Trident: Harnessing Architectural Resources for All Page Sizes in x86 Processors

Venkat Sri Sai Ram*†
Indian Institute of Science
Bangalore, India

Ashish Panwar*
Indian Institute of Science
Bangalore, India

Arkaprava Basu
Indian Institute of Science
Bangalore, India

ABSTRACT

Intel and AMD processors have long supported more than one large page sizes – 1GB and 2MB, to reduce address translation overheads for applications with large memory footprints. However, previous works on large pages have primarily focused on 2MB pages, partly due to a lack of evidence on the usefulness of 1GB pages to real-world applications. Consequently, micro-architectural resources devoted to 1GB pages have gone underutilized for a decade.

We quantitatively demonstrate where 1GB pages can be valuable, especially when employed in conjunction with 2MB pages. Unfortunately, the lack of application-transparent dynamic allocation of 1GB pages is to blame for the under-utilization of 1GB pages on today's systems. Toward this, we design and implement Trident in Linux to fully harness micro-architectural resources devoted for all page sizes in the current x86 hardware by transparently allocating 1GB, 2MB, and 4KB pages as suitable at runtime. Trident speeds up eight memory-intensive applications by 18%, on average, over Linux's use of 2MB pages. We then propose Trident^{pv}, an extension to Trident that virtualizes 1GB pages via copy-less promotion and compaction in the guest OS. Overall, this paper shows that adequate software enablement brings practical relevance to even GB-sized pages, and motivates micro-architects to continue enhancing hardware support for *all* large page sizes.

CCS CONCEPTS

• **Software and its engineering** → **Virtual memory**.

KEYWORDS

Virtual memory; TLB; page table walks; large pages

ACM Reference Format:
Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. 2021. Trident: Harnessing Architectural Resources for All Page Sizes in x86 Processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480062

*Both authors contributed equally.
†The author is currently affiliated with Nutanix Inc. The author contributed toward this work when he was a student at the Indian Institute of Science.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO '21, October 18–22, 2021, Virtual Event, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480062

1 INTRODUCTION

It is not uncommon for hardware features to require software enablement. Unfortunately, it is also common to find the micro-architectural resources to remain underutilized due to the lack of adequate software support. One pays for the underutilized hardware through both – the runtime cost (e.g., power dissipation), and the design and verification cost. Further, architects are left in the dark about the extent to which those features are beneficial to applications in practice, and whether they should continue enhancing them or drop them in future products.

In this work, we shed light on one such hardware feature – 1GB pages, that has been mostly languishing for a decade due to inadequate system software (here, Linux and KVM) enablement. Big-memory workloads are well known to witness significant slowdowns due to virtual to physical address translation (e.g., up to 20-50%). Modern processors support large pages to help reduce this overhead [35, 36, 42, 43]. A large page TLB (Translation Lookaside Buffer) entry maps a larger contiguous virtual address region to contiguous physical address region (e.g., 2MB compared to default 4KB). Consequently, the use of a large page increases TLB coverage and reduces the number of TLB misses that is the primary source of translation overheads.

The x86-64 processors have long supported two large page sizes – 2MB and 1GB, for over a decade. Intel's Sandybridge launched in 2010 first supported 1GB pages and had a four-entry L1 TLB dedicated for 1GB pages in each core [10]. Since then, both large page sizes have found patronage of the processor vendors. Recent Cascade Lake processors support 32-entries in L1 TLB and up to 1536 entries in L2 TLB for 2MB pages. It also has 4 and 16 entries for 1GB pages, in L1 and L2 TLB respectively [11]. The lastest Ice Lake Xeon processors can hold up to 1024 entries each for both 2MB and 1GB pages, per core, in its L2 TLBs [14].

While hardware vendors continue to enhance TLB capacities for both large page sizes; more so for 1GB pages in recent times, the system software – operating systems and hypervisors – continues to focus only on 2MB large pages. For example, Linux's Transparent Huge Pages (THP) for application-transparent dynamic allocation of large pages limits itself to only 2MB pages. Prior academic research works on improving large page support in software similarly ignore 1GB pages [36, 42, 43].

It is thus pertinent to wonder if micro-architects are justified in continuing to invest hardware resources for both 1GB and 2MB large pages. Our first contribution is an empirical analysis to answer the above question. We quantify the usefulness of 1GB pages, *over and above* 2MB pages, to various applications, with and without virtualization. We find that while most memory-intensive applications benefit from 2MB pages over 4KB, a subset of them speeds up further with 1GB pages. Even for applications in that subset,

use of the *other* large page size(s) (here, 2MB) *alongside* the largest page size (here, 1GB) is important. Mapping a virtual address range with a large page size requires the address range to be at least as long as that page size and to be aligned at that page size boundary. Consequently, larger the page size, lesser is the number of virtual address ranges that are *mappable* by that page size.

Often a significant portion of an application's address space is *not* mappable with 1GB pages, but mappable with 2MB. Importantly, such address ranges often witness relatively frequent TLB misses. Thus, if *only* the largest page size is used, then those address ranges would have to be mapped with the smallest (4KB) pages. Consequently, TLB misses would increase significantly.

A larger page size also needs equally longer contiguous physical memory chunk. The larger the page size, shorter is the supply for necessary physical memory chunks. Thus, it may not be possible to map a virtual address range with the largest page size even if it is mappable by that page size.

Besides our own analysis, Google recently reported that nearly 20% of CPU cycles across its data centers are attributed to serving TLB misses, *even after* deploying 2MB large pages [35]. Further, the advent of denser non-volatile memory (NVM) technologies promises to significantly increase the physical memory size [27, 38]. The ability to efficiently address a large amount of memory is essential to harness its full benefit.

While we find usefulness of 1GB beyond 2MB pages, the hardware support *alone* is not enough. Applications cannot use a large page size unless the system software maps its virtual address ranges with it. Importantly, the ease at which an application can leverage a given large page size determines its practical usefulness. For example, if modifications to application code and/or reservation of free physical memory are necessary for allocating larger page sizes, then hardware resources devoted for that large page are likely to remain underutilized.

Unfortunately, users willing to leverage 1GB page today need to apriori reserve free physical memory and may need application modifications or recompilation. Applications that incrementally allocate memory during their execution, e.g., in-memory object stores, are especially ill suited for such a static approach.

Towards this, our second contribution is to build Trident * in Linux to dynamically allocate *all available page sizes* in x86-64 processors to fully harness processor's TLB resources, without necessitating apriori reservation of physical memory or application modifications. A key challenge in dynamically allocating 1GB pages is the hardship in ensuring availability of 1GB *contiguous* physical memory chunks when needed. As the free physical memory gets naturally fragmented over time, finding 1GB chunks becomes more difficult than 2MB chunks. Thus, the dynamic allocation of large pages needs to periodically compact physical memory for making free memory contiguous. However, compaction for 1GB memory requires significantly more work than 2MB. Moreover, a compaction attempt fails if it encounters even a *single* page frame with unmovable contents, e.g., kernel objects like inodes, in a 1GB region. In short, compaction at the granularity of 1GB pages needs a *new* approach.

Trident introduces a smart compaction technique. We observe that the current approach of *sequential scanning* and moving contents of occupied page frames is not scalable to 1GB. This approach incurs a large amount of avoidable data movement. The smart compaction tracks the number of occupied bytes (i.e., the number of mapped page frames) within each 1GB physical memory region. Instead of scanning, smart compaction then frees a region with the *least* number of occupied bytes, which significantly reduces data movement. It also tracks unmovable contents, e.g., Linux's own data, within a 1GB region to avoid unnecessary data movement.

Even with smart compaction, 1GB memory chunks are not always immediately available. About a third of the attempts to allocate 1GB page fail due to the unavailability of contiguous physical memory. Unsurprisingly, 2MB chunks are more easily available. Trident thus maps address ranges with 2MB pages if it fails to map with 1GB pages. These 2MB page mappings are later *promoted* to 1GB pages, when suitable.

We then propose Trident^{pv}, an extension to Trident under virtualization for copy-less 1GB page promotion and compaction in the guest OS. The guest copies contents of guest physical pages to create contiguity in guest physical address space for page promotion and compaction. We observe that copying can be *mimicked* by exchanging the mapping between the guest physical address (gPA) and the host physical address (hPA) of the source and destination. This copy-less technique makes the promotion of 2MB pages to a 1GB page significantly faster than the traditional copy-based approach. The guest OS and the hypervisor coordinate to alter the desired gPA to hPA mappings via a hypercall for the purpose.

On a bare-metal system, Trident speeds up eight memory-intensive applications by 18%, over Linux's THP, on average. Trident^{pv} further improves performance under virtualization, by up to 10%.

In summary, we make the following contributions:

- We evaluate the usefulness of 1GB pages across various applications, both without and with virtualization.
- We empirically demonstrate why it is important to deploy *all* large page sizes, not only the largest one.
- We created Trident in Linux to dynamically allocate all page sizes available on x86-64 processors to speed up applications with large memory footprint.
- We then propose an optional extension to Trident called Trident^{pv} that employs paravirtualization to enable copy-less 1GB page promotion and compaction in the guest OS.

2 BACKGROUND

Applications with large memory footprints often spend significant time in address translation [19, 35, 42]. Translation overhead is primarily due to slow page table walks on TLB misses. Hits in the TLB are fast, but a page table walk may need up to four memory accesses to lookup the hierarchical page table. Newer processors require up to five memory accesses due to deeper page table structures [25]. Large pages help reduce translation overhead in two ways. ① It reduces the frequency of TLB misses by increasing the TLB coverage since a single entry for a large page maps a larger address range. ② It quickens individual walks by reducing the number of levels in the page table that need to be accessed. For example, a walk for

*The name draws from the fact that Trident uses *three* page sizes.

Table 1: Specification of the experimental system

Processor	Intel Xeon Gold 6140 @2.3GHz Skylake Family with 2 Sockets
Number of cores	18 cores (36 threads) per socket
L1-iTLB	4KB pages, 8-way, 128 entries 2MB pages, fully associative, 8 entries
L1-dTLB	4KB pages, 4-way, 64 entries 2MB pages, 4-way, 32 entries 1GB pages, fully associative, 4 entries
L2 TLB	4KB/2MB pages, 12-way, 1536 entries 1GB pages, 4-way, 16 entries
Cache	32K L1-d, 32K L1-i, 1MB L2, 24MB L3
Main Memory	384GB (192GB per socket)
OS / Hypervisor	Ubuntu with Linux kernel version 4.17.3 / KVM

Table 2: Specifications of the benchmarks

Name	Threads	Memory	Description
XSBench	36	117GB	Monte Carlo particle transport algorithm for nuclear reactors [55]
SVM	36	67.9GB	Support Vector Machine, kdd2012 dataset [7]
Graph500	36	63.5GB	Breadth-first-search and single-source-shortest-path over undirected graphs [5]
CC/BC/PR	36	72GB	Graph algorithms from GAPBS [20]
CG.D	36	50GB	Congruent Gradient algorithm from NAS Parallel Benchmarks [15]
Btree	1	10.5GB	Random lookups in a B+tree
GUPS	1	32GB	Irregular, memory-intensive microbenchmark [2]
Redis	1	43.6GB	An in-memory key-value store [24]
Memcached	36	79GB	An in-memory key-value caching store [29]
Canneal	1	32GB	Simulated cache-aware annealing from PARSEC [23]

a 1GB page requires up to 2 memory accesses, compared to 3 for a 2MB page and 4 for a 4KB page, in typical x86 processors.

Large pages under virtualization: Address translation involves two layers under virtualization through nested page tables. First, a guest virtual address (gVA) is mapped to a guest physical address (gPA) through the guest page tables (gPT) managed by the guest OS running on a virtual machine. gPA is then translated to host physical address (hPA) through host page tables (hPT) maintained by the hypervisor. Two layers of indirection increase the number of memory accesses required for a page walk. For example, with four-level page tables, a TLB miss requires up to 24 memory accesses for 4KB pages. Use of 2MB and 1GB pages at both layers reduce the number of accesses to 15 and 8, respectively.

OS support for large page allocation: OSes typically provide three mechanisms to allocate large pages. In the pre-allocation based mechanism, users are required to reserve physical memory for large pages and a helper library (e.g., libHugetlbfs) maps specific segments of an application’s memory with large pages from the reserved memory. Unfortunately, this *static* approach constrains the usability of large pages. The second approach needs explicit system calls to map certain virtual address ranges with large pages. This requires application modification (e.g., madvise syscall or extra flags in mmap). In the third approach, the OS allocates large pages *without* the user or programmer involvement. Linux’s support for Transparent Huge Pages (THP) is an example of this approach.

Internally, THP employs two mechanisms. On a page fault, it checks if the faulting address falls within a virtual address range that is at least as big as and aligned with the large page size. If yes, and a free contiguous physical memory chunk is available, THP maps the address with a large page. For address regions that were not immediately mappable with large pages during page faults, THP employs a background thread khugepaged to locate virtual address ranges mapped with 4KB pages and *promote* (remap) them to large pages, when possible. To ensure enough supply of contiguous physical memory, THP also *compacts* physical memory. Compaction moves contents of occupied pages to one end of the physical memory for creating contiguous free memory regions on the other end. Unfortunately, THP currently supports *only* 2MB large pages. Hosted hypervisors, such as KVM, use THP for allocating 2MB large pages in the host.

3 METHODOLOGY

Table 1 details the configuration of our experimental platform. We evaluate 12 multi-GB workloads (Table 2) across various domains such as machine-learning, graph algorithms, key-value stores, HPC, and micro-benchmarks (GUPS and Btree). We use Linux’s perf tool [8] to collect relevant microarchitectural events. Specifically, we measure the number of cycles spent on page walks via hardware performance counters DTLB_LOAD_MISSES.WALK_ACTIVE and DTLB_STORE_MISSES.WALK_ACTIVE [6].

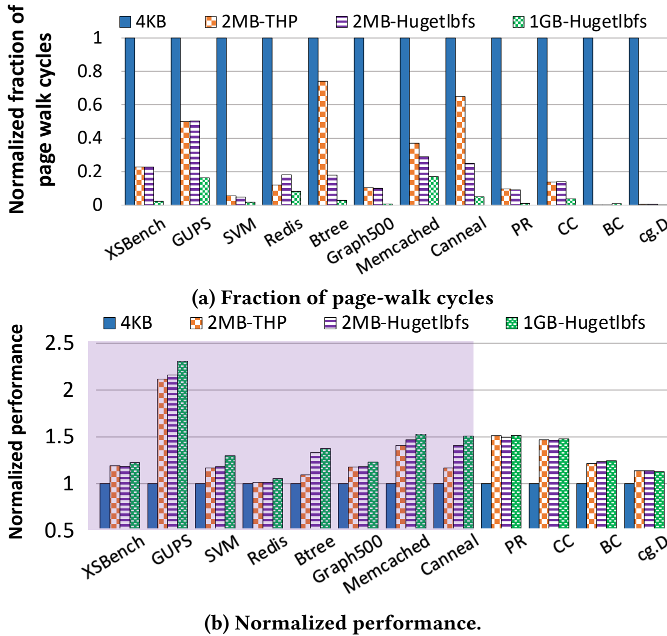
To study performance under different states of the system, we used an opensource tool to fragment the physical memory [57]. Fragmentation is measured using Free Memory Fragmentation Index (FMFI [36]) that lies between 0 (no fragmentation) and 1 (full fragmentation). Following the methodology used in [41], we fragment memory by first caching a large file in OS’s page-cache and then reading it at random offsets for 10 minutes via 24 user threads. Caching file increases FMFI to 0.95 and random accesses ensure that page reclamation frees memory pages in non-contiguous chunks.

4 HOW USEFUL ARE 1GB LARGE PAGES?

Hardware support for 1GB pages is not free, and the software running on x86 processors pays the price, irrespective of its use of 1GB pages. For example, modern Intel processors have 4-entry L1 TLB and 16-entry L2 TLB dedicated to 1GB pages. Those four L1 entries for 1GB pages are accessed on *every* load and store since the page size is not known apriori. Due to frequent accesses, L1 TLBs can contribute to a thermal hotspot in processors [50] and can account for 6% of a processor’s total power [53]. The presence of dedicated TLBs for 1GB pages adds to the cost. The continued increase in the number of TLB entries for 1GB pages would worsen it. It is, thus, natural to wonder if applications can benefit from 1GB pages. We analyze various applications under different execution scenarios to understand the usefulness of 1GB pages.

4.1 1GB pages in native execution

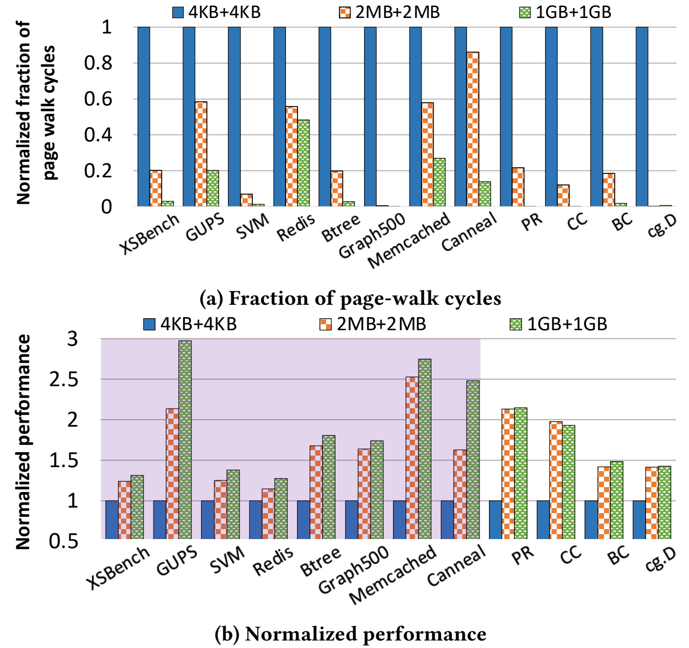
Figure 1a shows the normalized fraction of execution cycles spent on page walks for each application while using different page sizes. The four bars for each application represent walk cycles with ① 4KB pages, ② dynamically allocated 2MB pages via THP, ③ statically pre-allocated 2MB pages via libHugetlbfs, and, ④ statically pre-allocated 1GB pages via libHugetlbfs. The fourth bar *approximates* the performance achievable if the 1GB pages are deployed but not 2MB. Application-transparent dynamic allocation of 1GB pages (i.e., THP like) is not supported in Linux today.



(a) Fraction of page-walk cycles

(b) Normalized performance.

Figure 1: Analysis of different page sizes under native execution. Applications in shade benefit from 1GB pages.



(a) Fraction of page-walk cycles

(b) Normalized performance

Figure 2: Analysis of different page sizes under virtualization. Applications in shade benefit from 1GB pages.

Note that THP often performs as good as 2MB-libHugetlbfs. For Redis, THP reduces more walk cycles than libHugetlbfs. This is because Redis's stack memory is TLB sensitive, which cannot be mapped using libHugetlbfs. Importantly, THP does not require pre-allocation of physical memory, nor does it need users to statically decide which program segment(s) to be mapped with large pages.

Reduction in page walk cycles does not always lead to proportional performance gain on out-of-order cores. Rather, the speed up depends upon what portions of walk cycles are on the critical path of execution. Figure 1b shows the normalized performance. For all workloads, except Redis and Memcached, performance is calculated as the inverse of the execution time. For Redis and Memcached, we report performance in terms of throughput.

We observe non-negligible performance improvement (at least 3%) for eight applications (shaded left part of the figure) with use of 1GB pages over 2MB pages. For example, Canneal speeds up by 30% over THP. These eight applications' performance improves by 12.5%, on average, when 1GB pages are used via libHugetlbfs, relative to THP using 2MB pages. Rest of the applications witness benefits of using 2MB pages over 4KB, but barely gain any further with 1GB pages. This is not surprising; the walk cycles were already low with 2MB pages, and an out-of-order CPU could hide the rest. Henceforth, we thus focus on the first eight (shaded) applications.

We also observe that with THP, applications perform within 0.5% of that with libHugetlbfs using 2MB pages, even though it does not requires memory pre-allocation or user hints. This emphasizes the importance of THP for the wide deployment of 2MB pages; something that is lacking for 1GB pages.

4.2 1GB pages under virtualized execution

Two levels of translations under virtualization increase overheads. Each level may use a different page size. Thus, nine combinations of page sizes are possible. While we explored all, we discuss only 4KB-4KB, 2MB-2MB, and 1GB-1GB combinations where the first and second terms denote the page size used in the guest and in the host, respectively. We chose these configurations as they demonstrate the best performance achievable with a given page size.

Figure 2a shows the normalized fraction of page walk cycles under three different page size combinations. We notice significant reductions in walk cycles with 2MB and 1GB pages. For example, the fraction of walk cycles reduced by 80% for XSBench. Even a couple of 1GB page agnostic applications (e.g., PR, CC) experience a large reduction in walk cycles. Figure 2b shows the performance under virtualization. We observe that 1GB pages provide a bit more benefit here. The eight 1GB page sensitive applications speed up by 17.6% over 2MB pages, on average. The application BC, which did not benefit from 1GB pages under native execution, becomes slightly sensitive to 1GB pages under virtualization.

4.3 Importance of using all large page sizes

In the analysis so far, *only one* of the large page sizes was deployed as is the norm in today's software. However, we find that using all large page sizes together, can bring benefits that are not achievable using any one of them.

A virtual address range is *mappable* by a large page only if: ① it is at least as long as that large page, and ② the starting address is aligned at the boundary of that page size. All 1GB-mappable address ranges are, thus, mappable by 2MB pages but *not* vice-versa. When an application allocates, de-allocates, and re-allocates memory (e.g., Graph500), the virtual address space gets fragmented. Consequently, an application's entire address space may not be mappable by the largest page size.

We empirically find that often GBs of an application's virtual memory is 2MB-mappable but *not* 1GB-mappable. This depends on

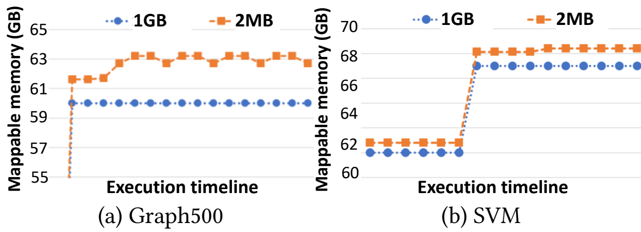


Figure 3: Total memory mappable with different page sizes.

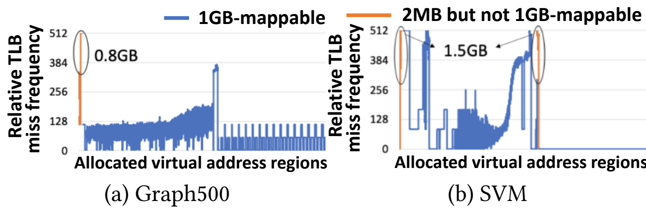


Figure 4: Relative TLB-miss frequency.

the application's memory allocation strategy – whether the application pre-allocates memory in large chunks (low *virtual memory* fragmentation) or incrementally allocate/de-allocate memory over time (high fragmentation).

We measure the size of 1GB-mappable and 2MB-mappable address regions with a kernel module that periodically scans the virtual address space of an application. Figure 3 shows the size of allocated virtual memory that is mappable with 2MB and 1GB over time for two representative applications – Graph500 and SVM. The x-axis represents the post-initialization execution timeline, and the y-axis is the size of allocated virtual memory in GB. The two lines in each graph show the amount of 1GB and 2MB-mappable memory. We observe that several GBs of memory is mappable by 2MB pages but not by 1GB (the gap between the two lines). If *only* 1GB pages are used, these memory regions have to be mapped with 4KB pages while wasting 2MB TLB resources.

We then analyzed the importance of mapping the regions that are un-mappable by 1GB pages, with 2MB. We wrote another module to measure the relative (sampled) TLB miss frequencies to the addresses that are mappable by 2MB but not by 1GB and those by both. We periodically un-set the access bits in PTEs (4KB) and then track which ones get set again by the hardware, signifying a TLB miss. Figure 4 presents the measurement. The x-axis shows the allocated virtual address regions, and the y-axis shows the relative TLB miss frequencies to pages in those regions. We use different colors for addresses that are 2MB-mappable but 1GB-unmappable, and those that are 1GB-mappable. We observe that the 1GB-unmappable regions witness frequent TLB misses. Particularly for Graph500, the spike in miss frequency on a relatively small 1GB-unmappable region (about 800MB) stands out (circled). Therefore, it is important to map these 1GB un-mappable address ranges with 2MB pages to reduce TLB misses.

Furthermore, it may not always be possible to map a 1GB-mappable address range with a 1GB page due to unavailability of 1GB contiguous physical memory. However, 2MB contiguous physical memory regions are more easily available. In short, it is *important to utilize all available page sizes.*

We also studied the usage of 1GB pages to Linux kernel itself. The kernel *direct* maps entire physical memory with the largest page size (here, 1GB). Using OS intensive workloads (e.g., apache web server and filebench [4, 54]), we found that 1GB pages improve kernel's performance by 2-3% over 2MB pages.

Summary of observations: ① A set of niche but important big-memory applications speeds up with 1GB pages over 2MB pages. In contrast, 2MB pages universally benefit memory-intensive applications. ② Application-transparent allocation of 2MB pages brings benefits of 2MB pages without user intervention – a capability that is lacking for 1GB pages. ③ It is important to utilize *all* large page sizes *not only* the largest.

Why the lack of 1GB software enablement? It is natural to wonder why there has not been an effort for wider enablement of 1GB pages. We hypothesize at least three reasons behind it. ① There was no significant quantification of the usefulness of 1GB pages, over and above 2MB pages. The above analysis tries to address that gap. ② Early processor designs had a limited number of TLB entries for 1GB pages (e.g., four in Sandy Bridge). Therefore, there was apprehension about the possible thrashing of TLB if 1GB pages were used by applications with poor locality [18, 44]. However, with newer processors accommodating 1GB pages in L2 TLB, we find no evidence of such thrashing in real-world applications. ③ Finally, with the advent of denser NVM technologies and five-level page tables, the need for low-overhead address translation has never been greater. Nevertheless, as the rest of the paper will demonstrate, managing 1GB pages in software needs special care, and thus, the bar for software enablement is non-negligible. However, given the necessity, it is imperative to eschew the software complexity for wider adoption of 1GB pages.

5 TRIDENT: DYNAMIC ALLOCATION OF ALL PAGE SIZES

We design and implement Trident in Linux for application-transparent dynamic allocation of all large page sizes on x86 processors without needing apriori reservation of physical memory. Trident minimizes TLB misses by mapping most of an application's address space with 1GB pages, failing which 2MB, and finally, 4KB pages are used.

Challenges: While the dynamic allocation of 2MB pages is not new, that for 1GB pages gives rise to new challenges. First, Trident needs to ensure a steady supply of free contiguous 1GB physical memory chunks even in the presence of fragmentation. We found that Linux's sequential scanning based memory compaction for creating 2MB chunks is not scalable to 1GB as it incurs excessive data copying.

Linux tracks only up to 4MB free physical memory chunks. However, the dynamic allocation of 1GB pages would require maintaining free memory up to 1GB granularity. Allocating 1GB pages during page faults are much slower than that for 2MB or 4KB pages due to the latency of zeroing entire 1GB memory. Low-latency 1GB page fault is necessary for an aggressive deployment of 1GB pages. Finally, Trident should map a virtual address range with the largest *large page* size deployable at any given time. It should then periodically look for opportunities to promote address ranges mapped with a smaller large page to a larger one wherever possible.

5.1 Design and implementation

At a high-level, Trident modifies four major parts of Linux. ① It enhances Linux to track up to 1GB free physical memory chunks. ② It updates the page fault handler to allocate a 1GB page on a fault when possible and fall back to smaller pages if needed. ③ Trident extends THP's khugepaged daemon thread to promote virtual address ranges to 1GB pages when possible. ④ Trident employs a novel smart compaction technique for a steady supply of 1GB physical memory chunks at low overhead.

5.1.1 Managing 1GB physical memory chunks. Linux's buddy allocator keeps an array of free lists of physical memory chunks of sizes 4KB up to 4MB in the power of 2 [9]. When free memory is needed, the buddy allocator provides a memory chunk from one of its lists based on the request size. Freed physical memory is returned to the buddy, and coalesced with neighboring free memory chunks to create larger ones. Unfortunately, the buddy only keeps track of regions up to 4MB. We thus extended it to include separate lists for tracking up to 1GB memory chunks.

5.1.2 Allocating large pages during page fault. Like THP, Trident allocates large pages either ① during a page fault (e.g., when a process accesses a virtual address for the first time) or ② later during attempts to promote an address range to a large page. We here detail the former. If the faulting virtual address falls in a 1GB-mappable address range, then Trident attempts to map it with a 1GB page. If it fails, Trident attempts to map the address with a 2MB page, and on failure, with 4KB. If the faulting address falls in a region that is 2MB-mappable but not 1GB-mappable Trident tries to map it with 2MB.

Asynchronous zero-fill: A 1GB page fault takes around 400 milli-seconds; compare that to 850 micro-seconds for 2MB. The additional latency is due to zero-filling of 1GB memory instead of 2MB†. We instead employ asynchronous zero-fill to speed up 1GB faults. A kernel thread periodically zero-fills free 1GB regions and Trident allocates a zero-filled region, if available. This reduces the average 1GB fault latency from 400 milli-seconds to 2.7 milli-seconds. While prior works [1, 42] explored asynchronous zero-fill for 2MB pages, we find it to be a necessity for 1GB pages in latency-critical workloads. For example, the boot time of a 70GB virtual machine dropped from 25 seconds to 13 seconds with asynchronous zero-fill.

Table 3 shows the portion of applications' memory footprints mapped by 1GB and 2MB pages under Trident's various allocation mechanisms that we will discuss in this section. The first data column shows applications' memory footprint. The first set of sub-columns capture the behavior with *un-fragmented* physical memory while the next set represents that under *fragmentation*. Physical memory is said to be fragmented if the free memory is scattered in small holes i.e., non-contiguous. Typically, physical memory is un-fragmented *only* if the system is freshly booted and/or there is little memory usage. But, the memory gets quickly fragmented as applications/OS allocate and de-allocate memory.

The sub-columns under un-fragmented shows that the page fault handler alone (Page-fault only) can map a large portion of application's memory with 1GB pages for three out of eight applications (XSBench, GUPS, Graph500). If an application pre-allocates memory in large chunks, then the fault handler would often find the faulting address to be in a 1GB-mappable region, and use 1GB pages. However, Redis and Memcached incrementally allocate memory while inserting key-value pairs. Thus, the fault handler could map a small portion of its memory with 1GB pages. SVM, Btree, Canneal also, do not pre-allocate their entire memory needs.

† Zero-fill ensures application's leftover data does not leak out.

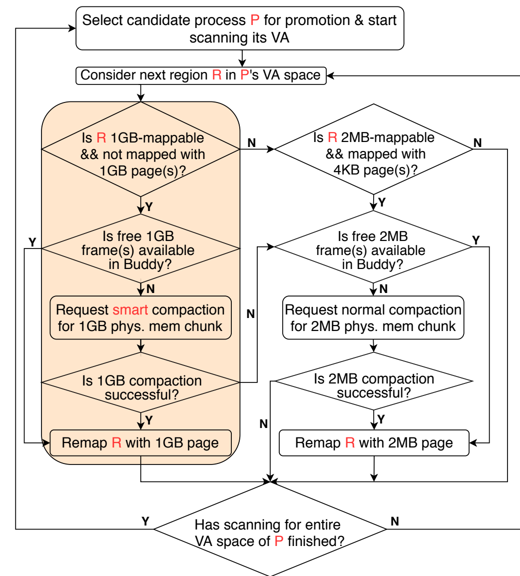


Figure 5: Trident's large-page promotion algorithm.

The behavior is different if physical memory is fragmented. Even if the fault handler finds a 1GB-mappable address range, it is unlikely to find a free 1GB physical memory chunk. Thus, it often falls back to smaller pages. This is evident from data in sub-columns for "Page-fault only" under fragmentation (Table 3) as only a few 1GB pages are allocated.

5.1.3 Large page promotion. If an application does not pre-allocate memory or when physical memory is fragmented, it becomes important to later re-map (promote) address ranges to larger pages, when possible. Trident extends THP's khugepaged thread to promote to both 1GB and 2MB pages.

Figure 5 shows a flowchart of Trident's page promotion algorithm (changes to THP are shaded). khugepaged first selects a candidate process for page promotion and sequentially scans its virtual address space. During scanning, Trident looks for 1GB-mappable virtual address ranges that are mapped with smaller pages. Subsequently, it looks for 2MB-mappable regions mapped with 4KB pages. If a candidate 1GB-mappable range is found, khugepaged requests the buddy allocator for a free 1GB physical memory chunk. If a 1GB chunk is unavailable, khugepaged requests compaction of the physical memory to create one. Trident extends THP's compaction functionality to create a 1GB physical memory chunk (will be detailed shortly). If the compaction fails, it attempts to map it with 2MB pages (if not already mapped with 2MB). Trident's policy of preferring 1GB pages but falling back to 2MB pages makes the most out of TLB resources.

Table 3's sub-columns under "normal compaction" shows the number of 1GB and 2MB pages allocated when the above-mentioned

Table 3: Comparison of 1GB and 2MB pages allocated via different mechanisms employed in Trident

	Memory footprint (in GB)	Un-fragmented (all data in GB)						Fragmented (all data in GB)					
		Page-fault only		Promotion Normal compaction		Promotion Smart compaction		Page-fault only		Promotion Normal compaction		Promotion Smart compaction	
		1GB	2MB	1GB	2MB	1GB	2MB	1GB	2MB	1GB	2MB	1GB	2MB
XSBench	117	114	2.94	116	1.2	116	1.2	6	5.3	79	38.1	80	37.1
GUPS	32	31	1	31	1	31	1	9	2.5	31	1	31	1
SVM	68.5	54	14.3	65	3.5	65	3.5	6	5	53	12.2	54	9.9
Redis	44	0	0.5	39	3.4	39	3.4	0	0	25	10.3	28	14.3
Btree	25	0	16.7	16	5.8	16	5.8	0	11.7	8	12.73	12	8.91
Graph500	63.5	59	4.01	60	3.35	60	3.35	5	5.8	37	24.2	38	23.6
Memcached	137	16	121	121	16	121	16	9	60	12	55	16	60
Canneal	32	8	1	30	2	30	2	6	1	6	21	8	22

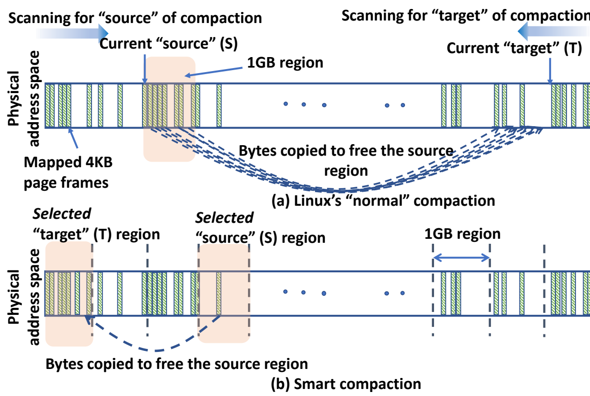


Figure 6: Comparison between Linux's (traditional) compaction of physical memory and smart compaction.

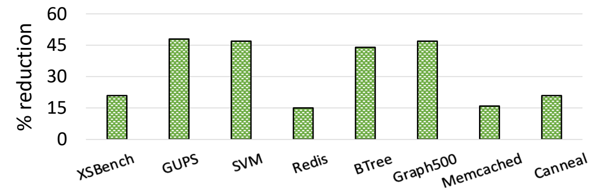


Figure 7: Reduction in bytes copied by smart compaction.

promotion policy is applied along with the page fault handler (under both un-fragmented and fragmented memory). For example, in the un-fragmented case, khugepaged is able to promote about 39GB of memory using 1GB pages for Redis, when the fault handler alone failed to allocate even a single 1GB page. SVM, Canneal also enjoyed many more 1GB pages due to page promotion.

When the physical memory is fragmented, page promotion helps applications get some 1GB pages, although slightly smaller in number compared to the un-fragmented case. For example, 1GB pages allocated to SVM drop from 65 to 53. This is expected; free 1GB memory chunks are scarce even after compaction.

Overheads of compaction for 1GB, however, can negate benefits of 1GB pages. Creating even a *single* 1GB chunk often requires significant memory copying. Copying data creates contention in memory controllers and pollutes caches. It also requires scanning large portions of physical memory. The applications threads could get a smaller fraction of CPU cycles as they can contend with kernel threads performing compaction. In short, it is necessary to reduce the cost of compaction for 1GB pages to harness its benefits.

Smart compaction: We, thus, propose a new compaction technique, called smart compaction, to reduce the cost of 1GB compaction while creating enough 1GB physical memory chunks. The primary goal is to reduce the number of bytes copied. This directly reduces the cost of compaction.

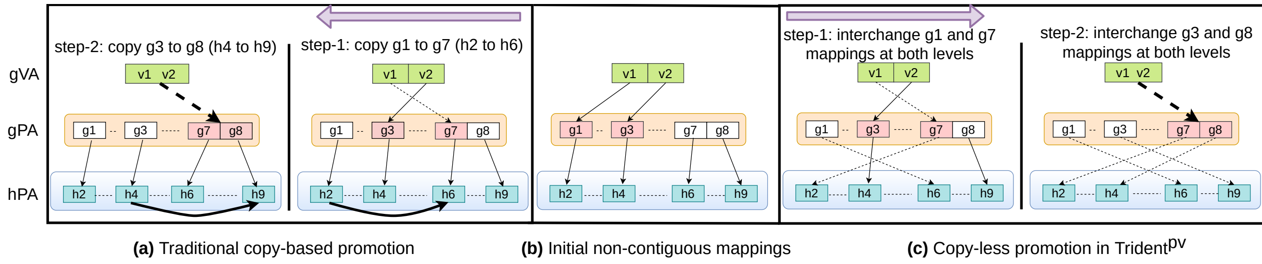
Figure 6 illustrates the difference between normal compaction as employed in Linux today and the smart compaction employed in Trident. Figure 6(a) shows the working of the normal compaction. On a compaction request, the khugepaged thread starts sequentially scanning physical memory from where it left last time it attempted to compact (remembered in *source* pointer). Scanning starts from the low to high physical addresses. As it finds an occupied physical page frame (4KB) it copies its contents to a free page frame found by scanning in the opposite direction from the *target* pointer. This continues until a free memory chunk of the desired size (e.g., 2MB) is created, or the entire memory is scanned without success.

We observe that this strategy is agnostic to how full or empty a physical memory region is. Consequently, this leads to redundant copying. Let us consider the example in Figure 6(a). The 1GB region starting at address S is mostly occupied and has only 256 free 4KB page frames. Thus, to free that 1GB region, Linux would require copying 999MB of data (512 × 512 - 256 4KB pages). Instead, if a *mostly* free region was freed, then the number of bytes copied would be much smaller. While such sub-optimal compaction could be fine for 2MB, it is *not* so for 1GB, as data copying increases with the page size.

Moreover, if the scan encounters a page frame with unmovable contents (e.g., inodes, DMA buffers), then all copying so far for a region, is wasted [42]. A free chunk cannot have any unmovable contents. The probability of encountering unmovable contents is much more for a 1GB region.

To address these shortcomings, we propose smart compaction. The key idea is to divide the physical memory into 1GB regions and *select* (*not* scan for) a region with the least number of occupied page frames for freeing (i.e., the source of copying). Similarly, a region with the most number of occupied page frames is preferred as the target for copying. This strategy minimizes data copy. We also track if a given 1GB region contains any unmovable contents. We avoid selecting regions with unmovable content for freeing (i.e, source). This eliminates unnecessary data copying in futile compaction attempts.

To implement the above idea, we first introduced two counters for each 1GB physical memory regions. One counter tracks the



(a) Traditional copy-based promotion (b) Initial non-contiguous mappings (c) Copy-less promotion in Trident^{pv}

Figure 8: Traditional copy-based versus Trident^{pv}'s copy-less page promotion.

number of free page frames, and the other one tracks the number of unmovable pages within a region. Whenever a page is returned to the buddy allocator (i.e., freed), we increment the counter for free frames of the encompassing 1GB region. Further, we decrement the counter for unmovable pages if the freed page frame(s) contained unmovable data. Whenever a page frame(s) is allocated from the buddy allocator the free counter for the encompassing region is decremented. We increment its unmovable page counter if the allocated page frame(s) would contain unmovable data (e.g., requested for allocating kernel data structures). Note that a 1GB region can also have a 2MB page allocated within it. We treat it as 512 base pages for ease of keeping statistics.

As depicted in Figure 6(b) the smart compaction starts by selecting a 1GB region with largest number of free page frames and without any unmovable pages as the source (S). It then selects a target region (T) to move the contents of occupied page frames in the source. The region with the least number of free page frames is selected as the target. It can happen that T may not have enough free frames to accommodate all of S's page frames. If so, a region with next least number of free frames is selected to accommodate the remaining pages (and, so on).

The sub-columns for smart compaction in Table 3 shows the number of 1GB and 2MB pages that were allocated under un-fragmented and fragmented physical memory. The number of 1GB pages allocated to each application is the same as that under normal compaction in the un-fragmented case. Under fragmentation, smart compaction typically provides even more 1GB pages. This is because the smart compaction always selects a 1GB region that is easiest to free, and thus, compaction succeeds more often.

Figure 7 shows the percentage reduction in the number of bytes copied with smart compaction over normal compaction. This measurement is performed when physical memory is fragmented as otherwise compaction is unnecessary. We observe that smart compaction often reduces the number of bytes copied by up to 85%. This demonstrates that smart compaction performs less work to create the same or more number of 1GB chunks. Only for XSBench, the improvement is less. XSBench uses a large fraction of total memory in the system and thus, even the ideal compaction algorithm would not be able to avoid data copy under fragmentation. All compaction algorithm will behave the same if the physical memory is fragmented, and an application needs all memory.

6 TRIDENT^{PV}: PARAVIRTUALIZING TRIDENT

Under virtualization, Trident can be deployed both in the guest OS and in the hypervisor to bring benefits of dynamic allocation of all page sizes, including 1GB pages, to both the levels of address translation. We observe that it is possible to further optimize certain guest OS operations with paravirtualization.

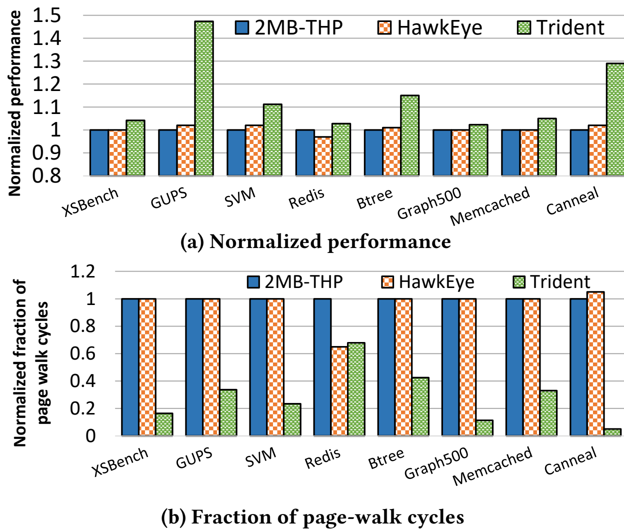
The guest OS *copies* contents of memory pages to ① to compact gPAs, and ② to promote address mapping between gVA and gPA to larger pages. While the cost of copying 4KB pages is not high, copying 2MB pages in order to compact or promote them to a 1GB page is slow. We observe that the effect of copying guest physical pages can be mimicked by simply altering the mapping between corresponding gPAs and hPAs. This *copy-less* approach quickens both compaction and 1GB page promotion in the guest but needs paravirtualization. We call this extension Trident^{pv}.

For brevity, we explain the idea behind Trident^{pv} with the help of large page promotion only (Figure 8). Let us assume that two contiguous guest virtual pages, v1 and v2, are currently mapped to two non-contiguous smaller pages g1 and g3 in guest physical memory (Figure 8(b)). For simplicity, we assume that a large page is twice the size of a small page. To remap gVA encompassing v1 and v2 with a large page, the guest OS first copies their content to two contiguous guest physical pages – g7 and g8. It then updates the mapping between gVA and gPA. This traditional way of promoting large pages by copying contents is shown in Figure 8(a).

Figure 8(c) shows Trident^{pv}'s approach for page promotion without actual copy. Instead of copying g1 to g7, the hypervisor exchanges the gPA to hPA mappings for g1 and g7. After the exchange, g1 maps to h6 and g7 maps to h2. Since, h2 contains the data originally mapped by g1, this is same as copying g1 to g7. Similarly, the hypervisor exchanges the gPA to hPA mappings for g3 and g8 to create the effect of copying g3 to g8. Later, gVA encompassing v1 and v2 is mapped by the guest with a large page to contiguous gPA encompassing g7 and g8.

The guest OS and the hypervisor need to coordinate for copy-less page promotion and, thus, the need for paravirtualization. Specifically, the guest OS supplies the hypervisor with a list of source and target guest physical pages via a hypercall. The hypervisor then updates the mapping from gPA to hPA in the manner explained above to create the effect of copying guest physical pages. Besides promotion, Trident^{pv} uses the same hypercall for compacting guest physical memory to create 1GB pages in the guest.

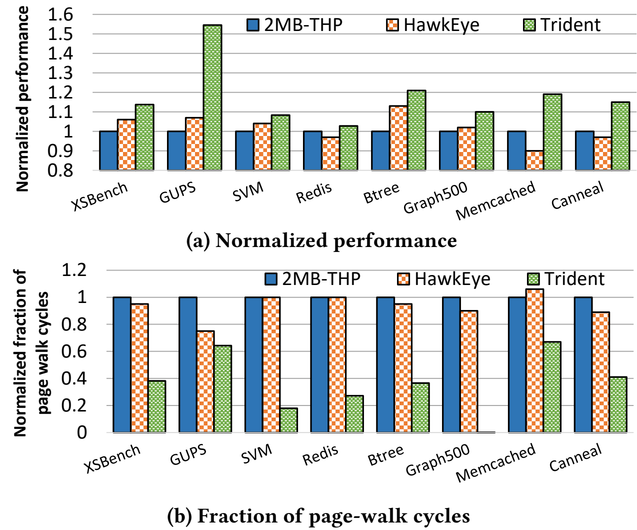
While promising, the cost of hypercall (≈ 300ns) to switch between guest and the hypervisors can outweigh the benefits of copy-less promotion. We thus batch requests for multiple page mapping exchanges in a single hypercall. Since a 1GB page is promoted via 512 2MB pages, batch size is known apriori and statically configured. We pre-define two 4KB pages for passing the list of page



(a) Normalized performance

(b) Fraction of page-walk cycles

Figure 9: Performance under no fragmentation.



(a) Normalized performance

(b) Fraction of page-walk cycles

Figure 10: Performance under fragmentation.

addresses to exchange between the guest and the hypervisor. One page contains source gPAs (here, g1 and g3) and the other contains the target gPAs (here, g7 and g8). In a single hypercall it is thus possible to request exchange for all 512 page addresses. Thus, a single hypercall is sufficient to promote entire 1GB region in gVA mapped with 2MB pages. The hypercall returns after switching all the requested pages or logs any failure in the same shared page used for passing list of pages. On failure, the guest falls back to individually copy contents of pages.

We empirically found that promoting 2MB pages to a 1GB page in the guest takes ≈ 600ms in the copy-based technique. Without batching, Trident^{pv} can promote the same in less than 30ms while batching reduces the time to ≈ 500μs. Note that Trident^{pv}'s copy-less promotion is less useful for promoting 4KB pages to 2MB since the cost of copying 4KB pages is not significant. Hence, we employ copy-less promotion and compaction for 1GB pages only.

7 EVALUATION

Our evaluation answers the following questions. ① Can Trident improve performance of memory-intensive applications over Linux's default THP and over a recent work HawkEye [42]? ② What are the sources of performance improvement for Trident? ③ How does Trident perform under virtualization? ④ Finally, how does Trident^{pv} impact page promotion/compaction the guest OS?

Performance under un-fragmented physical memory: Figure 9a shows the normalized performance for three configurations (higher is better) – ① Linux's THP (baseline), ② HawkEye, and ③ Trident. HawkEye is a recent related work that improved upon THP and other previous works (e.g., [36]). It does so by efficiently allocating 2MB pages to address regions that experience most TLB misses [42]. This allows us to compare against the academic state-of-the-art proposal. For each application, there are three bars in the cluster corresponding to three configurations. The height of each bar is normalized to the performance of the application under THP (Linux's default configuration). Measurements in Figure 9a were performed with un-fragmented physical memory.

First, we observe that Trident improves performance over Linux's THP by 14%, on average and up to 47% for GUPS. Applications like XSBench, SVM, Btree and Canneal witnessed 4.1%, 11.2%, 15% and 30% performance improvement, respectively. Even if we exclude the micro-benchmark GUPS, performance improvement is 12%, on average, over THP. Next, we observe that Trident also outperforms HawkEye by 14%, on average. This is expected due to the similarities between huge page management in Linux and HawkEye. Both these systems employ 2MB pages aggressively in the page-fault handler when physical memory is un-fragmented.

Performance under fragmented physical memory: Arguably, performance analysis under fragmented physical memory paints a more realistic execution scenario. Figure 10a shows the normalized performance under fragmented physical memory for the same three configurations as before. Trident speeds up applications *even more* under fragmentation. This is unsurprising since Trident's smart compaction adds a further edge here. On average, it improves performance by 18% over THP and GUPS quickens by over 50%. Even excluding GUPS, the improvement is 13% over THP.

Trident also outperforms HawkEye in all cases. In some cases under fragmentation, HawkEye performed worse than THP (e.g., Redis, Memcached). This might happen for large memory applications due to: ① CPU overhead of kbinmanager kernel thread that estimates relative TLB miss rates in HawkEye and ② potential lock contention between kbinmanager and khugepaged kernel threads and the page-fault handler.

We also measured how often the fragmented physical memory prevents Trident from mapping an address range with a 1GB page. Table 4 shows the percentage of attempts to allocate a 1GB page that fails due to fragmentation. The "NA"s under page fault for Redis and Btree signify that fault handler never attempts to allocate 1GB pages for them due to lack of 1GB-mappable virtual address ranges during faults. We observe that 71-94% of 1GB page allocations fail due to lack of contiguous physical memory. Even during promotion, 1GB allocations fail often. This reinforces the need to utilize *all* large page sizes. Even if the largest page size cannot be used, a smaller large page (2MB) could be deployed.

Table 4: Percentage 1GB memory allocation failures

	Page fault	Promotion		Page fault	Promotion
XSBench	94%	32%	GUPS	71%	0%
SVM	88%	19%	Redis	NA	36%
Graph500	91%	38%	Btree%	NA	25%
Memcached	43%	81%	Canneal	12%	92%

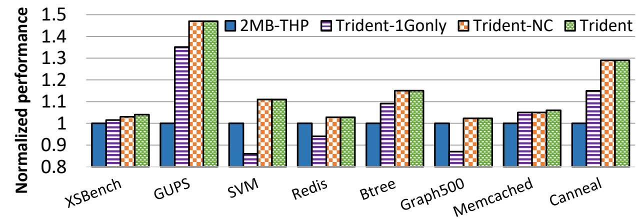
Table 5: Tail latency (ms) for Redis and Memcached

	Redis			Memcached		
	4KB	THP	Trident	4KB	THP	Trident
No-fragmentation	47.3	50.3	46.6	1.53	1.52	1.53
Fragmentation	53.3	53.3	52	1.55	1.55	1.54

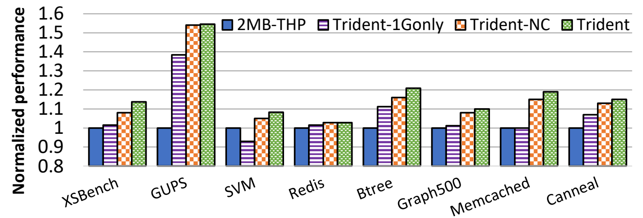
Impact on page walk cycles: Figure 9b and Figure 10b show the normalized fraction of walk cycles for THP, HawkEye, and Trident under un-fragmented and fragmented physical memory, respectively. The reductions in the page walk cycles with Trident over THP are significant – 38-85% and 40-97%, under no fragmentation and fragmentation, respectively. Across all configurations, the relative speedups largely correspond to relative reductions in walk cycles.
Impact on tail latency: Tail latency is an important metric for transnational interactive applications (e.g., Redis, Memcached) that should abide by strict SLAs [36]. Table 5 reports 99 percentile latency of Redis and Memcached under different configurations. Trident does not hurt tail latency relative to both 4KB and THP even though it employs 1GB pages dynamically. Trident reduced TLB misses in the critical path and ensured compaction, promotion and zeroing of 1GB pages happen in background to avoid affecting the tail latency.
Evaluating Trident's design components: Two of the key aspects of Trident's design philosophy are ① the use of *all* three page sizes, including 1GB pages, and ② smart compaction. It is natural to wonder how important are they in Trident's performance?

Figure 11 teases out the impact of these two factors in Trident's performance, with and without fragmentation (subfigures). Specifically, we introduce two new configurations. Trident-1Gonly denotes the configuration where Trident is disallowed to use 2MB pages. The difference between Trident-1Gonly and Trident highlights the importance of leveraging all large page sizes. Trident-NC denotes the configuration where Trident is allowed to use all three page sizes but barred from employing the smart compaction. Instead, it uses normal compaction available in Linux. The difference between Trident-NC and Trident shows the direct performance implications of smart compaction, beside reducing data movement. Smart compaction enables Trident to compact fragmented physical memory faster and thus, applications can get 1GB pages sooner in their execution, aiding performance.

First, we focus on the performance when the memory is unfragmented (e.g., a freshly booted system) in Figure 11a. We observe that there is a significant performance gap between Trident-1Gonly and Trident, justifying the need for using all three page sizes. Trident-1Gonly loses performance even relative to THP for several applications (e.g., Graph500, SVM). In hindsight, this is expected. Our analysis in § 4.3 revealed that these applications have significant portions of their virtual memory that is 2MB-mappable but not 1GB-mappable. Further, these portions also witness a relatively larger number of TLB misses. Trident-1Gonly is forced to



(a) Normalized performance under no fragmentation



(b) Normalized performance under fragmentation

Figure 11: Analysis of different components of Trident.

map these 1GB-unmappable regions with 4KB pages and thus experiences higher translation overheads compared to Trident that could deploy 2MB pages. In the process, Trident-1Gonly's benefits from using 1GB pages are more than negated by the overheads of mapping frequently accessed memory with 4KB pages.

Next, we observe no difference in performance between Trident-NC and Trident, i.e., no impact of smart compaction when the memory is un-fragmented (Figure 11a). This is expected since compaction is not required when the memory is un-fragmented (contiguous). Figure 11b shows the behavior when the physical memory is fragmented. Here, we see significant performance improvement with smart compaction for several applications. For example, smart compaction alone speeds up XSBench by 6%. Similarly, smart compaction is instrumental in improving performances of SVM, Btree, Graph500, Memcached, and Canneal by 2-5%. Only GUPS and Redis show no significant performance uplift with smart compaction. In short, the use of all large page sizes and smart compaction both play major roles in Trident's performance.
Comparison with static allocation: Since Trident is a dynamic page allocation technique, we compared it against alternative dynamic approaches such as THP, HawkEye. However, one may wonder how Trident compares against static technique of allocating 1GB pages via 1GB-Hugetlbfs (§ 4.1).

Unfortunately, 1GB-Hugetlbfs needs *apriori reservation* of contiguous physical memory for 1GB pages. Consequently, it fails when the memory is fragmented, as it often happens in real execution scenarios. Thus, 1GB-Hugetlbfs can be compared only when the physical memory is un-fragmented as in a freshly booted system (performance reported in Figure 1b). Here, also, Trident performs 3% better than 1GB-Hugetlbfs, on average, even though Trident does not require user intervention, recompilation or apriori reservation of the physical memory. This was possible since Trident could even map the stack portions using large pages, unlike 1GB-Hugetlbfs, and applications such as GUPS and Redis are sensitive to TLB misses on the stack region. Only in one application Btree, 1GB-Hugetlbfs performs better than Trident. This is because Btree allocates virtual memory incrementally over time. Thus, 1GB pages

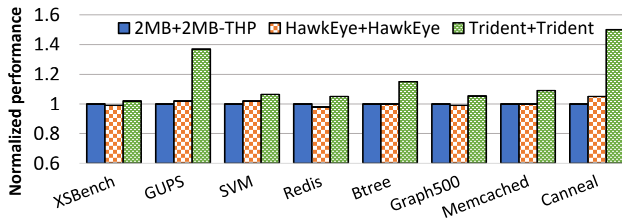


Figure 12: Performance under virtualization.

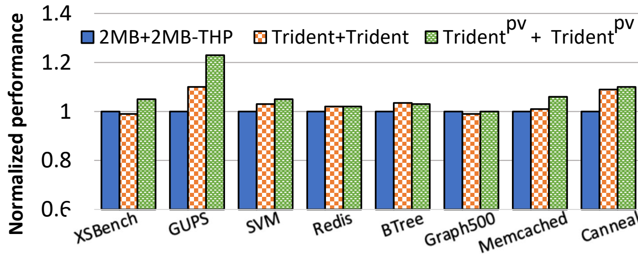


Figure 13: Trident^{pv}'s performance under fragmented gPA.

are allocated only during page promotion by Trident and not during page faults upon first access. In contrast, 1GB-Hugetlbfs uses 1GB pages irrespective of virtual memory allocation size at the cost of bloating memory footprint.

Performance under virtualization: We measured the performance of applications running inside a virtual machine with Trident deployed both at the guest OS and at the hypervisor (KVM). We do not fragment memory. For comparison, we do the same with HawkEye too. Figure 12 shows the speedups, normalized to THP deployed in the guest OS and KVM. Under virtualization, Trident improves performance by 16% on average, over THP and by 15% over HawkEye. Canneal saw biggest improvement (50%), but other applications also benefited significantly. For example, SVM and Graph500 witnessed 6% improvement each.

Performance with Trident^{pv}: When the gPA is fragmented, the guest OS must compact and promote pages using THP's khugepaged thread. However, a significant CPU usage in the guest OS could mean wasted vCPU time (cost) for a tenant in the cloud. In fact, Netflix reported how their deployments on Amazon EC2 can get adversely effected by high CPU utilization due to THP's threads [33]. We, therefore, evaluate Trident^{pv} with fragmented gPA but limit khugepaged's CPU utilization in the guest to maximum of 10% of a vCPU. This setup helps to find out whether Trident^{pv}'s faster copy-less promotion/compaction can be useful to use 1GB pages.

Figure 13 shows the performance of Trident and Trident^{pv} normalized to THP. Trident^{pv} is more effective than Trident for XSBench, GUPS, Memcached, and SVM by 5%, on average and by up to 10%. We observe that Trident^{pv} does not always improve performance over Trident. Recall that Trident^{pv}'s hypercall-based copy-less approach is quicker than the copy-based approach only during promotion/compaction of 2MB pages to 1GB pages. Otherwise, the overhead of the hypercall and that of altering PTEs overshadows the benefits of avoiding copy. In applications such as BTree, Graph500, Canneal, 4KB pages are often promoted directly to 1GB pages without needing to go via 2MB pages limiting Trident^{pv}'s scope for improving their performance.

Memory bloat: Large pages are well-known to increase memory footprint (bloat) due to internal fragmentation. Larger the page size more is the bloat. Trident causes bloat in two out of eight workloads. It adds 38GB and 13GB bloat for Memcached and Btree over THP. We were able to recover the bloat by simply incorporating HawkEye's technique for dynamic detection and recovery of bloat by demoting large pages and de-deuplicating zero-filled small pages [42]. However, we do not make any new contribution here and the tradeoff between bloat and large pages is well explored in the literature [36, 42]

8 RELATED WORK

Address translation overheads is a topic of several recent research efforts [12, 13, 18, 21, 30, 32, 35, 37, 45, 46, 49, 51, 52, 56].

Proposals that require hardware support: Hardware optimizations focus on reducing TLB misses and on accelerating page walks. Multi-level TLBs, and multiple page sizes are found in today's commercial CPUs [3, 26]. Further, page walk caches are used to make page walks faster [16, 22].

Direct segments [19] can significantly reduce address translation overheads through segmentation hardware. Coalesced-Large-Reach TLBs increase TLB coverage through contiguity-aware hints encoded in the page tables [48]. This approach can also be combined with page walk caches and large pages [22, 26, 47]. POM-TLB reduces page walk latency by servicing a TLB miss using a single memory lookup with a large in-memory TLB [51]. SpecTLB speculatively provides address translation on a TLB miss by guessing virtual to physical address mappings [17]. ASAP prefetches translations to reduce page walk latency to that of a single memory lookup [39]. It first orders page table pages to match that of the virtual memory pages and then uses a base-plus-offset arithmetic that directly indexes into the page tables. Large page support for non-contiguous physical memory has also been proposed [28]. Tailor page sizes use whatever contiguity OS can afford to allocate [34]. Park et. al. proposed to use large pages to allocate page table entries to reduce the height of the page table [46]. Skarlatos et. al. demonstrated benefits of using Elastic cuckoo hash table as the page tables data structure instead of radix tree [52]. In contrast, Trident needs no new hardware; new hardware enhancements are orthogonal to Trident's goal of fully utilizing current hardware.

Proposals with only software support: Software-only solutions mainly focus on better use of large pages. Navarro et. al., proposed reservation-based large page promotion in FreeBSD as compared to proactive large page allocation in Linux [40]. Ingens proposes to mix THP's aggressive large page allocation with FreeBSD's conservative approach to reduce memory bloat and latency while still leveraging large pages. Illuminator showed how unmovable kernel objects hinder compaction [43]. Quicksilver uses hybrid strategies across different stages in the lifetime of a large page. Specifically, it employs aggressive allocation, hybrid preparation, relaxed mapping creation, on-demand mapping destruction and preemptive deallocation to achieve high performance and lower latency and bloat [61]. Carrefour-LP showed how large pages can degrade performance in NUMA systems due to remote DRAM accesses and unbalanced traffic [31]. Trident is complementary to these works; while these works focused on KBs and MBs-sized pages, Trident focuses on

1GB pages which brings its unique challenges for compaction and page promotion. Many insights from these works on 2MB pages are applicable to Trident too e.g., HawkEye's fine-grained page promotion and Ingens's adaptive approach of balancing trade-offs with large pages can be applied to Trident too.

Translation Ranger proposed a new OS service that actively creates contiguity in physical memory [60]. Parts of this work have been posted on Linux mailing lists. For example, in-place compaction of physical memory can be performed using the approach discussed in [58]. This functionality, however, requires application modifications and is unsuitable for applications that incrementally allocate memory (e.g., Redis). Another part of this work provided a preliminary patch set to extend Linux's THP support to 1GB pages [59]. Their approach, however, *always promote* address regions to 1GB pages through a background service – even if contiguous memory is available during page faults. Consequently, it incurs unnecessary data movement even when physical memory is un-fragmented or moderately fragmented. In large-memory systems, this could amount to migrating terabytes of data to allocate 1GB pages. Further, the inability to immediately deploy 1GB pages, even when it was possible, leaves significant performance on the table. This patch set, however, is not fully functional and frequently causes kernel crashes when the physical memory is fragmented. We are therefore unable to present a full quantitative comparison. Wherever the patch set ran, Trident outperformed it by 2%-15% by deploying all large page sizes more efficiently.

9 CONCLUSION

While OS support for 2MB pages has matured over the years, 1GB pages have received little attention despite being present in the hardware for a decade. We propose Trident to leverage architectural support of all page sizes available on x86 processors, while also dealing with the latency and fragmentation challenges. Our evaluation shows that 1GB pages, in tandem with 2MB pages, significantly speed up several applications. Further, the paravirtualized extension of Trident, called Trident^{pv}, can effectively virtualize 1GB pages with copy-less page promotion and compaction. Trident's 18% performance gain over Linux THP is likely to motivate researchers to further explore the role of large pages, beyond 2MB.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their constructive suggestions. We also thank the members of our Computer Systems Lab for feedback on earlier versions of this work, and Sujay Yadalam for suggesting the name Trident. This work is supported by research grants from Semiconductor Research Corporation (grant number 2019-IR-2925), and from VMware Inc. Arkaprava is supported by a Young Investigator Fellowship by Pratiksha Trust, Bangalore. Ashish Panwar is supported by the Prime Minister's Fellowship Scheme for Doctoral Research, co-sponsored by Confederation of Indian Industry, Government of India, and Microsoft Research India.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact provides source of Trident (modified Linux kernel v4.17.3) and evaluated benchmark binaries with their input files where appropriate. The measurement infrastructure is provided through bash and python scripts which allow reproducing experimental results on an Intel Skylake machine with 18 cores (36 threads) and 192GB of main memory.

A.2 Artifact Check-List (Meta-information)

- **Compilation:** GCC version 7.4.0.
- **Binary:** Included for x86-64.
- **Data set:** Scripts provided to generate datasets.
- **Run-time environment:** Provided by the supplied Linux kernel binaries for x86-64 hardware, source code given. Scripts require sudo privileges.
- **Execution:** Using bash and python scripts on a Linux/KVM platform. Scripts to be executed on the host.
- **Output:** A CSV file for each experiment.
- **Disk space required:** 280GB including all datasets. 80GB excluding fragmentation related experiments.
- **Time needed to prepare workflow:** 3-4 hours.
- **Time needed to complete experiments:** 6-7 days.
- **Archived:** Yes. DOI: 10.5281/zenodo.5149740

A.3 Description

A.3.1 Availability. All scripts are available in the GitHub repository https://github.com/csl-iisc/Trident-MICRO21-artifact. All sources are included as public git submodules. The artifact is also available at https://doi.org/10.5281/zenodo.5149740.

A.3.2 Hardware dependencies. We recommend an Intel Skylake node with 18 cores (36 threads) and 192GB memory. Other x86-64 servers with similar memory and compute capability should produce comparable results.

A.3.3 Software dependencies. The compilation environment, binaries and scripts assume Ubuntu 18.04 LTS running Linux Kernel v4.17.3. Similar Linux distributions are also expected to work. In addition to the packages shipped with Ubunty 18.04 LTS, additional packages need to be installed as follows:

```
$ sudo apt install build-essential bison bc \
      libncurses-dev flex libssl-dev automake \
      libelf-dev libnuma-dev python3 git \
      wget libncurses5-dev libevent-dev \
      libreadline-dev libtool autoconf \
      qemu-kvm libvirt-bin bridge-utils \
      virtinst virt-manager hugepages \
      libgfortran3 libhugetlbfs-dev sshfs
```

A.4 Installation

To install, download the artifact from Zenodo or clone the GitHub repository. The repository contains scripts required to run the artifact along with all pre-compiled binaries. Sources (i.e., Trident and

HawkEye kernels) are included as public submodules in `./sources/`. Pre-compiled binaries are placed in `./bin/`.

A.4.1 Download artifact and compile kernel images. Execute:

```
$ git clone https://github.com/csl-iisc/ \
            Trident-MICRO21-artifact
$ cd Trident-MICRO21-artifact
$ PROJECT_DIR=$(pwd)
$ git submodule init
$ git submodule update

$ cd $PROJECT_DIR/sources/Trident
$ git fetch --all; git checkout trident
$ make menuconfig; make -j $(nproc)
$ sudo make modules_install; sudo make install

$ cd $PROJECT_DIR/sources/Trident
$ git fetch --all; git checkout hawkeye
$ make menuconfig; make -j $(nproc)
$ sudo make modules_install; sudo make install
```

A.4.2 Install and configure a virtual machine. Install a virtual machine using libvirt on your test machine. An example using command line installation is provided below (choose ssh-server when prompted for package installation). Once installed, run the second command to generate three VM configurations. The scripts will configure the number and affinity of vCPUs, memory size and default large page size. The appropriate configuration will be loaded by the run scripts themselves.

```
$ virt-install --name trident --ram 4096 \
   --disk path=~/trident.qcow2,size=50 \
   --vcpus 4 --os-type linux --os-variant \
   generic --network bridge=virbr0  \
   --graphics none --console pty, \
   target_type=serial --location 'URL' \
  --extra-args 'console=ttyS0,115200n8 serial'

 $ ./scripts/gen_vmconfigs.py trident
```

We recommend installing Ubuntu18.04 in the VM. To do so, use http://archive.ubuntu.com/ubuntu/dists/bionic/main/installer-amd64/ for URL in the installation command above. In addition, do the following to finish setting up the platform:

- In `./scripts/configs.sh`, edit guest user name and IP address (GUESTUSER and GUESTIP) and `libvirt`'s ID of the VM image (VMIMAGE) as per your installation.
- Setup password-less authentication between guest and host. This can be done, for example, by adding the RSA key of the host user in `$HOME/.ssh/authorized_keys` of guest and vice-versa (`ssh-copy-id $USER@$ADDRESS`).
- Install sshfs in the guest (`sudo apt install sshfs`). Set up guest Linux to auto-mount the artifact directory in the same path as the host using a network file system such as SSHFS (e.g., user's home directory). See GitHub README.md for an example.
- Grant sudo privileges to users in both host and guest; they should be able to execute sudo without entering password.
- Add user to `libvirtd` group in the host (`sudo adduser 'id -un' libvirtd`).

A.4.3 Prepare datasets. Prepare datasets as follows:

```
$ ./scripts/prep_all_datasets.sh
```

The datasets (approximately 4.4GB for Canneal, 21GB for SVM and 190GB for fragmentation related experiments) will be redirected to `./datasets/`.

A.5 Experimental Workflow

Run experiments exclusively i.e., no compute or memory-intensive application should be running concurrently. Use kernel command-line parameter `default_hugepagesz=2MB` by default. Running configuration 1GBHUGE (in Figure 1 and Figure 2) requires booting kernel with default large page size of 1GB. To run this configuration: (1) uncomment it in `./scripts/run_figure_1.sh` and `./scripts/run_figure_2.sh`, (2) append `default_hugepagesz=1G` parameter to the host kernel's command-line (`/etc/default/grub` file), and (3) reboot the host. All scripts are to be executed on the host. Run individual experiments as:

```
$ ./scripts/run_figure_1.sh
$ ./scripts/run_figure_2.sh
$ ./scripts/run_figure_9.sh
$ ./scripts/run_figure_10.sh
$ ./scripts/run_figure_11.sh
$ ./scripts/run_figure_12.sh
$ ./scripts/run_figure_13.sh
```

A.5.1 Estimated completion time. Approximate completion time for Figure 1, Figure 2, Figure 10, Figure 11 and Figure 13 is about 24 hours each. Figure 9 and Figure 12 require about 12 hours each.

A.5.2 Generate report. The raw output logs of the experiments are redirected to `./evaluation/`, in separate sub-directories for each benchmark. Process logs into CSV files (redirected to `./report/`) by executing:

```
$ ./scripts/compile_report.py
```

REFERENCES

[1] 2000. Clearing pages in idle loop. https://www.mail-archive.com/freebsd-hackers@freebsd.org/msg13993.html.

[2] 2006. GUPS: HPCC RandomAccess benchmark. https://github.com/alexandermerritt/gups.

[3] 2017. Hugepages. https://wiki.debian.org/Hugepages.

[4] 2021. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[5] 2021. Graph500. https://graph500.org/.

[6] 2021. Intel® Microarchitecture Code Named Skylake Events. https://download.01.org/perfmon/index/skylake.html.

[7] 2021. LIBSVM Data: Classification (Binary Class). https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html.

[8] 2021. Linux perf. https://en.wikipedia.org/wiki/Perf_(Linux).

[9] 2021. Page frame allocation via Buddy in Linux. https://wiki.osdev.org/Page_Frame_Allocation.

[10] 2021. Wikichip: Intel Sandy Bridge microarchitecture. https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client).

[11] 2021. Wikichip: Intel Sandy Bridge microarchitecture. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake.

[12] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 283–300. https://doi.org/10.1145/3373376.3378468

[13] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 457–468. https://doi.org/10.1145/3079856.3080209

[14] AnandTech. 2019. The Ice Lake Benchmark Preview: Inside Intel's 10nm. https://www.anandtech.com/show/14664/testing-intel-ice-lake-10nm/2.

[15] David Bailey, E. Barszcz, J. Barton, D. Browning, Robert Carter, Leonardo Dagum, Rod Fatoohi, Paul Frederickson, T. Lasinski, Robert Schreiber, Horst Simon, Venkat Venkatakrishnan, and Sisira Weeratunga. 1991. The NAS Parallel Benchmarks;Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[16] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[17] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 307–318. https://doi.org/10.1145/2000064.2000101

[18] Arkaprava Basu. 2013. *Revisiting virtual memory*. Ph.D. Dissertation. University of Wisconsin-Madison.

[19] 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

[20] 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

[21] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/1346281.1346286

[22] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

[23] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[24] Josiah L. Carlson. 2013. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA.

[25] Jonathan Corbet. 2017. Five-level page tables. https://lwn.net/Articles/717293/.

[26] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 435–448. https://doi.org/10.1145/3037697.3037704

[27] Ian Cutress. 2019. Intel's Enterprise Extravaganza 2019: Launching Cascade Lake, Optane DCPMM, Agilex FPGAs, 100G Ethernet, and Xeon D-1600. https://www.anandtech.com/show/14155/intels-enterprise-extravaganza-2019-roundup.

[28] Yu Du, Miao Zhou, Bruce R. Childers, Daniel Mossé, and Rami G. Melhem. 2015. Supporting superpages in non-contiguous physical memory. *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)* (2015), 223–234.

[29] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5.

[30] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, USA, 178–189. https://doi.org/10.1109/MICRO.2014.37

[31] 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

[32] Mel Gorman and Patrick Healy. 2012. Performance Characteristics of Explicit Superpage Support. In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*. Springer-Verlag, Berlin, Heidelberg, 293–310. https://doi.org/10.1007/978-3-642-24322-6_24

[33] Brendan Gregg. 2017. How Netflix tunes EC2 Instances for Performance. http://www.brendangregg.com/Slides/AWSreInvent2017_performance_tuning_EC2.pdf.

[34] Faruk Guvenilir and Yale N Patt. 2020. Tailored Page Sizes. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 900–912.

[35] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 257–273. https://www.usenix.org/conference/osdi21/presentation/hunter

[36] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 705–721.

[37] Joshua Magee and Apan Qasem. 2009. A Case for Compiler-driven Superpage Allocation. In *Proceedings of the 47th Annual Southeast Regional Conference (ACM-SE 47)*. ACM, New York, NY, USA, Article 82, 4 pages. https://doi.org/10.1145/1566445.1566553

[38] Kristie Mann. 2019. Five Use Cases of Intel Optane DC Persistent Memory at Work in the Data Center. https://itpeernetwork.intel.com/intel-optane-use-cases/.

[39] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[40] 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

[41] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. 2021. Fast Local Page-Tables for Virtualized NUMA Servers with vMitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 194–210. https://doi.org/10.1145/3445814.3446709

[42] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 14.

[43] 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

[44] M. Papadopoulou, X. Tong, A. Seznec, and A. 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

[45] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Huh Jaehyuk. 2020. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA '20)*. ACM, New York, NY, USA.

[46] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2020. Page Tables: Keeping them Flat and Hot (Cached). arXiv:1508.03619 https://arxiv.org/abs/2012.05079

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

[48] 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

[49] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2830772.2830773

[50] Kiran Puttaswamy and Gabriel Loh. 2006. Thermal Analysis of a 3D Die-stacked High-performance Microprocessor. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI '06)*. ACM, New York, NY, USA, 19–24. https://doi.org/10.1145/1127908.1127915

[51] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/3079856.3080210

[52] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1093–1108. https://doi.org/10.1145/3373376.3378493

[53] Avinash Sodani. 2011. MICRO keynote: Race to Exascale. https://www.microarch.org/micro44/files/Micro%20Keynote%20Final%20-%20Avinash%20Sodani.pdf.

[54] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *;login:* 41, 1 (2016). https://www.usenix.org/publications/login/spring2016/tarasov

[55] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. [n. d.]. 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.

[56] 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

[57] vMitosis. 2021. vMitosis ASPLOS'21 Artifact Evaluation. https://github.com/mitosis-project/vmitosis-asplos21-artifact/blob/main/scripts/helpers/fragment.py.

[58] Zi Yan. 2019. Generating physically contiguous memory after page allocation. https://patchwork.kernel.org/cover/10815945/.

[59] Zi Yan. 2020. 1GB PUD THP support on x86_64. https://lwn.net/Articles/832881/.

[60] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 698–710. https://doi.org/10.1145/3307650.3322223

[61] Weixi Zhu, Alan L. Cox, and Scott Rixner. 2020. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 829–842. https://www.usenix.org/conference/atc20/presentation/zhu-weixi