Translation Ranger: Operating System Support for Contiguity-Aware TLBs

Zi Yan
Rutgers University & NVIDIA
ziy@nvidia.com

Daniel Lustig
NVIDIA
dlustig@nvidia.com

David Nellans
NVIDIA
dnellans@nvidia.com

Abhishek Bhattacharjee
Yale University
abhishek@cs.yale.edu

ABSTRACT

Virtual memory (VM) eases programming effort but can suffer from high address translation overheads. Architects have traditionally coped by increasing Translation Lookaside Buffer (TLB) capacity; this approach, however, requires considerable hardware resources. One promising alternative is to rely on software-generated translation contiguity to compress page translation encodings within the TLB. To enable this, operating systems (OSes) have to assign spatially-adjacent groups of physical frames to contiguous groups of virtual pages, as doing so allows compression or *coalescing* of these contiguous translations in hardware. Unfortunately, modern OSes do not currently guarantee translation contiguity in many real-world scenarios; as systems remain online for long periods of time, their memory can and does become fragmented.

We propose Translation Ranger, an OS service that recovers lost translation contiguity even where previous contiguity-generation proposals struggle with memory fragmentation. Translation Ranger increases contiguity by actively coalescing scattered physical frames into contiguous regions and can be leveraged by any contiguity-aware TLB without requiring changes to applications. We implement and evaluate Translation Ranger in Linux on real hardware and find that it generates contiguous memory regions 40× larger than the Linux default configuration, permitting TLB coverage of 120GB memory with typically no more than 128 contiguous translation regions. This is achieved with less than 2% run time overhead, a number that is outweighed by the TLB coverage improvements that Translation Ranger provides.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Virtual memory**.

KEYWORDS

Translation Lookaside Buffers; Memory defragmentation; Operating system; Heterogeneous memory management

ACM Reference Format:
Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-Aware TLBs. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322223

1 INTRODUCTION

Virtual memory (VM) eases programming in many ways. It abstracts the complexity of physical memory, provides memory protection and process isolation, and facilitates communication between cores and/or compute units through a shared virtual address space. Virtual memory is used today not just for CPUs but also increasingly for GPUs and other accelerators [3, 19, 47]. Figure 1 shows an example system with a group of CPUs, GPUs, and other accelerators, where each type of device can directly access both its own memory as well the memory of other devices via the virtual memory system.

High performance virtual memory can be achieved by embedding a Translation Lookaside Buffer (TLB) in each computation unit to cover all physical memory. However, since covering all of physical memory would require considerable translation storage overheads [18, 45], vendors today choose to implement TLBs that cover only a portion of the total memory space. For example, Intel has been (approximately) doubling its CPU TLB resources every generation from Sandybridge through Skylake [20], resulting in TLBs with thousands of entries today. Vendors like AMD implement even larger TLBs for their GPUs [28, 49], but these large TLBs consume non-trivial area and power [5, 12, 24, 36] and are ill-suited for other types of accelerators with limited hardware resources [18, 41, 45].

Recent studies focusing on the address translation wall [7] propose using *translation contiguity* to mitigate VM overheads [4, 9, 23, 37, 39]. A contiguous region maps a set of contiguous virtual pages to a corresponding group of spatially-adjacent physical frames. Contiguous regions are desirable for emerging TLB designs that can compress these translations into a single TLB entry. Ideally, a contiguous region of *N* pages can be stored with just a single TLB entry rather than *N* entries. Figure 2 shows the general concept of contiguity [9, 39], and is representative of several aggressive proposals to exploit contiguity with range TLBs [23], devirtualizing memory [18], and direct segments [4]. These approaches, combined

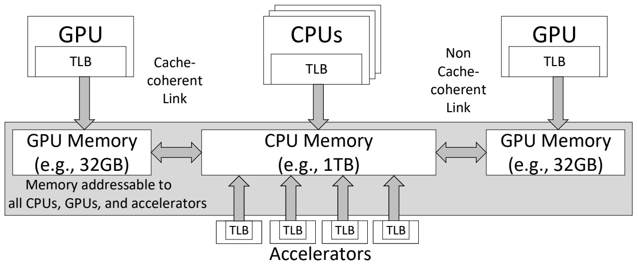


Figure 1: Hypothetical system comprised of CPUs, GPUs, and other accelerators utilizing a single shared virtual and physical address space.

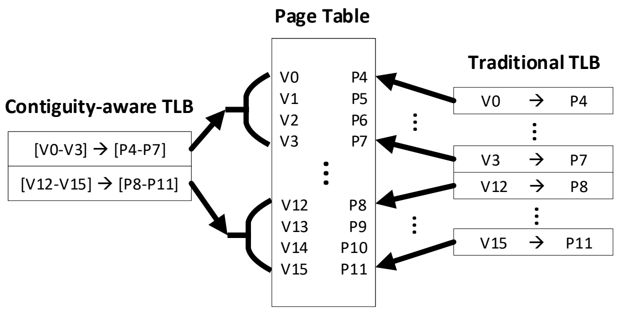


Figure 2: A contiguity-aware TLB (left) uses two entries to cache four translations each, while a traditional TLB (right) requires eight entries.

with traditional huge page techniques [32, 40, 46], may be able to drive address translation overheads to near zero in future systems.

Although influential, existing translation contiguity proposals typically require either specific amounts of contiguity (e.g., discrete page-sized contiguity for huge pages [12, 35, 44]), restricted types of contiguity (e.g. where virtual and physical pages must be identity mapped [18]), or serendipitously-generated contiguity (e.g., TLB coalescing [39]). Others still require large swaths of contiguity that can be created only at memory (pre)allocation time (e.g., direct segments and ranges [4, 23]). The question of how OSes can actively generate *unrestricted and general-purpose contiguity* from any arbitrary starting condition (e.g., after active use) remains open.

Our goal is to develop OS support for creating general-purpose translation contiguity in a robust manner across all execution environments. This means that unlike prior work [4, 18], we cannot generate contiguity only during initial memory allocations; we must also generate it throughout the workload's lifetime. It means that we must be able to generate contiguity on real-world systems with long uptimes, where memory may be (heavily) fragmented by diverse workloads that are spawned and terminated over time. Further, we cannot not rely on OS/application customization via mechanisms like identity mappings [18] or programmer-specified segments [4] which can preclude important OS features like copy-on-write or paging to disk, and may affect security features like address space layout randomization (ASLR) [18].

To achieve this, we propose Translation Ranger, a new OS service that *actively coalesces fragmented pages* from the same virtually contiguous range to generate unrestricted amounts of physical memory contiguity in realistic execution scenarios. This enables any previously proposed TLB optimization—e.g., from COLT [9, 12, 39], to direct segments [4], to Range TLB [23], to hybrid coalescing TLBs [37] to devirtualizing memory [18]—to compress information about address translations into fewer hardware TLB entries. The contributions of this work are:

(1) We propose active OS page coalescing to generate unbounded amounts of translation contiguity in all execution environments regardless of the amount of memory fragmentation in the system. Because it does not depend on custom hardware support, Translation Ranger is widely applicable in any system with TLB support for contiguity, whether that system is already commercially available [9, 39] or relies on emerging research proposals such as range TLBs [18, 23].
(2) We implement Translation Ranger in the Linux v4.16 kernel to assess the feasibility of our approach. Our real-system implementation sheds light on the subtle challenges of building Translation Ranger within real OSes. Chief among them are the challenges of reducing page migration overheads, dealing with the presence of pages deemed non-movable by the kernel, and understanding the impact of page coalescing on user application performance. We demonstrate that it is useful to coalesce pages not only at allocation time, but also post-allocation in highly-fragmented systems. This observation goes beyond prior work which generally avoids post-allocation defragmentation because of its presumed overheads [4, 12, 18, 23, 39]. To encourage further research on translation contiguity and coalescing TLBs, we have open-sourced our kernel implementation. [1]
(3) We show that Translation Ranger generates significant contiguity (> 90% of 120GB application footprints are covered using only 128 contiguous regions, compared to < 1% without coalescing). It does so with low overhead (< 2% of overall application run time while coalescing 120GB memory), thereby ensuring that the performance gain delivered via coalescing is a net win for applications.

2 BACKGROUND

The increasing overheads of address translation and virtual memory have prompted research on techniques to mitigate their cost [4, 9, 12, 18, 23, 37–39, 49]. We summarize these efforts in Table 1 and discuss them before describing Translation Ranger in detail.

2.1 Using Translation Contiguity

The earliest approaches to exploiting translation contiguity focused on huge pages [32, 40, 46]. OSes form huge pages by allocating groups of spatially-adjacent physical frames to a spatially-adjacent group of virtual pages in discrete-sized chunks at aligned memory boundaries. For example, the x86-64 architecture supports 2MB and 1GB huge pages if the OS can allocate 512 or 262,144 contiguous 4KB virtual pages and physical frames aligned to 2MB or 1GB address boundaries, respectively. This permits x86-64 compliant TLBs to use

[1] https://github.com/ysarch-lab/translation_ranger_isca_2019

Techniques	**Software Requirements**			**Hardware Requirements**	**Coverage Limits**
	Buddy Allocator	Reservation	Page Table		
hugetlbfs	Separate pools	Required	No change	2MB & 1GB page size TLB	2MB and 1GB
THP&khugepaged	2MB pages	None	No change	2MB page size TLB	2MB
COLT [39]	Contiguous pages	None	No change	Coalesced TLB	Up to 8× 4KB
Dir Segments [4]	Not related	Required	Segment table	Direct Segment registers	Any size
Redundant Memory Mappings [23]	Increase max order, eager paging	None	Range table	Range TLB	Any size in HW, but limited by SW
Hybrid TLB [37]	Contiguous pages	None	Anchor page table	Hybrid TLB	N× 4KB or 2MB
Devirtualizing Memory [18]	Increase max order, eager paging	None	A new page table entry	Access validation cache	4KB, 2MB, or 1GB in HW, but limited by SW

Table 1: Techniques used or proposed by industrial or academic research groups for high performance address translation.

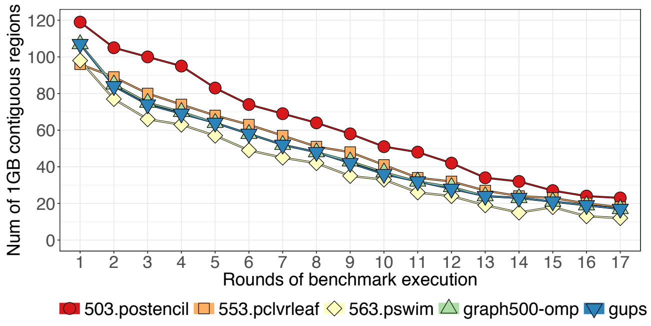


Figure 3: There is plenty of contiguity available at boot time, but memory becomes fragmented soon thereafter.

a single entry to cache what would otherwise take 512 or 262,144 entries..

Although huge pages can be effective, they only offer discrete chunks of contiguity. As memory capacities continue to grow, translation contiguity amounts in excess of 1GB or sizes between the discrete page sizes of 2MB and 1GB will be useful. For this reason, recent work has considered translation contiguity approaches complementary to traditional huge pages. All of these techniques require hardware support from the TLB, as outlined in Table 1. For example, *direct segments* [4], uses programmer-OS coordination to mark gigabyte- to terabyte-sized *primary segments* of memory that are guaranteed to be mapped using contiguous translations. While this approach can substantially reduce TLB misses, direct segments can be challenging to use for real-world workloads which need more than one primary direct segment (to allocate contiguous segments in different parts of their address space) and because of the need for explicit programmer intervention. *Redundant memory mappings* [23] support arbitrary translation ranges in TLBs, but require invasive OS changes to produce these contiguous translation ranges. *Devirtualizing memory* [18] extends the concepts of direct segments for area-constrained accelerators, but works under the optimistic assumption that OSes can always offer large contiguous memory regions for devirtualization.

2.2 Allocation-Time Contiguity

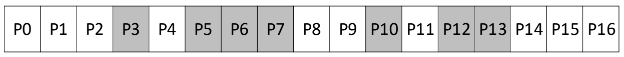
One way to create allocation-time contiguity is to reserve the memory in advance of application run. Libhugetlbfs achieves this by reserving memory at boot time [27]. Allocations for 2MB or 1GB pages are satisfied from these reserved memory pools. Similarly, device drivers often use customized memory allocators that perform similar reservation in advance [11].

Other approaches in Table 1 target contiguous allocations at run time. Prior work achieves this by making changes to the widely-used buddy memory allocator [25]. The buddy allocator can be a source of translation contiguity because it groups contiguous free memory into free page pools of different sizes, from 1 to 2^N pages, where N is called the *max order* of the buddy allocator. Standard allocations will thus contain a maximum of 2^N contiguous pages. For example, Linux's buddy allocator supports maximum contiguous allocations of 4MB. However, because the buddy allocator must support fast insertions and deletions, free pages are stored in unordered lists. This means that one or more memory allocations cannot guarantee contiguity greater than 2^N pages even if two or more contiguous 2^N pages are available in the same free list.

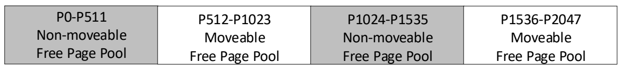
Prior work creates allocation-time contiguity by increasing the max order of the buddy allocator [18, 23]. However, this creates multiple problems. First, Linux's `sparsemem` (used to support discontiguous physical address spaces, which is common in modern systems) requires each contiguous physical address range to be aligned to 2^N [50]. This means that if the max order is increased to support contiguous free ranges of 1GB, many gigabytes of memory may be wasted. Second, increasing the max order does not solve the problem of fragmentation; it only allows programs to obtain large allocations in low fragmentation scenarios. Fragmentation does not directly affect the contiguity of in-use memory ranges, but it does affect the amount of contiguity available at allocation time.

2.3 Memory Fragmentation

To quantify the problem of fragmentation, we ran a set of benchmarks multiple times starting with a fresh-booted system with 128GB of memory (see Table 2) and we increased the max order of the buddy allocator to allocate large contiguous regions [18, 23]. Figure 3 shows that all benchmarks have >80% of their footprint covered by 1GB contiguous regions (not to be confused with 1GB pages, which must also be aligned) on their first execution immediately after boot. However, as the benchmarks keep running, the number of 1GB contiguous regions drops to only 20%. With Translation Ranger, we enable better contiguity in both fresh-booted systems *and* heavily-fragmented systems.



(a) In-use page fragmentation prevents 8-page contiguity even though there are 10 free pages available. Filled blocks are in-use pages.


P0-P511 Non-moveable Free Page Pool
P512-P1023 Moveable Free Page Pool
P1024-P1535 Non-moveable Free Page Pool
P1536-P2047 Moveable Free Page Pool

(b) Moveable/non-moveable free page pool fragmentation prevents user space getting contiguity beyond 512 pages.

Figure 4: Some possible types of fragmentation.

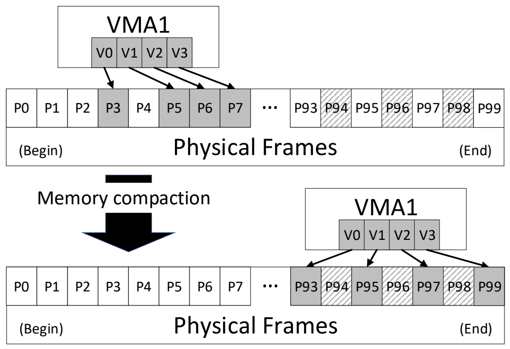


Figure 5: Defragmentation via memory compaction (e.g., in Linux) might destroy in-use contiguity as an unintended side effect of creating more free memory contiguity.

Large contiguous regions are often fragmented in long-running systems for several reasons. For example, in-use pages can prevent otherwise-free buddy pages from being promoted to a larger free page pool for allocation. We show an example of this in Figure 4a. In-use pages allocated are *non-movable* (also called *wired* in FreeBSD or *non-paged* in Windows [30, 42]), which means that they cannot be defragmented [34]. The use of *non-moveable free page pools*, which are dedicated for kernel page allocations, can minimize the interleaving of kernel pages with user pages, as shown in Figure 4b; this avoids non-movable page fragmentation, but prevents large contiguous regions beyond each pool size from being formed [34]. This also explains why benchmarks in Figure 3 lose 1GB contiguous regions over multiple rounds of executions.

Surprisingly, existing OS memory compaction techniques can sometimes *harm* contiguity. As shown in Figure 5, Linux uses memory compaction to move in-use pages to one end of physical address space, leaving the other end with contiguous free frames. However compaction moves only base pages and not transparent huge pages (THP) because existing TLBs, which cannot coalesce contiguous THPs into a single hardware entry, are unable to benefit from higher contiguity. Compaction is also unaware of the contiguity of in-use pages, so if a set of contiguous in-use pages are moved to a set of scattered free pages, the original contiguity may be destroyed.

3 TRANSLATION RANGER

We design Translation Ranger with several goals in mind. Translation Ranger should go beyond the restricted amount of contiguity offered by techniques like huge pages; it should support arbitrarily sized contiguity whenever possible. This contiguity should be generated even on systems with high load and memory fragmentation after page allocation. Finally, contiguity generation should be robust to system behavior no matter how many workloads have spawned, have died, or are executing on that system.

3.1 Design Overview

Translation Ranger creates contiguity from both virtual pages and physical frames. Contiguity in virtual pages depends upon the layout of a process's virtual address space. Most OSes organize each process's virtual address space as multiple non-overlapping virtual address ranges. In Linux, each virtual address range is described by a `struct vm_area_struct`, or virtual memory area (VMA). Applications obtain virtually-contiguous address ranges via `mmap` or `malloc`. Physical frames are allocated and assigned to virtual pages lazily when the virtual page is accessed for the first time. This means that contiguous regions are created when contiguous virtual pages in a VMA are assigned contiguous physical frames. However, since the OS may assign any physical frame to a faulting virtual page, contiguous virtual pages in a VMA are typically mapped to non-contiguous physical frames as shown in Figure 6.

Translation Ranger's approach to generating translation contiguity is to rearrange the system's physical memory mappings such that each VMA can be covered by as few contiguous regions as possible, with regions that are as large as possible. Ideally, a single VMA would constitute one contiguous region, and could be tracked using just one TLB entry. To minimize region counts and maximize region sizes, Translation Ranger does the following:

(1) It assigns an *anchor point* to each VMA. An anchor point is a virtual page (VPN) and physical frame (PFN) pair, (V_{anchor}, P_{anchor}), that acts as a reference around which Translation Ranger builds contiguity using all pages in this VMA.
(2) It actively coalesces memory within each VMA based on the assigned anchor point. Figure 6 shows an example of active coalescing for the case where (V0, P4) is the anchor point. The VMA is coalesced by ① migrating P2 to P4, ② exchanging P5 with P9, ③ exchanging P7 with P6, ④ exchanging P7 with P3, ultimately leading to V0-V3 mapping to P4-P7.
(3) As a background daemon, it periodically iterates over all active VMAs in the system to maintain contiguity during the course of VMA allocations, expansions, and contractions, which occur naturally through the lifetime of an application.

We detail each of these steps in the next three subsections.

3.2 Per-VMA Anchor Point Assignment

To achieve large regions of contiguity, Translation Ranger has to select anchor points carefully. It is natural to consider using the translation corresponding to the first in-use virtual page of a VMA as a good candidate for the anchor point when no anchor point has been selected for this VMA. However, to make this anchor point useful, the OS has to allocate the physical frames for these virtual pages to satisfy several requirements. The question of how multiple VMAs interact is central to this issue. Consider, for example, Figure

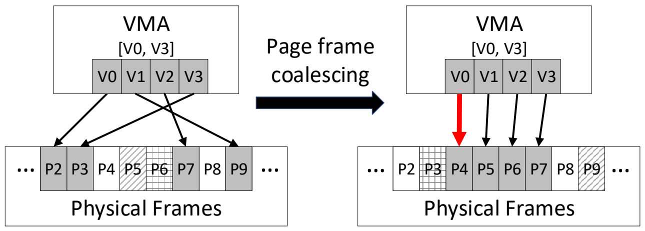


Figure 6: Coalescing the pages in a Virtual Memory Area (VMA): after coalescing page frames, virtual pages V0–V3 map to contiguous physical frames P4–P7. Filled page frame boxes denotes those mapped by V0–V3, marked boxes denotes the frames mapped by other VMAs, and blank boxes denotes free frames. The VMA's *Anchor Point* is (V0, P4).

7a, where VMA1 contains V0-V3, while VMA2 contains V12-V15. If VMA1 and VMA2 use (V0, P4) and (V12, P8) as their respective anchor points, each VMAs can maximize the contiguity it achieves.

On the other hand, Figure 7b shows that a less fortunate anchor point selection of (V12, P6) for VMA2 damages contiguity formation because this anchor point selections causes inter-VMA overlap of physical frames. In this situation, coalescing VMA2 will wipe out (some of) VMA1's contiguity and vice-versa. Even worse, when these VMA sizes change, we have to revisit these mappings, which can dramatically increase the number of page migrations needed to compensate for the inter-VMA interference and increase Translation Ranger's overheads.

To avoid interference between regions during coalescing, Translation Ranger tracks the physical address space as coalesced and uncoalesced regions. Translation Ranger assigns newly allocated VMAs to the uncoalesced physical region whenever possible. When the translations within a VMA are coalesced, this region is marked as coalesced. To find uncoalesced regions for new allocations, Translation Ranger simply uses an algorithm similar to first-fit [53], i.e., scan linearly through all coalesced regions and stop at the first sufficiently large hole between any two such regions. When no such hole is found, Translation Ranger tries to accommodate it by removing the smallest VMA's anchor point; otherwise, it skips this VMA. This approach avoids inter-VMA interference, and thus substantially mitigates Translation Ranger overheads by minimizing page migrations caused by the interference. This also provides useful information for tackling the problem of VMA size changes, which we discuss further in Section 3.4.

Another consideration for anchor point placement is physical page alignment. In general, virtual page and physical frame alignment are chosen based on the needs of TLB implementations. *Traditional TLBs* assume that virtual pages and physical frames are aligned based on page size. For example, TLBs that can cache 2MB huge pages require that the virtual page and physical frame of the huge page begin at 2MB address boundaries. For such cases, we ensure that anchor points are aligned to the largest architectural page size smaller than or equal to the region in question. For example, a 6MB region will be 2MB-aligned so that it can be in-place promoted to three 2MB large pages. On the other hand, newer *contiguity-aware TLBs* like range TLBs lift this restriction and do not require any form of alignment in the contiguous regions [23].

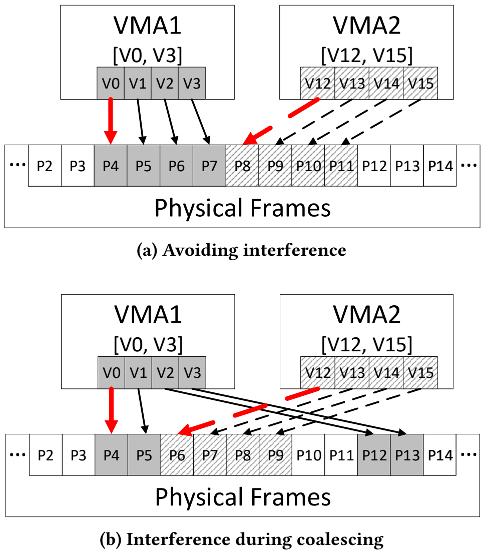

VMA1
[V0, V3]
V0 V1 V2 V3
VMA2
[V12, V15]
V12 V13 V14 V15
… P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 …
Physical Frames
(a) Avoiding interference
(b) Interference during coalescing

Figure 7: Anchor points (shown with red arrows) must be chosen carefully to prevent inter-VMA interference.

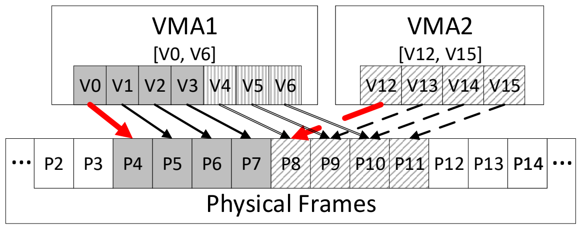
3.3 Intra-VMA Page Coalescing

After the anchor point of a VMA is selected based on the steps described above, Translation Ranger coalesces pages within the VMA to create contiguity. Figure 6 shows how coalescing proceeds after an anchor point selection within each VMA. When performing coalescing within a VMA, Translation Ranger has to handle several practical issues:

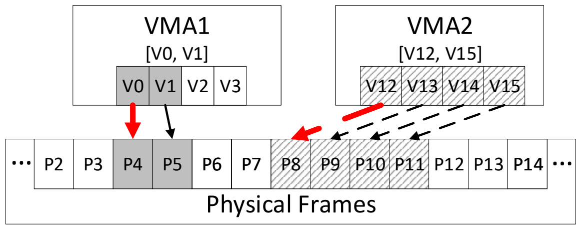
In-Use Page Frames. Target physical frames may be in one of two states: free or in-use. If a target page frame P_n is free, Translation Ranger use Linux's page migration mechanism to move the source frame's data to the target frame. If a target page frame P_n is in use, we cannot simply clobber it. One solution would be to move the contents of the in-use frame to an intermediate physical frame before migrating the source frame to the target frame, but this suffers from extra storage and copy time overhead. In addition, under memory pressure, allocating a new intermediate physical frame can trigger the page reclamation process, leading to significant performance degradation.

Therefore, we directly exchange two pages by unmapping the two pages at the same time, exchanging the content of the two pages, and finally remapping these two pages [54]. Instead of copying data into new pages, this patch transfers data between source and target pages using CPU registers as the temporary storage for in-flight iterative data exchange operations. This approach requires no extra storage or page allocation and even supports the exchange of THPs. It supports exchanging between two anonymous pages as well as between one anonymous page and one file-backed page, but it does not support exchange of two file-backed pages (due to complicated file system locking and their rare occurrence).

Non-Movable Pages. Another issue that Translation Ranger must handle is the presence of non-movable pages. Some examples of non-movable pages are those in use by the kernel (e.g., for



(a) VMA1 grows its size at the end. Its new virtual pages V4, V5, and V6 overlap with VMA2 during memory coalescing.



(b) VMA1 shrinks its size at the end. The freed virtual pages V2 and V3 leave a physical page gap and fragment memory.

Figure 8: VMA size changes: VMA growth and shrinkage. They cause inter-VMA interference and memory fragmentation, respectively

slab allocations, page tables, or other kernel data structures). Alternately, a page may be marked as busy because it is involved in I/O operations, migration, dirty page writeback, or DMA. We handle each of these non-movable page types in different ways. Since kernel pages usually have a long lifetime and limited OS support for page migration, we simply skip them during the coalescing process. However, busy pages are usually ephemeral; therefore, even if these pages cannot be moved currently, Translation Ranger will try to coalesce them again in the future using iterative coalescing. If Translation Ranger must skip a non-movable frame, it will continue coalescing the remaining frames using the originally selected anchor point. We investigated the value of creating new anchor points following non-movable pages, but we found that doing so yields little benefit. Generating additional anchor points needlessly splits a large contiguous region into two smaller contiguous regions should a busy page (the cause of a new anchor) become non-busy in the future.

3.4 Iterative Page Frame Coalescing

Applications can grow and shrink their VMAs over time. This is problematic if a VMA grows so that its physically contiguous region overlaps and interferes with another VMA, or if a VMA shrinks enough that the freed physical frames become a free memory fragment. We show both cases in Figure 8. To avoid this problem, Translation Ranger tracks per-VMA size along with each VMA's coalesced region size during each coalescing iteration. If, on future iterations, Translation Ranger discovers a VMA that has now grown and overlaps with another VMA (by examining coalesced region information), Translation Ranger relocates the coalesced region of one of the two VMAs by assigning a new anchor point. To minimize page frame relocation overheads, the smaller of the two is moved. On the other hand, if no overlapping occurs, the new addition to the VMA will also be coalesced and the VMA's coalesced region size will be adjusted accordingly.

Translation Ranger is also designed to coalesce large important VMAs and ignore smaller shorter-lived VMAs (< 2MB) such as those used to map data structures like thread stacks. The rationale is that these VMAs tend to be sufficiently small such that that coalescing their page frames them yields little additional contiguity relative to the overhead of the necessary page migrations. We believe more sophisticated strategies, e.g., consolidating multiple thread stacks, or lazy VMA deallocation to lengthen VMA lifetimes, could further decrease coalescing overhead and generate even larger contiguous regions, but we leave these for future work.

3.5 Additional Implementation Challenges

Synonyms and Copy-on-Write. To handle synonyms, Translation Ranger coalesces a physical range based on the anchor point from the first created VMA, and ignores the anchor points of the synonym VMAs. This can be done efficiently in Linux by checking `anon_vma` for anonymous VMAs and `address_space` for file-backed VMAs. Care needs to be taken when generating contiguity on pages created by copy-on-write (COW) to avoid unnecessary coalescing work. Translation Ranger skips forked VMAs that share the same physical pages, the same way it does for skipping synonym VMAs. After COW, Translation Ranger creates a new anchor point with the COW physical page for that VMA.

Reducing Runtime Overheads. Excessive page migration during coalescing can incur high runtime overheads. To avoid this, Translation Ranger selects anchor points to avoid inter-VMA interference (see Figure 7). Furthermore, it focuses on large and long-lived VMAs to avoid wasting coalescing effort (see Section 3.4). Moreover, we enable users or system administrators to control Translation Ranger's runtime impact through a tunable parameter. We envision that this tunable will be used the same way as the vast majority of other existing operating system services like khugepaged. That is, if there is hardware to take advantage of contiguity (via TLB optimizations like range TLBs), we can expect administrators to run Translation Ranger more frequently to aggressively generate contiguity, though we will reflect on a sane default in Section 5.3. Finally, we design Translation Ranger as a background daemon that is not on the critical path of application execution and can "steal" idle CPU cycles. This is similar in spirit to the design of already-existing daemons like khugepaged.

4 EXPERIMENTAL METHODOLOGY

Translation Ranger is widely deployable on systems with and without fragmentation and can leverage any previously-proposed TLB hardware that supports translation contiguity [4, 18, 23, 37, 39]. Naturally, Translation Ranger's performance benefits will vary depending on the target system's fragmentation levels and the contiguity-aware TLBs that leverage it. To achieve good performance, Translation Ranger must generate enough translation contiguity for contiguity-aware TLBs to offset any runtime overheads from Translation Ranger's page movement operations.

Experimental Environment	
Processors	2-socket Intel E5–2650v4 (Broadwell), 24 cores/socket, 2 threads/core, 2.2 GHz
L1 DTLB	4KB pages: 64-entry, 4-way set assoc. 2MB pages: 32-entry, 4-way set assoc. 1GB pages: 4-entry, 4-way set assoc.
L1 ITLB	4KB pages: 128-entry, 4-way set assoc. 2MB pages: 8-entry, fully assoc.
L2 TLB	4KB&2MB pages: 1536-entry, 6-way set assoc. 1GB pages: 16-entry, 4-way set assoc.
Memory	128GB DDR4 (per socket)
OS	Debian Buster — Linux v4.16.0

Table 2: System configurations and per-core TLB hierarchy.

4.1 Evaluation Platform

We implement Translation Ranger in Linux kernel v4.16 and evaluate it on a two-socket Intel server (see Table 2). We run a variety of benchmarks from SPEC ACCEL [22], the GAP benchmark suite[2] [6], graph500 [31], and GUPS [43] (see Table 3). Translation Ranger running as a service daemon is collocated with applications in the same memory node and tuned to periodically coalesce application memory; to produce contiguity statistics, application memory is scanned every 5 seconds to retrieve the virtual-to-physical mappings of each page belonging to the application. This statistics collection is engineered to have negligible impact on runtime and would not be present in production deployments.

A central insight from our work is that system load and fragmentation levels is critical to the question of how much contiguity can be generated. The success of contiguity-aware TLBs rests on the OSes ability to generate contiguity robustly across a wide variety of scenarios. To stress-test whether Translation Ranger indeed generates contiguity in heavily fragmented environments, we go beyond prior work [4, 18, 38–40] to use a methodology that *preconditions* memory before our evaluations.

We first use an existing methodology used by kernel developers to artificially fragment the free memory lists as if our system was long-running system with all its free lists randomized [52]. We then further load the system by run a synthetic benchmark, memhog, an application that allocates memory throughout the physical address space, and has been used in prior studies to create fragmentation. Together, these steps ensure that memory is in a fragmented state similar to a realistic steady state shown in Figure 3, which prevents applications obtaining unrealistic contiguity from sequential memory accesses. We also configure our benchmarks to use 95% of total free memory similar to many datacenter and HPC environments.

4.2 Experimental Configurations

To understand Translation Ranger's effectiveness on improving memory contiguity, we use Linux's default buddy allocator configuration (**Linux Default**) as our baseline. This baseline is what has been used by prior contiguity-aware TLB designs that rely on serendipitously generated contiguity [37–39]. We also compare against an enhanced buddy allocator with its max order increased to 20, permitting 2GB contiguous region allocations. This **Large Max Order** is representative of approaches from prior work like redundant memory mappings and devirtualizing memory because it relies on generating contiguity at allocation time and hence serves as a valuable point of comparison to our approach [18, 23]. Furthermore, we compare Translation Ranger to **Enhanced khugepaged**, which is another technique Linux uses to generate contiguity by collapsing scattered 4KB pages into a new THP. To conservatively assess Translation Ranger's relative benefits, we tune khugepaged to scan the entire application footprint every 5s as opposed to its default of defragmenting only 16MB of memory every 60s.

Finally, when profiling Translation Ranger, we quantify the results when coalescing every 5 seconds, and every 50 seconds to showcase the relationship between runtime overheads and Translation Ranger's ability to generate contiguity. For all five configurations, THPs are enabled by default and the buddy allocator uses Linux's default max order 11 to allow a maximum 4MB contiguous page allocation, except for Large Max Order.

[2]We scaled up two synthetic input graphs, Kron and Urand, and run three kernels, Between Centrality (bc), Connected Components (cc), and PageRank (pr) with these two input graphs.

Suite	Description	Benchmark	Footprint
SPEC ACCEL	Compute & memory intensive multi-threaded workloads using **OpenMP**	503.postencil	121GB
		551.ppalm	121GB
		553.pclvrleaf	121GB
		555.pseismic	120GB
		556.psp	113GB
		559.pmniGhost	120GB
		560.pilbdc	121GB
		563.pswim	117GB
		570.pbt	119GB
HPC	Generation and search of graphs	Graph500	122GB
	Random access benchmark	GUPS	128GB
GAP Bench-marks	Common graph processing kernels	bc-kron	121GB
		bc-urand	122GB
		cc-kron	116GB
		cc-urand	116GB
		pr-kron	116GB
		pr-urand	116GB

Table 3: Benchmark descriptions and memory footprints.

4.3 Contiguity Metrics

We focus on two metrics to evaluate the effectiveness of Translation Ranger on real systems. First, we count the total number of contiguous regions needed to cover the entire application memory footprint ($TotalNum_{ContigRegions}$). Our goal is to reduce the total number these regions such that their total amount is comparable to the most aggressive eager paging and identity mapping techniques from prior work (e.g., direct segments, range TLBs, devirtualizing memory [4, 18, 23]).

We calculate the percentage of total application footprint covered by the largest 32 contiguous regions ($MemCoverage_{32Regions}$) and the percentage of total application footprint covered by the

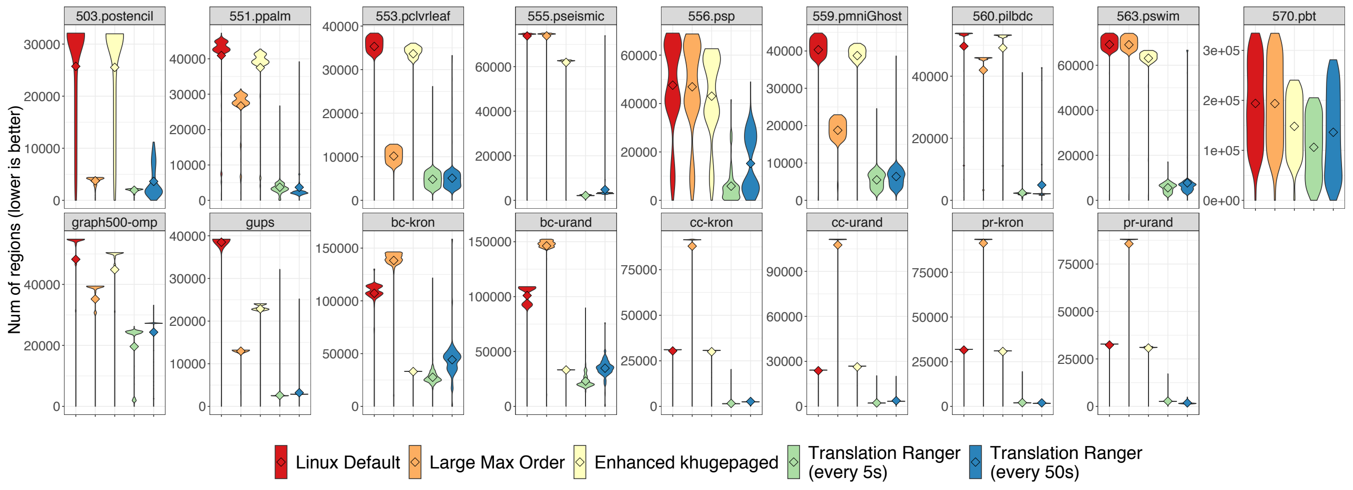


Figure 9: Total number of contiguous regions covering entire application memory (*$TotalNum_{ContigRegions}$*).

largest 128 contiguous regions ($MemCoverage_{128Regions}$). Our goal is to show that even small 32-128 entry contiguity-aware TLBs can capture the majority of the application footprint. This metric has been previously used by prior work [18, 23] to understand contiguity improvements.

4.4 System Overheads

Translation Ranger can add overhead to systems because pages undergoing migration are not accessible to applications and require TLB invalidations and shootdowns. Naturally, these overheads will be offset by improved TLB hit rates. Nevertheless, to mitigate even the cost of page migration, we present all benchmark runtimes for the five measured configurations in the conservative scenarios where contiguity-aware TLBs are **absent**. We normalize these runtimes to our baseline, Linux Default. Our platform supports discrete page sizes (4KB and 2MB) but does not take advantage of other kinds of contiguity, so excess application runtime due to coalescing can be viewed as the software tax of our system. Ultimately, we will show that Translation Ranger generates contiguity comparable to the most aggressive prior proposals, while simultaneously incurring such low overheads that the benefits will come virtually "for free" on most systems.

5 EXPERIMENTAL RESULTS

We begin by showing translation contiguity results for all benchmarks, followed by highlighting two interesting cases to provide more detail about how applications behave over time. Finally, we show system runtime overheads and discuss Translation Ranger's applicability to other OSes.

5.1 Overall Translation Contiguity Results

To concisely show individual results, we show several metrics, including $TotalNum_{ContigRegions}$, $MemCoverage_{32Regions}$, and $MemCoverage_{128Regions}$ in Figure 9 and Figure 10 using violin plots [51]. Violin plots aggregate all numbers over application runtime into a distribution represented by a"violin plot" parallel to the y-axis. The thickness of a violin plot indicates how often the values from y-axis occur. The arithmetic mean of all values is shown as a diamond symbol within each plot.

Figure 9 shows the $TotalNum_{ContigRegions}$ distributions for all benchmarks. We observe that most benchmarks show $TotalNum_{ContigRegions}$ aggregating in one primary area, which reflects their bulk memory allocation behavior; the number of regions does not change much over time. On the other hand, 556.psp and 570.pbt, which allocate memory recurrently, have their $TotalNum_{ContigRegions}$ spread across a range of values along the y-axis due to frequent small allocations.

Among the five configurations, Linux Default and Enhanced khugepaged require many more contiguous regions (violin plots are thick at the top of each plot) to cover each application's memory footprint. This is primarily because they are both limited by the buddy allocator, which yields contiguity regions up to 4MB. The Large Max Order configuration can generate much larger, and thus fewer contiguous regions, since it modifies the buddy allocator to provide up to 2GB contiguous regions. However, Translation Ranger (at either frequency) is able to coalesce memory more effectively and needs far fewer contiguous regions to cover each application footprint; i.e., it is the *most successful* technique for generating contiguity.

Figure 10 shows the coverage of each technique when using the largest 32 or 128 contiguous regions. The plots tend to cluster together. This is because TLB coverage is influenced by the size of the largest contiguous regions present in the system and not just the total number of regions. Among all five OS configurations, Linux Default and Enhanced khugepaged can typically only cover < 1% of each application footprint (with either 32 or 128 regions), since they can achieve at most 4MB contiguous regions. The Large Max Order configuration can typically cover up to 40% of the application footprint, but is also limited by 2GB contiguous regions from its buddy allocator modifications. Theoretically, the Large Max Order configuration should be able to cover all application footprints with 128 contiguous regions, if each region is at least 1GB. However,

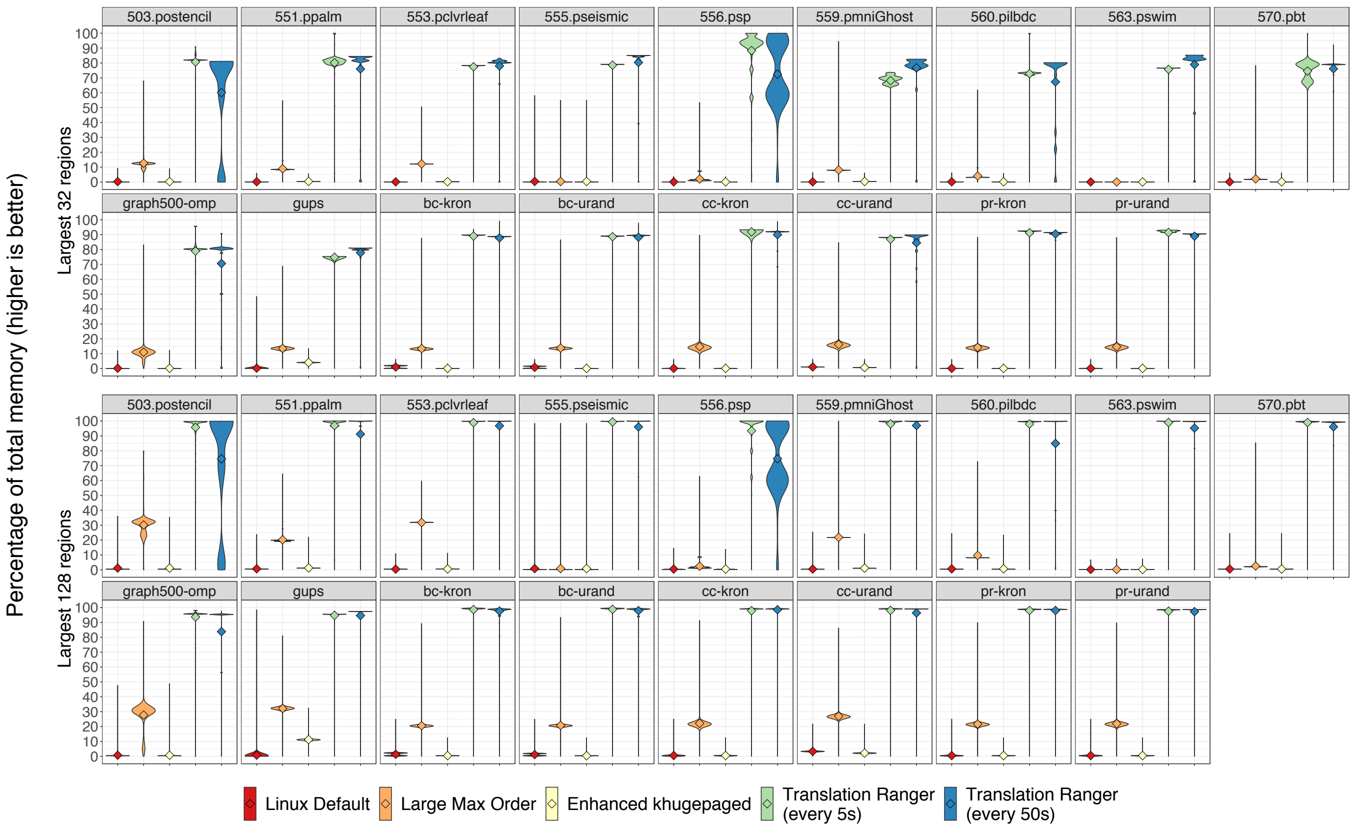


Figure 10: The percentage of total application footprint covered by the largest 32 contiguous regions ($MemCoverage_{32Regions}$) is shown in the top and percentage of total application footprint covered by the largest 128 contiguous regions ($MemCoverage_{128Regions}$) is shown in the bottom.

due to memory fragmentation, not all contiguous regions obtained from the buddy allocator are maximally-sized.

Translation Ranger (at all the frequencies we studied; we show 5s and 50s frequencies in our graphs) consistently creates much larger contiguous regions that can cover the majority of each application's footprint. Utilizing 128 contiguous regions, Translation Ranger can typically cover $> 90\%$ of a 120GB application footprint. In comparison, the last level TLB of a CPU today typically contains 1536 entries and hence even when they exclusively use THPs (2MB) they can only cover 2GB of footprint. Thus Translation Ranger combined with existing coalescing TLB proposals can typically improve TLB coverage by over 30× as shown in Table 4. Meanwhile, TLB storage could be reduced by 85% from a 1536-entry traditional TLB to a 128-entry range cache, where the former uses about 13.5KB[3] and the latter uses 2KB [48].

5.2 Notable Individual Benchmarks

In order to provide more insight into how Translation Ranger achieves contiguity, we highlight two workloads: 503.postencil, which allocates memory in bulk and 556.psp, which allocates and frees memory frequently.

503.postencil Figure 11a shows the contiguity results over the runtime of 503.postencil, which first creates a huge address region, fills it with physical frames, then processes all data in memory. The left most plot in Figure 11a shows the $TotalNum_{ContigRegions}$ over application runtime for the 5 contiguity producing approaches. This plot shows the same data as the left most plot of Figure 9, but spreads time out over the x-axis, thereby showing how the contiguity changes over time. To translate between them, first consider the left most plot of Figure 11a. We can see $TotalNum_{ContigRegions}$ for Linux Default increases from 0 to about 32,000 during the first 40% of application runtime and becomes stable for the remaining 60% runtime. In the left most plot of Figure 9, the Linux Default violin plot is thickest around the value of 32,000, then is fairly uniformly distributed between 0 and 32,000.

From the leftmost plot in Figure 11a, we see several interesting trends. For example, with Translation Ranger more frequent invocation (every 5 seconds) is able to generate large contiguous regions more quickly than the less frequent invocation (every 50 seconds). However, after application memory allocations become

[3]We assume the TLB use 36bit (or 4.5B) tag for VPN and 36bit (or 4.5B) data for PFN and permission bits, so $4.5B \times 2 \times 1536/1024 = 13.5KB$.

Benchmark	C_{max} of a 1536-entry TLB	Largest 32 regions		Largest 128 regions	
		C_{avg}	C_{avg} boost	C_{avg}	C_{avg} boost
503.postencil	3GB	70GB	23×	86GB	29×
551.ppalm	3GB	87GB	29×	104GB	35×
553.pclvrleaf	3GB	94GB	31×	117GB	39×
555.pseismic	3GB	96GB	32×	114GB	38×
556.psp	3GB	55GB	18×	56GB	19×
559.pmniGhost	3GB	92GB	30×	117GB	39×
560.pilbdc	3GB	81GB	27×	102GB	34×
563.pswim	3GB	92GB	31×	111GB	37×
570.pbt	3GB	90GB	30×	114GB	38×
graph500-omp	3GB	83GB	28×	99GB	33×
gups	3GB	100GB	33×	121GB	40×
bc-kron	3GB	106GB	35×	118GB	39×
bc-urand	3GB	107GB	36×	118GB	39×
cc-kron	3GB	102GB	34×	111GB	37×
cc-urand	3GB	97GB	32×	111GB	37×
pr-kron	3GB	104GB	35×	113GB	38×
pr-urand	3GB	102GB	34×	112GB	37×
Average	3GB	92GB	31×	107GB	36×

Table 4: Translation coverage (C) comparison between a 1536-entry traditional TLB and a contiguity-aware TLB using Translation Ranger (every 50s) with largest 32 regions and 128 regions.

stable (at approximately 40% of the application runtime), the less frequent invocation eventually results in a similar number of contiguous regions being produced for the remainder of the execution.

In the middle and right plots plots of Figure 11a we observe that the Linux Default and Enhanced khugepaged configurations can cover very little (< 1%) of the application footprint throughout the application lifetime. The Large Max Order configuration is a substantial improvement with 12.5% of the application footprint being covered with the largest 32 contiguous regions and 32.3% with the largest 128 contiguous regions respectively. However Translation Ranger can cover at least 80% of application footprint with just 32 contiguous regions, and over 95% of the 121GB footprint using 128 contiguous regions after just several iterations of coalescing. To summarize, contiguity-aware TLB designs will attain about 3× TLB coverage (or could reduce their hardware resources proportionally), if they simply use Translation Ranger instead of their own software enhancements.

556.psp Figure 11b shows the contiguity results for 556.psp, which frequently allocates and deallocates memory. The left most plot of Figure 11b shows high volatility across all configurations for $TotalNum_{ContigRegions}$ because frequent memory allocations add many fragmented small pages and deallocations remove existing contiguous regions. All five experimental configurations suffer from this type of application memory allocation pattern.

Because Translation Ranger iteratively coalesces memory to generate contiguous regions, it requires 40% fewer contiguous regions to cover the entire application footprint than Linux Default, Large Max Order, or Enhanced khugepaged. The middle and right plots in Figure 11b show that using the largest 32 or 128 contiguous regions, Linux Default and Enhanced khugepaged can cover < 1% of the application footprint. Large Max Order does a little better, covering about 8% initially, but decreases to < 2% because of the frequent memory allocations and deallocations.

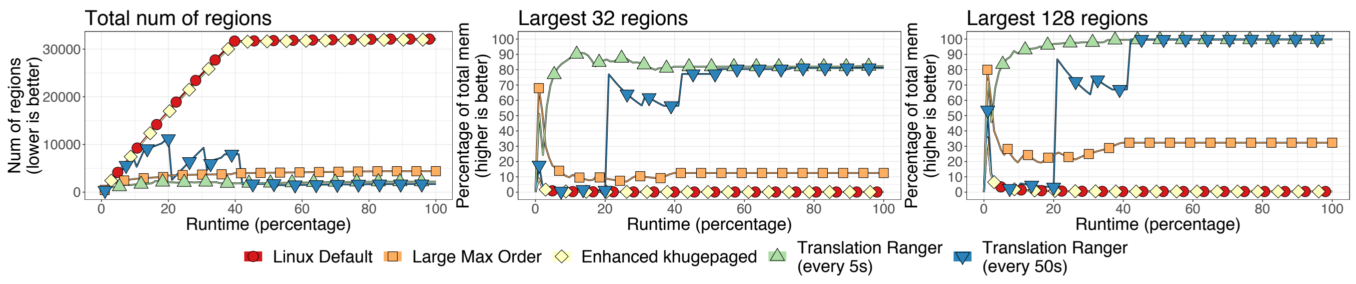
In contrast, Translation Ranger (with frequency set to every 50 seconds) can cover more than 64% of the application footprint with the largest 32 contiguous regions and the coverage is over 72% during the bulk of application runtime; increasing the number of regions to 128 further improves the coverage of the application's footprint to 75%. Additional improvements can come from running Translation Ranger more frequently but do not seem necessary in most cases.

5.3 Translation Ranger Overheads

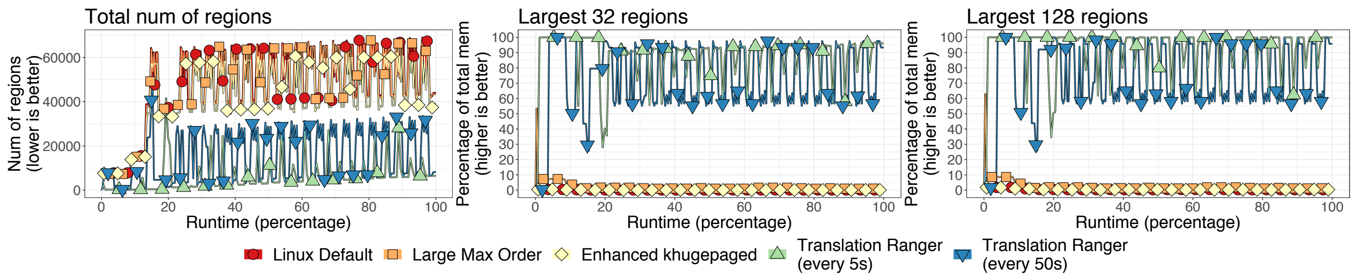
So far we have shown that Translation Ranger creates systematically larger and dramatically fewer contiguous regions compared to other approaches, but we must also consider its "software tax". Figure 12 shows the application execution time across all five configurations normalized to our baseline, Linux Default, averaged across 5 runs to account for variation. Large Max Order, which on average adds 1.1% runtime overhead, incurs very minor slowdowns due to zeroing every free large page at allocation time. The overhead of Enhanced khugepaged is negligible; although it runs very frequently (every 5 seconds), the majority of each workload's memory is already THPs, leaving few base pages for it to convert to THPs. This also explains its small improvements in contiguity for most workloads. Finally, we show that Translation Ranger adds, on average, 1.7% overhead when run every 50 seconds.

bc-kron provides an interesting study, because it actually runs faster than expected with Enhanced khugepaged and Translation Ranger. bc-kron cannot allocate as many THPs as possible at page allocation time, which causes Linux Default and Large Max Order suffer, whereas both Enhanced khugepaged and Translation Ranger are able to generate more THPs out of fragmented in-use pages after page allocation time. This improves the application performance by reducing TLB misses and page table walk overheads thanks to TLB support for huge pages on our test system.

When Translation Ranger is tuned to run more aggressively, every 5 seconds, it also produces more contiguity. We measure an average runtime overhead of of 2.3%, which is 0.6% more than running Translation Ranger every 50 seconds. This small overhead buys more pronounced contiguity with over 90% of application footprint consistently covered with just 32 contiguous chunks through application runtime. We conclude that Translation Ranger has minimally invasive software overheads comparable to prior solutions that are so small that it will will be more than offset by the performance improvements achieved via TLB efficiency, reported at 20-30% in prior proposals [18, 23, 39]. Because Translation Ranger generates substantially more contiguity than those studies, we also expect to see more performance, but an exhaustive study of numerous contiguity-aware TLB designs is beyond the scope of this comprehensive OS implementation of software support for contiguity aware TLBs.



(a) Contiguity results over time for 503.postencil.



(b) Contiguity results over time for 556.psp.

Figure 11: Total number of contiguous regions covering entire application memory ($TotalNum_{ContigRegions}$) is shown in the left most plot; percentage of total application footprint covered by the largest 32 contiguous regions ($MemCoverage_{32Regions}$) is shown in the middle plot; percentage of total application footprint covered by the largest 128 contiguous regions ($MemCoverage_{128Regions}$) is shown in the right most plot.

5.4 Discussion

Applicability to other OSes. The Translation Ranger concept is not Linux specific and is compatible with other OSes like Windows and FreeBSD, which maintain VMA-like data structures per process. For example, FreeBSD uses `vm_map_entry` to identify a contiguous virtual address range instead of a VMA, `vm_object` to represent a group of physical frames from one memory object instead of `anon_vma` for an anonymous memory object and `address_mapping` for a file in Linux. To port Translation Ranger to FreeBSD, we can generate contiguity on each `vm_map_entry` and assign one anchor point to it instead of each VMA.

Permission Checks. All recent work on contiguous regions, including ours, assumes that contiguous regions belong to single VMAs with uniform permissions for all their virtual pages. If the permission of a virtual address range in this VMA is changed, the VMA will be broken into two or more new VMAs with corresponding updated permissions and the page table entries will be updated respectively. This also breaks one contiguous region into multiple regions, and contiguity-aware TLBs will need additional entries to address the original memory address range. To mitigate this problem, new address translation designs [1, 4] that can decouple permission checks from virtual-to-physical address translation, could be helpful, as they allow the TLB to maintain the original contiguous region entry but with additional permission subsections.

NUMA Effects. This initial study focuses on only one memory node in our system as a tractable configuration to thoroughly understand Translation Ranger. Translation Ranger can easily be extended to multi-node NUMA systems, though Translation Ranger's cross-socket traffic overheads will have to be integrated with previously-proposed NUMA paging policies, like autoNUMA, Carrefour-LP, and Ingens [10, 15, 26] to minimize overheads.

6 RELATED WORK

Prior work on memory allocation has studied ways to increase contiguity and reduce memory fragmentation [8, 21, 33] and is discussed in Table 1.

Memory Allocation. Internal and external memory fragmentations are two major problems for memory allocations. To mitigate internal memory fragmentation, the SLAB allocator and others (e.g., jemalloc and tcmalloc) pack small memory objects with the same size together in one or more pages to avoid wasting space [8, 14, 16]. For external memory fragmentation, OSes use buddy allocators to achieve fast memory allocation and restrict external fragmentation [25]. Linux developers separate kernel and user memory allocations to further reduce external fragmentation [17]. Additionally, peripheral devices often require access to physically contiguous memory. Linux accommodates these devices with drivers that use boot-time allocation and reserve contiguous memory before others can request memory via Contiguous Memory Allocators (CMAs) [11]. CMAs use memory compaction to migrate fragmented

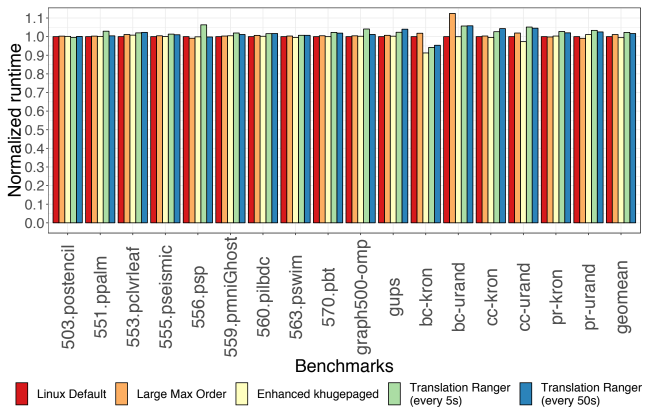


Figure 12: Benchmark runtime for all five configurations: Linux Default, Large Max Order, Enhanced khugepaged, and Translation Ranger with two running frequency. All runtime is normalized to Linux Default.

pages and offer large contiguous physical memory for devices use, especially for DMA data transfer [33]. These approaches try to preserve contiguity for future use, but cannot prevent large free memory blocks from being broken into small ones when memory requests are small in sizes.

Huge pages. Significant prior work has gone into improving huge pages. For example, a huge pages have one access bit, causing memory access imbalance problems in NUMA systems [15]. Several proposals try to manage huge page wisely by restricting huge page creation and splitting huge pages when they cause utilization imbalance issues [15, 26]. These utilization issues could also happen on contiguous regions created by Translation Ranger, thus integrating these policies with Translation Ranger could further improve application performance in NUMA systems.

In heterogeneous memory systems, classifying hot and cold pages to determine how to move them between fast/slow memories is important. Previous work by Thermostat analyzes huge page utilization by sampling sub-pages in each huge page and migrates these cold sub-pages to slow memory to make efficient use of fast memory [2]. Thermostat could provide useful utilization information to Translation Ranger to assess page hotness in identifying which VMAs are particularly worth coalescing.

The high cost of huge page allocation has been a problem for Linux and can lead to application performance degradation [13, 29]. Recent work on Linux prevents free page fragmentation and eliminates most huge page allocation costs by aggregating kernel page allocations [34]. With the help of this work, Translation Ranger could also improve in-use page fragmentation and free page fragmentation, generating even larger contiguous regions.

7 CONCLUSIONS

Translation Ranger is an effective low-overhead technique for coalescing scattered physical frames and generating translation contiguity. The enormous contiguous regions created by Translation Ranger can be used by emerging contiguity-aware TLBs to minimize address translation overhead for all computation units in heterogeneous systems. Accelerators in particular will benefit, as they often have limited hardware resources for address translation. Translation Ranger can scale easily with increasing memory sizes regardless of the limitations imposed by modern memory allocators. With less than 2% runtime overhead, Translation Ranger generates contiguous regions covering more than 90% of 120GB application footprints with at most 128 regions, which can be fully cached by contiguity-aware TLBs to minimize address translation overhead. To address ever-increasing memory sizes, contiguity-aware TLBs provide promising hardware support and Translation Ranger is the software cornerstone needed to enable these designs.

ACKNOWLEDGMENTS

The authors thank Ján Veselý and Guilherme Cox for their input on aspects of this work. This work was partially funded by NSF awards 1319755 and 1916817.

REFERENCES

[1] Reto Achermann, Chris Dalton, Paolo Faraboschi, Moritz Hoffmann, Dejan Milojicic, Geoffrey Ndu, Alexander Richardson, Timothy Roscoe, Adrian L. Shaw, and Robert N. M. Watson. 2017. Separating Translation from Protection in Address Spaces with Dynamic Remapping. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, New York, NY, USA, 118–124. https://doi.org/10.1145/3102980.3103000

[2] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/3037697.3037706

[3] AMD Corporation. 2014. Compute Cores. https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf. [Online; accessed 04-Aug-2018].

[4] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[5] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2012. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 297–308. http://dl.acm.org/citation.cfm?id=2337159.2337194

[6] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[7] Abhishek Bhattacharjee. 2017. Preserving Virtual Memory by Mitigating the Address Translation Wall. *IEEE Micro* 37, 5 (September 2017), 6–10. https://doi.org/10.1109/MM.2017.3711640

[8] Jeff Bonwick. 1994. The Slab Allocator: An Object-caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1 (USTC'94)*. USENIX Association, Berkeley, CA, USA, 6–6. http://dl.acm.org/citation.cfm?id=1267257.1267263

[9] Mike Clark. 2016. A new x86 core architecture for the next generation of computing. In *2016 IEEE Hot Chips 28 Symposium (HCS)*. 1–19. https://doi.org/10.1109/HOTCHIPS.2016.7936224

[10] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. http://lwn.net/Articles/488709/. [Online; accessed 04-Aug-2018].

[11] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc.

[12] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[13] Nelson Elhage. [n.d.]. Disable Transparent Hugepages. https://blog.nelhage.com/post/transparent-hugepages/. [Online; accessed 04-Aug-2018].

[14] Jason Evans. 2011. Scalable memory allocation using jemalloc. https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919. [Online; accessed 04-Aug-2018].

[15] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 231–242. http://dl.acm.org/citation.cfm?id=2643634.2643659

[16] Sanjay Ghemawat and Paul Menage. 2009. Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html. [Online; accessed 04-Aug-2018].

[17] Mel Gorman and Andy Whitcroft. 2006. The what, the why and the where to of anti-fragmentation. In *Linux Symposium*. 369–384.

[18] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/3173162.3173194

[19] HSA Foundation. 2014. HSA Platform System Architecture Specification - Provisional 1.0. http://www.slideshare.net/hsafoundation/hsa-platform-system-architecture-specification-provisional-verl-10-ratifed. [Online; accessed 04-Aug-2018].

[20] Intel. 2018. *Intel 64 and IA-32 Architectures Optimization Reference Manual.*

[21] Mark S. Johnstone and Paul R. Wilson. 1998. The Memory Fragmentation Problem: Solved?. In *Proceedings of the 1st International Symposium on Memory Management (ISMM '98)*. ACM, New York, NY, USA, 26–36. https://doi.org/10.1145/286860.286864

[22] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. 2015. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond (Eds.). Springer International Publishing, Cham, 46–67.

[23] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[24] Vasileios Karakostas, Jayneel Gandhi, Adrian Cristal, Mark Hill, Kathryn McKinley, Mario Nemirovsky, Michael Swift, and Osman Unsal. 2016. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 631–643. https://doi.org/10.1109/HPCA.2016.7446100

[25] Kenneth C. Knowlton. 1965. A Fast Storage Allocator. *Commun. ACM* 8, 10 (Oct. 1965), 623–624. https://doi.org/10.1145/365628.365655

[26] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[27] Adam Litke. [n.d.]. libhugetlbfs. https://lwn.net/Articles/171451/. [Online; accessed 04-Aug-2018].

[28] Jason Lowe-Power. [n.d.]. Inferring Kaveri's Shared Virtual Memory Implementation. http://www.lowepower.com/jason/inferring-kaveris-shared-virtual-memory-implementation.html. [Online; accessed 04-Aug-2018].

[29] MongoDB Manual. [n.d.]. Disable Transparent Hugepages (THP). https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/. [Online; accessed 04-Aug-2018].

[30] Marshall Kirk McKusick and George V. Neville-Neil. 2004. *The Design and Implementation of the FreeBSD Operating System.* Pearson Education.

[31] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. 2010. Introducing the Graph 500. In *Cray User's Group*.

[32] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 89–104. https://doi.org/10.1145/844128.844138

[33] Michal Nazarewicz. [n.d.]. A deep dive into CMA. https://lwn.net/Articles/486301/. [Online; accessed 08-Jul-2018].

[34] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[35] Myrto Papadopoulou, Xin Tong, Andre Seznec, and Andreas Moshovos. 2015. Prediction-based superpage-friendly TLB designs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 210–222. https://doi.org/10.1109/HPCA.2015.7056034

[36] Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. 2018. SEESAW: Using Superpages to Improve VIPT Caches. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 193–206. https://doi.org/10.1109/ISCA.2018.00026

[37] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 444–456. https://doi.org/10.1145/3079856.3080217

[38] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[39] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[40] Binh Pham, Jan Vesely, Gabriel Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have it Both Ways. In *International Symposium on Microarchitecture (MICRO)*.

[41] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 743–758.

[42] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. 2012. *Windows Internals, Part 2: Covering Windows Server 2008 R2 and Windows 7 (Windows Internals).* Microsoft Press, Redmond, WA, USA.

[43] V. Saxena, Y. Sabharwal, and P. Bhatotia. 2010. Performance evaluation and optimization of random memory access on multicores with high productivity. In *2010 International Conference on High Performance Computing*. 1–10. https://doi.org/10.1109/HIPC.2010.5713168

[44] Andre Seznec. 2004. Concurrent support of multiple page sizes on a skewed associative TLB. *IEEE Trans. Comput.* 53, 7, 924–927. https://doi.org/10.1109/TC.2004.21

[45] Yakun Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. 2015. Toward Cache-Friendly Hardware Accelerators. In *HPCA Sensors and Cloud Architectures Workshop (SCAW)* (2015-02-07). http://www.eecs.harvard.edu/~shao/papers/shao2015-scaw.pdf

[46] M. Talluri and M. D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of ASPLOS*.

[47] Vijay Tatkar. [n.d.]. What Is the SPARC M7 Data Analytics Accelerator? https://community.oracle.com/docs/DOC-994842. [Online; accessed 04-Aug-2018].

[48] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. 2008. A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 94–105. https://doi.org/10.1109/MICRO.2008.4771782

[49] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 161–171. https://doi.org/10.1109/ISPASS.2016.7482091

[50] Andy Whitcroft. [n.d.]. sparsemem memory model. https://lwn.net/Articles/134804/. [Online; accessed 04-Aug-2018].

[51] Wikipedia. [n.d.]. Violin plot. https://en.wikipedia.org/wiki/Violin_plot. [Online; accessed 04-Aug-2018].

[52] Dan Williams. 2018. Randomize free memory. https://lwn.net/Articles/767614/. [Online; accessed 07-Dec-2018].

[53] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic storage allocation: A survey and critical review. In *Memory Management*, Henry G. Baler (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–116.

[54] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA. https://doi.org/10.1145/3297858.3304024