


# Improving live migration efficiency in QEMU: a paravirtualized approach

Filippo Storniolò<sup>[0009-0002-3426-025X]</sup>, Luigi Leonardi <sup>[0000-0003-3014-1691]</sup>,  
and Giuseppe Lettieri<sup>[0000-0003-1005-7441]</sup>

University of Pisa, Pisa, Italy  
f.storniolò@studenti.unipi.it, luigi.leonardi@phd.unipi.it,  
giuseppe.lettieri@unipi.it

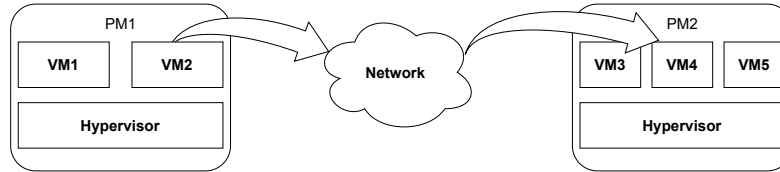
**Abstract.** Virtual Machines are the key technology in cloud computing. In order to upgrade, repair or service the physical machine where a Virtual Machine is hosted, a common practice is to live-migrate the Virtual Machine to a different server. This involves copying all the guest memory over the network, which may take a non-negligible amount of time. In this work, we propose a technique to speed up the migration time by reducing the amount of guest memory to be transferred with the help of the guest OS. In particular, during live-migration, a paravirtualized driver running in the guest kernel obtains, and sends to the Virtual Machine Monitor, the list of guest page frames that are currently unused. The VMM can then safely skip these pages during the copy. We have integrated this technique in the live-migration implementation of QEMU [3], and we show the effects of our work in some experiments comparing the results against QEMU default implementation and VirtIO-Balloon.

**Keywords:** Paravirtualization · Virtualization · Live Migration · QEMU

## 1 Introduction

Most of the cloud computing infrastructure relies on Virtual Machines (VMs) and usually has strong requirements on downtime periods to be as low as possible. This is not an easy task to achieve because Physical Machines (PMs) or Host Machines, i.e. where the VMs are hosted, may require maintenance for several reasons: hardware failures, periodic checks or upgrades. If one of such machines is turned off without any precaution, all the VMs, and the services running on them, become suddenly unavailable. One possible solution is to move all the hosted VM(s) to another computer before shutting down the PM. This should be completed as fast as possible to reduce downtime for the users. Migrating VMs can also be useful for load balancing [16, 1]: a VM that is running on an overloaded server and experiencing degraded performance, can be moved to a lightly loaded server. Finally, a server experiencing a low load can be turned off, after copying its VMs to another machine, thus saving power. Overall, migration can be used for optimizing the usage of all available PMs.

Live Migration [4] is a technique that is available in most of the hypervisors like Xen [2] or KVM [9] and for containers like Linux Containers LXC [13, 5]



**Fig. 1.** Example of Live Migration of VM2 from PM1 to VM4 in PM2

or Docker [17]. Because most of the migration time is spent for the data copy of the *guest* context to the *target* VM, the idea is to perform it while keeping the *source* machine running. It is not possible to copy the entire state because the running machine constantly modifies (a part of) its memory known as its *working set*, so the VM needs to be shut down at some point to copy this set (stop and copy). However, since the size of the *working set* is smaller than the entire *guest* state, the downtime is greatly reduced. Optimizing the downtime or the migration time is not an easy task and there are several routes that have been explored: in [8] the authors introduced a novel stopping condition based on the rate of page transmission during migration, Svärd et al. in [14] explained how delta compression techniques can reduce the amount of data to be sent.

The migration process can be further optimized by skipping all the guest pages that are marked as free, but because of the *semantic gap* that exists between the *host* and the *guest* system, the VMM is unaware of the status of each memory frame inside the VM. This gap can be overcome using VM Introspection [6] (VMI) that consists in analyzing the guest memory content from the hypervisor. Using the latter technique Wang et al. [15] implemented this optimization by ignoring caches and free pages, drastically reducing the data transferred by  $\approx 70\%$ . However, this promising method has some limitations: in the case of a VM with an encrypted RAM, as found in modern confidential computing environment, the method cannot be used, since the hypervisor cannot decrypt the guest memory. This limitation can be addressed by paravirtualization: the virtualized system, or at least some part of it, is aware that it is running inside a VM and is willing to exchange information with the Hypervisor(HV). One example is Ballooning [7]: with the latter technology the HV can reclaim part of the guest’s memory by “inflating” the balloon before starting the migration: this limits the amount of RAM available for the guest, and therefore also the amount of data to be transferred. However, this does not come for free: reducing the guest’s memory to give it back to the hypervisor, implies that the guest cannot use that memory until it is released by the host. In fact, the hypervisor may use the regained memory to run a new VM, for example, and if the guest could use that portion of memory, it could read/write the memory of another VM. In this work we propose a paravirtualized driver that sends to the HV the list of page frames that are free when the migration starts. In our solution, the guest still owns the memory for the duration of the migration and can use it as

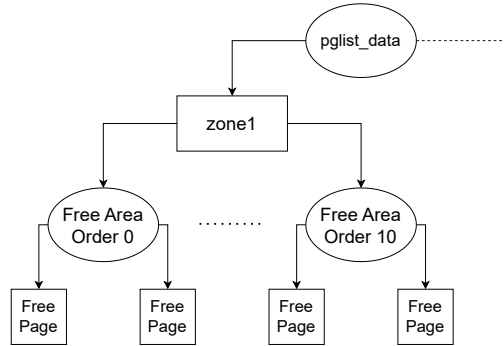
needed. A similar solution has been proposed for the Xen hypervisor [12]. We provide and evaluate a complete implementation for the QEMU hypervisor [3] by reusing a generic paravirtualization device [10].

The rest of the paper is organized as follows: Section 2 provides the necessary background on the existing technologies (Linux memory management, live migration and Ballooning in QEMU); Section 3 describes our proposed paravirtualized migration; Section 4 shows the results of some experiments and Section 5 concludes.

## 2 Background

### 2.1 Linux physical memory

Linux physical memory is organized in blocks of physical pages with different sizes. Each block is built from a set of consecutive physical pages, the number of which is always a power of 2. The power exponent is usually called *order* and it can vary from 0 to 10. This means that the size of the smallest and the biggest block of physical pages is respectively 4 KiB (that is the size of a single physical page) and 4 MiB (that is the size of  $2^{10}$  consecutive physical pages).



**Fig. 2.** Linux Physical Memory Structure

Whenever the Kernel needs to allocate memory for some processes, it needs to find a list of free blocks of physical pages. This information can be searched through a hierarchy of data structures. From top to bottom, the first layer is represented by the *node* data structure. There are as many nodes as the number of NUMA nodes, so in UMA machines just one node is present. Each node is responsible for a certain number of *zones*, which in turn handle an array of *free area* data structures: each of these contains a pointer to a list of free blocks of pages with the same order. Since memory allocation can occur concurrently, a synchronization mechanism to protect the critical section is required. For this reason, each *zone* contains a spin-lock.

## 2.2 QEMU live-migration

QEMU [3] is a widely used hypervisor originally based on binary-translation and later extended to hardware-assisted virtualization [9]. QEMU implements the so called *Pre-Copy live-migration*, since the memory is transferred before the *Guest* runs in the new environment. In fact—when the migration task is terminated—the *Guest* is ready to run as if nothing had ever happened.

QEMU implements this technique using three stages.

- In the first stage all the *Guest* RAM is marked as dirty.
- The second stage is an iterative one: when the latter starts, the Hypervisor keeps sending to the new Virtual Machine on the destination Host all the dirty memory. However, the Hypervisor may send the same portion of memory more than once, since the *Guest* kernel and the *Guest* processes may still perform write operations on memory, making it dirty again. In order to end the second stage, watermarks or specific ending conditions must be chosen and reached (such as watermarks on the minimum amount of memory left to be transferred or a maximum number of iterations).
- During the third and final stage, the Hypervisor momentarily stops the *Guest* in order to transfer the last portion of dirty memory, the CPU and the other peripherals state.

Metrics such as setup time (time spent in the first stage), downtime (time spent in the third stage) and total memory sent can be read by checking the migration state on the QEMU Monitor.

## 2.3 VirtIO Balloon

The idea behind VirtIO Balloon [11] is to give back to the hypervisor the unused memory of a *Guest*. To do that, a communication between *Guest* and Hypervisor is required. The balloon can be inflated or deflated. Inflating the balloon means increasing the portion of *guest* memory that is given back to the Host and can no longer be used by the *guest* kernel. Deflating the balloon is the opposite operation. The bigger the balloon, the lower is the amount of memory that the *Guest* can use, but this means that the Host has regained more memory.

VirtIO Balloon can inflate/deflate the balloon in two different ways:

- *static*: the administrator manually resizes (by inflating or deflating the balloon) the *guest* memory through the QEMU Monitor.
- *automatic*: the HV and the *guest* kernel communicate to dynamically resize the *guest* memory depending on the *guest* and host needs.

VirtIO Balloon can be used to speed up live-migration. To do so in QEMU, the balloon can be inflated from the QEMU Monitor so that the memory that the *guest* kernel can use is smaller. Now the hypervisor can copy just the memory that the *guest* can actually use. The balloon can then be deflated once the migration is complete. Note that, for the entire duration of the migration process, the *guest* kernel cannot use the memory contained in the balloon.

### 3 Paravirtualized migration

In Linux all the memory is divided in pages. These pages can be either allocated for different purposes, or can be marked as free. The HV, because of the semantic gap, is unaware of the status of each page and, while performing a migration, must copy all of them. Our idea to reduce the amount of data transferred is to communicate to the HV the list of free pages, along with their physical addresses, during the first state of the migration, so that the HV can skip them during the first iteration of the second stage.

To provide communication between the Guest OS, the Hypervisor and the Host OS the mechanism described in [10] has been exploited. It involves the use of a virtual device, attached to QEMU, that provides a TCP socket for the host system, and a readable buffer for the device-driver. In this way, the host system, using the socket, is able to send a message to the device, while the HV and the guest OS can communicate using the device buffer.

On the other hand, to send back information from the guest to the host, the guest OS can write—with the help of the device driver—inside a buffer available in the device. Since the device is virtual all the information written can be then sent to the host via the socket.

One example of communication is when the *host* system wants to enable or disable the new migration mechanism. In this case, it just needs to send a message to the device so that it will store this information until the migration procedure starts.

If migration optimization is enabled, the following steps show the entire process of our mechanism:

1. The migration thread raises a specific IRQ on the virtual device in order to communicate to the device driver that a migration is occurring. It then waits until the device driver terminates its task.
2. The device driver handles the IRQ. It acquires each zone spinlock and communicates to the HV the physical addresses of the Free Blocks of Physical Pages (FBPP from now on) with their relative sizes, causing a VM Exit.
3. The virtual device can now wake up the migration thread and it waits until the migration's first stage is terminated.
4. The migration thread can now read the guest physical addresses of FBPP from the device buffer. Since the HV is a Host's process, it needs to translate the Guest Physical Addresses to Host Physical Addresses. Once the translation is performed, it can remove every FBPP from the list of dirty pages. Then it will wake up the thread in charge of the device's emulation, and it will start the migration's second stage.
5. The VM Exit is concluded, so the device driver can now release the spinlocks.

The Guest cannot allocate, nor deallocate memory during this stage, but it can do it freely during the other stages, and in particular while memory is being transferred, since any modifications to the Guest kernel's memory-allocation data structures will be caught by the normal dirty-pages tracking mechanism.

## 4 Experimental results

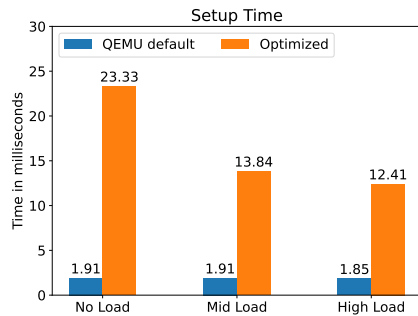
All the experiments were run on an Intel i7 8700K with 16 GiB DDR4 RAM. The Guest VM was assigned 4 virtual CPUs and 8 GiB of memory. Both the host and the guest were running v5.11.22 of the Linux kernel. To evaluate performance, we tested different scenarios varying the amount of free memory and the live-migration methodology: The improved live-migration mechanism has been compared against the current QEMU migration implementation with and without the use of VirtIO Balloon.

Our expectation in terms of performance was a decrease in the amount of transferred memory at the cost of a slight increase in setup time. The reduction in terms of transferred memory comes from the avoidance of sending the free memory pages from the VM in the source PM to the VM in the destination PM. On the other hand, this improvement does not come for free, since the list of the FBPP must be evaluated during the first live-migration stage, increasing the setup time. However, we expect that this is largely offset by the time saved from not sending a large portion of memory.

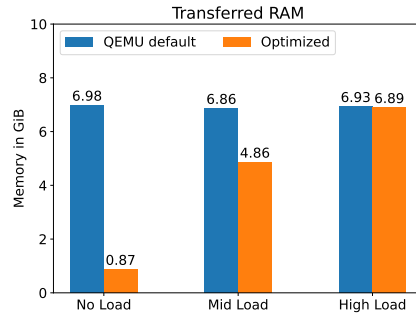
In the first testing session, we compared our optimized live-migration implementation with the QEMU native one.

Three scenarios with different memory loads have been tested. In order to simulate these scenarios, we used a program that used a fixed amount of memory and kept it allocated for the whole migration process.

- *No Load*: No memory from the program was allocated during the migration process.
- *Mid Load*: 4 GiB of memory from the program were allocated during the migration process.
- *High Load*: 6 GiB of memory from the program were allocated during the migration process.



**Fig. 3.** Setup time with and without the optimization in low mid and high load.



**Fig. 4.** Transferred RAM with and without the optimization in low mid and high load.

Figure 3 show that, in each scenario, we experienced a higher setup time compared to the standard QEMU migration, as expected. The setup time is about 25 milliseconds and tends to decrease when the load on memory increases. In fact, when a big amount of memory has already been allocated by the kernel, the number of FBPP is lower, so the driver of the emulated guest device that is running into the guest kernel finishes its job faster. Experiments also show that in the high load scenario, setup time is still not comparable to the one obtained by the QEMU default migration, since in this case there is still the overhead caused by synchronization between the migration thread and the guest device driver.

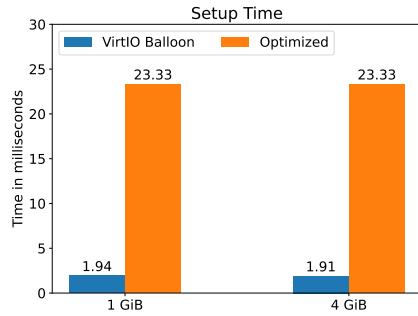
As for the amount of transferred memory, experiments in Figure 4 show an improvement in the no load and mid load scenario, while there is no difference in the high load one. In fact, in this last case, the amount of free memory is so low that there is no improvement at all since the hypervisor needs to copy the whole memory anyway. On the other hand, when there is a really high portion of free memory, as in no load scenario, the transferred memory reduction is quite significant.

Now let's compute the amount of time spared performing the live migration considering a 1 Gib/s transmission bitrate:

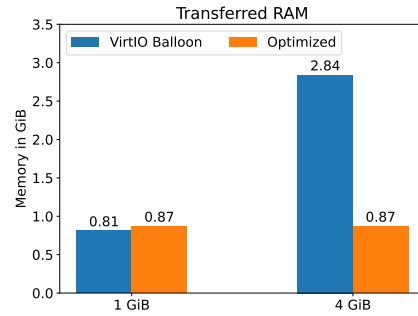
- *no load*: with the standard QEMU migration the hypervisor needs to transfer 6.98 GiB of memory, taking around 55,84 seconds. The optimized mechanism allows the hypervisor to just send 870.32 MiB of memory, taking just 6.96 seconds. The added setup time is so small compared to these numbers that it is negligible.
- *mid load*: with the standard QEMU migration the hypervisor needs to transfer 6.86 GiB of memory against the 4.86 GiB with the optimized mechanism. This leads to a spare of 16 seconds in terms of total migration time. Even in this case, the added setup time is so small that is still negligible.
- *full load*: this is the worst scenario, in fact the amount of transferred RAM is basically equal between the two migration mechanism, implying no improvement in terms of time saved.

Of course having an higher bitrate implies a less evident advantage in terms of time saved. In fact, while an higher bitrate allows the HV to copy the same amount of data in a shorter time, on the other hand it offers no setup time reduction. However, the transferred memory reduction given by this optimization for the no load and mid load scenario is still big enough to be advantageous even with a bitrate of 10 Gib/s.

The following results come from two simulations with different balloon sizes, statically set. In both cases, the guest had no load on memory, in order to maximize the transferred RAM reduction for both migration implementations.



**Fig. 5.** Setup time using the optimization and VirtIO Balloon in two VM setups (1 GiB and 4 GiB of RAM).



**Fig. 6.** Transferred RAM using the optimization and VirtIO Balloon in two VM setups (1 GiB and 4 GiB of RAM).

In these two simulations the Guest VM had 1 GiB and 4 GiB of memory respectively. The RAM was reduced by inflating the (VirtIO) balloon and thus given back to the HV.

Figure 5 shows that in each scenario we experienced a higher setup time compared to the standard QEMU migration with the use of VirtIO Balloon. This is expected and the motivation is the same as in the previous scenario.

As for the amount of transferred memory, Figure 6 shows that there is no significant improvement against the standard QEMU migration using VirtIO Balloon when the guest’s memory is shrunk to 1 GiB. In this scenario, the amount of memory left to the guest is basically comparable to the memory without the free pages. In other words the balloon reduced the amount of free pages, making our optimization less effective. However, when the balloon size is not as big as the free memory, like in the second scenario, our implementation performs better, allowing us to save around 15.76 seconds considering a bitrate of 1 Gib/s. Even in this case, the added setup time is so small that is still negligible.

## 5 Conclusions and future work

In conclusion, in this work we improved the QEMU live migration by not sending the guest’s free memory. This showed substantial performance gains against the standard migration, especially when the guest is not handling a high memory load. Future work should consider the use of eBPF to monitor, using probes, guest’s memory allocation. Doing this, we can reduce the setup time since we would not need to synchronize the guest’s device driver and the migration thread for the entire first migration stage. On the other hand, the guest kernel would be able to allocate memory even during the setup time, since the driver would not need to hold the *free\_areas* spin-locks for a long time.



## 6 Acknowledgments

Work partially supported by the Italian Ministry of Education and Research (MUR) in the framework of the FoReLab project (Departments of Excellence).

## References

1. Anjum, A., and Parveen, A.: A Dynamic Approach for Live Virtual machine Migration using OU Detection Algorithm. In: 2022 6th International Conference on Computing Methodologies and Communication (ICCMC), pp. 1092–1097 (2022). DOI: [10.1109/ICCMC53470.2022.9753974](https://doi.org/10.1109/ICCMC53470.2022.9753974)
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A.: Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.* 37(5), 164–177 (2003). DOI: [10.1145/1165389.945462](https://doi.org/10.1145/1165389.945462)
3. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX annual technical conference, FREENIX Track, p. 46 (2005)
4. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., and Warfield, A.: Live migration of virtual machines. In: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2, pp. 273–286 (2005)
5. Das, R., and Sidhanta, S.: LIMOCE: Live Migration of Containers in the Edge. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 606–609 (2021). DOI: [10.1109/CCGrid51090.2021.00070](https://doi.org/10.1109/CCGrid51090.2021.00070)
6. Garfinkel, T., Rosenblum, M., *et al.*: A virtual machine introspection based architecture for intrusion detection. In: *Ndss*, pp. 191–206 (2003)
7. Hines, M.R., and Gopalan, K.: Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pp. 51–60 (2009)
8. Ibrahim, K.Z., Hofmeyr, S., Iancu, C., and Roman, E.: Optimized Pre-Copy Live Migration for Memory Intensive Applications. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '11. Association for Computing Machinery, Seattle, Washington (2011). DOI: [10.1145/2063384.2063437](https://doi.org/10.1145/2063384.2063437)
9. Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A.: kvm: the Linux virtual machine monitor. In: Proceedings of the Linux symposium, pp. 225–230 (2007)
10. Leonardi, L., Lettieri, G., and Pellicci, G.: eBPF-based Extensible Paravirtualization. In: High Performance Computing. ISC High Performance 2022 International Workshops, pp. 383–393. Springer International Publishing (2022)
11. Liu, H., Jin, H., Liao, X., Deng, W., He, B., and Xu, C.-z.: Hotplug or Ballooning: A Comparative Study on Dynamic Memory Management Techniques for Virtual Machines. *IEEE Transactions on Parallel and Distributed Systems* 26(5), 1350–1363 (2015). DOI: [10.1109/TPDS.2014.2320915](https://doi.org/10.1109/TPDS.2014.2320915)
12. Ma, Y., Wang, H., Dong, J., Li, Y., and Cheng, S.: Me2: Efficient live migration of virtual machine with memory exploration and encoding. In: 2012 IEEE International Conference on Cluster Computing, pp. 610–613 (2012)

13. Stoyanov, R., and Kollingbaum, M.J.: Efficient Live Migration of Linux Containers. In: Yokota, R., Weiland, M., Shalf, J., and Alam, S. (eds.) High Performance Computing, pp. 184–193. Springer International Publishing, Cham (2018)
14. Svärd, P., Hudzia, B., Tordsson, J., and Elmroth, E.: Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. VEE '11, pp. 111–120. Association for Computing Machinery, Newport Beach, California, USA (2011). DOI: 10.1145/1952682.1952698
15. Wang, C., Hao, Z., Cui, L., Zhang, X., and Yun, X.: Introspection-based memory pruning for live VM migration. *International Journal of Parallel Programming* 45, 1298–1309 (2017)
16. Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M.: Black-Box and Gray-Box Strategies for Virtual Machine Migration. In: Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation. NSDI'07, p. 17. USENIX Association, Cambridge, MA (2007)
17. Xu, B., Wu, S., Xiao, J., Jin, H., Zhang, Y., Shi, G., Lin, T., Rao, J., Yi, L., and Jiang, J.: Sledge: Towards Efficient Live Migration of Docker Containers. In: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), pp. 321–328 (2020). DOI: 10.1109/CLOUD49709.2020.00052