Memory Externalization With *userfaultfd*

Red Hat, Inc.

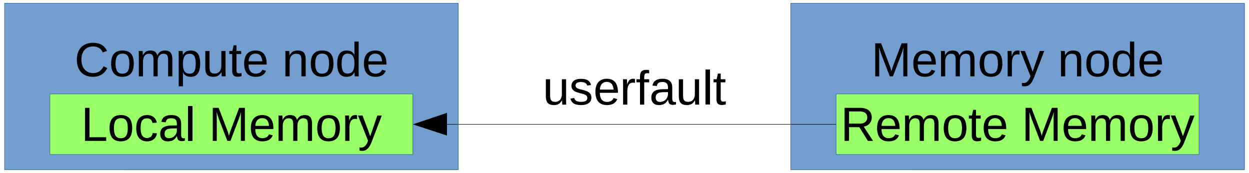
Andrea Arcangeli
aarcange at redhat.com

LSF/MM Summit
Boston, MA

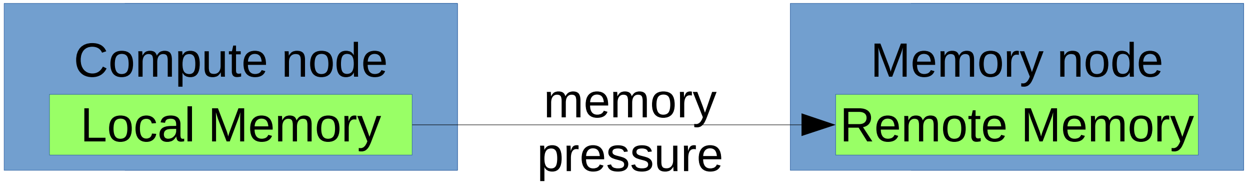
9 Mar 2015

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



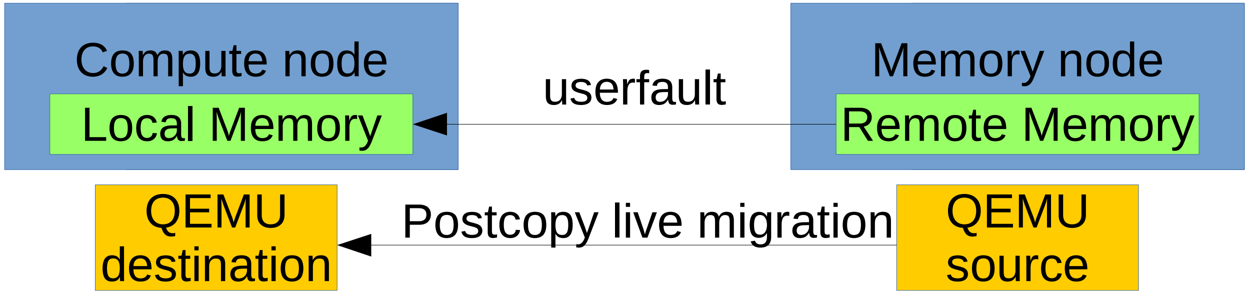
- Memory can be transferred from the compute node to the memory node if it's not frequently used during memory pressure



- 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 some form of 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

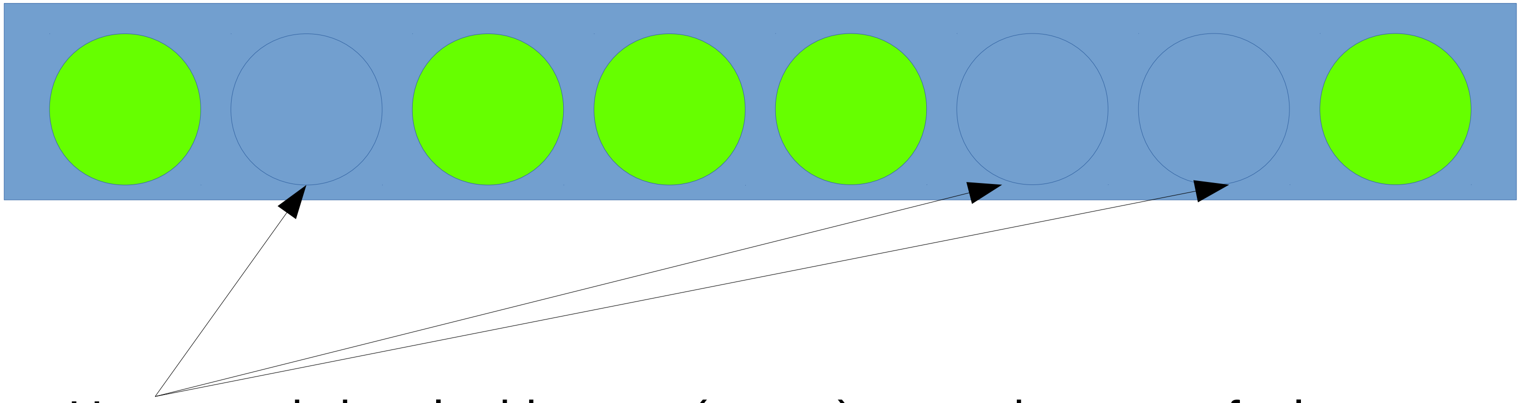
Initial Postcopy Live Migration

- 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

SIGBUS not enough

- SIGBUS is ok to trap userland accesses (like *volatile pages*)
- SIGBUS generates *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 **all** syscalls

SIGBUS not enough

- SIGBUS requires mprotect(PROT_NONE) at PAGE_SIZE granularity
  - Too many vmas
    - Too slow
    - -ENOMEM

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

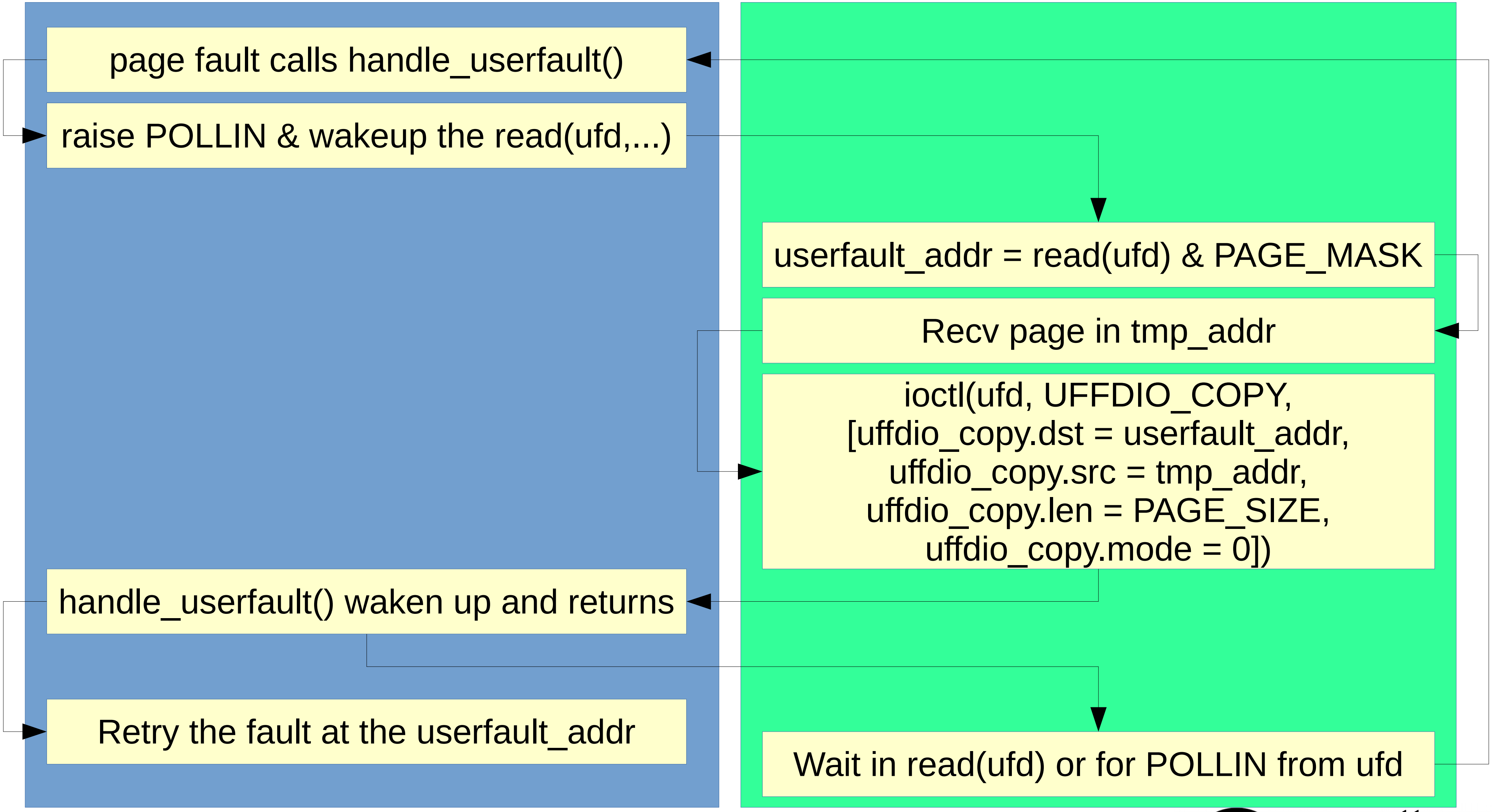
ufd = userfaultfd() - syscall

- 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

How to resolve an userfault

- We must fill the unmapped virtual address
- The unmapped virtual address must be filled ***atomically***
- UFFDIO_REGISTER returns the methods that can be used to resolve an userfault in the uffdio_register.ioctls field:
  - UFFDIO_COPY
  - UFFDIO_ZEROPAGE
  - UFFDIO_WAKE?
    - We must decide if UFFDIO_WAKE shall be retained, it's all about poll semantics..

userfaultfd + UFFDIO_COPY



page fault calls handle_userfault()

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

userfault_addr = read(ufd) & PAGE_MASK

Recv page in tmp_addr

ioctl(ufd, UFFDIO_COPY,
[uffdio_copy.dst = userfault_addr,
uffdio_copy.src = tmp_addr,
uffdio_copy.len = PAGE_SIZE,
uffdio_copy.mode = 0])

handle_userfault() waken up and returns

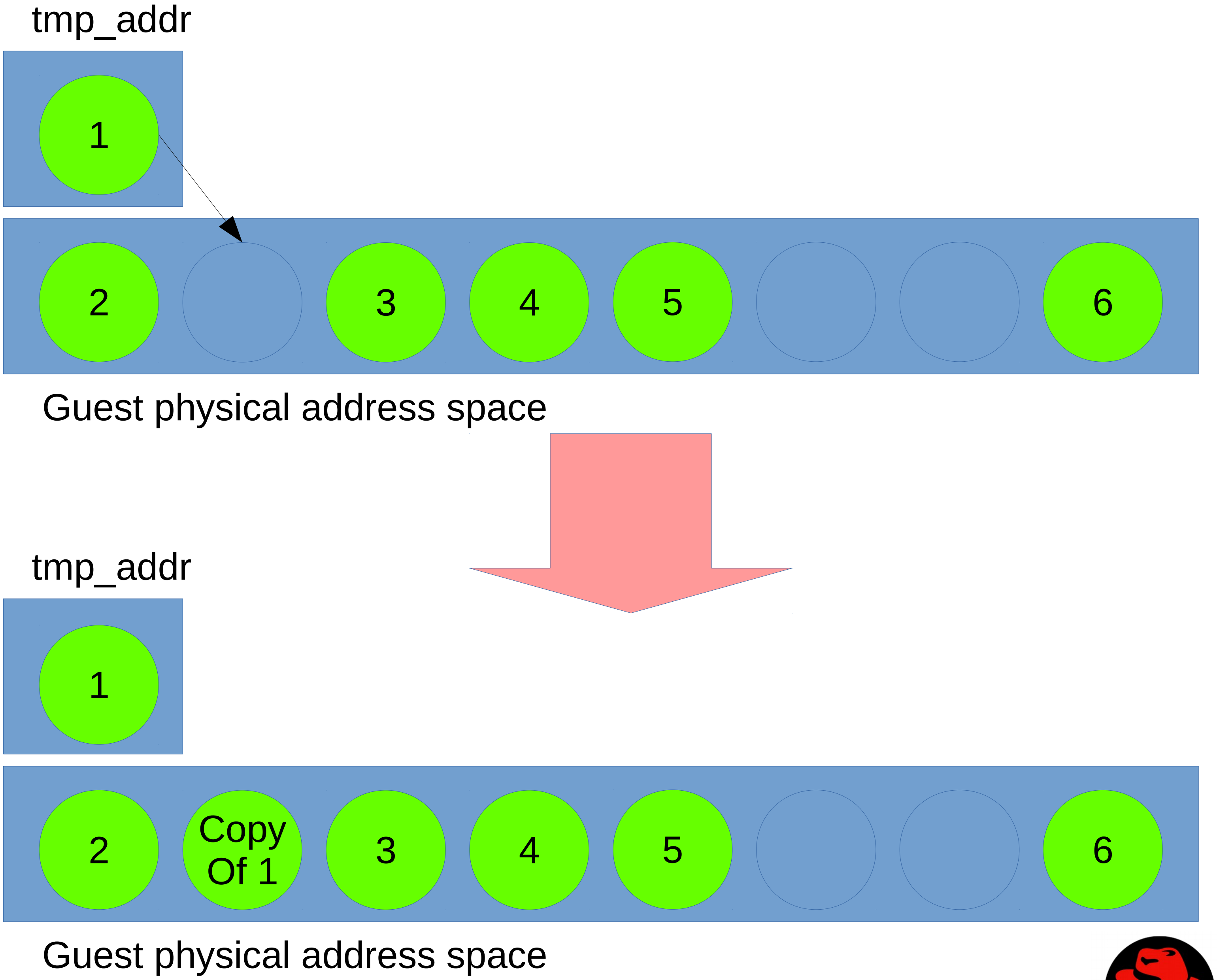
Retry the fault at the userfault_addr

Wait in read(ufd) or for POLLIN from ufd

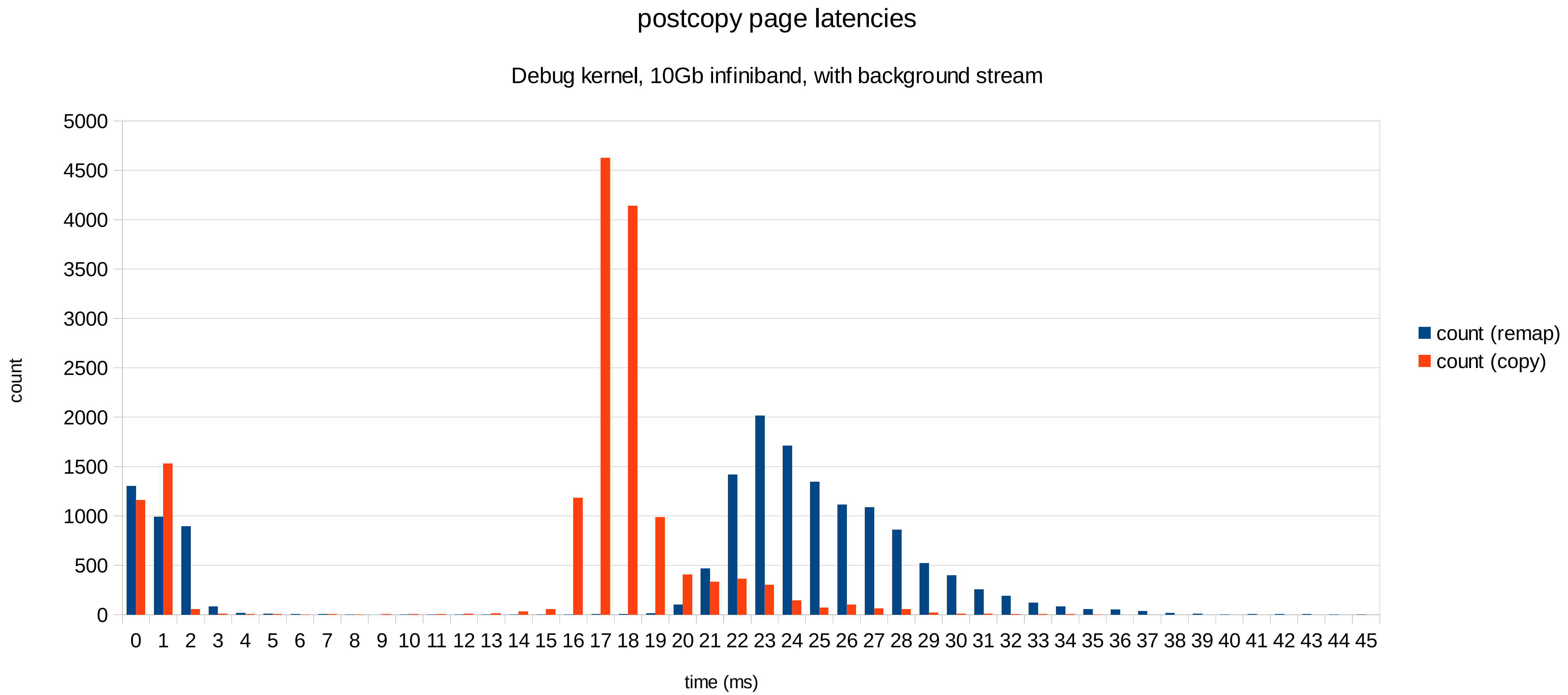
Kernel

Userland thread

UFFDIO_COPY


tmp_addr
1
2
3
4
5
6
Guest physical address space
tmp_addr
1
2
Copy Of 1
3
4
5
6
Guest physical address space

UFFDIO_COPY vs _REMAP



userfaultfd()

- Userfaultfd(flags)
  - Flags
    - UFFD_CLOEXEC
    - UFFD_NONBLOCK

UFFDIO_API

- ioctl(ufd, UFFDIO_API, struct uffdio_api *uffdio_api)

```
struct uffdio_api {
  /* userland asks for an API number */
  __u64 api;

  /* kernel answers below with the available features for the API */
#define UFFD_BIT_WRITE     (1<<0)/* this was a write fault */
  __u64 bits;
  __u64 ioctls;
};
```

- uffdio_api.api = UFFD_API
  - Enforces a known read() protocol

UFFD_API

- read(ufd, &buf, 8)
- read(ufd, &buf, 8*N)

read will write “address” into buf:

```
BUILD_BUG_ON(PAGE_SHIFT < UFFD_BITS);
address &= PAGE_MASK;
if (flags & FAULT_FLAG_WRITE)
        /*
         * Encode "write" fault information in the LSB of the
         * address read by userland, without depending on
         * FAULT_FLAG_WRITE kernel internal value.
         */
         address |= UFFD_BIT_WRITE;
if (reason & VM_UFFD_WP)
         /*
                  * Encode "reason" fault information as bit number 1
          * in the address read by userland. If bit number 1 is
          * clear it means the reason is a VM_FAULT_MISSING
          * fault.
          */
          address |= UFFD_BIT_WP;
```

UFFDIO_REGISTER

- ioctl(ufd, UFFDIO_REGISTER, struct uffdio_register *)

```
struct uffdio_register {
  struct uffdio_range range;
#define UFFDIO_REGISTER_MODE_MISSING   ((__u64)1<<0)
#define UFFDIO_REGISTER_MODE_WP        ((__u64)1<<1)
  __u64 mode;

  /*
   * kernel answers which ioctl commands are available for the
   * range, keep at the end as the last 8 bytes aren't read.
   */
  __u64 ioctls;
};
```

- uffdio_api.ioclts = _UFFDIO_COPY|_UFFDIO_ZEROPAGE

UFFDIO_COPY

- ioctl(ufd, UFFDIO_COPY, struct uffdio_copy *)

```
struct uffdio_copy {
  __u64 dst;
  __u64 src;
  __u64 len;
  /*
   * There will be a wrprotection flag later that allows to map
   * pages wrprotected on the fly. And such a flag will be
   * available if the wrprotection ioctl are implemented for the
   * range according to the uffdio_register.ioctls.
   */
#define UFFDIO_COPY_MODE_DONTWAKE              ((__u64)1<<0)
  __u64 mode;

  /*
   * "copy" and "wake" are written by the ioctl and must be at
   * the end: the copy_from_user will not read the last 16
   * bytes.
   */
  __s64 copy;
  __s64 wake;
};
```

- “copy” tells how many bytes copied successfully

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
    - It could use UFFDIO_ZEROPAGE to resolve the ballon deflate

userfault and live snapshotting

- Track wrprotect faults
  - Throttle the COW memory allocations
- UFFDIO_REGISTER
  - ufddio_register = {.mode = UFFDIO_REGISTER_MODE_WP}
- UFFDIO_WP ioctl
- Trouble:
  - Swap entries requires a wp bit
    - Otherwise even a read swapin fault could make the pte writable if the page is no shared
  - VM_FAULT_RETRY may be returned by a swapin just before UFFDIO_WP marks the swapentry wp
    - SIGBUS may be raised if the race triggers

userfault on shared memory

- Extend UFFDIO_COPY and VM_UFFD_MISSING to tmpfs
- uffdio_register.ioctls will include UFFDIO_COPY bitflag if UFFDIO_REGISTER is run on tmpfs backed memory

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 userfaultfd can provide the notification to applications using volatile pages after they've been reclaimed

Userfault kernel patchset

- Last submit against 3.19-rc:
  - http://thread.gmane.org/gmane.linux.kernel.mm/123575
  - https://lists.gnu.org/archive/html/qemu-devel/2015-03/msg01081.html
  - **git clone git://git.kernel.org/pub/scm/linux/kernel/git/andrea/aa.git -b userfault**
- Feedback is welcome to finalize the kernel API