Elastic Translations: Fast virtual memory with multiple translation sizes

Stratos Psomadakis* psomas@cslab.ece.ntua.gr
Chloe Alverti† xalverti@illinois.edu
Vasileios Karakostas‡ vkarakos@di.uoa.gr
Christos Katsakioris* ckatsak@cslab.ece.ntua.gr
Dimitrios Siakavaras* jimsiak@cslab.ece.ntua.gr
Konstantinos Nikas* knikas@cslab.ece.ntua.gr
Georgios Goumas* goumas@cslab.ece.ntua.gr
Nectarios Koziris* nkoziris@cslab.ece.ntua.gr

*National Technical University of Athens, Athens, Greece
†University of Urbana-Champaign, Champaign, Illinois, USA
‡University of Athens, Athens, Greece

***Abstract*—Large pages have been the de facto mitigation technique to address the translation overheads of virtual memory, with prior work mostly focusing on the large page sizes supported by the x86 architecture, i.e., 2MiB and 1GiB. ARMv8-A and RISC-V support additional intermediate translation sizes, i.e., 64KiB and 32MiB, via OS-assisted TLB coalescing, but their performance potential has largely fallen under the radar due to the limited system software support. In this paper, we propose *Elastic Translations (ET)*, a holistic memory management solution, to fully explore and exploit the aforementioned translation sizes for both native and virtualized execution. ET implements *mechanisms* that make the OS memory manager coalescing-aware, enabling the transparent and efficient use of intermediate-sized translations. ET also employs *policies* to guide translation size selection at runtime using lightweight HW-assisted TLB miss sampling. We design and implement ET for ARMv8-A in Linux and KVM. Our real-system evaluation of ET shows that ET improves the performance of memory intensive workloads by up to 39% in native execution and by 30% on average in virtualized execution.**

***Index Terms*—Operating Systems, Memory Management, Virtual Memory, Address Translation, TLB**

I. Introduction

The ever-growing memory footprints of modern workloads have been steadily increasing the pressure on the virtual memory subsystem [1]. Industry has responded by enabling the memory management unit (MMU) and the OS to support larger page sizes. Large pages store the virtual-to-physical translations higher up the page table hierarchy, increasing the Translation Lookaside Buffer (TLB) reach and shortening the page walks triggered by these misses [2]. On the downside, large pages often increase internal memory fragmentation and fault latency [3–8] and quickly become scarce as physical memory gets fragmented [3, 4, 9–11].

The x86 architecture supports two large page sizes, 2MiB and 1GiB. Industry [12–19] and academia [3, 4, 6–9, 20] have proposed a multitude of techniques, with different trade-offs, to enable and enhance OS support for these large page sizes. Table II provides an overview of these techniques and Section II discusses them in detail. Earlier designs adopted *non-transparent* interfaces [15, 17], which required large pages to be explicitly requested by userspace. State-of-practice and state-of-the-art eventually converged to *transparent* interfaces [12, 18] for large pages, tasking the OS memory manager with *selecting* which page size to use and when. The majority of these designs only support 2MiB pages transparently. 1GiB large pages exacerbate the aforementioned downsides of large pages, making their transparent support challenging [8]. However, the effectiveness of 2MiB pages diminishes as the memory footprint of applications keeps growing [8]. Additionally, when memory is fragmented, even 2MiB become scarce thus hard to allocate [9, 11].

ARMv8-A and RISC-V provide architectural support for additional *translation sizes*[1], via OS-assisted TLB coalescing, by adding a *contiguous bit* [21] in their page table entries. The OS can set the contiguous bit on *16* contiguously mapped pages in the first two levels of the page tables, effectively creating two intermediate translation sizes, 64KiB and 32MiB respectively. The contiguous bit acts as a marker for the page walker to cache these table entries as a single TLB translation (Figure 1). In this work we focus on ARMv8-A, but we also consider the RISC-V support for OS-assisted TLB coalescing [22]. As these architectures are making their way to the datacenter [23–25], where the ever-growing workload footprints stress the address translation (AT) hardware, *we argue that a larger arsenal of transparently-supported translation sizes, including intermediate-sized translations, would allow system software to better maneuver among the various trade-offs and address the limitations exhibited by the traditional 2MiB / 1GiB large page model.*

To that end, system software needs to: i) provide *mechanisms* to transparently *support* the larger number of translation sizes for both native and virtualized execution, and ii) devise *policies* to judiciously *select* between those sizes. State-of-practice and state-of-the-art have only partially addressed

This work was funded by the European Union under the Horizon Europe grant 101092850 (project AERO).

[1]We use translation size to refer to the granularity at which the TLB caches translations.

these challenges. Linux supports contiguous-bit intermediate translations via the non-transparent HugeTLB interface [15] and only for native execution. Preliminary transparent support for 64KiB intermediate-sized translations is under development [13, 14] (Section II). Policy-wise it follows a simple fallback mechanism, opting for the largest possible translation size first and falling back to smaller sizes upon failure. Our goal is to provide a holistic memory management solution that seamlessly and efficiently supports multiple translation sizes, extending beyond 2MiB, and remains resilient to external memory fragmentation. We design *Elastic Translations (ET)*, synergistic mechanisms and policies (Table I) to accomplish this goal. We make the following contributions:

1) We extend Linux and KVM [26] to support HugeTLB intermediate-sized translations for virtualized execution and comprehensively evaluate them for both native and virtualized execution (Section III). Our results showcase the performance potential of 64KiB and 32MiB translations, motivating the development of Elastic Translations.
2) We enable Linux to transparently and opportunistically manage the contiguous bit for both 64KiB and 32MiB translations for both native and virtualized execution (Section IV-A).
3) We design the *CoalaPaging* (Section IV-B) and *CoalaKhugepaged* (Section IV-C) coalescing-aware extensions to the Linux memory manager. *CoalaPaging*, based on contiguity-aware paging [27], *opportunistically* allocates suitable 4KiB and 2MiB pages *across faults* in order to lazily generate intermediate-sized contiguity that matches the coalescing size supported by the HW. *CoalaKhugepaged* extends Linux khugepaged to asynchronously create 64KiB and 32MiB translations via migrations. The two mechanisms work synergistically, i.e., when *CoalaPaging* fails to allocate all the contiguous pages required for a 64KiB or 32MiB translation at fault time, e.g., due to external fragmentation, *CoalaKhugepaged* will exploit the partial contiguity to migrate fewer pages. *CoalaPaging* and *CoalaKhugepaged* along with the transparent contiguous bit management comprise the *Elastic Translations (ET) in-kernel mechanisms*.
4) We use HW-assisted sampling to periodically record the TLB misses of workloads at runtime, and design a profiler, *Leshy*[2], which implements the *ET policies* for translation size selection. Leshy tracks the virtual address and page walk latency for each sampled miss and generates a translation-overhead heat-map of the address space. It then maps regions to translation sizes with the goal of minimizing translation costs based on the aforementioned heat-map (Section IV-D). Finally, Leshy drives the ET in-kernel mechanisms, by loading the generated translation-size profiles in the kernel. We use the ARMv8-A Statistical Profiling Extension (SPE) [21] for HW-assisted sampling and show that HW-assisted TLB miss sampling acts as a highly accurate low-overhead estimator for address translation performance.

We design and implement ET in Linux v5.18. Our evaluation on an ARMv8-A server shows that transparent 64KiB translations perform closely to 2MiB pages for memory-intensive workloads with small footprints (Section VII) for both native and virtualized execution. For larger workloads, transparent 32MiB translations improve performance by 10% on average and up to 39% over THP for native execution and by 30% and up to 150% for virtualized execution. Finally, Leshy's microarchitectural-aware policies guide ET to map the footprint of workloads by utilizing a mix of all of the available translation sizes, in order to minimize translation overhead. This improves overall performance under fragmentation by 12% on average and up to 20% over THP and state-of-the-art while consistently reducing the number of 2MiB pages used.

Component	Purpose	Main Contribution
Transparent Contig-Bit	Transparent, opportunistic creation of 64KiB and 32MiB translations	Native support for 32MiB translations Virtualization support for all sizes
CoalaPaging	Transparent opportunistic creation of contiguous 64KiB and 32MiB mappings across faults	Practical and scalable allocation policy for 32MiB mappings, with minimal impact on fault latency and memory bloat
CoalaKhuge	Asynchronous creation of 64KiB and 32MiB mappings via migrations	Enables the creation of 64KiB and 32MiB translations under memory pressure (fragmentation)
Leshy Profiler	Runtime translation size selection guidance via MMU overhead profiling	Optimal translation size selection from an extended range of sizes via lightweight HW-assisted TLB miss sampling

TABLE I: Elastic Translations Components

II. Background

As the working sets of workloads outgrew the TLB reach [1, 10, 27–30, 30–36], TLB miss rates increased significantly. Additionally, the page walks are expected to get costlier as i) paging transitions from 4 to 5-level tables [37, 38] and ii) virtualization has become ubiquitous. TLB misses in HW-assisted virtualization are notoriously costlier [27, 31, 32, 39], as they involve nested traversal of the guest and hypervisor page tables [40]. Industry's response to the problem at hand has been to steadily increase TLB capacity and add support for larger page sizes to expand the TLB reach and minimize page walk overheads. In particular, the x86 architecture supports two different large page sizes, 2MiB and 1GiB, which are implemented by storing virtual-to-physical translations higher up the radix tree.

A. OS support for Large Pages

OS large page interfaces can be broadly classified into two categories, non-transparent and transparent. Non-transparent interfaces support all available large page sizes, i.e., for x86, 2MiB and 1GiB, but require applications to explicitly request which specific size to use for which specific region of the address space. Additionally, the large pages need to be allocated in advance and are generally unavailable to the OS memory manager, e.g., for reclaim under memory pressure. This approach is adopted by Linux HugeTLB [15]. By contrast, transparent large page interfaces obviate the need

[2]Leshy is a mythological guardian spirit that can change in size.

	Transparent	Faults		Translation			Promotions	
		Supported Sizes	Policy	Supported Sizes	Policy	Virtualization Support	Supported Sizes	Policy
HugeTLB [15]	✗	4KiB, **64KiB** 2MiB, **32MiB** 1GiB	Pre-allocation Single user-defined size per VMA	4KiB, **64KiB** 2MiB, **32MiB** 1GiB	Defined by fault size	4KiB 2MiB 1GiB	✗	✗
mTHP [12, 13]	✓	4KiB, **64KiB** 2MiB	Eager allocation of the largest possible size Fallback on failure	4KiB, **64KiB** 2MiB	Defined by fault or promotion size	4KiB 2MiB 1GiB	2MiB	Migrate to 2MiB Region selection: Linear scan
FreeBSD [18]	✓	4KiB	2MiB reservation at first 4KiB fault Use reservation to serve the rest	4KiB 2MiB	Defined by fault or promotion size	4KiB 2MiB	2MiB	In-place promotion to 2MiB when every 4KiB page is faulted-in
HawkEye [4]	✓	4KiB 2MiB	Same as mTHP Asynchronous pre-zeroing	4KiB 2MiB	Defined by fault or promotion size	4KiB 2MiB	2MiB	Selectively migrate to 2MiB Region selection: Access frequency based on page table scanning
Trident [8]	✓	4KiB 2MiB 1GiB	Same as HawkEye	4KiB 2MiB 1GiB	Defined by fault or promotion size	4KiB 2MiB 1GiB	2MiB 1GiB	Migrate to largest size possible Fallback on failure Selection same as mTHP
Elastic Translations	✓	4KiB 2MiB	4KiB / 2MiB eager allocation based on VMA size **Opportunistic coalescing-aware allocations across faults**	4KiB, **64KiB** 2MiB, **32MiB**	4KiB or 2MiB based on fault or promotion size **Opportunistic promotion to 64KiB or 32MiB**	4KiB, **64KiB** 2MiB, **32MiB**	**64KiB** 2MiB, **32MiB**	Selectively migrate to 64KiB, 2MiB, 32MiB Region Selection: Size hints based on HW TLB miss sampling

TABLE II: State-of-practice and state-of-the-art large page interfaces.

for explicit opt-in by userspace applications and are tightly coupled with the core OS memory management subsystem. However, they typically only support 2MiB pages in modern OSes [12, 18, 19] (Table II) and task the OS with the responsibility of size selection.

Transparent large pages are formed either synchronously or asynchronously. The synchronous path is implemented via demand paging, when a page is first accessed (written to). When a page is accessed for the first time, a page fault is triggered, which the OS handles by allocating physical memory for the faulting page. With transparent large pages, the OS must decide i) whether a large page will be allocated to serve the fault, and ii) which large page size to use, when multiple large page sizes are supported transparently (*fault policy*). Page migrations can also be leveraged to create large pages asynchronously, off the fault path. The OS periodically scans the memory of running processes and finds discontinuous groups of pages suitable for promotion to a large page. It then allocates a large page in physical memory and migrates to it the aforementioned discontinuous pages. In that case, the OS must decide i) which virtual regions are worth promoting to larger pages and ii) what will be the target size, if multiple sizes are supported (*promotion policy*). For both cases, the allocated memory is mapped to userspace by updating the process page tables. At that point, the OS must select, from the list of available MMU-supported translation sizes, an appropriate size with which to map the allocated memory (*translation policy*). Table II shows the different policies adopted by state-of-practice and state-of-the-art.

Linux THP [12] opts for a greedy approach, that always allocates entire 2MiB pages at fault time. This has the advantage of backing the workload's address space with large pages as early as possible but performs poorly under memory pressure [3–7]. The unsolicited use of 2MiB pages leads to their sub-optimal distribution among processes and address space regions, when they run low in the system due to external fragmentation. Linux THP also includes a kernel thread, *khugepaged*, that asynchronously scans and promotes (to 2MiB) suitably-aligned 2MiB regions which are fully or partially backed by 4KiB pages, by migrating the constituent base pages to a contiguous 2MiB block of memory. State-of-the-art improves upon THP by using i) base page utilization [3, 7, 20], ii) access frequency sampling [3, 4], iii) coarse-grained MMU overhead profiling [4] and iv) user-provided profiles [5] to select which pages to promote to 2MiB, either at fault time [5] or asynchronously [3, 4, 7, 20].

Linux recently added support for multi-sized THP (mTHP) [13]. mTHP introduces a fault-time policy which enables the allocation and mapping of 64KiB blocks of memory. At fault time, Linux will attempt to allocate a 2MiB page. If the 2MiB allocation fails, it will then fall back to 64KiB instead of 4KiB. At the moment, mTHP works solely at fault-time, as there's no support for asynchronous mTHP promotions, and only for native execution. Additionally, it does not support 32MiB translations.

FreeBSD transparently supports 2MiB pages using a reservation based fault-time policy [18, 19]. It reserves a 2MiB block of memory at first fault, but faults the pages in at a 4KiB granularity, promoting them to a large page in place during the last such fault. This strategy keeps fault latency bounded but delays the formation and mapping of large pages [6]. FreeBSD does not employ any kind of asynchronous promotions via migrations.

Neither OS supports 1GiB pages transparently, as this would require the OS to track 1GiB-aligned free blocks and reserve them at fault time, potentially penalizing fault performance and increasing internal fragmentation. Moreover, 1GiB pages quickly become scarce [9–11], due to external memory fragmentation. Consequently, prior art for transparent 1GiB support [8] mainly relies on asynchronous promotions and aggressive compaction to generate the required contiguity. Neither OS or state-of-the-art design supports 32MiB pages.

B. OS-assisted TLB coalescing

TLB coalescing [41–43] is a technique that caches the translation of N contiguously mapped pages using a single TLB entry. While TLB coalescing can be implemented entirely in HW, the coalescing factor N is typically limited [44, 45] leading to diminishing results [46].

The ARMv8-A architecture supports OS-assisted TLB coalescing instead. Figure 1 shows how ARMv8-A enables a coalescing factor of $N = 16$ with OS assistance. Its page table entries include a *contiguous bit* (bit 52) which, if set by the OS in $N = 16$ consecutive page table entries, it indicates to the translation hardware that these N pages are contiguous and *properly aligned according to the coalescing factor*. In Fig. 1 the [V_A..V_{A+15}] virtual 4KiB pages are contiguously mapped to [P_A..P_{A+15}] physical 4KiB pages (1). V_A and P_A are also aligned to 64KiB (16 * 4KiB). Since every page in the 64KiB range meets the above criteria, the OS can set the contiguous bit in the 16 consecutive (yellow) PTE entries (2) mapping these 16 virtual pages to their physical frame numbers (PFNs). This allows the MMU to coalesce them into a single TLB entry and cache them as such in the TLB (3). Coalescing increases the TLB reach, effectively forming an intermediate translation size. Similarly, [V_B..V_{B+15}] contiguously mapped 2MiB pages (green) are coalesced to a 32MiB intermediate translation via setting the contiguous bit in their 16 consecutive (green) PMD entries. Finally, the contiguous bit is also supported in the PMD and PTE levels of nested page tables, which allows the MMU to coalesce contiguously mapped pages for virtualized workloads as well. RISC-V supports a similar OS-assisted TLB coalescing scheme with the Svnapot extension [22].

The performance potential of these intermediate translation sizes remains largely unexplored, as robust OS support for coalesced translations is mostly missing (Section II-A). Preliminary transparent support in Linux exists only for 64KiB translations and only for native execution via mTHP [13, 14]. The cumbersome HugeTLB interface supports both 64KiB and 32MiB translations, albeit *only for native execution* (Table II).

State-of-practice and start-of-the-art systems mainly focus on the 4KiB / 2MiB / 1GiB page sizes supported by x86 (Table II). Most designs target 2MiB pages, attempting to maximize performance by deciding when, how, and which 4KiB base pages to promote to a 2MiB large page. ARMv8 and RISC-V architectures support more translation sizes via OS-assisted TLB coalescing. Their transparent support remains limited and their performance largely unexplored.

III. Motivation

A. OS-assisted coalescing: Performance potential

To assess the performance potential of 64KiB and 32MiB translations, we use an ARMv8-A server to evaluate them, via the HugeTLB interface, versus 4KiB, 2MiB and 1GiB pages, for both native and virtualized scenarios. For virtualized execution, we extend Linux and KVM to support contiguous-bit intermediate translations. Figure 2 summarizes our results

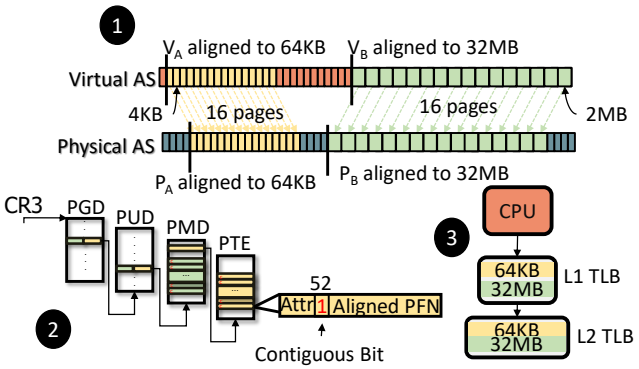


Fig. 1: ARMv8-A OS-assisted TLB coalescing

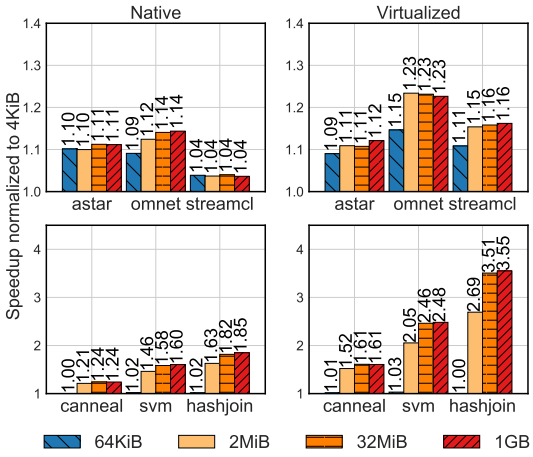


Fig. 2: Performance of HugeTLB intermediate-sized translations on a non-fragmented ARMv8-A machine

for two sets of workloads (discussed in Section VI): i) memory intensive workloads that operate on small objects (top) and ii) big-memory workloads (bottom).

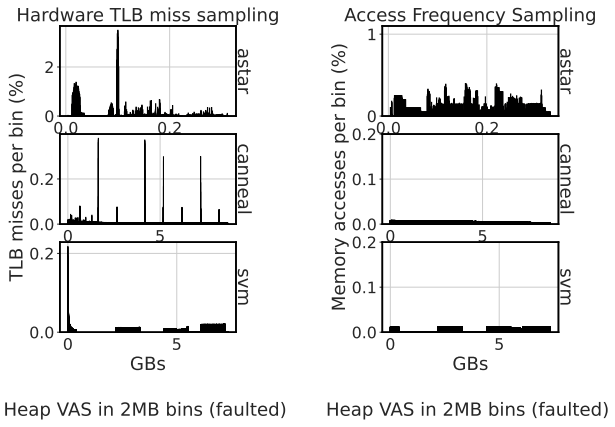
We observe that for the first set of workloads, 64KiB translations provide up to 10% (native) and 15% (virtualized) performance benefit over 4KiB pages; almost matching the performance of 2MiB pages in some cases. *64KiB translations could be leveraged to improve address translation performance while obviating the need for larger 2MiB allocations, especially under memory pressure or fragmentation [11].*

For the second set of workloads, 32MiB translations often outperform 2MiB large pages, by up to 30% in virtualized execution. Notably, they provide performance close to that of 1GiB pages. *32MiB translations can effectively mitigate translation costs that 2MiB pages are unable to cover, while relaxing the contiguity requirements of 1GiB pages, that are extremely hard to meet on a long-running system [9–11].*

64KiB and 32MiB translation sizes provide significant performance gains and can be exploited to address limitations exhibited by the 2MiB / 1GiB large page model.

B. The conundrum of translation size selection

Support for a single transparent large page size, as implemented in Linux and FreeBSD, reduces translation size



(a) (b)

Fig. 3: Narrow address space regions account for most of the translation overhead (MMU hotspots). HW-assisted sampling is able to detect them at a higher resolution than page access frequency sampling.

selection to a binary decision per 2MiB region of the virtual address space: whether to back each region by a 2MiB page or not. Intermediate-sized translations complicate size selection. We need a methodology to estimate the performance impact of different translation sizes on the different regions of a workload's address space.

We use MMU overhead as a proxy for estimating the performance impact of translation size [4, 5]. To that end, we use *fine-grained HW-based TLB miss sampling to identify the TLB-miss heavy regions of a process address space*. We leverage the ARMv8-A Statistical Profiling Extension (SPE) [21] to sample the TLB misses of workloads and track the virtual address and page walk latency for each sampled miss.

Figure 3a shows the distribution of misses for three MMU-intensive workloads, divided in 2MiB bins. There are wide regions which exhibit minimal overheads, and narrow spikes that are responsible for the majority of the TLB misses. Notably, a single 2MiB region is responsible for ∼*5%* of the total TLB misses for astar. Such *fine-grained translation-overhead information can be leveraged to optimally assign different translation sizes to different virtual regions based on the MMU pressure they generate.*

Prior art relies on page-based access frequency sampling to estimate MMU overhead [3, 4, 8] and guide 2MiB promotions [3, 4]. Figure 3b presents the access frequency heatmaps for the same workloads, generated by periodically sampling the access bit of each populated page of the address space [4]. HW-assisted sampling is able to identify MMU hotspots at a higher resolution. Not every frequently accessed page contributes equally to translation overhead. We elaborate on this in Section VII.

The address space of memory intensive workloads exhibits translation overhead hotspots. HW-based sampling manages to accurately detect them, unlocking the potential for informed translation size selection.

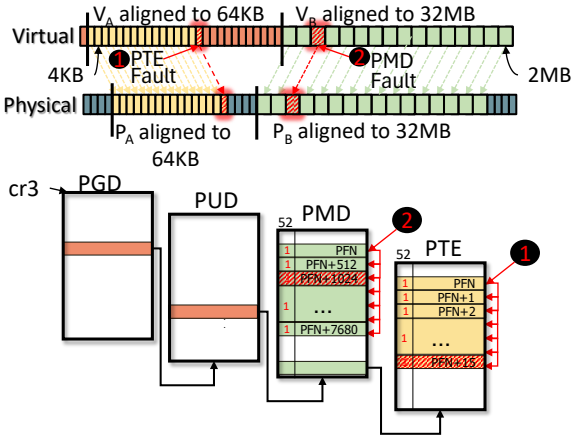


Fig. 4: ET contiguous bit management during a PMD (2MiB) (2) and PTE (4KiB) (1) fault

IV. Elastic Translations

We design and implement *Elastic Translations (ET)*, synergistic mechanisms and policies (Table I) that i) enable the OS to transparently generate and manage intermediate-sized translations in native and virtualized environments (*Transparent Contig-bit*, *CoalaPaging*, *CoalaKhugepaged*) and ii) optimize translation size selection from the now extended pool of available translation sizes (*Leshy*).

A. Transparent contiguous bit management

In order to transparently support and opportunistically create intermediate sized translations, ET needs to i) detect suitably-aligned contiguously-mapped pages, and ii) transparently promote them to intermediate-sized translations by setting the contiguous bit in each page table entry. Coalesced translations must also be demoted when their constituent pages are no longer contiguous. Figure 4 depicts our mechanism. Whenever a page table entry is created or modified, ET checks the N (16 in our case) page table entries which belong to the same coalescing range. That is the 16 neighbouring entries in a 64KiB range for 4KiB page entries (PTEs) or a 32MiB range for 2MiB page entries (PMDs), starting from the first 64KiB- or 32MiB-aligned entry. If every entry in this range is: i) suitably aligned, both physically and virtually, i.e., for a 4KiB PTE, mapping the virtual page number (VPN) to a physical frame number (PFN), $[PFN mod 16 == VPN mod 16]$, ii) physically contiguous with regard to the other entries in the range, i.e., for a 4KiB PTE $[PFN_{n+1} == PFN_n + 1]$, and iii) has compatible page flags and access permissions as the rest of the range entries, *we promote the range, by setting the contiguous bit in each PTE or PMD in the range*. Reversely, when a page entry modification invalidates any of the above, we demote the range by clearing the contiguous bit accordingly and flushing the corresponding TLB entry. When the range is not fully faulted in, ET falls back to the default Linux path for setting the PTE or PMD respectively. We quantify the latency overhead of our mechanism in Section VII.

Whenever the size of a translation entry changes, ARMv8-A mandates invalidating the entry and flushing it from the TLB.

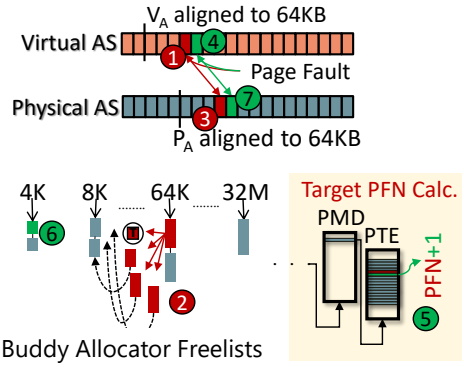


Fig. 5: CoalaPaging target PFN calculation and allocation

This rule is called *break-before-make* in the architecture reference manual [21]. This is always required when demoting an intermediate translation, since leaving stale contiguous entries in the TLBs can enable otherwise invalid memory accesses. However, transparently creating an intermediate translation by setting the contiguous bit does not invalidate its constituent page translations that may be still cached. They still map to the same memory locations with the same permissions. This obviates the need for a TLB flush, thus, as an optimization, *we opt for lazily flushing newly-promoted page table entries.*

Virtualization support. ET also supports virtualized execution under KVM, transparently managing the contiguous bit in the nested page tables. HW-assisted virtualization utilizes nested paging for memory virtualization. The guest OS page tables translate guest virtual addresses (GVA) to guest physical addresses (GPA). The nested page tables, managed by KVM, translate these GPAs to actual host physical addresses (HPAs). The TLB then caches GVA to HPA translations [32, 40].

For ET, the contiguous bit in the guest page tables is managed by the guest OS as described in the previous paragraphs. The contiguous bit in the nested page tables is managed during nested faults by KVM. Allocations triggered by nested faults will eventually need to update the host page tables of the virtual machine monitor (VMM). ET already hooks this path, as the VMM is a regular host process, and will thus detect and promote coalesce-able ranges in the VMM host page tables. We extend KVM so that these promotions are reflected to the nested page tables of the VM, by setting the contiguous bit of the corresponding shadow page table entries (SPTEs). Similarly, whenever the host demotes an intermediate translation, e.g., due to unmapping or migration, KVM is notified and demotes the corresponding SPTEs. By promoting intermediate translations in both host and guest, ET allows the caching of coalesced 2D GVA to HPA translations in the TLB.

B. Coalescing-aware Paging

To generate the inter-page contiguity required for intermediate-sized translations, we design a coalescing-aware allocation policy, *CoalaPaging*, based on contiguity-aware paging (CAPaging) [27]. Our goal is to maximize the formation of suitably aligned and contiguous ranges of pages, i.e., 64KiB ranges for PTEs and 32MiB for PMDs. The core idea is that we mirror the TLB coalescing logic in the allocation path. On each fault, we attempt to either create or extend a contiguous and aligned 64KiB or 32MiB range of pages, by scanning the page tables and selecting a suitable target page. Figure 5 depicts the coalescing-aware allocation process.

First fault. When handling a fault, CoalaPaging scans the page table entries of the 64KiB- or 32MiB-aligned range which the faulting address belongs to. For the first fault within such a range, we attempt to find a suitably sized and aligned free block. We then find and allocate the page of the block whose PFN alignment with regard to the coalescing factor matches the alignment of the faulting address.

For a 64KiB range of 16 4KiB pages, CoalaPaging finds a free 64KiB block, by searching the order-4 (64KiB) and higher free-lists of the buddy allocator and allocates the 4KiB page whose [`PFN mod 16 == VPN mod 16`], where PFN is the physical frame number of the page and VPN is the virtual page number of the faulting address. However, CoalaPaging *neither allocates nor reserves the block*. Once the target page is allocated, the remaining pages are added back to the allocator free-lists. To maximize the time window during which these pages remain available, we append them to the tail of their respective buddy lists. Figure 5's steps 1-3 depict the allocation process for the first fault in a coalescing range. CoalaPaging operates in a similar way for THP faults, but now has to find 32MiB (order-13) free blocks. Linux only tracks by default contiguous blocks up to 4MiB (order-10). We therefore configure it to track up to 32MiB blocks in its allocator free lists.

Subsequent faults. To identify the target physical page for subsequent faults, CoalaPaging scans the page table entries of the 64KiB or 32MiB range that the faulting address belongs to, searching for a populated page table entry. When such a previously faulted PFN is found, CoalaPaging uses it as an *anchor* to calculate the allocation target. Specifically, CoalaPaging first checks that the anchor PFN is properly aligned (as described in the preceding paragraph), and if not, it aborts the CoalaPaging allocation. We then align the anchor PFN to 64KiB or 32MiB, depending on fault type, and add the relative index of the faulting VA within the 64KiB or 32MiB range.

CoalaPaging extracts all the necessary information for the target PFN calculation from the page table state, eliminating any additional metadata requirement, in contrast to e.g., CAPaging. Figure 5's steps 4-6 depict the process. In Section VII we quantify the latency overhead of page table scanning.

Multi-programmed Execution. In a multi-programmed scenario, CoalaPaging coordinates fault-time allocations of different programs, directing them to different parts of the physical address space. As described in IV-B, the first CoalaPaging fault in a 32MiB range will allocate a single 2MiB page from a free 32MiB block and release the rest back to the buddy lists. Subsequent faults in this 32MiB VA range will use the page table to compute the anchor PFN and request the correct physical page based on the faulting VPN. Concurrent allocation requests from other programs will follow the same steps, either allocating a 2MiB page from a new 32MiB block or attempting to allocate one from the previously split 32MiB

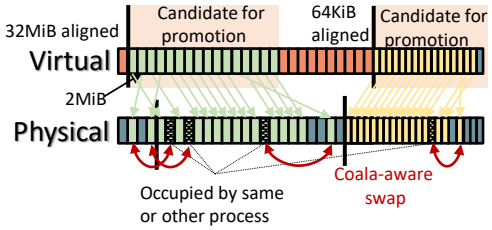


Fig. 6: Coalescing-aware khugepaged

block, based on information found in the page table. As a result, different programs under ET do not compete for the same buddy blocks and are all able to create 32MiB mappings across faults, on a best-effort basis, as long as there is 32MiB-contiguity available in the system. The same applies for 4KiB faults and 64KiB translations. We evaluate ET effectiveness in multi-programmed scenarios in Section VII-A.

Virtualization support. CoalaPaging works without any modifications in virtualized execution. It is independently employed by the guest and the host – generating contiguity independently in the two dimensions. As guest faults trigger nested faults on the host, this simple scheme is sufficient to generate 2D contiguity, similarly to THP [27].

C. Coalescing-aware promotions

ET also supports asynchronous promotions via CoalaKhugepaged. For 2MiB pages, Linux khugepaged periodically selects an active process, in a round-robin fashion, and performs a linear scan of its address space, promoting any suitable properly-aligned region, not yet backed by a large page, to 2MiB. In order to promote a region to 2MiB, khugepaged allocates a new 2MiB page and copies the constituent base 4KiB pages to the allocated large page. Khugepaged also includes knobs to control the allocation aggressiveness and CPU overhead of scanning and migrations, allowing the user to control how many pages to scan or collapse per second and including a back-off policy when large page allocations fail due to external fragmentation. CoalaKhugepaged augments khugepaged for optimized coalescing-aware promotions to intermediate-sized translations (64KiB and 32MiB). CoalaKhugepaged works synergistically with CoalaPaging by taking advantage of partially contiguous groups of pages to reduce the number of migrations required for promotion. When CoalaPaging is able to create only a partially contiguous range at fault time, CoalaKhugepaged will attempt to utilize in-place promotions, migrating only misplaced pages to their target PFN if possible (Figure 6). If any of the target PFN cannot be replaced (e.g., due to unmovable pages [9]), CoalaKhugepaged will fallback to the default khugepaged behavior, migrating the whole range to freshly allocated memory. To that end, we also tune the Linux compaction logic to work for intermediate-sized blocks (i.e., 32MiB). Asynchronous promotions to intermediate-sized translations, apart from improving resilience to external fragmentation, enable ET to take advantage of informed runtime promotion policies as discussed in IV-D.

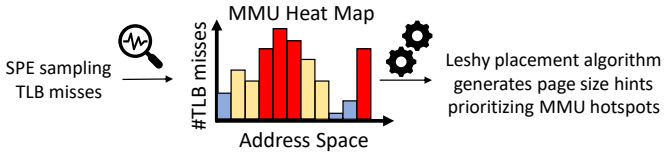


Fig. 7: Leshy tracks the MMU pressure per virtual page and uses this translation overhead heatmap of the address space to calculate translation size hints.

Fairness. CoalaKhugepaged prioritizes ET-enabled processes, instead of iterating over all running processes in the system (same as [4]), and substitutes linear address-space scan with priority-address-range scanning, guided by TLB miss profiling (as described in IV-D). When multiple ET-enabled processes run in the system, CoalaKhugepaged will distribute contiguity among them in a round-robin manner, similar to [4].

D. Translation size selection policies

With the ET in-kernel mechanisms in-place, we now devise selection policies to harness the performance potential of the expanded range of supported translation sizes.

Fault-time allocation. At fault time, CoalaPaging uses the size of the faulting virtual memory area (VMA) as an estimator to guide translation size selection. Specifically, when 64KiB translations are able to cover the entire faulting VMA while staying within TLB reach, CoalaPaging attempts to opportunistically create 64KiB translations via base 4KiB fault-time allocations. For larger VMAs, CoalaPaging aims for opportunistic 32MiB translations via THP faults (Section IV-B). Similarly to mTHP [13], we employ an incremental fallback policy to smaller translation sizes in case of allocation failure.

Asynchronous promotions. For asynchronous promotions, in contrast to khugepaged and similarly to prior art [3, 4], we attempt to estimate which memory regions to scan, migrate and promote to larger translations. We must also decide which translation size to use for each region. To that end, we design *Leshy*, a profiler that leverages ARMv8-A Statistical Profiling Extensions (SPE) to sample the TLB misses of running workloads. We decide to sample TLB misses instead of the per-page access frequency, as prior work does [3, 4], based on our analysis in Section III. In Section VII we quantify the accuracy and overhead of both methods. Leshy analyzes the TLB misses and generates a translation overhead heat-map of the address space, aggregating the misses per virtual page (Figure 7). Leshy then sorts regions by MMU hotness and attempts to optimally map the working set to translation sizes based on translation overhead.

We use Leshy to periodically profile workloads *at runtime* and compute optimal translation size hints for each region of the process address space *online*. We then load the computed hints into the kernel at runtime via an extended *madvise()* interface and use them to drive the in-kernel ET mechanisms. The hints are sorted by MMU overhead and are loaded and stored in the kernel in that order. As discussed in Section IV-C, for asynchronous promotions, CoalaKhugepaged will traverse the hints in sorted order, prioritizing promotions for the MMU

hotspots of the address space. When offline profiling is an option [5], the hints can be computed and loaded in advance, enabling CoalaPaging to utilize them at fault time, improving upon the greedy fault-time allocation policy. We retain the fault-time fallback policy in case of allocation failure.

Optimal size selection. In order to optimize translation size selection and generate translation size hints, Leshy needs to find a non-overlapping mapping of address space regions to translation sizes. This mapping should contain a limited number of translations, N and cover a target percentage of the sampled TLB misses. We use the TLB size (entries) for N and set the coverage target to 99.99% of the total sampled TLB misses. Additionally, the mapping should use the least contiguity-taxing combination of translation sizes that satisfy the above constraints.

To that end, we aggregate the sampled addresses in bins of different sizes and for each bin i of $size_i$ we calculate the total sampled TLB misses, $misses_i$, for all the addresses, $addresses_i$ belonging to it. We then formulate the optimization problem as follows:

$$\min \quad \sum_i size_i x_i \quad \text{s.t.} \quad \sum_i misses_i x_i \geq target \\ \sum_i x_i \leq N,\ x_i \in \{0,1\} \\ \bigcap_i addresses_i = \emptyset \tag{1}$$

Algorithm 1: Calculating size hints from TLB misses

```
TlbMisses = Sample(Workload, Duration)

for each Size
    AggregateMisses(Align(VA, Size), Bin[Size])
        for VA in TlbMisses
    Sort(Bin[Size])

for each Size:
    Entries = Take Entry from Bin[Size]
        while CoveredMisses(Entries) < Target
    if CoveredMisses(Entries) >= Target
        Selection = Entries
        InitialSize = Size

while CoveredMisses(Selection) >= Target
    for each Size < InitialSize
        Selection = Substitute(Selection,
            InitialSize, Size)

Sort(Selection)
Return Selection
```

To compute the translation size hints, we first sort the bins based on the total number of misses caused by each aggregated address (*entry*). We then follow a best-fit approach, whereby we first calculate the minimum translation size (*initial size*) that is able to cover the target TLB misses with N or less entries. Starting from this initial selection of M entries, we retain the $M-1$ entries with the most TLB misses and recursively attempt to substitute the discarded Mth entry with a sub-selection of smaller-sized entries that are able to match the target misses while not exceeding the configured TLB capacity N.

V. Discussion

A. Memory Management

Allocation policies. Another approach to generate the contiguity required for intermediate-sized translations is to eagerly allocate 64KiB (order-4) and 32MiB (order-13) pages during faults. As discussed in Section II, Linux recently added support for sub-2MiB faults [13] (mTHP). We evaluate mTHP in Section VII and find that, for 64KiB faults, the fault latency remains bounded. However, we also show that extending this design to 32MiB faults results in inflated fault latency. By contrast, CoalaPaging can seamlessly and efficiently support both 64KiB and 32MiB translations. We consider integrating mTHP to ET, as an alternative mechanism for generating *sub-2MiB* contiguity at fault time, for future work. Async pre-zeroing [4, 5, 8] can also be used to reduce fault latency for larger fault-time allocations; however, it comes with non-negligible CPU overhead. CoalaPaging can be nonetheless seamlessly integrated and take advantage of async pre-zeroing for faster 2MiB faults.

Reservation-based schemes [6, 18, 19, 47] could be used instead of eager allocations in order to reserve larger blocks of memory at fault-time without penalizing fault latency. Similarly to opportunistic designs [27], such as CoalaPaging, reservations trade-off the reduced fault latency with delayed creation of larger translations [6]. Compared to opportunistic designs, reservations opt for stronger guarantees for across-fault contiguity, which however incurs book-keeping overhead and increases memory bloat [6].

Transparent 1GiB support. ET focuses on the transparent support for intermediate translation sizes supported by OS-assisted TLB coalescing. We consider extending i) CoalaPaging to opportunistically create 1GiB mappings and ii) Leshy to take into account 1GiB translations and emit 1GiB hints as future work. That said, as we show in Section III, for a range of applications the ET-enabled 32MiB translations are sufficient to alleviate MMU overheads without resorting to the harder to allocate and manage 1GiB pages.

Demotions. ET does not currently support the dynamic demotion of translations to smaller sizes, which can lead to sub-optimal distribution of available contiguity. We consider extending Leshy to detect cold parts of the address space and generate demotion hints by periodically sampling the memory access frequency of previously promoted regions. For OS-initiated demotions (e.g., in the case of page migrations), ET will automatically demote the 32MiB translation (Section IV-A), if it exists, as well.

Hints in virtualized execution. Using TLB miss sampling to generate hints for virtualized workloads is challenging [48], as sampled VAs are not readily usable by the hypervisor. We use it only in the guest and fallback to access bit tracking in the host as a proxy for the MMU overhead of the VM ([3, 4]). We consider exploring a paravirtualized interface [49] to allow virtualized workloads to take full advantage of the Leshy-generated hints as future work.

Workload	Description	Footprint
astar	A* pathfinding algorithm [51]	400MiB
omnetpp	Network Simulator [51]	150MiB
streamcluster	Online Clustering [52]	100MiB
BFS	GAPBS [53] BFS on the Friendster [54] graph	88GiB
canneal	Chip Routing [52]	14GiB
XSBench	Monte Carlo Cross Section Lookup [55]	122GiB
SVM	Support Vector Machine library [56, 57]	39GiB
BTree	Lookups in a BTree [8]	33 GiB
hashjoin	Hashjoin microbenchmark	70GiB
GUPS	HPCC random updates benchmark [58]	32 GiB

TABLE III: Evaluation Workloads

B. Architectural considerations

TLB micro-architecture. The micro-architecture of the N1 ARMv8-A core features unified TLBs with regard to translation size. Every TLB entry can be use to store translations of any of the supported sizes. For split TLBs, translation size selection will need to take the different capacities into consideration [50]. Moreover, as discussed in Section IV, demoting coalesced translations requires invalidating and flushing the constituent pages. ARMv8-A supports HW-based invalidations for maintaining TLB coherence. Additionally, ARMv8-A has recently added support for range-based HW TLB flushes and invalidations, which should further accelerate TLB coherence. This is in contrast to x86, which handles TLB invalidations in SW with costly interprocessor interrupts. For the latter case, we should also factor in the cost and frequency of TLB shootdowns, potentially forgoing promotions if their benefit would not amortize the aforementioned costs [5].

Portability. While we focus on ARMv8-A, ET can be extended to different architectures and translation sizes. The Svnapot extension [22] adds support for OS-assisted TLB coalescing to RISC-V. RISC-V allocates more bits in the page table entries to encode the coalescing factor, hence extending the range of the supported translation sizes. We plan to port our prototype to RISC-V and evaluate ET with Svnapot.

Access and Dirty Bits. ARMv8-A supports HW-based tracking for page accesses (*access (A) bit*) and modifications (*dirty (D) bit*). When a page of an intermediate translation is accessed or modified, the architecture allows the MMU to set the AD bit of any of the constituent pages of the intermediate translation. This has the side-effect that the OS must now check the AD bit status of all the constituent pages of an intermediate translation in order to determine the AD status of a constituent page. For anonymous mappings, that we currently target with our design, this can affect the performance anonymous memory reclaim (swapping). We consider studying this effect for future work.

VI. Methodology

Experimental Setup. We implement ET for Linux v5.18 and evaluate it on Ubuntu 22.04 for both native and virtualized execution. For virtualized execution, we use KVM and Qemu v7. For the evaluation, we use an Ampere Altra server [59, 60], with 2 nodes of 80 ARMv8-2A+ Neoverse N1 cores [61], each with 256GiB of memory. The MMU includes separate data and instruction fully-associative L1 TLBs of 48 entries each, and a unified 5-way set-associative L2 TLB of 1280 entries of any size. L1 misses cost ∼3 cycles and L2 misses over 15 cycles. To minimize jitter, we use a single NUMA node, pin each thread on a single core and set the core frequency to 2.7GHz. We also replace GNU libc's malloc [62] with gperftools tcmalloc [63] similar to [27, 30, 35, 39].

Performance Metrics. We use end-to-end execution cycles and L2 TLB misses, reported by the HW performance counters of ARMv8 PMUv3 [21] as the main evaluation metrics. To quantify the ET effect on fault latency, we use the Linux tracing subsystem to instrument the kernel fault handling path.

Fragmentation. For the fragmentation scenarios, we allocate all of the node memory and then release small chunks at the start of each 2MiB page, similarly to [3–6]. For each workload, we release memory until i) the free memory in the system equals the footprint of the workload and ii) the Free Memory Fragmentation Index (FMFI) [64] for 2MiB (order-9) pages equals a defined threshold, reported as a percentage *X%*. Without asynchronous promotions, the workload would run with *X%* of its footprint backed by 2MiB pages.

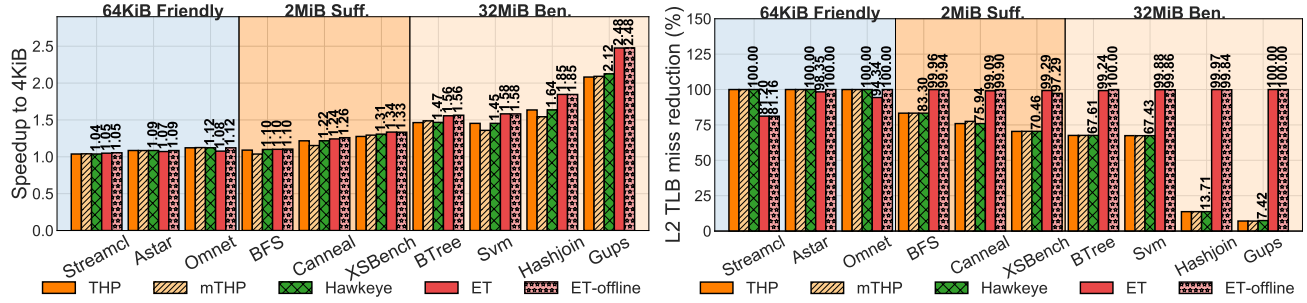
Workloads. We use applications that exhibit varying TLB sensitivity to evaluate the behavior and effectiveness of ET. We include workloads with large footprints and varying degrees of access irregularity. These workloads are typically backed by 2MiB pages and some can push 2MiB pages to their limit in terms of effectiveness. We also evaluate workloads with smaller footprints but highly irregular access patterns. Table III provides a description of the evaluation workloads.

Evaluation scenarios. We use the 4KiB performance of Linux as the baseline. We compare ET with Linux THP and mTHP. As discussed in Section II (Table II), mTHP enables 64KiB translations through faults, as a fallback to 2MiB allocations. We also port HawkEye [4] to Linux v5.18 and ARMv8-A and compare it with ET. For mTHP, we use Linux v6.8 and we also report the 4KiB performance of Linux v6.8 for reference. To understand the effect of runtime sampling and hint generation versus offline profiling, we use Leshy to sample workloads and generate translation-size profiles in advance, which we then load into the kernel when the workload is spawned (*ET-offline*). Finally, we compare the ET fault latency to 4KiB, 64KiB, 2MiB and 32MiB synchronous faults. As 32MiB faults are not transparently supported by (m)THP, we use a kernel built with a 16KiB base page size (granule) [21], which increases the THP large page size to 32MiB.

VII. Evaluation

A. Native Execution

We first run the workloads natively on a freshly booted machine. Figure 8 summarizes our results for (a) execution time speedup and (b) TLB miss reduction. Figure 9 shows the corresponding distribution of translation sizes for each method. We present a single bar for both THP and mTHP as their distributions are almost identical. Since the memory is not fragmented, asynchronous promotions are rare, which allows us to isolate the effect of fault-time *allocation policies*. ET uses the size of the faulting VMA to guide translation size selection, while ET-offline uses the Leshy generated hints



(a) Speedup

(b) TLB miss reduction

Fig. 8: Elastic Translations (ET) performance on a non-fragmented node for native execution

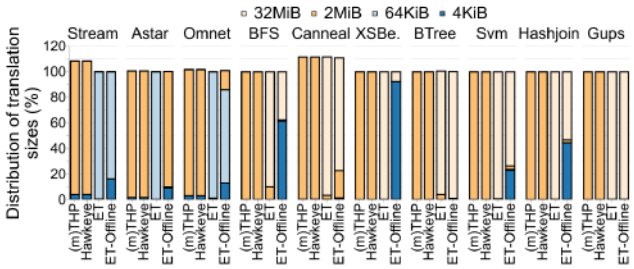


Fig. 9: Distribution of translation sizes

(Section IV-D). Based on the performance and translation size distribution, we discern three groups of workloads:

64KiB-friendly workloads. For workloads with small footprints, i.e. Astar, Omnetpp, Streamcluster, ET uses CoalaPaging to opportunistically map them with 64KiB translations, via coalescing-aware 4KiB allocations at fault time (Figure 9). This significantly reduces TLB misses (Figure 8b) and the overall performance is close to (m)THP (Figure 8a). The results are in line with our motivational analysis (Section III) and show that CoalaPaging is able to successfully generate 64KiB translations across 4KiB faults. mTHP does not leverage 64KiB faults as 2MiB allocations always succeed.

2MiB-sufficient workloads. For Canneal, XSBench and BFS, ET utilizes coalescing-aware 2MiB faults to eventually map their footprint with 32MiB translations (Figure 9). This results in a 16-30% reduction in TLB misses compared to (m)THP, but translates to only minor execution speedups up to 3-4%. 2MiB translations are sufficient for these workloads.

32MiB-beneficiary workloads. For the highly irregular workloads, BTree, SVM, Hashjoin and Gups, ET eliminates TLB misses, using 32MiB translations to cover 97-99% of their footprint. This boosts performance by 19% on average and up to 39% versus THP. These results match our motivational analysis (Section III) and demonstrate that ET effectively and transparently supports all translations sizes. No other design supports 32MiB translations.

For the larger workloads, mTHP appears to perform slightly worse than THP. This is only due to a slightly worse baseline performance (4KiB) of Linux v6.8 (2-3%) and not due to reduced address translation performance (Figure8b). HawkEye has identical performance to (m)THP as it always uses 2MiB faults [5] and its async prezeroing has negligible impact.

MMU hotspots. To further study the performance potential of multiple translation sizes, we run Leshy *offline* for all workloads and load the computed translation size hints into the kernel when each workload is spawned. This way CoalaPaging allocations are no longer eager; they are instead guided (Section IV). Figure 9 shows that TLB misses are frequently localized to specific address space regions (Section III). ET-Offline is able to detect these hotspots and map only them with larger translation sizes. For example, for XSBench, Svm, BFS and Hashjoin, it uses 4KiB pages for 93%, 34%, 64% and 45% of their address space, mapping the rest with a combination of 2MiB and 32MiB translations. This significantly reduces the usage of larger translations while sustaining performance (Figures 8a, 8b). For Canneal and BTree, Leshy uses a combination of 2MiB and 32MiB translations for their entire footprint. For Omnetpp and Astar, it uses a combination of 64KiB and 2MiB translation sizes to minimize MMU overheads, while for Streamcluster it exclusively uses 64KiB. For Astar and Omnetpp, Leshy overestimates the importance of some TLB misses, which results in ET-Offline using larger translations compared to ET, with only minor improvements in TLB miss reduction and overall performance. These results lay the ground for the online guided asynchronous promotions discussed later.

Takeaway 1: *One size does not fit all*. ET successfully generates 64KiB and 32MiB translations across faults, relaxing the need for 2MiB pages and improving performance by up to 39% over THP.

B. Virtualized Execution

We also evaluate ET in virtualized execution. Figure 10 presents the results without fragmentation. We omit the results for mTHP as it doesn't support virtualized execution. The costly nested page walks magnify AT overheads, necessitating larger translation sizes. Omnetpp, which was covered by 64KiB translations in native execution, requires some 2MiB pages to sustain performance in virtualized environment. Similarly, 2MiB pages are no longer sufficient for Canneal and XSBench, which now require 32MiB translations. Despite its opportunistic nature (Section IV), CoalaPaging manages to effectively generate contiguous 64KiB and 32MiB translations

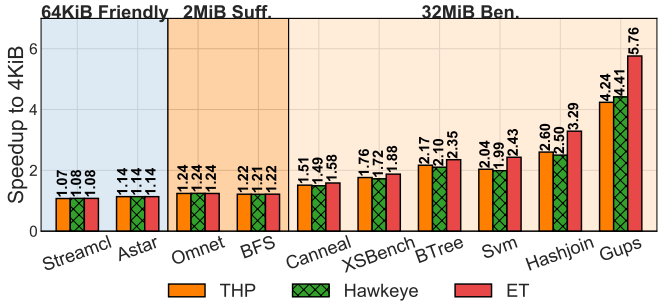


Fig. 10: ET performance in virtualized execution

in both guest and host. This translates to significant speedups for big-memory workloads, 30% on average and up to 150% over THP. HawkEye performs slightly worse than THP as there is no fragmentation, thus both systems eagerly allocate 2MiB pages at fault time [5], while HawkEye scanning and pre-zeroing are costlier in a 2D set-up.

Takeaway 2: ET successfully enables intermediate translation sizes in virtualized execution. The costlier pagewalks magnify ET benefits, speeding-up execution by 30% on average and up to 150% over THP for large workloads.

C. External fragmentation

Figure 11 presents the results for two fragmentation scenarios, 50% and 99% (Section VI) for native execution. For smaller workloads it was challenging to consistently generate fragmentation, due to their small footprints, so we omit their results. As the fragmentation increases, all methods increasingly rely on asynchronous migrations to generate large translations. This allows us to evaluate the effect of *asynchronous promotion policies*. ET asynchronous promotions are guided by Leshy translation size hints, which are generated *online* by periodically sampling the TLB misses of each running workload. We also show results for ET-offline, where ET asynchronous promotions are guided by optimal hints precalculated by Leshy *offline*. The fault allocation policy remains unchanged in both cases, unlike the previous section where offline hints were also used by ET during faults. As expected, increased fragmentation negatively impacts performance for all methods. However, ET outperforms or at least matches state-of-practice and state-of-the-art.

2MiB-Sufficient. For Canneal, all methods perform almost equally, as the workload runs long enough for all methods to promote the entire address space to large pages. For BFS, ET improves performance by 6% over both THP and HawkEye and for XSBench by 20% and 4% respectively. The reason is two-fold; a) Leshy successfully identifies at runtime the MMU hotspots and prioritizes their promotion and b) ET leverages 64KiB and 32MiB translations, which albeit unnecessary without fragmentation, are beneficial for the workloads when memory is fragmented. Consequently, ET manages to sustain higher performance while reducing large page usage by 50% on average compared to THP. HawkEye effectively detects

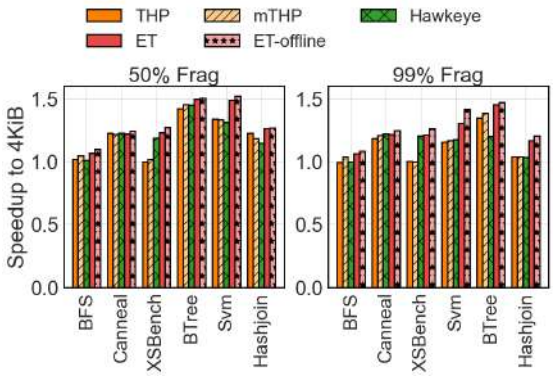


Fig. 11: ET native performance under fragmentation

the MMU hotspots for XSBench, but ET's higher resolution achieves slightly better performance while reducing 2MiB usage by 20%. mTHP falls back to 64KiB translations at fault time, which improves performance by ∼2% for some workloads. However, mTHP-khugepaged always promotes the formed 64KiB translations to 2MiB, without considering performance impact. These results underline that, while 64KiB contiguity can be utilized when 2MiB pages become scarce due to external fragmentation, efficiently taking advantage of them requires informed promotion policies.

32MiB-beneficiary. For BTree, SVM and Hashjoin, ET outperforms both state-of-practice and state-of-the-art; speeding-up performance by 12% over (m)THP and 17% over HawkEye on average when memory is 99% fragmented. Hashjoin and SVM have MMU hotspots at the tail of their address spaces, rendering THP linear promotion scanning ineffective. By contrast, Leshy successfully detects these hotspots at runtime and prioritizes their promotion to 32MiB, improving performance by 16% and 14%. HawkEye is unable to detect these hotspots as accurately and only promotes few regions to 2MiB. At 99% fragmentation, Hawkeye performs worse than (m)THP for the BTree workload, likely due to contention in its internal data structures, identified also by related work [8].

Online vs Offline. Figure 11 reveals that HW-assisted TLB miss sampling is able to guide asynchronous promotions at runtime accurately. In most cases, online profiling and hint generation (*ET*) is able to achieve comparable results to hints computed offline (*ET-offline*), resulting in similar translation size distributions. For SVM, the gap between offline and online performance under 99% fragmentation is attributed to the differences between a pro-active (offline) and a re-active (online) method. SVM exhibits a long initialization period with negligible MMU overheads and abruptly switches to the MMU intensive part of its execution. With pre-computed hints, ET-offline is able to start the migrations earlier and by the time SVM enters its second compute-intensive phase, a large part of its address space is already optimally mapped. ET's online profiling, on the other hand, triggers promotions only after SVM starts experiencing MMU overheads, which the profiler detects at runtime. Although longer running workloads might be able to amortize this cost, this spool-up effect also underlines the usefulness of offline profiling when possible.

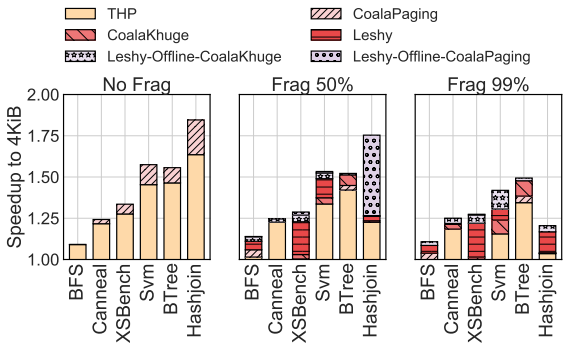


Fig. 12: Performance breakdown of ET components

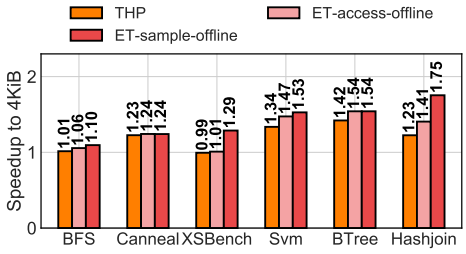


Fig. 13: TLB miss sampling vs access-bit monitoring accuracy

Takeaway 3: ET accurately detects MMU hotspots at runtime and prioritizes their optimal mapping to an educated mix of translation sizes, when running under fragmentation. This improves performance by 12% on average and up to 20% while reducing 2MiB occupancy by 30% on average.

D. Performance analysis

Figure 12 presents a breakdown of the impact of the various ET components (Table I) for native execution and increasing fragmentation levels. We stack the speedup provided by each component, relative to 4KiB, on top of each other, starting with vanilla THP. ET comprises a) CoalaPaging that transparently generates 64KiB and 32MiB translations across faults, b) CoalaKhugepaged that asynchronously promotes regions to 32MiB translations and c) Leshy that detects MMU hotspots at runtime via TLB miss sampling, computes translation size hints and drives CoalaKhugepaged promotions. We also present the benefit provided by pre-computed (offline) Leshy profiles, when they drive a) CoalaKhugepaged promotions from the beginning of a workload's execution and b) CoalaPaging fault-time allocations.

The impact of each component depends on fragmentation level and workload behavior. Under low fragmentation pressure, ET benefits are mostly driven by CoalaPaging. As fragmentation increases, CoalaPaging impact diminishes, with the exception of BFS. BFS exhibits a small MMU-intensive region at the beginning of its address space. CoalaPaging is able to map it to 64KiB translations and alleviate translation overheads, even when 2MiB pages are scarce. For the rest, CoalaKhugepaged and Leshy dominate performance gains as fragmentation increases. For BTree, where the distribution

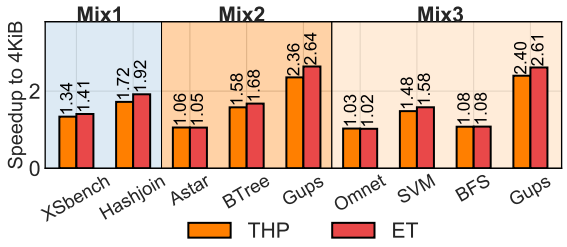


Fig. 14: ET performance for multi-workload mixes

of TLB misses is relatively uniform throughout its address space, CoalaKhugepaged's aggressive promotions to 32MiB translations, via linear scanning the workload's address space, are sufficient to alleviate the AT overhead. By contrast, TLB misses for XSBench, Hashjoin and SVM are clustered in small regions at the tail of their address space. For these workloads, ET performance gains stem from Leshy, as it is able to accurately detect these TLB-heavy clusters at runtime and guide CoalaKhugepaged promotions. Pre-computed (offline) Leshy profiles are mainly beneficial to Hashjoin and SVM, albeit for slightly different reasons. Hashjoin benefits from informed CoalaPaging faults when fragmentation is mild as, due to its short runtime, CoalaKhugepaged is unable to cover the MMU-intensive parts of its footprint in time. SVM on the other hand benefits from the fact that with pre-computed translation hints, CoalaKhugepaged asynchronous promotions are able to begin early, before the workload enters its MMU-intensive phase.

TLB miss vs access-bit sampling. We also evaluate the use of access-bit sampling to generate offline translation size hints via Leshy. We use hints to guide both CoalaPaging fault-time allocations and CoalaKhugepaged asynchronous promotions (similarly to ET-offline in Figures 12 and 8). Figure 13 shows that for 50% fragmentation translation size hints generated by Leshy based on sampled TLB misses exhibit higher accuracy. This corroborates our findings from Section III-B regarding the relative effectiveness of TLB miss sampling and to some extent explain why ET outperforms HawkEye even for 2MiB-sufficient workloads (Section VII-C).

E. Multi-workload experiments

Figure 14 presents the results for THP and ET when natively running mixes of workloads concurrently without fragmentation. We run three different mixes of workloads and plot the speedup achieved for each workload by THP and ET relative to 4KiB. ET is able to sustain its performance benefit over THP (cf. Figure 8) in multi-programmed execution due to the way CoalaPaging coordinates concurrent allocation requests from different programs (Section IV-B).

F. Overhead analysis

Fault latency. Figure 15 reports the cumulative distribution function (CDF) for the latency of CoalaPaging faults (64KiB and 32MiB), as well as 4KiB, 64KiB (mTHP), 2MiB (THP) and 32MiB (THP-16KiB granule) synchronous faults. We run a micro-benchmark that triggers 100K random anonymous

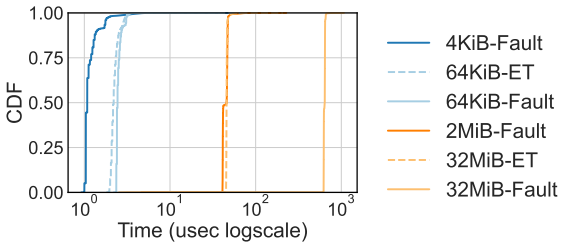


Fig. 15: Fault latency CDF

faults and collects the latency of the fault handler. 64KiB ET faults exhibit increased fault latency compared to 4KiB. Linux has an extremely fast path for allocating 4KiB pages (∼1us), utilizing lockless per-CPU page lists. 64KiB ET faults are slightly faster than mTHP's 64KiB faults, as the increased fault size incurs overhead, e.g., synchronous zeroing. On the other hand, 32MiB ET faults perform closely to THP and are an order of magnitude faster than synchronous 32MiB faults, since synchronous 32MiB faults have to zero large blocks of memory, while ET relies on smaller fault-time allocations (2MiB). These results support our design choice to opportunistically allocate contiguous pages across faults and underline its benefits versus an alternative design which relies on larger fault-time allocations [65].

Memory Bloat. In Figure 9, the normalized page distribution for canneal and streamcluster exceeds 100% for THP and ET. The reason is that 2MiB pages can increase the effective memory footprint of workloads [3, 4, 6] compared to 4KiB. 32MiB ET translations do not induce extra memory bloat compared to THP, owing to the opportunistic coalescing-aware allocation policy. For streamcluster ET favors 64KiB translations over 2MiB, which reduces memory bloat.

Takeaway 4: The opportunistic design of CoalaPaging keeps fault latency and memory bloat bounded while supporting translation sizes beyond 2MiB.

VIII. Related Work

Translation sizes and large pages. [66] propose HW and OS modifications to support a wider range of translation sizes. [8] study the effectiveness of 1GiB page sizes and design mechanisms to make their transparent support practical. [67] uses a mix of 2MiB and 1GiB pages to improve translation overhead modeling. We focus on harnessing the performance potential of the intermediate translation sizes enabled by TLB coalescing on real HW. Transparent OS large page management for the x86 architecture has been excessively studied for both Linux and FreeBSD [3–7, 20, 68]. ET is orthogonal and complementary to these works. ET alleviates fragmentation pressure by reducing 2MiB page usage (Section VII), Additionally, ET 32MiB translations build upon THP fault-time allocations, and are thus able to harness the improved THP performance of prior art.

Memory contiguity. Previous research focuses on generating physical memory contiguity, which can be exploited by novel HW components [10, 27] or used to improve THP performance [9, 34, 49]. Our work builds upon opportunistic allocation policies in the context of TLB coalescing.

Sampling-based profiling. [48] highlight the importance of TLB misses for guiding translation size selection and propose architectural extensions to accelerate scanning and promotion and assist the OS in page size selection. Per-core caches on the L2 TLB miss path track the number of misses per recently accessed 2MiB and 1GiB region. The contents of the caches are dumped to OS accessible memory at fixed intervals. Besides requiring bespoke HW, this solution is difficult to generalize for multiple translation sizes, requiring one cache per-size per-core. [69–73] use sampling-based profiling for memory deduplication and tiering. We follow a similar approach targeting translation performance, and corroborate their findings regarding the practicality and accuracy of this approach compared to page-based access frequency sampling.

Address Translation Hardware. Prior works improve translation performance via HW modifications [33, 74–85]. SpecTLB [86] and SpOT [27] exploit predictable contiguous mappings to speedup address translation. [41–43, 87] propose and improve upon HW TLB coalescing. Solomon et al. [46] evaluate the effectiveness of HW TLB coalescing on recent AMD processors [44, 45]. HW coalescing can be used together with OS-assisted coalescing to collectively reduce MMU pressure. The page table structure has also been extensively studied [32, 39, 40, 71]. Previous works have proposed new hashing-based schemes [28, 29, 88, 89], range tables [35, 90] as well as more radical changes [30, 31, 36, 91, 92] to the virtual memory hardware. Our work retains the radix tree structure and improves performance by enabling intermediate translation sizes.

IX. Conclusion

We design and implement *Elastic Translations (ET)* to take advantage of the extended range of translation sizes, supported, via OS-assisted TLB coalescing, by ARMv8-A and RISC-V. ET extend the OS memory manager to enable the transparent and opportunistic creation of intermediate-sized translations, both at fault time (*CoalaPaging*) and asynchronously (*CoalaKhugepaged*) for both native and virtualized execution. *Leshy*, a HW-assisted profiler, samples the TLB misses of applications at runtime, to estimate address translation overhead and implements the *ET policies* for translation size selection and drives the ET in-kernel mechanisms to optimally map the application footprint to the multiple available translation sizes. By leveraging multiple translation sizes and runtime profiling, ET is able to significantly speed-up execution for memory intensive workloads when compared to state-of-practice and state-of-the-art, for both native and virtualized execution, under varying levels of fragmentation.

Acknowledgments

We thank the anonymous reviewers and artifact evaluators for their valuable feedback. This work was funded by the European Union under Horizon Europe grant 101092850 (project AERO).

ARTIFACT APPENDIX

The artifact comprises a parent Git repository, hosted on GitHub, which includes the necessary instructions (*README.md*), scripts (*scripts/*), binaries (*bin/, benchmarks/*), datasets (*datasets/*) and source code (*src/*) to build, run and evaluate *Elastic Translations*. The source code for each component is split into its own Git repository, which is then included in the parent repository as a Git submodule.

ET is implemented on top of Linux v5.18.19. The *et-linux* repository also includes our Hawkeye port to Linux v5.18.19 on arm64 and the kernel configs we used to evaluate *ET* and *Hawkeye* for both native (Ampere Altra, NVIDIA GH200) and virtualized (QEMU) scenarios. The *Leshy* profiler along with various userspace tools and utilities (memory fragmentation tool, ET userspace configuration utility, accessbit sampler, etc.) are included in the etutils-rs repository. We also provide our slightly modified QEMU and gperftools tcmalloc. Finally, we include a repository with the v6.8rc Linux kernel source code we used to evaluate *mTHP* (multi-sized THP).

We provide, in the parent repository, the source code for the *hashjoin, svm, btree, gups and bfs* benchmarks we use in the evaluation as well as a patch, to enable profiling, for the PARSEC benchmarks we used (*canneal and streamcluster*). The SPEC CPU benchmarks (*astar, omnetpp*) do not require any modifications. We also include the scripts necessary to download and create or prepare the input datasets for the *canneal, svm and bfs* benchmarks. To ease the initial evaluation, we also provide pre-built images and binaries for the kernels, userspace tools and the benchmarks as well as the prepared datasets.

Using the provided scripts, one can prepare (*scripts/prepare.sh*) the host for building, running and evaluating ET. *scripts/build.sh* builds the *ET, Hawkeye and mTHP* kernels as well as the userspace utilities and benchmarks. The compiled artifacts are installed via *scripts/install.sh*. We also provide scripts (*scripts/run*.sh*), which configure the host and run the various evaluation scenarios. Finally, under *scripts/plots/*, we provide scripts which aggregate, summarize and plot the results, output from the aforementioned run scripts.

For the evaluation, an *ARMv8.2+-A* server is required. The paper-reported results were obtained on an *Ampere Altra Mt.Jade* 2-socket server with 80 Neoverse N1 cores and 256GiB of memory in each socket. For both native and virtualized scenarios, we used Ubuntu Jammy 22.04. Results might vary if a server with different ARM cores is used, especially if the TLB size differs.

A. Artifact check-list (meta-information)

- **Data sets:** KDD12 for SVM, Friendster SNAP graph for GAPBS/BFS, synthetically generated netlist for Canneal
- **Run-time environment:** Ubuntu Jammy 22.04
- **Hardware:** ARMv8.2+-A server, preferably one with Neoverse N1 cores (e.g., Ampere Altra)
- **Metrics:** Cycles, L2 TLB misses, wall-clock time, translation-size distribution
- **Experiments:** Native execution with and without fragmentation, virtualized execution without fragmentation
- **How much disk space required (approximately)?:** 100GiB
- **How much time is needed to prepare workflow (approximately)?:** 1hr
- **How much time is needed to complete experiments (approximately)?:** 12hr
- **Publicly available?:** Yes, on GitHub
- **Code licenses (if publicly available)?:** GPLv2 (for newly-developed code) and other free software and open source licenses used by projects included in the artifact
- **Archived (provide DOI)?:** 10.5281/zenodo.13621499

B. Description

1) How to access: The artifact is hosted on GitHub. To access it clone the repository and all of its submodules:

```
# git clone --recurse-submodules
    https://github.com/cslab-ntua/
        elastic-translations-MICRO2024
```

We also provide a script and a VM artifact bundle, to ease and speed-up the initial testing and evaluation phase. *scripts/install_vm_bundle.sh* will download and extract the artifact bundle, which includes a VM image (*artifact.img*), under *artifact_vm_bundle*. *run-vm.sh* can be used to spawn the QEMU VM. You can then access the VM either via the QEMU console, using the credentials *ubuntu / ubuntu*, or by SSHing to the VM:

```
# ssh -p65433 ubuntu@localhost
```

using the same credentials. The artifact bundle also includes an ED25519 SSH key pair. The public key is already installed in the artifact bundle for both *root* and *ubuntu* users.

Finally, you can also use the `run-vm-noefi.sh` script, for booting pre-built VM kernels directly from the host, without booting to GRUB. The artifact bundle includes pre-compiled VM kernels (ET, Hawkeye, vanilla) under *kernels/*.

2) Hardware dependencies: ET requires a machine with ARMv8-A CPUs with support for the contig-bit in their TLBs (cf. ARMv8-A architecture reference manual D8.6.1). Additionally, Leshy requires support for the ARMv8.2-A Statistical Profiling Extension (SPE) (cf. ARMv8-A architecture reference manual A2.14). The benchmarks have a maximum memory footprint of 122GiB. For the paper, we used a 2-socket Ampere Altra Mt.Jade server, with 80 Neoverse N1 (ARMv8.2+-A) CPUs and 256GiB memory in each socket. We've also verified that ET run on NVIDIA Grace CPU (ARMv9 Neoverse V2 cores) and provide the kernel config we used to build and boot our kernel on a SuperMicro NVIDIA GH200 server.

3) Software dependencies: For our evaluation, we used Ubuntu Jammy (22.04) for both native and virtualized execution. We list and install the required packages for building and running the artifact in *scripts/prepare.sh*.

4) Data sets:

- SVM: KDD12
- BFS: Friendster SNAP graph, converted to a GAPBS-ingestible (edgelist) format
- Canneal: synthetically generated netlist, created by the (provided) script *prepare_canneal_dataset.sh*

C. Installation

The artifact scripts depend on the `$BASE` environmental variable, which should point to the parent artifact repository. It can be set either by directly editing the scripts or by exporting it to the desired path, i.e.:

```
# export BASE="/path/to/repo"
```

Then, inside the cloned parent repository run:

```
# ./scripts/prepare.sh
# VM=1 KERNEL="et.full" ./scripts/build.sh
# VM=1 KERNEL="et.full" ./scripts/install.sh
# reboot
```

After installing and booting the desired kernel, one can configure and run *scripts/run_test.sh* to verify that everything works.

```
# ./scripts/run_test.sh
```

Both *build.sh* and *install.sh* include knobs to configure and build various Linux kernels and kernel configurations, controlled by the `$KERNEL` and `$VM` environmental variables. One can also navigate to the individual kernel, QEMU and benchmark source directories and manually configure and build each component as well as generate or download the required datasets.

D. Experiment workflow

In order to evaluate ET, one would generally:

- configure, build and boot the required kernel (ET, Hawkeye, Vanilla), either via *scripts/*{*prepare, build*}*.sh* or manually,
- use or modify any of the *run scripts (scripts/run*.sh)* to run the experiment,
- analyze, parse and plot the results under *results/*{*host, vm*} using the scripts under *scripts/plots/*.

E. Evaluation and expected results

For reproducing the paper evaluation results, the artifact includes several *run scripts*:

- *scripts/run-test.sh* is a minimal script to test that ET works. The script can be tweaked (via the env variables passed to *bin/run.sh*) to run different test scenarios.
- *scripts/run-fig2-hugetlb.sh* run the 64KiB and 32MiB intermediate translation performance evaluation via HugeTLB (Fig. 2). The script can be tweaked to change the workloads that should be run (`$BENCHMARKS`), the number of iterations for each workload (`$ITER`) and the translation sizes to evaluate (`$sizes`). The script will perform both native and virtualized runs, but either can be commented out and skipped.
- *scripts/run-fig15-pflat.sh* generates the fault latency CDF of Fig. 15. It requires a pfrace-enabled kernel (`CONFIG_PFTRACE`). Note that for the 64KiB and 32MiB non-ET fault latencies, different kernels are required, compiled with the `CONFIG_ARM64_64K_PAGES` and `CONFIG_ARM64_16K_PAGES` options set respectively.
- *scripts/run-fig14-multi.sh* will run the three workload mixes from Fig. 14. The `$RUN` variable controls whether to do a *baseline* (4K), *thp* (THP) or an *et* (ET) run. The script can be tweaked to evaluate different workload mixes.
- *scripts/run-fig10-virt.sh* reproduces the virtualized execution results of Fig. 10. The `$RUN` variable controls whether to do a *baseline* (4K), *thp* (THP), *et* (ET) or a *hwk* (Hawkeye) run. Similarly to the other scripts, the workloads (`$BENCHMARKS`), iterations (`$ITER`), and other options can be tweaked as needed.
- *scripts/run-eval-base.sh* is the bulkiest script, which can be used to reproduce the results from figures 8, 9, 11, 12, 13. Similarly to the other scripts, the `$RUN`, `$BENCHMARKS` and `$ITER` variables can be tweaked to change the parameters of the run. Additionally, `$FRAG_TARGET` sets the FMFI target for the run (e.g., 50% or 99%).

F. Experiment customization

The artifact's main driving scripts are *bin/run*.sh* and *bin/prctl.sh*. Each script can be configured via environmental variables to e.g., run different ET or Hawkeye scenarios. *run.sh* is the wrapper script which drives *run-benchmarks.sh*. Finally, for Hawkeye and ET, *run-benchmarks.sh* will call *prctl.sh* for ET and Hawkeye-specific configuration. The input environmental variables for these scripts are documented at the beginning of each script.

G. Notes

The ET Github repository includes an expanded artifact appendix with more details on i) the methodology used for the evaluation, describing the various tools and methods we used to measure the performance of ET and ii) how to troubleshoot ET, describing debugging tools and utilities that we used while developing ET for functionality and regression testing.

References

[1] A. Bhattacharjee, "Preserving Virtual Memory by Mitigating the Address Translation Wall," *IEEE Micro*, 2017. [Online]. Available: https://doi.org/10.1109/MM.2017.3711640

[2] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in Supporting Two Page Sizes," in *Proceedings of the 19th ACM/IEEE Annual International Symposium on Computer Architecture*, 1992. [Online]. Available: https://doi.org/10.1145/139669.140406

[3] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and Efficient Huge Page Management with Ingens," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016. [Online]. Available: https://doi.org/10.5555/3026877.3026931

[4] A. Panwar, S. Bansal, and K. Gopinath, "HawkEye: Efficient Fine-grained OS Support for Huge Pages," in *Proceedings of the 24th ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304064

[5] M. Mansi, B. Tabatabai, and M. M. Swift, "CBMM: Financial Advice for Kernel Memory Managers," in *Proceedings of the 2022 USENIX Annual Technical Conference*, 2022. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/mansi

[6] W. Zhu, A. L. Cox, and S. Rixner, "A Comprehensive Analysis of Superpage Management Mechanisms and Policies," in *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020. [Online]. Available: https://doi.org/10.5555/3489146.3489203

[7] T. Michailidis, A. Delis, and M. Roussopoulos, "MEGA: Overcoming Traditional Problems with OS Huge Page Management," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019. [Online]. Available: https://doi.org/10.1145/3319647.3325839

[8] V. S. S. Ram, A. Panwar, and A. Basu, "Trident: Harnessing Architectural Resources for All Page Sizes in X86 Processors," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021. [Online]. Available: https://doi.org/10.1145/3466752.3480062

[9] K. Zhao, K. Xue, Z. Wang, D. Schatzberg, L. Yang, A. Manousis, J. Weiner, R. Van Riel, B. Sharma, C. Tang, and D. Skarlatos, "Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters," in *Proceedings of the 50th ACM/IEEE Annual International Symposium on Computer Architecture*, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589079

[10] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation Ranger: Operating System Support for Contiguity-aware TLBs," in *Proceedings of the 46th ACM/IEEE International Symposium on Computer Architecture*, 2019. [Online]. Available: https://doi.org/10.1145/3307650.3322223

[11] M. Mansi and M. M. Swift, "Characterizing physical memory fragmentation," https://arxiv.org/abs/2401.03523, 2024.

[12] "Transparent Hugepage Support," https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[13] R. Roberts, "Multi-size THP for anonymous memory," https://lwn.net/Articles/954094/.

[14] R. Roberts, "Transparent contiguous PTEs for User mappings"," https://lore.kernel.org/linux-arm-kernel/87fs0xxd5g.fsf@nvdebian.thelocal/T/.

[15] "HugeTLB Pages," https://docs.kernel.org/arch/arm64/hugetlbpage.html.

[16] "HugeTLBpage on ARM64," https://www.kernel.org/doc/html/latest/arm64/hugetlbpage.html.

[17] "libhugetlbfs," https://github.com/libhugetlbfs/libhugetlbfs.

[18] J. Navarro, S. Iyer, and A. Cox, "Practical, Transparent Operating System Support for Superpages," in *Proceedings of the 5th ACM SIGOPS Symposium on Operating Systems Design and Implementation*, 2002. [Online]. Available: https://doi.org/10.1145/844128.844138

[19] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[20] A. Panwar, A. Prasad, and K. Gopinath, "Making Huge Pages Actually Useful," in *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018. [Online]. Available: https://doi.org/10.1145/3173162.3173203

[21] *Arm Architecture Reference Manual for A-profile architecture, Rev. J.a*, ARM Corporation, 2023, https://developer.arm.com/documentation/ddi0487/latest/.

[22] *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, RISC-V Foundation, 2021, https://wiki.riscv.org/display/HOME/RISC-V+Technical+Specifications.

[23] T. Prickett Morgan, "AWS Adopts Arm V2 Cores For Expansive Graviton4 Server CPU," https://www.nextplatform.com/2023/11/28/aws-adopts-arm-v2-cores-for-expansive-graviton4-server-cpu/.

[24] A. Vahdat, "Introducing Google Axion Processors, our new Arm-based CPUs," https://cloud.google.com/blog/products/compute/introducing-googles-new-arm-based-cpu, 2024.

[25] M. Awad, "Arm Collaborates with Microsoft on Custom Silicon to Unlock Sustainable, AI-Driven Infrastructure," https://newsroom.arm.com/news/microsoft-custom-silicon-on-arm, 2024.

[26] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux Virtual Machine Monitor," in *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'07)*, 2007. [Online]. Available: https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf

[27] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, "Enhancing and Exploiting Contiguity for Fast Memory Virtualization," in *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*, 2020. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00050

[28] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism," in *Proceedings of the 25th ACM International Conference on Architectural*

Support for Programming Languages and Operating Systems, 2020. [Online]. Available: http://doi.org/10.1145/3373376.3378493

[29] J. Stojkovic, D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022. [Online]. Available: https://doi.org/10.1145/3503222.3507720

[30] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *Proceedings of the ACM/IEEE 40th Annual International Symposium on Computer Architecture*, 2013. [Online]. Available: https://doi.org/10.1145/2485922.2485943

[31] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks," in *Proceedings of the 47th IEEE/ACM Annual International Symposium on Microarchitecture*, 2014. [Online]. Available: https://doi.org/10.1109/MICRO.2014.37

[32] T. Merrifield and H. R. Taheri, "Performance Implications of Extended Page Tables on Virtualized X86 Processors," in *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2016. [Online]. Available: https://doi.org/10.1145/2892242.2892258

[33] 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*, 2022. [Online]. Available: https://doi.org/10.1145/3503222.3507718

[34] 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*, 2021. [Online]. Available: https://doi.org/10.1145/3445814.3446704

[35] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant Memory Mappings for fast access to large memories," in *Proceedings of the 42nd ACM/IEEE Annual International Symposium on Computer Architecture*, 2015. [Online]. Available: https://doi.org/10.1145/2749469.2749471

[36] S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer, "Rebooting Virtual Memory with Midgard," in *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture*, 2021. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00047

[37] I. Corporation, "5-Level Paging and 5-Level EPT White Paper," 2017. [Online]. Available: https://cdrdv2-public.intel.com/671442/5-level-paging-white-paper.pdf

[38] CXL Consortium, "Compute Express Link Specification Revision 2.0." https://www.computeexpresslink.org/download-the-specification, 2023.

[39] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile Paging: Exceeding the Best of Nested and Shadow Paging," in *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture*, 2016. [Online]. Available: https://doi.org/10.1109/ISCA.2016.67

[40] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-Dimensional Page Walks for Virtualized Systems," in *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008. [Online]. Available: https://doi.org/10.1145/1346281.1346286

[41] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012. [Online]. Available: https://doi.org/10.1109/MICRO.2012.32

[42] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture*, 2014. [Online]. Available: https://doi.org/10.1109/HPCA.2014.6835964

[43] 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 44th ACM/IEEE Annual International Symposium on Computer Architecture*, 2017. [Online]. Available: https://doi.org/10.1145/3079856.3080217

[44] *Software Optimization Guide for AMD EPYC™ 7003 Processors, Rev 3.00*, AMD, 2020, https://developer.amd.com/resources/developer-guides-manuals/.

[45] M. Clark, "A new ×86 core architecture for the next generation of computing," in *Proceedings of the 2016 IEEE Hot Chips 28 Symposium*, 2016. [Online]. Available: https://doi.org/10.1109/HOTCHIPS.2016.7936224

[46] E. H. Solomon, Y. Zhou, and A. L. Cox, "An Empirical Evaluation of PTE Coalescing," in *Proceedings of the 2023 IEEE International Symposium on Memory Systems*, 2023. [Online]. Available: https://doi.org/10.1145/3631882.3631902

[47] A. L. Cox., "Medium-sized superpages on arm64 and beyond," https://www.freebsd.org/status/report-2022-04-2022-06/superpages/, 2022.

[48] A. Manocha, Z. Yan, T. Esin, J. L. Aragón, N. David, and M. Martonosi, "Architectural Support for Optimizing Huge Page Selection Within the OS," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023. [Online]. Available: https://webs.um.es/jlaragon/papers/manocha_MICRO23.pdf

[49] W. Jia, J. Zhang, J. Shan, and X. Ding, "Making Dynamic Page Coalescing Effective on Virtualized Clouds," in *Proceedings of the 18th ACM SIGOPS European Conference on Computer Systems*, 2023. [Online]. Available: https://doi.org/10.1145/3552326.3567487

[50] Y. Zhou, A. L. Cox, S. Dwarkadas, and X. Dong, "The Impact of Page Size and Microarchitecture on Instruction Address Translation Overhead," *ACM Trans. Archit. Code Optim.*, 2023. [Online]. Available: https://doi.org/10.1145/3600089

[51] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, 2006. [Online]. Available: https://doi.org/10.1145/1186736.1186737

[52] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008. [Online]. Available: https://doi.org/10.1145/1454115.1454128

[53] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017.

[54] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, 2012. [Online]. Available: http://arxiv.org/abs/1205.6233

[55] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. [Online]. Available: https://www.mcs.anl.gov/papers/P5064-0114.pdf

[56] "LibSVM," https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html.

[57] "KDD'12 dataset," https://www.kaggle.com/c/kddcup2012-track1, 2012.

[58] "GUPS: HPCC RandomAccess benchmark," https://github.com/alexandermerritt/gups.

[59] "WiWynn Mt.Jade," https://www.wiwynn.com/products/19-inch/sv328r.

[60] *Ampere® Altra® Rev A1 64-Bit Multi-Core Processor Datasheet, Rev 1.40*, Ampere Computing, 2023, https://amperecomputing.com/customer-connect/products/altra-family-device-documentation.

[61] *Arm® Neoverse™ N1 Core, Rev r4p1*, ARM Corporation, 2023, https://developer.arm.com/documentation/100616/0401/.

[62] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, "Learning-based Memory Allocation for C++ Server Workloads," in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020. [Online]. Available: https://doi.org/10.1145/3373376.3378525

[63] "gperftools," https://github.com/gperftools/gperftools.

[64] M. Gorman and A. Whitcroft, "The what, the why and the where to of anti-fragmentation," in *Proceedings of the 2006 Ottawa Linux Symposium*, 2006. [Online]. Available: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-369-384.pdf

[65] J. Corbet, "Large folios for anonymous memory," https://lwn.net/Articles/937239/.

[66] F. Guvenilir and Y. N. Patt, "Tailored Page Sizes," in *Proceedings of the 47th ACM/IEEE International Symposium on Computer Architecture*, 2020. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00078

[67] M. Agbarya, I. Yaniv, J. Gandhi, and D. Tsafrir, "Predicting Execution Times With Partial Simulations in Virtual Memory Research: Why and How," in *Processors of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00046

[68] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large Pages May Be Harmful on NUMA Systems," in *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014. [Online]. Available: https://doi.org/10.5555/2643634.2643659

[69] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. S. Lui, "SmartMD: A High Performance Deduplication Engine with Mixed Pages," in *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017. [Online]. Available: https://doi.org/10.5555/3154690.3154759

[70] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, "MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023. [Online]. Available: https://doi.org/10.1145/3600006.3613167

[71] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023. [Online]. Available: https://doi.org/10.1145/3582016.3582063

[72] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021. [Online]. Available: https://doi.org/10.1145/3477132.3483550

[73] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic, B. Morris, C. Mukherjee, J. Ren, G. Thelen, P. Turner, C. Villavieja, P. Ranganathan, and A. Vahdat, "Towards an adaptable systems architecture for memory tiering at warehouse-scale," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, 2023.

[74] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," in

Proceedings of the ACM/IEEE 37th Annual International Symposium on Computer Architecture, 2010. [Online]. Available: https://doi.org/10.1145/1815961.1815970

[75] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh, "Perforated Page: Supporting Fragmented Memory Allocation for Large Pages," in *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*, 2020. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00079

[76] 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*, 2021. [Online]. Available: https://doi.org/10.1145/3459898.3463902

[77] 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 ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017. [Online]. Available: https://doi.org/10.1145/3079856.3080210

[78] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched Address Translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019. [Online]. Available: https://doi.org/10.1145/3352460.3358294

[79] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, 2015. [Online]. Available: https://doi.org/10.1109/HPCA.2015.7056035

[80] M. A. Bender, A. Bhattacharjee, A. Conway, M. Farach-Colton, R. Johnson, S. Kannan, W. Kuszmaul, N. Mukherjee, D. Porter, G. Tagliavini, J. Vorobyeva, and E. West, "Paging and the Address-Translation Problem," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021. [Online]. Available: https://doi.org/10.1145/3409964.3461814

[81] D. Skarlatos, U. Darbaz, B. Gopireddy, N. S. Kim, and J. Torrellas, "BabelFish: Fusing Address Translations for Containers," in *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2020. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00049

[82] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs," in *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, 2015. [Online]. Available: https://doi.org/10.1109/HPCA.2015.7056034

[83] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017. [Online]. Available: https://doi.org/10.1145/3037697.3037704

[84] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, "CSALT: Context Switch Aware Large TLB," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017. [Online]. Available: https://doi.org/10.1145/3123939.3124549

[85] S. Bergman, M. Silberstein, T. Shinagawa, P. Pietzuch, and L. Vilanova, "Translation Pass-Through for Near-Native Paging Performance in VMs," in *Proceedings of the 2023 USENIX Annual Technical Conference*, 2023. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/bergman

[86] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," in *Proceedings of the ACM/IEEE 38th Annual International Symposium on Computer Architecture*, 2011. [Online]. Available: https://doi.org/10.1145/2000064.2000101

[87] B. Pham, J. Veselý, 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 IEEE/ACM 48th International Symposium on Microarchitecture*, 2015. [Online]. Available: https://doi.org/10.1145/2830772.2830773

[88] 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*, 2023. [Online]. Available: https://doi.org/10.1145/3582016.3582021

[89] I. Yaniv and D. Tsafrir, "Hash, Don'T Cache (the Page Table)," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, 2016. [Online]. Available: https://doi.org/10.1145/2896377.2901456

[90] D. Chen, D. Tong, C. Yang, J. Yi, and X. Cheng, "FlexPointer: Fast Address Translation Based on Range TLB and Tagged Pointers," *ACM Trans. Archit. Code Optim.*, 2023. [Online]. Available: https://doi.org/10.1145/3579854

[91] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing Memory in Heterogeneous Systems," in *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018. [Online]. Available: https://doi.org/10.1145/3173162.3173194

[92] B. Suchy, S. Campanoni, N. Hardavellas, and P. Dinda, "CARAT: A Case for Virtual Memory through Compiler- and Runtime-Based Address Translation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020. [Online]. Available: https://doi.org/10.1145/3385412.3385987