

SagePTE: Co-Locating Guest and Host PTEs to Accelerate Virtual Memory Translation

Amir Noohi
School of Informatics
University of Edinburgh
Edinburgh, UK
amir.noohi@ed.ac.uk

Mostafa Derispour*
Dept. of Electrical and Computer Engineering
Isfahan University of Technology
Isfahan, Iran
m.deris@ec.iut.ac.ir

Rodrigo C. O. Rocha
Huawei
Edinburgh, UK
rodrigo.rocha@huawei.com

Antonio Barbalace
School of Informatics
University of Edinburgh
Edinburgh, UK
antonio.barbalace@ed.ac.uk

Artemiy Margaritov
Huawei
Edinburgh, UK
artemiy.margaritov@huawei.com

Abstract—Hardware-assisted virtualization relies on two-dimensional (2D) nested address translation, trading the costly software traps of Shadow Page Tables for high access-time latency. This overhead is heavily concentrated at the leaf of the page walk: the final host page-table traversal required to resolve the data page. To eliminate this bottleneck, we introduce SagePTE, a hardware-software co-design that bridges the gap between nested and shadow paging by shadowing *only* the leaf PTEs, directly *in-place* within the guest page table. By expanding the guest page table layout on-demand in host memory, SagePTE co-locates guest and host mappings into a unified 128-bit extended entry. To maintain strict isolation while guaranteeing trap-free leaf synchronization, our design seamlessly repurposes existing silicon primitives, such as 128-bit compare-and-swap logic. Zero-delay hardware access mediation within the core’s memory execution path safely masks guest reads and upgrades guest writes, triggering automatic hardware synchronization. This allows the page walker to fetch both translations simultaneously and bypass the final host walk entirely. SagePTE operates purely beneath the virtualization boundary, supporting unmodified guests without para-virtualization. We formally verify SagePTE’s isolation guarantees using Alloy and evaluate it on ARM, demonstrating average end-to-end speedups over nested translation of $1.30\times$ with 4 KB pages and $1.22\times$ with Transparent Huge Pages (THP).

Index Terms—virtual memory, address translation, nested paging, page tables, virtualization, TLB

I. INTRODUCTION

Modern cloud infrastructure relies on hardware-assisted virtualization to securely multiplex resources—from general-purpose instances to virtualized network functions [1]—but this flexibility incurs a severe memory performance tax. Architects traditionally face a stark choice: pay high synchronization costs upfront with *Shadow Page Tables (SPT)* [2], which induce a VM exit on every guest page-table update, or pay high latency costs at access time with *hardware nested address translation* [3], [4].

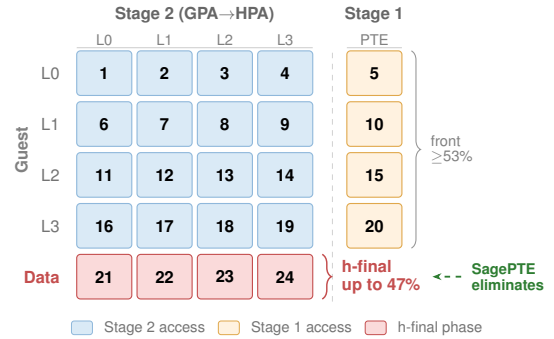


Fig. 1. Anatomy of a 24-reference nested walk. The terminal Stage 2 traversal (red, accesses 21–24) constitutes the *h-final phase*, dictating up to 47% of total walk latency. SagePTE structurally eliminates this bottleneck.

While nested translation avoids software traps, the penalty is steep: a single TLB miss can require up to 24 sequential memory references, because the walk is two-dimensional (2D)—every step of the *Stage 1* walk through the guest’s page tables (GVA→GPA) is itself translated by a *Stage 2* walk through the hypervisor’s tables (GPA→HPA). As application datasets outgrow TLB reach [5]–[9]—and AI inference now pushes working sets beyond a single node’s memory [10], [11]—this 2D walk increasingly dominates execution time. Yet, this latency is not uniformly distributed. Consistent with prior observations [12]–[14], our measurements show the overhead concentrates overwhelmingly at the *leaf* of the walk. Specifically, as shown in Figure 1, resolving the final host mapping for the data page—the *h-final phase*—accounts for up to 47% of total translation time across cache-hostile workloads. Eliminating this single bottleneck yields page-walk speedups of up to $1.91\times$.

This paper asks: can we achieve the low access latency of shadow paging without sacrificing the robust, trap-free execution of a 2D walk? Our key insight is that we do not need

*This work was done while the author was at University of Edinburgh.

a disjoint, software-managed shadow tree. Instead, we propose shadowing *only* the leaf PTEs, and doing so *in-place* directly within the guest page table. By co-locating the host mapping directly alongside the guest leaf, we bypass the latency penalty exactly where it hurts the most.

Crucially, maintaining this localized, in-place shadow without hypervisor intervention is now feasible thanks to existing microarchitectural capabilities. We do not invent complex new hardware; modern silicon already features robust datapath logic designed to support 128-bit compare-and-swap (CAS) instructions [15] and sub-page access permissions [16], [17]. Our design seamlessly repurposes these existing silicon primitives directly within the MMU to trigger automatic, trap-free synchronization during a standard page walk.

Because virtual memory optimizations are rarely one-size-fits-all, SagePTE provides *application-granularity control*. By design, the architecture shifts synchronization overhead strictly onto the low-frequency memory allocation path, actively removing it from the high-frequency memory access path. Shadow paging also keeps the miss path short, but pays for it on *every* guest page-table write—including the accessed- and dirty-bit updates that dominate page-table traffic—whereas SagePTE traps only when a Level 3 table is allocated or freed, an event orders of magnitude rarer; leaf updates are mediated in hardware without a VM exit. Consequently, the hypervisor selectively activates the mechanism exclusively for memory-aggressive workloads such as key-value stores, graph analytics or distributed LLM inference [18], ensuring the one-time setup cost is rapidly amortized over millions of subsequent, accelerated accesses.

Once enabled, SagePTE operates strictly beneath the virtualization boundary, demanding zero para-virtualization and supporting completely unmodified guest operating systems via three synergistic mechanisms. First, the page table layout is expanded strictly lazily. On the initial page walk to a standard guest leaf, the hardware raises a novel expansion interrupt—conceptually identical to a Stage 2 page fault. The hypervisor intercepts this trap to reallocate the guest’s 4 KB table into an 8 KB contiguous physical region. This structural expansion creates the capacity for interleaved 128-bit entries (Figure 2), tracked by a host-only page table metadata bit. Second, we introduce *hardware guest pte access mediation* within the core’s memory execution path to preserve the guest’s architectural illusion. Utilizing translation metadata, zero-delay combinational logic in the Load-Store Unit dynamically routes accesses: it masks guest reads to 64 bits to hide host data, and upgrades guest writes to 128-bit atomic CAS operations to enforce strict isolation without hypervisor exits. Finally, a *page walker extension* repurposes this same atomic datapath. During a standard traversal, the MMU executes a hardware CAS to populate the *hePTE*, verifying that the guest mapping remains unmodified during the walk. Together, this co-design guarantees safe, hardware-driven synchronization directly in silicon.

Because the entire SagePTE architecture remains completely invisible to the guest, it perfectly preserves execution

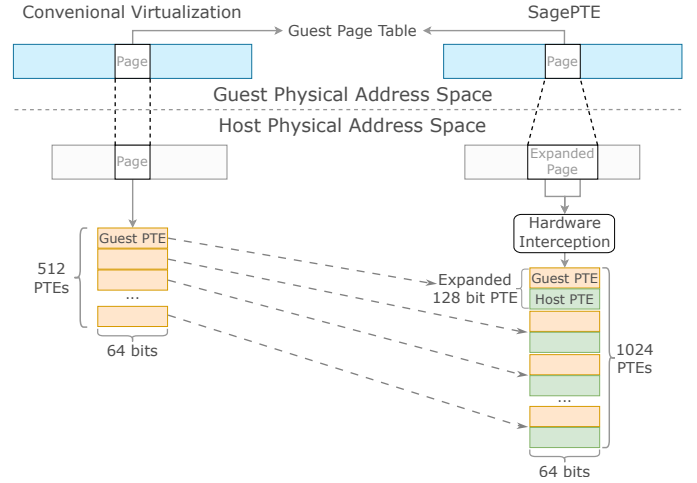


Fig. 2. Layout of the guest page table pages with nested translation and SagePTE.

for legacy kernels and seamlessly falls back to standard 2D walks if contiguous host memory fragments, ensuring robust, high-performance execution in modern datacenters.

In summary, this paper makes the following contributions:

- We show how hardware primitives similar to existing 128-bit atomics and sub-page write protection [15]–[17] enable local, trap-free synchronization at the PTE leaf, where most nested-walk latency arises.
- We propose *SagePTE*, an architectural co-design that co-locates guest and host PTEs, allowing the walker to bypass the final host page-table traversal while retaining an always-correct nested-walk fallback.
- We formalize *SagePTE* in Alloy, verifying its virtualization isolation and safety properties via bounded model checking.
- We prototype SagePTE on ARM in QEMU and in a DynamoRIO-based memory-hierarchy simulator, and show on representative memory-intensive workloads average end-to-end speedups of $1.30 \times$ (4 KB pages) and $1.22 \times$ (THP) over nested address translation.

The rest of the paper is organized as follows. Section II quantifies where nested-walk latency arises and derives the cost model that motivates synchronizing only the leaf. Section III presents SagePTE: the extended-PTE format and on-demand expansion (III-B), the page-walker and load-store datapath (III-C), the isolation and concurrency protocol (III-D), and correctness and portability (III-E). Section IV describes the simulation and prototyping methodology, Section V evaluates page-walk and end-to-end performance, overheads and sensitivity, and Section VI positions the design against prior work.

II. BACKGROUND & MOTIVATION

A. Virtual Memory & Virtualization

Virtualization enforces strict isolation without guest OS modifications, necessitating two distinct address translation

stages:

$$g : \text{GVA} \rightarrow \text{GPA} \quad \text{and} \quad h : \text{GPA} \rightarrow \text{HPA}.$$

In architectural terms, g is the *Stage 1* translation, walked through the guest’s own page tables, and h the *Stage 2* translation, walked through the hypervisor’s; each is a multi-level radix tree (Levels 0–3 on ARMv8, with Level 3 the leaf). For correctness, the hardware-implemented mapping f^* must rigorously refine this two-stage composition:

$$\forall a \in \text{GVA} : \quad f^*(a) = h(g(a)). \quad (1)$$

Historically, architects have satisfied (1) in one of two ways, each trading spatial tracking against temporal latency:

Shadow page tables (SPT): *Eager synchronization.* The hypervisor precomputes $s = h \circ g$ into a unified, host-owned structure. TLB misses are fast (native walk), but guest PTE updates trigger costly VM exits to synchronize s .

Nested page walk (NPW): *Sequential composition.* Hardware dynamically walks guest (G) then host (H) tables on every miss. Updates are trap-free, but translation latency is severe.

The Central Design Question: *Should we pay the cost upfront (synchronization via VM exits) or on the miss path (sequential memory accesses)?*

B. Where the Overhead Comes From

Let m be the TLB-miss rate, r_p the PTE update rate, r_c the context-switch rate, and $C_{\text{vm_exit}}$ the VM exit penalty. The amortized overheads are:

$$\text{Cost}_{\text{NPW}} = m \cdot \underbrace{(L_{\text{front}} + L_{\text{h-final}})}_{\text{nested page walk}}, \quad (2)$$

$$\text{Cost}_{\text{SPT}} = m \cdot L_{\text{shadow}} + (r_p + r_c) \cdot C_{\text{vm_exit}}, \quad (3)$$

where $L_{\text{h-final}}$ represents the terminal Stage 2 traversal resolving the data page, and L_{front} encapsulates the preceding walk.

Our simulations (Section IV) reveal that $L_{\text{h-final}}$ overwhelmingly dominates NPW latency, accounting for 41.4% of the walk time in Redis (Figure 3) and rising as high as 47% across other cache-hostile workloads (Section V-B). This concentration arises from fundamental cache dynamics. Upper-level translations benefit from page walk caches (PWCs) and exhibit high temporal locality. In contrast, the host PTE mapping the *data page* is indexed by the data page’s unique GPA. Its working set scales directly with the application’s memory footprint, thrashing the LLC and forcing accesses to main memory.

Conversely, while SPT minimizes walk latency (L_{shadow}), the $C_{\text{vm_exit}}$ penalty—often thousands of cycles [19], [20]—makes it prohibitive for modern dynamic workloads. SPT beats NPW only if the latency reduction ($L_{\text{front}} + L_{\text{h-final}} - L_{\text{shadow}}$) outweighs the continuous synchronization penalty $(r_p + r_c) \cdot C_{\text{vm_exit}}$, a mathematically rare scenario for cloud applications that rapidly map, unmap, and modify memory permissions.

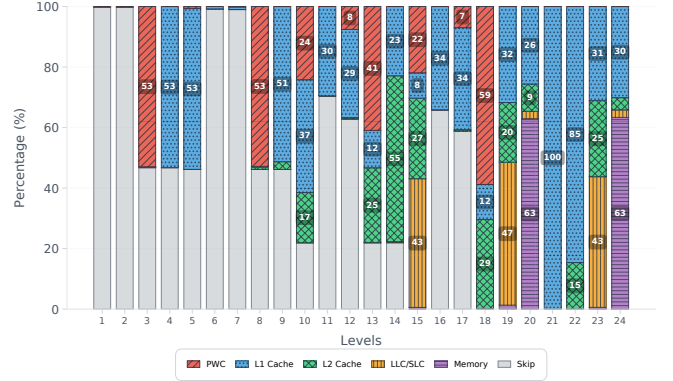


Fig. 3. Serving component of the memory hierarchy for each memory reference during a page walk in Redis.

C. A Route to Overhead-Free Synchronization

The cost models reveal our opportunity: rather than synchronizing the entire $h \circ g$ tree and repeatedly paying $C_{\text{vm_exit}}$, we can synchronize *only* the latency-critical $L_{\text{h-final}}$ component. When the hardware walker reaches the guest Level 3 leaf, it has already resolved the target GPA and verified Stage 1 permissions; the only remaining task is the GPA→HPA translation, which is exactly what the co-located *hePTE* provides. If we co-locate a precomputed HPA directly alongside the guest PTE, the walker can return the HPA immediately, structurally bypassing accesses 21–24.

Placing a host-owned word next to each guest PTE raises two requirements, and each maps onto a primitive that commodity cores already have. First, the host word must stay consistent with the guest word it annotates even while the guest rewrites the latter. If both live in one 16-byte cell that hardware can read and compare-and-swap atomically—a *local atomicity domain* $\mathcal{D}_{16}$ (ARM FEAT_LSE128, x86 CMPXCHG16B)—then any guest write and any host update are serialized, and a host update can be made conditional on the guest word being unchanged. Second, the guest must be unable to read or write the host word, which calls for access control at 8-byte granularity within a page. General sub-page permissions [16], [17] provide this, but carry per-page bitmaps and lookups of their own. Because the interleaved layout is fixed—guest words at even 8-byte slots, host words at odd ones—a hard-wired alternating mask, enabled by a single metadata bit per page, suffices; we call it *structured interleaved isolation* II. When the bit is set, the load-store logic exposes only the lower 8 bytes of every 16-byte block to the guest and reserves the upper 8 bytes for the host, at no metadata cost.

By expanding guest page tables purely within host memory, we fuse the guest-visible PTE and a host-only replica into a single $\mathcal{D}_{16}$ cell governed by II. The hardware transparently masks guest accesses, preserving the exact illusion of a standard 64-bit page table. Meanwhile, atomic updates within $\mathcal{D}_{16}$ keep the host and guest halves perfectly synchronized without a single hypervisor trap.

Putting it together: Exploiting Frequency Asymmetry. By design, the architecture exploits the fundamental asymmetry between allocation and access frequencies: it shifts synchronization overhead strictly onto the low-frequency memory allocation path, actively removing it from the high-frequency memory access path. This is where the design parts company with SPT: in Eq. (3) the synchronization term $(r_p + r_c) \cdot C_{vm_exit}$ is paid on every guest PTE update and context switch, whereas here the only trapping events are Level 3 table allocations and teardowns, whose rate is orders of magnitude below r_p ; leaf updates themselves never trap. This hybrid approach yields a fundamentally superior cost equation:

$$\text{Cost}_{\text{Mixed}} = m \cdot L_{\text{front}} + \varepsilon_{\text{build}},$$

$$\text{and } \varepsilon_{\text{build}} = \frac{C_{\text{build}}}{W} \xrightarrow{W \rightarrow \infty} 0.$$

Here, C_{build} encapsulates the one-time, localized cost to allocate and populate the co-located *ePTE*. Crucially, because memory footprint expansion is vastly outpaced by memory references, this initial overhead amortizes to zero ($\varepsilon_{\text{build}}$) over W subsequent reuses. By leveraging $\mathcal{D}_{16}$ and II, our design structurally eliminates the long-latency $L_{h\text{-final}}$ from the steady-state miss path while completely avoiding the C_{vm_exit} penalty on guest updates. Ultimately, this places SagePTE exactly in the optimal region of the virtual memory design space—delivering the execution speed of shadow paging with the trap-free robustness of nested walks.

III. OUR PROPOSAL: SAGEPTE

To bypass the severe latency of nested address translation, we propose SagePTE, a hardware-software co-design that synchronizes guest and host translations directly at the page-table leaf. The name SagePTE reflects the architecture’s role as a “wise” mediator that provides the hardware with pre-calculated host translations alongside standard guest entries, eliminating redundant traversals.

A. Overview and Design Invariants

Figure 4 illustrates SagePTE’s architectural blueprint. Recall from Section II that translation proceeds in two Stages: Stage 1 (GVA→GPA) walks the guest’s tables and Stage 2 (GPA→HPA) the hypervisor’s; each is a radix tree of Levels 0–3, with Level 3 the leaf. Instead of maintaining a disjoint shadow structure or performing a full 2D walk, SagePTE employs *in-place, co-located synchronization*. The guest PTE and a hypervisor-managed translation cache reside within a single 16-byte atomic domain ($\mathcal{D}_{16}$). This allows the hardware page walker to fetch both translations simultaneously, structurally bypassing the final host traversal without incurring hypervisor traps.

Our architecture enforces four non-negotiable invariants:

- 1) **Unmodified Guest:** The guest OS manages memory assuming standard 64-bit PTEs. SagePTE must require zero guest OS modifications and remain entirely invisible to guest software.

- 2) **Strict Isolation:** The guest is untrusted. It must be physically impossible for the guest to forge, manipulate, or exploit cached host translations to bypass hardware isolation.
- 3) **Trap-Free Leaf Synchronization:** The architecture must optimize for the common case. Steady-state page-table updates (Level 3 leaf mutations) must occur natively in hardware without triggering VM exits. We permit hypervisor traps strictly during rare, structural page-table allocations or teardowns (e.g., Level 2 directory updates) to manage physical capacity deterministically.
- 4) **On-Demand Activation:** Acceleration must not be a global mandate. The architecture must selectively enable synchronization for specific applications while seamlessly falling back to conventional 2D nested walks when disabled.

B. The SagePTE Abstraction: Format & Expansion

To physically enable in-place synchronization without violating the unmodified-guest invariant, we must decouple the guest’s logical view of the page table from its physical layout. To co-locate translations, SagePTE expands standard 64-bit guest PTEs into 128-bit extended PTEs (*ePTEs*). As shown in Figure 5, the lower 64 bits (*gePTE*) preserve the standard architectural Level 3 format, rendering them perfectly compatible with legacy guests. The upper 64 bits (*hePTE*) are strictly hypervisor-managed. They encode a 4-bit generation counter (which validates against a global Host Epoch Counter to instantly invalidate stale mappings, detailed in Section III-D), cached host attributes, and the resolved Stage 2 host physical address (HPA), while deliberately leaving bits reserved for future architectural extensions (e.g., 52-bit PA support).

Application-Granularity Control. To satisfy the on-demand invariant, SagePTE is not a global, always-on architectural mode. Instead, the hypervisor dictates precisely which applications or VMs receive acceleration via a dedicated *SagePTE Enable Bit*. When this bit is cleared, the MMU bypasses all SagePTE logic and seamlessly executes a standard 2D nested walk. This selective activation ensures the host dedicates expansion memory and hardware resources exclusively to the memory-bound workloads that genuinely require them.

Guest Page Table Expansion. Enforcing universal 128-bit allocations would violate the unmodified-guest invariant. Instead, SagePTE relies on the host OS to physically expand guest tables in host memory on-demand. When an expansion is required, the hypervisor allocates an 8 KB contiguous physical region (two consecutive 4 KB pages) from a host physical memory expansion pool. It then migrates the original table content into this expanded capacity using an interleaved layout: the host copies each 64-bit guest PTE into an even-offset slot ($2i$), initializing a *gePTE*, and leaves a 64-bit gap at the adjacent odd offset ($2i+1$), zeroing it to mark the future *hePTE* as invalid. The request is an order-1 (two-page) allocation, well below the order at which Linux lets a request fail rather than reclaim and compact for it (`PAGE_ALLOC_COSTLY_ORDER`,

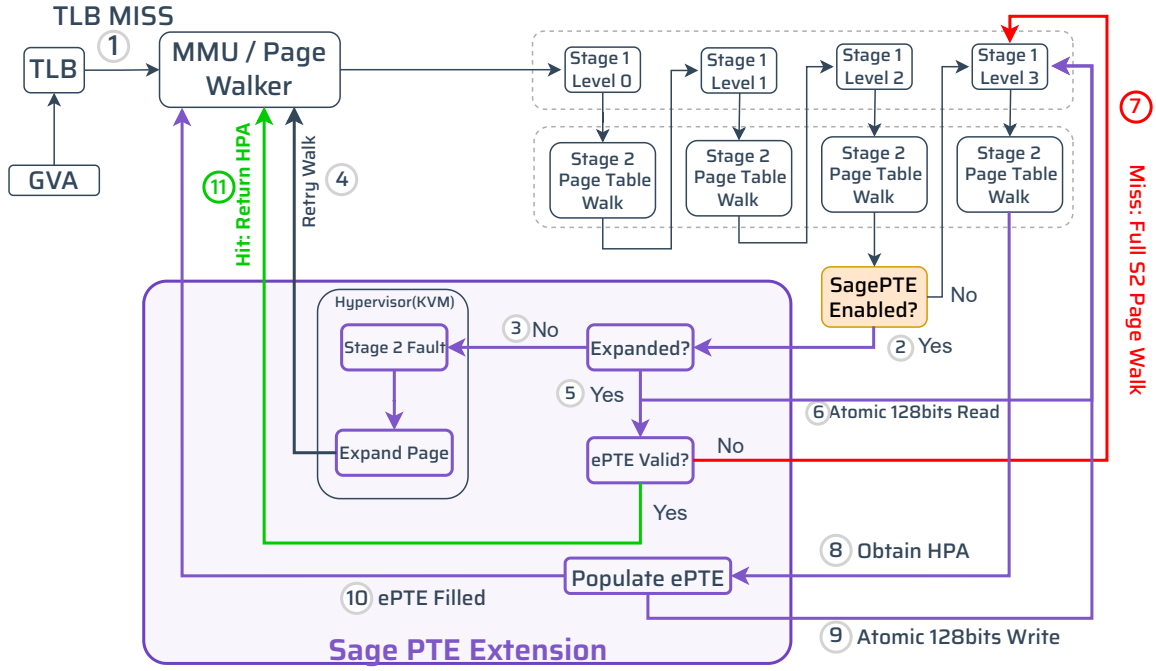


Fig. 4. SagePTE Design Overview. Numbered circles indicate the access sequence; red paths show miss scenarios requiring full Stage 2 walks, green paths show optimized hits.

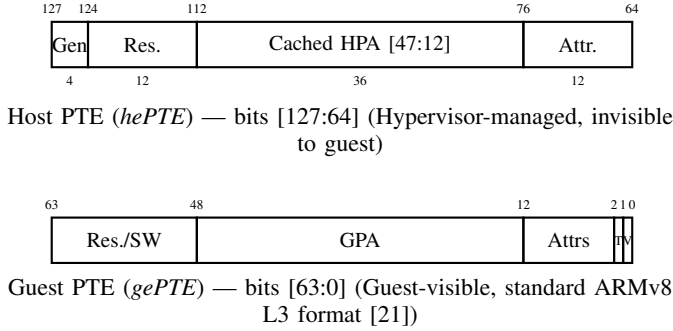


Fig. 5. 128-bit extended PTE (*ePTE*) format. The upper 64 bits (*hePTE*) cache the Stage 2 translation result and are invisible to the guest. The lower 64 bits (*gePTE*) follow the standard ARMv8 Level 3 page descriptor format [21].

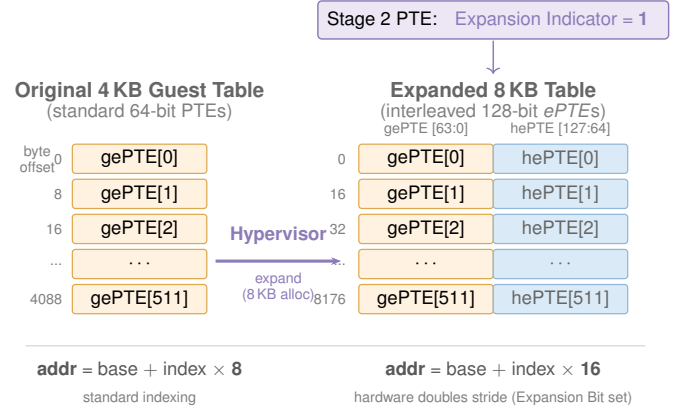


Fig. 6. On-demand expansion of a guest Level 3 table. The hypervisor allocates a contiguous 8 KB region and migrates each 64-bit guest PTE into an interleaved layout, leaving the adjacent *hePTE* slot zeroed (invalid). The hardware MMU doubles the index stride based on the *Expansion Indicator Bit* stored in the host's Stage 2 PTE.

order 3), so host fragmentation alone cannot deny it; only genuine memory exhaustion can. In that case the hypervisor leaves the table in its 64-bit state, preserving correctness via the standard 2D-walk fallback; Section V-E quantifies the cost of such partial expansion. Importantly, expansion doubles only the Level 3 page tables, leaving upper levels untouched. As evaluated in Section V-E, this adds a negligible 0.02%–0.27% to the VM's total memory footprint, avoiding any significant cache pressure.

To track this transformation, SagePTE introduces a dedicated *Expansion Indicator Bit* located strictly within the host system's Stage 2 PTE that maps the guest table. In ARMv8, this is safely encoded using one of the `IGNORED` bits in the Stage 2 Level 2 (PMD) descriptor that maps the guest's Level 3 table page, while the per-VM *SagePTE Enable Bit*

resides in a hypervisor system control register (e.g., a reserved bit in `VTCTR_EL2`). Because guests cannot access Stage 2 tables or hypervisor registers, these control structures cannot be forged by the guest. When the hardware MMU encounters a guest table where this bit is cleared (and the *SagePTE Enable Bit* is set), it safely traps to the hypervisor via a standard Stage 2 fault. This prompts the host OS to perform the 8 KB allocation and interleaved copy. Once the expanded table is constructed, the hypervisor atomically updates the Stage 2 PTE to point to the new 8 KB physical base, sets the *Expansion*

Indicator Bit, and write-protects the physical page containing the guest’s parent Level 2 directory within the host’s translation tables. This write-protection is strictly necessary to enable the synchronous capacity reclaim mechanism discussed in Section III-D. The guest remains completely oblivious; observing this set bit, the hardware MMU dynamically doubles the index offset ($\text{TABLE_BASE} + \text{index} \times 16$) for all subsequent accesses to this table.

C. Hardware Datapath: Page Walk & Mediation

To guarantee strict isolation while eliminating hypervisor traps during steady-state memory accesses, the hardware must intercept and mediate guest operations on the expanded tables natively. To exploit the *ePTE* securely, we introduce localized modifications to the hardware page walker and the Load-Store Unit (LSU), utilizing existing single-copy 128-bit atomic primitives. Crucially, these 128-bit operations do not necessitate new bus widths: existing interconnects (e.g., ARM AMBA CHI, Intel UPI) natively support 128-bit data granularity within standard 64-byte cache line transfers. SagePTE simply utilizes the existing architectural guarantees provided by modern ISAs (such as ARM FEAT_LSE128 or x86 CMPXCHG16B) on naturally 16-byte-aligned addresses. To manage enablement and invalidation across all cores, the architecture relies on the aforementioned *SagePTE Enable Bit* alongside a global *Host Epoch Counter*, both managed exclusively by the hypervisor.

The Page Walk Protocol. Figure 4 tracks the augmented hardware state machine. Upon a TLB miss (❶), the MMU performs the standard 2D walk for the upper table levels. The structural divergence occurs exclusively at the Stage 1 Level 3 leaf.

If the *SagePTE Enable Bit* is set (❷), the MMU consults the TLB-cached *Expansion Indicator Bit*. If cleared (❸), it immediately triggers a Stage 2 fault, prompting hypervisor-driven expansion before retrying the walk (❹).

If the table is already expanded (❺), the MMU issues an atomic 128-bit read (❻) to fetch the complete *ePTE*. It validates the *hePTE*’s generation counter against the global *Host Epoch Counter*. If valid, the MMU registers a *hePTE* hit (the green path, ⓫). It immediately returns the cached HPA to the core, completely bypassing the terminal Stage 2 traversal.

If the validation fails, the MMU registers a *hePTE* miss (the red path, ❷) and executes the full Stage 2 walk to obtain the requisite HPA (❸). To synchronize this result for future walks, the MMU executes a 128-bit atomic compare-and-swap (CAS) write (❹). Provided the guest has not concurrently modified the *gePTE*, this hardware operation successfully populates the *hePTE* (❺), concluding the sequence.

Guest PTE Access Mediation. When the guest directly reads from or writes to an expanded Level 3 table, SagePTE leverages the core’s memory execution path to mediate the operation natively, enforcing security and preserving the 64-bit architectural illusion. Although these memory operations are physically executed by the Load-Store Unit (LSU),

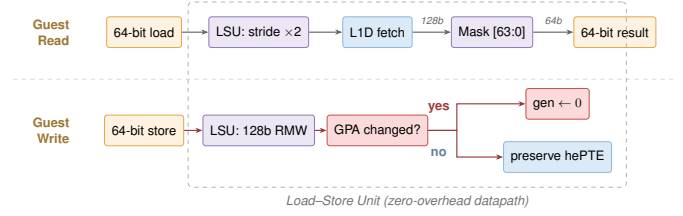


Fig. 7. Life of an instruction: Hardware guest PTE access mediation in the LSU. (1) For reads, the LSU initiates a 128-bit fetch and dynamic masking to hide host state from the guest. (2) For writes, the store logic upgrades the access to a 128-bit atomic RMW, using a hardware comparator to selectively invalidate the *hePTE* generation if the target GPA is altered, while preserving it for dirty/accessed-bit updates.

the datapath is dynamically steered by translation meta-data—specifically, the *Expansion Indicator Bit* cached by the MMU in the TLB during the virtual address translation phase.

Observing this indicator, the hardware MMU dynamically doubles the index stride (i.e., $\text{TABLE_BASE} + \text{index} \times 16$) for all subsequent accesses to this table. The guest remains completely oblivious to this physical reallocation. This tightly coupled logic allows the hardware to locate the correct 128-bit entry and the LSU to mediate the access transparently, with zero pipeline bubbles:

For guest reads, the LSU initiates a standard load using the scaled index. While the L1D cache returns the full 64-byte cache line, the LSU’s load data path—alerted by the *Expansion Indicator Bit*—dynamically masks the result immediately at the L1D interface. By enforcing this mask before the data can enter the core’s forwarding network, load queues, or speculative execution paths, the hardware guarantees the hypervisor-managed *hePTE* remains structurally isolated from all guest state.

For guest writes, the store logic mediates the standard 64-bit instruction and upgrades it to a 128-bit atomic read-modify-write (RMW) flow. This operation issues a 128-bit request to the L1D cache, upgrading the corresponding cache line to an exclusive state via the standard coherence protocol. Before the store commits, a hardware comparator examines the 64-bit guest update:

- If the guest’s write alters the physical address (GPA) field (bits [47:12]), the LSU controller automatically zeroes the *hePTE*’s generation field. This atomically invalidates the cached HPA, forcing the next page walk to re-validate the mapping.
- If the write only modifies permissions or metadata (e.g., setting an Accessed or Dirty bit), the hardware safely preserves the *hePTE*.

By embedding this indexing and selective invalidation directly into the hardware datapath, SagePTE achieves robust, trap-free synchronization that is entirely transparent to the guest OS.

D. Coherence, Isolation, and Concurrency

The invariants of Section III-A must survive concurrent guest activity and host remapping; this subsection gives the

protocol that keeps them. Our threat model assumes an adversarial guest capable of arbitrary memory access and race exploitation. SagePTE guarantees isolation purely through hardware atomicity and a host-driven invalidation protocol.

Hardware-Enforced Security. Guests cannot forge *hePTEs* because the LSU unconditionally masks reads and intercepts writes. Any attempt to redirect a mapping (altering the GPA) triggers the LSU’s hardware comparator, which deterministically invalidates the *hePTE*. Stale or forged mappings are physically impossible to commit.

Host-Triggered Invalidation. When the hypervisor alters GPA-to-HPA mappings (e.g., during page migration or KSM), it must invalidate all cached *hePTEs*. SagePTE achieves this instantaneously via the global *Host Epoch Counter*. The hypervisor increments this counter, temporarily unsets the *SagePTE Enable Bit*, and issues a TLB shutdown. Instantly, all existing *hePTEs* mismatch the new epoch and are treated as invalid. The hardware degrades gracefully to baseline nested paging during the hypervisor’s operation, then lazily repopulates *hePTEs* with the new epoch once the enable bit is restored.

Wrap-Around Safety. The 4-bit *Host Epoch Counter* encodes 15 valid generations (1–15; value 0 is reserved for guest-induced invalidation). To prevent a stale *hePTE* populated under epoch E from aliasing a future epoch E after the counter wraps, SagePTE enforces an explicit *wrap barrier* before reusing any value. When the hypervisor is about to advance the counter from 15 back to 1, it first traverses its own list of expanded page-table pages and synchronously zeroes the generation field of every *hePTE*, then issues a TLB shutdown. Only after this sweep completes does the counter resume at 1. Because the hypervisor already maintains the list of expanded pages for capacity reclamation, the sweep is a sequential pass over a small fraction of host memory, and because it occurs only once per 15 host remapping operations (e.g., page migrations, KSM merges, ballooning), its amortized cost is negligible: the sweep is bandwidth-bound and, for our largest case (Btree, $\sim 184\text{K}$ expanded tables, $\sim 1.5\text{GB}$), completes in under 100 ms once per 15 remaps—below 1% of runtime even at a remap every few seconds. Widening the counter to 8 bits, using four of the reserved *hePTE* bits, makes wraps $17\times$ rarer.

Table Lifecycle & Synchronous Capacity Reclaim. Classic shadow paging (SPT) guarantees strict synchronization and capacity management by write-protecting the entire guest page table hierarchy. However, this incurs severe VM-exit overheads on every leaf-level PTE update. SagePTE adopts a fundamentally asymmetric approach, exploiting the natural frequency disparity between page table levels. Because the vast majority of memory management operations target page-table leaves, SagePTE grants the guest OS unrestricted, trap-free write access to all Level 3 tables, relying purely on the hardware datapath for synchronization.

To maintain deterministic visibility over table lifecycles, the hypervisor selectively write-protects only the physical host pages containing the guest’s Level 2 (and upper) directory tables. Consequently, when the guest OS frees a Level 3

table and attempts to clear the corresponding Level 2 pointer, the hardware triggers a Stage 2 permission fault. Because structural directory updates are orders of magnitude less frequent than leaf mutations, this rare VM exit serves as an efficient, architectural callback. The hypervisor intercepts the fault, checks if the unmapped Level 3 table was physically expanded, and if so, synchronously reclaims the 8 KB block for the expansion pool before emulating the guest’s Level 2 update.

To strictly bound host memory overhead and prevent expansion-based resource exhaustion, the hypervisor couples this reclaim mechanism with a per-VM expansion quota. The host tracks the total number of 8 KB expanded pages allocated to each guest. If a VM exhausts its quota, the hypervisor ceases to service new expansion faults for that VM. It simply leaves newly allocated Level 3 tables in their native 4 KB, 64-bit format (with the Expansion Indicator cleared). This effectively pauses SagePTE acceleration for those specific tables, allowing the hardware to gracefully fall back to baseline 2D nested walks. Acceleration dynamically resumes only when the synchronous L2 reclaim mechanism frees sufficient capacity back to the pool.

Concurrency Resolution. Because every operation on an *ePTE* utilizes 128-bit single-copy atomics, race conditions resolve cleanly across all scenarios:

- **Concurrent Population:** If multiple cores attempt to populate the same *hePTE*, the CAS instruction ensures exactly one succeeds. Others observe a changed *ePTE* and cleanly abort.
- **Guest Writes During Updates:** If a guest writes to a PTE while the MMU is populating an *hePTE*, the atomic memory operations serialize. If the guest write wins, the MMU’s CAS fails. If the MMU wins, the guest’s subsequent atomic write triggers the hardware comparator, correctly destroying the now-stale *hePTE*.
- **Expansion During Invalidation:** If expansion occurs concurrently with a host epoch increment, the post-increment TLB shutdown forces the expanding core to observe the new epoch, ensuring no *hePTEs* are populated with stale generation metadata.

E. Correctness & Architecture Neutrality

SagePTE must strictly preserve the fundamental architectural refinement condition: the hardware translation must identically match the logical composition of guest and host tables ($f^*(a) = h(g(a))$).

Formal Verification. We formalized SagePTE’s atomic datapath, generation counter, and LSU masking in Alloy [22]. Bounded model checking (scope 4) exhaustively explored $\sim 10^9$ concurrent states. We successfully verified five architectural invariants (e.g., *hePTE* validity strictly follows atomic CAS, generation counters increment safely) and two security properties (no guest forgery, strict isolation). The model proved that under all race conditions, a guest can never inject an HPA, stale mappings are never consumed, and the translation always resolves accurately or correctly falls back to a full walk. Zero counterexamples were found.

Applicability to x86. While prototyped on ARMv8, SagePTE’s mechanisms are architecture-neutral and map naturally to x86 equivalents. The $\mathcal{D}_{16}$ abstraction maps directly onto CMPXCHG16B, available on modern x86-64 processors. Intel EPT serves as the Stage 2 equivalent, and EPT violations play the identical role as Stage 2 faults for triggering on-demand expansion. The required sub-page isolation mirrors Intel’s Sub-Page Protection (SPP) [16]. Because x86 utilizes similar multi-level radices, SagePTE targets identical latency bottlenecks across both major ecosystems.

IOMMU Applicability. Modern IOMMUs (e.g., ARM SMMU, Intel VT-d) utilize identical two-stage translation tables to map DMA requests. Consequently, an IOMMU page walker can seamlessly utilize the exact same *ePTE* format, generation counter, and validation logic. Because devices rarely initiate page table modifications, device-side writes trivially map onto the existing invalidation and synchronization paths, extending SagePTE’s latency reductions directly to I/O translation.

IV. METHODOLOGY

We evaluate SagePTE using a combination of hardware simulator and real workload runtime with hardware counters obtained using Linux perf tool [23]. Our evaluation infrastructure consists of extensions to the DynamoRIO simulator [24], [25] to simulate our work and other advanced designs, and instrumentation to track page table walk behavior across various workloads.

Following the methodologies of prior works [12], [13], [26], we first determine the page walk latency for nested paging and all advanced designs using a simulator-based approach. We then execute the workloads in a VM with vanilla Linux nested paging and measure the real page walk overhead using Linux Perf Tool and hardware counters. Finally, we utilize these measurements to measure the runtime under each design according to the following formula:

$$T^D = (T^{\text{NPW}} - T_{\text{PW}}) + \left(T_{\text{PW}} \times \frac{T_{\text{sim}}^D}{T_{\text{sim}}^{\text{NPW}}} \right) \quad (4)$$

where T^{NPW} is the runtime under vanilla Linux nested paging, T_{PW} is the real page walk overhead measured using Linux Perf Tool, T_{sim}^D is the simulated average page walk latency for design D , and $T_{\text{sim}}^{\text{NPW}}$ is the simulated average page walk latency for nested paging.

A. Simulation and Prototyping

We capture memory access traces using the DynamoRIO instrumentation framework [27] and dump both guest and host page tables from the hypervisor. These traces and page table snapshots serve as inputs to our simulator to replay page walks and measure latency under SagePTE and baseline designs. Page walk latency is measured in cycles from TLB miss to HPA resolution, averaged across all walks during workload execution.

For each memory reference in a page walk, the simulator determines which level of the cache hierarchy serves the access

(L1D, L2, SLC, or DRAM) using the configuration in Table I. The simulator additionally models the ARM Neoverse N1 page walk cache (PWC) [28], which short-circuits upper-level translations when entries are warm. For each design under evaluation, the simulator replays every captured page walk and aggregates per-reference service latencies into a total walk latency.

For functional validation and overhead analysis, we modified QEMU to support 128-bit PTEs, the ACTLR_EL1 generation register, and the atomic hardware protocols described in §III. We extended the Linux kernel and KVM to handle expansion page faults triggered when the guest allocates new page table levels. To measure expansion overhead, we invoke the page fault handler via a dedicated syscall and measure latency using RDTSC and high-resolution kernel timestamps.

B. Workload Selection and Characteristics

We evaluate SagePTE using seven workloads spanning database systems, scientific computing, and benchmark suites. These workloads exhibit diverse memory access patterns, TLB miss rates, and page table walk overheads, enabling us to understand when and why SagePTE provides performance benefits.

Our benchmarks include Redis (155 GB working set), an in-memory key-value store; Memcached (95 GB), a distributed caching system; GUPS (128 GB), a synthetic benchmark with purely random memory updates representing worst-case TLB pressure; Graph500 (123 GB), which performs breadth-first search on large graphs; BTrees (125 GB), implementing B-tree operations; Canneal (62 GB), a PARSEC benchmark performing circuit routing optimization; and XSBench (84 GB), simulating Monte Carlo particle transport algorithms.

Our THP experiments enable Transparent Huge Pages in both the guest and host. Under this configuration, SagePTE targets the page-table level that maps 2 MB pages (Level 2), and the h-final phase corresponds to the Stage 2 walk that resolves the 2 MB data page GPA.

V. EVALUATION

We evaluate SagePTE to understand precisely when, why, and by how much it structurally outperforms existing address translation mechanisms. Our evaluation answers three core questions: (1) How effectively does SagePTE eliminate page walk latency? (2) How do these microarchitectural savings translate to end-to-end performance? (3) What are the practical memory and expansion overheads?

Our results demonstrate that SagePTE fundamentally shifts the performance bottleneck of virtual memory, delivering large end-to-end speedups across diverse workloads at near-zero steady-state overhead.

A. Experimental Setup

We model SagePTE using a DynamoRIO-based simulator. The configuration reported throughout this section mirrors an ARM Neoverse N1 (Table I), matching the Ampere® Altra® Max M96-30 that we target. The simulator is not tied to that

TABLE I
SIMULATOR CONFIGURATION PARAMETERS [28]

Component	Specification
Processor	ARM Neoverse N1
Cores	96
L1I / L1D Cache	64 KiB, 4-way set associative
L2 Cache	1 MiB, 8-way set associative
SLC	16-way set associative
ITLB / DTLB	48-entry fully associative
STLB	1280-entry 5-way set associative

machine: the artifact also ships an x86 server configuration, and either can be replayed from the same capture, since the modelled microarchitecture is chosen at replay time rather than at collection time.

That separation matters for how the inputs are gathered. The simulator consumes a memory-reference trace together with guest and host page-table snapshots, and none of the three depends on the machine being modelled: the trace records which addresses a workload touches, and the snapshots record the translations in force while it does. We therefore collect them on an x86-64 server, an AMD EPYC 9255 with 629 GB of DRAM, running each workload inside a QEMU/KVM guest provisioned with 462 GB and Linux v5.15, with the host on Linux v6.8. The walk itself is reconstructed afterwards, when the trace is replayed against the modelled hierarchy.

Performance-Projection Methodology. To project end-to-end performance accurately, we follow the two-step methodology established in virtual-memory research [12], [13], [29]. We first measure the actual per-workload page-walk cost (T_{PW}) on real silicon using the hardware Performance Monitoring Unit (PMU). We then use our memory-hierarchy simulator to calculate the relative page-walk cycle ratio (T_{sim}^D/T_{sim}^{NPW}) between SagePTE and baseline nested paging. By scaling the hardware-measured baseline with this simulated ratio, we ensure our projections natively capture complex microarchitectural behaviors (OoO execution, MLP, and DRAM row-buffer locality) that simple simulations miss. As Equation (4) (Section IV) demonstrates, the common microarchitectural overlap cancels out in the ratio, yielding a reliable and conservative performance projection.

B. Page Walk Latency Reduction

We isolate SagePTE’s direct impact on the translation datapath by measuring page walk speedups against baseline nested paging and six state-of-the-art alternative designs (detailed in Section V-D). We evaluate across both 4 KB and Transparent Huge Page (THP) configurations to capture shifting bottlenecks.

SagePTE achieves a geometric mean page walk speedup of $1.71\times$ for 4 KB pages and $1.66\times$ for THP.

The Cache-Hostile Advantage (4 KB). Workloads with large, sparse memory footprints (Redis, GUPS, XSBench) suffer from chronic cache thrashing during page walks. Figure 10 reveals that for these workloads, the terminal Stage 2 traversal step (the h-final phase) accounts for an overwhelming 42.5% to 47.0% of total translation latency. Because SagePTE



Fig. 8. Page-walk latency speedup with 4KB pages normalized to nested paging.

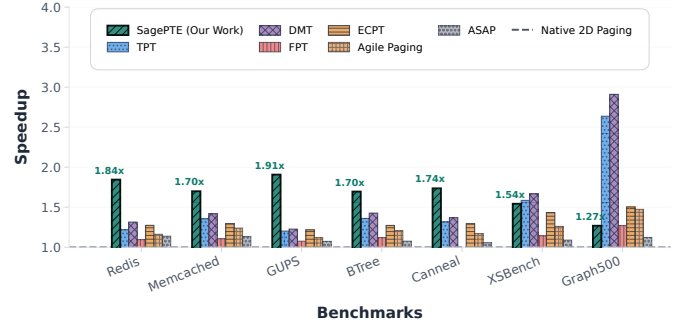


Fig. 9. Page-walk latency speedup with THP normalized to nested paging.

structurally bypasses these exact memory-bound accesses, it yields large speedups ranging from $1.71\times$ to $1.89\times$ (Figure 8).

The Cache-Friendly Shift. Conversely, Graph500 exhibits a highly cacheable page walk profile. Because its h-final phase accesses frequently hit in the LLC, they contribute only 20.5% to the total walk latency. Consequently, eliminating the h-final phase yields a proportionally smaller, though still substantial, speedup of $1.39\times$.

Access-Pattern Dependence. In short, the dominance of the h-final phase is a direct function of whether the data-page GPA stream exhibits sequential runs that can warm the cache. We observe a distinct three-point breakdown across our workloads: (1) *GUPS* exhibits purely random access, yielding a 44% h-final latency fraction; (2) *Graph500* exhibits batched-sequential access (CSR adjacency within frontier vertices), keeping leaf PTEs warm and dropping the fraction to 20.5%; and (3) *Redis* exhibits structured upper-level accesses but random leaf accesses, pushing the fraction to 42–47%. The asymmetry is structural: page-walk caches hold intermediate translations but never the leaf, which only the TLB caches, so an upper-level reference typically costs 10–20 cycles (a PWC or L2/LLC hit) while a leaf reference that misses to DRAM costs ~ 200 —a 10–20 $\times$ per-reference gap that lets 4 of 24 references account for 42–47% of the walk (Figure 3 shows it reference by reference for Redis). The same co-location could shadow the upper levels as well, but those are precisely the references the PWCs already serve; expansion memory spent there would buy little, which is why SagePTE targets the leaf

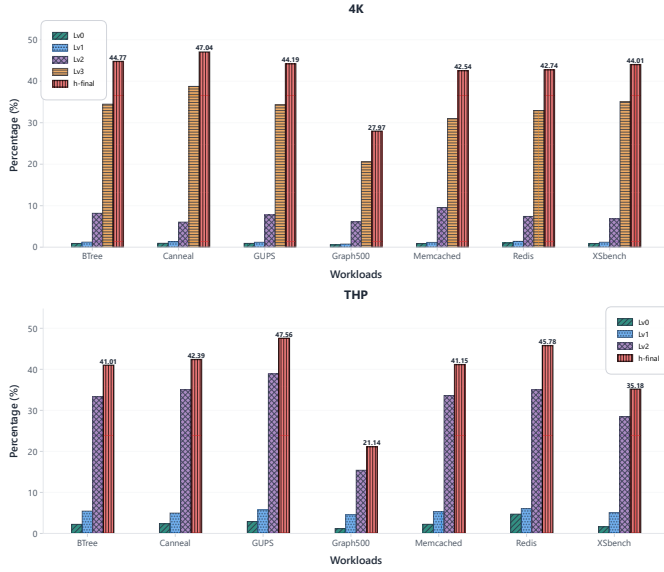


Fig. 10. Fraction of total translation latency contributed by the terminal Stage 2 traversal step per workload (4KB covers levels 21–24, THP covers 21–23).

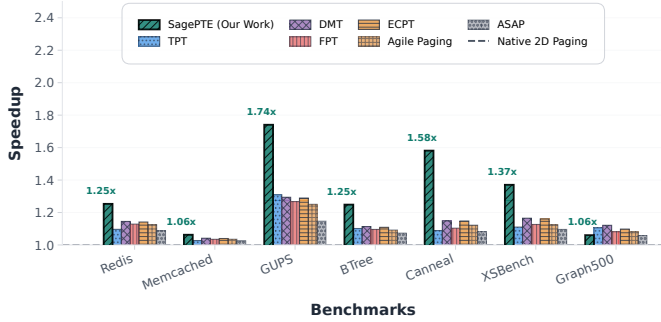


Fig. 11. Normalized end-to-end speedup with 4KB pages relative to nested paging.

alone.

The THP Tradeoff. THP introduces an architectural tradeoff. For workloads where huge pages naturally resolve cache thrashing (e.g., Canneal, XSBench), THP shrinks the absolute latency of the h-final phase. Consequently, SagePTE’s relative speedup compresses (e.g., dropping from $1.89\times$ to $1.74\times$ in Canneal). However, for workloads like Redis and GUPS, where access patterns defeat THP and h-final phase accesses remain stubbornly memory-bound, SagePTE continues to deliver dominant speedups of $1.84\times$ and $1.91\times$ respectively (Figure 9).

C. End-to-End Performance Impact

Microarchitectural latency reductions only matter if they move the needle on execution time. SagePTE achieves geometric mean end-to-end speedups of $1.30\times$ with 4 KB pages and $1.22\times$ with THP.

Performance scales strictly according to Amdahl’s Law (Figures 11 and 12). GUPS, which spends up to 96% of its

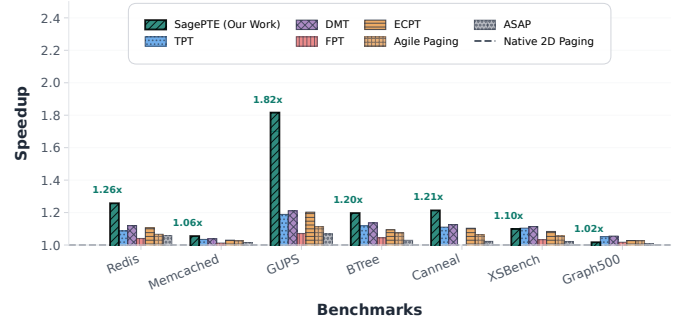


Fig. 12. Normalized end-to-end speedup with THP relative to nested paging.

TABLE II
END-TO-END GEOMETRIC MEAN SPEEDUP COMPARED ACROSS DESIGNS.

	ASAP	Agile	DMT	ECPT	FPT	TPT
4KB	$1.21\times$	$1.17\times$	$1.15\times$	$1.15\times$	$1.17\times$	$1.17\times$
THP	$1.18\times$	$1.15\times$	$1.09\times$	$1.11\times$	$1.22\times$	$1.11\times$

execution time stalled on page walks, translates SagePTE’s latency reductions almost perfectly into a large end-to-end win. Memory-intensive analytic workloads like XSBench and Canneal see similarly aggressive speedups. For Graph500 and Memcached, where page walks represent a smaller fraction of the total execution envelope, the end-to-end gains are naturally more bounded, though still strictly positive.

D. Comparison with Prior Works

We contrast SagePTE against six state-of-the-art acceleration strategies. While each attacks the 2D walk, they fundamentally fail to cleanly bypass the memory wall at the leaf:

Skipping Upper Levels (ECPT [30], DMT [12]): Both designs bypass the early Stage 1/Stage 2 steps using hashing or range descriptors, slashing the total access count from 24 to 3. However, the 3 remaining accesses lie precisely in the uncacheable h-final phase. They optimize the fastest part of the walk and ignore the slowest.

Tree Compression (FPT [31]): By fusing 4 KB nodes into 2 MB nodes, FPT halves tree depth. Yet, like ECPT/DMT, it still physically traverses the memory-bound tail entries, leaving the core bottleneck intact.

Prefetching (ASAP [13]): ASAP issues early prefetches for deeper levels while walking the upper levels. While effective for predictable walks, it still performs all 24 accesses, offering minimal relief when deep entries thrash the cache.

In-Place Mappings (TPT [32]): TPT, conceptually the closest design to SagePTE, embeds the HPA directly into the guest page table. However, to enforce isolation, it mandates an out-of-band permission table lookup on every access. Our simulations show this permission check behaves identically to an uncacheable page table read, reinjecting an 83% to 99% latency overhead and erasing the architectural benefit. SagePTE solves this via zero-overhead combinational logic in the LSU, requiring no extra memory accesses for security.

Because SagePTE is the only design that structurally eliminates the memory-bound h-final phase without introducing a new metadata bottleneck, it delivers superior end-to-end performance across nearly all workloads (Table II). The only exception is Graph500 under THP: because THP effectively caches the tail for this specific workload, designs that optimize the upper levels (like DMT) eke out a marginal advantage. However, because Graph500 spends only 7.9% of its time in page walks under THP, this microarchitectural edge translates to a statistically negligible end-to-end difference ($1.11\times$ for DMT vs. $1.10\times$ for SagePTE).

E. Overhead Analysis

a) *Memory Overhead.*: Expanding active 4 KB page tables to 8 KB mathematically doubles their capacity requirement. However, page tables inherently represent a tiny fraction of total memory. Across our evaluated workloads, absolute page table sizes range from 57 MB (Canneal) to 719 MB (Btree). Doubling this consumes only an additional 0.02% to 0.27% of total VM memory—a rounding error compared to the double-digit performance gains. A 64-byte line then holds four *ePTEs* instead of eight PTEs, doubling the leaf tables’ cache footprint; since leaf PTEs at these footprints already miss the caches—the very reason the h-final phase dominates (Section II-B)—the steady-state effect is small, and the upper-level tables, which the caches do serve, are untouched.

b) *Expansion Overhead.*: Constructing an *ePTE* table requires a one-time host intervention, costing an average of $3.01\mu\text{s}$. Profiling reveals this is dominated by the TLB shutdown ($1.34\mu\text{s}$, 44.6%) and the memory copy ($1.12\mu\text{s}$, 37.5%), with locking and barriers making up the rest.

Table IV quantifies this frequency. Sparse workloads like Btree trigger expansions aggressively (183,835 times under 4 KB), while dense workloads like Canneal expand rarely (14,581 times). Crucially, THP collapses this requirement entirely, plunging expansions to sub-400 across all workloads. In all cases, this one-time microsecond cost is thoroughly amortized. In the absolute worst-case scenario (Redis on 4 KB pages), the total expansion penalty consumes a mere 0.117% of total execution time. Crucially, unlike Shadow Paging or Agile Paging which trap on hot-path leaf PTE updates, SagePTE traps strictly on these rare structural expansions. For scale, recent analyses of shadow paging [12] measure trapping VM-exits as adding roughly 39% to total execution time; by contrast, SagePTE’s total trap overhead never exceeds 0.117%. Table III summarizes the trap and shutdown characteristics across the design space, demonstrating that SagePTE restricts all hypervisor interventions to rare, structural events.

c) *Fragmentation and Partial Expansion.*: Expansion requests an order-1 (8 KB) block. Linux serves such low-order requests by reclaiming and compacting rather than failing them—they lie below `PAGE_ALLOC_COSTLY_ORDER`—so host fragmentation alone does not deny them; only memory exhaustion, or the per-VM quota of Section III-D, leaves a table unexpanded, and that table is walked with the standard, functionally correct 64-bit 2D walk. Performance then

TABLE III
TRAP AND SHUTDOWN FREQUENCY ACROSS DESIGNS

Design	Hypervisor Interventions (VM Exits & Shootdowns)
Nested Paging	None (Zero exits)
ASAP, DMT, ECPT, FPT	None (Zero exits)
Shadow Paging (SPT)	Every guest leaf-PTE update (High frequency)
Agile Paging	Hot-path guest leaf-PTE updates (High frequency)
SagePTE	Rare structural only: L3 alloc/free ($< 0.117\%$ runtime) Shootdowns: Only on host remaps and epoch wraps

TABLE IV
EXPANSION OVERHEAD ANALYSIS

4K Pages			
Workload	#Expansions	Time (ms)	Overhead (%)
Btree	183,835	553.5	0.089
Redis	77,657	233.8	0.117
GUPS	65,554	197.4	0.070
Graph500	50,720	152.7	0.060
XSbench	42,724	128.2	0.012
Memcached	29,561	89.0	0.008
Canneal	14,581	43.9	0.002

Transparent Huge Pages (THP)			
Workload	#Expansions	Time (ms)	Overhead (%)
Btree	361	1.086	2.0×10^{-4}
Redis	158	0.475	3.0×10^{-4}
GUPS	132	0.397	2.0×10^{-4}
Graph500	111	0.334	2.0×10^{-4}
XSbench	87	0.262	3.0×10^{-6}
Memcached	58	0.175	2.0×10^{-6}
Canneal	31	0.093	4.0×10^{-6}

degrades gracefully rather than collapsing. If a fraction f of Level 3 tables stays unexpanded and walks hit them uniformly, the average walk time is $(1 - f)T_{\text{Sage}} + fT_{\text{NPW}}$, so to first order the geometric-mean page-walk speedup falls linearly from $1.71\times$ to $1.50\times$, $1.26\times$ and $1.09\times$ at $f = 20\%$, 50% and 80% with 4 KB pages (from $1.66\times$ to $1.47\times$, $1.25\times$ and $1.09\times$ with THP), never below the nested-paging baseline; end-to-end speedups scale with it through Eq. (4). Operators who want deterministic reserves can back the expansion pool with the Contiguous Memory Allocator (CMA).

d) *Compatibility with Page Table Sharing.*: Emerging proposals like `mshare` [33] reduce footprint by sharing page tables across processes. SagePTE naturally complements this: because expansion is tracked physically via the hypervisor’s Stage 2 PTE, a shared guest table is expanded exactly once. The resulting *hePTE* cache instantly accelerates every process mapped to that physical table, requiring zero deduplication or per-process metadata logic.

VI. RELATED WORK

Architects have long sought to mitigate the severe latency penalty of the 2D page walk. However, existing solutions universally demand architectural compromises—ranging from

intrusive guest OS rewrites to weakened hardware isolation—that SagePTE fundamentally avoids.

Radix Replacement Strategies. To reduce traversal depth, several proposals abandon the multi-level radix tree entirely. Hashed Page Tables [34], [35] compress the hierarchy into a flat, collision-prone inverted structure. Skip Page Tables [36] and DMT [12] replace intermediate tables with coarse-grained range descriptors. While these designs drastically reduce theoretical memory accesses, they often demand a fundamental departure from the pervasive multi-level radix abstraction assumed by dominant software ecosystems. They dictate extensive rewrites to the guest OS, require hardware changes that are not isolated to the datapath, and fail to degrade gracefully under severe memory fragmentation. SagePTE strictly preserves the standard architectural radix tree, ensuring unmodified guest execution.

Structural Flattening & Densification. Rather than replacing the radix tree, proposals like Compendia [37] and Every Walk’s a Hit [31] (which formalizes Flat Page Tables) attempt to compress it. They opportunistically merge adjacent 4KB page-table nodes into flat, dense 2MB contiguous blocks, halving the number of intermediate pointer chases. However, these techniques demand large, contiguous 2MB host allocations purely for the page-table structures themselves. When host memory fragments, the OS cannot satisfy these allocations, and the architecture brittly falls back to the deep, unoptimized tree. Furthermore, while flattening accelerates intermediate levels, it still physically performs the final memory-bound accesses at the uncachable leaf. SagePTE specifically accelerates this leaf without demanding megabyte-scale contiguous memory allocations.

Hybrid Datapaths (Ranges & Segments). Recognizing the limitations of strict radix trees, architectures like Redundant Memory Mappings (RMM) [38] and Direct Segments [39] attempt to mix standard 2D walks with hardware range descriptors. While these hybrid mechanisms excel for large, contiguous allocations, they are fragile: the moment memory fragments or an application demands fine-grained 4KB interleaving, the hardware falls back to the unoptimized, memory-bound nested walk. SagePTE provides robust, fine-grained leaf acceleration regardless of host memory fragmentation.

Shadow Paging Revivals. Acknowledging the VM-exit costs of traditional Shadow Page Tables (SPT), several works attempt to resurrect them via hardware-software co-design. SSP [40] intercepts TLB flushes to lazily synchronize disjoint tables, while Agile Paging [41] dynamically toggles the hardware walker between shadow and nested modes. While these architectures successfully reduce hypervisor interventions, they all still mandate the construction, maintenance, and eventual teardown of large, duplicated shadow structures in host memory. SagePTE achieves the exact access latency of a shadow leaf without ever maintaining a disjoint shadow tree or requiring specialized synchronization engines.

Datapath Prefetching & Prediction. Rather than altering the structure, ASAP [13] attempts to hide latency via base-plus-offset prefetching of deeper levels while the MMU

resolves the upper levels. Similarly, *Learning to Walk* [42] utilizes predictive models to replace the conventional radix-tree page table, predicting the physical address of the target PTE to collapse the walk. However, ASAP demands rigid physical contiguity from the OS to maintain its offset math. More importantly, both predictive and prefetching mechanisms still physically perform the memory accesses at the memory-bound leaf. As a result, they provide no relief when the deepest leaf entries actively thrash the cache; thus, they remain orthogonal and complementary to SagePTE, which deliberately preserves the existing radix-tree machinery while targeting the virtualization Stage-2 leaf.

Placement & Locality Optimizations. Recognizing the cache bottleneck, works like PTEMagnet [14] restructure PTE placement to maximize spatial locality and cache-line utility. While highly effective for native execution, this approach struggles against the realities of hardware virtualization. The terminal Stage 2 translation (the h-final phase) is inherently indexed by the data page’s Guest Physical Address (GPA)—a value that remains structurally scattered across the host physical memory as working sets scale. Locality optimizations can mitigate cache misses, but they cannot structurally eliminate the final traversal.

Translation Caches. Hardware page walk caches (PWCs, Intel EPT caches) successfully short-circuit the *front* of the nested walk. However, they systematically fail at the leaf because the final Stage 2 PTE mapping the data page exhibits near-zero temporal locality. SagePTE is strictly complementary: PWCs accelerate the front of the walk, while SagePTE structurally eliminates the uncachable tail. Nor can the leaf simply be cached away: Redis’s 155 GB footprint spans ~39 million leaf PTEs, about 310 MB of entries—far beyond any on-chip structure—and enlarging the TLB or PWC lengthens the critical path of every access, hit or miss [43]. SagePTE leaves the leaf in DRAM and makes the one fetch the walk already performs return the HPA.

Hybrid TLB Coalescing. Proposals like Park et al. [44] extend TLB reach by coalescing contiguous translations into single entries. This technique is orthogonal to our work: coalescing reduces the miss *rate*, whereas SagePTE minimizes the miss *penalty*. The two compose naturally.

The Trusted Leaf. TPT (Trusted Page Tables) [32] is the conceptually closest prior work, as it also embeds host physical addresses alongside guest PTEs. However, TPT carries a serious deployment limitation for modern cloud environments: it assumes a *trusted* guest OS. TPT grants the guest direct write access to host addresses, mandating an out-of-band permission table lookup on every single page walk to prevent isolation breaches. As our evaluation demonstrates, this permission check behaves identically to an uncached memory reference, reinjecting 83–99% of the total walk latency.

SagePTE assumes an *untrusted* guest. By utilizing zero-delay LSU masking and selective hardware invalidation, SagePTE guarantees isolation directly in the datapath. This yields a strict security advantage (zero guest trust required) and a substantial performance advantage (no per-access permission

check). Furthermore, SagePTE degrades gracefully to standard nested walks on a cache miss, whereas TPT stalls entirely if its permission entries are evicted.

Positioning in the Design Space. Traditional virtual memory design is trapped between two extremes. Shadow Page Tables (SPT) eagerly synchronize *all* translations, offering low access latency but high VM-exit costs. Nested Page Walks (NPW) lazily compose *all* translations, offering zero VM-exit costs but high access latency. SagePTE carves out a fundamentally new optimal point. By lazily synchronizing *only* the latency-critical leaf phase via co-located caching, and composing the upper stages dynamically, SagePTE breaks the dichotomy. It delivers the fast access latency of a shadow leaf without sacrificing the robust, trap-free execution of a nested walk.

VII. CONCLUSION

Sequential composition in nested page walks imposes up to 24 memory references per TLB miss, with the terminal Stage 2 traversal (the GPA-to-HPA resolution) dictating up to 47% of total walk latency across cache-hostile workloads. To structurally eliminate this bottleneck, we present SagePTE, an architecture that bridges the gap between nested and shadow paging. By expanding the guest page table layout strictly on-demand, SagePTE co-locates host translation results (*hePTE*) directly adjacent to guest PTEs (*gePTE*) within unified 128-bit entries (*ePTE*).

To preserve strict isolation while guaranteeing trap-free leaf synchronization, our design relies on zero-delay hardware access mediation within the core’s memory execution path. This securely masks guest accesses and triggers automatic hardware synchronization, allowing the hardware page walker to fetch both translations simultaneously and completely bypass the uncacheable host walk.

SagePTE operates purely beneath the virtualization boundary. It supports unmodified guest operating systems and relies on a rare, synchronous Level 2 capacity reclaim mechanism in the hypervisor to deterministically bound host memory usage. Our evaluation demonstrates robust end-to-end speedups of 1.30x and 1.22x for 4 KB pages and THP respectively, delivered with negligible memory capacity overhead (under 0.27%) and an amortized expansion cost of less than 0.12% of total execution time.

ACKNOWLEDGMENT

We thank the anonymous reviewers and the MICRO 2026 artifact evaluation committee for their feedback, which improved both the paper and its artifact. Claude (Anthropic) was used to edit and reorganize text and tables and to prepare the artifact appendix; all technical content, experiments, and claims are the authors’ own.

APPENDIX ARTIFACT APPENDIX

A. Abstract

This artifact contains the complete SagePTE evaluation pipeline: the memory tracer, the guest and host page-table

dumpers, the nested-page-walk simulator, and the analysis scripts. The captured inputs are not distributed—a single trace is 20 GB and decodes to 106 GB—so a reproduction captures them with the pipeline itself, which needs a QEMU/KVM host–guest pair and takes about four hours for Redis, and then replays the capture in 2.5 h per configuration. On our capture the simulator reports that the h-final phase contributes roughly 43% of Redis page-walk latency (Fig. 10) and that SagePTE achieves a 1.75× page-walk speedup over nested paging (Fig. 8), from which Eq. (4) projects the reported end-to-end gains. Every stage is documented below.

B. Artifact check-list (meta-information)

- **Algorithm:** in-place guest/host PTE co-location; 24-reference 2D page-walk replay.
- **Program:** DynamoRIO-based tracer and simulator (C++); page-table kernel modules (C); Python/Bash analysis.
- **Compilation:** gcc-7/g++-7, CMake ≥ 3.2 , driven by `build.sh`.
- **Data set:** produced by the artifact: a memory trace plus guest and host page-table dumps per workload (Redis: 155 GB working set, 20 GB trace, 39.7 M guest page-table entries). Not distributed.
- **Run-time environment:** Linux x86-64; the tracing guest needs glibc ≤ 2.33 (Ubuntu 18.04/20.04; we use 20.04), see Software dependencies. Root on guest and host, which run different kernels.
- **Hardware:** an x86-64 QEMU/KVM host–guest pair (guest Linux 5.15, host ≥ 6.1), guest memory for the working set (155 GB for Redis), AVX-512 hidden from the guest.
- **Execution:** for Redis, 4 h to capture, 11 min to decode, 2.5 h per simulated configuration.
- **Metrics:** page-walk latency in cycles and speedup per design; h-final latency share.
- **Output:** `Results/<workload>/sim_*.log` and `analysis_*.txt`, generated automatically.
- **Experiments:** `run.sh <workload>` end to end; each stage also has its own entry point.
- **How much disk space required (approximately)?:** 130 GB for Redis (20 GB capture, 106 GB decoded).
- **How much time is needed to prepare workflow (approximately)?:** 30 min, plus provisioning the guest.
- **How much time is needed to complete experiments (approximately)?:** about 7 h for Redis in one configuration; a second configuration replays the same capture (2.5 h).
- **Publicly available?:** yes.
- **Code licenses (if publicly available)?:** BSD (DynamoRIO-derived); GPLv2 (kernel modules).
- **Archived (provide DOI)?:** doi:10.5281/zenodo.21876937.

C. Description

1) *How to access:* The artifact is at <https://github.com/amirmnoohi/SagePTE-AE> and archived as doi:10.5281/zenodo.21876937. It holds the tooling; captures are produced by running it. The repository’s `README.md` expands this appendix and is the better reference while running it. The layout is:

```

SagePTE-AE/
|-- build.sh           build everything
|-- run.sh             capture and simulate
|-- clean.sh           remove captures, builds
|-- Tracer/            DynamoRIO memory tracer
|-- PageTables/Guest/  /proc/page_tables module
|-- PageTables/Host/   GPA->HPA module
|-- Simulator/         simulator and analysis
|-- Workloads/         benchmarks (vMitosis fork)
|-- Data/<workload>/    capture, written by run.sh
'-- Results/           simulator output

```

2) *Hardware dependencies*: Reproduction requires a QEMU/KVM host-guest pair with root on both machines, enough guest memory for the working set (155 GB for Redis), and 130 GB of disk in the guest for the capture and its decoded form. AVX-512 must be hidden from the guest (`-cpu host, -avx512f, -avx512dq, -avx512cd, -avx512bw, -avx512vl, -avx512ifma, -avx512vbmi, or clearcpuid=` on the guest kernel command line): the bundled DynamoRIO predates it and aborts at startup when it is exposed, which is the first thing to check if a run fails immediately on a recent server.

3) *Software dependencies*: The simulator needs gcc-7, CMake 3.2 or newer, and Python 3. Capture additionally needs QEMU/KVM, kernel headers matching the running kernel on both machines, and key-based SSH from the guest to the host.

The two machines need different kernels: the guest module reads `kvm_mmu` fields (`shadow_root_level`, `direct_map`) that later kernels removed, so the guest runs Linux 5.15; the host module uses the maple-tree VMA iterators of 6.1, so the host runs 6.1 or newer (we use 5.15 and 6.8).

The tracer is a DynamoRIO 7.0.0 fork whose private loader cannot load `glibc ≥ 2.34` (upstream issues #5437, #5431): Ubuntu 22.04 (`glibc 2.35`) crashes partway through tracing, whereas Ubuntu 18.04 (2.27) and 20.04 (2.31) work, so the tracing guest must run one of those. `Tracer/run.sh` checks and refuses on a newer `glibc`, saying why; decoding and simulation run standalone and are unaffected. `build.sh` installs the toolchain and reports any component it cannot build.

4) *Data sets*: Each run of `run.sh` writes `Data/<workload>/`: the drmemtrace capture, the guest page-table snapshot (VA, PE1..PE4, PA per mapped page) and the host snapshot keyed by guest-physical address; both formats are documented in `Simulator/README.md`. Captures are not distributed, for size, and no two are identical (see Evaluation and expected results).

D. Installation

```
./build.sh
```

This installs the toolchain and builds every component; those that cannot be built on the machine at hand are reported and skipped, so a machine without a KVM pair still gets a working simulator.

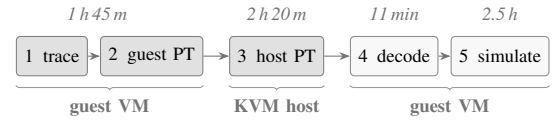


Fig. 13. Artifact workflow, timed for Redis on our evaluation machine. Shaded stages need the QEMU/KVM host-guest pair; stage 3 runs on the host and its result is copied back into the guest. `run.sh` drives all five.

E. Experiment workflow

Figure 13 shows the five stages and their cost for Redis. `./run.sh redis` runs all of them from inside the guest, reaching the KVM host over SSH for stage 3 (`--host, --host-user`); the stages can also be run by hand, as follows.

1) *Trace the workload (guest)*: The tracer starts the workload, begins recording once it signals that its working set is built (for Redis, after preloading 155 GB), then pauses it so the page table can be captured while the process is alive:

```
./Tracer/run.sh redis
```

2) *Capture the guest page table (guest)*: In a second terminal, while the tracer holds the workload; this reads `/proc/page_tables` and writes `Data/redis/pt_dump.guest` (39.7 million entries for Redis):

```
./PageTables/Guest/run.sh redis
```

3) *Translate on the KVM host*: Copy the guest snapshot to the host, resolve each guest-physical address against the running QEMU process, and copy the result back (2 h 20 min for Redis):

```
scp Data/redis/pt_dump.guest HOST:~/pt_dump.redis
ssh HOST ./PageTables/Host/run.sh ~/pt_dump.redis
scp HOST:~/pt_dump.host Data/redis/pt_dump.host
```

4) *Decode the capture*: Expanding the 20 GB capture to the 106 GB the simulator reads takes 11 minutes, once:

```
./Tracer/convert_trace.sh redis
```

5) *Simulate*: Each configuration replays the decoded trace in about 2.5 hours, locating the two page-table dumps automatically beside it:

```
cd Simulator
./run_x86.sh ../Data/redis
./run_arm.sh ../Data/redis
```

Each run reports progress and an estimate of the remaining time, and writes `Results/redis/sim_*.log` and `analysis_*.txt`.

F. Evaluation and expected results

On our Redis capture, under either configuration, nested paging costs 390 to 400 cycles per walk and the h-final phase accounts for roughly 43% of that latency, matching the 42.5 to 47% reported in Fig. 10. The page-walk speedups in Fig. 8 follow: $1.75\times$ for SagePTE, $1.35\times$ for DMT, $1.3\times$ for ECPT and FPT, and $1.2\times$ for Agile Paging. Scaling the measured Redis walk fraction of 47.8% by the simulated cycle ratio according to Eq. (4) projects an end-to-end speedup of $1.26\times$, against the $1.25\times$ reported in the paper.

One analysis pass reports every design (nested paging, SagePTE, DMT, ECPT, FPT, ASAP, Agile Paging); TPT (Table II) comes from a separate tool and is not reproduced.

Replaying a given capture is deterministic (fixed seed, LRU throughout), so re-simulating it reproduces its numbers exactly. A fresh capture is not expected to match ours: the guest builds a different page table every run, so the absolute latencies move, while the shape of the result—the h-final phase dominating the walk and the ordering of the designs—does not. That shape is what a reproduction should confirm.

G. Experiment customization

The modelled machine is entirely TLB, cache and core parameters passed as flags, so modelling a new one means copying a 40-line run script. Results redirect with `-o`, logs re-parse with `Simulator/scripts/analyze_log.sh`, and a workload is one definition file in `Workloads/`.

H. Failure checks and cleanup

Every stage exits non-zero on failure and names the log to read. The tracer reports success only if the DynamoRIO client acknowledged that recording started and the capture holds at least a million entries: the client polls for the start signal every 2^{26} instructions, so a workload that exits sooner records nothing, and such a run fails with that explanation rather than ending in an empty “complete” trace. The simulator refuses a truncated decode. `./clean.sh redis` removes a workload’s capture, decoded trace and results (`--decoded`: only the 106 GB decoded trace; `--build`, `--host`: build trees and host-side staging), listing sizes and asking first.

I. Methodology

Submission, reviewing and badging methodology: <https://www.acm.org/publications/policies/artifact-review-badging>, <http://cTuning.org/ae/submission-20201122.html>, <http://cTuning.org/ae/reviewing-20201122.html>.

REFERENCES

- [1] A. Khater, A. Noohi, M. R. Hashemi, and Z. Zali, “An NFV-based framework for autonomous deployment of new protocols in SDN networks,” *IEEE Access*, vol. 12, pp. 148 727–148 740, 2024.
- [2] E. Alkassar, E. Cohen, M. Hillebrand, M. Kovalev, and W. J. Paul, “Verifying shadow page table algorithms,” in *Formal Methods in Computer Aided Design*, 2010, pp. 267–270.
- [3] B. Pham, J. Vesely, G. H. Loh, and A. Bhattacharjee, “Large pages and lightweight memory management in virtualized environments: can you have it both ways?” in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–12. [Online]. Available: <https://doi.org/10.1145/2830772.2830773>
- [4] Z. Yan, J. Vesely, G. Cox, and A. Bhattacharjee, “Hardware translation coherence for virtualized systems,” *SIGOPS Oper. Syst. Rev.*, vol. 52, no. 1, p. 57–70, Aug. 2018. [Online]. Available: <https://doi.org/10.1145/3273982.3273988>
- [5] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, “Badgertrap: a tool to instrument x86-64 tlb misses,” *SIGARCH Comput. Archit. News*, vol. 42, no. 2, p. 20–23, Sep. 2014. [Online]. Available: <https://doi.org/10.1145/2669594.2669599>
- [6] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, “Thin servers with smart pipes: designing soc accelerators for memcached,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 36–47. [Online]. Available: <https://doi.org/10.1145/2485922.2485926>
- [7] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, “Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 469–480. [Online]. Available: <https://doi.org/10.1145/3079856.3080210>
- [8] K. Gosakan, J. Han, W. Kuszmaul, I. N. Mubarek, N. Mukherjee, K. Sriram, G. Tagliavini, E. West, M. A. Bender, A. Bhattacharjee, A. Conway, M. Farach-Colton, J. Gandhi, R. Johnson, S. Kannan, and D. E. Porter, “Mosaic pages: Big tlb reach with small pages,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 433–448. [Online]. Available: <https://doi.org/10.1145/3582016.3582021>
- [9] W. Jia, J. Zhang, J. Shan, and X. Ding, “Effective huge page strategies for tlb miss reduction in nested virtualization,” *IEEE Transactions on Computers*, vol. 73, no. 8, pp. 1983–1996, 2024.
- [10] A. Noohi, M. Derispour, and A. Barbalace, “RMAI: Rethinking memory for AI (inference): In-kernel remote shared memory as a software alternative to CXL,” in *Proceedings of the 5th Workshop on Machine Learning and Systems (EuroMLSys)*, 2025, pp. 122–131.
- [11] A. Noohi, M. Derispour, S. Ebrahimi, and A. Barbalace, “NTSM: OS-level memory middleware for scaling AI without boundaries,” in *Proceedings of the 27th ACM/IFIP International Middleware Conference (Middleware)*, 2026, to appear.
- [12] J. Ahn, S. Jeong, J. Lee, and J. Kim, “Direct memory translation unit for virtualization,” in *Proceedings of the 2024 Annual Conference on Computer Architecture (ACCA)*. ACM, 2024, pp. 1–12.
- [13] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, “Prefetched address translation,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1023–1036. [Online]. Available: <https://doi.org/10.1145/3352460.3358294>
- [14] A. Margaritov, D. Ustiugov, A. Shahab, and B. Grot, “Ptemagnet: fine-grained physical memory reservation for faster page walks in public clouds,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 211–223. [Online]. Available: <https://doi.org/10.1145/3445814.3446704>
- [15] ARM Ltd., *ARM Architecture Reference Manual - Large System Extensions for 128-bit Atomics (LSE128)*, ARM Ltd., 2023, aRM DDI 0487, Armv8.7-A Feature.
- [16] Intel Corporation, “Intel EPT-based sub-page write protection support,” <https://lwn.net/Articles/736322/>, 2017.
- [17] ARM Ltd., *ARM Architecture Reference Manual - Translation Hardening Extensions (THE)*, ARM Ltd., 2023, aRM DDI 0487.
- [18] A. Noohi, B. Asodeh, and A. Barbalace, “Towards a solution to the management scaling paradox in distributed LLM inference,” in *Proceedings of the 6th European Workshop on Machine Learning and Systems (EuroMLSys)*, 2026.
- [19] K. Adams and O. Agesen, “A comparison of software and hardware techniques for x86 virtualization,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, 2006, pp. 2–13.
- [20] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: Lightweight virtualization for serverless applications,” in *Proceedings of the 17th USENIX Symposium*

- on *Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2020, pp. 419–434.
- [21] ARM Ltd., *ARM Architecture Reference Manual, ARMv8, for A-profile architecture*, ARM Ltd., 2023, aRM DDI 0487.
 - [22] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, revised ed. Cambridge, MA: MIT Press, 2012, alloy model checker and specification language.
 - [23] I. Molnar, “Performance counters for linux,” <https://lwn.net/Articles/337493/>, Jun. 2009.
 - [24] D. Bruening, T. Garnett, and S. Amarasinghe, “An infrastructure for adaptive dynamic optimization,” in *Proceedings of the 1st International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2003, pp. 265–275.
 - [25] DynamoRIO Developers, “Dynamorio dynamic instrumentation tool platform,” <https://dynamorio.org/>, 2026, available at <https://github.com/dynamorio/dynamorio>.
 - [26] H. Alam, T. Zhang, M. Erez, and Y. Etsion, “Do-it-yourself virtual memory translation,” *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 457–468, Jun. 2017. [Online]. Available: <https://doi.org/10.1145/3140659.3080209>
 - [27] DynamoRIO, “DynamoRIO: Dynamic instrumentation tool platform,” <https://github.com/DynamoRIO/dynamorio>, Dec. 2023.
 - [28] ARM Ltd., “ARM neoverse n1 core technical reference manual,” https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_n1, 2019, aRM DDI 0575.
 - [29] A. Agbarya *et al.*, “Predicting execution times with partial simulations in virtual memory research,” in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020.
 - [30] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, “Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Mar. 2020.
 - [31] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, “Every walk’s a hit: Making page walks single-access cache hits,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Feb. 2022.
 - [32] S. Bergman, M. Silberstein, T. Shinagawa, P. Pietzuch, and L. Vilanova, “Translation Pass-Through for Near-Native paging performance in VMs,” in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 753–768. [Online]. Available: <https://www.usenix.org/conference/atc23/presentation/bergman>
 - [33] J. Corbet, “Sharing page tables with mshare(),” <https://lwn.net/Articles/895217/>, 2022.
 - [34] I. Yaniv and D. Tsafirir, “Hash, don’t cache (the page table),” in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2016, pp. 695–707.
 - [35] V. Karakostas and D. Skarlatos, “Memory-efficient hashed page tables,” in *Proceedings of the 29th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 421–435.
 - [36] T. W. Barr, A. L. Cox, and S. Rixner, “Translation caching: Skip, don’t walk (the page table),” in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2010, pp. 48–59.
 - [37] S. Ainsworth and T. M. Jones, “Compendia: Reducing virtual-memory costs via selective densification,” in *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM)*. ACM, 2021, pp. 40–52.
 - [38] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, “Redundant memory mappings for fast access to large memories,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2015, pp. 66–78.
 - [39] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, “Efficient virtual memory for big memory servers,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2013, pp. 237–248.
 - [40] S. Sha, Y. Zhang, Y. Luo, X. Wang, and Z. Wang, “Swift shadow paging (ssp): no write-protection but following tlb flushing,” in *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 29–42. [Online]. Available: <https://doi.org/10.1145/3453933.3454012>
 - [41] J. Gandhi, M. D. Hill, and M. M. Swift, “Agile paging: Exceeding the best of nested and shadow paging,” in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, Jun. 2016.
 - [42] Anonymous, “Learning to walk: Architecting learned virtual memory translation,” in *Proceedings of the International Symposium on Computer Architecture*, 2024.
 - [43] S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer, “Rebooting virtual memory with Midgard,” in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 512–525.
 - [44] C. H. Park, T. Heo, J. Jeong, and J. Huh, “Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations,” in *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA)*. Phoenix, AZ, USA: ACM, 2019, pp. 444–456.