set as well.
How to resolve an userfault

- We must fill the unmapped virtual address
- The unmapped virtual address must be filled *atomically*
- We cannot remove MADV_USERFAULT if other threads could access the unmapped address while we map the virtual address
- A new syscall can fill unmapped virtual pages atomically
  - `remap_anon_pages(userfault_addr, tmp_addr, PAGE_SIZE)`
- `remap_anon_pages` allows also to atomically “remove” a mapped page from the userfault virtual range, to turn it into a unmapped hole
  - It works both ways
remap_anon_pages

tmp_addr

Guest physical address space

2 1 3 4 5 6

Guest physical address space

2 3 4 5 6
userfaultfd + remap_anon_pages

Kernel

page fault calls handle_userfault()

raise POLLIN & wakeup the read(ufd,...)

handle_userfault() waken up and returns

Retry the fault at the userfault_addr

Userland thread

userfault_addr = read(ufd) & PAGE_MASK

Transfer page to tmp_addr page aligned

remap_anon_pages(userfault_addr, tmp_addr, PAGE_SIZE)

write(ufd, [userfault_addr, userfault_addr+PAGE_SIZE], 16)

Wait in read(ufd) or for POLLIN from ufd
Atomic mcopy

• A new syscall could also be exposed to userland to fill unmapped holes in anonymous or tmpfs regions atomically
  - mcopy_atomic(userfault_addr, tmp_addr, PAGE_SIZE*N)

• Pros:
  - Likely more efficient because it doesn't require TLB flushes
  - No src_addr, dst_addr page alignment constraints
  - It would work more easily for tmpfs backed userfaults, regardless of the type of memory at the source address
  - Simpler and self contained

• Cons
  - Unable to remove pages from the userfault virtual range
    • remap_anon_pages could still be used for that
mcopy_pages

tmp_addr

Guest physical address space

Copy Of 1

Guest physical address space
userfault and KVM

- Thanks to the KVM design (as usual)
  - No change to KVM kernel driver was required
  - All changes are in the core Linux Virtual Memory
  - THP/KSM/NUMA balancing/NUMA bindings are transparently supported on the userfault memory ranges

- Only the qemu balloon driver will need special handling during postcopy live migration as MADV_DONTNEED would create unmapped regions in the userfault area
  - If the guest touches ballooned pages inflated during postcopy live migration, the migration thread should not get confused about it
  - The userfault feature could also be used to enforce that the guest cannot deflate the balloon
userfault and volatile pages

- Volatile pages are virtual memory ranges that the kernel can discard under memory pressure without swapping them out.
- The volatile pages patchset contemplated optionally to provide the *userfault-like* SIGBUS behavior on access.
- The userfault in addition to solving postcopy live migration and the memory externalization feature, can provide the SIGBUS notification to applications using volatile pages after their eviction by setting `MADV_USERFAULT` on the volatile page ranges.
  - In addition volatile pages could also use the `userfaultfd` protocol to allow the kernel to access the volatile pages.
  - Without `userfaultfd` only userland access is allowed to avoid getting unreliable errors from syscalls or `get_user_pages`.
Userfault kernel patchset

- Last submit against 3.17-rc:
  - [http://thread.gmane.org/gmane.linux.kernel.mm/123575](http://thread.gmane.org/gmane.linux.kernel.mm/123575)
  - `git clone git://git.kernel.org/pub/scm/linux/kernel/git/andrea/aa.git -b userfault`

- Implements:
  - `gup_locked|unlocked` (kernel internal dependency)
  - `gup_fast` calling `gup_unlocked` (kernel internal dependency)
  - `MADV_USERFAULT|NOUSERFAULT`
    - `SIGBUS info->si_addr`
  - `remap_anon_pages(dst,src,len)`
  - `ufd = userfaultfd(flags)`

- Stress tested with thousands of postcopy live migrations
- Feedback is welcome to finalize the kernel API
Normal (i.e. Precopy) migration

- Keep copying state over until it's **almost** all there; long enough you can allow it to be down

- Downtime is:
  - Time to copy device state across
  - Time to copy last bit of memory across
    - Depends on guest work load – if it changes ram quickly it might never finish.
Postcopy migration

- Start the destination straight away – before all the RAM is over
- Downtime just the time to transfer other devices
- Each page copied once – upper bound on migration time
- Destination CPU stalls as it waits for pages of RAM
  - Performance of destination reduced until finished
- Can mix with precopy
  - e.g. precopy, switch to postcopy if it's taking too long)
  - Source sends pages anyway before waiting for postcopy requests
- Many previous attempts
  - Yabusame, Hecatonchire, Hines and Gopalan
'Destination CPU stalls as it waits for pages of RAM'

- 'userfault' to mark all of RAM
- 'remap_anon_pages' to place RAM as it arrives
  - Guest CPUs are running – this must be atomic
- Not just guest CPUs
  - QEMU device threads
  - Tricky when loading device state
    - Must be able to service page requests while loading device state from same stream.
## Flow

<table>
<thead>
<tr>
<th>Kernel</th>
<th>mmap USER-FAULT</th>
<th>Pause guest notify fd</th>
<th>remap</th>
<th>restart guest</th>
</tr>
</thead>
<tbody>
<tr>
<td>QEMU</td>
<td>Init</td>
<td>Start guest</td>
<td>read fd, ask for page</td>
<td>read page</td>
</tr>
<tr>
<td>Guest</td>
<td>Access</td>
<td></td>
<td></td>
<td>Full access</td>
</tr>
<tr>
<td>Net-work</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote page</td>
<td>Provide page</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Components

- **Return path** – Dest->src path along same socket
- **Command section** – for sending commands to destination (to change postcopy state)
  - Both Return path and Commands designed to be general
- **Sent map** – source records pages it sent – used by....
- **Discard** – for discarding pages that have been sent during precopy, but are now dirty on the source
- **Incoming map** – destination records pages received and pages requested
- **Userfault handler**
The migration stream

1 'advise' command – Postcopy might happen later
2 normal precopy migration stream of pages
3 'discard' – Sparse bitmap of pages in (2) that have become dirty
4 'package' – A chunk of data loaded off the wire in one go
   4a – 'listen' command – mark RAM as userfault
   4b – device state
   4c – 'run' command – starts destination CPUs running
5 background page transfers
6 Postcopy page transfers - Exactly the same on the wire as (5)
Threads

- Extra threads started before loading device state
  - Because it needs to be able to request pages during device load.
- 'User fault' thread deals with kernel requests and sending them back to source
- 'listener' thread carries on dealing with page loads
Page latencies

- With low wmem – latencies ~1ms
  - (host qemu userspace-userspace)
Page latencies...

- But with standard wmem it shoots up to ~10ms+
- Todo: Limit background page transfer rate to reduce impact on postcopy pages