Not All Huge Pages Are Real: Detecting and Avoiding Misaligned Huge Pages in Public Cloud Virtual Machines

Mani Tofigh
Hofstra University
Hempstead, NY, USA
Columbia University
New York, NY, USA
mt4022@columbia.edu

Edward Guo
Columbia University
New York, NY, USA
eg3415@columbia.edu

Zhaoxi Shi
University of Rhode Island
Kingston, RI, USA
johnson.shi@uri.edu

Weiwei Jia
University of Rhode Island
Kingston, RI, USA
weiwei.jia@uri.edu

Xiaoning Ding
New Jersey Institute of Technology
Newark, NJ, USA
dingxn@njit.edu

Zirui Neil Zhao
University of Texas at Austin
Austin, TX, USA
neil.zhao@utexas.edu

Asaf Cidon
Columbia University
New York, NY, USA
asaf.cidon@columbia.edu

Jianchen Shan
Hofstra University
Hempstead, NY, USA
jianchen.shan@hofstra.edu

Abstract

Huge pages reduce the cost of translating virtual to physical memory addresses, but in a virtual machine (VM), a guest huge page can provide this benefit only if the host also backs it with a huge page (i.e., a *real* huge page). However, a guest huge page can be backed by host base pages, forming a *fake* huge page. In order to address this *misaligned huge page* problem, existing solutions require custom changes to hosts/hypervisors, an approach that is unavailable to the tenants of public cloud VMs.

We present **vHuge**, a system that can probe real and fake huge pages using two novel in-guest probing mechanisms: (i) by using the TLB hit/miss latency of accessing pages; and (ii) by analyzing the cache line prefetcher's behavior when lines are accessed on huge pages versus base pages, *without* requiring custom hardware or hypervisor support. vHuge further uses its knowledge of real/fake huge pages and provides real huge pages to new anonymous huge page allocations. When real huge pages are scarce, vHuge only backs *hot* application data with the scarce real huge pages.

Using vHuge, we measure huge page backings across 636 VMs on AWS, Azure, and Google Cloud and find fake huge pages at every provider; on Azure AMD VMs, about two-thirds of sampled guest huge pages turn out to be fake. Using a similar setup, we evaluate vHuge where the host backs only 30% of the VM's memory with huge pages. vHuge improves the performance of a wide variety of applications by 20.45% on average (up to 97.1%) and reduces TLB misses by 49.78% on average (up to 88.01%), with an average overhead of 0.785%.

CCS Concepts

• **Computer systems organization** → *Cloud computing*; • **Software and its engineering** → **Virtual machines**.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
SoCC '26 Vol. 2, Singapore, Singapore
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2816-7/2026/11
https://doi.org/10.1145/3828161.3857854

Keywords

huge pages, virtualization, address translation, public cloud

ACM Reference Format:
Mani Tofigh, Edward Guo, Zhaoxi Shi, Weiwei Jia, Xiaoning Ding, Zirui Neil Zhao, Asaf Cidon, and Jianchen Shan. 2026. Not All Huge Pages Are Real: Detecting and Avoiding Misaligned Huge Pages in Public Cloud Virtual Machines. In *ACM Symposium on Cloud Computing Volume 2 (SoCC '26 Vol. 2), November 18–20, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3828161.3857854

1 Introduction

Huge pages are a standard mechanism for reducing address translation overhead. By mapping a larger memory region with one translation entry, huge pages increase translation lookaside buffer (TLB) coverage and reduce expensive page walks [47, 52, 71]. Modern operating systems therefore create huge pages opportunistically, most commonly through Transparent Huge Pages (THP) [30, 61].

Virtualization adds a second address-translation stage, which makes a TLB miss more expensive: the processor may need to walk both the guest and the nested page tables before it can access data [24, 44]. Huge pages reduce how often this happens, but only when both stages use them.

For each guest memory access, the processor first uses the guest page table to translate a guest virtual address (GVA) to a guest physical address (GPA). It then uses the nested page table (Intel EPT or AMD NPT), which the hypervisor maintains, to translate that GPA to a host physical address (HPA). The guest receives the TLB reach of a huge page only when both page tables map the region with a huge page. Figure 1 (left) shows this real huge page: the guest page table maps a 2 MiB GVA region to a 2 MiB GPA range, and the nested page table maps that GPA range to one host huge page. The nested page table can instead map the same GPA range with 4 KiB entries [28, 35]. Figure 1 (right) shows this fake huge page: the guest page table still holds one 2 MiB mapping, but every 4 KiB piece of the GPA range needs its own nested translation to a host base page.

This mismatch occurs not only in locally hosted VMs, but also in public cloud VMs. Host fragmentation, compaction, ballooning, NUMA balancing, swapping, deduplication, and page migration

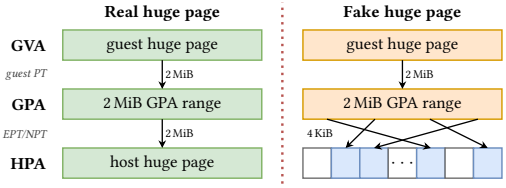


Figure 1: Real (left) and fake (right) huge pages in a VM. The guest page table (guest PT) maps both with a 2 MiB entry; only the nested page table (EPT/NPT) differs.

can split or rearrange the physical memory backing a guest [2, 10, 23, 35, 51, 52, 66, 69]. Across 636 VMs from AWS, Azure, and Google Cloud, we find many cases where guest huge pages are backed by host base pages, which we call *fake huge pages*. Existing systems address this guest-host misalignment from the host side. Ingens [35] changes how the host promotes and compacts huge pages, while Gemini [28] and HugeGPT [29] make the host and the guest coordinate so that guest huge pages are placed on host huge pages. All of them require changing the host kernel or the hypervisor, which is outside the control of a public cloud tenant. As a result, guest workloads rely on huge pages for performance while the guest kernel cannot determine which of its huge pages are backed by host huge pages. At the same time, the guest can observe which of its processes and memory regions are active. If it could identify which guest huge pages receive real host huge page backing, it could spend those pages on the data for which the translation benefit matters most [20, 42, 50, 54].

1.1 Challenges of VM Page Quality Detection

Throughout the paper, we refer to *page translation quality* as *page quality* for short. Designing a practical in-guest huge page management system is challenging for several reasons. First, host page backing is *opaque* to the guest. The guest can inspect its own page tables, but it cannot read the nested page tables or host physical addresses, so nothing in the guest reveals whether a region is backed by a host huge page [28, 29, 36].

Second, for a solution that does not rely on custom hypervisor extensions, the page quality must be inferred from *indirect microarchitectural signals*. Guest performance monitoring unit (PMU) events, which can provide straightforward low-level measurements, are often unavailable in public clouds due to security concerns [56], and virtual PMUs, when available, can be unreliable [70] and expose limited events [25]. Timing a single memory access is also ambiguous because its latency can include a data cache miss, a TLB miss, or an interrupt, and the guest cannot tell these apart from one measurement. The relevant cache and translation behavior also varies across processor generations [13, 21].

Third, page quality is *dynamic*. Host compaction, migration, ballooning, reclaim, and NUMA management can change the backing of a guest page after it has been classified [10, 19, 35, 51]. Any guest-side solution must therefore be lightweight enough to be used periodically in a practical systems solution.

Finally, detecting real huge pages does not determine where they provide the greatest benefit, which depends on the memory that the workload accesses the most [42, 50, 52]. When real huge pages are scarce, a practical system must account for both page quality and hotness.

1.2 This Paper

In this paper, we show that the guest can infer host page backing from the latency of a series of controlled memory accesses.

Our **key insight** is that there are various page-size-dependent microarchitectural operations that affect timing, and thus allow observations that reveal a guest's page backing on the host. vHuge implements this core idea with two methods.

First, a real huge page allows one TLB entry to cover the whole 2 MiB region of a guest huge page, whereas a fake huge page requires a separate TLB entry for every 4 KiB region. vHuge prepares the cache and TLB state so that the same access results in a TLB hit on a real huge page and a TLB miss on a fake one, while it hits in the data cache in both cases. The resulting latency difference allows us to classify huge pages from inside the VM. This *TLB method* works on both Intel and AMD processors.

Second, we observe that newer processors often demonstrate different cache line prefetching behaviors when accesses occur on a base page compared to a huge page. vHuge exploits this varying prefetching behavior as a "tell", given that this property is tied to the host's mapping. This *prefetcher method* is faster, but it applies only to processors with this behavior. vHuge uses the probe results to give real huge pages to the hottest memory, entirely from within the guest and without any hypervisor modification.

This paper makes the following contributions:

① We confirm that guest huge pages in public cloud VMs are not always backed by host huge pages. Through a large-scale study of 636 VMs across AWS, Azure, and Google Cloud, we observe fake huge pages at every provider (§ 2.3, Figure 2a).

② We introduce two guest-side methods for classifying real and fake huge pages. The TLB method works across both older and newer Intel/AMD processors, while the prefetcher method provides a faster probing method on newer processors that expose different cache line prefetch behavior for base and huge pages (§ 3).

③ We design, implement, and open-source vHuge in the Linux kernel: a guest-side huge page manager that keeps verified real huge pages in a pool and uses them when Linux allocates a new anonymous huge page. vHuge also extends Linux's Data Access MONitor (DAMON) [38] to find the hot huge pages of running processes. When the pool runs low on real huge pages, vHuge probes these hot huge pages and migrates the fake ones onto the remaining real huge pages (§ 4, § 5). vHuge is available at https://github.com/manitofigh/vHuge.

④ We evaluate vHuge on local VMs with 13 workloads. When the host backs 30% of the VM's memory with huge pages, vHuge improves performance by 20.45% on average (up to 97.1%) and reduces TLB misses by 49.78% on average. When the host backs all of the VM's memory with huge pages, where vHuge cannot improve page quality, it incurs an average overhead of 0.785% (§ 6).

2 Background and Motivation

2.1 Nested Address Translation

Virtualized address translation consists of two stages: first, the guest page tables are used to translate a guest virtual address (GVA) to a

guest physical address (GPA), and second, the nested page tables are maintained by the hypervisor and are used to translate the GPA to a host physical address (HPA). Intel and AMD implement this hardware-assisted second stage similarly: the processor's page table walker traverses the guest and nested page tables and combines their results into the final GVA-to-HPA translation. Intel refers to its mechanism as Extended Page Tables (EPT) [24], while AMD calls it Nested Page Tables (NPT) [1]. The processor caches the resulting GVA-to-HPA translation in the TLB. A TLB hit therefore skips both the guest page table walk and the nested EPT/NPT translations required to resolve it. When the GVA-to-HPA translation is not cached, the processor walks the guest page table, and every GPA that this walk produces must itself be translated through the nested page table. On a system with five-level guest and nested page tables, a complete walk can require as many as 35 memory references [44]. Processors also cache intermediate results, including GPA-to-HPA translations, so a TLB miss does not always traverse every level of both page tables. In all cases, however, a TLB miss costs additional work that a TLB hit avoids.

The page sizes used by the two stages together determine the size of the TLB entry. The processor caches a GVA-to-HPA translation in a 2 MiB TLB entry only when *both* the guest page table and the nested page table map the address with a 2 MiB page. If either one uses a 4 KiB mapping, the processor caches a 4 KiB TLB entry that covers only the accessed 4 KiB region. When a 2 MiB guest mapping is backed by a 2 MiB host mapping, one TLB entry therefore covers the complete region. If the host maps that region using 4 KiB entries, the guest still holds one 2 MiB mapping, but each 4 KiB region inside it needs its own TLB entry [28, 29]. The TLB method uses this difference to distinguish real from fake huge pages (§ 3.1).

2.2 Fake Huge Pages in Virtual Machines

Huge pages increase TLB coverage by allowing one translation entry to map a larger memory region. Linux provides this benefit transparently through Transparent Huge Pages (THP), a mechanism that backs eligible memory regions with 2 MiB huge pages instead of 4 KiB base pages [30, 47, 52, 61]. On a physical machine, this promotion directly determines the page size used by address translation. In a VM, however, the guest and host manage their huge pages independently. Promoting a region inside the guest therefore does not guarantee that the host will also map that region using a huge page.

On x86-64 Linux, a 2 MiB huge page is mapped by one entry at the page middle directory (PMD) level of the page table. We use *PMD mapping* to refer to this 2 MiB mapping, and *THP* to refer to a 2 MiB huge page that the THP mechanism created. Linux manages each THP as a folio, a single object that groups its 512 contiguous base pages [59].

This independent management creates three possible backing configurations for a guest huge page. Its 2 MiB GPA range may be mapped by the nested page table to one host huge page, to 512 host base pages that are physically contiguous, or to host base pages scattered across physical memory. We call the first configuration a *real huge page*, as both translation stages use a huge page mapping. We call the latter two configurations *fake huge pages*: the guest page table holds a 2 MiB mapping, but the nested page table maps the same range with 4 KiB entries. The second configuration occurs, for example, when the 2 MiB boundaries of the GPA range and of the host memory that backs it are not aligned, or before the host has promoted the range. Physical contiguity alone is therefore insufficient; the nested page table must also hold a huge page mapping for the guest to receive the expected TLB reach [28].

On our local setup, where a custom hypercall reports the host backing of every guest huge page, contiguous host base pages back less than 1% of the guest huge pages (§ 6.1). vHuge treats both cases backed by host base pages as fake because neither one provides the TLB coverage of a huge page.

Moreover, the backing of a guest page is not necessarily permanent. Fragmentation, compaction, ballooning, reclaim, NUMA balancing, swapping, page migration, and the host's own THP promotion and splitting can split, move, or reassemble the host memory assigned to a VM [2, 10, 35, 51, 66, 69]. A guest huge page backed by a host huge page may later fall back to base page translations, and a previously fake huge page may later be promoted by the host, making page quality opaque and dynamic.

2.3 Motivating Experiments

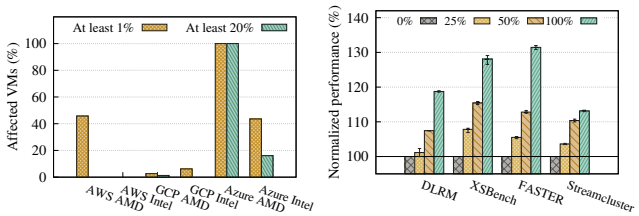
2.3.1 Cloud Fake Huge Pages. Cloud platforms try to preserve host huge page mappings for VMs [28]. For example, an Azure production study reports no hypervisor-level huge page fragmentation because each VM's memory is allocated contiguously when the VM is created [19]. Our measurements show that this property does not hold for all Azure instances, and many instances contain a substantial fraction of fake huge pages.

We collected measurements from 636 on-demand VMs across AWS, Azure, and Google Cloud: 357 on AWS, 121 on Azure, and 158 on Google Cloud. The VMs span AMD and Intel processors, 20 regions, 59 zones, and 24 VM shapes (Table 4). In each VM, we repeatedly allocate guest 2 MiB huge pages and probe them with the method described in Section 3. Each probe round reports one value: the percentage of the guest huge pages allocated in that round that the prober classifies as fake. A single round can occasionally report a high percentage that the rounds before and after it do not report, for example when the vCPU is preempted while probing. We treat such an isolated value as a false positive of the prober and remove it. For example, in the sequence 0%, 100%, 0% we remove the 100% value, while in 70%, 85%, 80% we keep all three values.

We call the result the *sustained fake huge page fraction*. Thus, a VM reaches the 1% or 20% threshold only when consecutive probe rounds report that level. Section 6.2 presents the complete breakdown by region, VM shape, role, and time.

Figure 2(a) shows fake huge pages on all three providers, with substantially different frequency and severity. A sustained fake huge page fraction of at least 1% occurs in 16.5% of AWS VMs, 4.4% of Google Cloud VMs, and 71.1% of Azure VMs. Severe cases are concentrated on Azure: 57.0% of Azure VMs reach at least 20% fake huge pages, including every tested Azure AMD VM. No AWS VM reaches 20%; one Google Cloud AMD VM does. Thus, provider-side efforts to preserve contiguous VM memory do not guarantee that guest huge pages receive aligned host huge page backing [19, 28].

2.3.2 Performance Increases with THP Coverage. We next measure how workload performance changes as a larger fraction of



(a) Cloud fake huge pages **(b) Performance with THP limit**

Figure 2: (a) Cloud VMs reaching 1% or 20% of sustained fake huge pages. (b) Performance at each THP limit, normalized to 0% THP.

its memory is backed by THPs. The four workloads represent a recommendation model, Monte Carlo lookup, concurrent key-value processing, and online clustering. We run this experiment on our local KVM host (§ 6). Before each VM boots, we reserve host huge pages and use them to back all of the VM's memory, so that every guest THP is a real huge page. We then modify the guest THP allocation path to back 0%, 25%, 50%, or 100% of the eligible 2 MiB regions with guest THPs; the remaining regions are backed by 4 KiB pages. We run one warmup and ten measured runs at each setting for all workloads, and report the mean of the measured runs. FASTER is configured with a mix of 95% reads and 5% updates [8, 12]. We measure THP coverage from each workload's anonymous memory and report mean performance relative to its 0% THP mean.

Figure 2(b) shows a monotonic performance increase for all four workloads. With nearly complete THP coverage, FASTER improves by 31.5%, XSBench by 28.1%, DLRM by 18.7%, and Streamcluster by 13.1%. The benefit is already visible at 50% coverage: XSBench, FASTER, Streamcluster, and DLRM improve by 15.4%, 12.8%, 10.4%, and 7.4%, respectively. These results show that losing effective huge page coverage can directly reduce the applications' performance.

3 In-Guest Huge Page Quality Detection

To detect real or fake huge pages from within the guest, a seemingly direct approach could be classifying a candidate using virtual PMU events for completed 2 MiB and 4 KiB page walks. However, these events increment only when the measured access actually triggers a page walk. For instance, if the translation is already cached, neither event increments, so a zero count cannot distinguish a fake page from a page for which no walk was observed. The counters also include matching walks from unrelated accesses during the measurement window. These ambiguities make PMU events insufficient for determining whether a candidate has real host backing. vHuge therefore avoids using PMUs due to their unreliable and limited nature in VMs [25, 56, 70]. We next describe two timing-based probes that classify a guest huge page as real or fake using only memory accesses made inside the guest.

3.1 The TLB Method

The TLB method relies on one property of nested address translation (§ 2): the processor caches a guest translation in a 2 MiB TLB entry only when both the guest page table and the nested page table map the address with a 2 MiB page. Every candidate that vHuge probes is already mapped by a 2 MiB entry in the guest page table. An access to a real huge page therefore creates one TLB entry that covers the whole 2 MiB region, while an access to a fake huge page creates a TLB entry that covers only the accessed 4 KiB region.

A simple probe would access one address in a candidate guest huge page, then time an access to another address in the same page. If the candidate is real, the first access creates a TLB entry that also covers the second address, so the second access hits in the TLB. If the candidate is fake, the second access misses in the TLB. However, the latency of the second access does not isolate the TLB miss. A high latency can come from a data cache miss that fetches the data from DRAM, which makes a real huge page appear fake. A low latency can mean that the translation of the second address was already cached from an earlier access, which makes a fake huge page appear real.

The TLB method addresses both problems by preparing the data cache and the TLB state separately, so that the timed access always hits in the data cache and differs only in whether it hits in the TLB. Figure 3 shows the four steps for one candidate huge page.

At ①, the probe loads a cache line at the beginning of the candidate. The load brings the line into the L1 data cache and creates a TLB entry for it. At ②, the probe invalidates the TLB entry that covers this beginning address, while the cache line stays in the L1 data cache. Invalidating a TLB entry requires a privileged instruction (`invlpg`), so the userspace probe uses PTEditor [45], a library whose kernel module performs the invalidation for a given virtual address. After this step, no TLB entry covers the beginning address.

At ③, the probe accesses a line 1 MiB away from the beginning of the candidate. This address is inside the same guest huge page but outside the first 4 KiB region. If the candidate is real, the access creates a 2 MiB TLB entry, which also covers the beginning address. If the candidate is fake, the access creates a 4 KiB TLB entry, which covers only the accessed 4 KiB region. Both addresses are arbitrary choices: any two addresses in the candidate work, as long as they are not in the same 4 KiB region.

At ④, the probe times a second access to the beginning line. This access hits in the L1 data cache because of ①. For a real candidate, it also hits in the TLB because of ③. For a fake candidate, no TLB entry covers the beginning address after ②, so the access misses in the TLB and the processor must translate the address again. The measured latency therefore reflects only a TLB hit or a TLB miss [13, 21]. The extra latency of the miss comes from the page walk that rebuilds the TLB entry. This walk may reuse translations that the processor has cached for the nested page table, so it does not always traverse every level of both page tables, but it is still measurably slower than a TLB hit.

Threshold calibration. vHuge classifies a candidate by comparing the measured latency against a threshold that it calibrates on every VM when it starts; the threshold is not a fixed value shared across VMs. vHuge measures an intentional TLB hit 2,000 times and an intentional TLB miss 2,000 times on a huge page that it allocates for this purpose. A hit repeats steps ① and ④ without the invalidation, and a miss invalidates the TLB entry directly before the timed access. vHuge takes the median latency of each case and sets

$$threshold = \frac{2 \cdot median_{hit} + 3 \cdot median_{miss}}{5}.$$

We place the threshold closer to the miss median because, in our measurements, a threshold at the midpoint classifies some real huge

pages as fake (§ 6.1). When probing a candidate, vHuge repeats the four steps five times and compares the median of the five latencies with the threshold.

Robustness to noise. The probe depends only on the L1 data cache and the TLB, which are private to a core. Contention in the shared last-level cache from co-located VMs therefore does not change the measured latency. A vCPU preemption or an interrupt during the timed access can inflate one measurement; one probe takes about 2 μs, which makes this rare, and using the median of repeated measurements removes such outliers.

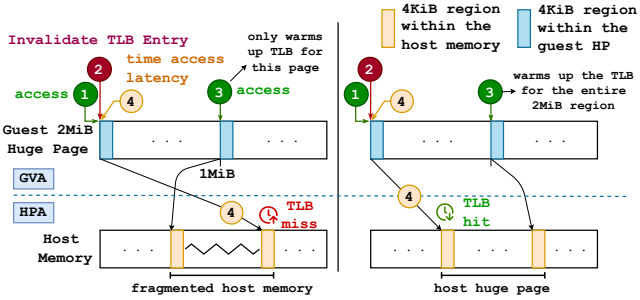


Figure 3: Probing a fake huge page (left) and a real huge page (right).

3.2 The Prefetcher Method

Data cache line prefetching hides memory latency by fetching data before the processor explicitly requests it. Hardware prefetchers use observed access patterns to fetch future cache lines [9].

We observe that, on newer processors, prefetcher behavior can differ according to whether the accessed lines belong to a base page or a huge page. For example, on our Emerald Rapids processor, the same access pattern can prefetch a later line when the accesses operate on a huge page, but not when they operate on a base page. We use this difference as a signal for the page size of a mapping from inside a VM, since the behavior depends on the host's mapping.

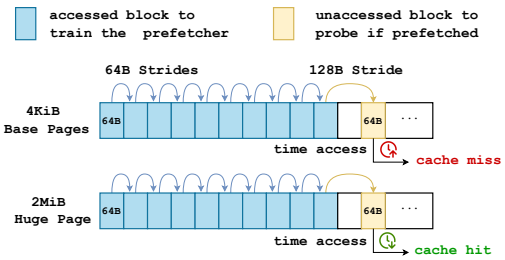


Figure 4: The prefetcher probe on Intel Emerald Rapids. Uncovering a real huge page (bottom) and a fake huge page (top).

The prefetcher method turns this observation into a classifier for real and fake huge pages. It first needs to find, once per processor model, an access pattern that the prefetcher treats differently on the two kinds of pages. At step 1, the method obtains one real huge page and one fake huge page, which the TLB method can identify. These two pages are the known cases on which it searches for such a pattern.

At step 2, the method tries a set of *configurations*. A configuration is one access pattern, defined by three numbers: how many accesses the pattern contains, the stride between consecutive accesses, and the distance from the last access to a *probe line* that the pattern itself never accesses. Our search covers 128 configurations: 5 to 12 accesses, strides of 64 to 256 B in steps of 64 B, and a probe line 1 to 4 strides beyond the last access. The search is needed because prefetcher behavior differs across processor generations. For example, on our Emerald Rapids processor, we found a configuration in which nine accesses are 64 B apart, and the cache line 128 B beyond the last access is prefetched when the accesses operate on a real huge page, but not when they operate on a fake huge page. Figure 4 shows this trained configuration for the two mappings.

At step 3, the method runs the configurations in *rounds* on each of the two known pages. One round flushes the cache lines that the configuration uses, performs the accesses of the configuration, and then times an access to the probe line: a cache hit means that the prefetcher fetched the probe line, and a cache miss means that it did not. The method first runs every configuration for 20 rounds and discards those whose results on the two pages differ in fewer than 18 rounds. It then runs each remaining configuration for 1,000 rounds. For each of them, the method computes the fraction of rounds that hit on the real huge page and on the fake huge page, and takes the difference between the two. For example, a configuration that hits in 99% of the rounds on the real huge page and in 0% of the rounds on the fake huge page has a difference of 99%. The method keeps the configuration with the largest difference. The complete search, over all configurations and both pages, takes less than one second on our Emerald Rapids processor and runs once; later probes reuse the selected configuration.

At step 4, the method classifies a candidate guest huge page with one round of the selected configuration: it flushes the lines, performs the accesses, which we call *training accesses*, and times the access to the probe line. On Emerald Rapids, a cache hit classifies the candidate as real and a cache miss as fake.

On our Emerald Rapids processor, each prefetcher probe takes 260 ns per huge page, compared with 2,187 ns for the TLB method, making it about 8.4 times faster. This advantage is processor-specific: it requires a prefetcher that treats base pages and huge pages differently. We observe this behavior on Emerald Rapids but not on Skylake-SP (§ 6.1). vHuge runs the search once when its daemon starts. It uses the prefetcher method for the pool only if the best configuration differs between the two known pages in at least 90% of the rounds; otherwise, and for all probing inside the kernel, it uses the TLB method, which works across the Intel and AMD systems that we target and does not depend on prefetcher behavior.

4 vHuge Design

vHuge uses the TLB method from Section 3 to give real huge pages to the memory that benefits from them. This section first describes the components of vHuge and how they interact, and then describes each one in detail.

4.1 Overview of vHuge

vHuge separates page quality from page hotness. Page quality says whether a guest huge page is real or fake; the guest cannot read it

Table 1: Components of vHuge and what each one requires.

Component	Requires
Pool daemon (poold)	Userspace daemon; PTEditor kernel module for TLB invalidation and page table inspection
Real huge page pool, allocator hook	Guest kernel changes (THP fault, khugepaged, and page free paths)
Hot page monitor	Guest kernel changes; DAMON (in mainline Linux) with our extension that maps hot regions to huge pages
Kernel prober, migration worker	Guest kernel changes; Linux page migration
Cloud measurements (§ 6.2)	Userspace prober and PTEditor module only; unmodified provider kernel

from the host page tables, but vHuge recovers it through timing. Page hotness says which huge pages of a process are accessed the most; vHuge observes it through DAMON [38, 42, 50]. Combining the two lets vHuge spend a limited supply of real huge pages on the memory that needs them most.

vHuge has four components (Figure 5, Table 1):

(1) The pool daemon. vhuge-poold, which we call poold below, is a userspace daemon. It allocates candidate huge pages, probes them, and hands the real ones to the kernel, which keeps them in the *real huge page pool*. It also periodically probes the pages that wait in the pool again.

(2) The allocator hook. When Linux allocates a new anonymous huge page for an application, the hook takes a real huge page from the pool. If the pool cannot provide one, Linux allocates the huge page as usual.

(3) The hot page monitor. When poold can no longer find enough real huge pages, vHuge starts DAMON to find the hot huge pages of the running processes.

(4) The kernel prober and the migration worker. The kernel prober classifies the hot huge pages that DAMON reports, and the migration worker moves the hot fake ones onto real huge pages from the pool. We call this *repair*.

Components (1) and (2) always run. Components (3) and (4) run only while real huge pages are scarce, i.e., after refills leave the pool below its *low watermark*.

Consider a workload that allocates anonymous memory. At ①, poold allocates candidate huge pages (①.①), probes them (①.②), and filters the results (①.③). A candidate classified as real enters the real huge page pool at ②, while a fake candidate returns to Linux's buddy allocator, which manages free physical memory. When the workload faults on a new huge page, or when Linux's khugepaged background thread replaces the base pages of a region with a huge page, the allocator hook at ③ and ③.① takes a real huge page from the pool. If real huge pages become scarce, vHuge starts DAMON at ⑤ to observe the workload, and the kernel prober at ④ classifies the workload's hot huge pages. The migration worker then moves each hot fake huge page onto a real huge page from the pool.

4.2 Real Huge Page Pool Placement

Building the pool. vHuge fills its pool in the background, so that an application fault never waits for a probe. poold checks the number of pages in the pool once per second. When the pool is below its *low watermark*[1], poold performs a *refill*: it allocates a batch of 256 anonymous 2 MiB regions as candidates, asks Linux to back them with huge pages using madvise, and accesses them. Because madvise is only a request, poold then reads its own page table and drops every candidate that the guest did not map with a 2 MiB entry, so that a candidate is never classified as fake only because the guest itself backed it with base pages. poold probes the remaining candidates, reports those classified as real to the kernel, and unmaps the batch. A hook in the kernel's page free path intercepts each reported huge page before it returns to the buddy allocator and places it in the real huge page pool. The fake candidates return to the buddy allocator.

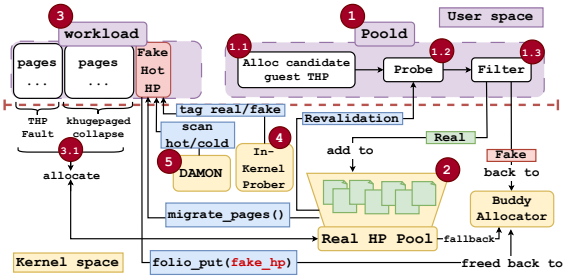


Figure 5: End-to-end organization of vHuge.

Pool construction only needs poold's own mappings, so we keep it in userspace, which also makes the daemon simpler to develop and update; nothing prevents implementing it in the kernel. To bound vHuge's overhead, each refill allocates at most one batch. When a refill adds fewer than 25% of its candidates to the pool, poold waits 5 seconds before the next refill, and when it adds at most 10% it waits 10 seconds. Thus, vHuge searches every second while the host provides real huge pages, but does not keep allocating and probing candidates when the host does not [23, 51, 53, 68].

Revalidating the pool. Host memory management can change the backing of a pooled page after poold first classifies it [10, 69]. poold therefore revalidates up to 512 pooled pages every 10 seconds. It maps the selected pages into its own address space and runs the TLB probe again. Pages that are still classified as real stay in the pool. The others return to the buddy allocator, where a later refill can pick them as candidates again. Revalidation also runs in the background, so a workload does not wait for a probe when it receives a page from the pool.

Using the pool. When the real huge page pool is above the low watermark, vHuge uses it for every new anonymous 2 MiB huge page. On an anonymous huge page fault or an anonymous khugepaged collapse, the vHuge allocator first tries to take a real huge page near the requested NUMA node [15, 20]. If it succeeds, Linux continues through its ordinary THP mapping path with that page. If the pool cannot provide a page, Linux follows its normal THP allocation path. Thus, vHuge never blocks a new THP only because a real huge page is unavailable.

After Linux maps a real huge page from the pool, vHuge records the process that owns it and its virtual address. We call this an *ownership record*. It lets the repair path recognize a huge page that vHuge already verified as real. vHuge removes the record when the huge page is freed or split.

[1]We set the low watermark to 2% of the free memory at boot, which is about 320 huge pages on our system. Since poold checks once per second, this leaves enough pages for new allocations until the next refill.

4.3 Hot Huge Page Monitoring and Repair

Finding hot huge pages. The allocator hook only helps huge pages that are created after real huge pages enter the pool. A long-running workload can already hold hot fake huge pages when real huge pages become scarce. To find them, vHuge uses DAMON VADDR, DAMON's monitor for the virtual address space of a process [38].

vHuge registers a process with DAMON the first time the allocator hook sees it allocate a huge page; later huge pages of the same process need no new registration. vHuge identifies a registered process by its thread-group identifier (TGID) and looks the TGID up every time it uses it, so a process that has exited produces no probing or migration work, and vHuge removes it from DAMON later.

DAMON periodically samples whether regions of the virtual address space are accessed and reports the access frequency of each region [38]. Its regions have variable sizes and do not correspond to huge pages. vHuge therefore extends DAMON VADDR: it converts each hot region into the 2 MiB-aligned addresses that the region overlaps. We call each such address a *candidate*; it consists of a TGID, a 2 MiB-aligned virtual address, and the access frequency of the region that contains it. One hot region can produce several candidates, which all carry the region's access frequency.

Scouting and pressure mode. vHuge starts DAMON after the first refill that leaves the pool below its low watermark. We call this first period *scouting*: DAMON collects access frequencies, and vHuge keeps the pages that remain in the pool for allocations at addresses that DAMON reports as hot. DAMON samples accesses every 20 ms, aggregates the samples every 500 ms, and updates its regions every second. vHuge treats a region as hot when it is accessed in at least 50% of the samples, a threshold commonly used in Linux [37]. A successful refill during scouting ends scouting.

If three consecutive refills leave the pool below its low watermark, vHuge enters *pressure mode*. In pressure mode, only a huge page at an address that DAMON reports as hot may take a real huge page from the pool; all other huge pages are allocated through Linux's normal path, and vHuge repairs hot fake huge pages as described below. vHuge leaves pressure mode when the pool reaches the *recovery watermark*, which is halfway between the low watermark and the pool target, and pressure mode has been active for at least 30 seconds. These two conditions avoid starting and stopping DAMON repeatedly while the pool is close to its low watermark.

Classifying and repairing candidates. `poold` can only probe pages in its own address space. A hot huge page of a running workload must be probed at the virtual address that the workload uses. vHuge therefore includes a kernel prober that temporarily switches to the address space of the target process and runs the TLB probe there (§ 5.2). The workload does not stop, remap memory, or cooperate with vHuge.

Once per second, vHuge scans the candidates in order of their access frequency. A candidate that vHuge already knows to be real stays in place. A candidate of unknown quality is sent to the kernel prober, and vHuge records the result. A candidate that an earlier scan classified as fake is sent to the migration worker if the pool holds a real huge page. The worker uses Linux's `migrate_pages()` to move the contents of the fake huge page onto the real one and to update the page table [60]. Probing and migration happen in separate scans so that a candidate is migrated only if it is still hot one scan after it was found to be fake.

Before probing and before migrating, vHuge checks that the candidate's address is still mapped by the same huge page. If the huge page has been split, unmapped, or replaced, if the process has exited, or if the migration fails, vHuge drops the candidate and leaves the workload's memory unchanged; an unused real huge page returns to the pool. If the pool has no real huge page, the candidate stays as it is and vHuge considers it again when DAMON next reports it as hot. One scan sends at most 16 candidates to the prober and 16 to the migration worker.

When allocated huge pages are checked again. While the pool is above its low watermark, every new anonymous huge page comes from the pool and has already been verified as real. Once the pool runs low, new huge pages fall back to Linux's normal allocation path and are not probed (as are huge pages allocated before `poold` first fills the pool), so some are fake, and some of those hold hot data. DAMON lets vHuge find these hot pages and spend the remaining real huge pages on them. vHuge probes the pages in the pool every 10 seconds, but while the pool stays above its low watermark, it does not probe a huge page again after an application has received it: the monitor runs only during scouting and pressure mode, and the kernel prober and the migration worker only during pressure mode. If the host changes the backing of such a page, it stays fake until it is freed or real huge pages become scarce. We made this choice because host backing changes rarely in our cloud measurements: the fake huge page fraction of the GCP VM in Figure 6 changes once in about 3.5 hours (§ 6.2). Probing every allocated huge page periodically would add overhead to all workloads to detect such rare changes, while a workload that runs for minutes frees its huge pages before a change is likely.

5 Implementation

vHuge is implemented inside the guest on Linux 6.15.2. The kernel part contains about 4,450 lines of code and the userspace daemon about 1,200 lines, excluding the libraries and modules that they use, such as PTEditor.

5.1 Linux Integration and Pool Maintenance

Allocation hooks. vHuge modifies the two paths that allocate an anonymous 2 MiB huge page. In the page fault path, Linux asks the vHuge allocator for a real huge page near the local NUMA node. The helper for this lookup is `vhuge_alloc_real_folio()`. If the allocator returns a page, Linux continues through its normal map-and-charge path [61]. If it does not, the code follows the unmodified Linux allocation path. The hook never probes, refills, or migrates on the fault path.

vHuge uses the same rule when `khugepaged` creates an anonymous huge page. Huge pages for file-backed and shared memory (shmem) regions use the normal Linux allocator [48, 61]; Section 7 explains why. When Linux maps a huge page that came from the pool, vHuge creates its ownership record, and the split and free paths remove it.

`poold`. `poold` uses PTEditor [45] for two purposes. First, PTEditor invalidates the TLB entry of a virtual address, which requires kernel privilege and which the TLB method needs at step ②. Second,

PTEditor reads the guest page table, which `poold` uses to check that the guest maps a candidate with a 2 MiB entry after `madvise` [61]. PTEditor already provides both operations through a kernel module, so we did not reimplement them. `poold` talks to the kernel through a control device: it reports the candidates that it classified as real, and the kernel's free-path hook moves their physical 2 MiB blocks into the pool when `poold` unmaps them.

`poold` calibrates the TLB threshold when it starts (§ 3.1); users can also set it manually. The kernel prober calibrates its own threshold with the same formula, because its timed access carries less measurement overhead than the userspace one. It also makes a second timed access right after each probe, which is a TLB hit, and classifies the huge page by how far the probe is above that hit. For pool revalidation, `poold` maps a batch of pooled pages through the control device, probes them, and reports which pages are still real.

5.2 Hot Fake Huge Page Migration

Monitoring and classification. During classification, vHuge resolves each candidate against the process's current page table rather than assuming that the mapping observed by DAMON remains unchanged. A candidate must cover a complete 2 MiB range within a readable anonymous mapping and be mapped by one 2 MiB huge page. vHuge excludes I/O, direct PFN, mixed, hugetlb, and huge zero mappings, since these mapping types do not follow the anonymous THP migration path used by vHuge. Folios mapped more than once are also excluded because migrating them could affect mappings outside the region selected by DAMON, while a folio that passes these checks but cannot be isolated is rejected later by the migration worker without changing the original mapping. For eligible mappings, vHuge leaves known real THPs in place, queues unknown THPs for probing, and retains recent fake results so that they can be considered for migration without another probe[2].

Kernel probing and migration. The kernel prober runs on a dedicated kernel worker thread. For each request, the worker temporarily switches to the address space of the target process and runs the TLB probe on the same virtual address that the application uses. The kernel invalidates the TLB entry directly, so this path does not need PTEditor. The application keeps running and can access the huge page while it is probed.

The worker protects each probe in three ways. First, it holds a reference to the process address space and to the huge page, so neither can be freed while the probe runs. Second, it checks that the page table still maps the same huge page at that address before and after the probe, and discards the result if either check fails. Third, it disables page fault handling around its accesses (`pagefault_disable()`). If an access would fault, for example because the huge page was unmapped in the meantime, the access fails immediately and vHuge discards the measurement, instead of timing the fault and reading the delay as a TLB miss. The worker stays on one CPU during a probe. Like `poold`, it classifies a huge page by the median of its samples.

Migration requests execute on a separate kernel worker. After confirming that the mapping is unchanged, the worker takes a real huge page from the pool, removes the fake huge page from the kernel's LRU lists so that no other kernel path (e.g., reclaim) operates on it during the migration, and calls `migrate_pages()` [60], which copies the contents and updates the page table. If the migration does not complete, the application keeps its original huge page and vHuge returns the real huge page to the pool.

Control interfaces. vHuge exposes the pool target, the low watermark, and the DAMON and repair limits through `sysfs`; the refill batch size is an option of `poold`. `/proc/vhuge` reports the pool, DAMON, probe, and migration state and the real huge pages that each process owns.

6 Evaluation

Objectives. Our evaluation covers five parts. We first measure the accuracy and cost of the TLB and prefetcher methods on local VMs, where we know the host backing (§ 6.1). We then use the TLB method to measure fake huge pages in public cloud VMs (§ 6.2). Next, on local VMs, we compare vHuge with Linux THP when the host backs only 30% of the VM's memory with huge pages (§ 6.3). We then limit the number of real huge pages that a workload can receive and compare vHuge with and without DAMON (§ 6.4). Finally, we measure vHuge's overhead when the host backs all of the VM's memory with huge pages, and the cost of its components (§ 6.5).

Unless stated otherwise, each workload experiment uses one warmup run and ten measured runs, and we report the mean of the measured runs.

Performance evaluation environment. We use public cloud VMs to measure how often guest huge pages are fake, but we run all performance experiments on local VMs. The performance of a shared cloud VM changes with cache contention from co-located tenants and with vCPU preemption [7, 15, 22, 27], and this contention varies over hours [3, 55, 56, 63]. If the contention drops during a vHuge run, the workload speeds up, and the speedup would be wrongly attributed to vHuge. We therefore run the performance experiments on local hosts without co-running VMs. Our performance results thus show the benefit of vHuge under a controlled fraction of real huge pages; we do not measure application performance in cloud VMs.

Table 2: Local platforms used in experiments.

	Skylake-SP	Emerald Rapids
CPU	Xeon Gold 6138, 2.00 GHz	Xeon Gold 6554S, 2.20 GHz
Cores	4 sockets, 80 cores	2 sockets, 72 cores
Memory	251 GiB	499 GiB
L1d	32 KiB/core	48 KiB/core
2 MiB TLBs	L1 data TLB: 32 entries; L2 TLB: 1536 entries	L1 load-only data TLB: 32 entries; L2 unified TLB: 1024 entries
Use	TLB probe, evaluation, overhead	Prefetcher probe

Platforms. Table 2 summarizes the local machines used for experiments. Both hosts and guests run Linux 6.15.2, and vHuge is built atop the same Linux version. Hyperthreading is disabled. Unless stated otherwise, each local VM has 10 vCPUs and 32 GiB of memory, with vCPUs pinned 1-to-1 to physical cores in one socket. THP is set to `always`. We enable NUMA interleaving for the workloads so that their memory is spread evenly across NUMA nodes

[2] vHuge keeps a fake result for one minute and does not probe that huge page again within that time; a huge page found real is not probed again until it is freed. Huge page backing in cloud VMs changes rarely (§ 6.2), so one probe per minute is sufficient.

in every run. Without it, the share of local and remote memory accesses differs between runs; the throughput of GUPS, for example, varies by up to 10×, which would hide the effect of vHuge. We use KVM/QEMU VMs for local experiments [34].

6.1 Probe Accuracy and Cost

We measure the accuracy of both methods on local VMs, where we can observe the host backing. For the TLB method, we add a hypercall to our local KVM host that takes a guest physical address and returns the host physical address and whether the host backs it with a huge page. This hypercall exists only for validation; vHuge does not use it. It distinguishes the three configurations of Section 2: one host huge page, contiguous host base pages, or scattered host base pages. Contiguous host base pages back less than 1% of the guest huge pages that we observe. For the prefetcher method, we allocate memory on Emerald Rapids whose backing we control, a real huge page or a fake huge page, run the access pattern in Figure 4, and compare the result with the known backing.

In both cases, the probed pages are anonymous huge pages that the prober allocates itself; no application runs in the VM. We classify 10,000 candidate huge pages with each method.

Table 3: Accuracy and cost of the two huge page classifiers.

Method	Ground truth	Accuracy	Mean cost
TLB	Custom hypercall reports HPA and host backing on Skylake-SP	99.91%	2187ns/huge page
Prefetcher	Known huge and base page allocation	99.41%	260ns/huge page

Table 3 shows that both methods classify huge pages with high accuracy. The remaining errors of the TLB method are mostly real huge pages classified as fake: a TLB hit whose latency lies above the threshold is read as a miss. This is why the calibrated threshold is placed closer to the miss median than to the hit median (§ 3.1). The prefetcher method is about 8.4× faster than the TLB method, but it is processor-specific [13]. Its errors occur when the training accesses do not trigger the expected prefetch in a round. In the rest of the paper, we use the TLB method for the cloud measurements and for all vHuge experiments.

6.2 Fake Huge Pages in Public Cloud VMs

We next measure whether guest huge pages receive host huge page backing in public cloud VMs. The cloud experiments use the provider-supplied kernel; we run the userspace vHuge prober and the PTEditor module needed for TLB invalidation [45]. The prober calibrates its threshold on every VM (§ 3.1), so no threshold is carried over from our local machines. We have no access to the cloud hosts and therefore cannot verify these classifications against the host page tables; the accuracy in Table 3 is measured only on our local setup. Each probe round allocates at most 4 GiB of anonymous memory, requests THP with `madvise`, checks which regions were promoted to guest huge pages through `/proc/self/pagemap` and `/proc/kpageflags` [57], and then probes those guest huge pages. *Observer* VMs probe periodically while otherwise idle. *Kernel build* VMs run a clean Linux kernel rebuild between probe rounds to trigger some memory pressure before probing again.

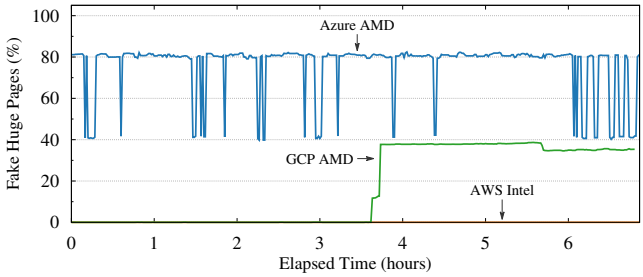


Figure 6: Fake huge pages over time in Azure, GCP, and AWS VMs.

Table 4 reports cumulative probe events and sampled guest huge pages across AWS, Azure, and GCP on-demand VMs[3]. A sampled guest huge page is one probe observation, not a distinct resident huge page; across repeated allocations, the prober samples whatever guest frame numbers the cloud VM allocator returns. The affected VM summary in Figure 2(a) already shows the fraction of VMs whose sustained fake huge page fraction reaches 1% or 20%. Section 2 describes how we remove a high reading that does not have support from the neighboring probe rounds before computing this sustained fraction. Table 4 includes the provider, processor, region, and workload-role breakdown behind that summary.

Fake huge pages are not evenly distributed across clouds. Azure AMD is the most severe case in our measurements: 66.55% of sampled observer HPs and 64.35% of sampled kernel build HPs are fake. Azure Intel also exhibits fake huge pages but at lower rates. AWS and GCP are usually much closer to full host huge page backing, although not necessarily for all instances. We observed that kernel build runs do not meaningfully change the memory backing of VMs under the applied memory pressure; most instances that observe fake huge pages tend to observe them from the beginning of the instance's lifetime.

Figure 6 shows three VMs that we selected to illustrate the three behaviors over time that we observe: a VM whose fake huge page fraction is high from the start (Azure AMD), a VM that stays real throughout (AWS Intel), and a VM that becomes fake over time (GCP AMD). The Azure AMD VM continuously observes fake huge pages, but the fraction changes from roughly 40% to 80% depending on which part of the VM's assigned memory is returned to the prober in a given round. The GCP AMD VM remains near 0% for about 3.5 hours, later jumps to about 20% fake huge pages for approximately 10 minutes, and finally to 40%. The AWS Intel VM remains close to 0% throughout the entire measurement window. These examples show that page quality depends on the provider and can change over time, but that such changes are infrequent. This is why vHuge probes pages before using them and revalidates pooled pages over time (§ 4.2, § 5.1) [10, 19].

6.3 End-to-End Performance

We compare vanilla Linux THP against vHuge on the same local KVM setup. Both configurations run with guest THP set to `always` [61]. Vanilla Linux treats all guest THPs as equally useful, while vHuge lets anonymous THP faults and `khugepaged` collapses use a verified real page from the pool for the resulting THP

[3]We also ran the same tests in spot VMs on all providers, and they achieved almost identical results in all cases. The reported data only includes the on-demand instances.

Table 4: Probed fake huge pages in public cloud VMs across AWS, Azure, and GCP.

Env.	Regions	VM families (cores/GiB DRAM)	Observer (Idle)					Kernel build				
			VMs	Probe events	Sampled HPs	Fake HPs	Fake rate	VMs	Probe events	Sampled HPs	Fake HPs	Fake rate
AWS AMD	us-east-1, us-east-2, us-west-1, us-west-2	m6a: 2xlarge, 4xlarge, 8xlarge (4/32, 8/64, 16/128)	86	26,612	26.6M	78.6K	0.30%	43	2,982	3.0M	10.0K	0.34%
AWS Intel	us-east-1, us-east-2, us-west-1, us-west-2	m6i/m7i.{2,4,8}xlarge (4/32, 8/64, 16/128); r6i.{2,4}xlarge (4/64, 8/128)	138	33,331	33.3M	1.4K	0.004%	90	4,348	4.3M	2.1K	0.049%
Azure AMD	centralus, eastus, eastus2, northcentralus, southcentralus, westus, westus2, westus3	D8as_v6, D16as_v6, D32as_v6 (4/32, 8/64, 16/128)	31	17,302	17.3M	11.5M	66.55%	28	3,390	3.4M	2.2M	64.35%
Azure Intel	centralus, eastus, eastus2, northcentralus, southcentralus, westus, westus2, westus3	D8s_v6, D16s_v6, D32s_v6 (4/32, 8/64, 16/128)	31	18,207	18.2M	357.5K	1.96%	31	3,720	3.7M	110.6K	2.97%
GCP AMD	us-central1, us-east1, us-east4, us-south1, us-west1, us-west2, us-west3, us-west4	n2d-standard-{4,8,16,32} (2/16, 4/32, 8/64, 16/128)	47	28,157	28.1M	139.1K	0.49%	30	2,156	2.2M	500	0.023%
GCP Intel	us-central1, us-east1, us-east4, us-south1, us-west1, us-west2, us-west3, us-west4	n2-standard-{4,8,16,32} (2/16, 4/32, 8/64, 16/128)	52	28,620	28.6M	6.8K	0.024%	29	2,236	2.2M	55	0.002%

before falling back to Linux (§ 4.2, § 5.1). In this experiment, the host backs 30% of the guest memory with huge pages and the remaining 70% with base pages. We choose 30% because the Azure AMD VMs in our cloud study observe about 70% fake huge pages on average (§ 6.2). We reserve the 30% as host huge pages using `libvirt`'s `memoryBacking` section [16]. The rest is ordinary host memory, which the host maps with scattered base pages while Kernel Samepage Merging (KSM) [58] is active [51, 53]. In this setup, about 30% of the candidates that `poold` probes are real. Before each run with vHuge, we let the pool fill to 3,500 real huge pages. All results in this section hold for this 30% configuration; the benefit of vHuge is smaller when the host provides more real huge pages and is zero when all guest huge pages are real (§ 6.5).

Throughput-oriented workloads are shown as normalized throughput, while latency-sensitive workloads are shown as normalized end-to-end latency, so lower is better for the three workloads to the right of the separator. Across all 13 workloads, vHuge improves performance by 20.45% on average after converting latency reductions into positive improvements. The largest improvement is GUPS, which improves by 97.1%; its bar is capped in the figure for readability. Across these workloads, vHuge reduces data TLB (dTLB) misses by 66.08% on average and instruction TLB (iTLB) misses by 33.48% on average; the 49.78% TLB miss reduction that we report is the mean of the two. GUPS has the largest reductions for both metrics, at 99.97% for dTLB misses and 76.05% for iTLB misses. Figure 7 also reports the dTLB and iTLB misses with vHuge, normalized to the respective vanilla Linux miss counts. Lower is better for both TLB metrics. For these counters, we report the mean of ten runs for each configuration.

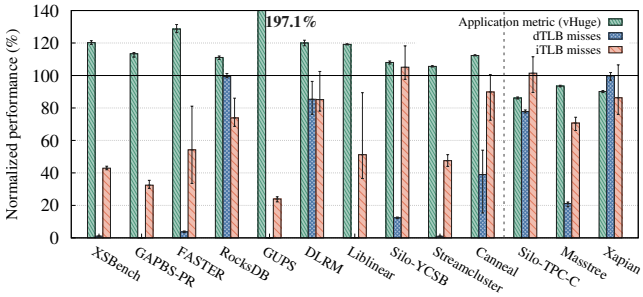


Figure 7: Application metrics and TLB misses with vHuge, normalized to vanilla Linux at 100%. Higher application performance is better for throughput-oriented workloads; lower latency is better for workloads to the right of the dashed separator. Lower is better for dTLB and iTLB misses. Workloads are summarized in Table 5.

6.4 Hot Mapping Repair Under Scarcity

Pool placement only helps create limited new real THPs, which may hold cold data or become fake over time. We therefore evaluate whether vHuge can use DAMON's hotness observations and a limited supply of real huge pages to back the hot regions of an existing process. The controlled THP experiment in Figure 2(b) already shows that workloads lose performance when effective huge page coverage falls. Here, we evaluate whether vHuge can preserve that benefit by selecting the regions that receive the remaining real huge pages.

The benefit of this selection depends on how accesses are distributed across a workload's address space. Figure 8 shows two examples from the DAMON profiles used by vHuge. FASTER repeatedly accesses several concentrated address ranges, while GUPS spreads its random updates across a wider part of its active mapping. These examples show the skewed hot-access patterns that DAMON reports, and vHuge uses them to select the PMD candidates that it later probes and repairs [38, 42].

For each workload, we calibrate 100% real huge page availability to its peak anonymous-memory footprint, expressed as the number of 2 MiB huge pages. We then limit the host huge pages available to back the VM to 75%, 50%, 25%, or 10% of that value. This limit does not prevent the guest from promoting memory to THPs; it only limits the *effectiveness* of those THPs in use. Vanilla Linux uses THP set to `always`, so it can still promote guest huge pages but cannot select which ones receive the limited real host huge page backing.

We compare two ways of using real huge pages from the pool, each limited to the same number of them. *vHuge without DAMON*

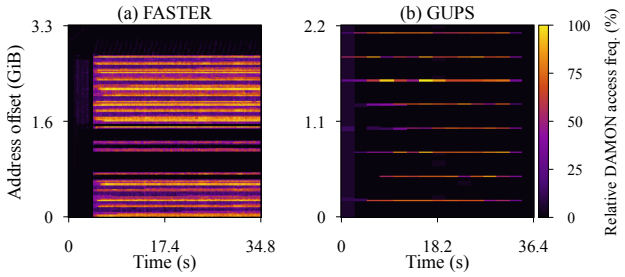


Figure 8: DAMON's observed access frequency over time and virtual address offset for FASTER and GUPS. Each workload is normalized to its maximum observed DAMON access frequency.

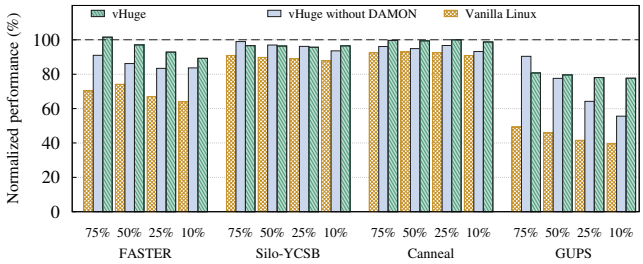


Figure 9: Performance when the real huge pages available to a VM are limited to 75%, 50%, 25%, or 10% of the workload's peak anonymous footprint. vHuge without DAMON gives them out in allocation order; vHuge migrates hot fake huge pages to them. Values are normalized to vHuge with 100% real huge page availability for each workload.

gives a real huge page to each new huge page fault of the workload until the supply is used up; later faults receive whatever Linux allocates. Its real huge pages are real from their first access and need no monitoring or migration, but which regions receive them depends on the order in which the workload allocates memory, not on how often it accesses them. *vHuge* lets Linux allocate every huge page as usual, uses DAMON to find the hot ones, and migrates the fake hot ones to real huge pages (§ 4.3). This directs the real huge pages to the hot regions, at a cost: a hot huge page stays fake until it is found and classified, and each migration copies 2 MiB and briefly blocks access to that page.

Figure 9 normalizes all configurations to vHuge with 100% real huge page availability for the same workload. Both vHuge configurations outperform vanilla Linux at every availability level. Which of the two is better depends on how much there is to gain from the limited selection of real huge pages. When real huge pages are scarce, allocation order leaves most hot regions on fake huge pages, and the gain from repair outweighs its cost [40, 54]: for FASTER, vHuge is 6.6% to 12.6% faster than vHuge without DAMON. When real huge pages are plentiful, or when a workload's accesses are spread more evenly as in GUPS, allocation order already covers most hot regions. Selection then has little left to gain, and the cost of repair can make vHuge slightly slower than vHuge without DAMON. Workloads that are less sensitive to huge pages, such as Silo-YCSB, show the benefit of selection only at the lowest availability levels.

Table 5: List of workloads used in evaluation.

App.	Workload description
XSBench	Monte Carlo neutron cross-section lookup proxy [64].
GAPBS-PR	PageRank graph analytics from the GAP Benchmark Suite [5].
FASTER	Concurrent key-value store with YCSB-B requests [8, 12].
RocksDB	Persistent key-value store serving random reads [17].
GUPS	RandomAccess benchmark measuring random memory updates [41].
DLRM	Recommendation model with large embedding-table lookups [26].
Liblinear	Sparse linear classification benchmark [18].
Silo-YCSB	In-memory YCSB transactions with Silo concurrency control [49, 65].
Streamcluster	Online clustering workload from the PARSEC benchmark suite [6].
Canneal	Simulated annealing workload from the PARSEC benchmark suite [6].
Silo-TPC-C	TailBench in-memory transactional database with TPC-C [14, 32, 65].
Masstree	TailBench in-memory key-value store using Masstree [32, 43].
Xapian	TailBench search engine workload using Xapian [32, 62].

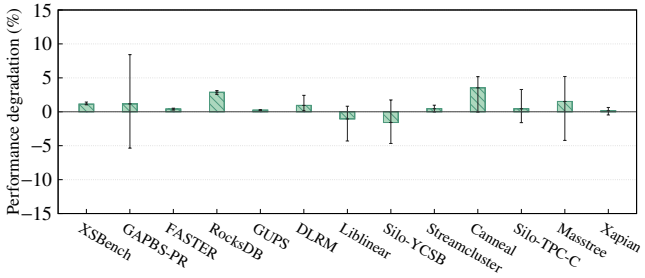


Figure 10: End-to-end overhead of vHuge when VM memory is fully backed by host huge pages. Bars show the mean, whiskers show the minimum and maximum across measured runs, and negative values indicate speedup.

Table 6: Component costs of vHuge.

Component	Unit	Mean cost
Allocation and pool maintenance		
Allocation hook	PMD fault	1.2 μs empty; 9.2 μs hit
Pool refill	256 candidates	0.59 ms/page
Pool validation	512 pages / 10 s	0.335%/core
Hot page repair		
DAMON VADDR aggregation	VADDR aggregation	0.0253%/core
Unknown PMD probe	Hot unknown PMD	9.134 μs/probe
Migration repair	Migrated PMD	0.649 ms/repair

6.5 Overhead and Repair Cost

For this experiment, the host backs all of the VM's memory with huge pages. Thus, every huge page is real with both vanilla Linux and vHuge, and vHuge cannot improve page quality. The performance difference between the two is therefore the cost of vHuge's allocator hook, periodic pool refill, and revalidation. Figure 10 shows that this cost is small: vHuge adds 0.785% overhead on average across the overhead workload suite. Liblinear and Silo-YCSB slightly improve in this experiment, but the setup does not give vHuge a page quality advantage; we treat those small improvements as run-to-run variation that should approach zero with more repetitions.

Table 6 separates the component costs. The allocation hook checks the real huge page pool on the PMD fault path, but most vHuge work is outside application fault handling. Refilling the pool allocates candidate THPs, probes them, and returns only real pages

to the kernel pool (§ 4.2, § 5.1). Most of the refill batch cost comes from allocating and faulting the candidate THPs through the normal Linux allocation path. The daemon then probes those pages and returns only real candidates to the pool, so this work remains off application faults. Validation periodically maps pooled pages back into the daemon and re-probes them so stale real huge page records are removed (§ 5.1). The DAMON VADDR aggregation and unknown PMD probe costs are each averaged over 1,000 consecutive samples. When repeated refills leave the pool below the configured watermark, vHuge starts DAMON to find hot variable-sized virtual regions in registered processes, resolves the PMD mappings that overlap those regions, probes unknown hot pages, and migrates hot fake pages to available real pool pages (§ 4.3, § 5.2). The unknown PMD probe number is the time for one in-kernel probe of a hot PMD THP. The migration number is measured under a controlled repair benchmark, where the number of migrated huge pages is known. It is therefore a per-repair cost, not the time to discover that a natural workload page became fake.

7 Discussion

Comparison with host-side huge page managers. Ingens and HawkEye manage huge page promotion, splitting, compaction, and migration in the host memory manager, while Gemini makes the host find the GPA ranges of guest huge pages and back them with host huge pages [28, 35, 50]. These systems *create more* real huge pages than the VM would otherwise receive, and they require changing the host. Because the two approaches differ in what they control, we do not compare them experimentally. A host-side system raises the upper bound on the real huge pages a VM receives, while a guest-only system like vHuge makes the best use of the real huge pages it already has. The two are complementary: host-side management does not guarantee that every guest huge page is real.

Scope: anonymous huge pages. vHuge currently manages anonymous 2 MiB THPs [61], which give huge pages to unmodified applications and can be moved with Linux's existing page migration [60, 69]. The TLB probe itself can classify any 2 MiB guest mapping, but using its result requires changes in the subsystem that owns the mapping, which we leave for future work. *Hugetlbfs* pages come from a pool that the administrator reserves in advance [39]; supporting them means probing that pool and choosing real pages inside it. *File-backed* huge pages belong to the page cache, where the same contents can be mapped by several processes or accessed without being mapped [48], so the per-process ownership records and the migration path of vHuge do not apply directly. *Shared memory* (shmem/tmpfs) has its own allocation policy and choice of huge page sizes [4].

Limitations. (1) vHuge cannot make the host create huge pages. When the host provides few real huge pages, `poold` backs off (§ 4.2) and Linux allocates huge pages as usual; vHuge's measurements can then still inform decisions such as moving the workload to another VM. (2) While the pool is above its low watermark, vHuge does not probe a huge page again after an application has received it (§ 4.3), so a page whose host backing changes stays fake until it is freed or repair starts, and hot fake huge pages left over from a period of scarcity are not repaired after the pool recovers. A pooled page can also become fake between two revalidations, which are 10 seconds apart. (3) We evaluate application performance only on local VMs with a controlled fraction of real huge pages, and we cannot verify the cloud classifications against the host page tables (§ 6). (4) The prefetcher method works only on processors whose prefetcher treats base pages and huge pages differently; among our machines, this is Emerald Rapids.

8 Related Work

Huge page management. Operating systems use huge pages to reduce translation overhead [20, 42, 47, 52, 71]. Linux THP creates them opportunistically, but their benefit is reduced by fragmentation, compaction delays, and page splitting [30, 35, 51, 53, 68]. vHuge asks a different question: after the guest receives a huge page, is that page backed by a host huge page?

Virtualized huge pages. Gemini, HugeGPT, and Ingens reduce guest-host misalignment with host support [28, 29, 35] (§ 7). vHuge classifies page quality from inside the VM and therefore works when the host cannot be changed [23, 36].

Guest-side hardware inference. CacheInspector, vSched, and CacheX recover cache or scheduling information from inside the guest through timing [13, 21, 22, 56, 63, 67]. vHuge applies the same approach to huge page backing.

Hot page placement. Several systems identify hot memory and place it on the faster resource, such as DRAM rather than slower memory [33, 40, 42, 46, 54, 69, 72]. vHuge is complementary: it supplies the page quality signal for VMs, i.e., which guest huge pages are real and therefore worth giving to hot data [11, 31].

9 Conclusion

This paper introduces vHuge, which probes from inside a VM whether a guest huge page is backed by a host huge page, without hypervisor or hardware support. It gives verified real huge pages to new allocations and migrates hot fake huge pages onto real ones when real huge pages are scarce. Our measurements across 636 VMs on AWS, Azure, and Google Cloud confirm that some guest huge pages are backed by host base pages in production environments. On local VMs in which the host backs 30% of the memory with huge pages, vHuge improves the performance of 13 workloads by 20.45% on average and by as much as 97.1%, and reduces TLB misses by 49.78% on average with an average overhead of 0.785%.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This work was funded in part by NSF grants CSR-2324923, CSR-2440456, and CRII-SHF-2348066.

References

[1] AMD. 2026. AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://docs.amd.com/v/u/en-US/24593_3.44_APM_Vol2

[2] Kapil Arya, Yury Baskakov, and Alex Garthwaite. 2014. Tesseract: reconciling guest I/O and hypervisor swapping in a VM. *Acm Sigplan Notices* 49, 7 (2014), 15–28.

[3] Anthony O Ayodele, Jia Rao, and Terrance E Boult. 2015. Performance measurement and interference profiling in multi-tenant clouds. In *2015 IEEE International Conference on Cloud Computing*. IEEE, 941–949.

[4] Baolin Wang. 2024. Support large folios for tmpfs. https://lwn.net/Articles/997497/ (Accessed: 10-12-2026).

[5] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015). https://arxiv.org/abs/1508.03619

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08)*. ACM, 72–81. doi:10.1145/1454115.1454128

[7] Bao Bui, Djob Mvondo, Boris Teabe, Kevin Jiokeng, Lavoisier Wapet, Alain Tchana, Gaël Thomas, Daniel Hagimont, Gilles Muller, and Noel DePalma. 2019. When eXtended Para - Virtualization (XPV) Meets NUMA. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, 1–15. doi:10.1145/3302424.3303960

[8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD/PODS '18)*. ACM, 275–290. doi:10.1145/3183713.3196898

[9] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers* 44, 5 (1995), 609–623. doi:10.1109/12.381947

[10] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. 2013. Working set-based physical memory ballooning. In *10th International Conference on Autonomic Computing (ICAC '13)*. 95–99.

[11] Cloud Alibaba. 2026. HugeText. https://www.alibabacloud.com/help/en/alinux/user-guide/hugetext Accessed: 07-14-2026.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SOCC '10)*. ACM, 143–154. doi:10.1145/1807128.1807152

[13] Keith Cooper and Xiaoran Xu. 2018. Efficient characterization of hidden processor memory hierarchies. In *Computational Science–ICCS 2018: 18th International Conference, Wuxi, China, June 11–13, 2018 Proceedings, Part III 18*. Springer, 335–349.

[14] Transaction Processing Performance Council. 2005. Transaction processing performance council. *Web Site, http://www. tpc. org* (2005). https://www.tpc.org/tpcc/

[15] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: a holistic approach to memory placement on NUMA systems. *ACM SIGPLAN Notices* 48, 4 (2013), 381–394.

[16] Libvirt Documentation. 2026. Domain XML Format. https://libvirt.org/formatdomain.html#memory-backing

[17] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32. doi:10.1145/3483840

[18] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A Library for Large Linear Classification. *Journal of Machine Learning Research* 9 (2008), 1871–1874. https://www.jmlr.org/papers/v9/fan08a.html

[19] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-harvesting vms in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 583–594. doi:10.1145/3503222.3507725

[20] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large pages may be harmful on {NUMA} systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 231–242.

[21] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *Proceedings 2017 Network and Distributed System Security Symposium (NDSS 2017)*. Internet Society. doi:10.14722/ndss.2017.23271

[22] Edward Guo, Weiwei Jia, Xiaoning Ding, and Jianchen Shan. 2025. Optimizing Task Scheduling in Cloud VMs with Accurate vCPU Abstraction. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*. ACM, 753–768. doi:10.1145/3689031.3696092

[23] 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*. 31–37.

[24] Intel. 2025. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf

[25] Intel. 2025. Profile KVM Kernel and User Space on the KVM System. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/on-kvm-project-system.html

[26] Rishabh Jain, Scott Cheng, Vishwas Kalagi, Vrushabh Sanghavi, Samvit Kaul, Meena Arunachalam, Kiwan Maeng, Adwait Jog, Anand Sivasubramaniam, Mahmut Taylan Kandemir, and Chita R. Das. 2023. Optimizing CPU Performance for Recommendation Systems At-Scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. ACM, 1–15. doi:10.1145/3579371.3589112

[27] Weiwei Jia, Cheng Wang, Xusheng Chen, Jianchen Shan, Xiaowei Shang, Heming Cui, Xiaoning Ding, Luwei Cheng, Francis CM Lau, Yuexuan Wang, et al. 2018. Effectively Mitigating I/O Inactivity in vCPU Scheduling. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 267–280.

[28] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, and Xiaoning Ding. 2023. Making Dynamic Page Coalescing Effective on Virtualized Clouds. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. ACM, 298–313. doi:10.1145/3552326.3567487

[29] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, Yiming Du, Xiaoning Ding, and Tianyin Xu. 2023. HUGEGPT: Storing Guest Page Tables on Host Huge Pages to Accelerate Address Translation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '23)*. IEEE, 345–357. doi:10.1109/PACT58117.2023.00014

[30] Jonathan Corbet. 2009. Transparent hugepages. https://lwn.net/Articles/359158/ (Accessed: 10-15-2023).

[31] Jonathan Corbet at LWN. 2022. Yet another try at the BPF program allocator. https://lwn.net/Articles/915005/

[32] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. doi:10.1109/iiswc.2016.7581261

[33] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 163–181.

[34] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: The Linux Virtual Machine Monitor. In *Proceedings of the 2007 Linux Symposium*. 225–230. https://www.kernel.org/doc/mirror/ols2007v1.pdf#page=225

[35] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 705–721.

[36] Chuandong Li, Sai Sha, Yangqing Zeng, Xiran Yang, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2023. FHPM: Fine-grained Huge Page Management For Virtualization. *arXiv preprint arXiv:2307.10618* (2023).

[37] Linux Documentation. 2026. DAMON-based LRU-lists Sorting. https://www.kernel.org/doc/html/next/admin-guide/mm/damon/lru_sort.html#hot-thres-access-freq

[38] Linux Documentation. 2025. DAMON: Data Access MONitor. https://docs.kernel.org/mm/damon/index.html

[39] Linux Kernel Documentation. 2026. HugeTLB Pages. https://docs.kernel.org/admin-guide/mm/hugetlbpage.html (Accessed: 06-25-2026).

[40] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered memory management beyond hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 731–747.

[41] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Vol. 213. 1. doi:10.1145/1188455.1188677

[42] Aninda Manocha, Zi Yan, Esin Tureci, Juan L Aragón, David Nellans, and Margaret Martonosi. 2023. Architectural support for optimizing huge page selection within the OS. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1213–1226. doi:10.1145/3613424.3614296

[43] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*. 183–196. doi:10.1145/2168836.2168855

[44] Timothy Merrifield and H. Reza Taheri. 2016. Performance Implications of Extended Page Tables on Virtualized x86 Processors. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '16)*. ACM, 25–35. doi:10.1145/2892242.2892258

[45] Michael Schwarz. 2026. PTEditor. https://github.com/misc0110/PTEditor (Accessed: July 2026).

[46] Alan Nair, Sandeep Kumar, Aravinda Prasad, Ying Huang, Andy Rudoff, and Sreenivas Subramoney. 2024. Telescope: telemetry for gargantuan memory footprint applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 409–424.

[47] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 89–104.

[48] Neil Brown. 2016. Transparent huge pages in the page cache. https://lwn.net/Articles/686690/ (Accessed: 06-25-2026).

[49] Riki Otaki. 2023. tpcc-runner: An Implementation of Various In-memory Concurrency Control Protocols. https://github.com/rotaki/tpcc-runner. (Accessed: 07-05-2026).

[50] Ashish Panwar, Sorav Bansal, and K Gopinath. 2019. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and*

Operating Systems. 347–360.

[51] 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 (APSys '16)*. ACM, 1–8. doi:10.1145/2967360.2967371

[52] Ashish Panwar, Harshitha Patil, 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 (ASPLOS '18)*. ACM, 679–691. doi:10.1145/3173162.3173203

[53] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. 2020. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA '20)*. IEEE, 913–925. doi:10.1109/ISCA45697.2020.00079

[54] Chandra Prakash, Aravinda Prasad, Sandeep Kumar, and Sreenivas Subramoney. 2025. Efficient Memory Tiering in a Virtual Machine. *arXiv preprint arXiv:2506.06067* (2025).

[55] Xiaowei Shang, Weiwei Jia, Jianchen Shan, and Xiaoning Ding. 2021. CoPlace: Effectively Mitigating Cache Conflicts in Modern Clouds. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 274–288. doi:10.1109/pact52795.2021.00027

[56] Weijia Song, Christina Delimitrou, Zhiming Shen, Robbert Van Renesse, Hakim Weatherspoon, Lotfi Benmohamed, Frederic De Vaulx, and Charif Mahmoudi. 2021. CacheInspector: Reverse Engineering Cache Resources in Public Clouds. *ACM Transactions on Architecture and Code Optimization* 18, 3 (June 2021), 1–25. doi:10.1145/3457373

[57] The Linux Kernel Documentation. 2026. Examining Process Page Tables. https://docs.kernel.org/admin-guide/mm/pagemap.html (Accessed: 07-11-2026).

[58] The Linux Kernel Documentation. 2026. Kernel Samepage Merging. https://docs.kernel.org/admin-guide/mm/ksm.html (Accessed: 07-11-2026).

[59] The Linux Kernel Documentation. 2026. Memory Management APIs. https://docs.kernel.org/core-api/mm-api.html (Accessed: 07-13-2026).

[60] The Linux Kernel Documentation. 2026. Page migration. https://docs.kernel.org/mm/page_migration.html (Accessed: 07-11-2026).

[61] The Linux Kernel Documentation. 2026. Transparent Hugepage Support. https://docs.kernel.org/admin-guide/mm/transhuge.html (Accessed: 06-25-2026).

[62] The Xapian Project. 2026. The Xapian Project. https://xapian.org/. (Accessed: 07-05-2026).

[63] Mani Tofigh, Edward Guo, Weiwei Jia, Xiaoning Ding, Zirui Neil Zhao, and Jianchen Shan. 2025. Optimizing CPU Cache Utilization in Cloud VMs with Accurate Cache Abstraction. *arXiv preprint arXiv:2511.09956* (2025).

[64] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *Proceedings of the International Conference on Physics of Reactors (PHYSOR '14)*. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[65] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 18–32. doi:10.1145/2517349.2522713

[66] Rik van Riel and Vinod Chegu. 2014. Automatic NUMA balancing. *Red Hat Summit* (2014).

[67] Georgios Vavouliotis, Gino Chacon, Lluc Alvarez, Paul V Gratz, Daniel A Jiménez, and Marc Casas. 2022. Page size aware cache prefetching. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 956–974.

[68] 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 ATC '23)*. 897–914.

[69] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*. 19–35.

[70] Xiong Zhang. 2025. Introduce Passthrough vPMU. https://lore.kernel.org/all/20240126085444.324918-1-xiong.y.zhang@linux.intel.com/

[71] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation ranger: operating system support for contiguity-aware tlbs. In *Proceedings of the 46th International Symposium on Computer Architecture*. 698–710.

[72] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. 2025. cache_ext: Customizing the Page Cache with eBPF. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 462–478.