ORACLE

QEMU live migration device state transfer parallelization via multifd channels

Oracle

Maciej S. Szmigiero
Linux Virtualization and Security
September 22, 2024

 | |

Problem statement

Aka: What this talk is about?

- Current QEMU live migration device state transfer is done via the (single) main migration channel
- Such way of transferring device state migration data reduces performance and severally impacts the migration downtime for VMs having large device state that needs to be transferred during the switchover phase

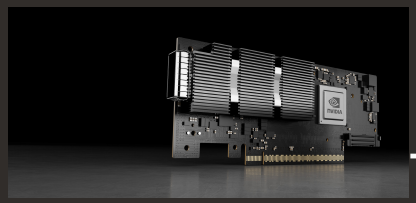
Problem statement

Aka: What this talk is about?

- Current QEMU live migration device state transfer is done via the (single) main migration channel
- Such way of transferring device state migration data reduces performance and severally impacts the migration downtime for VMs having large device state that needs to be transferred during the switchover phase
- It's 2024, multi-core / SMP systems *do* exist
- Some examples of devices that have such large switchover phase device state are:
  - Some types of VFIO SmartNICs
  - GPUs

Problem statement

Aka: What this talk is about?



The effort concentrated on setups with Mellanox ConnectX-7 smart NIC

- So some information is specific to this device and its mlx5 Linux kernel driver
- 128 GiB of on-board RAM, so *a lot* of possible device state
  - Realistic setups are on order of hundreds of MiB of device state per VF
    - Unfortunately, only small part can be pre-copied
  - But most of the talk is generic to any VFIO device
    - GPUs likely have similar amounts of (Video)RAM, if not more
    - With some optimizations even more generic

Problem statement

Aka: What this talk is about?

This talk describes the efforts to parallelize the transfer and loading of device state for these VFIO devices by utilizing QEMU existing support for having multiple migration connections

- That is, by utilizing multifd channels for their transfer, together with other parallelization improvement
- The main metric being optimized is live migration downtime
  - Especially how it scales with number of VFIO devices and their device state size

Multifd

Multifd introduced in QEMU 2.11 in late 2017 by Juan Quintela

- Prior art showed that single migration stream becomes CPU-bound at about 10 Gbit/s
  - `https://wiki.qemu.org/Features/Migration-Multiple-fds`
  - Current data center NICs are 100+ GBit/s
- RAM transfer only
- Single thread queuing RAM pages to be sent
- Multiple threads (channels) doing actual sending and receiving
  - Easy implementation for RAM transfer since these threads just need to read/write appropriate HVA

 | |

Multifd

Multifd introduced in QEMU 2.11 in late 2017 by Juan Quintela

- Prior art showed that single migration stream becomes CPU-bound at about 10 Gbit/s
  - `https://wiki.qemu.org/Features/Migration-Multiple-fds`
  - Current data center NICs are 100+ GBit/s
- RAM transfer only
- Single thread queuing RAM pages to be sent
- Multiple threads (channels) doing actual sending and receiving
  - Easy implementation for RAM transfer since these threads just need to read/write appropriate HVA
- A good starting point but needs some extending to be usable to transferring device state

 | |

Current state

- Single migration stream introduces interlocking between different devices
  - Because device state is saved / loaded in-order
  - Since every NIC VF means a separate VFIO QEMU device this scales really poorly
  - Theoretically partially fixable by decoupling data transfer operation from its handling
    - AFAIK no one announced such effort
- Until recently, kernel-side VFIO state saving and loading wasn't pipelined

 | |

History of improvements

- QEMU changes only show performance benefits insofar there aren't bottlenecks on the kernel/device side
- Mellanox/NVIDIA improving loading
- Chunk mode merged in Sep 2023
  - Double buffering
  - Separate device reading task ("work") from the QEMU reading thread
  - `https://lore.kernel.org/kvm/20230911093856.81910-1-yishaih@nvidia.com/`
  - Released in kernel 6.7

Channel headers

Channel type detection is based solely on whether the received data starts with QEMU_VM_FILE_MAGIC

- If yes - that's the main migration channel
- Everything else is considered a multifd channel
- Not very flexible

Channel headers

Channel type detection is based solely on whether the received data starts with QEMU_VM_FILE_MAGIC

- If yes - that's the main migration channel
- Everything else is considered a multifd channel
- Not very flexible
- Channel headers to the rescue
  - Introduce explicit header for migration channels
  - Originally developed by Avihai Horon from NVIDIA
  - Included in the RFC (version 0) of my patch set

Channel headers

Channel type detection is based solely on whether the received data starts with QEMU_VM_FILE_MAGIC

- If yes - that's the main migration channel
- Everything else is considered a multifd channel
- Not very flexible
- Channel headers to the rescue
  - Introduce explicit header for migration channels
  - Originally developed by Avihai Horon from NVIDIA
  - Included in the RFC (version 0) of my patch set
  - Unfortunately, weren't accepted :(
    - We have to live with extending the existing migration bit stream

 | |

Dedicated channels

The RFC version (version 0) of patch set had dedicated device state transfer multifd channels

- These seem to improve downtime somewhat
  - Dedicated device state transfer multifd channels don't need to participate in RAM sync process
  - ~14% improvement over shared multifd channels configuration in simple/naive tests (1250 msec vs 1100 msec)
    - 15:4 multfd channels total:dedicated config vs 11:0 or 15:0 multfd total:dedicated channels config
    - Higher channel counts did not improve downtime, above certain count even negative impact
  - At that point lacked an automated testing tool that would allow collecting many downtime samples

 | |

Dedicated channels

The RFC version (version 0) of patch set had dedicated device state transfer multifd channels

- These seem to improve downtime somewhat
  - Dedicated device state transfer multifd channels don't need to participate in RAM sync process
  - ~14% improvement over shared multifd channels configuration in simple/naive tests (1250 msec vs 1100 msec)
    - 15:4 multfd channels total:dedicated config vs 11:0 or 15:0 multfd total:dedicated channels config
    - Higher channel counts did not improve downtime, above certain count even negative impact
  - At that point lacked an automated testing tool that would allow collecting many downtime samples
- Shared channels weren't exactly liked by reviewers so need to find out if it is possible to remove them
  - Whether it is possible to achieve the same performance (downtime) with just shared channels

 | |

No dedicated channels

- It tuned out that achieving the same performance (downtime) with shared channels *is* indeed possible
- The key is to avoid using *too many* multifd channels
  - The same downtime as in 15:4 can also be achieved in the 6:0 config - that, is by reducing the total multifd channel count to 6
    - This actually needed a new dedicated automated testing tool
  - Above certain point further channel count increases actually *increase* downtime
- We probably want to learn the detailed reason for this someday
  - It is *not* about RAM syncs as these were measured to have hardly any time impact
  - In general, having to tune the channel count to the particular workload is bad
  - We probably want to have as wide “channel count tolerance band” as possible

Current design

- 1:1 VFIO devices to threads mapping
  - Saving/queuing thread
    - Reads device state from the device (kernel driver) and queues it to a multifd channel
  - Loading thread
    - Tries to write back buffers to the device (kernel driver) in-order once they become available
- Limit of buffered device state size to cap the max receiving QEMU process memory allocations

Current thread design

- Loading threads in multifd-managed thread pool
  - Possible to use fewer threads than devices
- Details still being discussed with community
  - We probably be basing the generic QEMU thread pool on Glib's GThreadPool rather than QEMU AIO thread pool
  - Requires some extensions to GThreadPool like thread_pool_wait()
  - Might want to upstream this to Glib first rather than open code in QEMU
- Other threads still VFIO-driver managed
  - It's unclear if managing them in centralized way would add benefits that offset extra complexity

Current design

Device state change only partially parallelized

- Only save-side VFIO_DEVICE_STATE_STOP_COPY -> VFIO_DEVICE_STATE_STOP
- Other state transitions might be parallelized in the future too
  - In case of VF of single NIC requires that these be truly separate performance-wise in the device
- Some challenges in further load parallelization
  - For example, current VFIO VMState has to be loaded in the main migration thread
    - Calls QEMU core address space methods which AFAIK require BQL
    - Probably long-term fixable with some QEMU core adaptations/extensions

Benchmarks

Test setup

- Source machine:
  - 2x Xeon Gold 5218
  - 192 GiB RAM
  - Mellanox ConnectX-7 with 100GbE link
  - 6.9.0-rc1+ kernel
- Target machine:
  - 2x Xeon Platinum 8260
  - 376 GiB RAM
  - Mellanox ConnectX-7 with 100GbE link
  - 6.6.0+ kernel
- VM:
  - vCPU 12 cores x 2 threads
  - 15 GiB RAM
  - From 1 to 4 Mellanox ConnectX-7 VFs
    - Unfortunately, this setup does not allow testing more than 4 VFs

Benchmarks

QEMU setup

- x-return-path=true
- x-switchover-ack=true
- ~100 MiB of device state per VF
- Benchmarking methodology:
  - Multiple runs to catch truly idle guest using a dedicated automated testing tool
    - Sometimes takes up to 30 live migrations for downtime value to stabilize
    - 70+ runs done to be sure
  - Guest NOT restarted between runs
    - Guest could be busy during some of these runs
  - Guest restarted for different configurations

Performance results

Downtime with large VFIO total device state size

- 6 multifd shared channels
- VFIO device state size roughly the same each run
- Lowest downtime value taken
  - Not measuring dirty RAM transfer speed after all

Results:

	4 VFs	2 VFs	1 VF
Multifd device state transfer disabled	1783 ms	614 ms	283 ms
Multifd device state transfer enabled	1068 ms	434 ms	274 ms
IMPROVEMENT	~67%	~40%	~3%

Performance results

Downtime with large VFIO total device state size with different multifd channel counts

- 4 VFs
- Around 430 MiB of device state

Results:

	Lowest	25th percentile	Median	75th percentile
6 channels	1068 ms	1092 ms	1130 ms	1204 ms
8 channels	1067 ms	1116 ms	1157 ms	1251 ms
12 channels	1072 ms	1209 ms	1263 ms	1310 ms
15 channels	1170 ms	1256 ms	1299 ms	1364 ms
20 channels	1117 ms	1273 ms	1310 ms	1376 ms
25 channels	1225 ms	1279 ms	1315 ms	1356 ms

Patch set status

- Version 2 of patch set posted, versions 3 will be posted soon
  - At the beginning there was an RFC - which was essentially the version zero of this patch set
- Hopefully the basic patch set will make QEMU 9.2

Additional concurrency

- QEMU migration code isn't async and does not return to the main loop
  - Would require significant effort as QEMU migration API consumer / user drivers would need to be event-driven too
- AIO support for kernel still missing
  - Requires kernel API changes
- Unordered VFIO device state loading would simplify userspace (QEMU) code
  - The kernel mlx5 driver doesn't even support non-blocking / poll()-style device state loading currently
    - OTOH, non-blocking / poll()-style device state *reading* is already possible

Additional concurrency

- QEMU migration code isn't async and does not return to the main loop
  - Would require significant effort as QEMU migration API consumer / user drivers would need to be event-driven too
- AIO support for kernel still missing
  - Requires kernel API changes
- Unordered VFIO device state loading would simplify userspace (QEMU) code
  - The kernel mlx5 driver doesn't even support non-blocking / poll()-style device state loading currently
    - OTOH, non-blocking / poll()-style device state *reading* is already possible
- In the end, we probably want to be fully CPU bound
  - Either host CPU or VFIO device on-board {C,G}PU - best by both

 | |

Future directions

- Live stage device state transfer
  - Won't improve downtime but might improve overall migration time
  - More complex scheduling as devices can constantly generate new *dirty* data to be transferred
  - Would need switchover point estimation improvements to be truly effective
    - Having large device state that can't be transferred during the VM live phase breaks expected downtime estimation
    - Currently one has to manually set migration channel bandwidth (avail-switchover-bandwidth) for things to work correctly
- Extending more devices with multifd device state transfer
  - Quite obvious future direction but scales poorly in terms of amount of benefit per amount of work
- Generic framework for parallel VMState transfer?
- Multifd compression support?

Q & A

Questions?