virtio-mem: Paravirtualized Memory Hot(Un)Plug

David Hildenbrand
Martin Schulz
Technical University of Munich
Munich, Germany
{hildenbr,schulzm}@in.tum.de

Abstract

The ability to dynamically increase or reduce the amount of memory available to a virtual machine is getting increasingly important: as one example, cloud users want to dynamically adjust the memory assigned to their virtual machines to optimize costs. Traditional memory hot(un)plug, such as hot(un)plugging emulated DIMMs, and memory ballooning can dynamically resize virtual machine memory. However, existing approaches provide limited flexibility, are incompatible with important technologies like vNUMA and fast operating system reboots, or are unsuitable when hosting untrusted virtual machines.

To overcome these limitations, we introduce virtio-mem, a VIRTIO-based paravirtualized memory device, designed for fine-grained, NUMA-aware memory hot(un)plug in cloud environments. To showcase the adaptions needed in a hypervisor and a guest operating system to support virtio-mem, we describe our implementation in the QEMU/KVM hypervisor and Linux guests. We evaluate virtio-mem against traditional memory hot(un)plug and memory ballooning, showing that our approach enables assignment of memory in substantially smaller granularity per NUMA node than traditional memory hot(un)plug, such as 4 MiB on x86-64. In contrast to memory ballooning, virtio-mem is fully NUMA-aware and supports fast operating system reboots by design, while guaranteeing that malicious virtual machines, which try using more memory than agreed upon, can be detected reliably.

We conclude that using paravirtualized memory devices for dynamically resizing virtual machine memory significantly increases flexibility and usability compared to state-of-the-art. A first version of virtio-mem for x86-64 has been integrated into upstream Linux and QEMU.

VEE '21, April 16, 2021, Virtual, USA

ACM ISBN 978-1-4503-8394-3/21/04…$15.00
https://doi.org/10.1145/3453933.3454010

CCS Concepts: • **Software and its engineering** → **Virtual machines**; **Memory management**; • **Computer systems organization** → *Cloud computing*.

Keywords: Paravirtualization, Memory Resize, Memory Hotplug, Memory Hotunplug, Memory Ballooning

ACM Reference Format:
David Hildenbrand and Martin Schulz. 2021. virtio-mem: Paravirtualized Memory Hot(Un)Plug. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21), April 16, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453933.3454010

1 Introduction

Hardware virtualization is key to ensuring modern data centers' high efficiency and utilization, and is therefore one of the most important building blocks of modern cloud computing. Even with the advent of containers, traditional hardware virtualization remains important. For example, Kata Containers [1] implements *lightweight Virtual Machines (VMs)*, running containers inside VMs to benefit from better workload isolation and security [6].

Key to the success of hardware virtualization is the ability to dynamically adjust resource usage of VMs; this is especially true for memory, which represents both a tightly restricted, yet critically important resource. As one example, users want to dynamically increase or reduce the memory available to VMs based on needs to optimize costs and to adapt to changed workload requirements, avoiding downtimes [20, 43]. As another example, there is need to control the memory assignment of lightweight VMs dynamically to optimize memory utilization and to avoid waste of memory, as each application running in a container has different memory demands that can change over time.

Simply providing each VM the illusion of having access to a lot of physical memory is not an option: Operating Systems (OSs) like Linux will make use of all available memory for caches [28, 35]. Dynamically resizing the memory available to VMs requires a mechanism that can expose additional memory to a VM and that can remove previously exposed memory from a VM at runtime: coordination with the guest OS running inside the VM is mandatory.

Existing approaches either provide limited flexibility, are incompatible with important technologies, or are unsuitable

when hosting untrusted VMs. On one hand, traditional memory hot(un)plug, i.e., hot(un)plugging emulated Dual Inline Memory Modules (DIMMs) or reassigning parts of physical DIMMs between logical partitions, relies on architecture-specific properties and has granularity restrictions, resulting in limited flexibility and reduced usability. On the other hand, memory ballooning, i.e., controlling a VM's memory footprint by relocating pages between the VM and its hypervisor, either has to tolerate that even well-behaving VMs might exceed memory limits and reuse inflated memory, making it impossible to detect malicious VMs, or has to enforce memory limits and disallow access to inflated memory, which results in problematic reboot handling.

Memory ballooning is incompatible with important technologies like vNUMA and fast OS reboots, especially when running untrusted VMs. vNUMA exposes a virtual Non-Uniform Memory Architecture (NUMA) topology to a VM and is required to support VMs without performance degradation if the size of a VM could exceed the size of a single NUMA node during its lifetime [36]. Fast OS reboots, i.e., using kexec under Linux to boot a new OS directly by bypassing firmware and bootloaders, are a crucial mechanism for reducing reboot times, resulting in increased system availability, for example, on OS updates [9, 34].

To overcome these limitations, we introduce *virtio-mem*, a novel approach for memory hot(un)plug in virtualized environments based on Paravirtualized Memory Devices (PMDs), allowing for *fine-grained, NUMA-aware assignment of memory suitable for cloud environments*. virtio-mem is flexible, enables fine-grained memory control per NUMA node, is architecture agnostic, and can be integrated into any hypervisor that supports VIRTIO, a standard for virtual I/O devices. Novel aspects and core design points of virtio-mem in comparison to existing work include:

- Providing a flexible amount of memory via separate PMDs in a VM, for example, having one device per virtual NUMA node.
- Using device-local target size requests to control the size provided via PMDs in a VM.
- Never exposing PMD-provided memory via firmware-provided memory maps to the guest OS.

virtio-mem is architecture-independent and focuses on memory hot(un)plug capabilities in cloud environments, such as dynamically managing metadata for hot(un)plugged memory in the guest OS and allowing for reliable detection of malicious VMs in the hypervisor. In particular, we make the following contributions:

- We give a detailed description of traditional memory hot(un)plug and state of the art memory ballooning (Section 2).
- We compare traditional memory hot(un)plug with memory ballooning and provide the motivation for a new memory hot(un)plug mechanism in cloud environments (Section 3).
- We introduce virtio-mem, a VIRTIO-based PMD, designed for fine-grained, NUMA-aware dynamic assignment of memory in cloud environments (Section 4).
- We describe our implementation of virtio-mem in the open-source hypervisor QEMU/KVM [8, 22] and Linux guests, showcasing the adaptions needed in a hypervisor and a guest OS to support virtio-mem (Section 5).
- We provide an extensive evaluation of virtio-mem under QEMU/KVM with Linux guests (Section 6), comparing it against traditional memory hot(un)plug and memory ballooning.

Our evaluation of virtio-mem shows that many limitations known from previous work do not apply, allowing for dynamic assignment of memory in substantially smaller granularity than traditional memory hot(un)plug per NUMA node, such as 4 MiB on x86-64. Also, specifying a target size when running virtio-mem and not managing individual hot(un)pluggable units, such as DIMMs, fundamentally increases flexibility and usability. In contrast to memory ballooning, virtio-mem naturally supports fast OS reboots and is NUMA-aware by design, while allowing the detection of malicious VMs reliably. As virtio-mem operates on small, flexible granularity, it can hotunplug significantly more memory from a VM compared to traditional memory hot(un)plug when unmovable allocations might be placed on hotplugged memory by a guest OS, thereby preventing hotunplug.

2 Background and Related Work

Historically, hot(un)plugging physical memory has two applications: memory modules that are failing can be removed or replaced without downtime, and the capacity of a system can be upgraded at runtime [38]. Nowadays, though, the same hot(un)plugging techniques can also be applied without physical changes in the system configuration, effectively up- or downgrading the capacity of a system, e.g., to save energy when the full memory capacity is not required [31]. This is especially applicable to virtualized environments, where hardware virtualization abstracts physical resources to virtual resources, allowing dynamic adjustment of the memory resource usage of a VM by hot(un)plug of virtual memory resources [38].

There are different approaches to dynamically resize VM memory. In this section, we explain traditional memory hot(un)plug, including section-based memory hot(un)plug in Linux, discuss memory ballooning and introduce the concept of Paravirtualized Memory Devices (PMDs).

Although there are approaches for providing VMs access to extended memory in the hypervisor via Transcendent Memory (TMEM) [27, 28], shared memory (e.g., via MemExpand [46]), *Discontiguous Saved Segments* [35] under z/VM

and VM swap storage backed by RAM in the hypervisor [13], such memory is not universal system RAM for the guest OS.

2.1 Traditional Memory Hot(Un)Plug

The traditional way to hot(un)plug memory in virtualized environments is by reusing existing mechanisms introduced for physical machines: physically plugging and unplugging DIMMs, usually with a granularity of a few gigabytes. The maximum number of hardware slots for such modules is limited. Hypervisors can emulate DIMM devices for their VMs. For example, QEMU on x86-64 supports up to 256 DIMMs per VM, emulated as separate `pc-dimm` devices [3].

Logical Partitioning. Logical partitioning hypervisors extended architecture-specific interfaces to re-assign parts of physical DIMMs between partitions; the primary difference to DIMMs is smaller granularity. While providing more flexibility, there are still granularity restrictions and the firmware-provided memory map, such as the device tree on POWER [32], has to support exposing all individual hot(un)pluggable units to partitions: having many individual units is problematic.

IBM pSeries DLPAR [19] with its *dynamic reconfiguration of memory* can hot(un)plug memory in *LMBs (Logical Memory Blocks)* as small as 16 MiB to logical partitions, however, the minimum granularity depends on the physical memory size of the server [18] and is usually 256 MiB. For example, QEMU on POWER uses this interface to implement hot(un)plug of up to 32 emulated `pc-dimm` devices.

On IBM z Systems with *Dynamic Storage Reconfiguration* [23], logical partitions and VMs under z/VM can request to use initially defined *reserved storage* in increments of at least 1 MiB; however, the increment size depends on the maximum size of the VM and is usually much larger.

Section-based Memory Hot(Un)Plug in Linux. Memory to be exposed to the page allocator in Linux can only be added and removed in *Linux memory block* granularity, which corresponds to at least one *memory section*. In this *SPARSEMEM* [38] memory model, metadata is allocated and managed per section. For example, the section size in upstream Linux [44] on x86-64 is 128 MiB and on aarch64 it is 1 GiB. The Linux memory block size, and thereby the section size, defines the minimum naturally supported hot(un)pluggable unit.

Memory hot(un)plug under Linux is a two-step process. During memory hotplug, memory is first added to Linux, whereby metadata for the new memory is allocated ("struct page" for each page [15]) and page tables for the identity mapping are created. Afterwards, the added memory is *onlined*, whereby the new pages are exposed to the page allocator, from which point on they can be used by the system.

In case of memory hotunplug, the memory first has to be *offlined*, whereby all pages are hidden from the page allocator again. This will fail if unmovable allocations, such as page tables, ended up on the memory to unplug, because the memory cannot get freed up. Finally, the metadata and the identity mapping are removed from Linux.

The Linux page allocator provides different zones, describing which allocations can use which memory. When onlining memory, a zone can be specified. Memory in the `NORMAL` zone can be used for most allocations. Memory in the `MOVABLE` zone can only be used for movable allocations. While it seems desirable to add all hotplugged memory to the `MOVABLE` zone, this is, in general, impossible; many allocations in the kernel are unmovable. Further, having too much `MOVABLE` memory can result in *zone imbalances*: the kernel will run out of memory for unmovable allocations and might crash, although there is still free memory available in the `MOVABLE` zone. As an additional consequence, this could cause memory in the `NORMAL` zone to not be hotunpluggable anymore.

Safe ratios depend on the workload; especially setups where a lot of memory might get hotplugged later require hotplugged memory to be onlined to the `NORMAL` zone, which restricts the amount of memory that can get hotunplugged later.

2.2 Memory Ballooning

Memory ballooning is a simple mechanism to relocate physical memory between a VM and its hypervisor, dynamically adjusting the memory footprint of a VM. Memory ballooning was originally introduced in VMware ESX to optimize memory overcommit and avoid swapping in the hypervisor by reclaiming memory from VMs [45]; however, it can more generally be used to dynamically resize VM memory.

A *balloon driver* in the guest OS communicates with the hypervisor to inflate or deflate the balloon, for example, via a *balloon device*; all memory inflated in the balloon is no longer available to the VM. To inflate the balloon, the balloon driver in the guest allocates memory and notifies the hypervisor about allocated memory ranges. To deflate the balloon, the balloon driver frees previously allocated memory to the guest OS again, usually after coordination with the hypervisor.

Most hypervisors support memory ballooning, including XEN [7, 26], VirtualBox [33], Hyper-V with Dynamic Memory [30], IBM PowerVM with Active Memory Sharing [10], and VIRTIO-based hypervisors via *virtio-balloon* [37].

Balloon Inflation and Deflation. In the context of improving memory overcommit in the hypervisor, speed and simplicity are important. As one example, reusing the basic page allocator from the guest OS allows for supporting many OSs easily and automatically triggers memory reclaim mechanisms inside the guest OS on demand [45].

Traditionally, balloon inflation/deflation works with base pages, such as 4 KiB. Using huge pages, such as 2 MiB, instead can improve VM performance and ballooning speed [17]; in

Linux, the VMware balloon and the Hyper-V balloon support huge pages, falling back to base pages when necessary [44]. HUB [17] implements huge page ballooning for virtio-balloon, however, it is not supported upstream.

Request Handling. The requested balloon size is usually determined by the hypervisor and communicated via a *target balloon size* to the guest. xenballoond [26] implements dynamic *self-ballooning* for XEN, whereby the guest will dynamically adjust the size of the balloon itself. Most balloon drivers deflate the balloon without hypervisor requests in some scenarios; examples in Linux include unloading the balloon driver or memory pressure [44].

Request handling in the Hyper-V balloon works slightly differently: the hypervisor sends explicit requests [44]. While the balloon driver is free to chose which memory to inflate, deflation requests specify explicit memory ranges: the balloon driver cannot optimize memory layout when deflating.

Growing Beyond Initial VM Size. Traditional memory ballooning does not support increasing the size of a VM beyond its initial size; the hypervisor has to expose additional memory to the VM, which the balloon driver then exposes to the guest OS. The basic idea to extend memory ballooning with section-based memory hot(un)plug capabilities was first envisioned by Schopp et al. [39], with a focus on Linux guests. XEN was the first hypervisor to add memory hotplug capabilities to its memory ballooning mechanism; when required, the Linux driver selects new memory sections to hotplug [21]. Work by Smith et al. [40] proposed optimizations to make memory hotplug under XEN more responsive. U-tube [25] extended the XEN balloon by fine-grained memory hot(un)plug in base page granularity.

The Hyper-V balloon also supports hotplugging additional memory. Similar to deflation requests, the hypervisor requests to hot-add specific memory ranges [41].

System Reboots. Rebooting with an inflated balloon is problematic: either, the physical memory layout of the VM has to be adjusted to reflect the new VM size, or the balloon driver has to inflate the balloon while booting up.

Changing the physical memory layout of a VM when rebooting, or when shutting down to start again later, is often problematic and undesired in hypervisors. As one example, in our experiments we identified that Hyper-V performs such layout changes, however, it also disables the memory balloon completely if the guest could hibernate, as both mechanisms are incompatible [11]. As another example, live migration support in QEMU relies on the fact that the VM memory layout on the migration source and the migration target matches [3].

Requiring the balloon driver to inflate while booting up is not suitable in cloud environments: if the balloon driver is not available, fails, or inflation is delayed, even well-behaving guests might consume more memory than desired. XEN uses a technique called "populate-on-demand" whereby it will detect VMs that consume more memory than intended and crashes them [12].

Fast OS Reboots. Fast OS reboots, whereby firmware and bootloaders are bypassed, are a fundamental problem for memory ballooning. For example, during kexec under Linux, the old OS creates the initial memory map for the new OS; the original memory map is forwarded. As balloon inflation is not restricted to specific guest physical memory regions, balloon inflation can fragment the original memory map heavily, something which cannot be reflected in architecture-specific memory maps.

In our experiments, Hyper-V VMs crashed reliably when trying to use kexec under Linux for fast OS reboots with an inflated balloon. Other memory ballooning mechanisms either have to temporarily deflate the whole balloon or allow access to inflated memory, which is undesired in cloud environments.

NUMA-awareness. Supported ballooning mechanism are not NUMA-aware; inflation requests neither target specific NUMA nodes nor guest physical memory ranges. vNUMA-mgr [36] is the only work to discuss *NUMA-aware ballooning* for XEN, however, it also requires the guest OS to strictly obey a given (virtual) NUMA configuration: something that cannot be expected in cloud environments. If the guest OS is not NUMA-aware, the basic page allocator cannot handle allocation requests on specific NUMA nodes; even well-behaving guests might not be able to handle requests or even mishandle them. While range-based memory ballooning is imaginable, it conflicts with fast inflation. Having VMs inflating or deflating on different nodes than requested can result in serious performance issues also affecting unrelated VMs.

2.3 Paravirtualized Memory Devices

In order to provide new approaches to dynamically resize VM memory, we need the ability to expose a dynamic amount of memory to VMs; Paravirtualized Memory Devices (PMDs) provide us with the right level of abstraction for this.

Similar to memory devices found in physical environments, such as DIMMs, a PMD provides and manages a region in physical address space; however, such devices are only found in VMs and the memory semantics can conceptually differ. The only PMD we identified is *virtio-pmem* [14], an emulated Non-Volatile DIMM (NVDIMM) with a paravirtualized flushing interface; it is based on VIRTIO.

VIRTIO. VIRTIO is a standard for virtual I/O devices. Important building blocks of VIRTIO include the configuration space, to expose the device configuration to the driver and let the driver configure the device, and *virtqueues*, to let the device and the driver communicate by posting buffers to a queue [37].

3 Traditional Memory Hot(Un)plug vs. Memory Ballooning

In this section we compare traditional memory hot(un)plug (see Section 2.1) with memory ballooning (see Section 2.2) along seven criteria (C1-C7) and establish the need and the requirements for a new memory hot(un)plug mechanism.

C1: Use Case. The primary use case of traditional memory hot(un)plug in the context of VMs is to dynamically adjust the maximum memory available to a VM; the speed of hot(un)plug, however, is less important. This is in contrast to the traditional use case of memory ballooning, the efficient handling of memory overcommitment in the hypervisor: fast inflation and reliably triggering memory reclaim in the guest OS are important there [45].

C2: Granularity. Memory ballooning relies on the basic page allocator in the guest OS. The theoretical maximum granularity, therefore, is defined by that page allocator and its limits, such as 4 MiB on x86-64 Linux; some implementations support huge pages. In contrast, traditional memory hot(un)plug uses much larger granularity, typically hundreds of megabytes or gigabytes. The minimum granularity, however, also depends on the guest OS; x86-64 Linux supports smallest hot(un)pluggable units between 128 MiB and 2 GiB [44].

C3: Memory Fragmentation. Balloon inflation with base pages can heavily fragment guest physical memory, resulting in performance degradation when failing to use Transparent Huge Page (THP) in the guest or the hypervisor; huge page ballooning mitigates this issue [17]. Under Linux, some memory ballooning implementations support balloon compaction [4] of inflated base pages, allowing for defragmentation. No memory ballooning implementation in Linux removes complete Linux memory blocks to free up metadata. In comparison, Linux will free up metadata when unplugging DIMMs [44].

C4: NUMA-awareness. Traditional memory hot(un)plug naturally support NUMA, for example, by exposing the relationship between nodes and DIMMs via ACPI; no memory ballooning mechanism supports vNUMA. While NUMA-aware memory ballooning for VMs that respect the virtual NUMA topology has been proposed, especially untrusted VMs that might use a different topology are problematic and necessary handling would contradict with the basic idea of fast inflation and simplicity.

C5: Resize Capability. Basic memory ballooning cannot grow a VM beyond its initial size; extensions as implemented by XEN or Hyper-V are required. Otherwise, to grow a VM later, the VM has to be started with more memory than desired, and the balloon has to be inflated while booting up the guest OS. Traditional memory hot(un)plug usually only supports hotunplug of units that were previously hotplugged.

C6: Resize Request Handling. The size of the memory balloon is controlled via resize requests, which gives the guest OS a lot of flexibility to select fitting memory ranges. One exception is the Hyper-V balloon: the hypervisor selects memory ranges to deflate and hot-add. Traditional memory hot(un)plug usually relies on the hypervisor specifying units to hot(un)plug. One exception is IBM pSeries DLPAR [19] with an interface to request hotunplug of a given number of LMBs, independent of a NUMA node [44]. Requests to hot(un)plug DIMMs can often fail silently.

C7: (Re)boot Handling. With traditional memory hot(un)plug, the guest OS will only use currently plugged memory during boot, during reboots and during fast OS reboots. In contrast, fast OS reboots impose a fundamental problem for memory ballooning when hypervisors want to guarantee that a well-behaving VM cannot exceed the configured memory size. Requiring to inflate the balloon when (re)booting is infeasible when running untrusted VMs; the alternative of reorganizing the physical memory layout of a VM is impracticable and undesired.

Summary. The main limitations of memory ballooning are (re)booting with an inflated balloon (C7), missing NUMA-awareness (C4), that metadata for inflated memory might be wasted in the guest OS, and that physical memory can get fragmented heavily (C3). On the upside, clear advantages are that basically any architecture and any guest OS can be easily supported by relying on the basic page allocator (C2).

Traditional memory hot(un)plug is architecture-specific, has strong granularity limitations and gives the guest OS little flexibility (C2). Another important limitation is hotunplug request handling (C6): trying hotunplug of random DIMMs until it eventually succeeds is highly impracticable. However, traditional memory hot(un)plug is NUMA-aware (C4), only fragments memory in bigger granularity (C3), can free up metadata of hotunplugged memory in the guest OS, and naturally supports different kinds of reboots (C7).

4 Our Approach: virtio-mem

Overcoming the limitations of traditional memory hot(un)plug and memory ballooning, therefore, requires a novel approach. In this section we give an overview of the architecture of virtio-mem, explaining its memory hot(un)plug process, and we describe how the communication between the hypervisor and the guest OS is architected via VIRTIO.

4.1 Overview of the Architecture of virtio-mem

On a high level, separate virtio-mem devices expose a dynamic amount of memory to the VM, for example, having one device per virtual NUMA node. Conceptually, each device looks like a large DIMM that is partitioned into *fixed-sized blocks* that can either be in the state *plugged* or *unplugged*. The hypervisor and the guest OS communicate via the device

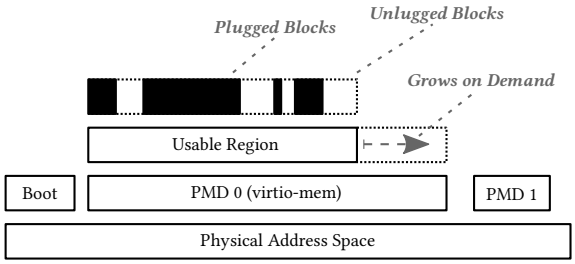

Figure 1. Overview of a virtio-mem device in guest physical address space.

to coordinate hot(un)plug of blocks guided by *resize request* from the hypervisor: it is up to the guest OS to select device blocks to hot(un)plug when processing a resize request.

Each virtio-mem device exposes the location of its device-managed region in guest physical address space via device properties. The firmware-provided memory map never exposes memory provided by virtio-mem to the guest OS; especially not when booting, during reboots, and during fast OS reboots. Instead, the device driver in the guest OS handles detection and communication with virtio-mem devices.

The current amount of *plugged device memory* is available via a device property — the *plugged size* — which is updated after successful (un)plugging of blocks. The hypervisor indicates the *requested size* via a device property; property changes correspond to resize requests. The guest OS can determine whether it should plug or unplug blocks by comparing the plugged size and the requested size.

To enable optimizations in the hypervisor, only some part of the device-managed memory region might be available for plugging blocks: the *usable region*, also exposed via a device property. The hypervisor can grow the usable region on demand, e.g., when increasing the requested size. Figure 1 visualizes the guest physical address space layout with two PMDs, whereby one of them is a virtio-mem device.

The hypervisor determines the *block size* based on various factors, such as the THP size under Linux, and exposes it to the guest via a device property. Plugged blocks behave just like ordinary system RAM. Access to unplugged blocks is undefined, however, unplugged blocks in the usable region can be read to simplify creation of memory dumps.

Each virtio-mem device can be assigned to a NUMA node; in case of ACPI, a device property exposes a *PXM (Proximity Domain)* to the VM, indicating the node. As multiple virtio-mem devices per VM are possible, requesting to resize a virtio-mem device can effectively trigger resizing VM memory residing on a specific NUMA node in the hypervisor.

4.2 Memory Hot(Un)Plug Process

When the hypervisor wants to resize a VM, it selects a configured virtio-mem device, for example, based on a NUMA preference, and modifies the requested size; this results in the device notifying the guest about a configuration change.

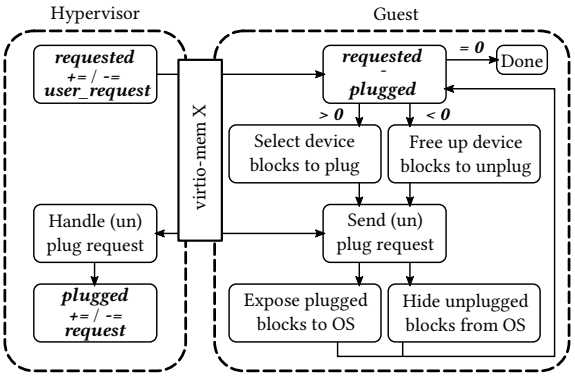

Figure 2. Overview of the device-local hot(un)plug process when the hypervisor instructs a specific virtio-mem device X to resize by adjusting the requested size, for example, because of a user request to either grow or shrink. For simplicity, error handling and rejected device requests are not visualized.

It is up to the device driver in the guest OS to reach the requested size by selecting blocks to (un)plug and coordinating with the device. When requested to add memory, the driver will select unplugged blocks belonging to the device to plug; it will ask the device to plug them and expose them to the guest OS. When requested to remove memory, the driver will find plugged blocks belonging to the device it can free up; it will then ask the device to unplug the blocks and hide them from the guest OS. A device does not allow plugging of new blocks if doing so would result in the plugged size exceeding the requested size.

Usually, all plugged blocks are set to be unplugged by the hypervisor when the VM reboots and the device driver is responsible for reaching the requested size by selecting and plugging blocks; additionally, guests can trigger the unplugging of all plugged blocks to start with a clean slate, for example, during fast OS reboots with plugged blocks.

Figure 2 visualizes the device-local hot(un)plug process when the hypervisor requests a specific virtio-mem device to resize by changing the requested size of the device.

4.3 Communication Between Hypervisor (Device) and Guest OS (Driver) via VIRTIO

We model virtio-mem as a VIRTIO-based Paravirtualized Memory Device (PMD) (see Section 2.3), because VIRTIO allows for easy creation of an architecture-independent mechanism to expose a special memory region in guest physical address space; the VIRTIO communication mechanism enables the hypervisor to communicate with the guest OS via the device, e.g., for the hypervisor to send resize requests and the guest OS to indicate memory blocks within the device-managed region to hot(un)plug.

All device properties are exposed to the driver in the guest OS via the VIRTIO configuration space. The device notifies the driver about a configuration update whenever device properties, like the requested size, change.

The driver communicates with the device via a VIRTIO queue; it sends requests and waits for a response from the device before continuing. Currently supported requests include plugging and unplugging one or multiple contiguous blocks, and sensing the state (`plugged`, `unplugged` or `mixed`) of one or multiple contiguous blocks. The device can either accept or reject a request, indicating the reason. A reject is possible if the device is busy (e.g., if the hypervisor cannot handle (un)plugging of blocks while migrating a VM) or if plugging new blocks would exceed the requested size.

5 Implementation of virtio-mem in QEMU/KVM and Linux

To showcase the adaptions needed in a hypervisor and guest OS to support virtio-mem we extended the QEMU/KVM hypervisor and implemented a Linux guest driver for virtio-mem. Integrating virtio-mem into QEMU (~1400 LOC) and Linux (~2200 LOC) required a significant number of core extensions, refactorings and fixes in both components; no KVM changes were required. In this section we give an overview of our implementation, which has already been successfully upstreamed — excluding some QEMU optimizations discussed in this paper — and is available to the wider community.

5.1 QEMU Implementation

We implemented virtio-mem in QEMU by building upon the infrastructure introduced for `virtio-pmem` devices (see Section 2.3) to map VIRTIO-based PMDs into guest physical address space.

The `virtio-mem` device implements the core logic and manages the memory region; a `virtio-mem-pci` device encloses it, exposes it to the VM and realizes the pci-specific VIRTIO transport mechanism. The `virtio-mem-pci` device implements the `TYPE_MEMORY_DEVICE` interface, making it look internally like a memory device comparable to a DIMM.

Similar to QEMU's `pc-dimm` implementation, a user has to specify a *memory backend*, which describes memory properties, such as the size, and an optional NUMA node.

Device Memory Region. Internally, a memory backend corresponds to a memory region in QEMU's virtual address space; for example, specifying a `memory-backend-ram` with 64 GiB results in QEMU mapping a 64 GiB region of anonymous memory into its virtual address space using `mmap`.

When adding a `virtio-mem-pci` device to a VM, QEMU processes the `TYPE_MEMORY_DEVICE` interface, allocating a free area in guest physical address space with the memory region size and mapping the memory region into guest physical address space. In case of KVM, this involves registering a KVM memory slot in the kernel, which describes the relation between the new guest physical address range and the memory region in QEMU's virtual address space.

Request Handling and Device Blocks. Whenever the device is notified about a new request in the request queue, it processes the request and responds. We use a bitmap to remember the state (plugged/unplugged) of each block: one bit per block. We determine the device block size based on the THP size, corresponding to 2 MiB on x86-64. When processing requests, the bitmap is queried and modified on success, accordingly. Initially, all blocks are unplugged. Just as most VIRTIO request in QEMU, requests are handled by a single thread, avoiding the need for manual locking.

When unplugging blocks, we use the same mechanism as virtio-balloon to discard physical pages used for backing virtual memory locations; e.g., for anonymous memory this is achieved using the `madvise` system call with `MADV_DONTNEED`; discarded pages will be freed by Linux, resulting in the virtual memory location no longer consuming actual physical memory. Once accessed, Linux either populates fresh physical pages or uses the shared zeropage if possible.

(Un)plug requests are rejected while the VM is getting migrated; as one example, discarding pages is problematic during postcopy live migration [16] in QEMU.

User Interface. With its *QEMU monitor*, QEMU provides a way to communicate with a running VM instance. We reuse the commands `qom-get` and `qom-set` to observe the plugged size and modify the requested size of a specific `virtio-mem-pci` device at runtime.

System Reset Handling. On QEMU system resets, all device-managed memory is discarded and the bitmap is cleared, resulting in all device-managed memory being unplugged after a reboot.

Resizeable Allocations. We implemented one optimization to avoid large, accessible memory regions in QEMU when the requested size of a device is initially substantially smaller than its maximum size: the resizeable usable region specified for virtio-mem devices (see Section 4.1). We adjust the usable region size and the actual accessible memory mapping size on demand.

As one example, Linux's memory overcommit handling might deny large memory mappings that are completely accessible. We expect similar optimizations to be relevant on other type-2 hypervisors with limited control over memory management. We apply this optimization to anonymous memory only for now, as this is the main use case.

QEMU already implements resizeable memory regions, however, they are only used internally and do not use resizeable allocations; there is still a large accessible memory region in QEMU's virtual address space. We allow specifying that a memory backend should create a resizeable RAM memory region, which starts up empty, and the owning device triggers resizing when required. In our case, a `virtio-mem` device grows the memory region when the requested size

requires it. For now, the area is only shrunk during reboots, as shrinking requires coordination with the guest.

We implement resizeable allocations via smart `mmap` handling. We reserve a large, inaccessible area via `mmap` and map the first part corresponding to the current memory region size accessible. When resizing, the accessible part is adjusted using `mmap` calls. Linux is able to merge adjacent, compatible mappings automatically to minimize individual mappings.

5.2 Linux Driver Implementation

We implement virtio-mem in Linux by extending section-based memory hot(un)plug (see Section 2.1), adding/removing separate Linux memory blocks, but dynamically exposing memory within these blocks in smaller, *subblock* granularity.

Basic Building Blocks. We reuse existing functionality to add/remove Linux memory blocks, to control which pages to expose to the page allocator when onlining memory blocks, to mark pages as logically offline, and to try to allocate/free physical memory ranges using the Linux range allocator. However, we implement support for adding memory that should be ignored by kexec (e.g., not adding it to the memory map of the new kernel), for offlining memory blocks from device drivers, and for offlining memory that still contain allocations, however, relevant pages are marked logically offline by a driver.

Driver Initialization. Driver initialization consists of generic virtio initialization, preparing the virtqueue and querying the device configuration. We also calculate the subblock size: it determines in which granularity we can logically hot(un)plug memory and spans at least one device block. The subblock size is the maximum of the `virtio-mem` device block size (e.g., 2 MiB in our QEMU implementation) and the maximum page allocator granularity in Linux (e.g., 4 MiB on x86-64). The latter represents a limitation of the Linux range allocator; extending it to support smaller granularity is left for future work.

We use a *workqueue* to process hot(un)plug requests asynchronously. Driver initialization finishes by queuing work to the workqueue, to process any initial resize request.

Tracking the State of Linux Memory Blocks. We partition the device-managed memory region into properly aligned Linux memory blocks and remember for each block the current state, e.g., whether the block is currently added to Linux or not. In addition, we store the state of each subblock in a bitmap — if the corresponding device blocks are currently either plugged or unplugged in the device. We dynamically grow tracking data on demand. We use a mutex to protect our data structures.

Request Handling. Whenever we receive a configuration update notification from the device, we queue work to the workqueue. When processing work in the workqueue, we determine whether to hotplug or hotunplug subblocks by comparing the current requested size with the plugged size. In case we fail to fully process a request, we setup a timer to retry again later.

Plugging Subblocks. To plug memory, we first traverse all added Linux memory blocks in *increasing address order*, searching for unplugged subblocks; once successful, we request to plug the corresponding device blocks of a subblock via the device and expose them to the page allocator.

Once all added Linux memory blocks are fully plugged, we have to add additional Linux memory blocks; for each new Linux memory block, we first request to plug the corresponding device blocks of the subblocks we want to plug via the device, followed by adding that new block to Linux.

Adding Linux Memory Blocks. To add memory blocks we use a variant of `add_memory()` we introduced to the Linux kernel, instructing kexec to not expose it to the new kernel during a fast OS reboot. We intercept onlining of memory blocks to control which pages to actually expose to the page allocator. In case a subblock is not plugged, we do not expose it to the page allocator and instead mark the corresponding pages logically offline.

Unplugging Subblocks. To unplug memory, we traverse all added Linux memory blocks in *decreasing address order*, trying to allocate plugged subblocks within a block in decreasing address order using the Linux range allocator, i.e., `alloc_contig_range()`. Once we succeed, we request to unplug the corresponding device blocks via the device; we then mark the allocated pages logically offline. Whenever we process a fully plugged Linux memory block and the unplug request permits it, we try allocating the whole Linux memory block first, before falling back to individual subblocks.

We note that this allocation scheme is slower than scanning for free subblocks first, as we might end up migrating allocated memory possibly to some other device memory location that we might try allocating to unplug later. However, for now we consider removing whole Linux memory blocks more important than hotunplug speed; we leave optimizing the unplug process for future work.

Removing Linux Memory Blocks. Once all subblocks of a Linux memory block are unplugged by virtio-mem, we trigger offlining and removal from Linux. We extended memory offlining code in Linux to skip pages that are marked logically offline by a driver, but are still looking like ordinarily allocated pages.

Optimizing Adding Linux Memory Blocks. Our implementation adds one Linux memory block at a time, which has a negative side effect when using `ZONE_NORMAL`. Onlining a Linux memory block places all newly onlined pages to the head of the page allocator's free page lists. When adding and onlining successive Linux memory blocks, this means that metadata for new blocks will be allocated immediately from

one of the just added blocks, creating a dependency chain of unmovable allocations: if one Linux memory block cannot be offlined and removed, all dependent ones also cannot be offlined and removed. To compensate, we modify memory onlining such that onlined pages are instead placed to the tail of the free page lists.

6 Evaluation

In this section we evaluate virtio-mem using our QEMU/-KVM and Linux implementation on x86-64, comparing it with traditional memory hot(un)plug, via emulated DIMMs, and memory ballooning, via virtio-balloon.

6.1 Environment

All experiments were performed on a 12-core AMD Ryzen 9 3900X CPU with 3.8 GHz and 64 GiB of DDR4. We used exactly one VM with 10 VCPUs and a differing amount of memory per experiment.

In the hypervisor, we ran Fedora 33 with the supplied Linux kernel 5.9.13-200.fc33.x86_64 and a modified[1] QEMU v5.2.0 [3], compiled with "gcc (GCC) 10.2.1 20201125 (Red Hat 10.2.1-9)". Inside the VM, we ran CentOS 8.2.2004 with a Linux kernel v5.10 [44]. The kernel config used was based on config-4.18.0-193.6.3.el8_2.x86_64 supplied by CentOS. We used "gcc (GCC) 8.3.1 20191121 (Red Hat 8.3.1-5)" to compile the Linux kernel. Transparent Huge Pages (THPs) were enabled in both Linux instances. SMT and AMD Turbo CORE were enabled.

6.2 Experiments and Results

As a general note, experiments that use the `MOVABLE` zone in the Linux guest reliably show low variability; we, therefore, only report results of a single execution of the experiment, except for experiments using the `NORMAL` zone. In our experiments we consider `MOVABLE:NORMAL` ratios up to 4:1 safe.

6.2.1 NUMA-Aware Resizing. Because memory ballooning, i.e., virtio-balloon, is not NUMA-aware, we first provide a separate experiment to evaluate to what degree virtio-mem allows for fine-grained, NUMA-aware resizing of VMs. In general, virtio-mem and traditional memory hot(un)plug behave in NUMA configurations just like they do in Uniform Memory Architecture (UMA) configurations; we, therefore, restrict all further experiments to a single virtual NUMA node.

Setup. We started a 8 GiB VM with two NUMA nodes (4 GiB on each node), and two virtio-mem devices with a maximum size of 16 GiB each (one for each node). We configured memory onlining in the guest to automatically online all added Linux memory blocks to the `MOVABLE` zone. We adjusted the size of the nodes to differing sizes by requesting to resize the respective virtio-mem devices in a sequence of steps. We monitored the total VM size in QEMU[2] and the total memory size in Linux[3] after each step.

Table 1. Resizing a 8 GiB VM with two NUMA nodes (4 GiB on each node) using virtio-mem. For each entry, the first part corresponds to Node 0, the second part to Node 1.

Step	Requested (MiB)	QEMU (MiB)	Linux (MiB)
Start	4096 / 4096	4096 / 4096	3940 / 4030
1	20480 / 12288	20480 / 12288	20324 / 12222
2	12288 / 20480	12288 / 20480	12132 / 20414
3	12288 / 12288	12288 / 12288	12132 / 12222
4	4596 / 4596	4596 / 4596	4440 / 4530
5	4598 / 4594	4596 / 4596	4440 / 4530
6	4600 / 4592	4600 / 4592	4444 / 4526
7	4096 / 4096	4096 / 4096	3940 / 4030

Results. Table 1 shows the result of this experiment. We observe that virtio-mem allows for fine-grained, NUMA-aware resizing of VM memory. The difference between the values monitored via Linux and QEMU is constant (156 MiB on node 0, 66 MiB on node 1), and mainly because of memory accounting in Linux during boot. As our Linux implementation can (un)plug memory in 4 MiB granularity, requests not aligned to 4 MiB as in Step 5 cannot be fully processed. The achieved results are neither possible via DIMMs — for example, in Step 4 we request a hotplugged size of 500 MiB, which is impossible with 128 MiB DIMMs — nor with virtio-balloon.

6.2.2 Metadata Handling in the Guest OS. Second, we evaluate how virtio-mem handles metadata for memory when resizing in comparison to memory ballooning and traditional memory hot(un)plug in Linux guests.

Setup. We started a 4 GiB VM prepared for a maximum of 20 GiB. We configured memory onlining in the Linux guest to automatically online all added Linux memory blocks to the `MOVABLE` zone. We grew in 4 MiB steps and once we reached 20 GiB, we shrunk in 4 MiB steps again. For virtio-mem, we used a single virtio-mem device with a maximum size of 16 GiB. For DIMMs, we hot(un)plugged individual 128 MiB DIMMs when the requested size was properly aligned. We monitored the number of Linux memory blocks in the Linux guest after every resize request.

Results. Figure 3 shows the number of Linux memory blocks while growing, Figure 4 while shrinking. As virtio-balloon does not dynamically manage metadata, the number stayed fixed at 160. Both, DIMMs and virtio-mem, start with 32 memory blocks and end up with 160 memory blocks after hotplugging all 16 GiB; when hotunplugging memory, all added Linux memory blocks are removed again. While virtio-mem added a new Linux memory block when plugging the

[1] https://gitlab.com/virtio-mem/qemu/-/tree/vee21

[2] Via the QEMU monitor command "info numa".

[3] Via "MemTotal" in, e.g., */sys/devices/system/node/node0/meminfo* for node 0.

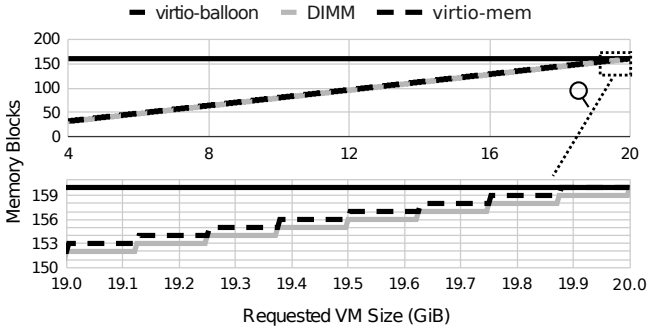

Figure 3. Number of Linux memory blocks in the VM while growing a 4 GiB VM prepared for a maximum of 20 GiB in 4 MiB steps to 20 GiB.

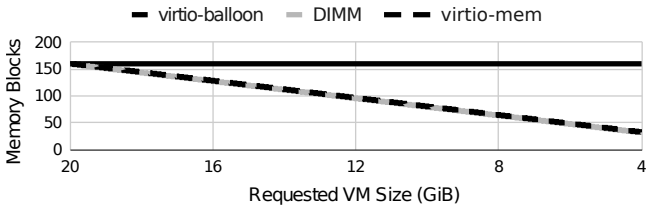

Figure 4. Number of Linux memory blocks in the VM while shrinking a 20 GiB VM prepared for a maximum of 20 GiB in 4 MiB steps to 4 GiB again, after growing it to 20 GiB.

first 4 MiB of such a block, DIMMs could only be added in one piece when the requested size was aligned to 128 MiB.

In Linux, the metadata overhead is usually 1/64: a 128 MiB memory block consumes at least 2 MiB of metadata; as a consequence, in our example virtio-balloon initially wasted at least 256 MiB of memory for metadata. DIMMs and virtio-mem allocated metadata dynamically on demand.

6.2.3 Shrinking with Unmovable Allocations.

Third, we evaluate how virtio-mem behaves in contrast to memory ballooning and traditional memory hot(un)plug when using the `MOVABLE` zone in Linux is impossible. Although some workloads can heavily fragment the `NORMAL` zone with unmovable allocations, we consider this a corner case and evaluate which behavior to expect in the common case.

Setup. We started a 4 GiB VM prepared for a maximum of 60 GiB, grew it to 60 GiB and executed memory-intensive workload. We used a single virtio-mem device with a maximum size of 56 GiB and hot(un)plugged individual 256 MiB DIMMs, to not exceed 256 slots. We configured memory onlining in the Linux guest to automatically online all added Linux memory blocks to the `NORMAL` zone. After the workload finished, we tried shrinking back to 4 GiB. We hotunplug DIMMs in the reverse order in which they were hotplugged. We measured the total VM size in QEMU[4] and the number of Linux memory blocks inside the Linux guest. We repeated the experiment 10 times.

Table 2. Selected SPECrate2017 benchmarks [42] used to simulate memory intensive workload. We measured the memory consumption of a single copy and calculated the number of copies to consume at most 57 GiB.

Benchmark	1 Copy (MiB)	Copies	Total (MiB)
502.gcc_r	1338	43	57534
531.deepsjeng_r	700	83	58100
557.xz_r with	726	80	58080
503.bwaves_r	822	71	58362
507.cactuBSSN_r	789	73	57597
526.blender_r	590	98	57820
527.cam4_r with	861	67	57687
549.fotonik3d_r	848	68	57664
554.roms_r	842	69	58098

Workload. We ran a subset of the SPECrate2017 suite [42], focusing on benchmarks that reach a substantial memory consumption after a short runtime. Note that we did not care about the actual benchmark results: we wanted to evaluate how this workload triggers fragmentation of unmovable allocations even after it finished. Table 2 lists the selected benchmarks, including the number of copies we ran to consume at most 57 GiB[5]. We ran each benchmark for 5 minutes.

Results. Figure 5 shows the total VM size and Figure 6 shows the number of Linux memory blocks inside the Linux guest after shrinking the VM. The standard deviation of our measurements for the total VM size is 662.14 [6] (DIMMs) and 12.87 (virtio-mem). The standard deviation of our measurements for the number of Linux memory blocks is 5.17 (DIMMs) and 3.4 (virtio-mem).

virtio-balloon can hotunplug all memory, as it operates on base page granularity and all VM memory. While a single unmovable allocation blocked a whole 256 MiB DIMM in our experiment from getting hotunplugged, it only blocked a 4 MiB memory block of a virtio-mem device. In *best / worst / average* case, virtio-mem hotunplugged 57232 MiB (99.80%) / 57188 MiB (99.73%) / 57205 MiB (99.76%) of 57344 MiB, while using DIMMs only hotunplugged 51712 MiB (90.18%) / 49408 MiB (86.16%) / 50918 MiB (88.79%).

Hotunplugged DIMMs never wasted memory on metadata and virtio-balloon did not remove any metadata. With a Linux memory block size of 128 MiB, 4 GiB require 32 blocks and 60 GiB require 480 blocks—memory ballooning kept metadata for additional 448 blocks allocated, corresponding to at least 896 MiB.

virtio-mem removed metadata for hotunplugged memory once possible (i.e., a complete Linux memory block was

[4] Via the QEMU monitor command "info numa" for virtio-mem and DIMMs, via "info balloon" for memory ballooning.

[5] A Linux VM with 60 GiB has roughly 58 GiB of memory available due to OS overhead, especially also for memory metadata.

[6] The large deviation is caused by the very coarse granularity of 256 MiB.

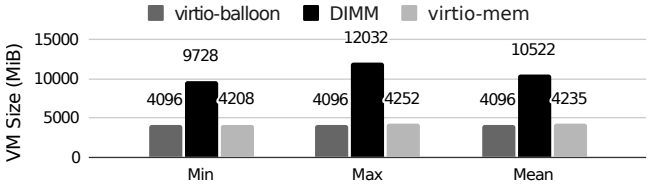

Figure 5. Final VM size when trying to shrink a 4 GiB VM prepared for a maximum of 60 GiB back to 4 GiB after growing it to 60 GiB and running memory intensive workload.

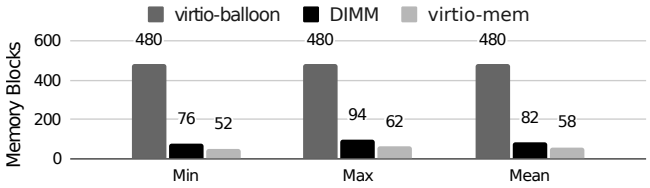

Figure 6. Final number of Linux memory blocks when trying to shrink a 4 GiB VM prepared for a maximum of 60 GiB back to 4 GiB after growing it to 60 GiB and running memory intensive workload.

hotunplugged). A VM with 4208 MiB requires 33 Linux memory blocks in the optimal case (removing 447), a VM with 4252 MiB and 4235 MiB requires 34 (removing 446). virtio-mem removed in the best case 428 (95.75%) out of 446, in the worst case 418 (93.72%) out of 447, and on average 421.8 (94.57%) out of 447.

6.2.4 Impact On System Performance While Shrinking.

Finally, we evaluate how shrinking a VM affects system performance, for example, on reduced system load after a temporary peak that required additional memory resources.

Setup. We started a 2 GiB VM prepared for a maximum of 10 GiB and grew it to 10 GiB. We used a single virtio-mem device with a maximum size of 8 GiB and a single 8 GiB DIMM for simplicity. We configured memory onlining in the Linux guest to automatically online all added Linux memory blocks to the `MOVABLE` zone. First, we reused the workload from Section 6.2.3, similarly configured to consume at most 9 GiB and running each workload for 60 seconds to simulate a busy system, e.g., randomizing the free page lists inside the guest OS. Second, after the workload finished, we started a benchmark and shrunk back to 2 GiB after 5 seconds, while the benchmark was still running.

Benchmarks. We used two different benchmarks that continuously measure the current system performance, before shrinking, while shrinking, and after shrinking. First, we used a modified STREAM benchmark [29] with 100 iterations on a single CPU core to measure the average memory bandwidth of a total of 1 GiB for each iteration; after every iteration of the benchmark, we recorded the end time and the memory bandwidth during that iteration. Second, we used the Fixed Time Quanta (FTQ) v1.1 [24] benchmark on a single CPU core that detects noise by measuring how much CPU-intensive work can be performed in a given time frame; we used 28 bits in the sampling interval and 300 samples.

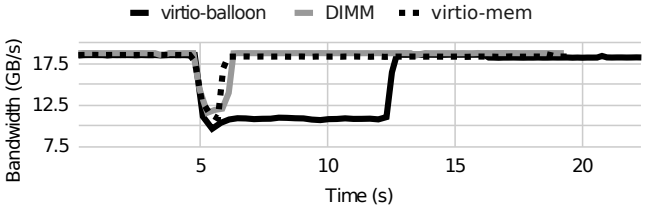

Figure 7. Memory bandwidth inside a 2 GiB VM prepared for a maximum of 10 GiB while shrinking it back to 2 GiB after growing it to 10 GiB. Memory bandwidth is measured using a modified STREAM [29] benchmark on 1 GiB of memory with 100 iterations.

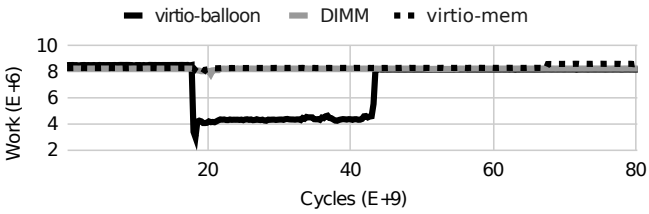

Figure 8. Noise inside a 2 GiB VM prepared for a maximum of 10 GiB while shrinking it back to 2 GiB after growing it to 10 GiB. Noise is measured using the FTQ v1.1 [24] benchmark with 28 bits in the sampling interval and 300 samples.

Result. Figure 7 shows the measured memory bandwidth and Figure 8 shows the measured noise of one execution of the experiment; we repeated the experiment four times and obtained comparable results with slight variations; however, we are only interested in the rough shape of the curves as minor variations are expected in virtualized environments. virtio-mem performed very similar to DIMMs: unplugging 8 GiB only took around one second, whereby the memory bandwidth was degraded and the CPU performance was affected minimally. The degraded memory bandwidth is expected: unplugging requires the guest OS to migrate busy memory content to other memory locations, whereby memory is temporarily not accessible. We assume that the short noise was due to hypervisor activity.

virtio-balloon needed significantly longer to shrink the VM and impacted memory bandwidth and CPU performance over the whole period. Investigating why balloon inflation impacts system performance that severely revealed that the root cause was QEMU discarding inflated memory using single `madvise` system calls for each base page that got inflated: temporarily removing these calls resulted in no observable impact with memory ballooning, meaning the overhead was almost completely caused by the hypervisor and not the guest OS. We expect balloon inflation time to be significantly shorter and system performance impact less severe with huge page ballooning in our experiment.

In all three cases, system performance was no longer affected once shrinking was complete. While the impact with virtio-mem and DIMMs was mainly due to guest OS activity, the hypervisor seemed to be responsible for the impact with virtio-balloon.

6.3 Discussion

We are now able to compare virtio-mem against traditional memory hot(un)plug and memory ballooning using the seven criteria introduced in Section 3.

C1: Use Case. virtio-mem has exactly one primary use case: dynamically adjusting the maximum memory available to a VM in cloud environments. In contrast to memory ballooning, memory hotunplug speed or minimizing guest OS impact has lower priority.

C2: Granularity. The granularity of virtio-mem is flexible, significantly smaller than with traditional memory hot(un)plug and independent of the maximum VM size. Our x86-64 implementation uses an effective granularity of 4 MiB. While virtio-mem with unmovable allocations on hotplugged memory cannot always hotunplug all memory like memory ballooning with base pages, it can hotunplug significantly more memory compared to traditional memory hot(un)plug.

C3: Memory Fragmentation. Similar to huge page ballooning, virtio-mem avoids breaking up THPs in the hypervisor. However, in contrast to memory ballooning, virtio-mem tries removing complete Linux memory blocks, freeing up metadata of unplugged memory. With unmovable allocations on hotplugged memory, some added Linux memory blocks cannot get hotunplugged completely, requiring that metadata for the whole block must be kept; however, the improvement compared to memory ballooning is substantial.

C4: NUMA-awareness. Similar to traditional memory hot(un)plug, but in contrast to memory ballooning, virtio-mem is completely NUMA-aware by design: having separate PMDs assigned to virtual NUMA nodes combined with device-local requests does not require the VM to respect the virtual NUMA topology to work reliably.

C5: Resize Capability. Similar to traditional memory hot(un)plug, virtio-mem can naturally grow a VM beyond its initial size, and cannot shrink below the initial VM size; however, virtio-mem allows for exposing additional memory via virtio-mem devices during system boot, which can get hotunplugged later.

C6: Resize Requests Handling. Similar to memory ballooning, virtio-mem requests the OS inside the VM to resize using a target size, giving the guest OS significantly more flexibility, and resulting in a significantly better user experience compared to managing separate units, such as DIMMs, in the hypervisor. In contrast to memory ballooning, resize requests are local to a virtio-mem device.

C7: (Re)boot Handling. As memory provided by virtio-mem is never exposed via firmware-provided memory maps, well-behaving VMs will not reuse unplugged memory, especially not when booting, during reboots and during fast OS reboots, allowing for a reliable detection of malicious VMs.

7 Conclusion

In this paper we explained traditional memory hot(un)plug and state of the art memory ballooning, compared both approaches, and provided the motivation for a new memory hot(un)plug mechanism to overcome the identified limitations. We then introduced and described a novel approach for fine-grained, NUMA-aware memory hot(un)plug in cloud environments: virtio-mem. We outlined the required changes to QEMU/KVM and Linux guests to implement virtio-mem, showcasing the adaptions needed in a hypervisor and a guest OS. Based on our evaluation of virtio-mem, we conclude that using PMDs for dynamically resizing VM memory significantly increases flexibility and usability compared to state-of-the-art.

A first version of virtio-mem for x86-64 has been integrated into upstream Linux and QEMU, except for some individual optimizations discussed in this paper. We have working prototypes for aarch64 with 4 KiB base pages and s390x that similarly allow for dynamic assignment of memory in 4 MiB granularity. Further, Kata Containers integrated experimental support for virtio-mem. An independent, first version of virtio-mem for x86-64 has been included into cloud-hypervisor [2], which proves that virtio-mem can be ported to other hypervisors.

Future Work. Reliably hotunplugging memory from Linux guests currently requires hotplugged memory to be onlined to the `MOVABLE` zone; we will improve hotunplug reliability in Linux when the `MOVABLE` zone cannot be used for all hotplugged memory by using a combination of the `NORMAL` and `MOVABLE` zone, by better locally grouping unmovable allocations within the `NORMAL` zone and by supporting smaller granularity, such as 2 MiB, in our virtio-mem driver.

Efficiently managing large, sparse memory mappings in Linux processes as used by our QEMU implementation is challenging. For example, using `mprotect` to disallow access to some regions is slow and impractical; we will investigate detecting access to unplugged memory blocks efficiently in QEMU using alternative approaches, like `userfaultfd` [5].

Finally, we will explore exposing memory with different semantics to VMs via PMDs, and using PMD-provided memory in novel ways – for example, as resizeable swap backend.

Acknowledgments

We would like to thank Red Hat for supporting the development of virtio-mem and Dr. David Alan Gilbert for his valuable review feedback. Further, we would like to thank the Linux and QEMU community for review and guidance.

References

[1] [n.d.]. About Kata Containers. Retrieved 2021-01-04 from https://katacontainers.io/

[2] [n.d.]. cloud-hypervisor/cloud-hypervisor: A rust-vmm based cloud hypervisor. Retrieved 2020-08-10 from https://github.com/cloud-hypervisor/cloud-hypervisor

[3] 2020. Official QEMU source repository at v5.2.0. Retrieved 2020-12-21 from https://git.qemu.org/?p=qemu.git;a=tree;h=553032db17440f8de011390e5a1cfddd13751b0b;hb=553032db17440f8de011390e5a1cfddd13751b0b

[4] Rafael Aquini. 2012. make balloon pages movable by compaction. Retrieved 2021-01-14 from https://lwn.net/Articles/523312/

[5] Andrea Arcangeli and Dr. David Alan Gilbert. 2014. Memory Externalization With userfaultfd. In *KVM Forum*.

[6] Christian Bargmann and Marina Tropmann-Frick. 2019. A Survey On Secure Container Isolation Approaches for Multi-Tenant Container Workloads and Serverless Computing. In *Proceedings of the SQAMIA 2019: 8th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications*, Zoran Budimac and Bojana Koteska (Eds.). Ohrid, North Macedonia.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, Bolton Landing, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[8] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference*. USENIX Association, Anaheim, CA, USA. https://doi.org/10.5555/1247360.1247401

[9] Antonio Bovenzi, Javier Alonso, Hiroshi Yamada, Stefano Russo, and Kishor S. Trivedi. 2013. Towards fast OS rejuvenation: An experimental evaluation of fast OS reboot techniques. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Pasadena, CA, USA, 61–70. https://doi.org/10.1109/ISSRE.2013.6698905

[10] Allyson Brito, Loïc Fura, and Bartłomiej Grabowski. 2011. IBM PowerVM Virtualization Active Memory Sharing. *IBM Redpaper* (2011).

[11] Dexuan Cui. 2019. [PATCH v2 1/2] x86/hyperv: Implement hv_is_hibernation_supported(). Retrieved 2020-12-06 from https://www.spinics.net/lists/linux-arch/msg57366.html

[12] George Dunlap. 2014. Ballooning, rebooting, and the feature you've never heard of. Retrieved 2020-12-06 from https://xenproject.org/2014/02/14/ballooning-rebooting-and-the-feature-youve-never-heard-of/

[13] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, Boston, MA, USA, 649–667.

[14] Pankaj Gupta. 2019. virtio pmem driver. Retrieved 2020-07-30 from https://lwn.net/Articles/791687/

[15] Dave Hansen, Mike Kravetz, Brad Christiansen, and Matt Tolentino. 2004. Hotplug Memory and the Linux VM. In *Proceedings of the Linux Symposium*, Vol. 1. Ottawa, Ontario, Canada.

[16] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-Copy Live Migration of Virtual Machines. *SIGOPS Operating Systems Review* 43, 3 (2009), 14–26. https://doi.org/10.1145/1618525.1618528

[17] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2018. HUB: Hugepage Ballooning in Kernel-Based Virtual Machines. In *Proceedings of the International Symposium on Memory Systems*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/3240302.3240420

[18] Joefon Jann. 2011. Dynamic Logical Partitioning for POWER Systems. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Vol. 44. Springer US, Boston, MA, USA, 587–592. https://doi.org/10.1007/978-0-387-09766-4_194

[19] Joefon Jann, Luke M. Browning, and R. Sarma Burugula. 2003. Dynamic reconfigurations: Basic building blocks for autonomic computing on IBM pSeries servers. *IBM Systems Journal* 42, 1 (2003), 29–37. https://doi.org/10.1147/sj.421.0029

[20] Marcus Kempe. 2017. VM online resize of CPU/Memory – Google Cloud Platform Feedback. Retrieved 2020-12-08 from https://googlecloudplatform.uservoice.com/forums/302595-compute-engine/suggestions/31659856-vm-online-resize-of-cpu-memory

[21] Daniel Kiper. 2011. [PATCH] xen/balloon: Memory hotplug support for Xen balloon driver. Retrieved 2020-05-05 from https://lkml.org/lkml/2011/3/28/108

[22] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux Virtual Machine Monitor. *Proceedings of the Linux Symposium* (2007).

[23] Octavian Lascu, Bill White, John Troy, Jannie Houlbjerg, Kazuhiro Nakajima, Paul Schouten, Anna Shugol, Frank Packheiser, Hervey Kamga, and Bo Xu. 2020. *IBM z15 (8561) Technical Guide* (1 ed.). IBM Redbooks.

[24] Lawrence Livermore National Laboratory. 2014. CORAL Benchmark Codes. Retrieved 2020-12-15 from https://asc.llnl.gov/coral-benchmarks

[25] Haikun Liu, Hai Jin, Xiaofei Liao, Wei Deng, Bingsheng He, and Cheng Zhong Xu. 2015. Hotplug or Ballooning: A Comparative Study on Dynamic Memory Management Techniques for Virtual Machines. *IEEE Transactions on Parallel and Distributed Systems* 26, 5 (2015), 1350–1363. https://doi.org/10.1109/TPDS.2014.2320915

[26] Dan Magenheimer. 2008. Memory Overcommit… without the commitment. *Xen Summit 2008* (2008).

[27] Dan Magenheimer. 2009. Transcendent Memory on Xen. *Xen Summit* (2009).

[28] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. 2009. Transcendent Memory and Linux. In *Proceedings of the Linux Symposium*.

[29] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995).

[30] Microsoft Corporation. 2010. Windows Server 2008 R2: Implementing and Configuring Dynamic Memory. Retrieved 2020-05-25 from http://download.microsoft.com/download/D/1/5/D15951B6-B33C-4A57-BCFB-76A9A6E54212/Implementing_and_Configuring_Dynamic_Memory_WP_SP1_final.pdf

[31] Shinobu Miwa, Masaya Ishihara, Hayato Yamaki, Hiroki Honda, and Martin Schulz. 2020. Footprint-Based DIMM Hotplug. *IEEE Trans. Comput.* 69, 2 (2020), 172–184. https://doi.org/10.1109/TC.2019.2945562

[32] OpenPOWER Foundation. 2020. *Linux on Power Architecture Reference – A PAPR Linux Subset (Rev 2.9 Final)*. OpenPOWER Foundation.

[33] Oracle Corporation. 2020. Oracle VM VirtualBox User Manual Version 6.1.6. Retrieved 2020-05-25 from https://download.virtualbox.org/virtualbox/6.1.6/UserManual.pdf

[34] Andy Pfiffer. 2003. Reducing System Reboot Time with kexec. Retrieved 2020-12-10 from http://web.archive.org/web/20030605203213/http://www.osdl.org/docs/reducing_system_reboot_time_with_kexec.pdf

[35] Shlomit S. Pinter, Yariv Aridor, Steven S. Shultz, and Sergey Guenender. 2008. Improving machine virtualisation with 'hotplug memory'. *International Journal of High Performance Computing and Networking* 5, 4 (2008), 241–250. https://doi.org/10.1504/IJHPCN.2008.022300

[36] Dulloor Subramanya Rao and Karsten Schwan. 2010. vNUMA-mgr: Managing VM memory on NUMA platforms. In *2010 International Conference on High Performance Computing*. IEEE, 1–10. https://doi.org/10.1109/HIPC.2010.5713191

[37] Rusty Russell. 2008. Virtio: Towards a de-Facto Standard for Virtual I/O Devices. *SIGOPS Operating Systems Review1* 42, 5 (2008), 95–103. https://doi.org/10.1145/1400097.1400108

[38] Joel Schopp, Dave Hansen, Mike Kravetz, Hirokazu Takahashi, Yasunori Goto, Matt Tolentino, and Bob Picco. 2005. Hotplug Memory Redux. In *Proceedings of the Linux Symposium*, Vol. 2. Ottawa, Ontario, Canada.

[39] Joel H. Schopp, Keir Fraser, and Martine J. Silbermann. 2006. Resizing Memory with Balloons and Hotplug. In *Proceedings of the Linux Symposium*, Vol. 2. Ottawa, Ontario, Canada.

[40] Rebecca Smith and Scott Rixner. 2017. A Policy-Based System for Dynamic Scaling of Virtual Machine Memory Reservations. In *Proceedings of the 2017 Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 282–294. https://doi.org/10.1145/3127479.3127491

[41] K. Y. Srinivasan. 2003. [PATCH V2 5/6] Drivers: hv: balloon: Implement hot-add functionality. Retrieved 2020-05-05 from https://lkml.org/lkml/2013/3/15/526

[42] Standard Performance Evaluation Corporation. 2020. SPEC CPU 2017. Retrieved 2020-12-21 from https://www.spec.org/cpu2017/

[43] Theo Thompson. 2015. When to use Hyper-V Dynamic Memory versus Runtime Memory Resize. Retrieved 2020-12-08 from https://techcommunity.microsoft.com/t5/virtualization/when-to-use-hyper-v-dynamic-memory-versus-runtime-memory-resize/ba-p/382223

[44] Linus Torvalds. 2020. Linux kernel source tree at v5.10. Retrieved 2020-12-21 from https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?h=v5.10

[45] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. *SIGOPS Operating Systems Review* 36, Special Issue (2002), 181–194. https://doi.org/10.1145/844128.844146

[46] Qi Zhang, Ling Liu, Calton Pu, Wenqi Cao, and Semih Sahin. 2018. Efficient Shared Memory Orchestration towards Demand Driven Memory Slicing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1212–1223. https://doi.org/10.1109/ICDCS.2018.00121