Enhancing and Exploiting Contiguity for Fast Memory Virtualization

Chloe Alverti Stratos Psomadakis Vasileios Karakostas Jayneel Gandhi† Konstantinos Nikas
Georgios Goumas Nectarios Koziris
National Technical University of Athens †VMware Research
{xalverti, psomas, vkarakos, knikas, goumas, nkoziris}@cslab.ece.ntua.gr gandhij@vmware.com

***Abstract*—We propose synergistic software and hardware mechanisms that alleviate the address translation overhead, focusing particularly on virtualized execution. On the software side, we propose contiguity-aware (CA) paging, a novel physical memory allocation technique that creates larger-than-a-page contiguous mappings while preserving the flexibility of demand paging. CA paging applies to the hypervisor and guest OS memory manager independently, as well as to native systems. Moreover, CA paging benefits any address translation scheme that leverages contiguous mappings. On the hardware side, we propose SpOT, a simple micro-architectural mechanism to hide TLB miss latency by exploiting the regularity of large contiguous mappings to predict address translations in both native and virtualized systems. We implement and emulate the proposed techniques for the x86-64 architecture in Linux and KVM, and evaluate them across a variety of memory-intensive workloads. Our results show that: (i) CA paging is highly effective at creating vast contiguous mappings, even when memory is fragmented, and (ii) SpOT exploits the created contiguity and reduces address translation overhead of nested paging from ∼16.5% to ∼0.9%.**

***Index Terms*—virtualization, address translation, virtual memory, memory management**

I. Introduction

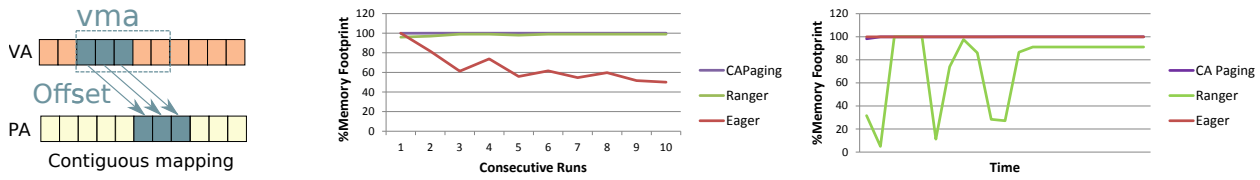
Page-based address translation overheads are alleviated by caching translations in Translation Look-aside Buffers (TLBs). However, the growing demand for physical memory is limiting the efficacy of TLBs, increasing the rate of costly TLB misses. To make things worse, the adoption of virtualized cloud infrastructure amplifies these overheads. The state-of-practice MMU virtualization technique (hardware-assisted nested paging [1]) requires two-dimensional address translation that increases the TLB miss penalty up to 6× compared to native execution. Looking forward, persistent memory will hugely increase physical memory, requiring 5-level paging [2], further exacerbating the cost of TLB misses. Ideally, software and hardware support for address translation should minimize overheads, maintain memory availability with flexible allocations, avoid memory waste by minimizing fragmentation and preserve resource-saving mechanisms like demand paging and copy-on-write, under both native and virtualized systems.

In response, industry has increased the page size and translation hardware [3]. We show that 2MB huge pages still fail to cover the needs of irregular workloads, and nested paging magnifies the problem. Even though increasing the page size might seem tempting, larger pages increase internal fragmentation and restrict fine-grained memory management [4]–[8].

Prior works [9]–[16] have shown the potential of breaking the traditional page-based mapping between hardware translation and OS memory management. They usually exploit larger-than-a-page contiguous mappings [17] but fail to achieve the desired flexibility. For example, previous proposals rely on pre-allocation [10], [11], which suffers from external fragmentation, and is antagonistic to demand paging. Additionally, prior hardware schemes [11], [16] track the exact boundaries of contiguous mappings, requiring complex extensions when applied in virtualized execution. Finally, prior approaches for increasing TLB reach have been limited by indexing and alignment requirements [12]–[16] reducing their efficacy.

In this paper, we aim to reduce the address translation overhead, focusing particularly on virtualized execution. We generate and exploit larger-than-a-page contiguous mappings while avoiding pre-allocation to preserve the flexibility of existing paging-based mechanisms. We base our approach on the key observation that contiguity can be expressed simply through *offsets*, decoupling contiguous mappings from virtual boundary checks. To capitalize on our observation, we take a two-fold approach: (i) we introduce *contiguity-aware (CA) paging* which promotes contiguity as a first-class citizen in the OS memory manager, and (ii) we harvest the generated contiguity to accelerate address translation via *SpOT* at the microarchitecture level. Note that, these proposed mechanisms can improve both native and virtualized execution.

CA paging enables the OS memory allocator to generate contiguous mappings beyond the page-size limit, using minimal per-process metadata. Specifically, CA paging allocates contiguous physical pages to map contiguous virtual memory regions of processes, working across page faults and on a best-effort basis. In this way, CA paging creates and extends contiguous mappings gradually while preserving the increased memory utilization and low tail latency of demand paging. CA paging can improve the performance of any hardware design that relies on contiguous mappings [11]–[16] and can be used both in native and virtualized execution. Our results show that CA paging significantly boosts the creation of vast contiguous mappings, achieves performance similar to pre-allocation, and outperforms it in the presence of external fragmentation. Compared to asynchronous defragmentation [18], CA paging operates on the allocation path and generates contiguity instantly, increasing the opportunity to exploit contiguity and avoiding the cost of the post-allocation page migrations.



(a) (b) (c)

Fig. 1: (a) Larger-than-a-page contiguous virtual-to-physical mapping. (b,c) Trade-offs between pre-allocation (eager paging), asynchronous defragmentation (ranger), and CA paging. (b) Pre-allocation suffers from external fragmentation, as the coverage of 32 largest mappings drops when PageRank runs repetitively. (c) Asynchronous defragmentation delays contiguity generation.

To better exploit the large generated contiguous mappings of CA paging, we propose *Speculative Offset-based Address Translation (SpOT)*. SpOT is a simple hardware mechanism on the TLB miss path that speculates based on large contiguous mappings to predict translations while performing verification of page walks in the background. SpOT exploits the OS provided contiguity at the micro-architectural level, requiring minimal hardware support. In contrast to prior speculation designs [19], [20], SpOT predicts translations far beyond the huge page limit and is completely independent of virtual addressing and alignment. SpOT supports both native and virtualized systems. Our evaluation in a virtualized system shows that SpOT combined with CA paging reduces the address translation overhead of nested paging from ∼16.5% to ∼0.9% on average. SpOT performs close to prior schemes [10], [11] but without pre-allocation and complex virtualization extensions at the architecture level. While speculation introduces security concerns, SpOT uses the same generic mitigation mechanisms proposed [21], [22] for other speculation attacks.

II. Background

Virtualization introduces an extra layer of indirection between guest applications and physical memory: (i) the guest virtual (gVA), (ii) the guest physical (gPA), and (iii) the host physical (hPA) address space. With nested paging [1], the guest OS controls guest page tables (gPT) that hold gVA→gPA mappings, and the hypervisor maintains nested page tables (nPT) that hold gPA→hPA mappings independently. In case of a TLB miss, the 2D address translation (gVA→hPA) is retrieved by the hardware, which walks both tables in a nested fashion, requiring up to 24 memory references.

Huge pages increase the TLB reach and decrease the page walk latency by reducing the required memory accesses. Their efficiency [23] has made them the state-of-practice mitigation technique and modern operating systems support transparently 2MB pages (e.g., THP [24]). However, they still fail to eliminate translation overheads for big-memory irregular workloads. Section VI shows that ∼16.5% of applications execution time is still spent in nested page walks even when >99% of their footprints is backed by 2MB.

Huge page sizes larger than 2MB can push the translation performance barrier further away. In x86-64, though, the out-of-the-box eligible sizes are 1GB and 512GB due to the page table layout. However, such huge gaps bear challenges. First, paged translation requires alignment, but large aligned free blocks quickly become scarce in long running systems [18]. Moreover, transparent management is not straightforward. For example, intermediate sized mappings will either use larger pages wasting physical resources or will use multiple smaller ones suffering from translation penalties. In fact, all considerations around 2MB management, including page fault tail latency, fairness, and NUMA placement [4]–[8], manifest more severely as the page size increases.

Segmentation. Direct Segments [9], [10] use segmentation for a primary virtual region of an application that is contiguously mapped to a physical segment. Paging for that region is disabled but not for the rest of the virtual address space. The translation for addresses falling into the region is fast but the method is rigid in terms of memory management as segment memory cannot be reclaimed until the application terminates.

Contiguous mappings. State-of-the-art approaches [11]–[16] preserve paging and increase the TLB reach by leveraging contiguous mappings, i.e., contiguous virtual pages mapped to contiguous physical pages (Figure1a). TLB coalescing [12]–[14], [16] compacts such mappings to a single translation entry cached to slightly modified TLBs. However, it provides limited reach due to indexing and alignment restrictions. Redundant Memory Mappings (RMM) [11] extend Direct Segments by introducing range translations, i.e., [Base,Limit,Offset] representations of unaligned mappings with unlimited size, redundant to paging. However, RMM relies on pre-allocation and requires complex extensions to track and retrieve the exact boundaries of contiguous mappings (Section IV).

III. Software Technique: Contiguity-aware Paging

A. Key design concepts

All the state-of-the-art techniques that increase TLB reach require contiguity across pages [11]–[16]. Prior proposals lie on two extremes: they either rely on randomly generated contiguity [12]–[14], [16], or on brute-force pre-allocation schemes via reservation [9], [10] and eager paging [11]. The former clearly wastes opportunities for contiguity, while the latter abandons key OS mechanisms for flexible memory allocation and is sensitive to external fragmentation. Translation Ranger [18] is a recent technique that creates contiguity performing asynchronous and iterative memory defragmentation. A system daemon scans periodically process memory and migrates random physical pages to contiguous ones. Ranger is an effective contiguity mechanism, but migrations may delay to coalesce

an application's footprint; migrations also penalize memory accesses latency and trigger costly TLB shootdowns [25], [26].

We propose CA paging, an extension to the core OS memory manager that operates at allocation time and creates large contiguous mappings preserving the flexibility of demand paging. Figure 1b shows the percentage of the PageRank's memory footprint that is covered by the 32 largest contiguous mappings for 10 consecutive runs of the benchmark (Section V describes our methodology in detail). We observe that eager paging is sensitive to external fragmentation as the coverage drops progressively. CA paging, instead, sustains contiguity, harvesting unaligned physical contiguity in the system. Figure 1c shows the percentage of XSBench's memory footprint that is covered by the 32 largest contiguous mappings during the entire execution of the benchmark. We observe that Ranger's migrations delay to coalesce the application's footprint to contiguous memory. CA paging, instead, avoids unnecessary post-allocation migrations harvesting the system's available contiguity at page fault time.

B. Overview of CA paging

Contiguity-aware (CA) paging relies on the existing demand paging mechanism, but instead of allocating physical pages randomly, it steers the allocation of physical pages to create contiguous mappings. We introduce lightweight mechanisms and policies to the core physical OS allocator to lazily create vast contiguous mappings across page faults. CA paging requires minimal metadata, i.e., an *Offset* per virtual memory area (VMA) and a system-wide *contiguity map*. To decide the placement of a VMA's pages, CA paging uses a next-fit policy. CA paging deals with external fragmentation, supports multithreaded applications, and serves all common page fault types. We design and prototype CA paging in Linux for native and virtualized systems with KVM nested paging.

Demand Paging. Process VMAs are contiguous virtual address ranges, not necessarily backed by physical memory, represented by the `vma struct` in Linux. The OS allocates physical memory on demand, when each virtual page of a VMA is touched for the first time. The core memory manager is a power-of-two buddy allocator, maintaining [0, MAX_ORDER] lists. Each list is populated by free aligned blocks of 2^{order} pages. Allocation requests are served by the first available block of order 0 (4KB) or order 9 (2MB pages [24]) lists. Demand paging enables flexible memory management, but its random page allocations inhibit the creation of large contiguous mappings. Similar mechanisms are used by other OSes and hypervisors for demand paging.

Basic mechanism. CA paging leverages the unaligned and unlimited *Offset* representation of larger-than-a-page contiguous virtual-to-physical mappings. *Offset* is defined as the common $[virtual\_address - physical\_address]$ identifier for all pages belonging to the same mapping. CA paging tracks the *Offset* of the first page mapping created for each VMA (first page fault) and stores it as minimal metadata to the corresponding `vma struct`. On a future fault in the same VMA, CA paging uses the *Offset* to identify a target physical page for allocation. It

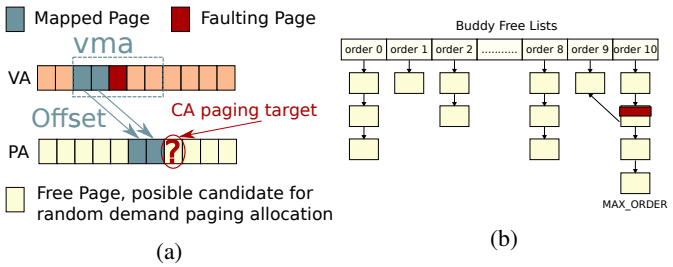


Fig. 2: Overview of contiguity-aware paging.

examines the occupational status of the target page, and if free, allocates it, extending the current VMA mapping contiguity. Figure 2a illustrates how CA paging exploits the notion of *Offset* to perform contiguous allocations.

CA paging examines the availability of the target page relying completely on existing OS metadata. In Linux, it retrieves a handle to the target page's structure using the system memory map (`mem_map`), indexed by page physical address. Dedicated attributes (`_mapcount`, `_count`) indicate if the target page is already in use. If the target page is free, it can be of the requested size or part of a larger block. In the latter case, CA paging splits the block using the default buddy allocator routine. In both cases, it retrieves the target page from buddy's lists (Figure 2b). CA paging is independent to the allocation order and serves both 4KB and 2MB page faults.

Contiguity Map. The first page allocation for a VMA can greatly affect the later-on generated contiguity. To maximize contiguity, CA paging directs the mapping to a region of the physical memory where there is enough free contiguity. To achieve this, a map of the system's free contiguous space is necessary. In Linux, the maximum size of tracked free memory is limited by the MAX_ORDER attribute. Typically that equals to 11, and the allocator maintains up to 4MB aligned free blocks. Prior research [11] proposes increasing MAX_ORDER but that approach is sensitive to external fragmentation (Section VI-A).

We introduce the *contiguity_map*, an indexing structure on top of the buddy allocator's MAX_ORDER list (Figure 3) to record unaligned contiguity at scales larger than the buddy heap. Each entry of the map represents a variable length sequence (cluster) of free MAX_ORDER blocks. It stores the starting physical address and the total size of the cluster. Updates to the map are triggered by all insertions/deletions to the corresponding buddy list. To avoid search operations on every update, all physically indexed base blocks of a cluster point to their corresponding *contiguity_map* entry (re-purposing the existing `mapping` attribute of the `page struct` which is not used when a page is free). We currently implement the map as a linked list sorted by physical address. Even if a tree could yield better performance, our evaluation shows that keeping the map up to date does not affect performance. A separate *contiguity_map* instance is maintained per NUMA node (`struct zone`), as the OS maintains a separate buddy instance per NUMA node.

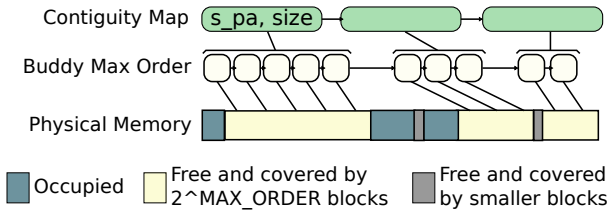


Fig. 3: The *contiguity_map* of CA paging.

C. CA paging Mechanism

Placing the first page. During the first page fault for a VMA, CA paging searches the *contiguity_map* for a free physical region that could fit it. Using the total VMA size as a key, it applies the next-fit placement policy; it searches for an available free block of the requested size, starting from where it left off the previous time. If no block larger or equal to the requested size is available, next-fit selects the largest found. CA paging allocates the first page of the selected region and sets the *Offset* attribute of the corresponding `vma struct`. Figure 4 visualizes the steps following the first fault in a VMA. Note that with CA paging, unlike the traditional segmentation case, a placement decision does not result in the allocation of the entire VMA. Instead, CA paging directs the forthcoming page faults of the same source VMA to the selected free block through the *Offset* attribute (best effort approach).

As CA paging does not allocate memory beyond the page size, competition for the same free blocks can arise when multiple faults by different processes or different VMAs of the same process trigger placement decisions. We opt for next-fit policy because it can defer such racing. The block that is selected to serve a placement request is the last one to be considered for the next request. To implement next-fit, we use a simple rover pointer over the *contiguity_map*.

Handling unsuccessful CA allocations. A CA paging allocation target may be unavailable, either because the end of a free physical block is reached or some other running process has allocated the target page. Upon failure, if the fault is for a huge page, CA paging runs again the placement decision routine using as key the size of the remaining unmapped VMA region (see below sub-VMA placement) and tracks the new *Offset*. If, instead, the fault is for a 4KB page, CA falls back to the default arbitrary allocation mechanism and skips the *Offset* tracking. Making decisions on top of huge pages is more effective when targeting vast contiguous mappings. Also huge allocations amortize placement overhead as they include costly large block zeroing operations.

Dealing with external fragmentation. If there is no available free block to fit an entire VMA (due to external fragmentation), CA paging makes multiple sub-VMA placement decisions and distributes the VMA to multiple smaller free physical blocks. The sub-placement decisions are triggered by unsuccessful allocation attempts (see above). In such scenarios CA paging performance depends on the fault pattern. To support multiple sub-VMA regions, we track multiple *Offsets* per VMA (instead of a single one) combined with the virtual address of the fault that created them. During a page fault, CA paging picks

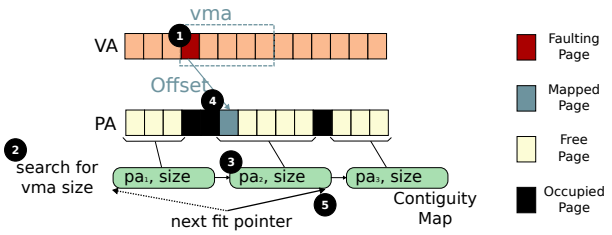


Fig. 4: Placement avoids fragmented regions: 1) first page fault 2) *contiguity_map* search for free region, 3) allocation, 4) Offset update, 5) next-fit rover pointer update.

the *Offset* associated with the virtual address closest to the currently faulting. To control the search latency we track up to 64 *Offsets* per VMA and apply a FIFO policy.

To restrain fragmentation, we also apply a general optimization. We keep the MAX_ORDER buddy list sorted by physical address, using neighbors address computation and recursive logic for fast operation (similar to buddy coalescing). This sorting prevents small random (4KB) page allocations (e.g., the fallback path for CA failures) from using scattered physical pages and fragmenting large free contiguous blocks. As discussed in Section VI, CA paging acts as a prevention strategy with respect to external fragmentation. Keeping an application's footprint coalesced and isolated from other processes reduces the fragmentation of the physical address space.

Avoiding multithreading pitfalls. In multithreaded applications, different threads may fault concurrently triggering parallel allocations for different virtual addresses. We use spin locks to protect CA paging VMA metadata updates. Nevertheless, concurrent allocations inside the same VMA stress the *Offset* selection of CA paging. For example, if two different threads fault for virtual addresses of the same sub-VMA region and both fail (target physical pages occupied), they will both trigger re-placement decisions. Without proper handling, this race results in multiple *Offset* updates for the same region and unnecessarily stresses the next-fit mechanism.

To handle such cases and support concurrent faults, CA paging allows only the first thread that enters the allocation path to trigger a re-placement and *Offset* update in case of a failure (with an atomic flag per VMA). If another thread fails, there are two options: (i) fallback to the default allocation, or (ii) retry until replacement is allowed or the allocation succeeds. We choose the latter to not penalize fault latency.

Supported faults. CA paging supports all anonymous and copy-on-write (4KB or 2MB) page faults, preserving demand paging. CA paging works also for the readahead allocations of the system's page cache, tracking an *Offset* attribute per file (`struct address_space`). Page cache mappings improve the performance of applications that use memory-mapped files. However, readahead allocations are usually interleaved with anonymous faults, as applications tend to read file data to populate heap structures. Moreover, page cache mappings tend to outlive processes, increasing the possibility to be reused. If they are scattered, they tend to fragment the physical address space. Instead, CA paging allocates them contiguously restraining fragmentation.

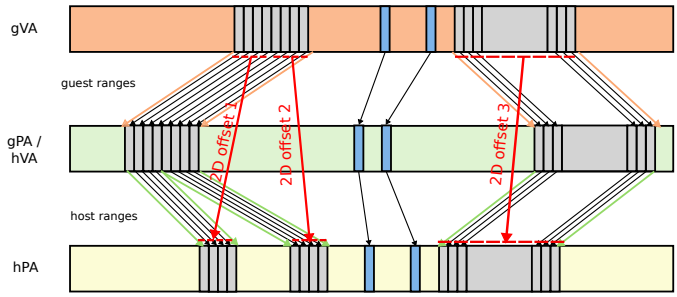


Fig. 5: Overview of unaligned arbitrarily-sized contiguous mappings (ranges) in virtualized execution. CA paging in the guest/host OS creates independently guest/host ranges while SpOT tracks the full 2D Offsets (red) to predict translations.

Virtualized execution. As CA paging is embedded in the OS memory management, it is applied in each dimension (guest/host) independently, elegantly enabling contiguous allocations to span the virtualization level and complying to nested paging. In the guest OS, it boosts the creation of gVA→gPA contiguous mappings across guest page faults (1^{st} dimension) and in the host, the creation of gPA→hPA across nested faults (2^{nd} dimension). A larger-than-a-page mapping is effectively contiguous, only if contiguous in both dimensions.

Independently using CA paging in each dimension creates such mappings on a best-effort basis. On a freshly booted virtual machine (VM), all guest page faults lead to nested faults as the guest physical pages are not mapped to host physical memory. In this early phase, CA paging is triggered in both dimensions consecutively. However, with nested paging the mappings of the second dimension (gPA→hPA) remain as long as the virtual machine is alive or until the host OS reclaims them. Thus, the 2^{nd} dimension contiguity persists as a VM ages, while the guest CA paging creates new 1^{st} dimension contiguous mappings for new applications running inside the VM. This leads to a less controlled generation of full 2D contiguous mappings, e.g., 1^{st} and 2^{nd} dimension mappings can be unaligned, smaller or larger with respect to each other (Figure 5). Our experiments indicate that CA though is still effective and creates significant 2D contiguity.

D. Discussion

VMA size. CA paging targets big-memory applications that suffer from high translation overheads. Such applications typically have a few large VMAs. If an application has multiple small VMAs, CA paging will inherently create multiple contiguous mappings due to the discontinuities in its virtual address space. Such applications may not benefit from the translation schemes that CA paging supports.

Reservation. Under severe memory pressure, different processes or VMAs may end up competing for the same scarce contiguous physical blocks. To shield contiguity, CA paging could employ reservation [15], [27]. In this paper we opt for best-effort strategies and consider reservation for future work.

IV. Hardware Technique: Speculative Offset-based Address Translation

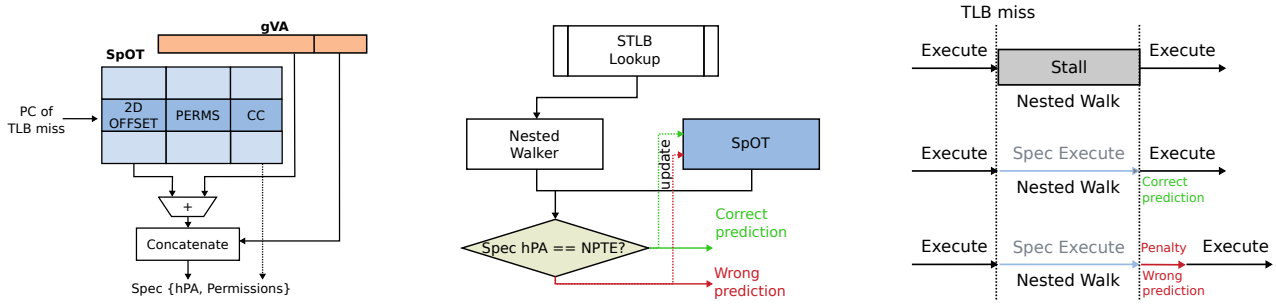
To exploit the contiguity of CA paging and improve application performance, we propose *Speculative Offset-Based Address translation (SpOT)*, a simple *micro-architectural* mechanism to predict missing address translations.

A. Motivation

Any address translation scheme that leverages contiguous mappings [11]–[16] benefits from CA paging (Table I). However, the goal of our work is to mitigate translation costs in the challenging setup of virtualization. In this scope, we find that the most prominent, high performant and paging compatible, proposals [11], [16], were originally proposed for native execution. Their complex design [11] or alignment restrictions [16], though, make them expensive or less effective for virtualization. In response, we propose SpOT, a micro-architectural speculation mechanism that predicts address translation with minimal and simple hardware support, sustains comparable performance, and supports both native and virtualized systems.

RMM [11] is extremely effective in capturing contiguity through range translations. However, RMM requires extensive architectural support analogous to and redundant to paging: a hardware range TLB per processor, OS-managed range tables per process, and hardware range table walkers. Virtualizing RMM (named as vRMM in this paper) would require a mechanism to traverse the ranges of both dimensions and retrieve a full 2D (gVA→hPA) range translation to be cached in the range TLB (Figure 5). A straightforward implementation of vRMM would add nested range tables per virtual machine and include hardware walkers to perform nested range walks. However, nested walking of the range tables is challenging as they are B-trees. Moreover, guest/host ranges would often mismatch in their size and alignment, i.e., one guest range may be backed by two or multiple host ranges. Therefore the nested walker should include logic to intersect guest and host ranges. All this additional overhead to an already complex and most importantly redundant design makes vRMM a less appealing design choice for adoption by processor vendors.

Hybrid coalescing [16], on the other hand, combines contiguous page translations into one translation entry, and augments TLBs to hold both coalesced and regular page translations. The coalesced entries are aligned at variable granularity (anchor distance). The OS stores the coalesced entries in modified page tables, and dynamically adjusts the anchor distance to reflect the process's average contiguity. Virtualizing hybrid coalescing (named as vHC in this paper) involves separate anchor distances for the guest and the host OS and therefore would require: (i) the hypervisor to maintain the host coalesced entries in the nested page tables, and (ii) an augmented nested page walker to intersect guest/host entries and calculate the 2D coalesced entry, respecting guest alignment. Even though the nested walk complexity increases, vHC requires simpler architectural support than vRMM. However, vHC suffers from its alignment restrictions. Table I shows the number of vRMM



(a) (b) (c)

Fig. 6: SpOT predicts the physical address of missing translations, inferring the offsets of contiguous mappings. It consists of a micro-architectural prediction table tracking the [offset,permissions] of recently missed translations (a), it is integrated in the L2 TLB miss path (b), and hides nested page walk latency under speculative execution (c).

ranges and vHC coalesced entries required to cover the 99% of big-memory workload's footprint in virtualized execution. We observe that CA successfully supports both techniques, significantly reducing the total number of entries for both methods compared to default THP. However, we observe that vHC fails to fully exploit the contiguity generated by CA as the anchor entries are 38× compared to ranges. This is due to the method's virtual alignment restrictions, confirming the important performance potential of unaligned contiguity.

Observation. We find that the root cause of vRMM's complexity and vHC's low performance potential is the requirement for explicit tracking of the mappings' virtual and physical boundaries. We pose the research question: *Can we have high performance translation leveraging unaligned contiguity of unlimited size with simpler hardware support?* Figure 5 shows the key idea of SpOT; instead of tracking mappings boundaries in guest and host, SpOT tracks only gVA→hPA *offsets* (red arrows) and uses them to predict missing address translations.

B. Overview of SpOT

We present SpOT in the context of virtualized execution as its operation in native execution can be inferred in a straightforward manner. SpOT works on the *micro-architectural* level and it primarily consists of a simple *prediction table* that caches *[2D offset, permissions]* translation tuples (Figure 6a). Each *offset* maps from gVA→hPA and is dynamically calculated and stored in the prediction table by the nested page walker (HW) at the end of the walk for a missing gVA translation. On a last level TLB miss, SpOT uses the *offset* generated by the previous TLB miss of the same memory instruction and predicts a host physical address (hPA). In a sense, SpOT speculates that the specific instruction is accessing a contiguously mapped range of pages and transparently tracks its corresponding offset to perform predictions. SpOT feeds the processor with the predicted hPA to continue execution in a speculative mode and the verification page walk happens in background. Thus, SpOT hides the latency of page walks.

	Mem. (GB)	default THP Ranges	default THP vHC entries	CA paging Ranges	CA paging vHC entries
SVM	29G	3759	6224	10	422
PageRank	78G	37453	39355	11	828
hashjoin	102G	4152	4260	7	403
XSBench	122G	4658	4968	11	644
BT				931	7061
geomean	-	7223	8485	23	914

TABLE I: Number of ranges (vRMM), and anchor entries (vHC) to map 99% of the footprint of big-memory workloads, using (i) default THP, (ii) CA paging in virtualized execution.

Speculating address translation has been proposed by SpecTLB [19] and Glue [20]. SpOT, albeit motivated by those designs, differs in some key ways. Their target is to predict the physical addresses of multiple reserved (SpecTLB) or splintered (Glue) base 4KB pages that belong to a huge 2MB page. Because such base pages are aligned with respect to their huge page boundaries, they are promoted to a single speculative huge page TLB translation entry. Hence, SpecTLB and Glue target to sustain huge page performance under different memory management conditions. We are the first, on the other hand, to leverage speculation to exploit unaligned, larger-than-a-(huge)-page contiguous mappings while completely avoiding the complexity of maintaining them in software (OS) and hardware. SpOT targets to predict translations far beyond the huge page limit and the mechanism is completely independent to virtual addressing and alignment. In a sense, SpOT builds on top of the idea of range translations without tracking them; instead it exploits instructions memory locality combined with inferred mappings contiguity to predict translations. Finally, SpOT's prediction mechanism bears similarities to SIPT [28] as they both use a PC indexed prediction table of offsets, but they target different problems and use different mechanisms. SIPT targets to speculatively index larger L1 data caches, predicts just a few (e.g., 1-3) bits of a physical address, and requires a complex perceptron confidence mechanism to throttle mispredictions. Instead, SpOT targets to predict the entire physical address translation to hide the cost of TLB misses without any complex confidence mechanism.

C. SpOT Mechanism

Prediction Table. To lookup and fill the prediction table with offsets, we use the program counter (PC) for indexing and tag matching. Only a few instructions are typically responsible for most TLB misses and therefore PC-indexing keeps the table

size small (64 entries in our experiments). It also serves the core idea of SpOT to correlate instructions with contiguous mappings. In the presence of a contiguity-aware allocator (CA paging) and thanks to locality, each memory instruction usually performs accesses (regular or irregular) inside a contiguously mapped range of pages at different execution phases.

TLB miss path. We consider address translation speculation only for the last-level TLB misses. Such misses trigger both the default nested page walk and a parallel lookup in SpOT's separate prediction engine (Figure 6b). SpOT uses the [offset,permissions] retrieved by the prediction table to predict a host physical address (hPA) translation for the guest virtual address (gVA) that caused the miss (Figure 6a). It subtracts the *offset* from the gVA and predicts a *spec* $hPA = gVA - offset$. It also speculates that this memory access will have the same permission rights as the previous access of the same instruction. Then it feeds the processor with the *spec hPA* to continue its execution but in speculative mode. It is worth mentioning that unlike previous designs [20], we do not fetch the predicted translation in the regular TLB hierarchy, leaving the TLB design intact.

As the speculative execution proceeds, the verification nested page walk happens in the background (Figure 6b, 6c). When the walk completes, the spec hPA is compared to the original hPA retrieved from the nested page table entry (NPTE) and two scenarios exist: (i) the speculation was *correct* and SpOT managed to hide the walk latency with useful speculative execution, (ii) the speculation was *incorrect* and SpOT must flush the pipeline and replay the memory instruction. Flush is necessary because following instructions may have consumed incorrect data. Incorrect predictions (mispredictions) affect performance, as the cost of flushing is added on top of the regular nested page walk latency.

Prediction table fills. The prediction table is updated at the end of a nested page walk. The offset of the missing translation is calculated ($offset = gVA - hPA$) and fetched in the table along with the permission rights of the access. To minimize possible PC conflicts, SpOT's prediction table is a set associative structure and uses an LRU replacement policy.

Building confidence. As mis-predictions restricts SpOT's effectiveness, we add a 2-bit saturating counter for each prediction entry. When an offset is firstly fetched into the prediction table the counter is set to 1. Correct predictions increase the counter by 1 and mis-predictions decrease it. Predicted physical addresses are fed to the processor only when the counter is >1. When the counter is $<=1$ no speculation is performed. Predictions, though, are still calculated and compared to the original hPA at the end of each nested page walk to update the confidence counters. Finally, an entry is replaced with a new offset only when the counter equals 0.

Preventing thrashing. To further boost SpOT accuracy, we involve the OS into filtering offsets with low prediction potential. If the prediction table is updated on every TLB miss, offsets that do not belong to large contiguous mappings may thrash it. Such offsets will never gain confidence to enable predictions and will evict valuable offsets from the table.

We mark translations that belong to larger contiguous mappings using a reserved bit in their corresponding page table entry (PTE), similar to [11]. In detail, the OS (CA paging) sets this bit at the end of a successful allocation (page fault) when updating the PTE. The OS checks if the neighboring PTEs have the bit set, i.e., belong to contiguous physical pages. If that bit is not set, the OS examines whether the last allocation extended a contiguous mapping beyond a *threshold* size. If so, the OS sets the bit to all PTEs that belong to that contiguous mapping, establishing its offset as candidate to trigger predictions. Even CA paging could dynamically adjust the threshold based on its contiguity statistics, we currently empirically set it to 32 contiguous pages. With nested paging, the guest and host OS sets the bit in the gPTs and nPTs, and the nested page walker updates the prediction table only if both bits are set. This optimization crosses the border of micro-architecture but we consider it a very simple and cheap mechanism. Note that even if the OS is involved, still the accurate size and boundaries of contiguous mappings are not calculated or tracked.

Hiding the verification page walk cost. With SpOT the page walk can be entirely or partially overlapped with useful work in case of correct prediction. Useful work can include prefetching data using the speculative address translation, overlapping the page walk with the data fetch cost [20], [29], [30]. In case the processor allows aggressive speculative execution, it can execute instructions that depend on the missing translation/data, increasing further performance opportunity.

D. Security Considerations

Speculation has been identified as a source of security vulnerabilities through cache side-channel attacks [31], [32]. Transient unsafe loads (USLs [32]) executed at a hardware mis-speculated control/data path can transmit secret data via micro-architectural covert channels before the mis-speculation is resolved. In example, USLs are the loads that are executed after branch predictions (Spectre attacks [33]) or the loads executed after exceptions (Meltdown attacks [34]). SpOT introduces a new unsafe memory instruction, i.e., load missing in the TLB, that an attacker can exploit to read data from unauthorized memory locations. The loads that follow a SpOT prediction are considered USLs until the prediction is verified. Fortunately, proposed mitigation techniques [21], [22], [35] for Spectre/Meltdown-type attacks also mitigate SpOT's vulnerabilities. Specifically, those techniques fetch and keep the data of USLs in a speculative buffer and do not commit their changes to the cache hierarchy until the loads are considered safe. Preventing USLs from changing the cache micro-architectural state effectively blocks all cache side channel attacks. Note that such mitigation techniques are necessary for secure execution regardless of SpOT's presence. Section VI-B discusses the performance impact of SpOT when such mitigation techniques are employed. Finally, SpOT does not speculatively change the TLB state, so no additional mitigation for MMU attacks [36] is required.

Native Environment	
Processors	2-socket Intel Xeon CPU E5-2630 v4 (Broadwell) 10 cores/socket, hyperthreading disabled, 2.2GHz
L1 DTLB	4K: 64-entry, 4-way set associative 2M: 32-entry, 4-way set associative
L2 DTLB	4K/2M: 1536-entry, 6-way set associative
Memory	256G (128G per socket)
OS	Debian Linux v4.19
Fully-Virtualized Environment	
VMM	QEMU (KVM) v2.1.2 20vCPUS 2-socket
Memory	256G (128G per socket)
Host/Guest OS	Debian Linux v4.19
Emulated hardware	
Direct Segment	Dual direct mode (single 2D segment)
vRMM	Range TLB: 32-entry, fully associative
SpOT	Prediction Table: 32-entry, 4-way set associative

TABLE II: System Configuration.

Workloads		
OpenMP (10 threads)	hashjoin microbenchmark	102G
	XSBench [39]	128G
Serial	Liblinear SVM [40], kdd12 dataset	29G
	Ligra PageRank [41], *friendster* graph [42]	78G
	BT (NPBe [43]) class E	167G

TABLE III: Workloads description and memory footprint.

Performance Model	
Ideal execution time	$T_{ideal} = T_{THP} - C_{THP}$
Native 4K/THP overhead	$O_{4K/THP} = C_{4K/THP}/T_{ideal}$
Virtual. 4K/THP overhead	$O_{v4K/vTHP} = C_{v4K/vTHP}/T_{ideal}$
vRMM overhead	$O_{vRMM} = (M_{SIM} * AvgC_{vTHP})/T_{ideal}$
DS overhead	$Over_{DS} = (M_{SIM} * AvgC_{v4K})/T_{ideal}$
SpOT overhead	$O_{SpOT} = ((NP_{SIM} * AvgC_{vTHP}) + MP_{SIM} * (AvgC_{vTHP} + MP_{penalty}))/T_{ideal}$
T: Total execution cycles	AvgC: average cost of page walk
C: Cycles spent in page walks	M_{SIM}: Simulated page walks
$MP_{penalty}$: 20 cycles	MP_{SIM}: Simulated mispredictions
v4K/vTHP: 4K+4K/THP+THP	NP_{SIM}: Simulated no predictions

TABLE IV: Performance model based on hardware performance counters and hardware emulation with BadgerTrap [38].

V. Methodology

OS prototype and server machine. We prototype CA paging in Linux v4.19 for anonymous/copy-on-write page faults and page cache allocations. We use Qemu/KVM v2.1.2 for virtualized execution. Our code and scripts are publicly available on GitHub[1]. Table II summarizes the configuration details of our experimentation system. In our study, we focus only on the costly L2 STLB misses that trigger page walks; hence, we refer to those as TLB misses for simplicity.

Contiguity results. We collect statistics for contiguous mappings through page table information. We use the standard `pagemap` [37] API for native execution, and we develop an in-house virtual-machine introspection (VMI) tool with similar functionality for virtualized execution. For the latter, the guest OS exposes the application's guest page table to the host (registering its location in the guest physical address space), and then the host reads and combines the guest and the nested page tables info to calculate a full 2D translation.

Hardware emulation. We emulate various hardware address translation schemes by instrumenting the TLB misses that trigger page walks in our real system as applications run with BadgerTrap [38] in the guest OS. BadgerTrap uses page table marking to force TLB misses to cause page faults and enables hardware emulation in special fault handlers. For SpOT, we use a 4-way set associative prediction table of 32 entries. For Direct Segments (DS), we use the dual direct mode [10] that allows direct 2D gVA→hPA translation through a single direct segment. For vRMM, we use a fully-associative range TLB of 32 entries and we implement the guest and nested range tables as flat arrays, rather than B-trees. To identify the boundaries of 2D translations inside the guest OS, we expose the nested range table to the guest at a reserved guest physical address area, using the standard nested page tables.

Performance model. We collect statistics from performance counters with *perf* (CPU cycles, TLB misses, page walk cycles) to quantify virtual memory overhead. In more detail, we use PAPI [44] commands injected in the benchmarks code to exclude their initialization phase. To keep a common baseline, we adopt the same methodology of prior works [9]–[11], [14], [20]. We identify the ideal execution time of zero address translation overhead (T_{ideal}) and then compare all measured and simulated overheads to the ideal execution time using a simple linear performance model. For vRMM, we assume that the latency of the nested range table walk is hidden entirely in the background. For SpOT, we assume that: (i) correct speculations hide the entire TLB miss cost, (ii) decisions to not apply speculation expose the entire TLB miss cost, and (iii) mis-speculations add extra 20 cycles for flushing the pipeline [20] on top of the TLB miss cost. For DS, we assume the dual direct mode that provides gVA→hPA address translation. Table IV summarizes how we compute virtual memory overheads for the various configurations.

Workloads. We use a set of memory/TLB intensive workloads, single- and multi- threaded, from graph analytics, high performance, and machine learning domains (Table III). Note that we run PageRank with a single thread to enable comparison with Translation Ranger [18] as multi-threaded execution was erroneous. CA paging results remain similar for both the single- and multi-threaded version.

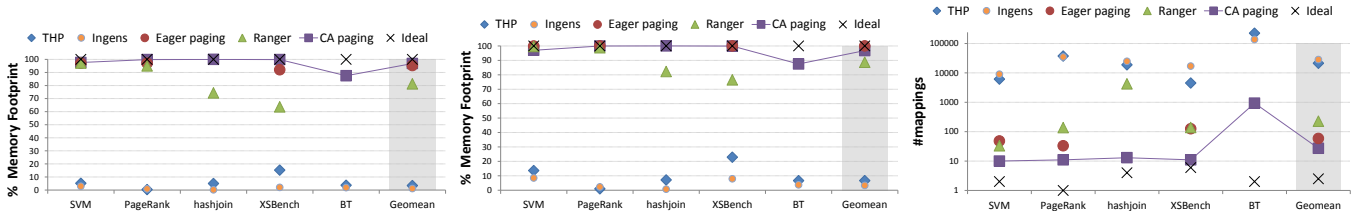
VI. Results

We first evaluate the impact of CA paging on the creation of contiguous mappings in both native and virtualized environments. We then evaluate SpOT that exploits the contiguity of CA paging to mitigate the address translation overhead in virtualized execution.

A. Contiguity-aware Paging

We compare *CA paging* with: (i) *default paging–THP*, the default OS technique that supports transparent 2M allocations,

[1]https://github.com/cslab-ntua/contiguity-isca2020.git



(a) Coverage with 32 largest mappings (b) Coverage with 128 largest mappings (c) #mappings to cover 99% of memory

Fig. 7: Contiguity performance without memory pressure for native execution.

(ii) *Ingens* [4], a transparent huge page management framework that performs asynchronous huge page promotions, (iii) *eager paging* [11] that increases the kernel MAX_ORDER attribute to allow the buddy allocator to maintain larger blocks and uses them to perform pre-allocations, (iv) *translation ranger* [18] that coalesces application's memory footprint asynchronously using post-allocation page migrations, and (v) *ideal paging* that applies an offline best-fit algorithm to find the maximum contiguity that could be provided based on the *contiguity_map*'s state before execution. As CA paging is applicable to native and virtualized execution (Section III), we first present extensive native results to allow comparison with the other techniques. We summarize virtualized execution performance at the end of this section.

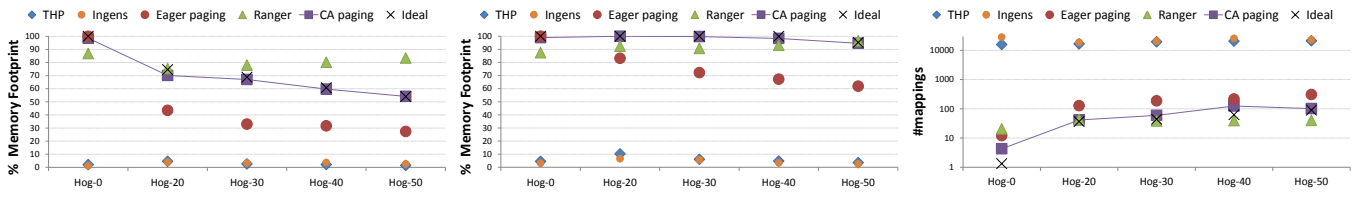
To evaluate the impact on the virtual-to-physical mapping contiguity, we use the memory footprint coverage of the 32 and 128 largest mappings (higher is better) and the average number of mappings required to cover 99% of the total footprint (lower is better), averaged throughout application's execution time [18]. For all configurations, we use a modified TCMalloc [45], that increases maximum allocation as proposed for eager paging [11]. Note that, CA paging and ranger are independent to the user space allocator. We did CA paging experiments with standard libc and the results remain unchanged.

Contiguity in the absence of memory pressure. Figure 7 summarizes the contiguity results when applications execute natively on a machine without external fragmentation. Both THP and Ingens perform similarly, generating thousands of non contiguous mappings to cover applications footprint. This is expected behavior as both techniques control and manage contiguity up to 2MB (huge page). CA paging generates contiguity comparable to that of eager paging and improved compared to translation ranger, avoiding pre-allocations and page migrations. It covers on average 99% of applications footprint with ∼27 mappings, orders of magnitude less than default paging. The effectiveness of translation ranger for XSBench and hashjoin decreases, as their allocation phase is significant compared to total execution, and post-allocation migrations takes time to fully coalesce their footprints (Figure 1c). CA paging performance drops for the BT workload as irregular faults compete for the last contiguous free blocks of the first NUMA node, right before it spans to the second. We plan to study this side-effect in the future.

Note that we exclude hashjoin from eager paging results and BT from both eager paging and translation ranger results. The two benchmarks, either due to memory bloat (hashjoin) or own requirements (BT), span over two NUMA nodes and those techniques currently do not support NUMA topologies.

Fragmentation Impact. To profile external fragmentation impact we use a "hog" micro-benchmark [4], [12], [13]. Due to the increased memory pressure, all workloads footprint span two NUMA nodes as there is not enough free memory in a single node to cover them. For that reason, we turn NUMA off, via Linux kernel boot parameters, to enable comparison of CA paging with the other techniques. Figure 8 summarizes the geometric mean contiguity results for all benchmarks when memory pressure increases from 0% to 50%. We exclude BT as its 167G footprint does not fit in the "hogged" memory.

Both THP and Ingens perform poorly and similar to the no memory pressure case. This is expected as our hogging micro-benchmark fragments physical memory in coarse granularities (>2MB) and thus, there are plenty of free huge pages to back benchmarks' footprints. CA paging is fairly robust, outperforming eager paging. It covers ∼94% of the footprints with only 128 mappings under maximum pressure (hog-50) and always follows Ideal paging (with small deviations). Therefore, CA paging manages to fully exploit the available unaligned free contiguity in the system. On the other hand, eager paging is highly sensitive to fragmentation due to alignment restrictions. It relies on buddy allocator's higher order blocks and the allocator tracks only aligned contiguous blocks. Finally, translation ranger remains almost unaffected by the increasing memory pressure, outperforming all allocation techniques in 32 mappings coverage (better than Ideal paging), as it relies on post-allocation migrations. CA paging, however, achieves similar performance with respect to 128 mappings and 99% coverage. Generally, we consider the two approaches orthogonal and mutually assisted; CA paging



(a) Coverage with 32 largest mappings (b) Coverage with 128 largest mappings (c) #mappings to cover 99% of memory

Fig. 8: Contiguity performance under memory pressure/external fragmentation. Geomean results for all benchmarks.

TABLE V: Total number of page faults and 99th latency (us).

99th latency (us)			Total Number of page faults		
THP	CA paging	Eager paging	THP	CA paging	Eager paging
515	526	80372	45148	45148	67

TABLE VI: Bloat [memory (overhead%)] compared to 4KB.

	SVM	PageRank	hashjoin	XSBench	BT
THP (MB)	13.3(0.0%)	5.2(0.0%)	3.8(0.0%)	4.7(0.0%)	136(0.1%)
Ingens (MB)	1.4(0.0%)	3.3(0.0%)	0.4(0.0%)	1.4(0.0%)	89(0.0%)
CA (MB)	13.1(0.0%)	6.8(0.0%)	3.3(0.0%)	6.2(0.0%)	137(0.1%)
eager (GB)	2.3(8.0%)	5(6.5%)	48(47.5%)	0.5(0.4%)	0.1(0.1%)

can generate early-on contiguity, and if required, ranger's migrations can further boost it, similarly to how khugepaged [24] complements THP allocations.

Fragmentation restraint. Previously we evaluated contiguity on an already fragmented machine; CA paging, though, can delay fragmentation as a machine ages. Figure 9 depicts the distribution of the unaligned free block sizes after a set of benchmarks runs to completion using default and CA paging. We notice that a significantly larger portion of free memory is backed by >1GB blocks. This is attributed to the allocation (and consecutive release) of contiguous pages and to the long-lived contiguous page cache mappings (Section III).

Multi-programmed case. Figure 10 depicts contiguity results while running two instances of the SVM workload without fragmentation. CA paging provides increased contiguity, avoiding eager pre-allocations, as the next fit placement policy successfully prevents workloads interference over the same free blocks. Translation ranger fails to coalesce the two footprints, migrating pages between them across the entire execution. Note that the code released for ranger is not optimized to serve multiple processes. Multi-programmed workloads require ranger to scan serially all processes' footprint at every defragmentation epoch, penalizing its response time.

Software Overhead Analysis. We evaluate the isolated software overheads of the different mechanisms when there is no gain from novel larger-than-a-page address translation schemes. Figure 11 depicts the normalized execution time of benchmarks running on our commodity hardware. Hashjoin does not run with eager paging as the benchmark spans to two NUMA nodes with this method and this is not supported. Translation ranger penalty is ∼3% on average due to page migrations. Eager and CA paging add no overhead. We also run a set of TLB friendly workloads from Spec2017 and find that the execution time is not affected by CA paging. However, the two paging methods behave differently with respect to

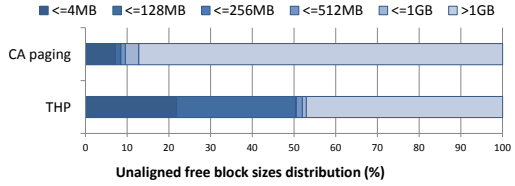


Fig. 9: Free block size distribution after benchmarks execution.

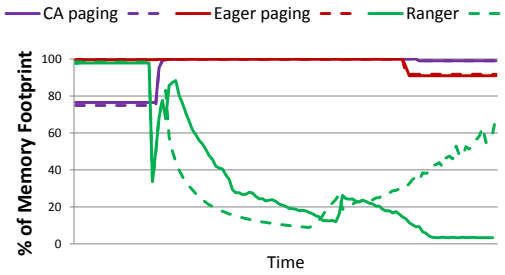


Fig. 10: 32 largest mappings coverage while running two instances of SVM (straight and dotted line for every method).

tail latency and resource utilization. Table V summarizes the number of page faults and their average latency (us) measured for all benchmarks with ftrace [46]. CA paging does not affect latency while eager paging magnifies it due to zeroing large blocks. The latter though decreases the total number of faults.

Finally, Table VI summarizes the extra memory allocated by the different techniques compared to demand paging (bloat) with 4K pages. We observe that CA paging and THP perform the same (bloat up to 136MB), as CA paging builds on top of THP and does not affect the page size decision. Ingens, on the other hand, decreases it as it asynchronously promotes 4K pages to huge based on utilization. Note that CA paging can add mechanisms from Ingens to boost contiguity while preserving the low internal fragmentation that Ingens offers. We plan to study this combination for future work. Finally, eager pre-allocation suffers the most as it leads to occupation of multiple GBs that the application will not eventually use.

Virtualized execution. Figure 12 summarizes the results for virtualized execution. We employ CA paging in both guest and host OS independently, without any form of coordination, and measure the 2D gVA→hPA mappings contiguity. On average, CA paging decreases the number of mappings required for 99% coverage by an order of magnitude (∼90) compared to default paging and covers ∼86%/∼96% with 32/128 mappings. However, we observe that 32 mappings coverage is slightly worst compared to native execution. This is expected as the contiguous mappings in the guest and host dimensions are created independently and on a best-effort basis. Note also that our applications run consecutively without VM reboots. Therefore, unaligned mismatches between the guest and the host contiguous mappings are more frequent, as the gPA-to-hPA mappings persist across benchmarks runs (Section III).

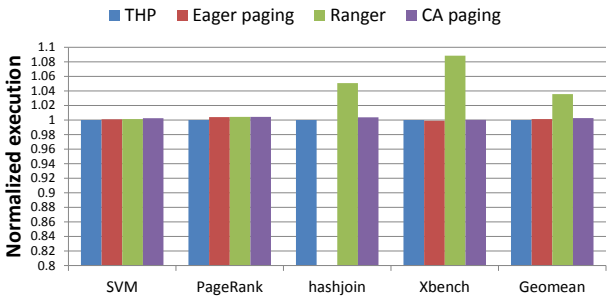
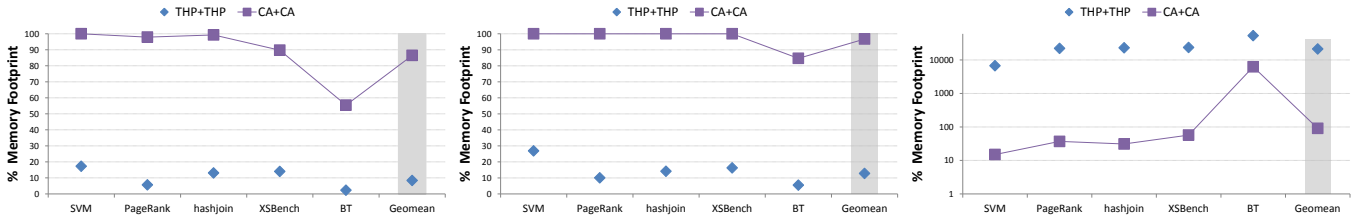


Fig. 11: Software runtime overheads normalized to THP.



(a) Coverage with 32 largest mappings (b) Coverage with 128 largest mappings (c) #mappings to cover 99% of memory

Fig. 12: Contiguity performance without memory pressure for virtualized execution.

B. SpOT

We now quantify the execution overhead of address translation and evaluate SpOT in virtualized execution. Figure 13 summarizes our findings. We use performance counters to measure the translation penalty in native (blue hashed bars) and virtualized (blue solid bars) execution with base pages (4K, 4K+4K) and Transparent Huge Pages (THP, THP+THP) and emulate the performance of SpOT, vRMM and DS [10].

Paging Overheads. For native execution, our results corroborate all past studies indicating that address translation overheads are exceptionally high with 4K pages. Overheads above 100% are due to overlapped page walk cycles and comparison with the ideal baseline. THP reduces substantially the overhead but fails to eliminate it, bringing it to ∼7% on average and up to ∼13% for SVM. Note >99% of the memory footprint of the workloads is mapped with 2M pages. In virtualized execution, the address translation overhead is magnified due to nested page walks. Even with THP on, it grows to ∼16.5% on average and up to ∼28% for SVM.

SpOT Performance. For the evaluation of SpOT we apply CA paging in both the guest and host OS. SpOT reduces the overhead to ∼0.85% on average. The performance improves significantly for all applications, but less for SVM and BT. For BT, CA paging (Figure 7a,12a)) fails to provide optimal contiguity when the application expands to the second NUMA node as discussed earlier in this section. For SVM, despite CA paging successfully maps 99% of the application's footprint with less than 32 mappings (Figure 12a), a portion of the observed TLB misses (∼4%) are on a few virtual addresses that fall outside these mappings. SVM has also high number of irregular TLB misses triggered by the same instruction. SpOT sensitivity to the access pattern is highly exposed by the hashjoin micro-benchmark as well. Note however that hashjoin makes random accesses. To better understand SpOT performance, Figure 14 breaks down the percentage of TLB misses predicted correctly, mis-predicted, and not predicted at all. We observe that correct predictions can be over 99% (PageRank), while mis-predictions never more than 4% (hashjoin).

Comparison with RMM and Direct Segments. For vRMM we use CA paging in both the guest and host OS. Note that vRMM was proposed with eager paging [11] pre-allocation. We observe that vRMM with CA paging reduces the translation overhead to less than 0.1%. It performs slightly worse for SVM and BT due to CA paging stressing, discussed also in the previous paragraph. However, RMM requires architectural support that (as discussed in Section IV) increases significantly in virtualized execution. Finally, we compare SpOT with direct segments (DS) dual mode [10]. We observe that DS eliminate TLB miss penalty. Despite its prominent efficiency, the method is rigid, reserving the segment when a virtual machine boots and abolishing paging. SpOT, combined with CA paging, preserves the benefits of demand paging sustaining high address translation performance comparable to DS.

Security mitigation techniques discussion. As discussed in Section IV, SpOT can be exploited to leak data from unauthorized memory locations through cache side-channel attacks. Fortunately, proposed Spectre/Meltdown mitigation techniques [21], [22], [35] can also mitigate SpOT vulnerabilities. However, such techniques introduce performance overheads proportionate to the number of Unsafe Loads (USLs) [22], i.e., loads that are executed in speculative state. Studying accurately the impact of SpOT USLs requires full

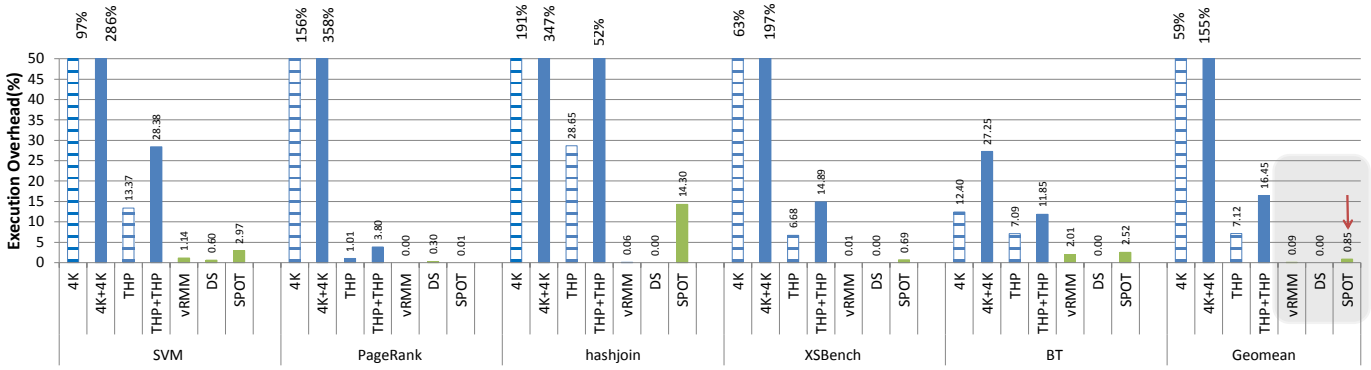


Fig. 13: Execution time overheads due to data TLB misses that trigger page walks in virtualized execution.

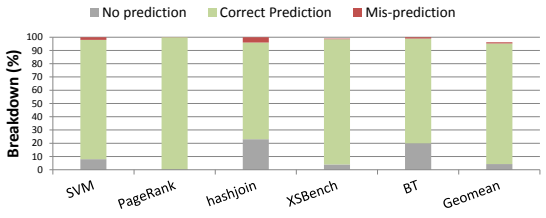


Fig. 14: Percentage of TLB misses that SpOT made (i) correct predictions, (ii) mispredictions, and (iii) no predictions.

system cycle-accurate simulation that is prohibitively slow for our TLB studies. However, we make some rough estimations for the number of SpOT USLs and their impact on performance, assuming the InvisiSpec design [22], [35]. We use performance counters to measure the number of TLB misses, loads, cycles, and the average latency of page walk, and we calculate the number of SpOT USLs (Equation 2 in Table VII). To put our results into perspective, we also measure the number of branches, and we compare the number of SpOT USLs with the number of Spectre USLs, i.e., unsafe loads that are executed due to branch predictions (Equation 1 in Table VII). We assume a linear distribution of load instructions over time. Table VII summarizes the results for all our workloads (geometric mean). We observe that the events that trigger speculative execution with SpOT, i.e., TLB misses, are a small fraction (0.25%) compared to Spectre's branch predictions (5%). However, SpOT's transient window of speculative execution is much larger (the average page walk latency is $\sim$81 cycles in our experiments) compared to branch resolution ($\sim$20 cycles [20]). In total, $\sim$3% of total instructions would execute as USLs with SpOT, whereas the percentage of USLs with Spectre would be $\sim$16%. As InvisiSpec for mitigating Spectre USLs has been shown to add $\sim$5% overhead [35], we expect that extending InvisiSpec for SpOT USLs would introduce $< 2\%$ overhead. Hence, SpOT's performance translation benefits would remain still beneficial.

C. Summary

CA paging significantly boosts the creation of contiguous mappings in native and virtualized execution. It is more robust compared to pre-allocation while preserving demand paging. It also performs closely to ranger [18] avoiding page migrations. We consider the two approaches mutually assisted and their combination a good strategy to shield contiguity against external fragmentation. On the hardware level, SpOT successfully exploits CA paging's contiguity and reduces significantly the translation overhead from $\sim$16.5% to $\sim$0.9% requiring minimal micro-architectural support.

Branches/ Instructions(%)	DTLB misses/ Instructions(%)	Spectre USL/ Instructions(%)	SpOT USL/ Instructions(%)
5.87	0.25	16.5	2.9

Spectre USL = #Branches * Branch Resolution Cycles * Loads/cycle (1)

Spot USL = #DTLB misses * Page Walk Cycles * Loads/cycle (2)

TABLE VII: Estimation of Unsafe Load Instructions (USL).

VII. Other Related Work

Memory Management. Multiple software proposals [4], [5], [7], [27] improve huge page management, addressing issues like fairness, memory bloat, increased tail latency, and fragmentation. Instead, CA paging targets the reduction of translation overheads that persist in the presence of huge pages, and builds on top of huge page management to create larger-than-a-page contiguous mappings for novel translation hardware. Other proposals control external fragmentation [6], [47] again in the scope of huge pages, focusing on the allocation [6] and the reclamation [47] OS routines. In contrast, we study fragmentation in coarser granularities and show that contiguous allocation beyond the page size can delay fragmentation.

Address Translation Hardware. Bhargava et al. [1] analyzed nested paging translation overhead and proposed MMU caching and large page sizes. Our experiments show that such support–that is present in commodity processors–is not sufficient, as the address translation overhead still remains significant. Other works have focused on the implications of huge pages and have proposed specialized hardware to support them better [15], [48]–[54]. Still, those designs provide limited TLB reach and suffer from alignment issues. SpOT harvests unaligned contiguity to hide the page walk latency.

Multiple works [55]–[57] combine shadow and nested paging to minimize the MMU virtualization overhead. Our evaluation focuses on nested paging, the state-of-practice virtualization technique, but both CA paging and SpOT are agnostic to the virtualization technology and directly applicable to shadow and hybrid paging. Ahn et al. [58] proposed an inverted shadow page table combined with a flat nested page table, and used speculative execution to relax the synchronization between the tables. That design modified paging subsystem extensively. Instead, our approach is completely compatible with paging and requires minimal micro-architectural support.

DVM [30] introduces regions for which the virtual address equals the physical address (identity mappings) and caches only the translation permissions. An optional enhancement speculates whether a mapping is identity. DVM restricts the flexibility of common OS mechanisms, e.g., copy-on-write and fork. In contrast, our approach is compatible with such mechanisms and SpOT predicts translations without any virtual or physical special address requirements.

Several mechanisms reduce the cost of page walks either targeting alternative page table representations [59]–[61], enhanced MMU caches [62], [63], direct page table indexing [64], or page table replication [65]. SpOT is orthogonal as it hides page walk latency under speculative execution. TLB prefetching can also reduce TLB misses by predicting the next missing translation [66]–[68]. Instead, SpOT predicts the actual address translation itself.

Finally, prior works propose: (i) storing TLB data as part of the memory subsystem [69], [70], (ii) pinning frequently accessed pages with poor temporal locality to reduce the number of TLB misses [71], (iii) modifying TLBs to better accommodate chip multiprocessors [72]–[74] and (iv) reducing

TLB shootdown overheads through hardware [75]–[78] or OS [25], [26], [79] optimizations. Our approach is orthogonal to those mechanisms.

VIII. Summary

We propose complementary software and hardware methods to mitigate the address translation overhead, focusing on the challenging setup of nested paging. On the OS level, we propose CA paging to generate vast mapping contiguity across page fault allocations. On the hardware side, we propose SpOT to predict translations in the TLB miss path. Combined with CA paging, SpOT significantly reduces the translation overhead of nested paging from ∼16.5% to ∼0.9%.

Acknowledgements

We would like to thank our anonymous reviewers, Michael Swift, Wisconsin Multifacet research group, Dionisios Pnevmatikatos, Nikela Papadopoulou, and all members of Computing Systems Laboratory at NTUA for their valuable feedback.

References

[1] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[2] "5-level paging and 5-level ept white paper," Intel, Tech. Rep., 2017.

[3] "Intel® Xeon® Processor E5-2600 V4 Product Family Technical Overview," 2016.

[4] 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.

[5] 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.

[6] A. Panwar, A. Prasad, and K. Gopinath, "Making Huge Pages Actually Useful," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[7] A. Panwar, S. Bansal, and K. Gopinath, "HawkEye: Efficient Fine-grained OS Support for Huge Pages," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[8] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble Page Management for Tiered Memory Systems," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[9] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[10] 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 Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[11] 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 Annual International Symposium on Computer Architecture*, 2015.

[12] 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.

[13] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, 2014.

[14] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[15] M. Talluri and M. D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[16] 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 Annual International Symposium on Computer Architecture*, 2017.

[17] A. Bhattacharjee, "Preserving Virtual Memory by Mitigating the Address Translation Wall," *IEEE Micro*, vol. 37, no. 5, Sep. 2017.

[18] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation Ranger: Operating System Support for Contiguity-aware TLBs," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[19] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[20] 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 48th International Symposium on Microarchitecture*, 2015.

[21] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," in *Proceedings of the 56th Annual Design Automation Conference*, 2019.

[22] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[23] T. Merrifield and H. R. Taheri, "Performance Implications of Extended Page Tables on Virtualized X86 Processors," in *Proceedings of The 12th International Conference on Virtual Execution Environments*, 2016.

[24] "Transparent Huge Pages in 2.6.38," http://lwn.net/Articles/423584/.

[25] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy Translation Coherence," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[26] N. Amit, "Optimizing the TLB Shootdown Algorithm with Page Access Tracking," in *Proceedings of the USENIX Annual Technical Conference*, 2017.

[27] J. Navarro, S. Iyer, P. Druschel, and A. L. Cox, "Practical, Transparent Operating System Support for Superpages," in *Proceedings of the 5th Symposium on Operating System Design and Implementation*, 2002.

[28] T. Zheng, H. Zhu, and M. Erez, "SIPT: Speculatively Indexed, Physically Tagged Caches," in *IEEE International Symposium on High Performance Computer Architecture*, 2018.

[29] A. Bhattacharjee, "Translation-Triggered Prefetching," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[30] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing Memory in Heterogeneous Systems," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[31] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[32] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *Proceedings of the 27th USENIX Conference on Security Symposium*, 2018.

[33] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019.

[34] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[35] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[36] S. van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[37] "Pagemap utility," https://www.kernel.org/doc/Documentation/vm/pagemap.txt.

[38] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "BadgerTrap: A Tool to Instrument x86-64 TLB Misses," *SIGARCH Comput. Archit. News*, vol. 42, no. 2, Sep. 2014.

[39] 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.

[40] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *Journal of Machine Learning Research*, vol. 9, 2008.

[41] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[42] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[43] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks summary and preliminary results," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1991.

[44] D. Terpstra, H. Jagode, H. You, and J. J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing*, 2009.

[45] "TCMalloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html,.

[46] "Linux ftrace utility," https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[47] M. Gorman and P. Healy, "Supporting Superpage Allocation Without Additional Hardware Support," in *Proceedings of the 7th International Symposium on Memory Management*, 2008.

[48] C. 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 IEEE 47th International Symposium on High Performance Computer Architecture*, 2020.

[49] Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee, "Reevaluating Online Superpage Promotion with Hardware Support," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[50] N. Ganapathy and C. Schimmel, "General Purpose Operating System Support for Multiple Page Sizes," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 1998.

[51] A. Seznec, "Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB," *IEEE Trans. Comput.*, vol. 53, no. 7, Jul. 2004.

[52] M. Swanson, L. Stoller, and J. Carter, "Increasing TLB Reach Using Superpages Backed by Shadow Memory," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[53] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, 2015.

[54] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs," in *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, 2015.

[55] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile Paging: Exceeding the Best of Nested and Shadow Paging," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[56] Y. Zhang, R. Oertel, and W. Rehm, "Paging Method Switching for QEMU-KVM Guest Machine," in *Proceedings of the International Conference on Big Data Science and Computing*, 2014.

[57] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li, "Selective Hardware/Software Memory Virtualization," in *Proceedings of the 7th ACM International Conference on Virtual Execution Environments*, 2011.

[58] J. Ahn, S. Jin, and J. Huh, "Revisiting Hardware-assisted Page Walks for Virtualized Systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[59] I. Yaniv and D. Tsafrir, "Hash, Don'T Cache (the Page Table)," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, 2016.

[60] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-It-Yourself Virtual Memory Translation," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[61] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[62] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don'T Walk (the Page Table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[63] A. Bhattacharjee, "Large-reach Memory Management Unit Caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[64] 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.

[65] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, "Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[66] A. Bhattacharjee and M. Martonosi, "Inter-core Cooperative TLB for Chip Multiprocessors," in *Proceedings of the 15th Annual Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[67] G. B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-driven Study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[68] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based TLB Preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[69] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[70] 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.

[71] H. Elnawawy, R. B. R. Chowdhury, A. Awad, and G. T. Byrd, "Diligent TLBs: A Mechanism for Exploiting Heterogeneity in TLB Miss Behavior," in *Proceedings of the ACM International Conference on Supercomputing*, 2019.

[72] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors," in *Proceedings of the 43rd Annual International Symposium on Microarchitecture*, 2010.

[73] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-level TLBs for Chip Multiprocessors," in *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[74] L. Zhang, E. Speight, R. Rajamony, and J. Lin, "Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation," in *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010.

[75] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all," in *Proceedings of the the 16th International Symposium on High-Performance Computer Architecture*, 2010.

[76] Z. Yan, J. Veselý, G. Cox, and A. Bhattacharjee, "Hardware Translation Coherence for Virtualized Systems," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[77] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[78] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries," in *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*, 2017.

[79] N. Amit, A. Tai, and M. Wei, "Don't Shoot down TLB Shootdowns!" in *Proceedings of the 15th European Conference on Computer Systems*, 2020.