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 \texttt{qemu_fflush()}
- That calls: \texttt{buffered_put_buffer()}

\begin{itemize}
  \item Guest memory
  \item \texttt{qemu_put_file()}
  \item QEMUFFile
\end{itemize}
Copy again

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

QEMUFile

buffered_put_buffer()

QEMUBufferedFile
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

<table>
<thead>
<tr>
<th></th>
<th>1GB</th>
<th>16GB</th>
<th>64GB</th>
<th>256GB</th>
<th>512GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit/page</td>
<td>32KB</td>
<td>512KB</td>
<td>2MB</td>
<td>8MB</td>
<td>16MB</td>
</tr>
<tr>
<td>1 byte/page</td>
<td>256KB</td>
<td>4MB</td>
<td>16MB</td>
<td>64MB</td>
<td>128MB</td>
</tr>
</tbody>
</table>

- 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 synchronize 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

Total migration time

<table>
<thead>
<tr>
<th>Size</th>
<th>Mig-thread 20121029</th>
<th>qemu.git 1.2.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>768G/80vcpu (12.1G)</td>
<td>952</td>
<td>1522</td>
</tr>
<tr>
<td>512G/40vcpu (8.2G)</td>
<td>599</td>
<td>1090</td>
</tr>
<tr>
<td>256G/20vcpu (4.2G)</td>
<td>451</td>
<td>851</td>
</tr>
<tr>
<td>128G/10vcpu (2.4G)</td>
<td>214</td>
<td>414</td>
</tr>
<tr>
<td>64G/10vcpu (1.4G)</td>
<td>107</td>
<td>207</td>
</tr>
</tbody>
</table>

Actual "Downtime"

<table>
<thead>
<tr>
<th>Size</th>
<th>Mig-thread 20121029</th>
<th>qemu.git 1.2.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>768G/80vcpu</td>
<td>6.7</td>
<td>24</td>
</tr>
<tr>
<td>512G/40vcpu</td>
<td>4.6</td>
<td>18</td>
</tr>
<tr>
<td>256G/20vcpu</td>
<td>3.7</td>
<td>8</td>
</tr>
<tr>
<td>128G/10vcpu</td>
<td>3.3</td>
<td>6</td>
</tr>
<tr>
<td>64G/10vcpu</td>
<td>2.7</td>
<td>5</td>
</tr>
</tbody>
</table>

Mig-thread 20121029  qemu.git 1.2.50
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" = 4 secs

Total migration time: 238 secs, Actual "downtime": 5.7 secs. Transferred RAM: ~
## OLTP workload

(128G/80Vcpu, 40% SGA)

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

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

~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

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
<th>Process</th>
<th>Module</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>__copy_user_nocache</td>
<td>9.26%</td>
<td>qemu-system-x86</td>
<td>[kernel.kallsyms]</td>
<td>[k]</td>
</tr>
<tr>
<td>memcpy</td>
<td>7.33%</td>
<td>qemu-system-x86</td>
<td>libc-2.12.so</td>
<td>[.]</td>
</tr>
<tr>
<td>is_dup_page</td>
<td>3.72%</td>
<td>qemu-system-x86</td>
<td>qemu-system-x86_64</td>
<td>[.]</td>
</tr>
<tr>
<td>_raw_spin_lock</td>
<td>3.57%</td>
<td>qemu-system-x86</td>
<td>[kernel.kallsyms]</td>
<td>[k]</td>
</tr>
<tr>
<td>ktime_get</td>
<td>3.20%</td>
<td>qemu-system-x86</td>
<td>[kernel.kallsyms]</td>
<td>[k]</td>
</tr>
<tr>
<td>rcu_check_callbacks</td>
<td>2.94%</td>
<td>qemu-system-x86</td>
<td>[kernel.kallsyms]</td>
<td>[k]</td>
</tr>
<tr>
<td>cpu_physical_memory_get_dirty</td>
<td>2.84%</td>
<td>qemu-system-x86</td>
<td>qemu-system-x86_64</td>
<td>[.]</td>
</tr>
<tr>
<td>cpu_physical_memory_clear_dirty_flags</td>
<td>2.20%</td>
<td>qemu-system-x86</td>
<td>qemu-system-x86_64</td>
<td>[.]</td>
</tr>
<tr>
<td>vcpu_enter_guest</td>
<td>2.15%</td>
<td>qemu-system-x86</td>
<td>[kvm]</td>
<td>[k]</td>
</tr>
<tr>
<td>vmx_vcpu_run</td>
<td>1.80%</td>
<td>qemu-system-x86</td>
<td>[kvm_intel]</td>
<td>[k]</td>
</tr>
<tr>
<td>memory_region_get_dirty</td>
<td>1.76%</td>
<td>qemu-system-x86</td>
<td>qemu-system-x86_64</td>
<td>[.]</td>
</tr>
<tr>
<td>intel_idle</td>
<td>1.65%</td>
<td>swapper</td>
<td>[kernel.kallsyms]</td>
<td>[k]</td>
</tr>
<tr>
<td>hrtimer_interrupt</td>
<td>1.37%</td>
<td>qemu-system-x86</td>
<td>[kernel.kallsyms]</td>
<td>[k]</td>
</tr>
<tr>
<td>cpu_physical_memory_get_dirty_flags</td>
<td>1.36%</td>
<td>qemu-system-x86</td>
<td>qemu-system-x86_64</td>
<td>[.]</td>
</tr>
</tbody>
</table>
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 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

![Graph of actual downtime](image)
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.