Live Migration: Even faster, now with a dedicated thread!

Juan Quintela <quintela@redhat.com>

Orit Wasserman <owasserm@redhat.com>

Vinod Chegu <chegu_vinod@hp.com>

KVM Forum 2012

Agenda

- Introduction
- Migration thread
- Live migration of large guests

Introduction

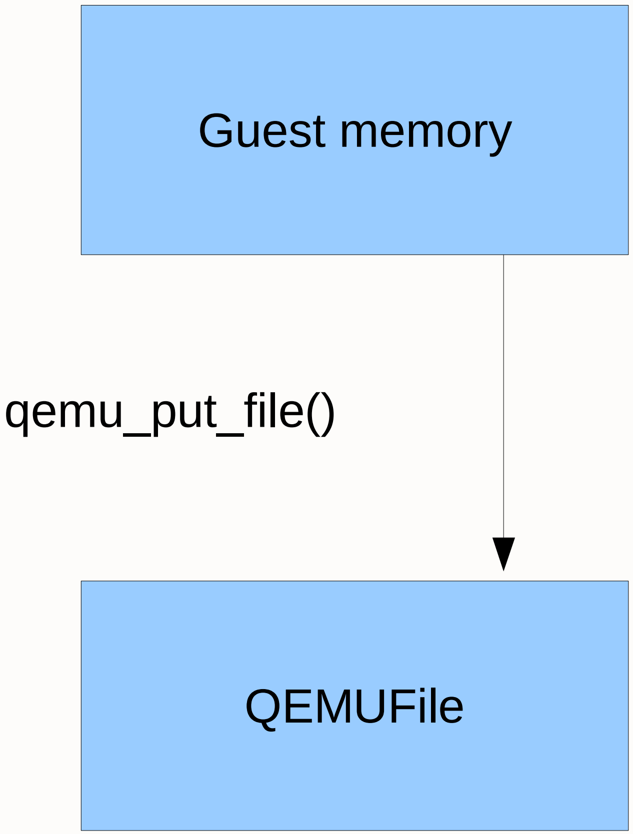
- The problem:

  Moving a guest running in a host to a different host
- To make things interesting:

  Do it without stopping the guest
- Even more interesting:

  And do it fast
- Yes, there are some trouble ahead

Copy in



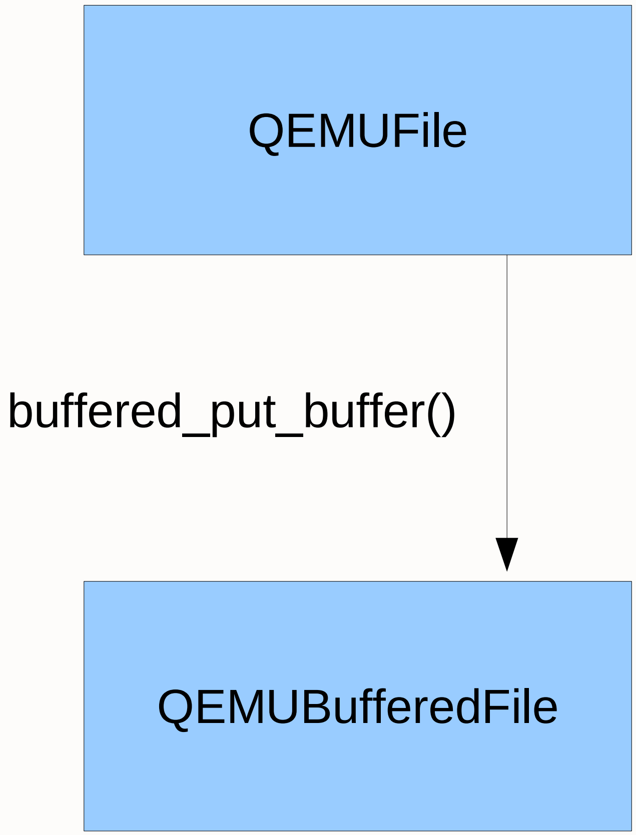
- Copy one
- At some point we call

  qemu_fflush()

- That calls:

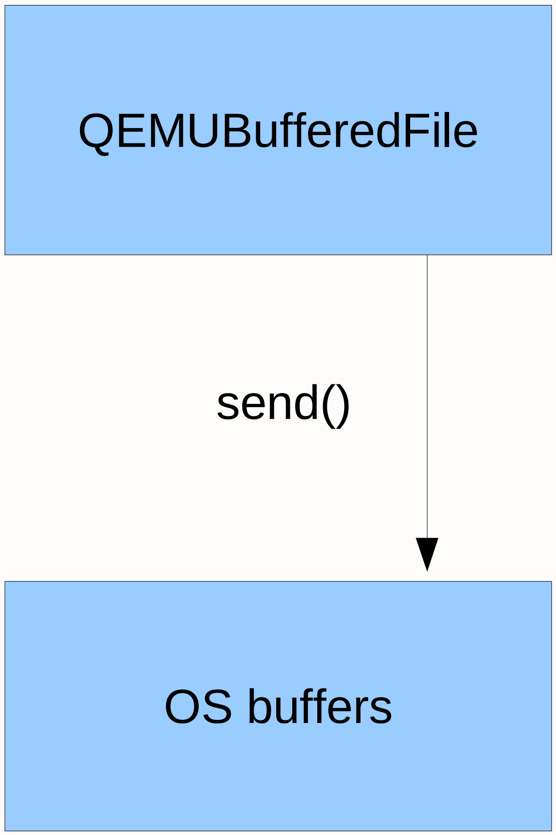
  buffered_put_buffer()

Copy again



- Copy two
- Dynamic buffer
- Grows as needed
- AKA, we can have more copies here

Slow me more



- Copy three
- We can do this synchronously

Migration bitmap

- For each page we use:
  - 1 bit for migration
  - 1 bit for VGA
  - 1 bit for TCG (the mother of all the problems)
- So we end using 1 byte/page
- We can move to three bitmaps of 1bit/page
  - 1 VGA: not all memory regions need it
  - 1 migration
  - 1 TCG: not used while in kvm mode

Bitmap sizes

	1GB	16GB	64GB	256GB	512GB
1 bit/page	32KB	512KB	2MB	8MB	16MB
1 byte/page	256KB	4MB	16MB	64MB	128MB

- We don't need the other bits during migration
- With bigger guests, we blow the cache obviously

Bitmap Sync: Fast algorithm ever

- Qemu alloc a bitmap (1 bit/page) and calls kvm
- KVM kernel module: fill the bitmap
- Qemu kvm wrapper: walk the bitmap and fill the dirty bitmap (1 byte/page)
- Migration code: walk the dirty bitmap, and create a new bitmap (1 bit/page)
- Why it takes 8s to synchonize the bitmap for a 256GB guest? Any idea?

Migration thread: Why?

- Reduce the number of copies
- Separate memory walking and socket writing
- Do writes synchronously (we are in our own thread)
- Makes much, much easier to do bandwidth calculations
- Buffered file not needed anymore

Migration thread: How?

```
while (true) {
    copy_some_dirty_pages_to_buffer();
    write_buffer_to_the_socket();
}
```

Migration thread: How? (II)

```
while (true) {
        mutex_lock_iothread();
        copy_some_dirty_pages_to_buffer();
        mutex_unlock_iothread();
        write_buffer_to_the_socket();
}
```

Downtime

```
synchronize_bitmap();  /* 8 seconds */
write_all_pending_memory_to_buffer();
write_buffer_to_socket();
```

- max_downtime = 2s
- max_speed = 10Gb
- Buffer ~ 2GB
- I hope we were not migrating because we were low of memory
-

Live migration of large guests: Motivation

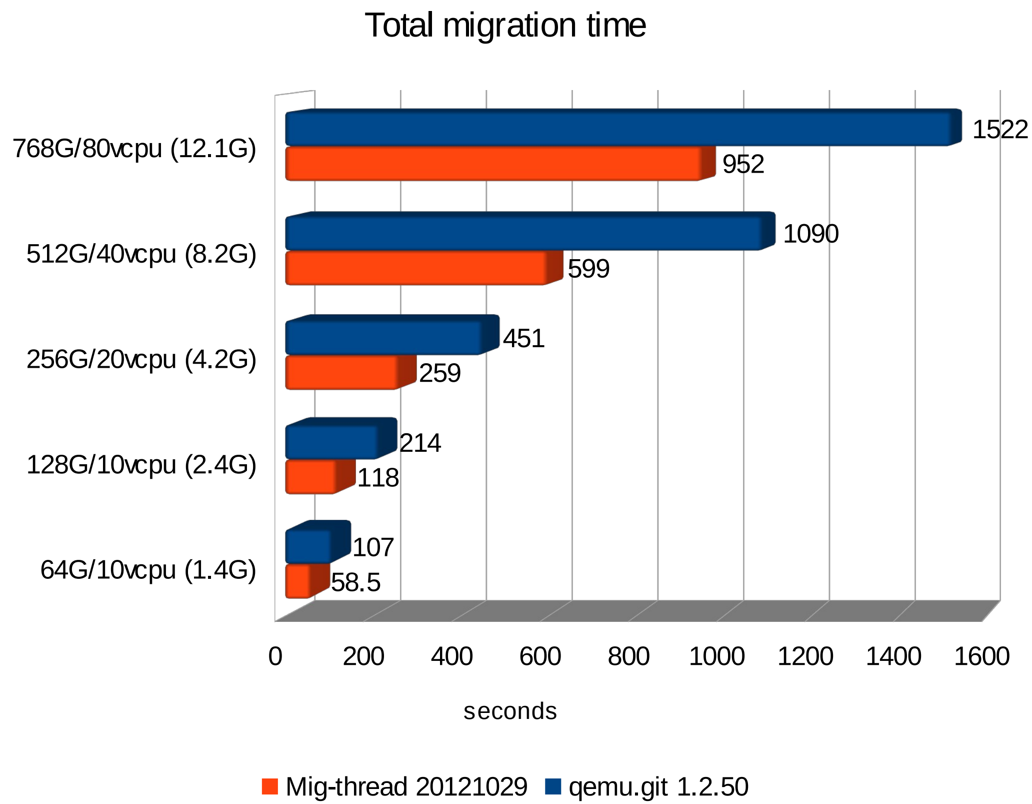
- Increasing need for very large sized VM's
  - Non-enterprise class of workloads (16G-64G)
  - Enterprise workloads (32G - much higher) . E.g. scale-up in-memory DBs
    - Need good scaling & predictable performance
    - Mission critical SLAs and HA.

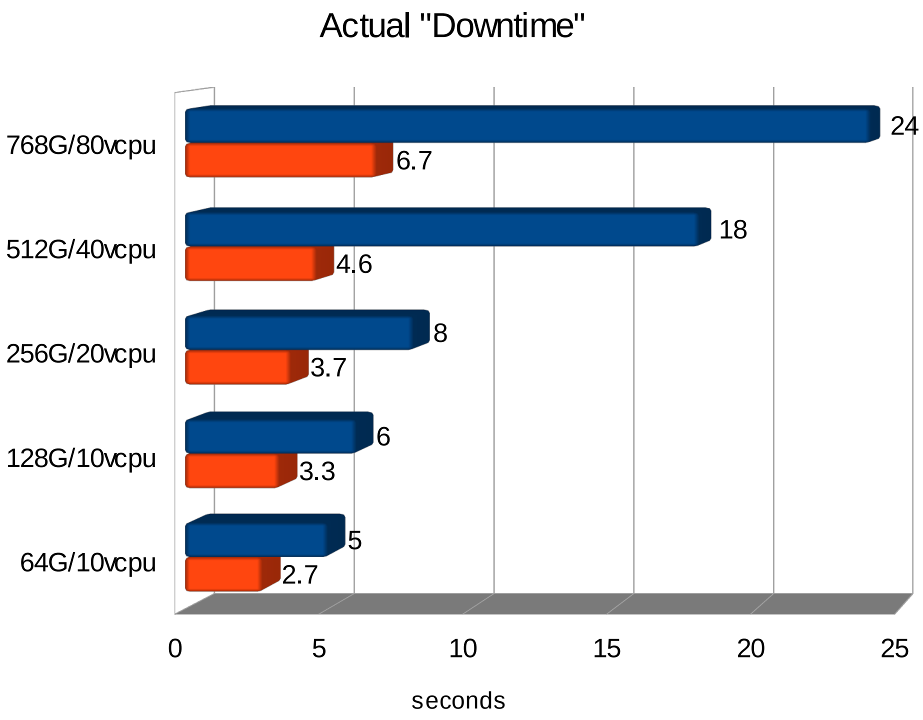
- Demands on Live guest migration
  - Convergence and predictable total migration time.
  - Freeze time (aka "Downtime") – how live is live guest migration?
    - Typically < 5 seconds to avoid dropped tcp/http connections (some apps are more sensitive < 2 seconds)
  - Performance impact on the workload.

Recent optimizations

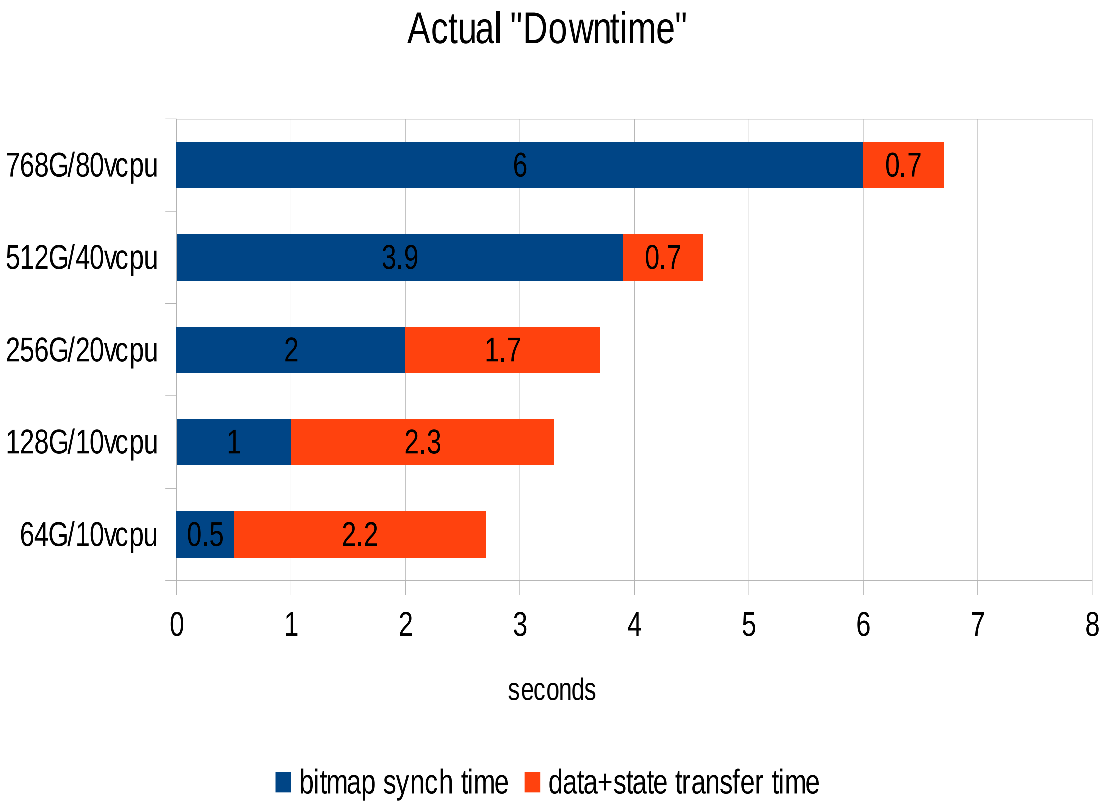
(Idle guest)

Migration speed = 10G & “downtime” = 2secs





Observations

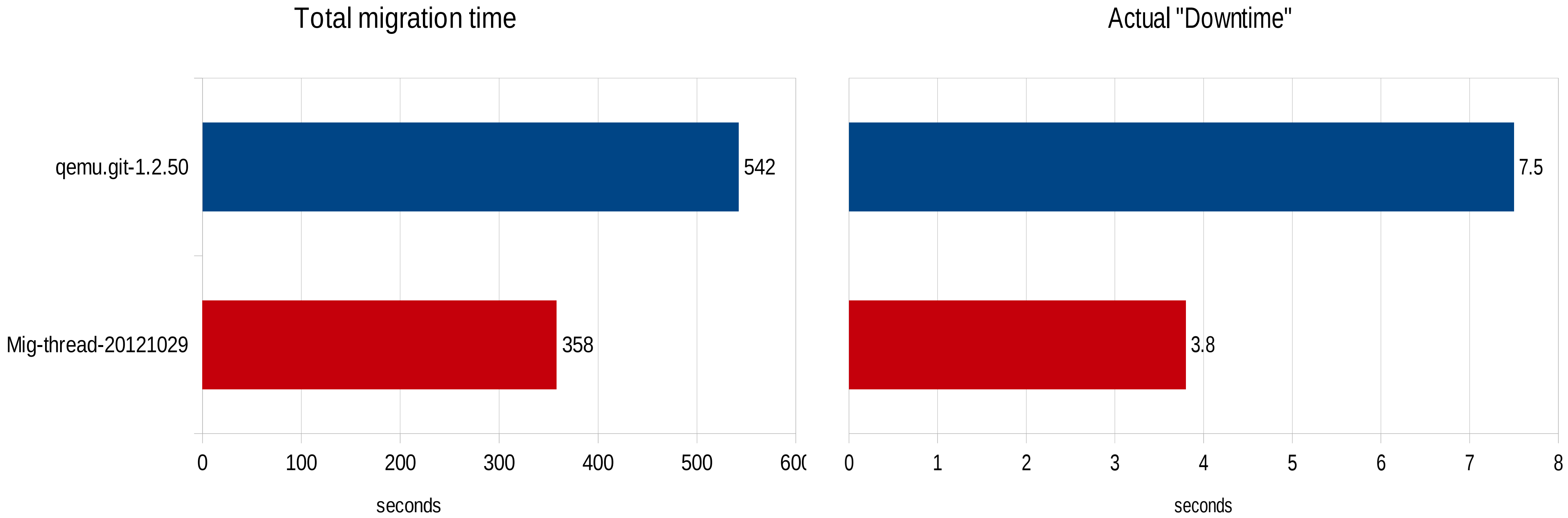


- Bitmap synch-ups for large guests
  - Major contributor to the actual "downtime".
  - Guest freezes during the start of the migration !
- Utilization of allocated B/W
  - Peaks at ~3 Gbps.
  - Perhaps not enough data ready to be sent through the allocated pipe. i.e. Unable to saturate.

SLOB

(256G/80Vcpus, SGA:50G, 96 users)

Migration speed =10G , “downtime” = 2secs



~15-20% degradation in performance during iterative pre-copy phase.

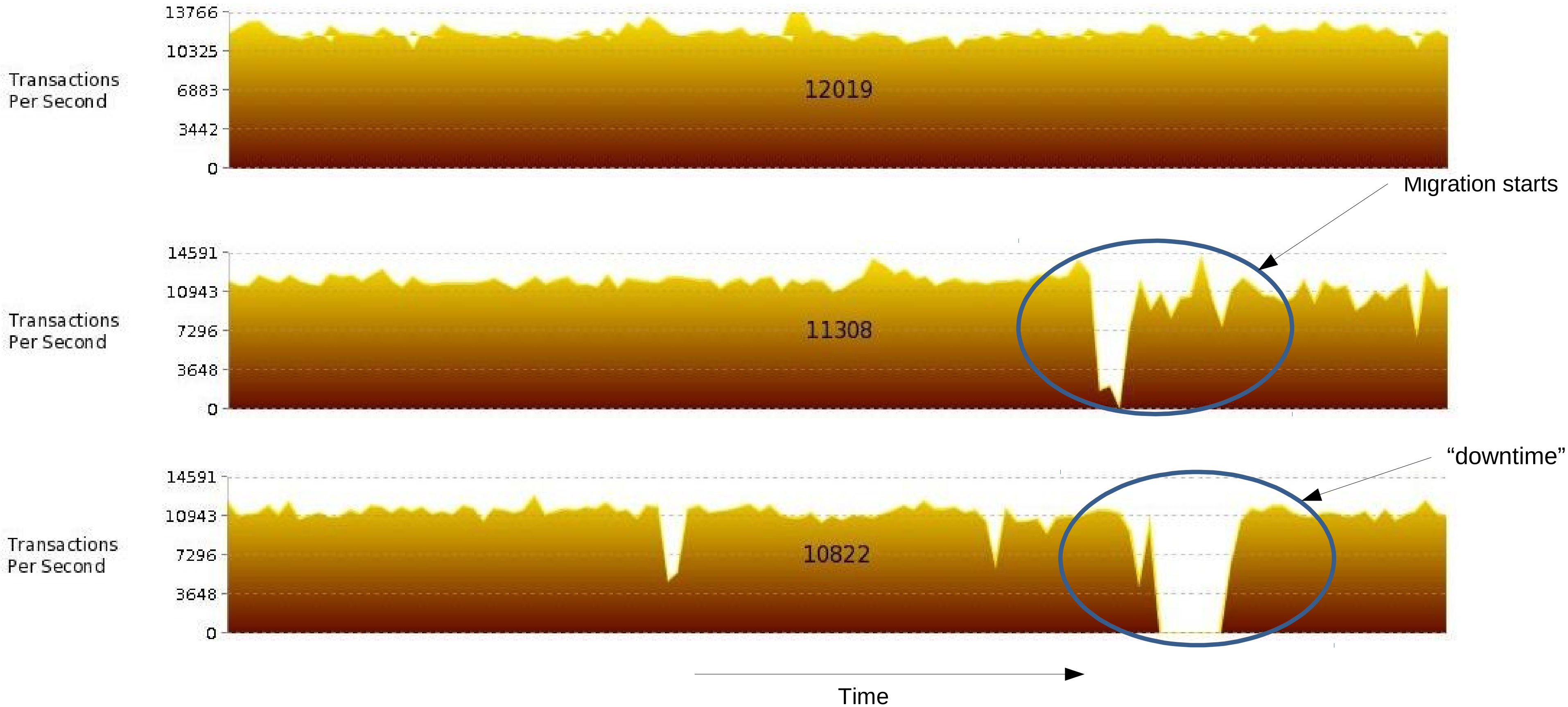
OLTP workload

- Swingbench used to emulate an OLTP type workload
  - 40% of the guest memory is SGA.
  - Using tmpfs instead of real I/O (Note: experiments with I/O will follow later)

OLTP workload

(128G/80VCPUs, 40% SGA, 75 users (CR, BP, OP, PO, BO))

Migration speed =10G , “downtime” = 4secs



Total migration time : 238 secs, Actual “downtime”: 5.7 secs. Transferred RAM: ~

OLTP workload
(128G/80Vcpu, 40% SGA)

Migration speed =10G , “downtime” = 4secs

# users (CR, BP, OP, PO, BO)	% idle	Actual “downtime”	Total migration time
20	~90%	5.5s	195s
40	~70%	5.8s	236s
60	~50%	5.4s	261s
80	~32%	5.3s	255s
100	~25%	No convergence	

~10-15 % degradation in TPS during iterative pre-copy phase.

Observations

- “Visible” impact of guest freezes at the start of the migration.
- Difficult to converge as the workload gets busy.

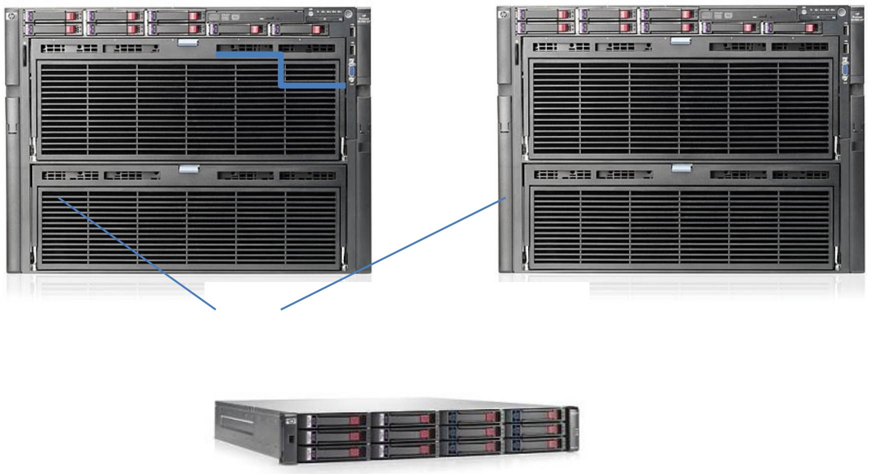
- Need further improvements:
  - Eliminate the freeze time during the start of the migration.
  - Faster Bitmap synch-up
  - Improve usage of allocated bandwidth utilization.
  - Ability to pin the migration thread to a specific pcpu/numa-node.

- Other alternatives:
  - Throttle the workload (via cgroups) – last resort!
  - Post-copy + RDMA approaches.

Backup

Configuration

- Hosts:
  - Pair of HP ProLiant DL980 G7 Server
    - 8 Westmere sockets, 1TB RAM
    - 10Gb NIC's connected back to back.
      - MTU set to 9000, irqbalance off etc.
  - OS: 3.6.0+



- Large sized guests:
  - 2MB Huge pages backed.
  - x2apic enabled
  - PLE turned off (single guest)
  - OS: 3.6.0+

Live migration requirements

- Convergence and predictable – user wants migration to end.
- Downtime – Large downtime can cause guest timeouts.
- Reasonable performance impact on workload

Live migration of large guests convergence problem why ?

- More memory to transfer
- Memory is always much faster than network
- We don't saturate the network

Live migration of large guests profiling

- Guest is 512G with 40vcpu
- Running SLOB with 96 users (all readers) with tmpfs
- Downtime 2 second and migration speed 10G
- Results:

  total time: 685263 milliseconds

  downtime: 7854 milliseconds

  transferred ram: 45472011 kbytes

  total ram: 536879552 kbytes

  duplicate: 125410753 pages

Live migration of large guest profiling

9.26%	**109658**	**qemu-system-x86**	**[kernel.kallsyms]**	**[k]**	**__copy_user_nocache**
7.33%	**82716**	**qemu-system-x86**	**libc-2.12.so**	**[.]**	**memcpy**
3.72%	**41940**	**qemu-system-x86**	**qemu-system-x86_64**	**[.]**	**is_dup_page**
3.57%	324943	qemu-system-x86	[kernel.kallsyms]	[k]	_raw_spin_lock
3.20%	344910	qemu-system-x86	[kernel.kallsyms]	[k]	ktime_get
2.94%	329581	qemu-system-x86	[kernel.kallsyms]	[k]	rcu_check_callbacks
2.84%	**31932**	**qemu-system-x86**	**qemu-system-x86_64**	**[.]**	**cpu_physical_memory_get_dirty**
2.20%	**24742**	**qemu-system-x86**	**qemu-system-x86_64**	**[.]**	**cpu_physical_memory_clear_dirty_flags**
2.15%	239082	qemu-system-x86	[kvm]	[k]	vcpu_enter_guest
1.80%	206891	qemu-system-x86	[kvm_intel]	[k]	vmx_vcpu_run
1.76%	**19763**	**qemu-system-x86**	**qemu-system-x86_64**	**[.]**	**memory_region_get_dirty**
1.65%	18969	swapper	[kernel.kallsyms]	[k]	intel_idle
1.37%	151269	qemu-system-x86	[kernel.kallsyms]	[k]	hrtimer_interrupt
1.36%	**15249**	**qemu-system-x86**	**qemu-system-x86_64**	**[.]**	**cpu_physical_memory_get_dirty_flags**

Live migration of large guests convergence – what can we do?

- Reduce data copies – remove data copies from Qemu code (buffered file)
- Use copy-less networking
- Pinning of the migration thread on a different core than the the vcpu thread

Live migration of large guests convergence – what can we do?

- Reduce bitmap syncing cost – we need to sync between the dirty log in the kvm module (kernel) to Qemu (userspace).
- Reduce bitmap walking cost – by using one bit per page and 64 bit word operations

Live migration of large guests convergence

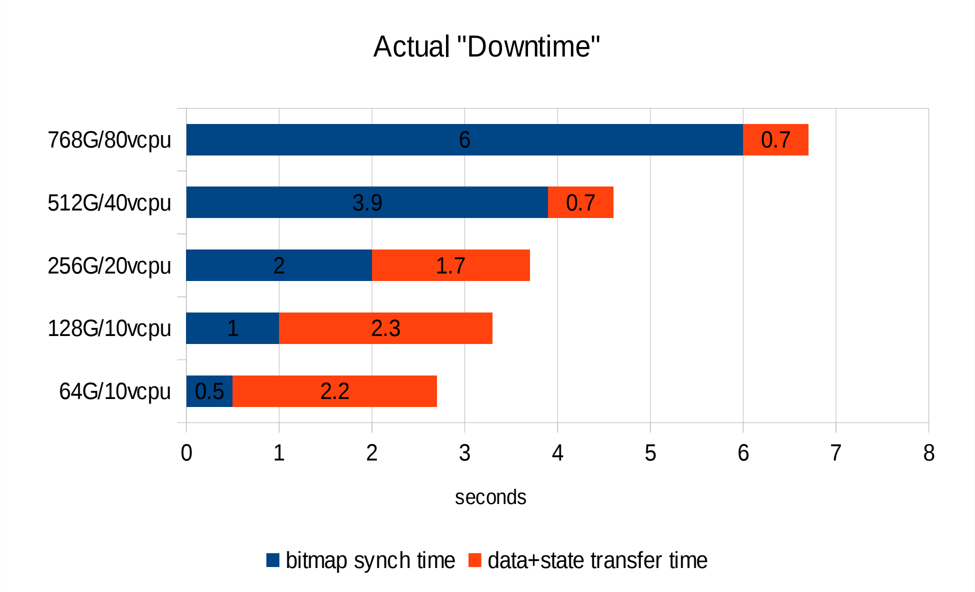
- Packet batching – improve TCP throughput
- Parallel the work
- Faster network (for example we can bond several network card to get higher bandwidth)
- RDMA

Live migration of large guests convergence – what can we do?

- Postcopy live migration – start running the destination immediately, copy the guest memory to destination when the guest access it. More in the next session.
- Slow down the guest (CPU throttling) with cgroup – Red Hat performance team demo in Oracle Open world

Live migration on large guests downtime problem

- Migration bitmap size is proportional to guest memory size
- Syncing/Walking on larger bitmap is more expensive
- Downtime increases with guest size



Reduce downtime for large guest - solutions

- Copy-less networking
- Reduce data copies – instead of coping guest memory page we can use pointers (use writev)
- Bitmap per RamBlock (separate bitmap for migration, VGA and TCG)
  - We can use bit per page (optimize the walk by using 64 bit word at a time)
  - Use memcpy for syncing with the kernel
  - Fine grained locking
  - Allocate bitmap on demand

Reduce downtime for large guest - solutions

- For large guest the bitmap is too large to be in the cache. We can divide the bitmap into ranges that fit in the cache, handle one range at a time.
- Parallelism

Live migration on large guests - guest performance degradation - solutions

- Using EPT/NPT for dirty page logging – the ptes are 64bit long, more work in syncing between kernel and userspace.
- Reduce copies from kernel to user space by using shared memory
- 2M huge pages support in migration – the problem is that the whole 2M page will be marked as dirty, which result in higher traffic.


THANK YOU!