HyperAlloc: Efficient VM Memory De/Inflation via Hypervisor-Shared Page-Frame Allocators

Lars Wrenger
wrenger@sra.uni-hannover.de
Leibniz Universität Hannover
Hannover, Germany

Kenny Albes
albes@sra.uni-hannover.de
Leibniz Universität Hannover
Hannover, Germany

Marco Wurps
marco.wurps@proton.me
Leibniz Universität Hannover
Hannover, Germany

Christian Dietrich
dietrich@ibr.cs.tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Daniel Lohmann
lohmann@sra.uni-hannover.de
Leibniz Universität Hannover
Hannover, Germany

Abstract

The provisioning of the *right* amount of DRAM to virtual machines (VMs) is still a major challenge and cost driver in virtualization settings. Many VMs run applications with highly volatile memory demands, which either leads to massive overprovisioning in low-demand phases or poor QoS in high-demand phases. *Memory hotplugging* and *ballooning* have become established techniques (in Linux/KVM available via *virtio-mem* and *virtio-balloon*) to dynamically de/inflate the physical memory of a VM cooperatively, by having the guests give back unused memory to the hypervisor. However, current VM deflation techniques are either not DMA-safe, preventing the passthrough of important devices like GPUs or NICs, or are not flexible or fast enough to cope with the frequently changing demands of the guest.

We present HyperAlloc, a DMA-safe and extremely efficient mechanism for virtual machine de/inflation. The core idea is to provide the hypervisor direct access to the guest's page-frame allocator, greatly reducing the communication overhead. HyperAlloc can shrink virtual machines 362 times faster than *virtio-balloon* and 10 times faster than *virtio-mem* while having no measurable impact on the guest's performance. HyperAlloc's *automatic reclamation* provides for better memory elasticity by reducing the average memory footprint of a clang compilation by 17 percent compared to *virtio-balloon*'s free-page reporting while, again, having no measurable impact on the guest's performance.

CCS Concepts: • **Software and its engineering → Virtual machines**; *Cloud computing*; **Allocation / deallocation strategies**; *Virtual memory*; • **Computing methodologies → Shared memory algorithms**.

Keywords: Virtual Machines, Ballooning, Overcommitment, Allocators

ACM Reference Format:
Lars Wrenger, Kenny Albes, Marco Wurps, Christian Dietrich, and Daniel Lohmann. 2025. HyperAlloc: Efficient VM Memory De/Inflation via Hypervisor-Shared Page-Frame Allocators. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3689031.3717484

1 Introduction

Physical memory is generally considered to be the scarcest resource in cloud computing. Its provisioning remains a major challenge for providers due to the high hardware and energy costs of DRAM on the one side and *quality of service (QoS)* demands on the other. DRAM already accounts for over 30 percent of Meta's rack costs and power consumption [35]. Hence, good utilization of the scarce physical memory resources across multiple VMs is of utmost importance.

However, compared to other resources, the preemption and virtualization costs of memory are much higher: While it is technically easy and cheap for a hypervisor to dynamically detect (and redistribute) underutilized processors or network interfaces, it is a lot more expensive to do the same with idle memory. This limits elasticity, as many VM workloads exhibit highly fluctuating memory demands over the different phases of their execution [24]. Fuerst et al. have shown [19] that the memory resources of VMs running on Azure and Alibaba could be deflated by 30–50 percent most of their time for a performance impact of less than 1 percent.

Memory overcommitment [53] would increase utilization; however, cloud providers often refrain from doing this aggressively due to the difficulties of still providing their customers a defined QoS [6]. Instead, they strive towards more elasticity by finer-grained cost models for physical memory usage. An example is Amazon, which charges customers by *GiB·s* (GiB times seconds) on their Lambda *function-as-a-service (FaaS)* infrastructure. However, compared to the on-demand microservices in FaaS settings, VM instances running in the cloud have much longer lifespans and much higher preemption costs, which imposes challenges for transferring such pricing model to *infrastructure-as-a-service (IaaS)* settings.

Cooperative VM Memory De/Inflation Nevertheless, many clients would prefer to pay only for the memory they actually need at a given time [5, 11]. Some authors have even suggested real-time auctioning of physical memory among VMs [6]. However, being accustomed to virtual memory, clients usually do not know how much physical memory they need. But their OS does! With an extra component running as a proxy *inside* the guest VM's OS, the hypervisor can approach the guest's physical memory-management subsystem to find idling (unutilized) page frames in low-demand phases, which it then safely can *reclaim*. Examples of such cooperative reclaiming techniques include memory ballooning [24], memory hotplugging [23], and memory probing [55]. While these techniques have proven useful in practice, we argue that they are still not flexible or fast enough to cope with frequently changing guest demands because of their high overheads for probing, communication, and guest-side defragmentation. Additionally, some of them are *not* DMA-safe, preventing the pass though of devices like GPUs into VMs [55].

Our Contributions We present HyperAlloc[1], a new approach for virtual machine de/inflation. HyperAlloc integrates the concept of cooperative memory management directly into the guest's page-frame allocator, which it accesses via a lock-free memory-mapped interface, greatly reducing the communication overhead. In our evaluation with Linux/QEMU, HyperAlloc shrinks VMs up to 362 times faster than *virtio-balloon* and 10 times faster than *virtio-mem*.

By its direct integration into the guest's page-frame allocator, HyperAlloc additionally provides DMA safety by design, including also reclamation in VMs that require device passthrough. Its *automatic reclamation* mode reduces the memory footprint (in GiB·s) of a clang compilation by 17 percent compared to *virtio-balloon*'s free-page reporting without any significant impact on the guest's performance.

2 Problem Analysis

We first discuss the fundamental challenges for the hypervisor-guest memory interface on the example of the existing approaches, laying the ground for our HyperAlloc design, as well as providing an overview of the directly related work. Fundamentally, in all cases, the hypervisor relies on a *cooperating* guest that points it to reclaimable memory, which ideally does not contain data that has to be migrated, saved, or restored.

In the case of memory ballooning [45, 53], the hypervisor interacts with a guest-level kernel module (called the balloon driver) that allocates guest-physical frames from the guest's page-frame allocator and reports them back to the hypervisor. The hypervisor then can remove the respective frames from the VM and its *extended page tables (EPTs)*, shrinking the amount of host-physical memory available for the guest. To give back memory, the hypervisor instructs the guest-level kernel module to free the previously allocated frames, which on the next access are faulted (back) into the EPT.

Elasticity The virtio-balloon implementation [43] for Linux/QEMU provides an additional *automatic mode* (free page reporting), where the balloon driver periodically reports free frames to the hypervisor to be reclaimed, facilitating dynamic memory elasticity. Automatically reclaimed frames are *not* allocated from the guest's allocater, but remain logically free for the guest so that they can still be allocated. If this happens, they are as above faulted (back) into the EPT on the next access.

As virtio-balloon reclaims individual 4 KiB pages, it has to issue a lot of hypercalls and subsequent unmap syscalls on the host and may induce many EPT faults. This can lead to a substantial performance overhead. Hu et al. [24] have shown that the overhead could be significantly improved by increasing the granularity to 2 MiB huge pages.

DMA Safety The substantial limitation of virtio-balloon is that it cannot be used in conjunction with device passthrough. As described above, a reclaimed guest-physical frame is logically still available to the guest's allocator, which itself is not aware of reclamation and, hence, might select it upon some allocation. If this frame is then accessed by a CPU, an EPT fault occurs, in which the hypervisor actually installs a host-physical frame for it. However, if the guest has instead given the frame to a peripheral device for DMA, the DMA transfer will fail, as most DMA-capable devices are unable to trigger IO page faults [8, 51]. Additionally, the DMA memory has to be *pinned* on the hypervisor side to prevent swapping or migration (e.g., by Linux's memory compaction or same-page merging). So due to its reliance on page faults, virtio-balloon is inherently incompatible with device passthrough, which limits its applicability for IO-intensive applications.

Hildenbrand and Schulz [23] suggest to use memory hotplugging [45] as a DMA-safe alternative to ballooning. In *virtio-mem*, the hypervisor interacts with a guest-level hotplug driver to extend/shrink the guest-physical memory by adding/removing virtual DIMMs at 2 MiB granularity. DMA safety is achieved by prepopulating *all* guest-physical frames for blocks when they are "plugged-in"; hence, no EPT and IO page faults will occur later on. However, this pre-population leads to overprovisioning. Furthermore, virtio-mem does not support elasticity by automatic reclamation.

To overcome this, Wang et al. [55] propose VProbe, an automatic deflation mechanism that also provides for DMA safety. Here, the hypervisor gets memory-mapped access to the Linux guest's physical-frame metadata (`struct page`),

[1]Available at github.com/luhsra/hyperalloc-bench

which also contains the frame's reference counter. Thereby, VProbe can automatically detect and reclaim unused memory (refcount=0) without the need for explicit host–guest communication. For DMA safety, VProbe needs to detect when a reclaimed frame is allocated by the guest's allocator. For this, it write-protects the underlying `struct page` in the EPT. As a side effect of allocation, the Linux buddy allocator increases the refcount in `struct page`, so an EPT fault occurs, in which the hypervisor can repopulate the guest-physical page. However, as the guest's allocator still remains agnostic to reclaimed memory, contradicting allocation patterns may lead to a high number of faults and un/map operations – when the guest frequently allocates reclaimed frames even though others are available. The Linux buddy allocator, for instance, maintains per-core caches of free frames to reduce contention; the respective frames have a much higher probability of being allocated next [56].

3 HyperAlloc: Bilateral Memory Allocation

Instead of such *indirect* interaction with the guest's page-frame allocator via guest-level proxies or side effects of allocation, HyperAlloc overcomes all these limitations by the *direct* integration with the guest's allocator. In particular, we give the hypervisor write access to the allocator state so that it can detect *and* directly mark guest-physical pages as allocated or reclaimed; the allocator is used *bilaterally* by both guest and hypervisor. For our implementation, we build upon LLFree, a scalable page-frame allocator suggested by Wrenger et al. [56], which replaces the Linux buddy allocator. LLFree is particularly suitable for our approach due to its lock-free and pointer-free design, which constructively avoids control-flow dependencies between hypervisor and guest, as all operations are implemented by atomic memory transactions.

3.1 HyperAlloc in a Nutshell

Fig. 1 gives an overview of the HyperAlloc approach and illustrates the process of reclaiming unused memory from a VM without transitioning to the guest: In a QEMU/KVM setup, the virtual-machine monitor is split into an in-kernel part (KVM) that abstracts hardware-virtualization primitives (i.e., EPTs, virtual CPUs) for a user-space monitor process (QEMU) that emulates devices and decides on high-level resources (e.g., memory size).

With HyperAlloc, the QEMU monitor has shared access to the guest-physical allocator's state to identify unused memory and to mark memory as reclaimed/allocated for the guest. For example, if we want to shrink the maximally available guest memory, we can remove the host-physical frame 47 (HP47), which is available as guest-physical frame 1 (GP1) and currently marked as free, as follows:

❶ HyperAlloc marks GP1 as evicted *and* allocated in the guest-physical allocator's state, which tells the *guest* not to allocate or access this huge page.

❷ It unmaps the host-physical frame HP47 from the EPT and the IOMMU page tables via standard KVM interfaces, giving it back to the host allocator.

❸ The QEMU monitor updates HyperAlloc's authoritative reclamation state for GP1 to *hard reclaimed* (H), which marks, in contrast to *soft reclaimed* (S), that the frame should not be repopulated on demand.

3.2 Reclamation States

In the following, we look at the abstract state of a single memory page frame and its state transitions during reclamation. While we usually reclaim memory on huge-page granularity, our HyperAlloc concept is not restricted to this granularity. In Sec. 4.1, we will discuss the mapping of these states to the LLFree allocator.

Page States For HyperAlloc, the state of a page (Fig. 2) is a tuple with four elements that consists of a host (M, R) and a guest part (E, A). Only the guest part is accessible by both parties to ensure safety and security. On the host side, we have:

$M \mapsto \{0, 1\}$: Mapped indicates whether the page is backed with host-physical memory in all relevant hypervisor-level page tables (i.e., EPT, IOMMU).

$R \mapsto \{I, S, H\}$: In the reclamation state, HyperAlloc keeps track whether a page is currently **I**nstalled, **S**oft reclaimed, or **H**ard reclaimed. For a reclaimed page $M = 0$ holds.

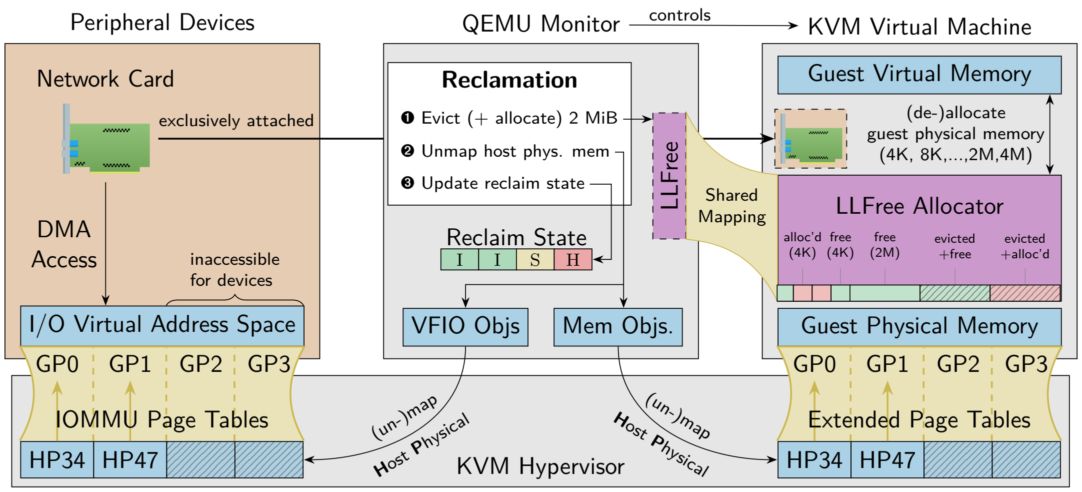
On the guest side, we additionally maintain:

$E \mapsto \{0, 1\}$: The evicted hint informs the guest that the page was reclaimed and is not backed by physical memory. E is a one-way synchronized copy of $\neg M$.

$A \mapsto \{0, 1\}$: Allocated indicates whether the huge page, or parts of it, is allocated within the *guest*.

Guest-Level Allocation The guest's physical-memory allocator can free and allocate non-evicted frames by toggling the allocated flag (blue arrows in Fig. 2) without hypervisor interaction. Only for allocations of evicted pages ($E = 1$), the guest has to trigger the hypervisor (via virtio-queue) once to install a host-physical page and also remove the evicted hint ($E \leftarrow 0$).

Reclamation HyperAlloc can reclaim memory that is not allocated ($A = 0$) by the guest in two modes: *hard* and *soft* reclamation. For *hard reclamation* ($R \leftarrow$ H), where the goal is to remove the memory permanently (i.e., reducing the maximal guest memory), the host marks the frame for the guest as allocated *and* evicted ($A \leftarrow 1, E \leftarrow 1$). This ensures that the frame is not available for the guest allocator. For *soft reclamation*, the QEMU monitor only sets the evicted hint ($A = 0, E \leftarrow 1$), keeping the frame as usable for guest



The controlling QEMU monitor has shared-memory access to the VM's allocator state and marks those huge pages as evicted/allocated that it removes from the EPT and the IOMMU page table. The virtual machine requests (not shown) evicted huge pages on allocation from the monitor, thus ensuring DMA safety.

Figure 1. Overview of HyperAlloc Concept.

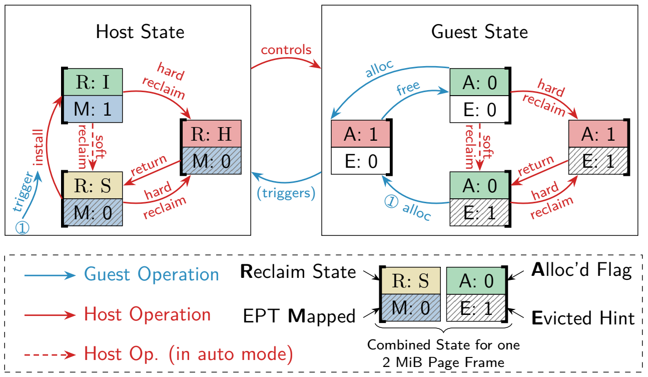


Figure 2. State Transition Diagram for *one* Memory Frame

allocations but at a higher cost. After updating the guest state, we remove ($M \longleftarrow 0$) the frame from all guest-accessible mappings (i.e., EPT and IOMMU), perform TLB invalidations, return the memory to the host allocator, and update R.

In both reclamation modes, the guest is informed about the reclaimed state of the frame, which he can use to steer its allocation policy. In our prototype, we extended LLFree to prefer non-evicted over evicted frames for allocations.

Return and Install The hypervisor can explicitly be instructed (e.g., via the management console) to *return* hard-reclaimed memory to the guest by setting the state of the respective frames to soft-reclaimed on both sides ($A \longleftarrow 0$, $E = 1$) and ($R \longleftarrow \mathrm{S}$). This allows us to implement a flexible soft limit while having an adaptable hard limit.

To actually *install* soft-reclaimed frames, we let the guest allocator issue a hypercall on allocation, which triggers the hypervisor to provide host memory, map it in all guest-accessible page tables, and update its reclamation state. The allocation waits for this hypercall to terminate before returning the allocated frame. This is necessary because the conventional method of installing memory on access (i.e., waiting for the EPT fault) is not sufficient for DMA-safety, as the OS is then still free to reclaim or remap the accessed pages at any time (see Sec. 2). Instead, we explicitly *pin* the VM's memory pages with the hypercall when they are allocated by the guest *before* they can be accessed. Still, this install-on-allocate should perform equally good, as: (a) An explicit hypercall is, performance-wise, not inherently more expensive than an implicit EPT fault. (b) Unlike with virtual memory, page frames requested from the guest's *physical* memory allocator are likely to be accessed shortly thereafter, so we cannot expect significant benefits from delayed provisioning.

Invalid Guest States Like with any approach for cooperative host–guest memory management (e.g., ballooning [24, 45, 53], hotplugging [23, 45]), both sides have to adhere to an interaction protocol. For our reclamation protocol, both guest and host inspect and update the shared per-frame guest state. Therefore, we need to discuss the potential safety/security implications of non-conforming or malicious guests:

HyperAlloc never makes decisions upon E but has its own frame-state tracking (R), making the E flag a mere read-only copy of $E \longleftarrow (R \neq \mathrm{I})$. Thus, a maliciously manipulated E has no impact on the hypervisor. Similarly, HyperAlloc updates A on the hard reclamation and return transition, where we set $A \longleftarrow (R = H)$. Only for the reclamation decision, the hypervisor inspects A to find reclaimable pages. While this allows a non-conforming guest to resist memory reclamation (i.e., to *not* cooperate), it bears no safety or security implications. Given a fine-grained memory pricing model, the guest would just have to pay for the extra memory.

If the guest never triggers the install of a reclaimed frame, we do not guarantee that accesses to it are successful. For example, DMA transfers to such frames might fail, leading to guest-side errors or even termination. However, this affects only the non-conforming guest itself and cannot compromise the hypervisor's or other VMs' security.

Regarding safety, we must also consider concurrent host/guest operations: As we access the shared state exclusively through atomic operations, the shared state itself does not pose a problem. However, on the host side, concurrent reclaim, return, and install operations may impose race conditions inside the monitor. In our current implementation, the hypervisor synchronizes these operations with a per-VM lock. We also considered per-frame locking via a "lock" reclamation state but left this for future work, as we could barely notice any contention, even for highly parallel workloads.

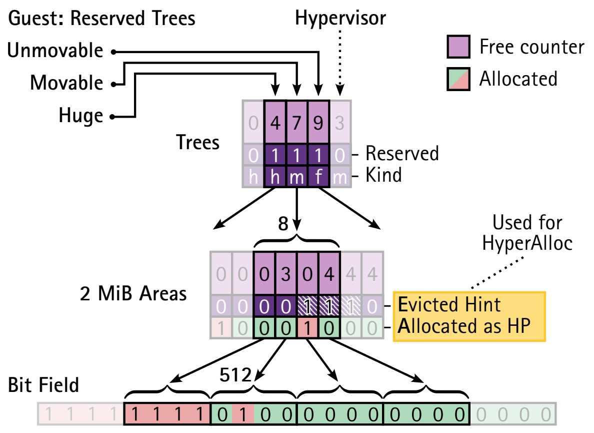
3.3 Management Policies

HyperAlloc uses the provided reclamation mechanisms (hard and soft) to implement two management policies:

Adaptable Memory Hard Limit QEMU already supports an adjustable upper memory limit, whose reduction below the initial allocation is usually implemented via memory ballooning or hotplugging. With HyperAlloc, we use hard reclamation to decrease the maximal memory size of a guest if triggered from the QEMU console or QEMU's QOM API. In contrast to others [23, 24, 45, 53], HyperAlloc usually does not have to transition to the guest or stop it. Only if there is not enough free memory in the guest's allocator, we instruct the guest to free the remaining memory from its caches and retry our hard reclamation afterward. This cache purge combined with the reservation of all free pages induces the same memory pressure as virtio-balloon, potentially triggering further memory reclamation mechanisms within the guest.

To increase the upper limit, we use the return operation to add more soft-reclaimed guest-physical memory, delaying the actual memory allocation until the guest triggers *install*. While our current implementation does not allow growing a VM beyond its initial memory allocation, it would be possible to combine HyperAlloc with memory hotplugging (Sec. 6).

Soft Limit by Automatic Reclamation With automatic reclamation enabled, HyperAlloc periodically removes unused frames from the running guest, shrinking the currently attached host-physically memory. Every 5 seconds, we scan the reclamation-state array for installed ($R = I$) pages and inspect the guest's allocator state if the page is free ($A = 0$). Both data structures are densely packed, so this linear search bears only a tiny cache load and, thus, minimal performance impact. In our current implementation (2 bits for R, 16 bits for A), we access $\frac{2 \cdot 512}{8 \cdot 64} + \frac{16 \cdot 512}{8 \cdot 64} = 18$ consecutive cache lines to scan 1 GiB of guest-physical memory for free huge pages.



Interlayer connections (arrows) are *not* implemented with pointers but via offset arithmetic.

Figure 3. LLFree Allocator State [56].

4 HyperAlloc in Linux

To integrate HyperAlloc with Linux, we use and extend LLFree [56], a lock- and log-free allocator that replaces the Linux buddy allocator and primarily focuses on multicore scalability and fragmentation avoidance. For us, its lock-free design enables the efficient access to the guest allocator from different privilege levels, while its fragmentation-awareness improves the availability of free huge frames to reclaim.

4.1 LLFree Overview

From a high-level perspective (Fig. 3), LLFree is a bitmap allocator that uses two levels of free-counter indexes (trees consist of areas) to speed up and steer the search for free memory: For each base frame, a bit in the bit field indicates whether it is free. An *area* covers 512 such bits, which corresponds exactly to one huge frame and is associated with a 16-bit index entry. This entry includes a 9-bit counter indicating how many base frames are free and an allocated flag that allows for the atomic allocation of the entire huge frame. If the area counter is 512 and the flag indicates free, the covered huge frame is entirely free and can be allocated as a huge frame with a single *compare-and-swap (CAS)* operation.

A fixed number of consecutive areas (e.g., 8) form a *tree*, whose index entry also contains a free-frame counter. Trees are also essential for LLFree's anti-fragmentation policy, which tries to avoid allocating frames from "almost full" trees (where most frames are free). For this, CPUs dynamically *reserve* trees, preferring "half depleted" and "almost depleted" over "almost full" trees, from which they allocate memory until allocations fail. Due to this reservation policy, "almost full" trees (and areas) defragment without active memory compaction.

For HyperAlloc, two properties of the LLFree data structures are most important: (1) Bit fields and counter indices are stored as densely packed arrays, where frame states can be located through simple offset arithmetic without relying on guest-side pointers. (2) All operations on the state are performed lock-free using atomic CPU instructions only – there are no locks involved.

4.2 Integration with LLFree and KVM

For our integration, we chose to reclaim unused memory on the granularity of huge frames, which reduces the reclamation overhead but also ties the reclamation effectiveness to the huge-page fragmentation behavior of the guest.

State Mapping We integrate the guest part (A, E) of HyperAlloc's per-frame state (see Fig. 2) into the area-index entry and use the existing huge-frame–allocated flag for A. As the area counter and flag require 11 bits, we can choose one bit for the evicted hint (E) from the 5 remaining bits. By co-locating counter, allocated flag, and evicted hint in the same 16-bit word, it is also ensured that the host can induce guest transitions atomically with a CAS operation.

Reservation Policy Besides using the eviction hint for the allocation policy, we also modified the tree-reservation policy to further improve its fragmentation avoidance: The original LLFree uses *per-core* tree reservations to avoid false sharing. In our experiments, we saw that a few long-living allocations (e.g., in the page cache) provoke higher huge-frame fragmentation. Therefore, we removed the per-core reservations in favor of *per-type* reservations: Linux distinguishes between three allocation types, which usually have different lifetimes: *unmovable* kernel allocations, *movable* user allocations, and *huge* allocations. We separate these types into different trees by having one global reservation per type and by introducing a 2-bit type field in the tree-index entry. Our application-level experiments showed no negative performance impact of removing the per-core trees. We assume that other bottlenecks within the memory-management of Linux, as measured by Wrenger et al., dominate the results [56].

The per-type reservations lead to less fragmentation in the long run. They also increase the effectiveness of Linux's active defragmentation (memory compaction). While Linux developers have undertaken attempts to separate allocations, our experiments showed that our LLFree-based type separation performs better, and increases the availability of huge frames (see Sec. 5.5). Additionally, we reduced the tree size from 32 areas (64 MiB) to 8 areas (16 MiB) to make the reservation policy and its fragmentation avoidance more accurate.

Linux additionally divides the physical memory into *zones* based on their physical address and NUMA locality. On x86, there are the global *DMA* (16-bit addressable) and *DMA32* (32-bit addressable) zones, plus for each NUMA node a *Normal* zone[2] Every populated zone has its individual LLFree instance. When reclaiming memory, the host starts with the LLFree instances of the Normal zones before continuing with the DMA32 zone. The tiny DMA zone (16 KiB) is ignored.

Locating the Allocator State To interact with the guest allocator, HyperAlloc has to locate the allocator state in memory. During boot, the guest uses `virtio` queues to communicate the guest-physical address of the LLFree metadata to the QEMU monitor. LLFree's compact state, consisting mainly of the three state arrays, is well suited for sharing with the hypervisor as it is only accessed by LLFree and does not contain any unrelated metadata (unlike the `struct page` used by VProbe [55]). The monitor maps the state into its own virtual address space and creates a cloned LLFree object that works on the shared state. From then on, both sides can inspect and modify the same LLFree instance directly over shared memory without a host-guest transition. This is done for every memory zone of the guest and, respectively, every LLFree instance.

KVM/QEMU Integration For our prototype, we decided to integrate HyperAlloc into QEMU, the user-space monitor for kernel-managed KVM guests (see Sec. 3.1). Thereby, HyperAlloc requires no modifications to the host's kernel. The downside is that HyperAlloc, as a user-level component, has no direct access to the related page tables but instead has to use system calls to manipulate guest mappings. For example, we have to use `madvise(DONT_NEED)` to remove EPT mappings and VFIO for IOMMU mappings. Installing a frame requires two mode switches (guest – QEMU – kernel), whereas only one would be necessary if HyperAlloc were part of KVM. To ease this issue, we aggregate huge frames during reclamation and unmap them with a single syscall, which has proven effective due to LLFree's compact allocation behavior (linear scan) and anti-fragmentation policy.

Another disadvantage, which we share with all other monitor-level deflation techniques [23, 24], is that KVM handles EPT faults directly within the kernel without informing the monitor process. This is a known limitation of KVM, by which a non-conforming guest may allocate host-physical memory for evicted frames without giving HyperAlloc the possibility to update its reclamation state. The effects of such behavior would be similar to those discussed in Sec. 3.2: The extra memory does not imply security/safety issues, and the host can detect it by comparing the reclamation state with the resident-set size (RSS) of the QEMU process. Thus, in the case of fine-grained memory accounting, the guest does not benefit from this extra memory.

[2]The per-node *Movable* zone is usually empty on x86, as active defragmentation is done in smaller granularities on this architecture.

Table 1. Evaluation candidates and their properties.

Name	Granu-larity	Manual Limit	Auto Mode	DMA Safety	Implementation taken from
virtio-balloon	4 KiB	✓	✓	×	Debian 12
…-huge [24]	2 MiB	✓	✓	×	Own reimpl.[1]
virtio-mem [23]	2 MiB	✓	×	✓	Debian 12
~~VProbe~~ [55]	4 KiB	×	✓	✓	*unavailable*
HyperAlloc	2 MiB	✓	✓	✓	Own[1]

[1]All artefacts are available at github.com/luhsra/hyperalloc-bench

5 Evaluation

Memory reclamation techniques compete in two dimensions: The *overhead* of reclamation and the *elasticity*, that is, how tight we can shrink a guest to its actual memory demand.

5.1 Benchmark Competitors

We compare HyperAlloc to the state-of-the-art memory deflation techniques (see Tab. 1). For ballooning, we chose virtio-balloon, which is supported by both QEMU and Linux out of the box. As its performance is limited by its 4 KiB page granularity, we recreated huge-page ballooning from Hu et al. [24] (virtio-balloon-huge). Both variants support manual and automatic memory reclamation via *free-page reporting* but, as they rely on page faults, are *not* DMA-safe on their own [8, 51].

For memory hotplugging, we pick virtio-mem [23], which is also part of QEMU but is mainly designed for growing VMs efficiently. While shrinking the VM is possible, it can only reliably reclaim memory from the *Movable* zone and does not support automatic reclamation. However, it provides DMA safety, as all plug/unplug operations are explicit. To quantify the performance impact of DMA safety, we measure HyperAlloc and virtio-mem with and without device passthrough of a VFIO-managed network card. Although our benchmarks do not use the card, its IO page tables must be kept synchronized, resulting in additional runtime costs.

We also would have liked to compare against VProbe, which provides for both auto deflation and DMA safety [55]. Unfortunately, the authors could not provide us with its source code, as it relies on additional proprietary modifications that are only available within Alibaba's environment. We discuss their concept further in Sec. 6.

5.2 Environment

All experiments were conducted on a machine with two Intel Xeon Gold 6252 CPUs (2x24 cores @ 2.1 GHz) and 384 GB of DDR4 memory, split evenly across two NUMA nodes. To increase reproducibility, we disabled Intel Hyper-Threading and Turbo Boost, locked the cores to their maximum clock speed, and pinned the VMs to the first node.

Both hypervisor and guest used Debian 12 (bookworm) with a Linux 6.1 kernel and QEMU/KVM 8.2.50. While we employed the provided Debian configuration for the host, the guests used the default x86 kernel configuration with enabled virtio, VFIO, and transparent huge pages. For the HyperAlloc scenarios, we additionally replaced the buddy allocator with LLFree [56] in the guest's kernel and used HyperAlloc instead of virtio-balloon/virtio-mem. For virtio-balloon-huge, we used our reimplemented version from [24]. All kernel- and QEMU-variants were built with the LLVM 14.0.6 toolchain and the default compiler flags.

Unless specified otherwise, we used a VM with 12 vCPUs and 20 GiB of memory. For virtio-mem, we split that into 2 GiB of regular and 18 GiB hotpluggable system memory by allowing virtio-mem to plug it into the *movable* zone, so it can be unplugged later.

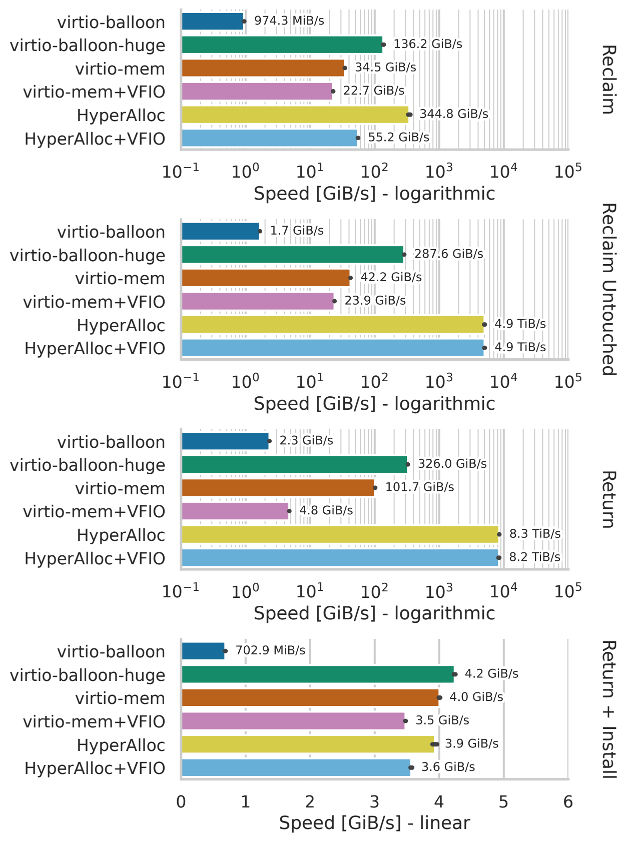


Figure 4. Speed of reclaiming/returning memory (logarithmic scale). For HyperAlloc this measures the *hard* reclamation.

5.3 Reclamation Speed

First, we determine each candidate's raw performance for resizing a VM with four micro benchmarks:

Reclaim This is the speed for reclaiming *and* unmapping memory from the VM, shrinking its memory footprint. We ensure that the memory is present by writing into 19 GiB of guest pages[3] before the benchmark.

Reclaim Untouched This measures the speed for reclaiming memory that was unmapped before but has not been

[3]Requesting all 20 GiB would trigger an OOM error.

accessed and installed since. For this, we reclaim the VM in advance and grow it again before starting this benchmark.

Return This measures how fast we can return pages to a VM, increasing the memory limit *without* allocating or touching the returned memory.

Return+Install This measures how fast we can increase the VM's memory limit *and* access the returned pages. For this, we use a guest-kernel module to allocate 19 GiB of memory[3] and write into each 4 KiB frame.

For the reclamation, we shrunk the VM's hard limit to 2 GiB (from 20 GiB) and vice versa for returning. We repeated this procedure 10 times for each candidate. The guest-kernel module in *return+install* was executed single-threaded. Fig. 4 shows the achieved grow/shrink rate, while the error bars denote the 95 percent confidence interval. Please note the logarithmic x-axis for the first three graphs.

Reclaim The reclamation of touched memory is primarily affected by the used granularity. Virtio-balloon, due to its 4 KiB page granularity, performs poorly with a speed of only 0.95 GiB/s. Here, each 4 KiB page is allocated, sent to the hypervisor[4] and discarded with an *madvise* syscall, which results in significant transition overheads. These also occur if the guest did not touch the memory. The speedup results only from the reduced EPT-manipulation costs. Virtio-balloon-huge with its 2 MiB granularity mitigates this bottleneck, increasing performance 143 times.

Similarly, virtio-mem also works with 2 MiB huge pages. Its performance falls in between the two previous candidates, reaching speeds of 34 GiB/s. Reclaiming untouched memory is faster, as it is not faulted in and does not have to be unmapped by the hypervisor. The main bottleneck in both cases appears to be the hot(un)plugging infrastructure. With an attached device, virtio-mem also has to manage the IOMMU memory mappings with VFIO, which results in a 52 percent slowdown. Since virtio-mem does neither interact with the guest's allocator nor uses a virtual IOMMU [8, 51], it only achieves DMA safety by immediately pinning and mapping *all* memory when the memory limit grows. When virtio-mem+VFIO shrinks the VM, these operations are not only reversed, but they also have to flush the IOTLB, even if the memory was never touched. Because of this pre-population, virtio-mem+VFIO shows no real difference between removing touched and untouched memory.

With a shrink rate of 344.8 GiB/s, HyperAlloc outperforms all competitors, being 10 and 3 times faster than virtio-mem and virtio-balloon-huge. Removing untouched memory is even faster (4.92 TiB/s), since we only modify allocator and reservation state, and can skip the expensive unmap operations. With an attached device, the IOMMU-management overheads make shrinking 6.3 times slower; still HyperAlloc is the best DMA-safe technique. Removing untouched memory remains unaffected, as we only have to update the IOMMU for memory that the guest previously allocated.

[4] Even though the hypercalls are aggregated (up to 256 pages per hypercall), the other syscalls and page operations are not.

Return Growing the VM's memory limit is faster for most candidates, as the returned pages are populated lazily (on EPT faults / *install* hypercalls). Again, virtio-balloon is the slowest competitor and can only grow the VM with 2.3 GiB/s as deflating the balloon requires that the previously allocated 4 KiB frames are returned one-by-one[4] to the guest allocator. Virtio-balloon-huge provides a sizable performance increase (139×), with growing being about twice as fast as shrinking.

Virtio-mem can grow the memory limit by 102 GiB/s, once again falling short of virtio-balloon-huge. The reason for this difference is that virtio-mem makes hypercalls for every plugged 2 MiB block, while the virtio-balloon(-huge) guest driver returns pages without extra hypercalls (both ultimately populate on EPT-fault). Virtio-mem with VFIO is 21 × slower than without VFIO because it has to pre-populate the memory for DMA-safety.

HyperAlloc outperforms all candidates by a considerable margin, working at 84 and 26 times the speed of virtio-mem and virtio-balloon-huge. As with removing untouched memory, returning it just modifies the respective bits in the allocator state, taking 229 ns per huge page (compared to 388 ns for reclaiming an untouched huge page). As expected, adding a device to the VM does not affect the performance. Memory is only mapped to the IOMMU once the guest allocates it.

Return+Install As ballooning, virtio-mem without VFIO, and HyperAlloc all populate the returned memory lazily, we also measured the speed of returning *and* accessing the memory (Return+Install). Again, virtio-balloon's 4 KiB granularity makes it the slowest candidate. Virtio-balloon-huge reaches the highest data rate of 4.2 GiB/s, shortly followed by both virtio-mem and HyperAlloc with 4 GiB/s. To put this into perspective: our benchmark accesses mapped pages at 17 GiB/s. Even though HyperAlloc's install hypercalls are about 6 percent slower than virtio-mem's EPT faults (due to having an additional context switch to QEMU which then uses madvise to manipulate the VM), the faster initial *return* time compensates for this difference. Therefore, the combined return+install times are almost equal. The same is true for device passthrough (VFIO), where virtio-mem pre-populates the IOMMU, thus having more upfront costs than HyperAlloc, which pays the mapping costs on demand.

Overall, we see that HyperAlloc is significantly faster than the competition for reclaiming and returning memory while being DMA-safe. Only installing returned memory is slightly slower than virtio-balloon-huge due to the additional context switch to the QEMU monitor. However, this overhead would probably disappear if we integrated HyperAlloc into KVM itself, removing the extra context switch.

Table 2. 1st percentile for STREAM and FTQ benchmarks.

	STREAM [GB/s] Threads			FTQ [e^6] Threads		
Candidate	1	4	12	1	4	12
Baseline	10.3	26.0	69.0	9.4	10.2	30.6
virtio-balloon	6.2	10.9	30.9	5.9	7.5	24.9
virtio-balloon-huge	10.1	25.5	67.8	9.5	10.1	30.1
virtio-mem	10.2	13.1	31.9	9.5	8.6	28.7
virtio-mem+VFIO	10.3	12.6	18.4	9.4	8.4	28.3
HyperAlloc	10.3	26.3	70.1	9.5	10.2	30.7
HyperAlloc +VFIO	10.3	26.1	70.3	9.5	10.2	30.7

The green backgrounds highlight the highest 1st percentiles, where the performance degrades the least during resizing.

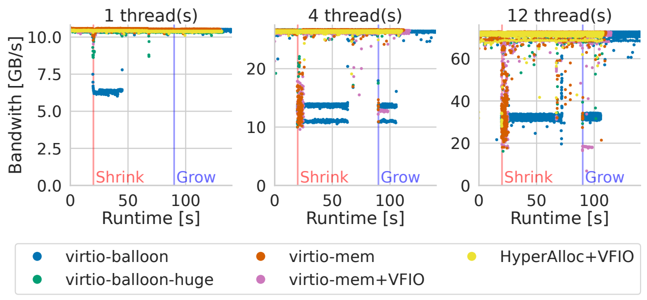


Figure 5. Memory bandwidth over time as reported by STREAM running on different numbers of threads.

5.4 Guest Performance Impact

In addition to raw speed, we also analyzed the impact of reclamation on the guest performance. To do so, we change the VM's memory limit while running memory- and CPU-intensive workloads. For comparability to previous work, we based our procedure on Hildenbrand and Schulz [23].

Experiment Procedure The VM is prepared by simulating a realistic workload: We execute 9 memory-intensive SPECrate2017 benchmarks [49], recreating the preparation step from [23]. For each benchmark, we start as many instances as needed to consume close to 19 GiB of memory and run it for 180 seconds. This preparation grows the VM to its maximum size and randomizes the guest's allocator state. After a 20 s cool-down period, we start the actual benchmark and decrease the VM's hard limit to 2 GiB. At 90 s, we increase it back to its original 20 GiB. While we largely follow the procedure of Hildenbrand and Schulz [23], we extended it in two ways: (1) Instead of running the benchmarks only single threaded, we also explore multi-threaded workloads (4 and 12 threads) to better understand different system loads. (2) While the original benchmark only shrank the VM, we also include a subsequent growing phase.

As baseline, we use the virtio-balloon configuration, but do not resize it (Tab. 2). However, we exclude it from the plots to improve readability. Since there were no significant differences between HyperAlloc with and without device passthrough, the plots only include the former.

Memory Bandwidth To simulate a memory-intensive task, we use a customized version of the STREAM [37] benchmark, which repeatedly measures the bandwidth of memcopy operations (≈1 GiB per operation). We modified the benchmark to only run one of its four measurements (Copy) and to export per-sample memcopy bandwidth rates. As STREAM's iteration time varies between thread counts, we chose the number of iterations for each thread configuration so that the slowest candidate took 140 s.

The scatter plots in Fig. 5 show the bandwidth of each iteration over time. To judge the impact of high-frequency resizing on latency-sensitive tasks, Tab. 2 contains 1st percentile bandwidths. Running the experiment inside a virtualized environment introduces some noise, particularly for larger thread counts. However, analysis of our baseline indicates that the influence on the 1st percentile is negligible compared to the actual observed performance degradation. While there are slight differences in memory bandwidth between candidates while idling (before 20 s and once resizing is complete), they are within run-to-run variance.

With STREAM running on a single thread, only shrinking via virtio-balloon significantly impacts the guest's performance due to its 4 KiB page granularity and subsequent communication overhead. Our virtio-balloon-huge implementation eliminates this overhead almost completely. While virtio-mem shows a negligible spike at 20 s, HyperAlloc does not show any measurable impact on performance. As a result, with HyperAlloc STREAM finishes ≈8.9 s faster compared to virtio-balloon. Apart from virtio-balloon, the 1st percentiles show no significant performance degradation.

On multiple threads, resizing the VM becomes increasingly noticeable. In addition to shrinking, virtio-balloon starts to cause slowdowns while growing the VM as well. Virtio-mem has a noticeable impact while shrinking, performing even worse than virtio-balloon for ≈10 s with lows reaching 31.9 GB/s. When unplugging memory, virtio-mem removes blocks in decreasing address order, requiring the guest OS to migrate used subblocks to other memory locations. Still, growing has no immediate effect on memory bandwidth. New memory blocks are plugged in, but no memory is preallocated on the host. However, when passing a device to the VM, the memory needs to be populated and pinned, resulting in an even larger performance degradation than virtio-balloon (1.7 ×). Virtio-balloon-huge outperforms virtio-mem but still shows a small impact while shrinking. Even under full system load, HyperAlloc does not have a significant impact on memory bandwidth, with its 1st percentile bandwidth being 2.3 and 2.2 times higher than virtio-balloon and virtio-mem. In contrast to the other evaluated solutions, reclaiming and returning memory from/to the guest does not involve fine granular guest-hypervisor communication or expensive memory migration.

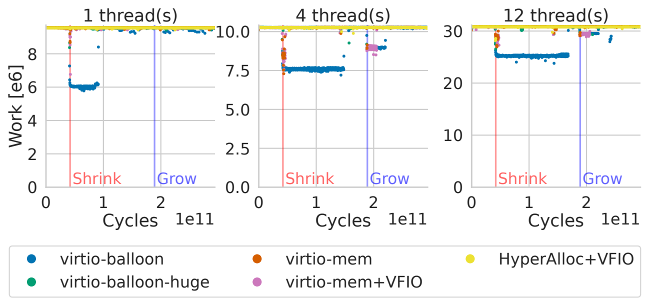


Figure 6. Aggregated *work* as measured by FTQ for different numbers of threads.

CPU Utilization To assess the impact on CPU-intensive workloads, we employed the Fixed-Time-Quantum (FTQ) [29] benchmark. It samples the amount of *work* performed by a CPU thread within a fixed time interval by repeatedly incrementing a counter. Usually, each thread is measured independently, but to present the data more clearly, we aggregate the work of all threads. This approach could introduce inaccuracies due to desynchronization. In practice, however, the sampling interval was sufficiently large to ensure that the overall noise was negligible for all our experiments. We sampled 1096 times at 2^{28}-cycle intervals (≈140 s total).

The scatter plots in Fig. 6 show the amount of work performed per time interval. For easy comparison, we have chosen the number of samples so that the runtime is equal to the STREAM runs. The impact on CPU performance appears closely related to memory bandwidth, though far less noticeable, as evidenced by the 1st percentiles (Tab. 2). Virtio-balloon causes the most significant performance degradation, with shrinking being more expensive than growing. Both virtio-mem and virtio-balloon with huge pages have a negligible impact at higher thread counts, even though the duration is much shorter than in the previous experiment. Notably, virtio-mem with device passthrough generates no extra CPU overhead, regardless of memory pinning. HyperAlloc has no significant effect. Even under full system load, its 1 percent lows are above the baseline. As a result, its minimum CPU performance is 23 and 6.8 percent higher than virtio-balloon and virtio-mem.

5.5 Automatic Soft Reclamation

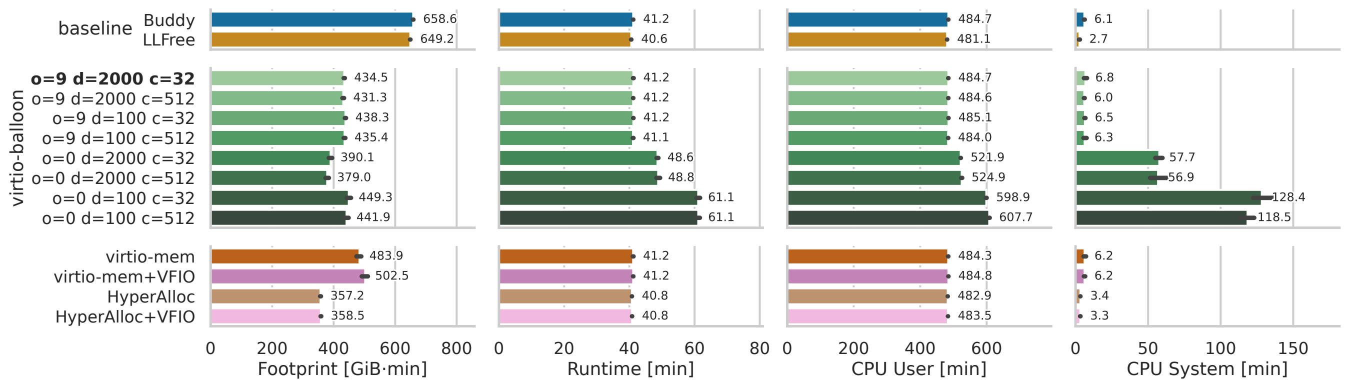
Continuous Integration (CI) jobs often require large and varying amounts of memory for short bursts of time. Build-farm VMs, which provide strict isolation, must accommodate peak memory demands regardless of job frequency. If we can deflate these VMs dynamically and efficiently, more VMs could run on the same physical host. We evaluate HyperAlloc's suitability for this scenario by compiling Clang 16.0.0. To increase memory pressure, we reduce the VM's memory to 16 GiB for our measurements, which is the observed maximum of the workload. As the automatic reclamation mechanisms were designed to have no significant performance impact, we focus on the memory footprint (in GiB·min), which is calculated from the *resident set size (RSS)* of the QEMU process, representing its actually consumed memory (sampled at 1 Hz). Similar metrics are also used by cloud providers (e.g., AWS Lambda) to price memory usage.

HyperAlloc and virtio-balloon (with free-page-reporting) can automatically reclaim memory. As virtio-mem lacks an automatic reclaim mechanism, we *simulated* one: We track the number of free huge pages in the guest and (un-)plug memory with a granularity of 1 GiB with a frequency of 1 s. Frequency and granularity were hand tuned for this benchmark to minimize the overhead while still avoiding out-of-memory errors. If directly integrated into virtio-mem, automatic reclamation would most likely be more efficient, but our simulation already shows that it is also limited by huge-page availability, like virtio-balloon.

Fig. 7 compares the Buddy and LLFree allocator baselines against virtio-balloon's free-page-reporting, virtio-mem and HyperAlloc. The baselines, which statically use 16 GiB for the entire runtime, have the highest memory footprint. LLFree's footprint is slightly smaller because of its shorter runtime. The different auto-reclamation techniques can reduce the memory footprint from 24 to 45 percent, usually without noticeable runtime overheads (which is by design). HyperAlloc has the lowest memory footprint, followed by the different configurations of virtio-balloon and lastly the simulated virtio-mem mechanism.

The rightmost columns contain the total CPU times of all 12 threads of the QEMU processes, separated into user and system (page faults, KVM exits, and other syscalls). We see that system and user time of LLFree-based benchmarks is shorter than Buddy-based ones. An in-depth investigation revealed that those runs incur about half as much EPT faults and TLB misses. LLFree's contiguous allocation pattern appears to be more suitable for VMs that are often backed by huge pages. As the user time also includes the VM workload, we can infer the reclamation overheads by comparing it to their respective baselines. The overhead of HyperAlloc (0.51 ± 0.18%) is a minimally higher than virtio-balloon (default: 0.15 ± 0.42%). This is because HyperAlloc reclaims more memory, leading to more work in the QEMU process and more madvise syscalls.

The default configuration for virtio-balloon (Fig. 7 in bold) reduces the memory footprint by 34 percent. We tuned the configuration parameters of virtio-balloon to see if we can increase its efficiency. These parameters include the REPORTING_ORDER (o), denoting the size of reclaimed memory blocks (we used 4 KiB and 2 MiB), the REPORTING_DELAY (d), specifying the delay between the freeing of chunk of the specified order and the subsequent reclamation (from 2 s to 100 ms), and the REPORTING_CAPACITY (c), denoting the



	Footprint [GiB·min]	Runtime [min]	CPU User [min]	CPU System [min]
baseline Buddy	658.6	41.2	484.7	6.1
baseline LLFree	649.2	40.6	481.1	2.7
o=9 d=2000 c=32	434.5	41.2	484.7	6.8
o=9 d=2000 c=512	431.3	41.2	484.6	6.0
o=9 d=100 c=32	438.3	41.2	485.1	6.5
o=9 d=100 c=512	435.4	41.1	484.0	6.3
o=0 d=2000 c=32	390.1	48.6	521.9	57.7
o=0 d=2000 c=512	379.0	48.8	524.9	56.9
o=0 d=100 c=32	449.3	61.1	598.9	128.4
o=0 d=100 c=512	441.9	61.1	607.7	118.5
virtio-mem	483.9	41.2	484.3	6.2
virtio-mem+VFIO	502.5	41.2	484.8	6.2
HyperAlloc	357.2	40.8	482.9	3.4
HyperAlloc+VFIO	358.5	40.8	483.5	3.3

The average of 6 runs per candidate is displayed.
For virtio-balloon, we compare different parameters with the **default configuration in bold**.

Figure 7. Memory footprint, total runtime, and user/system CPU times of the QEMU process for a clang compilation.

size of the reclaim buffer that is sent to the host (from 32 to 512). If the mechanism uses huge pages (o=9) we see no significant difference between the delay (d) and capacity (c) values. Only for 4 KiB pages (o=0), they have a noticeable effect. Two configurations can even further reduce the memory footprint (by 42% for d=2 s and c=512). However, they also increase the runtime by 19 percent. Similarly, the user and system CPU times are significantly higher.

In-depth Analysis To better understand the mechanisms, we expand on the Clang benchmark: On the time axis, we wait for 200 s after the build finished and run `make clean` to remove any build artifacts; after another 200 s, we drop the guest's page cache to see how much memory can be reclaimed at best. We sample four memory-usage metrics (see Fig. 8) with a frequency of 1 Hz: (1) The memory consumed by (partially) used *huge* pages in the guest allocator. (2) The memory consumed by actually allocated *small* pages (4 KiB) in the guest. The difference between *small* and *huge* is an indicator of the degree of fragmentation within the page allocator. (3) The size of the guest's page *cache*. (4) The amount of assigned *VM memory* (RSS) which reclamation reduces. As *small* and *cached* are defined by the workload, they are expected to remain the same across all candidates. In the best-case scenario, the assigned VM memory, the used *huge* pages, and the allocated *small* pages would all be equal, showing perfect memory efficiency.

Fig. 8 shows the results of a single run for virtio-balloon (o=9, d=2000, c=32) and HyperAlloc. As expected, the size of the page cache and the guest's memory utilization are consistent. However, one minor difference in the page cache size can be observed at around 29 min: The VM with virtio-balloon reaches its hard memory limit, resulting in page cache eviction. As HyperAlloc has less fragmentation, it does not suffer from this. Over the entire runtime, HyperAlloc's *VM memory* follows the guest's memory consumption

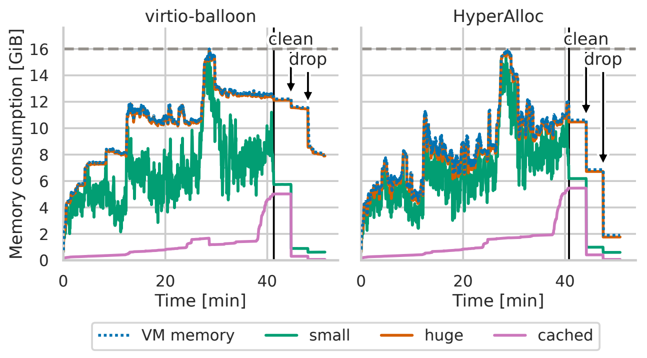


Figure 8. Clang compilation with virtio-balloon's free-page-reporting (default) and HyperAlloc's automatic soft-reclamation.

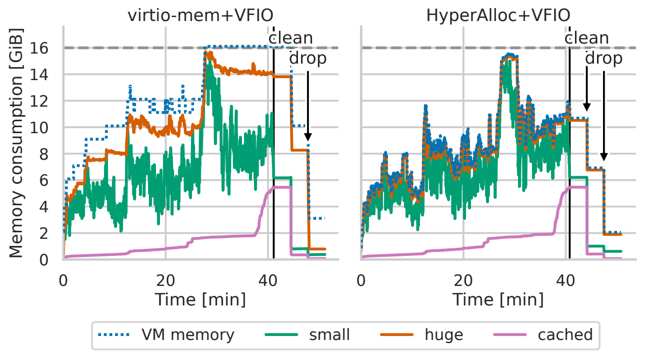


Figure 9. Clang compilation with HyperAlloc and virtio-mem with VFIO-based DMA safety.

(*small*) much more closely due to LLFree's efficient fragmentation avoidance. This decreases the memory footprint by 17 percent compared to virtio-balloon, without noticeable runtime costs.

By the end of compilation, the page cache occupies a substantial part of the guest's total memory. This prevents many poorly utilized huge pages from being freed and reclaimed.

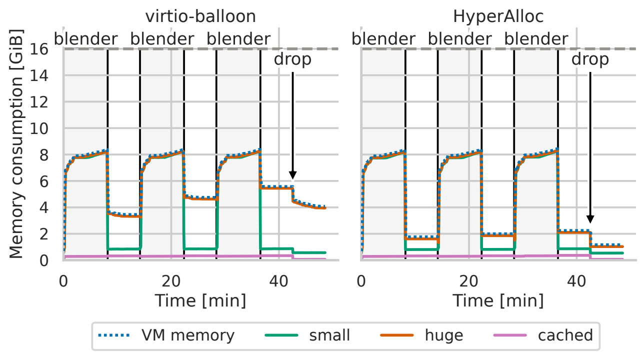
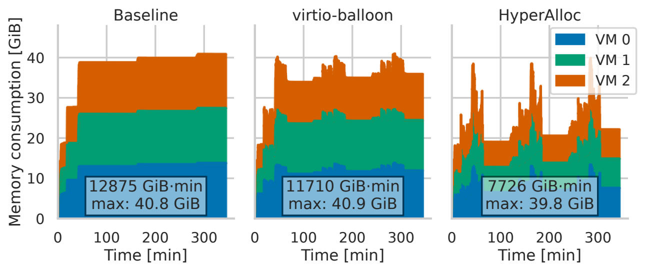
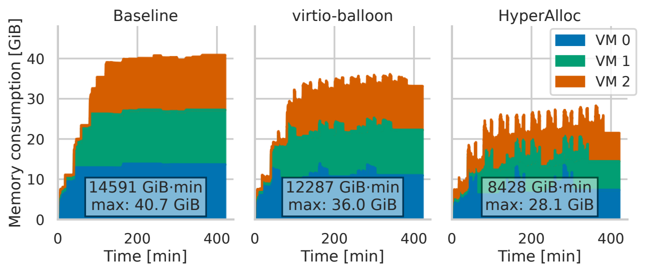


Figure 10. Repeated SPEC2017 blender runs with auto deflation.



(a) Concurrent clang compilations.



(b) Offsetted clang compilations.

Figure 11. Memory footprint of multiple VMs (colors).

Running `make clean`, thereby removing all build artifacts, reduces the cache size significantly. As a result, HyperAlloc can shrink the VM by 3.8 GiB. In contrast, virtio-balloon only reduces the size by 0.7 GiB, due to internal fragmentation of the buddy allocator. Even when *dropping* the entire cache, virtio-balloon only decreases the VM size to 8 GiB compared to HyperAlloc's 1.9 GiB. Generally, in these file-intensive workloads, we see that the page cache has a major impact on the memory footprint. We discuss its role further in Sec. 6.

Regarding DMA-safety, we compared virtio-mem and HyperAlloc in Fig. 9, both using VFIO for device passthrough. Even though both mechanisms have no significant runtime costs, virtio-mem has a 39.8 percent higher memory footprint than HyperAlloc. For virtio-mem, we track the number of free *huge* pages and resize accordingly. We also tried to follow the free base pages, which leads to more aggressive reclamation. However, in this case, virto-mem has to compact and migrate memory, which turned out to be too slow, and virtio-mem was unable to resize the VM fast enough to prevent OOMs. Therefore, our virtio-mem-based reclamation is still limited by the availability of huge pages, which is a significant advantage of basing HyperAlloc on the LLFree allocator. Virtio-mem without VFIO is 3.7 percent more efficient because it does not pre-populate memory. For HyperAlloc, the additional overhead to maintain the IO page tables is negligible.

Repeated Workloads Another common use case for VMs is (micro-) services that are executed on demand or periodically. Here, the VMs might idle for significant amounts of time between runs. Even though the host can easily detect idle vCPUs and schedule accordingly, detecting *idle* memory is not so simple and an ideal use-case for memory reclamation. We simulated such a repeated workload with idle periods using the Blender benchmark from SPEC2017. We executed three consecutive runs with 4 min idle time in between. The page cache was dropped once at the end, again to see its impact on the VM's memory consumption.

The two candidates in Fig. 10 perform roughly the same while Blender is running due to its static allocation behavior. However, a huge difference can be observed in between runs. After the first iteration, HyperAlloc reduces memory consumption by 49 percent compared to virtio-balloon. This difference increases to 60 percent after the third iteration. Overall, this leads to an overall reduction in memory footprint from 300 GiB·min to 234 GiB·min. This difference becomes even more pronounced if the idle times increase.

After dropping the page cache, the memory consumption drops to 1.17 GiB for HyperAlloc and 4.08 GiB for virtio-balloon. This large difference can be attributed to LLFree's better fragmentation characteristics and shows that HyperAlloc allows for greater elasticity of VMs.

5.6 Multiple VMs

Memory reclamation becomes more relevant if multiple VMs compete for memory resources. The reclaimed memory could be used to run additional VMs, increasing the overall utilization of the host. Hence, we also compare the memory efficiency of virtio-balloon and HyperAlloc in a multi-VM setup. The workload is, again, the compilation of clang 16.0.0, executed in parallel on three VMs. Each compiles clang three times with a 2h delay in between. Each VM is configured with 16 GiB of host memory, resulting in a total provisioning of 48 GiB of host memory. Fig. 11 shows the accumulated memory footprint of all three VMs over time.

For Fig. 11a (top), we started the workloads simultaneously, so that the peak memory consumptions is reached at the same time in all three VMs. Without ballooning, this sums up to a peak of 40.8 GiB. This mimics a worst-case scenario with respect to memory reclamation: While ballooning can reduce the peak memory times, it cannot reduce the peak memory demand, even though the overall memory footprint (GiB·min) is reduced by 9.1 / 40 percent with virtio-balloon / HyperAlloc, respectively. The important point here is that even in such a worst-case scenario, HyperAlloc does not increase the run time and peak memory demand.

Fig. 11b (bottom) shows the same setup, but peak memory consumptions of the three VMs is now offsetted by 40 min. This is the best case for memory reclamation[5] Here the overall peak memory demand drops from 40.74 GiB to 35.98 GiB for virtio-balloon and to 28.11 GiB for HyperAlloc. In this case, virtio-balloon's free-page-reporting would reclaim enough memory to run one additional VM within the 48 GiB of available memory, while HyperAlloc would even make it possible to run two additional VMs.

6 Discussion

VProbe Due to the discussed availability issues (see Sec. 5.1), we could not compare HyperAlloc quantitatively to the seemingly similar VProbe [55] approach. VProbe avoids explicit communication, achieves DMA-safe auto deflation, and gives the hypervisor access to the guest state. However, VProbe aims to be transparent for the guest and tracks the guest-side page-frame allocations only *indirectly* by write-protecting the guest's page-frame metadata (`struct page`). Thereby, VProbe tracks the same guest events as HyperAlloc but relies on the side effects of these events, which has two disadvantages: (1) VProbe's coupling to the guest allocator is fragile when (newer) guest kernels perform different steps on allocation, resulting in unreliable allocation detection. (2) The hypervisor has to back the page-frame metadata with host-physical base frames instead of huge frames, inducing higher TLB pressure. In contrast, HyperAlloc's explicit *install* hypercall evolves with the guest kernel code and requires no fine-grained backing of `struct page`.

Adoption in Production HyperAlloc requires the user to replace the guest's page allocator, so our co-design approach can be considered as more intrusive as having only an extra balloon or hotplug driver within the guest. Therefore, the question arises whether HyperAlloc is a viable solution for production environments – and whether customers should switch to HyperAlloc. We expect that economic reasoning will become a good argument: In the longer term, IaaS will follow the trend of FaaS [46] and start billing memory by the second, giving customers a monetary incentive to give back unused memory immediately. Until now, memory often becomes a stranded asset [32] for cloud providers when they are confronted with a CPU-intense workload mix, as they cannot shift memory between VM hosts. However, the emerging CXL [16] technology allows building disaggregated memory pools [32], making physical memory not only an expensive [35] but also more valuable commodity, as unused memory could be redistributed among the complete rack. Compared to existing techniques, HyperAlloc achieves higher reclamation rates at basically no interference (Sec. 5.5), making it an ideal feature for the disaggregated cloud.

[5]The hypervisor could enforce this by preempting VMs when memory runs out. VM preemption and swapping is beyond the scope of this paper, but we discuss possible combinations in Sec. 6.

Concept Generalization Since HyperAlloc requires offset-addressable access to per-frame data, integration with other guest allocators is challenging, as they usually rely on lock-based synchronization and pointer-linked data. Letting the hypervisor directly participate in those guest protocols poses a safety and security risk. Nevertheless, if host and guest agree on an auxiliary memory-mapped interface to exchange A and E, HyperAlloc is applicable.

More generally, we believe that lock-free write access to the guest state is a promising direction to improve resource management in IaaS settings without introducing (much) interference and latency variations. For example, a logical next step could be to also expose the page cache to HyperAlloc, which could then shrink the VM from the outside. Although this requires rethinking even more kernel components, the resulting lock-free kernel structures often prove to be more scalable than the existing mechanisms [56].

Beyond Memory Reclamation Our prototype can scale a VM between its initial size and the currently allocated memory. While we share the later boundary with many reclamation techniques [23, 53, 55], virtio-mem [55] can grow the VM beyond the initial size. HyperAlloc could also support this, either by hotplugging integration or by starting with a large guest-physical memory but low hard limit.

Similarly, HyperAlloc does not address the case where the accumulated memory consumption of guests temporarily exceeds the available host memory (e.g., during short peaks). Here, hypervisors usually fallback to swapping. Still, HyperAlloc, because of its better memory efficiency, is expected to cause fewer and shorter out-of-memory situations (see Sec. 5.6). Furthermore, HyperAlloc could also enable better swapping strategies for VMs [10, 47], as the tree index entries contain the allocation type (see Sec. 4.2). Additionally, with the six remaining area-entry bits, the guest could expose even more useful information about data-filled frames (e.g., hotness).

If fine-grained memory pricing models are just over the horizon, we have to develop efficient guest methods that actively react to memory-price pressure (i.e., executed via auctioning [6]). For example, with a price tag at each frame, we have an objective measure to decide if starting memory compaction is actually worth it. Suddenly, actively shrinking the page cache instead of caching as much as possible could make economic sense.

7 Related Work

Dynamic paging has been a challenging topic for OS developers for a long time, especially regarding memory reclamation [17, 25, 27], TLB invalidation [7, 9, 30], and the fragmentation of huge pages [21, 40, 41, 50, 56]. These challenges become even more relevant in combination with virtual machines, where we have an additional interaction and EPT/NPF walks are more expensive [4, 12, 28, 38, 54].

Memory Reclamation The problem of detecting idle memory and estimating working sets is more difficult for the OS [17, 18, 57] than detecting idle CPUs. This, again, is even more complicated for a hypervisor with even less information about the running workloads [26, 38, 53]. Consequently, transparent VM deflation techniques, like swapping [10, 22, 42, 53] or content-based sharing [15, 53], face challenges to determine what to reclaim. Cooperative deflation techniques, like ballooning [24, 45, 53], hotplugging [23, 44, 45], or transient memory [20, 33], try to solve the reclamation-information deficit by *indirectly* interacting with the guest's OS frame allocator. However, the interaction via in-guest proxy drivers is costly [24, 55]. By integrating reclamation *directly* into the guest's frame allocator, HyperAlloc drastically reduces this overhead and enables the guest to improve its allocation policy based on the reclamation state.

DMA safety With the emergence of the IOMMU [1–3], the OS got another virtual memory component to keep in sync [13]. Moreover, most devices cannot [31, 55] trigger IO page faults, which are, however, necessary for most deflation techniques [24, 38, 45, 53]. Only a few techniques [23, 55] have been designed for DMA safety, while the others require further IOMMU virtualization [8, 51], which comes with its own costs for tracking DMA buffers and invalidating IOTLBs. Still, HyperAlloc could be combined with IOMMU virtualization to reduce the VFIO overhead further.

Resource Orchestration Deciding how to manage these different deflation techniques at a large scale is a topic on its own. Several policies and heuristics have been proposed for VM monitoring [26, 53], resource distribution [19, 34, 39, 48, 52], and market-like pricing models [6, 36]. They usually combine transparent and cooperative deflation and sometimes even interface with applications [14, 48]. With HyperAlloc, these orchestration mechanisms could monitor memory utilization more precisely and reclaim memory faster with less latency.

8 Conclusion

For the hypervisor, memory is, until now, a "viscous" resource that is hard to add to the guest and even harder to reclaim. But with the emergence of CXL and disaggregated memory pools, both cloud provider and customer get a monetary incentive to de/inflate VMs faster and more frequently. However, existing techniques often fall short with device passthrough or induce disruptive overheads and latency spikes. For example, virtio-mem results in a throughput disruption of up to −73 percent for the STREAM benchmark.

With HyperAlloc, we propose a novel VM memory reclamation technique based on sharing the guest's page-frame allocator with the hypervisor. Without a mode switch, we can, thereby, mark frames as allocated or reclaimed within the guest, allowing its allocator to prefer regions already backed by host-physical memory. Still, our bilateral state management is safe and secure as we build upon LLFree, an existing lock-free page-frame allocator with good scalability and huge-frame fragmentation properties. HyperAlloc shrinks, without measurable disruption, the hard limit of a virtual machine faster (362× *virtio-balloon*, 10× *virtio-mem*). With its automatic reclamation, we reclaim 17 percent more memory than virtio-balloon's free-page reporting for a realistic workload, resulting in a tighter resource assignment.

Acknowledgments

We thank our anonymous reviewers and our shepherd Reto Achermann for their helpful and constructive comments. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 501887536.

References

[1] 2022. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.* https://cdrdv2.intel.com/v1/dl/getContent/671200

[2] 2024. *AMD64 Architecture Programmer's Manual Volume 2: System Programming.* https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf

[3] 2024. *Arm System Memory Management Unit Architecture Specification - Version 3.* https://developer.arm.com/documentation/ihi0070/latest

[4] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/1168857.1168860

[5] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. 2014. The rise of RaaS: the resource-as-a-service cloud. *Commun. ACM* 57, 7 (jul 2014), 76–84. https://doi.org/10.1145/2627422

[6] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. 2014. Ginseng: market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Salt Lake City, Utah, USA) *(VEE '14)*. Association for Computing Machinery, New York, NY, USA, 41–52. https://doi.org/10.1145/2576195.2576197

[7] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*. 27–39.

[8] Nadav Amit, Muli Ben-Yehuda, IBM Research, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: Efficient IOMMU Emulation. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. USENIX Association, Portland, OR. https://www.usenix.org/conference/usenixatc11/viommu-efficient-iommu-emulation

[9] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't shoot down TLB shootdowns!. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. 1–14. https://doi.org/10.1145/3342195.

3387518

[10] Nadav Amit, Dan Tsafrir, and Assaf Schuster. 2014. VSwapper: a memory swapper for virtualized environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 349–366. https://doi.org/10.1145/2541940.2541969

[11] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (apr 2010), 50–58. https://doi.org/10.1145/1721654.1721672

[12] 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 19th ACM Symposium on Operating Systems Principles (SOSP '03) (ACM SIGOPS Operating Systems Review, Vol. 37, 5)*. ACM Press, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[13] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. 2007. The price of safety: Evaluating IOMMU performance. In *Proceedings of the Linux Symposium*. 9–20.

[14] Callum Cameron, Jeremy Singer, and David Vengerov. 2015. The judgment of forseti: economic utility for dynamic heap sizing of multiple runtimes. In *Proceedings of the 2015 International Symposium on Memory Management* (Portland, OR, USA) *(ISMM '15)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/2754169.2754180

[15] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. 2011. An Empirical Study on Memory Sharing of Virtual Machines for Server Consolidation. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*. USENIX Association, USA, 244–249. https://doi.org/10.1109/ISPA.2011.31

[16] Compute Express Link Consortium, Inc. 2020. *CXL Specification, Revision 2.0.*

[17] P.J. Denning. 1980. Working Sets Past and Present. *IEEE Transactions on Software Engineering* SE-6, 1 (1980), 64–84. https://doi.org/10.1109/TSE.1980.230464

[18] Peter J. Denning. 1967. The working set model for program behavior. In *Proceedings of the First ACM Symposium on Operating System Principles (SOSP '67)*. Association for Computing Machinery, New York, NY, USA, 15.1–15.12. https://doi.org/10.1145/800001.811670

[19] Alexander Fuerst, Ahmed Ali-Eldin, Prashant Shenoy, and Prateek Sharma. 2020. Cloud-scale VM-deflation for Running Interactive Applications On Transient Servers. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/3369583.3392675

[20] Luis A. Garrido, Rajiv Nishtala, and Paul Carpenter. 2019. SmarTmem: Intelligent Management of Transcendent Memory in a Virtualized Server. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 911–920. https://doi.org/10.1109/IPDPSW.2019.00151

[21] Mel Gorman and Patrick Healy. 2008. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management - ISMM '08*. ACM Press, Tucson, AZ, USA, 41. https://doi.org/10.1145/1375634.1375641

[22] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. 1999. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) *(SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 154–169. https://doi.org/10.1145/319151.319162

[23] 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* (Virtual, USA) *(VEE 2021)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3453933.3454010

[24] 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* (Alexandria, Virginia, USA) *(MEMSYS '18)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/3240302.3240420

[25] Song Jiang and Xiaodong Zhang. 2001. Adaptive Page Replacement to Protect Thrashing in Linux. In *5th Annual Linux Showcase & Conference (ALS 01)*. USENIX Association, Oakland, CA. https://www.usenix.org/conference/als-01/adaptive-page-replacement-protect-thrashing-linux

[26] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2006. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/1168857.1168861

[27] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner. 1962. One-Level Storage System. *IRE Transactions on Electronic Computers* EC-11, 2 (April 1962), 223–235. https://doi.org/10.1109/TEC.1962.5219356

[28] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, Vol. 1. Dttawa, Dntorio, Canada, 225–230.

[29] Lawrence Livermore National Laboratory. 2014. CORAL Benchmark Codes. https://asc.llnl.gov/coral-benchmarks, visited 2024-05-04.

[30] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. In *Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data* (Seattle, WA, USA). ACM, New York, NY, USA. https://doi.org/10.1145/3588687

[31] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. 2017. Page Fault Support for Network Controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. 449–466. https://doi.org/10.1145/3037697.3037710

[32] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23), Volume 2* (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3575693.3578835

[33] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. 2009. Transcendent memory and linux. In *Proceedings of the Linux Symposium*. Citeseer, 191–200.

[34] Sunilkumar S. Manvi and Gopal Krishna Shyam. 2014. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. *Journal of Network and Computer Applications* 41 (2014), 424–440. https://doi.org/10.1016/j.jnca.2013.10.004

[35] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of*

the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23), Volume 3 (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[36] Hasan Al Maruf, Yuhong Zhong, Hongyi Wang, Mosharaf Chowdhury, Asaf Cidon, and Carl Waldspurger. 2023. Memtrade: Marketplace for Disaggregated Memory Clouds. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 2, Article 41 (may 2023), 27 pages. https://doi.org/10.1145/3589985

[37] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* 2 (Dec. 1995), 19–25.

[38] Debadatta Mishra and Purushottam Kulkarni. 2018. A survey of memory management techniques in virtualized systems. *Computer Science Review* 29 (2018), 56–73. https://doi.org/10.1016/j.cosrev.2018.06.002

[39] Germán Moltó, Miguel Caballer, and Carlos de Alfonso. 2016. Automatic memory-based vertical elasticity and oversubscription on cloud platforms. *Future Generation Computer Systems* 56 (2016), 1–10. https://doi.org/10.1016/j.future.2015.10.002

[40] Ashish Panwar, Naman Patel, and K. Gopinath. 2016. A Case for Protecting Huge Pages from the Kernel. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, Hong Kong) *(APSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 15, 8 pages. https://doi.org/10.1145/2967360.2967371

[41] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[42] Jan S. Rellermeyer, Maher Amer, Richard Smutzer, and Karthick Rajamani. 2018. Container Density Improvements with Dynamic Memory Extension using NAND Flash. In *Proceedings of the 9th Asia-Pacific Workshop on Systems* (Jeju Island, Republic of Korea) *(APSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3265723.3265740

[43] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.* 42, 5 (jul 2008), 95–103. https://doi.org/10.1145/1400097.1400108

[44] Joel Schopp, Dave Hansen, Mike Kravetz, Hirokazu Takahashi, Toshihiro Iwamoto, Yasunori Goto, Hiroyuki Kamezawa, Matt Tolentino, and Bob Picco. 2005. Hotplug memory redux. In *Proceedings of the Linux Symposium*. 151.

[45] Joel H Schopp, Keir Fraser, and Martine J Silbermann. 2006. Resizing memory with balloons and hotplug. In *Proceedings of the Linux Symposium*, Vol. 2. 313–319.

[46] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1063–1075. https://doi.org/10.1145/3352460.3358296

[47] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. 2019. Resource Deflation: A New Approach For Transient Resource Reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 33, 17 pages. https://doi.org/10.1145/3302424.3303945

[48] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. 2019. Resource Deflation: A New Approach For Transient Resource Reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 33, 17 pages. https://doi.org/10.1145/3302424.3303945

[49] SPEC. 2022. SPEC CPU® 2017. https://www.spec.org/cpu2017/, visited 2024-05-03.

[50] Matthias Springer and Hidehiko Masuhara. 2019. Massively parallel GPU memory compaction. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*. 14–26.

[51] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. 2020. coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 479–492. https://www.usenix.org/conference/atc20/presentation/tian

[52] Manohar Vanga, Arpan Gujarati, and Björn B. Brandenburg. 2018. Tableau: a high-throughput and predictable VM scheduler for high-density workloads. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 28, 16 pages. https://doi.org/10.1145/3190508.3190557

[53] Carl A. Waldspurger. 2003. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.* 36, SI (dec 2003), 181–194. https://doi.org/10.1145/844128.844146

[54] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. 2011. Selective hardware/software memory virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Newport Beach, California, USA) *(VEE '11)*. Association for Computing Machinery, New York, NY, USA, 217–226. https://doi.org/10.1145/1952682.1952710

[55] Yaohui Wang, Ben Luo, and Yibin Shen. 2023. Efficient Memory Overcommitment for I/O Passthrough Enabled VMs via Fine-grained Page Meta-data Management. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 769–783. https://www.usenix.org/conference/atc23/presentation/wang-yaohui

[56] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. 2023. LLFree: Scalable and Optionally-Persistent Page-Frame Allocation. In *2023 USENIX Annual Technical Conference (USENIX '23)*. USENIX Association, Boston, MA, 897–914. https://www.usenix.org/conference/atc23/presentation/wrenger

[57] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, USA) *(ASPLOS XI)*. Association for Computing Machinery, New York, NY, USA, 177–188. https://doi.org/10.1145/1024393.1024415

A Artifact Appendix

A.1 Abstract

Our artifact is packaged as a docker container and contains the necessary tools and resources required to evaluate our HyperAlloc VM reclamation. All of our benchmarks, results, and meta information about hardware and software configuration are open source. This also includes the figures from the paper. The only benchmark we could not release is SPEC CPU 2017, but we included a step-by-step guide on how to install and use it. Our 6 benchmarks evaluate the latency, performance impact (memory bandwidth, CPU performance), and memory efficiency of HyperAlloc compared to virtio-balloon and virtio-mem.

A.2 Description & Requirements

A.2.1 How to access. The artifact is packaged in a Docker image, which is hosted on the GitHub package registry ghcr.io/luhsra/hyperalloc_ae and 10.5281/zenodo.14917769. All benchmarks are open source at the following repositories.

- hyperalloc-bench: Benchmarking scripts and results. This repository also includes the documentation for the artifact evaluation and the Dockerfile.
- hyperalloc-linux: Modified Linux kernel
- hyperalloc-qemu: Modified QEMU monitor
- linux-alloc-bench: Kernel module for benchmarking the page allocator
- hyperalloc-stream: Memory bandwidth benchmark
- hyperalloc-ftq: FTQ CPU work benchmark
- llfree-c: C-based implementation of the LLFree page allocator
- llfree-rs: Rust-based implementation of the LLFree page allocator, including some micro benchmarks, like `bench/src/bin/write.rs`.

A.2.2 Hardware dependencies. The benchmarks require an x86 based system. Most benchmarks need 12 cores and 32 GiB RAM, only the `multivm` benchmark requires 24 cores and 48 GiB RAM. We recommend disabling SMT (Hyper-Threading), setting a fixed CPU frequency, and disabling powersaving modes for more stable results. We also have a few benchmarks that require VFIO device passthrough. Thus, the system has to have an IOMMU and device group that can be passed into the VMs, as described below.

A.2.3 Software dependencies. The benchmarks require a Linux-based system with KVM (tested on Debian 12 and Fedora 41).

A.2.4 Benchmarks. The container is mostly self-contained. One benchmark requires the SPEC CPU 2017 1.1.9 benchmark suite, which is not public.

A.3 Set-up

The general process is to download the container, connect to it with ssh, and run the benchmarks.

```
docker pull ghcr.io/luhsra/hyperalloc_ae:latest
# Give yourself access to /dev/kvm
sudo chown $USER /dev/kvm
# Start the container
docker run --network=host --device=/dev/kvm \
  -e AUTHORIZED_KEYS="$(cat ~/.ssh/id_rsa.pub)" \
  --rm ghcr.io/luhsra/hyperalloc_ae
# Connect to the container
ssh -i ~/.ssh/id_rsa -p2222 user@localhost
```

After connecting to the container, you can execute the `inflate` benchmark (which takes about 20 min) to test if everything works.

```
# (inside the container)
# you might again have to give yourself access to kvm
sudo chown $USER /dev/kvm
cd hyperalloc-bench
source venv/bin/activate
./run.py bench-plot -b inflate --fast
```

The hyperalloc-bench repo contains additional scripts in the `artifact-eval` directory.

A.3.1 VFIO Device Passthrough. There are a few benchmarks that use device passthrough. However, if you do not have a system supporting device passthrough, you can skip this step and the benchmarks by omitting the `-vfio <device>` argument for the `run.py` runner.

The scripts/bind_vfio.py script can be used to bind IOMMU groups to VFIO. Executing it (outside the docker container), shows you all IOMMU groups and their corresponding devices. You can then enter a group number to bind it to VFIO. Note that if you bind an IOMMU group, all devices of this group cannot be used by the host anymore. Also, you will need a device ID from the group (like `08:00.0`) later for the benchmark runner. If the script shows you no devices, you might have to enable the IOMMU on the host (e.g., with `intel_iommu=on`).

The next step is to pass VFIO into the container and allow the container to lock memory:

```
# Stop any running containers before this
docker run --network=host --device=/dev/kvm \
  --device /dev/vfio \
  --ulimit memlock=53687091200:53687091200 \
  -e AUTHORIZED_KEYS="$(cat ~/.ssh/id_rsa.pub)" \
  --rm ghcr.io/luhsra/hyperalloc_ae
```

A.4 Evaluation workflow

A.4.1 Major Claims. In the paper, we use the following benchmarks:

- `inflate` (section 5.3): Inflation/deflation latency
- `stream` (section 5.4): STREAM – memory bandwidth
- `ftq` (section 5.4): FTQ – CPU work
- `compiling` (section 5.5): Clang compilation with auto VM inflation
- `multivm` (section 5.6): Compiling clang on multiple concurrent VMs

- `blender` (section 5.5): SPEC CPU 2017 blender benchmark

The `inflate` benchmark measures the latency for shrinking and growing VMs. In the paper we claim that HyperAlloc is significantly faster that all other techniques for reclaiming memory (touched and untouched) and returning memory. When returning and installing (accessing) memory to the VM, HyperAlloc is as fast as virtio-mem and slightly slower than virtio-balloon-huge.

The `stream` and `ftq` benchmarks measure the impact of VM resizing on the memory bandwidth and CPU performance of the guest. We claim that HyperAlloc has no measurable impact, other than virtio-mem and especially virtio-balloon.

The `compiling` benchmark evaluates the efficiency of automatic memory reclamation for a clang compilation, a workload with a highly fluctuating memory consumption. We claim that HyperAlloc has a smaller memory footprint than virtio-balloon and virtio-mem, without runtime overheads.

The `multivm` benchmark has been added in the shepherding phase and compares the memory footprint and peak memory consumption of virtio-balloon and HyperAlloc on multiple VMs. We claim that when the peak memory consumptions of the VMs do not coincide, virtio-balloon's free-page-reporting reclaims enough memory to run a single additional VM within the 48 GiB of available memory, and HyperAlloc even two additional VMs.

The `blender` benchmark shows a workload that temporarily consumes a lot of memory, which is executed three times. We claim that HyperAlloc can reclaim more memory between the workload runs than virtio-balloon. *This benchmark requires access to SPEC CPU 2017. The hyperalloc-bench repository contains information on how to install it.*

A.4.2 Experiments.

Generally, our benchmark setup has been automated as much as possible, so you only have to start the process and take a look at the resulting figures.

Optional Building the Artifacts [5 min human, 1h compute]: The container contains pre-built artifacts, so this can be skipped. Still, you can build our modified Linux kernels and QEMU binaries yourself with:

```
# (inside the container)
cd hyperalloc-bench
source venv/bin/activate
./run.py build
# (this takes about 1h)
```

Benchmarks [10 min human, compute depends on benchmark]: The benchmarks can be executed with the runner as shown below. The results are inside the container in `hyperalloc-bench/artifact-eval/<benchmark>`.

```
# (inside the container)
cd hyperalloc-bench
source venv/bin/activate
./run.py bench-plot -b <benchmark> --vfio <device-id>
```

Arguments of the benchmark runner:

- `bench-plot` can be replaced with `bench` or `plot` to only run the benchmarks or redraw the figures.
- If you want to run the additional `compile` benchmarks that evaluate the virtio-balloon parameters (Fig. 7), add the `-extra` argument. This extends the runtime by about 8h.
- The VFIO `<device-id>` has to be a device ID (like `08:00.0`) from the VFIO group passed to the container. You can omit this if you want to skip the VFIO benchmarks.

Benchmark compute time:

- `inflate` about 20 min
- `stream` about 15 min
- `ftq` about 15 min
- `compiling` about 6h and +8h with `-extra`
- `multivm` about 42h
- `blender` about 40 min

A.5 Exploring the Artifact

This section might be helpful if you want to explore the contents of the docker container more easily. The container has a running ssh server that allows you to create an `sshfs` mount. This requires `sshfs` to be installed on your system.

```
# (outside the container)
mkdir -p hyperalloc_ae
sshfs -p 2222 user@localhost:/home/user \
  -o IdentityFile=~/.ssh/id_rsa hyperalloc_ae
```

Now, you can explore the `hyperalloc_ae` directory with your file manager. The home directory contains the following subdirectories:

- hyperalloc-bench: Benchmarking scripts and results.
- hyperalloc-linux: Modified Linux kernel
- hyperalloc-qemu: Modified QEMU monitor
- hyperalloc-stream: Memory bandwidth benchmark
- hyperalloc-ftq: FTQ CPU work benchmark
- linux-alloc-bench: Kernel module for benchmarking the page allocator

A.6 Notes on Reusability

The parameters for the STREAM and FTQ benchmarks were chosen based on the memory bandwidth and CPU frequency of our test system. The results on your hardware might be skewed a bit. However, the overall trends should be similar.

If the stream benchmark terminates before growing the VM, you might have to increase the `-stream-iters` parameter (see `./run.py -h`).

The FTQ benchmark highly depends on the CPU frequency. Thus, the "shrink" and "grow" markers might not be aligned correctly in the plots. This is especially the case if the CPU frequency varies (frequency scaling, TurboBoost). However, the general trends (noticeable reductions in work for virtio-balloon and virtio-mem+VFIO) should be similar to the paper. Also, you can increase the runtime with the `-ftq-iters` parameter.