How to Move an Immovable Page

David Zalman
Technion — Israel Institute of Technology
Haifa, Israel
davidzalman@campus.technion.ac.il

Nizan Kafman-Raz
Technion — Israel Institute of Technology
Haifa, Israel
nizan.kafman@campus.technion.ac.il

Omer Daube
Technion — Israel Institute of Technology
Haifa, Israel
omer.daube@campus.technion.ac.il

Nadav Amit
Technion — Israel Institute of Technology
Haifa, Israel
namit@technion.ac.il

Abstract

I/O devices require fixed physical addresses for the memory they access, forcing the OS to pin pages in place. This locks out memory management operations that depend on page migration—NUMA rebalancing, defragmentation, memory tiering, and hardware error recovery. The problem is acute for device-assigned virtual machines, where the entire guest address space (tens to hundreds of gigabytes) remains pinned for the VM's lifetime. Existing alternatives demand I/O page-fault support that most devices lack, or add per-I/O overhead incompatible with persistently mapped buffers.

We present *virtual pinning*, a migration protocol that keeps device mappings permanently valid while safely relocating the underlying physical page. Virtual pinning atomically redirects each virtual-to-physical translation to the new location, using hardware ordering guarantees to detect concurrent writes and ensure no update is lost. We implement virtual pinning in the Linux kernel and evaluate it on device-assigned VMs running PostgreSQL and on bare-metal RDMA. Virtual pinning can reduce query latency by 16% and RDMA latency by 29% through NUMA-aware migration of pinned pages, and restores huge page availability that pinning previously destroyed—all impossible with standard pinning on existing hardware.

CCS Concepts: • **Software and its engineering → Virtual memory**; *Input / output*; *Virtual machines.*

Keywords: page migration, pinned memory, IOMMU, DMA, RDMA, virtualization, device passthrough

ACM Reference Format:
David Zalman, Nizan Kafman-Raz, Omer Daube, and Nadav Amit. 2027. How to Move an Immovable Page. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3845814.3850068



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850068

1 Introduction

Virtual memory gives the OS flexibility to relocate physical pages transparently—defragmenting memory to form huge pages, balancing data across Non-Uniform Memory Access (NUMA) nodes, tiering pages between fast and slow memory, and relocating data away from failing hardware. But this flexibility breaks down when pages are *pinned*: locked at a fixed physical address because the accessor—the kernel or a device—cannot tolerate faults. The kernel pins pages it must reach in atomic contexts, where it cannot sleep, and I/O device drivers pin memory for Direct Memory Access (DMA), where the device reads and writes physical addresses directly without CPU involvement. If a page moves while a device is writing to it, data is silently corrupted.

The consequences are severe and growing. Pinning reduces huge page allocation success rates to as low as 5% on average [38]; at Meta's datacenters, unmovable allocations—dominated by networking buffers—scatter across 31% of physical memory [56]. Device-assigned virtual machines (VMs) are the worst case: the entire guest memory—often tens or hundreds of gigabytes—must be pinned for the VM's lifetime, blocking NUMA rebalancing and defragmentation of that memory entirely. As device passthrough becomes the default for high-performance I/O, the cost of immovable memory will only increase.

Existing approaches address facets of this problem but none solves it. The PCIe standard defines Address Translation Services with Page Request Interface (ATS/PRI), which lets devices request fault resolution from the OS, but fewer than 10% of deployed servers support it and no commodity NIC implements the device side [31, 51]. Mellanox's On-Demand Paging (ODP) provides similar functionality for RDMA, but is limited to individually registered memory regions and does not help device-assigned VMs. Even where I/O page-fault support exists, fault latencies of 10–427 μs can cascade into hundreds of milliseconds of network interruption [16, 18]. Proposed software approaches each target a specific I/O subsystem and require specialized hardware [51]

or invasive protocol changes [44]. Input/Output Memory Management Unit (IOMMU) emulation [5] can track active DMA mappings but imposes prohibitive per-I/O overhead. Others require vendor firmware [40] or last-level cache extensions [56] unavailable today. None enables transparent *migration* of pinned pages on commodity hardware.

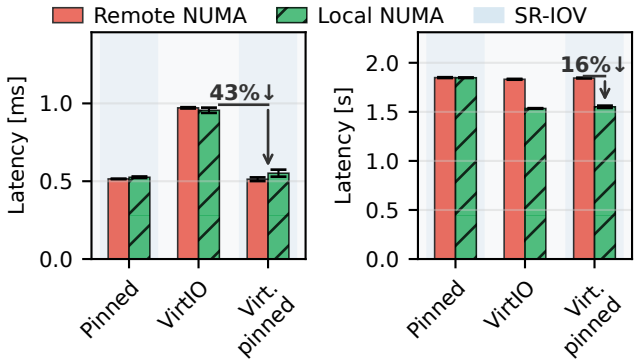
The closest prior work relaxes a different constraint. Nomad [53] makes migration non-blocking for *userspace* accesses: it keeps the source page readable while copying and uses the dirty bit to abort if a write intervenes. But between clearing the old page-table entry and installing the new one, the page is briefly *unmapped*. That is safe only because a user thread that touches it faults and waits. Absent ATS/PRI, a device gets no such chance: an absent IOMMU translation is not a fault to resolve but a lost write or a device abort, and a kernel access in atomic context cannot wait either. That unmapped window, not the choice of dirty bit over access bit, is what keeps Nomad from migrating pages under I/O.

We present *virtual pinning*, a software-only mechanism that decouples the mapping-validity guarantee from physical-location fixity on existing hardware. The key observation is that DMA and kernel access do not require a fixed physical address—only that no accessor ever finds one missing. Virtual pinning guarantees exactly that: no access, whether through the Memory Management Unit (MMU) or the IOMMU, ever encounters a missing translation, because migration redirects each live mapping present-to-present, in one atomic update.

The central challenge is ensuring that no write to the old page goes undetected during the copy: a lost write would silently corrupt data. Our protocol exploits a hardware ordering guarantee—both the MMU and IOMMU set the access bit in the page-table entry *before* the corresponding data access becomes visible—to detect any concurrent access and safely abort the migration, without ever exposing an invalid translation. Our protocol is compatible with Intel, AMD, and RISC-V, whose MMUs and IOMMUs provide the required ordering (§3.3). Arm's SMMU specification does not currently permit the required present-to-present update. We implement virtual pinning in Linux 6.19 (~2,700 SLoC) with no device firmware or hardware modifications. Virtual pinning applies wherever the IOMMU maintains access bits and the device operates in a translated domain, and enabling a pinning subsystem takes a small one-time adaptation (§3.6).

Keying on the access bit rather than the dirty bit is deliberately conservative: a concurrent *read*, which a dirty-bit scheme would tolerate, aborts our commit. Under a tight loop of RDMA atomics aimed squarely at the pages being migrated, 1.3% of migrations abort over a 128-page working set; in the pointer-chase workload, where DMA and migration touch different pages, none do (§4.6). An abort is safe: the page stays put and is retried.

We evaluate virtual pinning on memory defragmentation, device-assigned VMs, and bare-metal Remote DMA (RDMA) (Figure 1). For compaction, pinning 32 GB of RDMA-registered memory destroys 2 MB huge page availability (90% to 41% of the node's memory); once the pages can move, defragmentation recovers it to 89% (§4.4). For a VM running PostgreSQL, Device passthrough achieves 43% lower network latency than paravirtual I/O, but pins all guest memory, blocking NUMA migration. Virtual pinning delivers both: passthrough network speed and 16% lower hash-join latency after migrating to the local NUMA node. For bare-metal RDMA, virtual pinning reduces pointer-chasing latency by 29%—matching On-Demand Paging with lower transient latency and no device-side page-fault support.



(a) Point-lookup latency **(b)** Hash join latency

Figure 1. Only virtual pinning (rightmost) achieves both low network latency and NUMA-optimal placement in a PostgreSQL VM. (a) An assigned device achieves 43% lower latency than VirtIO (paravirtual I/O). (b) NUMA migration cuts hash join latency by 16%, but a pinned assigned device cannot migrate (§4.1).

Our contributions are:

- A software-only migration protocol for pinned pages that atomically redirects device and kernel mappings on commodity hardware, using access-bit ordering to guarantee that no concurrent DMA write is lost.
- A Linux kernel implementation that integrates with the Intel VT-d IOMMU to support atomic page redirection for both CPU and DMA mappings; adapting an I/O subsystem (RDMA, `io_uring`, splice, the DMA-IOMMU layer) took under 100 lines of new code each. The same interface applies to AMD and RISC-V IOMMUs.
- An evaluation on device-assigned VMs, bare-metal RDMA, and `io_uring`, showing NUMA migration and defragmentation of pinned pages—impossible with standard pinning today.

2 Background

We first describe why memory migration matters, then explain how pinning blocks it, and survey existing solutions.

2.1 The Need for Memory Migration

Memory migration—the ability to relocate physical pages while preserving virtual address mappings—is critical for performance in modern systems.

NUMA Balancing. In NUMA systems, remote memory access is 1.5–2× slower than local access; proper page placement can reduce execution time by over 40% [1, 11, 34, 50].

Transparent Huge Pages. Modern processors support huge pages (2 MB or 1 GB) to reduce TLB pressure, improving performance by up to 18% [56]. Allocating contiguous physical memory for huge pages requires migrating existing pages; without migration, fragmentation accumulates and severely limits THP effectiveness [29, 36, 38, 58].

Memory Tiering. Cache-coherent interconnects such as CXL, together with persistent memory, extend the memory hierarchy beyond local DRAM, enabling memory pooling across hosts [32] and a path for VM live migration via shared memory [45]. Exploiting these tiers requires migrating hot pages to fast memory and cold pages to slower, higher-capacity tiers [2, 14, 30, 35, 46, 52, 54].

Error Recovery. Approximately 8% of DIMMs experience ECC-correctable errors per year [43]. The OS can migrate data away from degrading memory before single-bit errors accumulate into multi-bit failures that ECC cannot correct—but pinned pages cannot be relocated, leaving them vulnerable.

2.2 Memory Pinning

Memory pinning prevents pages from being paged out or migrated; if a page moves while an accessor holds its physical address, data might be silently corrupted. Pinning is required in two distinct scenarios: when the kernel needs to access user memory directly, and when I/O devices perform DMA operations.

Kernel-Initiated Accesses. The kernel frequently needs to access user-space memory directly, bypassing the user's page tables. This occurs when the kernel must access user buffers while the process is not running, from interrupt context, or when the kernel needs write access to a page that is write-protected in the user's page tables [20]. The kernel accesses such pages through the linear mapping—a direct one-to-one mapping of physical memory into kernel address space.

DMA Accesses. I/O devices access memory via DMA, bypassing the CPU. The device is programmed with physical addresses (or IOMMU virtual addresses that map to fixed physical locations). This affects all high-performance I/O: zero-copy networking, device assignment for virtual machines [55], and RDMA [8, 47]. Device assignment is usually implemented with Single-Root I/O Virtualization (SR-IOV), which splits one physical device into virtual functions that a guest drives directly.

Pinning duration varies across use cases. Transient pins (direct I/O, `vmsplice`, streaming DMA) last milliseconds or less and can usually be waited out by retrying migration after a brief delay. Long-term pins are the real obstacle: long-lived I/O buffers (e.g., `io_uring` registered buffers) remain pinned for the lifetime of the application; device assignment for VMs pins the entire guest memory for the VM's lifetime, potentially hundreds of gigabytes [40]; RDMA memory regions remain pinned as long as remote nodes may access them [8, 47]. These cannot be waited out and require a fundamentally different approach.

Linux tracks pinning by elevating the page's reference count once per pin [20]; the migration path checks this count and refuses to relocate any page whose count indicates outstanding pins.

2.3 Existing Solutions

The most widely discussed solution is I/O page faults: the PCIe standard defines Address Translation Services with Page Request Interface (ATS/PRI), which enables devices to request page-fault resolution from the OS. Mellanox's proprietary On-Demand Paging (ODP) provides similar functionality for RDMA. Both allow the OS to unmap pages and rely on the device to re-request translations on access. However, fewer than 10% of servers support ATS/PRI on the IOMMU and no commodity NIC implements PRI. Even where supported, fault latencies of tens to hundreds of microseconds cascade into hundreds of milliseconds of network interruption through queue blocking and retransmission amplification [16, 18, 31, 51]. In practice, these mechanisms are designed to avoid upfront pinning by lazily resolving device mappings on fault, not to *migrate* pages that are already mapped for device access. Nor would wider deployment settle the question. Driving an ATS/PRI-capable accelerator at full rate for 60 s while migrating its buffers, we measured 1.12M I/O page requests, because the standard migration path unmaps each page before copying it. Keeping the mapping present throughout (§3) sustains the same throughput (4.4 GB/s versus 4.3 GB/s) with *zero* page requests. The asymmetry matters: a page request stalls the device engine, whereas a migration that gives up costs only a host-side retry.

Other approaches—software, emulation-based, and device-assisted—each address a facet of the problem but none enables transparent migration of pinned pages on commodity hardware; we discuss them further in §6.

3 Design

Memory pinning conflates two distinct guarantees: preventing pages from being swapped out, and preventing them from being migrated. The first is essential—a DMA operation cannot wait for a page to be fetched from disk. But the second is an artifact of implementation, not a fundamental

requirement. What DMA and kernel access actually need is for mappings to remain valid; they are indifferent to which physical page backs that mapping, as long as the data is preserved. Linux already makes this distinction for CPU-accessed memory: the `mlock` system call prevents swap-out while still allowing the kernel to migrate pages between NUMA nodes or for compaction. Virtual pinning extends this principle to DMA-pinned pages, which today are both non-swappable and non-migratable.

Our approach exploits this observation by creating virtual mappings for kernel and IOMMU access that no accessor ever finds missing, and that can be atomically redirected to point to different physical pages. This requires making kernel mappings relocatable and maintaining reverse mappings from physical pages to the PTEs that reference them. We describe virtual pinning in Linux terms, though nothing in the protocol is specific to Linux.

The key challenge is ensuring consistency during migration: we must guarantee that no writes are lost when we redirect a mapping to a new physical page. The difficulty arises because modern processors cache page-table entries in the TLB (Translation Lookaside Buffer), a hardware address translation cache. When we modify a page-table entry, any CPU core that has the old translation cached in its TLB will continue using the stale mapping until the cache is invalidated.

We cannot simultaneously invalidate all TLBs and perform the migration in a single operation. However, we can detect whether any access occurred while the copy was in progress. Our approach relies on a fundamental ordering property of hardware page-table walkers: when a page-table entry (PTE) is not cached in the TLB, the processor must walk the page tables to resolve the access, and this walk sets the PTE's access bit *before* the data read or write takes effect [22]. If no access occurs during the copy, the bit remains clear, proving that no write was lost and no stale translation is cached. We first describe how kernel and IOMMU mappings are made remappable (§3.1), then present the migration protocol that uses this property to atomically redirect the PTE (§3.2), and verify that the required ordering holds on Intel, AMD, and RISC-V CPU MMUs and IOMMUs (§3.3).

3.1 Remappable Kernel and IOMMU Mappings

Dynamic Kernel Mappings. While remapping DMA transactions can be handled through the IOMMU page tables, kernel access requires special treatment. The kernel typically accesses physical memory through a fixed linear mapping—a direct one-to-one mapping of physical addresses to kernel virtual addresses whose page tables cannot be safely modified at runtime.

We create an alias virtual address for each pinned page using `vmap`—the kernel's dynamic mapping facility. Kernel subsystems access pinned pages through these aliases instead of the linear mapping, allowing migration to redirect the underlying PTE without modifying the fixed linear map.

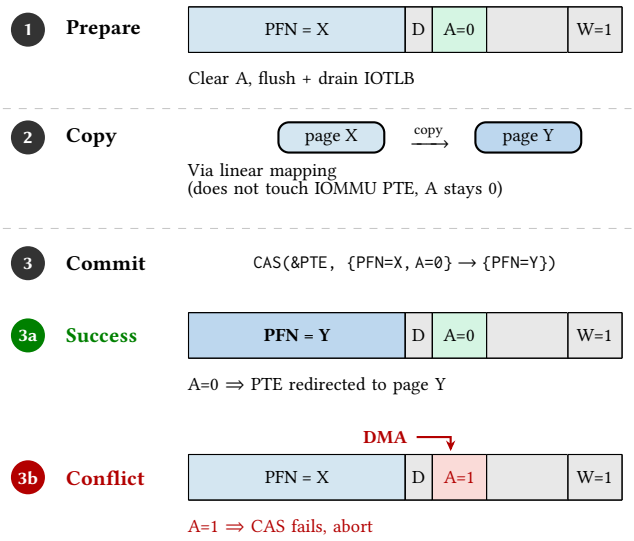


Figure 2. The three-phase migration protocol for an IOMMU PTE (VT-d second-stage layout). The commit uses CAS to atomically redirect the PTE while detecting concurrent DMA writes.

Reverse Mappings. Migrating a page requires locating every PTE that references it. Linux already maintains reverse mappings for userspace PTEs (the `rmap` subsystem), but no such infrastructure exists for kernel or IOMMU mappings. We add per-page reverse mappings for both: a pointer to the `vmap` PTE for kernel access, and the IOMMU domain plus I/O virtual address for DMA access. We track at most one mapping of each type per page, since our protocol can only atomically redirect a single PTE (§3.2); a page with both a kernel and an IOMMU mapping is supported (§3.4), but pages with multiple kernel mappings or multiple IOMMU mappings are detected and left in place.

Superpage Splitting. When the migration granularity is finer than the PTE, superpages must be split before individual pages can be redirected. Linux already splits userspace huge pages during migration; we extend this to IOMMU superpages and to kernel huge-page `vmap` mappings.

3.2 Migration Protocol

Unlike conventional migration, which refuses to relocate pages with elevated reference counts, virtual pinning can migrate a pinned page when its pin is backed by a redirectable mapping—a `vmap` alias and/or an IOMMU PTE that can be atomically updated. The protocol operates in three phases: *prepare*, *copy*, and *commit* (Figure 2).

In the *prepare* phase, we clear the access bit in the fault-intolerant PTE (kernel vmap or IOMMU) and flush the corresponding TLBs and IOMMU TLB (IOTLB) entries—including device-side translation caches when the device uses PCIe ATS (§3.3)—draining any in-flight DMA that used previously cached translations. After the flush completes, no stale translation remains cached; any subsequent access must walk the page tables, and the hardware sets the access bit *before* the read or write takes effect.

In the *copy* phase, we copy the page contents via the kernel's direct (linear) mapping, which does not set the access bit in the vmap or IOMMU PTEs.

In the *commit* phase, we atomically verify that no access occurred and redirect each PTE in a single compare-and-swap (CAS): the expected value is the original PTE with the access bit clear and the old page address; the new value contains the new page address. If no access has re-set the bit, the CAS succeeds and the PTE now points to the new physical location. A clear access bit guarantees two things: first, no write reached the old page during the copy, so no data was lost; second, no translation is cached in any TLB, so no future access can bypass the updated PTE and reach the old page. If a concurrent access has re-set the bit, the CAS fails and the migration is safely aborted (the page remains in place and can be retried later).

The protocol as described operates on a single fault-intolerant PTE per page. After a successful commit, no IOTLB flush is required: the prepare-phase flush evicted the old entry, and the CAS success proves the IOMMU did not re-cache it during the copy. When a page has both a kernel vmap and an IOMMU mapping, we exploit the DMA ownership discipline to simplify the commit (§3.4).

Note that CAS is necessary because the page is writable (W=1 in Figure 2): a concurrent write could modify the old page during the copy, and we must detect this.

This present-to-present PTE update is a deliberate departure from the *break-before-make* (BBM) discipline that operating systems conventionally follow during page migration: invalidate the old PTE, flush the TLB, then install the new entry. OSes use BBM because several architectures require it—Arm mandates it for all present-to-present output-address changes [7], and the POWER ISA prescribes an analogous invalidate–`tlbie`–reinstall sequence. BBM is safe for user mappings because the resulting page fault is recoverable, but it is precisely what fault-intolerant mappings cannot tolerate. On x86 and RISC-V, however, present-to-present PTE updates are architecturally well-defined—the hardware page-table walker maintains coherence and will use either the old or new translation without faulting. Our CAS-based redirect exploits this guarantee, keeping the PTE present and valid at every point; the access-bit guard ensures correctness without ever exposing an invalid translation. The correctness of this protocol depends on architectural ordering guarantees that we examine in §3.3.

Optimizations. The most expensive step in the protocol is the prepare-phase TLB and IOTLB flush. The kernel TLB flush is particularly costly: vmap mappings are global—they can be cached on any core—so flushing them requires a TLB shootdown to all CPUs. Two optimizations reduce this cost.

First, the prepare phase is batched on the IOMMU side: access bits are cleared across all pages in the migration set, and one queued invalidation covers them all. The kernel TLB flush is not batched—each page whose access bit was set costs its own shootdown—because Linux can invalidate one kernel range or all of them, not a list, and the aliases in a batch are scattered across the vmap area. This is why the vmap case is the more expensive one.

Second, most flushes can be eliminated entirely [4]. If a PTE's access bit is already clear before the prepare phase, no TLB entry can be cached for it, so the flush can be skipped. We further exploit this by creating vmap aliases and IOMMU PTEs with the access bit initially clear, since the protocol also leaves the access bit clear after a successful migration; only pages whose access bit was set by a hardware page-table walk between migrations require a flush. For DMA-only workloads such as RDMA, no kernel vmap is involved, so this eliminates all CPU TLB shootdowns from the migration path.

Why the Access Bit, Not the Dirty Bit. Since only writes to the old page during the copy can cause data loss, using the dirty bit as the CAS guard may seem attractive: reads would not trigger an abort, reducing false retries. But the commit must establish whether the old translation is cached *anywhere*. A read during the copy can re-populate the IOTLB with the old translation without leaving a dirty-bit trace, so the CAS succeeds despite a cached stale entry. After commit, a write through that stale entry lands on the old physical page—but the IOMMU's locked dirty-bit update targets the PTE, which now points to the new page. The result is a silent lost write: the data goes to old page while the PTE marks the new page as dirty. Using the access bit avoids this entirely: any access—read or write—sets the bit and causes the CAS to fail, proving that no stale IOTLB entry exists after a successful commit. Note that speculative access-bit updates (e.g., on a mispredicted branch) are harmless false positives: the CAS fails and migration retries later.

The dirty bit could have a use here, just not as the guard. Let the prepare phase clear *both* bits and let each retry re-clear only the access bit, so the dirty bit accumulates across attempts as a record of whether any write has occurred. Clearing it costs no extra atomic: the CAS already compares the whole PTE, so expecting both clear is free.

A retry then re-clears the access bit, flushes, and drains exactly as prepare did. Once that drain completes, any write since prepare must have set the dirty bit, because no stale translation survives the flush and a fresh walk sets both bits before the write takes effect. A retry that finds the dirty

bit still clear therefore knows the source page is byte-for-byte what it copied, and can commit the copy it already has instead of making a new one. Nothing is added to the retry path—the clear, flush and drain happen anyway—so the saving is a full page copy for every abort caused by a read. It grows with page size: a huge-page copy is long enough that an unrelated read is likely to interrupt it. The refinement pays only where aborts are frequent, which is not the regime we measure (§4.6).

3.3 Architectural Ordering Guarantees

The protocol's correctness relies on two hardware properties: **(1) Access-bit ordering**—when a page-table walker sets the access bit in a PTE, the update must be visible to other agents *before* the corresponding data access becomes visible, ensuring that a successful CAS on the access bit proves no concurrent access occurred; and **(2) drain on invalidation**—when a TLB or IOTLB is invalidated, all in-flight accesses that used the now-invalidated entries must complete before the invalidation signals completion. Without drain, the OS could never safely recycle virtual addresses, as a stale in-flight write could corrupt a page mapped at the reused address.

Table 1 summarizes support across four platforms (full quotes in §A). Intel and AMD specify the condition the first property rests on. Intel guarantees that a translation is not cached "unless the accessed flag is 1 in each of the paging-structure entries" [22]; AMD, that a table entry is "never cached in the TLB without its Accessed bit being set at the same time" [12]. RISC-V requires that "the hart [hardware thread] must not perform the memory access that caused the PTE update before the PTE update is globally visible" [23]. Being set is not enough on its own: the commit needs that update to be globally observable, and ordered before the access it precedes. RISC-V states this outright; on Intel and AMD it follows from their conditions because the walker sets the bit atomically, with a locked cycle, which cache coherence orders against our CAS.

On Intel, AMD, and RISC-V, the hardware page-table walker sets the access bit using a locked cycle. Some architectures (notably AMD) do not guarantee that this update is ordered with respect to *regular* loads of the PTE. Our protocol never *relies* on such a load: the only correctness-bearing read of the access bit is the atomic compare inside the commit CAS itself. Both the walker's update and the CAS are locked operations on the same cache line, and cache coherence totally orders locked operations, so exactly one of two interleavings exists: either the walker's update came first, in which case the CAS observes A=1 and fails (migration aborts, the old page remains live); or the CAS came first, in which case any later walk reads the updated PTE and translates to the *new* page. A successful CAS therefore proves no walker cached a translation and no data access was issued during the window—without ever relying on load ordering. RISC-V needs no such argument, its ordering being stated outright above. These three platforms' IOMMUs provide access-bit ordering.

	CPU	IOMMU	
	A-bit	**A-bit**	**Drain**
Intel [21, 22]	✓	✓ (SSADE)	wait desc.
AMD [12, 13]	✓	✓ (HA)	`COMPL_WAIT`
RISC-V [23, 24]	✓	✓ (Svadu)	`IOFENCE`
Arm [3, 6]	✗*	✗†	`CMD_SYNC`

Table 1. Architectural support for virtual pinning's ordering requirements. IOMMU A-bit support requires optional hardware features (shown in parentheses). * A fault-based handler could provide CPU-side ordering on Arm. † Arm's SMMU break-before-make requirement prohibits the present-to-present output-address update our protocol requires (§A).

IOMMU access-bit support is itself an optional hardware feature; without it, the IOMMU does not set access bits and the CAS-based protocol cannot detect concurrent DMA. Drain on invalidation is a baseline IOMMU guarantee, obtained through each architecture's explicit invalidation-completion command: Intel's invalidation-wait descriptor, AMD's `COMPLETION_WAIT`, Arm's `CMD_SYNC`, and RISC-V's `IOFENCE` complete only after in-flight DMA that used the invalidated entries has drained. Linux issues these commands as part of every IOMMU mapping change, and our prepare phase reuses the same flush primitives. The CPU needs no such command: a shootdown is not acknowledged until the remote core has finished with the invalidated entry, which is why table 1 lists drain only for the IOMMU.

PCIe Address Translation Services (ATS). A device using ATS caches translations in its own TLB and issues DMA with pre-translated addresses that bypass the IOMMU walk. Both requirements still hold. The IOMMU sets the access bit before returning a translation to the device (§A), so any translation fetched after the prepare-phase clear re-sets it and fails the commit CAS. The prepare-phase flush invalidates the device TLB as well, and PCIe ordering pushes posted writes that used the invalidated translations ahead of the device's invalidation completion, which the flush's wait descriptor already awaits—so in-flight DMA drains before the copy, as it does without ATS. ATS *without* PRI—a device that caches translations but cannot recover from a fault—is precisely the configuration that benefits most from a guarantee that it never encounters a missing translation.

Of the platforms we examined, Arm is the exception. Access-bit ordering could in principle be achieved via a fault-based CPU handler and SMMU ATS, but the SMMU's break-before-make (BBM) requirement prohibits the present-to-present output-address update our protocol relies on—though it is unclear whether this reflects a true hardware constraint or an overly conservative specification (§A).

3.4 DMA Ownership and Multi-Domain Pages

A page used for device communication is typically mapped in both the kernel (via `vmap`) and the IOMMU, so the protocol must redirect two PTEs. Since it can only atomically redirect one PTE at a time (§3.1), multiple IOMMU mappings to the same page would be problematic; we discuss when this arises and how it can be eliminated in §3.6.

For pages with both a `vmap` and an IOMMU mapping, we rely on the ownership discipline that the kernel's DMA interface requires of device drivers [48]: explicit synchronization calls[1] transfer buffer ownership between the CPU and the device, ensuring that only one of them accesses the buffer at a time. Virtual pinning inherits this invariant and migrates a doubly-mapped page only while the *device* owns it; a page the CPU currently owns is refused at prepare. Only the owner's PTE therefore needs the atomic CAS at commit. The other alias is not merely left inactive: the prepare phase unmaps it and flushes its TLB entry. A non-present entry cannot be cached by anything—including a speculative page-table walk, which sets the access bit without any architectural access and would otherwise defeat a check made at commit time. The commit's update to that alias is then an infallible store into an empty slot, and an aborted migration restores it. Ownership transitions are observed on the DMA synchronization path; one that lands in an open migration window takes a slow path under a per-page lock, which closes the window— deterministically aborting the migration—and restores the alias for the incoming owner. The same lock serializes IOMMU mapping creation with migration. The alias is thus absent only while its owner is the other accessor, so no accessor ever encounters a missing translation—the guarantee of §3.2 still holds.

Beyond this common scenario, a page may be pinned multiple times—for example, if two I/O buffers happen to share a physical page. For single-page mappings, migration could be enabled by directing both pins through the same IOVA (I/O virtual address) and `vmap` alias with proper accounting. However, when the page is part of a multi-page scatter-gather list, each mapping requires it to be contiguous with different neighbors in IOVA space, so the mappings cannot share a single PTE. In our prototype we simply mark any page with multiple mappings of the same type as non-migratable (§3.6). This case is more likely with short-lived streaming DMA mappings than with the long-lived registrations (RDMA, VFIO) that virtual pinning primarily targets.

3.5 Linux Implementation

Subsystem Adaptation. Enabling virtual pinning for a kernel subsystem requires a one-time adaptation. Today, subsystems that pin pages hold a `struct page` pointer, from which they derive the kernel linear-mapping address for CPU access or the physical address for DMA. Both become stale after migration. To enable migration, the subsystem creates an *alias*—a `vmap` or IOMMU mapping that can be atomically redirected—and stores the alias's virtual address instead of the `struct page` pointer. All subsequent access goes through this address, which remains stable across migration. When multiple pages are pinned as a contiguous unit, a single alias covers the range. When the pin is released, the alias is torn down.

If a subsystem temporarily needs the physical address (e.g., to pass a `struct page` to an interface that does not accept virtual addresses), it acquires the page lock to exclude concurrent migration, elevates the reference count, and releases the lock; migration rechecks the reference count during the commit phase and aborts if an additional pin was taken.

Per-Page Metadata. Each physically pinned page needs reverse-mapping pointers and serialization state. We store this in a 48-byte sidecar per page using the kernel's `page_ext` framework, including the kernel `vmap` address, the IOMMU domain pointer, the IOMMU I/O virtual page frame number (with the current write-owner—kernel or DMA—encoded in a spare bit), a spinlock that serializes migration with DMA ownership transfers and IOMMU mapping creation, and a reference count that lets registered buffers sharing a physical page share its `vmap` alias. This avoids modifying the core `struct page` layout, which is size-critical, at the cost of an extra pointer dereference per access. Before migration, if the existing pin-count bias [20] indicates a pin but no alias exists in the sidecar, the page is skipped, so unadapted subsystems that still pin via `pin_user_pages` are handled safely.

IOMMU Driver Interface. We extend the IOMMU subsystem's domain operations structure with three migration callbacks—*prepare*, *commit*, and *abort*—that the migration core invokes at the corresponding points in the protocol, plus two supporting hooks: one records the reverse mapping when a page is mapped, the other reports buffer-ownership transitions from the DMA synchronization path (§3.4). This design keeps the protocol generic: our Intel VT-d implementation provides these callbacks for both first-stage and second-stage page-table formats, and any IOMMU driver that supports hardware access bits could implement the same interface.

Adapted Subsystems. We have adapted four subsystems following the model described in §3.5: RDMA memory regions and the DMA-IOMMU layer (which covers VFIO device assignment) for device DMA, and `io_uring` registered buffers and the pipe/splice (`vmsplice`) subsystem for direct kernel access.

[1] `dma_sync_single_for_cpu` and `dma_sync_single_for_device`. These exist for cache maintenance on non-coherent architectures (e.g., Arm SoCs); on x86, DMA is cache-coherent and the calls are no-ops, but drivers must issue them for portability.

3.6 Limitations

Our approach has three main limitations. First, each page may have at most one kernel and one IOMMU fault-intolerant mapping; pages with multiple mappings of the same type, or with conflicting DMA ownership states, are left in place (§3.4). In practice, device-assigned VMs—the dominant use case—are unaffected: all passthrough devices share a single IOMMU domain, so each guest page has exactly one IOMMU mapping. Sharing PTEs between the CPU and device (e.g., first-stage translation in VT-d) would eliminate this limitation entirely.

Second, redirecting DMA mappings during migration requires the IOMMU to be enabled for translation (not in passthrough mode), so that software controls the device-visible physical addresses. The main overhead is not translation itself—the IOTLB amortizes that—but creating and tearing down IOMMU mappings on every DMA operation. Linux's deferred IOTLB invalidation reduces this cost significantly. In practice, virtual pinning can be limited to long-lived mappings (e.g., RDMA registrations, VFIO), where the migration benefit is largest and the per-operation overhead is amortized over the mapping's lifetime.

Third, virtual pinning adds per-page memory overhead: the `page_ext` sidecar (§3.5) is 48 bytes per page. When allocated system-wide, this adds ∼1.2% memory overhead. The spinlock could be replaced with hash-based locks to reduce per-page overhead. Ongoing kernel work on variable-sized per-page metadata [10] would allow attaching the sidecar only to pages that are actually pinned, eliminating the system-wide cost. Each pinned region additionally requires a `vmap` allocation; for long-lived mappings such as VFIO and RDMA, regions are typically large and contiguous, so this overhead is small.

4 Evaluation

We implement virtual pinning in Linux 6.19, adding ∼2,700 SLoC across four domains: the page-alias tracking subsystem that maintains per-page reverse mappings, the Intel VT-d IOMMU driver with prepare/commit callbacks for both first- and second-stage translation, the core migration path with the two-phase protocol described in §3, and subsystem integrations for `io_uring` registered buffers, pipe/splice, and RDMA `ib_umem`.

All experiments run on a two-socket 4th Generation Intel Xeon Scalable (Gold 5420+) server (112 cores, 256 GB RAM across two NUMA nodes) with address-space randomization disabled and CPU frequency governor set to `performance` for reproducibility (§B.1). Baseline measurements use an unmodified Linux 6.19 kernel with identical boot parameters; virtual pinning measurements use our modified kernel. Each experiment is repeated five times; bar charts and line plots show means with standard-deviation error bars.

We validate correctness with stress tests that perform `io_uring`, vmsplice, and RDMA transfers under sustained concurrent migration, with zero data errors (§B.2). Virtual pinning imposes no steady-state overhead: before migration is triggered, the pinned and virtual pinning configurations are indistinguishable in all our benchmarks (identical latency and throughput).

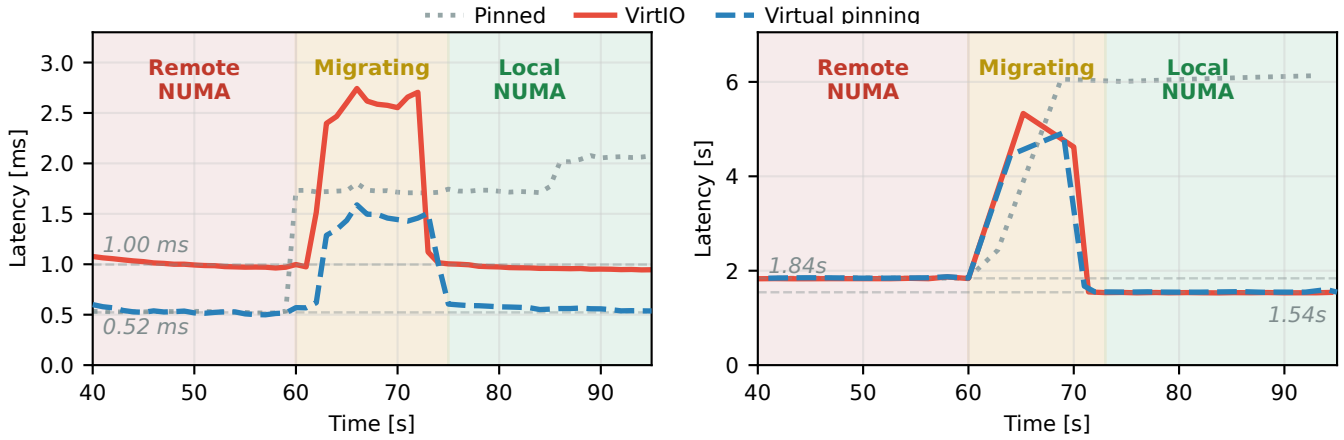
We first evaluate NUMA migration of pinned memory on three I/O subsystems—a device-assigned VM (§4.1), bare-metal RDMA (§4.2), and `io_uring` pre-registered buffers (§4.3)—then evaluate the second use case, compaction, by measuring how much huge-page capacity memory defragmentation recovers once pinned pages can move (§4.4), and finally per-page migration cost (§4.5), and CAS contention under active DMA (§4.6).

4.1 Device-Assigned VM

We evaluate whether virtual pinning can deliver both SR-IOV network performance and NUMA migration for device-assigned VMs—a combination impossible today, since SR-IOV pins all guest memory (§1). The paravirtual alternative, VirtIO, leaves guest memory unpinned and therefore migratable, but routes every packet through the host network stack and an extra copy. We use PostgreSQL 12, whose workload naturally spans both regimes: point lookups exercise the network path (each query is fast, network round-trip dominates), while hash joins exercise DRAM (each query probes a ∼500 MB hash table, DRAM latency dominates). The VM has 12 GB of RAM with a single vCPU on NUMA node 1 (where the NIC is attached), while its memory is initially allocated on the remote NUMA node 0. PostgreSQL is configured with 4 GB of `shared_buffers` and 512 MB `work_mem`. We populate two tables with 2 million rows each (∼500 MB per table) and run the standard `pgbench` benchmark from the host via the SR-IOV data path.

Network-Dominated: Point Lookups. We first measure `pgbench` in select-only mode (`-S`), which issues random primary-key lookups. Each query performs a single index scan and returns one row—a fast operation where network round-trip time dominates. We run 10 concurrent connections for 120 seconds; at t=60 s we invoke `migratepages` (a Linux command that triggers the kernel's page migration path for a process) to move the VM's memory to the CPU-local node.

Figure 3a shows the results. SR-IOV passthrough (pinned or virtual pinning) achieves 43% lower per-query latency than VirtIO due to hardware-mediated I/O. Memory migration has negligible effect on latency for all three configurations because each transaction's DRAM access time (∼1 μs for one index probe) is small relative to the network round-trip and query processing overhead.



(a) Point lookups: SR-IOV achieves 43% lower latency than VirtIO.

(b) Hash joins: both drop 16% after migration to local DRAM.

Figure 3. PostgreSQL latency during NUMA migration.

Memory-Dominated: Hash Joins. We next run a hash join[2] that builds a ~500 MB hash table from `t1` and probes it with all 2 million rows of `t2`. Each probe performs a random lookup into a hash table that far exceeds the caches, making the query DRAM-latency-bound. We execute 60 consecutive joins (each taking ~2 s) and trigger `migratepages` at t=60 s.

Figure 3 shows both workloads during NUMA migration. Point lookups (Figure 3a) are unaffected by migration: SR-IOV with virtual pinning (dashed) maintains 43% lower latency than VirtIO (solid) throughout. Hash joins (Figure 3b) show the opposite: both VirtIO and virtual pinning drop from 1.8 s to 1.6 s (16% improvement) after migration completes.

With standard SR-IOV pinning (dotted), latency *increases* sharply after `migratepages` is invoked and continues to degrade throughout. Although the DMA-pinned pages cannot actually move, the kernel's migration path unmaps each page's PTE before checking whether the page is pinned—sending a TLB-shootdown IPI to the VM's CPU for every page—only to discover the elevated reference count and remap the original page. Each such page fails with `-EAGAIN` and is retried up to ten times within the same call, one page at a time with a synchronous shootdown each—whereas a successful migration pays a single pass with batched flushes. With ~3 M guest pages, this unmap-fail-remap cycle takes over 70 s, during which the vCPU is continuously interrupted by TLB-shootdown IPIs. While this overhead could be mitigated by checking the pin state before unmapping, the fundamental problem remains: DMA-pinned pages cannot be migrated.

Figure 1 (page 2) summarizes both workloads side by side. Point-lookup latency (Figure 1a) is 43% lower with SR-IOV than VirtIO, while NUMA placement has no effect (remote and local bars are equal). Hash join latency (Figure 1b) drops 16% after migration for both VirtIO and virtual pinning, while pinned SR-IOV remains flat. No single configuration wins in both charts: standard pinning sacrifices migration; VirtIO sacrifices network latency. Only virtual pinning delivers the best of both.

[2] `SELECT count(*) FROM t1 JOIN t2 ON t1.id = t2.ref_id`

4.2 Bare-Metal RDMA

Bare-metal RDMA applications—key-value stores [15, 25], distributed file systems, and machine-learning parameter servers—pin large memory regions for direct NIC access. Applications typically allocate and register these buffers early, before NUMA-aware placement is established; subsequent thread migration or memory pressure can leave registered regions stranded on a remote node. NUMA placement of these buffers directly affects both DMA and CPU access latency [26]. We use `ib_write_lat` from perftest [37], patched to add a pointer-chase option: between RDMA writes, each side performs 64 dependent loads through a 256 MB registered buffer (64K pages) organized as a random permutation of cache-line pointers, defeating hardware prefetchers and making the CPU path NUMA-sensitive. The pointer-chase pattern is common in graph traversals and index lookups. The chase buffer is registered with the NIC—and therefore pinned—like the application index structures it models, and the buffer the NIC writes into sits on the same remote node, so migration moves memory the device is actively using.

We run a loopback test on a ConnectX-6 Dx VF in full IOMMU translation mode; both sides run on NUMA node 1 (where the NIC resides), but each side's chase buffer and RDMA data buffer are allocated on the remote node 0 via `mbind`. At t=60 s we invoke `migratepages` to move them to the NIC-local node. For the On-Demand Paging (ODP) configuration, we register the same buffer as on-demand

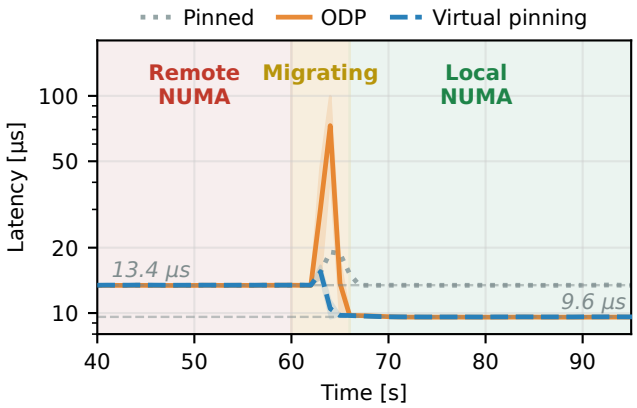


Figure 4. RDMA write latency with pointer chasing on both sides, measured at the client. Shaded bands show ±1 stddev. Note the logarithmic y-axis.

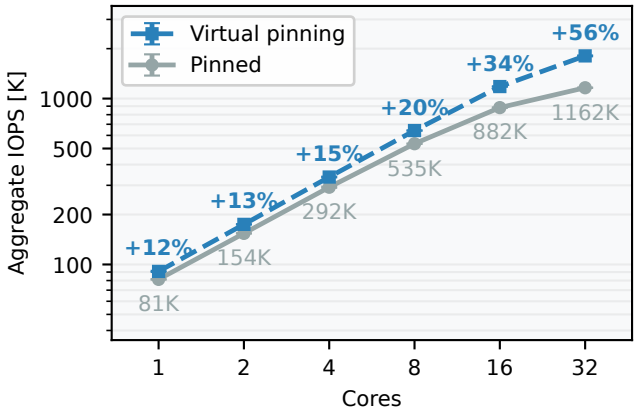


Figure 5. `io_uring` random-read aggregate IOPS after cgroup NUMA migration.

paged[3] on the unmodified kernel; the mlx5 driver handles page-fault resolution transparently.

We compare three configurations in Figure 4. With standard pinning, migration is refused: latency returns to 13.4 μs after a one-second 19 μs bump—the same unmap-fail-remap cost as §4.1, over 64K pages rather than 3 M. With virtual pinning, all 64K pages migrate successfully—average latency drops from 13.4 μs to 9.6 μs (29% improvement) as the dependent loads now hit local DRAM. Before migration, all three configurations are indistinguishable—13.4 μs, confirming no steady-state overhead.

ODP achieves the same post-migration latency (9.6 μs) as virtual pinning, but with a larger and less predictable transient spike (mean per-run peak 81 μs, with high run-to-run variance): each migrated page must be invalidated in the NIC's page tables through the MMU-notifier path, a per-range driver round-trip that prolongs the migration window. Virtual pinning instead updates IOMMU PTEs directly, and because the NIC never touches the chase pages, all but two of those migrated, the access-bit optimization elides their IOTLB flushes—migration completes in a single fast pass (15 μs spike), producing a smaller and consistent transient—making it preferable even on ODP-capable hardware when tail latency matters. Moreover, ODP is a vendor-specific stand-in for ATS/PRI—it serves the same purpose by a proprietary route—and it still requires the device to stall on a fault until the host resolves it, which most deployed hardware cannot do. Virtual pinning needs no device-side fault support at all, only an IOMMU that maintains access bits.

[3] `IBV_ACCESS_ON_DEMAND` in the `ibv_reg_mr` call instructs the NIC driver to use MMU-notifier callbacks instead of pinning.

4.3 `io_uring` Fixed Buffer Migration

We now test virtual pinning on kernel-internal mappings rather than device DMA. `io_uring` allows applications to pre-register I/O buffers[4] so the kernel can reuse their mappings across requests; these buffers are pinned and cannot be NUMA-migrated on a stock kernel.

We run `fio` 3.36 with the `io_uring` engine and registered buffers inside a Linux cgroup, which allows the operator to control which NUMA node backs a container's memory and to trigger migration of all its pages at once. Each of 1–32 workers registers 4,096 buffers of 32 KB (128 MB per worker, exceeding the 52 MB per-socket LLC) and reads randomly from its own tmpfs file. Initially, the cgroup confines all memory to the remote NUMA node 0 while workers run on node 1 CPUs. After a 15-second warm-up we reassign the cgroup to node 1 with automatic migration enabled, triggering the kernel to move all process pages to the local node.

Figure 5 shows aggregate IOPS after migration on a log scale. On the stock kernel, the cgroup flip moves code, stack, and heap to node 1 but the pinned registered buffers stay on node 0; post-migration throughput is essentially unchanged. With virtual pinning, the buffers migrate alongside everything else via the vmap CAS protocol, and the throughput gap grows with core count: 12% at a single core, rising to 56% at 32 cores (1808 K vs. 1162 K IOPS) as cross-node interconnect traffic increases.

4.4 Memory Defragmentation

We measure whether virtual pinning can recover huge page availability destroyed by pinning—the dominant source of unmovable pages in datacenters [56]. We allocate and pin memory on NUMA node 0 using `ibv_reg_mr` (the RDMA buffer registration call), scattering the pinned pages by first

[4] `io_uring_register_buffers`

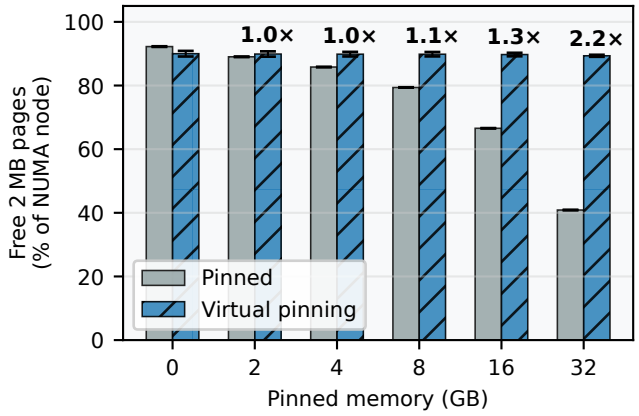


Figure 6. Free 2 MB pages on a NUMA node as pinned memory increases.

filling memory with a fragmentation workload that holds alternating 4 KB pages, then allocating pinned pages from the resulting gaps. After releasing the fragmentation workload, pinned pages are interleaved with free pages throughout the node. We then invoke `migratepages` to move the pinned pages to node 1 and measure how many 2 MB-aligned free regions are recovered.

Figure 6 shows the results. On the stock kernel, migration is refused for all pinned pages, so contiguity degrades as pin size increases: from 90% of the node's 2 MB blocks free with no pinning down to 41% at 32 GB. Virtual pinning migrates all 8M pages successfully, recovering the node to 89% free at 32 GB.

4.5 Migration Overhead

We measure the per-page migration cost under two conditions that exercise the access-bit optimization described in §3.2: *cold* pages, where no DMA has occurred since registration (IOMMU access bit clear), and *hot* pages, where every page has been DMA-accessed (access bit set). We migrate 512 MB between NUMA nodes using `migratepages`.

Unpinned migration costs 3.5 μs per page, dominated by the 4 KB page copy. For cold virtually pinned pages, the access-bit optimization skips all IOTLB flushes—only the atomic IOMMU PTE update and reverse-map bookkeeping are needed—bringing the cost to 3.7 μs (1.04× unpinned). For hot pages, each page additionally requires an IOTLB invalidation and drain, increasing the cost to 4.5 μs (1.27× unpinned). The 0.8 μs difference is the per-page IOTLB flush cost eliminated by the optimization. In VM passthrough, most guest pages are cold at any given time, so the cold path dominates. Cold migration of 512 MB (131K pages) completes in 0.5 s wall-clock time.

The CPU-side counterpart is more expensive. Repeating the experiment on pages reached through a kernel `vmap` alias, cold pages again cost what unpinned migration costs, but hot pages rise to 4.11× unpinned. The gap between this and the IOTLB case is structural: `vmap` mappings are global, so invalidating one requires a shootdown to every CPU, whereas the IOTLB invalidation is confined to the IOMMU. The access-bit optimization therefore matters most exactly where the flush is most expensive, and it removes CPU shootdowns from the migration path entirely for DMA-only workloads such as RDMA, which have no `vmap` alias at all. The setup and per-page costs are in §B.3.

4.6 CAS Contention Under Active DMA

We measure how often the commit-phase CAS fails under varying DMA pressure. If a device re-accesses a page between the prepare-phase flush and the commit, the access bit is reset and migration aborts safely for that page.

We run RDMA atomic fetch-and-add operations in a tight loop while concurrently migrating the target buffer between NUMA nodes. We vary the buffer size from 1 to 128 pages. The NIC issues atomic operations at a fixed rate, so larger buffers dilute the per-page access frequency. Figure 7 shows the result. With a single-page buffer, the device re-accesses the page faster than migration can commit, and the CAS fails 99.7% of the time. As the working set grows, each page is accessed less frequently, widening the per-page CAS window: at 128 pages the failure rate drops to 1.3%. In all cases, data integrity is preserved: every one verified that each counter matches its expected value after the test.

The stress test above intentionally targets the same pages being migrated. In realistic workloads, DMA and migration typically target different pages: in the pointer-chase benchmark of §4.2, all 64K pages migrate with zero CAS failures.

A failed page is retried under Linux's existing policy rather than a new one: the asynchronous and compaction paths retry immediately, while the synchronous path sets aside pages that still fail and revisits them on a later pass. Because an abort leaves the page exactly where it was, a page that never wins the race simply stays put—the outcome the migration would have had without virtual pinning at all.

5 Discussion

We evaluated NUMA rebalancing and defragmentation across three I/O subsystems—VFIO, RDMA, and `io_uring`—but the migration mechanism is subsystem-agnostic: each subsystem required only a thin adaptation layer, while the same CAS-based protocol redirects any fault-intolerant PTE. Other use cases currently blocked by pinning could benefit from the same protocol. *Memory tiering* across heterogeneous memory (DRAM, CXL, persistent memory) requires migrating pages between tiers based on access frequency; pinned pages are excluded from all existing tiering systems, and a new detection policy is needed (§5). *Cross-host VM live migration* via CXL shared memory requires moving guest pages to a shared pool; pinned pages make this impossible for

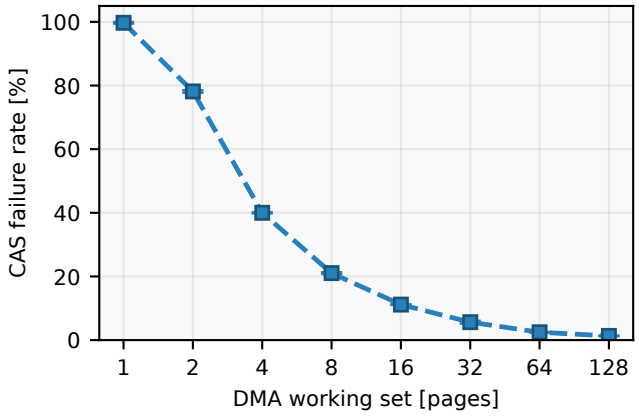


Figure 7. CAS failure rate vs. DMA working set size.

passthrough VMs today. *Memory overcommitment* of device-assigned VMs—paging cold guest memory to a slower tier to reclaim DRAM for other tenants [57]—is blocked entirely because pinned pages cannot be reclaimed or relocated. In each case, virtual pinning provides the missing mechanism by making pinned pages visible to the kernel's migration infrastructure.

Integration with Migration Policies. Our experiments trigger migration explicitly via `migratepages`, as a cloud orchestrator or administrator would. On single-machine deployments, however, users expect the kernel to rebalance NUMA placement transparently. Virtual pinning provides the *mechanism* to migrate pinned pages, but a policy must still decide *which* pages to move and *when*. Linux's default NUMA balancer (AutoNUMA) detects remote accesses by temporarily unmapping pages and promoting them on fault—incompatible with virtually pinned pages, whose accessors cannot take it. Fault-free alternatives avoid this problem: Liu et al. propose an Amortized Offcore Latency metric that accounts for memory-level parallelism [34]; Lee et al. scan IOMMU access bits to detect hot device-mapped pages [30]; and the kernel community is consolidating PTE access-bit scanning, hardware sampling (AMD IBS, Intel PEBS), and CXL hotness monitoring into a unified `pghot` framework [42]. These detect misplaced pages without unmapping them, making them compatible with virtual pinning and extending them to device-assigned VMs and RDMA-registered memory for the first time.

Enabling Shared Virtual Addressing Without ATS. In Shared Virtual Addressing (SVA), the IOMMU walks the *process's own page tables* rather than a separate I/O page-table hierarchy, so the CPU and device share a single PTE per mapping. This lets kernel-bypass frameworks such as DPDK and SPDK pass virtual addresses directly to the device, removing the need to pin pages or register memory regions before each DMA. SVA requires the device to tag DMA requests with a Process Address Space ID (PASID) so the IOMMU selects the correct page table, and requires the IOMMU to share the CPU page-table format.[5] Linux additionally gates SVA on ATS/PRI support (§2), because without fault handling the device cannot recover when a page is swapped out or migrated. Virtual pinning removes the second reason: pages stay present and mapped throughout migration, so SVA becomes usable on devices that support PASID but lack ATS. The device still cannot take a fault at all, however, so the memory it touches must be prepopulated and `mlock`ed, as VFIO already requires—though, unlike today, it need not be immovable.

An SVA-based virtual pinning implementation would also be substantially simpler. Because the CPU and device share a single PTE (except where the buffer also carries a separately pinned mapping), a single CAS atomically redirects both CPU and DMA translations, eliminating DMA ownership tracking (§3.4), per-page `vmap` allocations, and subsystem-specific integrations (io_uring, splice, RDMA). The present-to-present discipline still applies, and for the same reason as before: with one shared `PTE`, making it briefly non-present is harmless for the CPU—a userspace fault is always recoverable—but fatal for a device that cannot fault, which is exactly the device this enables. What our protocol contributes is that the shared entry is redirected atomically, so no fault is needed on either side.

6 Related Work

I/O Page-Faults. §2.3 introduces the main I/O page-fault mechanisms (ATS/PRI, ODP); here we discuss specific systems that build on them. VPRI [18] deploys IOPF on existing DPUs without requiring ATS/PRI on the device, but does not enable page *migration*—pages under active DMA remain fixed, and its adaptive policy re-pins memory under high I/O load. Lesokhin et al. [31] demonstrated I/O page-fault handling for InfiniBand and Ethernet, but require NIC firmware modifications not available in commodity hardware. Fukuoka et al. [16] confirmed that ODP's retransmission-amplified stalls render it impractical for latency-sensitive workloads. PART [41] makes IOPF practical for HPC RDMA by tightly coupling the NIC with the CPU's IOMMU, but requires an on-die NIC not available in PCIe-attached devices. All these systems tolerate brief mapping invalidity, incurring per-fault latency and queue stalls—none enables transparent page *migration*. Virtual pinning never exposes an accessor to a non-present mapping, enabling migration without faults.

[5] Intel's first-stage IOMMU translation already uses the x86-64 page-table format, satisfying this requirement on current hardware.

Migration of Unmovable Pages. Contiguitas [56] confines unmovable allocations to a dedicated region to reduce fragmentation, but cannot migrate them; full migration requires last-level cache extensions not available in current systems. Mosaic pages [17], Redundant Memory Mappings [27], and Perforated Pages [39] sidestep the contiguity requirement by packing non-contiguous, range-based, or hole-tolerant translations into fewer TLB entries, but require hardware changes not available in current processors. The dominant source of unmovable allocations is networking buffers (73%), which virtual pinning migrates on existing hardware.

Device Assignment and Memory Reclaim. Polyakov et al. [40] enable cross-host live migration of VMs with passthrough RDMA NICs via firmware-level save/restore on NVIDIA ConnectX-7, with sub-second downtime. Virtual pinning addresses the orthogonal problem of relocating pinned pages *within* a host; the two are complementary.

NP-RDMA [44] is a software protocol that allows RDMA applications to leave pages unpinned and swap them out under memory pressure. When a page is reclaimed, the IOMMU redirects its mapping to a decoy "signature page" filled with magic values; the initiator detects faults by checking returned data after every read and issuing an auxiliary read-back after every write. Both communicating peers must run modified software, and even without faults, throughput halves for writes that signal completion. Virtual pinning is transparent, applies to any DMA subsystem, and imposes no per-operation overhead.

VIO [51] intercepts VirtIO submission rings to fault in pages before the device sees the request, recovering cold memory without PRI hardware. However, it requires VirtIO and a DPU, cannot migrate pages (only reclaim), and re-pins all memory under high IOPS.

Emulating a virtual IOMMU (vIOMMU) in the hypervisor can track which guest pages are under active DMA, identifying pages safe to migrate [5]. However, the emulation requires a VM exit on every guest DMA map/unmap operation [9, 49], and persistently mapped buffers—such as the entire guest memory in device-assigned VMs—remain pinned regardless of tracking. Virtual pinning uses the *host* IOMMU directly and requires no guest-visible emulation.

MMU Notifiers. Linux's MMU notifier framework coordinates page-table consumers before migration, but assumes consumers can tolerate brief mapping invalidity. Nvidia's HMM [33] uses this to mirror CPU page tables on GPUs, which can pause I/O during faults. Devices such as NICs cannot suspend incoming traffic. Virtual pinning atomically redirects IOMMU mappings so the device never encounters an absent translation.

Reducing Migration Cost. Nomad [53] keeps the source PTE present during the page copy so that the application is not blocked waiting for migration to complete, using the dirty bit to detect concurrent writes and abort if needed. As §1 discusses, this makes migration non-blocking for CPU accesses but still leaves the mapping briefly absent when the old PTE is cleared and the new one installed, which a device cannot tolerate. Copier [19] optimizes the copy itself by overlapping `memcpy` with SIMD and DMA hardware. LATR [28] reduces migration cost from a different angle by deferring TLB shootdowns to scheduler ticks, avoiding synchronous cross-core IPIs. All three target unpinned pages; virtual pinning solves a different problem—making migration *possible* for pinned pages—and its CAS-based commit redirects the live PTE in place (present→present).

7 Conclusion

Pinned pages have been treated as immovable since the introduction of DMA mapping APIs, forcing the OS to choose between I/O performance and memory management flexibility. Virtual pinning shows that this trade-off is unnecessary: the hardware ordering guarantees already present in modern MMUs and IOMMUs are sufficient to migrate pinned pages safely, without device modifications or I/O page-fault support.

Virtual pinning makes pinned pages first-class citizens of the kernel's migration infrastructure. Device-assigned VMs gain both SR-IOV network performance and NUMA-aware placement; bare-metal RDMA gains migration benefits without requiring device-side page-fault hardware; and huge page availability recovers under pinning pressure that previously destroyed it. Beyond these, virtual pinning lays the groundwork for memory tiering and live migration of passthrough VMs, which remain directions for future work. On architectures whose specifications require break-before-make (BBM) for output-address changes, in-place PTE redirection is not possible today. Relaxing this constraint, potentially a specification clarification rather than a hardware change, would extend transparent pinned-page migration to those platforms.

Availability

Our implementation is available as a patch series against Linux 6.19, archived with the scripts that reproduce our experiments at:
https://doi.org/10.5281/zenodo.21909764.

Acknowledgments

We thank our anonymous shepherd and reviewers for their careful reading and for suggestions that improved the paper. This research was supported by the Israel Science Foundation (grant No. 3310/25). Claude (Anthropic) assisted with writing and code; the authors reviewed, verified, and stand behind all text, code, and results.

References

[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 283–300. doi:10.1145/3373376.3378468

[2] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 631–644. doi:10.1145/3037697.3037706

[3] Jade Alglave, Richard Grisenthwaite, Graeme Sherwood-Smith, Alasdair Armstrong, Liam Sherwood, Samuel Sherwood, and Thomas Sherwood. 2024. Puss In Boots: on formalising Arm's Virtual Memory System Architecture. In *Formal Methods in Computer-Aided Design (FMCAD)*. doi:10.1109/FMCAD.2024.10918450

[4] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *USENIX Annual Technical Conference (ATC)*.

[5] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*. 73–86.

[6] Arm Limited. 2020. *Arm System Memory Management Unit Architecture Specification, SMMU architecture version 3.* Document No. ARM IHI 0070 D.a.

[7] Arm Limited. 2024. *Arm Architecture Reference Manual for A-profile architecture.* Document No. DDI 0487, §D8.17.1: Using break-before-make when updating translation table entries.

[8] Christian Bell and Dan Bonachea. 2003. A new DMA registration strategy for pinning-based high performance networks. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. doi:10.1109/IPDPS.2003.1213363

[9] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. 2007. The Price of Safety: Evaluating IOMMU Performance. In *Ottawa Linux Symposium (OLS)*. 9–20.

[10] Jonathan Corbet. 2024. Fleshing out memory descriptors. LWN.net. https://lwn.net/Articles/974937/.

[11] 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 SIGARCH Computer Architecture News (CAN)* 41, 1 (mar 2013), 381–394. doi:10.1145/2490301.2451157

[12] Advanced Micro Devices. 2024. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming.* Publication No. 24593, Rev 3.44.

[13] Advanced Micro Devices. 2025. *AMD I/O Virtualization Technology (IOMMU) Specification.* Publication No. 48882, Rev 3.10.

[14] Thaleia Dimitra Doudali, Daniel Zahka, and Ada Gavrilovska. 2020. The Case for Optimizing the Frequency of Periodic Data Movements over Hybrid Memory Systems. In *ACM International Symposium on Memory Systems*. 137–143. doi:10.1145/3422575.3422788

[15] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 401–414.

[16] Takuya Fukuoka, Shigeyuki Sato, and Kenjiro Taura. 2021. Pitfalls of InfiniBand with On-Demand Paging. In *IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*. 265–275. doi:10.1109/ISPASS51385.2021.00049

[17] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A Bender, Abhishek Bhattacharjee, et al. 2023. Mosaic pages: Big TLB reach with small pages. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 433–448. doi:10.1145/3582016.3582021

[18] Kaijie Guo, Dingji Li, Ben Luo, Yibin Shen, Kaihuan Peng, Ning Luo, Shengdong Dai, Chen Liang, Jianming Song, Hang Yang, Xiantao Zhang, and Zeyu Mi. 2024. VPRI: Efficient I/O Page Fault Handling via Software-Hardware Co-Design for IaaS Clouds. In *ACM Symposium on Operating Systems Principles (SOSP)*. doi:10.1145/3694715.3695957

[19] Jingkai He, Yunpeng Dong, Dong Du, Mo Zou, Zhitai Yu, Yuxin Ren, Ning Jia, Yubin Xia, and Haibo Chen. 2025. How to Copy Memory? Coordinated Asynchronous Copy as a First-Class OS Service. In *ACM Symposium on Operating Systems Principles (SOSP)*. doi:10.1145/3731569.3764800

[20] David Hildenbrand, Martin Schulz, and Nadav Amit. 2023. Copy-on-Pin: The Missing Piece for Correct Copy-on-Write. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 176–191. doi:10.1145/3575693.3575716

[21] Intel Corporation. 2023. *Intel Virtualization Technology for Directed I/O Architecture Specification.* Revision 4.1, Order Number: D51397-016.

[22] Intel Corporation. 2024. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Order Number: 325462-084US.

[23] RISC-V International. 2024. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.* https://github.com/riscv/riscv-isa-manual/releases/tag/20240411 Version 20240411 (ratified).

[24] RISC-V International. 2025. *RISC-V IOMMU Architecture Specification.* https://github.com/riscv-non-isa/riscv-iommu/releases/tag/v1.0.1 Version 1.0.1 (ratified).

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *ACM SIGCOMM Conference*. 295–306. doi:10.1145/2619239.2626299

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *USENIX Annual Technical Conference (ATC)*. 437–450.

[27] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 66–78. doi:10.1145/2749469.2749471

[28] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselỳ, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy translation coherence. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 651–664. doi:10.1145/3173162.3173198

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

[30] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *ACM Symposium on Operating Systems Principles (SOSP)*. 17–34. doi:10.1145/3600006.3613167

[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 *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 449–466. doi: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 *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 574–587. doi:10.1145/3575693.3578835

[33] Linux Kernel Documentation. 2017. Heterogeneous Memory Management (HMM). Linux Kernel Documentation, https://www.kernel.org/doc/html/latest/mm/hmm.html. Accessed: 2026-04-10.

[34] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered Memory Management Beyond Hotness. In *USENIX Symposium*

on Operating Systems Design & Implementation (OSDI).

[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 *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 742–755. doi:10.1145/3582016.3582063

[36] Deok-Jae Oh, Yaebin Moon, Do Kyu Ham, Tae Jun Ham, Yongjun Park, Jae W Lee, Jung Ho Ahn, and Eojin Lee. 2022. MaPHeA: A Framework for Lightweight Memory Hierarchy-aware Profile-guided Heap Allocation. *ACM Transactions on Embedded Computing Systems* 22, 1 (2022), 1–28. doi:10.1145/3527853

[37] Open Fabrics Enterprise Distribution (OFED). 2024. linux-rdma/perftest: InfiniBand Verbs Performance Tests. https://github.com/linux-rdma/perftest. Accessed: 2026-04-13.

[38] Ashish Panwar, Aravinda Prasad, and K Gopinath. 2018. Making huge pages actually useful. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 679–692. doi:10.1145/3173162.3173203

[39] 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 *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 913–925. doi:10.1109/ISCA45697.2020.00079

[40] Artem Y. Polyakov, Gal Shalom, Asaf Schwartz, Aviad Yehezkel, Omri Ben David, Omri Kahalon, Ariel Shahar, and Liran Liss. 2025. Device-Assisted Live Migration of RDMA Devices. In *ACM Symposium on Operating Systems Principles (SOSP)*. doi:10.1145/3731569.3764795

[41] Antonis Psistakis, Nikos Chrysos, Fabien Chaix, Marios Asiminakis, Michalis Gianioudis, Pantelis Xirouchakis, Vassilis Papaefstathiou, and Manolis Katevenis. 2022. Optimized Page Fault Handling During RDMA. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 33, 12 (2022), 3990–4003. doi:10.1109/TPDS.2022.3175666

[42] Bharata B Rao. 2025. Unifying Sources of Page Hotness Information. In *Linux Plumbers Conference (LPC)*. https://lpc.events/event/19/contributions/2063/ Device and Specific Purpose Memory MC.

[43] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. DRAM errors in the wild: a large-scale field study. *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* 37, 1 (2009), 193–204. doi:10.1145/1555349.1555372

[44] Huijun Shen, Guo Chen, Bojie Li, Xingtong Lin, Xingyu Zhang, Xizheng Wang, Amit Geron, Shamir Rabinovitch, Haifeng Lin, Han Ruan, Lijun Li, Jingbin Zhou, and Kun Tan. 2023. NP-RDMA: Using Commodity RDMA without Pinning Memory. *CoRR* abs/2310.11062 (2023). doi:10.48550/arXiv.2310.11062

[45] Dragan Stancevic. 2023. nil-migration: Nearly Instantaneous Live Migration of Virtual Machines over CXL. Linux Storage, Filesystem, MM & BPF Summit (LSF/MM/BPF). https://nil-migration.org/nil-migration-BoF-LFSMMBPF-2023.pdf.

[46] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2025. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. doi:10.1145/3676641.3711999

[47] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. 1998. Pin-down cache: A virtual memory management technique for zero-copy communication. In *IEEE International Parallel Processing Symposium (IPPS)*. 308–314. doi:10.1109/IPPS.1998.669932

[48] The Linux Kernel community. 2024. Dynamic DMA mapping Guide. Linux Kernel Documentation, https://www.kernel.org/doc/html/latest/core-api/dma-api-howto.html. Accessed: 2026-04-10.

[49] 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 *USENIX Annual Technical Conference (ATC)*.

[50] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *ACM Symposium on Operating Systems Principles (SOSP)*. doi:10.1145/3694715.3695968

[51] Yun Wang, Liang Chen, Jie Ji, Xianting Tian, Ben Luo, Zhixiang Wei, Zhibai Huang, Kailiang Xu, Kaihuan Peng, Kaijie Guo, Ning Luo, Guangjian Wang, Shengdong Dai, Yibin Shen, Jiesheng Wu, and Zhengwei Qi. 2025. To PRI or Not To PRI, That's the Question. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*.

[52] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 609–621. doi:10.1145/3503222.3507731

[53] 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 *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*. 527–544.

[54] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble page management for tiered memory systems. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 331–345. doi:10.1145/3297858.3304024

[55] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. 2010. On the DMA mapping problem in direct device assignment. In *ACM International Systems and Storage Conference (SYSTOR)*. ACM, 18:1–18:12. doi:10.1145/1815695.1815718

[56] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, et al. 2023. Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 1–15. doi:10.1145/3579371.3589079

[57] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*.

[58] Weixi Zhu, Alan L Cox, and Scott Rixner. 2020. A comprehensive analysis of superpage management mechanisms and policies. In *USENIX Annual Technical Conference (ATC)*. 829–842.

A Specification Basis for Access-Bit Ordering

Virtual pinning relies on the guarantee that each platform's page-table walker sets the access bit before the corresponding data access is visible (§3.3). This appendix gives the precise specification language backing that claim.

Intel (x86). The Intel 64 and IA-32 SDM [22] states the ordering guarantee directly in Section 4.10.2.3 ("Details of TLB Use"):

> "the processor does not cache a translation for a page number unless the accessed flag is 1 in each of the paging-structure entries used during translation; before caching a translation, the processor sets any of these accessed flags that is not already 1."

This establishes that A-bit updates precede TLB caching, and therefore precede any data access that uses the cached translation. The SDM further specifies that these updates use *locked cycles* (Section 9.1.2.1). Since our CAS is also a locked operation on the same cache line, the two are serialized by cache coherence: if CAS observes A=0, the walker's locked update has not yet occurred.

For the IOMMU, the VT-d specification [21] (§4.2.3.1) states:

> "accessed … and dirty flags are set at the time of processing a translation-request before a translation-completion is returned to the endpoint."

Draining is likewise architectural: a hardware implementation with Major Version 2 or higher (`VER_REG`) "always performs required drain without software explicitly requesting a drain in IOTLB invalidation" (Section 11.4.2, which also marks the capability bits that once gated draining as deprecated), with draining "completed before the next Invalidation Wait Descriptor (`inv_wait_dsc`) is completed by hardware" (Section 6.5.4). For devices using ATS, Section 6.5.4 further requires that "the ATS invalidation completion message is issued on the posted channel and pushes all writes from the device (including any translated writes) ahead of it," and that the remapping hardware "process ATS invalidation completion messages after all preceding writes are considered drained."

AMD. The AMD64 Architecture Programmer's Manual [12] (Section 5.4.2) guarantees:

> "a table entry is never cached in the TLB without its Accessed bit being set at the same time."

This establishes that TLB caching and the A-bit update are atomic: if A=0 in the PTE, no TLB entry for that page exists on any core. Since the walker's A-bit update uses a locked cycle (as on Intel), and our CAS is also a locked operation on the same cache line, cache coherence serializes the two: our locked CAS must see the result of any prior locked A-bit write. If CAS observes A=0, the walker's locked update has not yet occurred, so no TLB entry was cached and no data access was issued.

The specification also notes that "the Accessed bit can be set for instructions that are speculatively executed by the processor," including mispredicted branch paths. This is a harmless false positive for our protocol (see §3.3).

For the IOMMU, the AMD IOMMU specification [13] structures its page walk as a sequence of numbered steps (Section 2.2.7.4). A/D updates occur at step 4 ("Set A and D bits in the guest page tables"), before the translated access itself at step 5 ("Access the GPA"). For draining, Section 2.4.1 guarantees that after a `COMPLETION_WAIT` following an IOTLB invalidation:

> "there are no transactions in flight anywhere in the system fabric that read or write the invalidated pages."

RISC-V. The privileged specification [23] states:

> "the hart [hardware thread] must not perform the memory access that caused the PTE update before the PTE update is globally visible."

Hardware access-bit updates (the Svadu extension) use a compare-and-swap on the PTE; A-bit updates may be speculative. The RISC-V IOMMU specification [24] (Section 3.4) provides the same guarantee for DMA:

> "the PTE update must be globally visible before a memory access using the translated address … becomes globally visible."

Arm. Arm's CPU MMU does not provide access-bit ordering. Without FEAT_HAFDBS, an access to a page with AF=0 generates a synchronous fault rather than a silent hardware update; with FEAT_HAFDBS, the architecture permits *spontaneous* access-flag updates without guaranteeing their visibility before the data access [3].

The SMMU v3 [6] supports hardware access-bit updates via HTTU, but for non-ATS translations both access flag and dirty state have relaxed visibility (Section 3.13.4). The ATS path provides the required SMMU-side ordering. Section 3.13.7.1 states:

> "When an ATS TR is made, it must be assumed that a device will subsequently access the page. If the page is otherwise valid and an ATS response will be returned, AF is set to 1 in the descriptor in the same way as a direct transaction access through the SMMU."

This guarantees that the access flag is set before the translation response reaches the device, and therefore before any DMA using that translation can occur—the ordering our CAS protocol requires. For draining, `CMD_SYNC` after `CMD_TLBI` guarantees (Section 4.7.3):

> "all transactions in progress that were translated using any of the TLB entries targeted by the invalidates are globally observable."

Arm BBM limitation. Although ATS provides the access-bit ordering needed by our CAS protocol, the SMMU's break-before-make (BBM) requirement prevents the present-to-present output-address update our protocol relies on. Section 3.21.1.1 of the SMMU specification [6] lists "changes to output address" among the modifications that require a BBM procedure, and the relaxed BBM levels (SMMU_IDR3.BBML) introduced in SMMUv3.2 apply only to page-size changes. Section 3.21.1.3 notes:

> "Arm expects that the only change that is made to a valid translation table descriptor is that of

> changing the page or block size of a range of addresses. A change to memory type, Cacheability, nG or output address might impair coherency or ordering semantics of accesses using the translation."

Consequently, migrating a page on Arm requires briefly invalidating the SMMU PTE before installing the new output address, creating a window during which an ATS translation request would be denied.

B Evaluation Details

B.1 Hardware and Kernel Configuration

All experiments run on a two-socket 4th Generation Intel Xeon Scalable (Gold 5420+) (112 cores, 256 GB RAM, 128 GB per NUMA node). The Intel IOMMU is configured in scalable mode with 4-level page tables (`intel_iommu=on,sm_on,sp_off`), which is required for hardware access-bit support (SSADE). Bare-metal RDMA experiments disable IOMMU passthrough (`iommu.passthrough=off`) so the IOMMU translates DMA for the RDMA VF. Kernel address-space layout randomization is disabled via boot parameter (`nokaslr norandmaps`), and userspace ASLR is disabled at runtime (`randomize_va_space=0`). The CPU frequency governor is set to `performance` on all cores.

Three further boot parameters suppress sources of run-to-run variance that would otherwise obscure the effects we measure. Processor idle states are disabled (`idle=poll`): an idle core would otherwise drop into a low-power C-state, and waking from one costs on the order of a microsecond—the same scale as the RDMA latencies we report. Automatic NUMA balancing, in which the kernel samples memory accesses and migrates pages toward the socket that is using them, is switched off (`numa_balancing=disable`); left enabled it would migrate pages on its own initiative, which is precisely the operation under measurement. Finally, 16 GB of 2 MB huge pages are reserved at boot (`hugepagesz=2M hugepages=8192`), enough to back the device-assigned VM's 12 GB of memory, which QEMU preallocates. Transparent huge pages are set to `never` by the PostgreSQL and `io_uring` runners, so that the pages under test are the ones each experiment allocates explicitly. Turbo Boost and simultaneous multithreading are left at their platform defaults, both enabled.

Both the stock Linux 6.19 baseline and our virtual pinning kernel use identical boot parameters, and the experiment driver re-applies them on every `kexec`, so a kernel switch never silently changes the configuration under which a measurement is taken.

B.2 Correctness Validation

We validate the correctness of virtual pinning under concurrent migration stress tests.

io_uring registered buffers. A test registers fixed buffers with `io_uring`, performs fixed-buffer writes, migrates the underlying pages between NUMA nodes via `move_pages`, then reads back and verifies data integrity. All data matches after migration, confirming that the `vmap` alias created during buffer registration is correctly redirected.

Kernel vmap (vmsplice). A vmsplice-based test exercises the kernel `vmap` path by splicing user buffers into a pipe while concurrently migrating the pages. Zero integrity errors after sustained concurrent migration.

RDMA under active DMA. An RDMA stress test registers a buffer and runs atomic fetch-and-add operations in a tight loop while a second thread migrates the buffer between NUMA nodes via `move_pages`. After the test, every counter is verified against its expected value. Zero mismatches after sustained concurrent migration, confirming that the two-phase protocol preserves data integrity under active device DMA.

B.3 Kernel TLB Flush Cost (vmap Aliases)

§4.5 gives the kernel `vmap` flush cost as a ratio and says why it is the expensive case; this is the measurement behind it. We repeat the IOTLB ablation with `io_uring` registered buffers (kernel-vmap-pinned): *cold* buffers are registered but never accessed through the alias (alias PTEs are created with the access bit clear), while *hot* buffers have every page written through the alias by fixed-buffer I/O. Migrating 512 MB (5 runs): unpinned migration costs 3.5 μs per page, matching the baseline of §4.5; cold `io_uring`-pinned pages cost 3.5 μs and hot pages 14.5 μs. The 11.0 μs difference is one global kernel-TLB shootdown per page across all 112 CPUs, an order of magnitude above the per-page IOTLB flush of §4.5. The current implementation flushes each hot page individually; amortizing this would mean flushing the whole span the aliases occupy, as `vmalloc`'s lazy purge does, or a kernel primitive that invalidates a list of ranges.

C Artifact Appendix

C.1 Abstract

The artifact reproduces the paper's five experiments and rebuilds its figures: PostgreSQL in a device-assigned VM (four flavors), bare-metal RDMA latency across a migration, io_uring fixed-buffer IOPS across a memory-node change, memory defragmentation with pinned memory (plus the per-page migration-cost ablations), and CAS commit contention under active DMA. Everything is driven from a single Docker container; the experiments themselves execute on a separate server it drives over SSH. That server holds both kernels (the Virtual pinning kernel and an unmodified Linux 6.19 baseline) and reboots between experiments via kexec. The hardware the protocol depends on is not commodity; the

machine used for the paper's measurements is described below.

C.2 Artifact check-list (meta-information)

- **Algorithm:** pinned-page migration protocol (Linux kernel patches: MM, IOMMU, io_uring).
- **Program:** PostgreSQL/pgbench, patched perftest (`ib_write_lat`), fio, custom C microbenchmarks (included; built on the server by the artifact).
- **Binary:** both kernels, built and installed on the machine the experiments run on; the artifact carries their source as a patch series against Linux v6.19.
- **Data set:** generated (pgbench scale 100; two 2M-row tables), pre-loaded in a golden VM image.
- **Run-time environment:** driving machine: any OS with Docker and an SSH client. Server: Ubuntu 24.04, Linux 6.19 (Virtual pinning and vanilla).
- **Hardware:** dual-socket Intel Xeon (4th gen), 128 GB+ per NUMA node, VT-d scalable mode with SSADE (IOMMU-maintained access/dirty bits, which the protocol relies on; §B.1), Mellanox ConnectX-6 Dx with SR-IOV.
- **Execution:** one command; the server must be otherwise idle, since a pass kexec-reboots it many times.
- **Metrics:** transaction throughput and latency, RDMA latency, aggregate IOPS, hugepage recovery, CAS retry rate.
- **Output:** per-run CSVs, aggregated tables, and the paper's figures as PDFs, drawn by the paper's own plotting scripts.
- **Experiments:** the paper's five, at its five-run protocol; `RUNS=1` gives a single-run pass.
- **How much disk space required (approximately)?:** ~1 GB on the driving machine; ~80 GB on the server.
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes (build the container; server access needs no client install).
- **How much time is needed to complete experiments (approximately)?:** about 7 hours for the full protocol, and about 2 hours with `RUNS=1`, as measured on the machine described in §B.1.
- **Publicly available?:** yes, as an archival deposit.
- **Archival DOI:** https://doi.org/10.5281/zenodo.21909764.

C.3 Description

C.3.1 How to access. The artifact is archived under a DOI that always resolves to the most recent version of the deposit (https://doi.org/10.5281/zenodo.21909764). It contains the experiments, the plotting scripts, and the kernel patch series the paper describes. Because the experiments need hardware that is not commodity (§B.1), a suitably equipped machine was made available for the artifact evaluation, with its address and credentials sent privately to the AE chairs as the evaluation instructions direct for such arrangements.

C.3.2 Hardware dependencies. The protocol rests on an IOMMU that records device accesses in hardware, so the machine must have Intel VT-d in scalable mode with SSADE (Scalable-mode Accessed and Dirty Flags Enable), which 4th-generation Xeon Scalable processors and later provide. This requirement cannot be worked around: without it there is no way to learn which pages a device has touched, and the experiments have nothing to measure. Beyond that, the RDMA and device-assignment experiments need an SR-IOV-capable RDMA NIC with two virtual functions configured (we use a Mellanox ConnectX-6 Dx), and the migrations need two NUMA nodes with 128 GB or more each. The measurements in this paper were taken on a two-socket Xeon Gold 5420+ with 112 cores and 256 GB, which satisfies all of this.

C.3.3 Software dependencies. The machine driving the experiments needs only Docker, an SSH client and a network connection; every tool the experiments themselves use is inside the container. The server needs Ubuntu 24.04 or similar, both kernels built and installed, QEMU built from source (distribution packages predate the `-object iommufd` support the device-assigned VM needs), a patched `perftest` for the RDMA latency measurements, and a VM image with the PostgreSQL tables preloaded. Installing them is described below.

C.3.4 Security, privacy and ethical concerns. The experiments require root on the machine they run on, and they use it intrusively: they replace the running kernel by `kexec`, load kernel modules, create SR-IOV virtual functions and bind devices to VFIO. They should therefore be run on a machine dedicated to the purpose, holding nothing of value. The artifact collects and transmits nothing about whoever runs it.

C.4 Installation

Two sides: the container that drives the run, and the server it drives. On the driving machine:

```
tar xzf pin-migration-artifact-<rev>.tar.gz
cd pin-migration-artifact
docker build -t pinmig-ae -f docker/Dockerfile .
```

Write the server's address and account into `benchmarks/config.local.sh`, and put the SSH key where the container can read it. Those two are the only configuration; `benchmarks/config.sh` lists the defaults they override.

The server is prepared once; reviewers using the machine we provided can skip it. Build the two kernels from `kernel/` — the series applied to Linux v6.19, and an unmodified v6.19 — and install them where `AE_KERNEL_DIR` points, as `kernel/README.md` describes. Then `machine/setup-machine.sh`, run from the container with the mounts shown below, installs the packages, builds QEMU and the patched `perftest`, and preloads the PostgreSQL VM image. It skips any step whose output exists, and takes about 90 minutes.

C.5 Experiment workflow

One command runs everything — all five experiments at the paper's protocol, followed by aggregation and figure

generation. The three mounts are the settings, the SSH key and the directory the results are written to:

```
docker run --rm \
  -v "$PWD/benchmarks/config.local.sh:\
      /artifact/benchmarks/config.local.sh:ro" \
  -v "<key file>:/root/.ssh/key:ro" \
  -v "$PWD/results:/artifact/data" \
  pinmig-ae ./benchmarks/run_everything.sh
```

Add `-e RUNS=1` for a single-run pass. The container checks the mounts and the connection to the server before starting any experiment.

C.6 Evaluation and expected results

Results land in `results/` on the driving machine: `figures/` holds the paper's figures, drawn by the paper's own scripts; `aggregated/` holds the tables behind them, including before/after summaries per experiment. Expected outcomes, comparing shape and rough magnitude (machines differ; error bars are the spread across runs): point lookups run at ~19 ktps under SR-IOV against VirtIO's ~10 ktps, with Virtual pinning matching SR-IOV throughout; hash joins improve by ~16% after migration for VirtIO and Virtual pinning, while unmodified SR-IOV degrades; RDMA latency drops ~29% after migration for ODP and Virtual pinning, with ODP's transient an order of magnitude taller; io_uring IOPS improve after the node change on the Virtual pinning kernel (growing with core count) and stay flat on the baseline; hugepage recovery stays ~89% under Virtual pinning versus collapsing to ~41% on the baseline at 32 GB pinned; CAS retries decay from ~99.7% at one page under DMA toward ~1.3% at 128 pages.

C.7 Experiment customization

`RUNS` sets repetitions per experiment; `benchmarks/config.sh` documents every other setting (kernel image paths, working directories, network fabric). The defaults match the machine described in §B.1.

C.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae