Architectural Support for Optimizing Huge Page Selection Within the OS

Aninda Manocha
amanocha@princeton.edu
Princeton University
Princeton, New Jersey, USA

Zi Yan
ziy@nvidia.com
NVIDIA
Westford, Massachusetts, USA

Esin Tureci
esin.tureci@princeton.edu
Princeton University
Princeton, New Jersey, USA

Juan L. Aragón
jlaragon@um.es
University of Murcia
Murcia, Spain

David Nellans
dnellans@nvidia.com
NVIDIA
Austin, Texas, USA

Margaret Martonosi
mrm@princeton.edu
Princeton University
Princeton, New Jersey, USA

ABSTRACT

Irregular, memory-intensive applications often incur high translation lookaside buffer (TLB) miss rates that result in significant address translation overheads. Employing huge pages is an effective way to reduce these overheads, however in real systems the number of available huge pages can be limited when system memory is nearly full and/or fragmented. Thus, huge pages must be used selectively to back application memory. This work demonstrates that choosing memory regions that incur the most TLB misses for huge page promotion best reduces address translation overheads. We call these regions *High reUse TLB-sensitive* data (*HUBs*). Unlike prior work which relies on expensive per-page software counters to identify promotion regions, we propose new architectural support to identify these regions dynamically at application runtime.

We propose a promotion candidate cache (PCC) that identifies HUB candidates based on hardware page table walks after a last-level TLB miss. This small, fixed-size structure tracks huge page-aligned regions (consisting of N base pages), ranks them based on observed page table walk frequency, and only keeps the most frequently accessed ones. Evaluated on applications of various memory intensity, our approach successfully identifies application pages incurring the highest address translation overheads. Our approach demonstrates that with the help of a PCC, the OS only needs to promote 4% of the application footprint to achieve more than 75% of the peak achievable performance, yielding 1.19-1.33× speedups over 4KB base pages alone. In real systems where memory is typically fragmented, the PCC outperforms Linux's page promotion policy by 14% (when 50% of total memory is fragmented) and 16% (when 90% of total memory is fragmented) respectively.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Software and its engineering** → **Virtual memory**; **Memory management**; **Operating systems**.

MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3614296

KEYWORDS

hardware-software co-design, cache architectures, memory management, virtual memory, operating systems, graph processing

ACM Reference Format:
Aninda Manocha, Zi Yan, Esin Tureci, Juan L. Aragón, David Nellans, and Margaret Martonosi. 2023. Architectural Support for Optimizing Huge Page Selection Within the OS. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3614296

1 INTRODUCTION

Increasingly large application memory footprints are becoming problematic for modern virtual memory hierarchies as they incur high address translation overheads that can limit overall workload throughput. Specifically, irregular memory access patterns in applications cause high translation lookaside buffer (TLB) miss rates. While TLB miss rates can be reduced by modifying the TLB size, changing organization, or replacement policies [12, 42], these solutions do not scale with large increases in memory size. Modern TLB organizations still cover only a fraction of the total system memory in production systems [48]. A cost-effective alternative is to utilize *huge pages* [8], which map large, e.g. 2MB on x86, contiguous regions of virtual memory to contiguous regions of physical memory, increasing individual TLB entry coverage.

Fig. 1 presents a performance comparison when using 4KB pages for all data (black) vs. 2MB pages for all data (blue) vs. greedy huge page allocation in a system with 50% fragmented memory (orange) while running a variety of applications selected to show varying levels of sensitivity to huge pages. In all cases, huge pages provide measurable performance improvements with speedups as high as 2× (geomean 1.3×), and TLB miss rate reduction by 2.9× on average. However, this ideal performance is rarely achieved in practice. When Linux greedily allocates Transparent Huge Pages (THPs) while memory is 50% fragmented, its performance rarely exceeds the performance of using base pages alone. Many database applications also often suffer performance degradation when using huge pages [7, 10, 13, 20, 40, 43, 49, 50, 53, 57, 62, 65]. Several factors contribute to this. First, huge pages require additional time to prepare and map into an application's memory space compared to base pages [21, 60, 63]. Second, aggressive use of huge pages can bloat an application's memory footprint, wasting precious free

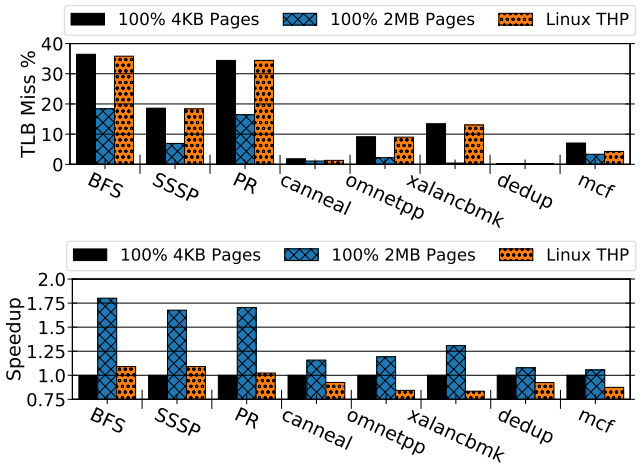


Figure 1: TLB miss rate and application performance comparison under various page sizes and Linux's default huge page promotion policy. 2MB pages have significant performance potential, however, Linux's greedy THP allocation fails to achieve this performance in the presence of just 50% memory fragmentation.

memory [29, 44]. Last, memory pressure can limit the availability of large contiguous regions of physical memory to form huge pages [45, 66].

Modern operating systems such as Linux handle these issues inefficiently. This is because Linux lacks knowledge of individual application behavior and cannot classify application data pages by TLB sensitivities, despite its continuous effort to lower huge page performance pitfalls (e.g. removing huge page creation at page fault time, demoting underutilized huge pages, and adding anti-fragmentation mechanisms). When system memory pressure is high, limited huge pages are not allocated optimally to prioritize TLB-sensitive data, hampering performance gains and wasting CPU effort to create huge pages with low utility [37].

Prior works have aimed to improve on Linux's aggressive allocation policy by adding heuristics about page regions to huge page promotion decisions [29, 39, 44, 68]. The OS monitors page usage (including recording, aggregating, and resetting data about all pages present in the system), incurring significant overheads which degrade both kernel and application performance. Additionally, because tracking rich data information requires expensive per-page storage overheads and using too much memory for such data is always a concern, most proposals only use 1-2 bits, which do not provide the desirable high resolution on page activity.

To minimize page tracking overheads, enrich page activity information, and improve OS huge page management, we propose a hardware-OS co-design: the hardware tracks page utilization and ranks huge page promotion candidates for high precision and low overheads, while the OS continues to make huge page promotion decisions for maximum flexibility. Our work discovers a strong correlation between page reuse distance and the performance improvement brought by promoting these pages to huge pages. By utilizing our hardware-tracked reuse information, Linux is able to identify huge page promotion candidates faster, more precisely, and with lower overheads than when using software-only approaches. Our contributions can be summarized as follows:

- We develop a new characterization of memory access patterns using page reuse distance at different page sizes, which identifies huge page candidates whose promotions will improve TLB miss rates the most.
- We design a novel hardware *promotion candidate cache (PCC)* to track huge page (2MB)-aligned regions that collectively incur the most page table walks from constituent base page accesses. This tracking information serves as a proxy of our access pattern characterization.
- We decouple OS page promotion decisions from page data tracking performed by the PCC to relieve the OS from scanning overheads and enable quicker promotion of candidates, especially when the system is under memory pressure and huge page resources are limited.
- We implement this approach in the Linux v5.15 kernel based on offline PCC simulation and perform *real-system* performance evaluations. We demonstrate single-thread speedups of 1.19-1.33× over base pages alone, achieving 69-77% of idealized huge page performance while requiring only 1-4% of the application memory footprint be backed by huge pages. When memory is 50% and 90% fragmented, the PCC outperforms Linux by 1.14× and 1.16×, respectively. The PCC concept is also multithread- and multiprocess-friendly, achieving 1.07-1.18× and 1.07-1.15× speedups respectively.

2 BACKGROUND AND MOTIVATION

Hardware support for multiple page sizes in systems dates back nearly three decades [63]. However, OS techniques to transparently manage the use of huge pages is much more recent [35]. Today, most modern operating systems, including Linux and FreeBSD, are equipped with some variation of one or more levels of huge page support. In this work, without loss of generality, we focus on Linux's huge page management policy and its implementation and drawbacks.

2.1 Linux's Existing Huge Page Policies

In Linux, a user can explicitly manage huge pages via the *hugetlbfs* library [33] or the OS can automatically manage them via Transparent HugePage (THP) support [35]. The latter allows users to seamlessly execute applications without the source code modifications, manual management of huge page reservations, or memory allocation API interceptions that *hugetlbfs* requires. Linux performs THP promotion of 4KB pages to 2MB huge pages in two ways: *synchronously* upon the first access to a huge page region and *asynchronously* via the kernel daemon, *khugepaged*, that runs in the background while applications are executing on the system.

Synchronous Promotion: In Linux, the first access to a memory region results in a page fault to back the virtual address range with a physical page. Linux leverages this opportunity to allocate the region of memory as a 2MB huge page rather than a 4KB base page. It checks the huge page eligibility of the faulting virtual address, then allocates a 2MB huge page in physical memory to back the virtual address if possible. This aggressive huge page allocation

policy can have a beneficial prefetching effect by preemptively faulting in additional memory beyond just 4KB. However, this can also lead to memory bloat, thus wasting free memory, if the additional page data allocated does not end up being accessed [29, 44]. To alleviate this problem, Linux provides settings that can be configured via `madvise`, so that a user has direct control (using the `MADV_HUGEPAGE` flag) over which memory allocations are backed by huge pages. Unfortunately, this increases code complexity and similar to Linux's aggressive policy, performance improvement depends on the availability of free huge pages.

Furthermore, backing a newly touched page with a huge page can incur a longer page fault latency because 512× data (2MB vs. 4KB) needs to be zeroed and huge page allocation might take a much longer time if no huge page is readily available. Because Linux needs to reclaim in-use pages and compact memory, this process can dramatically lengthen page fault time [40, 53]. In reality, huge page allocations that typically take less than 1 second can spike to 90 seconds [21].

Asynchronous Promotion: To avoid lengthy page fault time in synchronous huge page allocation, Linux provides a daemon called *khugepaged* to promote huge pages system-wide and asynchronously from application execution. This daemon periodically scans virtual memory regions for matching physically contiguous 2MB regions that it can promote. System administrators can tune *khugepaged* to be more or less aggressive or they can configure via `madvise` which memory regions can be promoted. However, determining the settings that might benefit an application the most or deciding whether to use *khugepaged* globally across applications in a machine is difficult at best.

Huge Page Demotion: When a huge page is no longer mapped as a whole, Linux demotes the page by first splitting it into 512 4KB base pages when memory pressure is high and then updating page table entries to point to the new split pages so that unmapped 4KB pages can be reclaimed. This avoids huge page capacity underutilization. In addition, if a huge page is swapped out, it is split into base pages, which are faulted back in individually instead of as a huge page.

2.2 Prior Work on Huge Page Promotion

To address Linux's huge page management inefficiencies, prior research has aimed to strike a balance between performance, OS overheads, and memory bloat [28, 29, 39, 44, 46, 68]. To remove huge page creation from the application execution's critical path, these works have proposed software techniques such as page pre-zeroing, huge page pre-allocation, user-directed promotion, and asynchronous huge page creation [58]. These works utilize software-based page monitoring, such as huge page region utilization, to guide asynchronous promotion and demotion decisions.

For example, HawkEye [44] proposes an *access coverage* metric that counts the number of base pages accessed within a huge page region, within a predefined interval (1 second of tracking). It then categorizes all huge page regions into buckets that are stored in linked lists (for dynamic addition/removal of a region) based on their base page access coverage count, ranging from 0 to 512. Regions with coverage counts of 0-49 are placed in bucket 0, 50-99 in bucket 1, and so on. HawkEye prioritizes regions in bucket 9 (access coverage count is 450-512) for promotion and works backwards through buckets as it promotes data.

While effective for some applications, decisions to trigger promotion and demotion processes using heuristics rely on specific tunable thresholds that may not be suitable for a wide variety of applications. Furthermore, these solutions require the OS to record, aggregate, and reset data about all pages present in the system, which can incur significant tracking and scanning overheads that degrade both kernel and application performance. Most proposals, including HawkEye, only use 1-2 bits within the struct page structure maintained by the Linux kernel for physical page metadata (64B per page) due to storage limits [27, 29, 39, 68]. These bits do not provide the desirable high resolution on page activity. When running HawkEye in a system with 256GB of RAM for example, the OS must mark, monitor, and scan across access coverage data from 64 million pages. Even if this data itself comprises a small percentage of system memory, it spans across 4GB total of struct page metadata. Thus, scanning it continually significantly impacts application performance. It is due to these tracking and scanning overheads that Hawkeye's promotion process must sleep for 30 seconds between 1-second measurement intervals (scanning a limited number of pages) to alleviate OS-level overheads.

3 PROMOTION CANDIDATE CACHE

This work builds upon state of the art by recognizing the value of hardware-assisted page promotion. This section describes new hardware support that alleviates the storage and kernel overheads of tracking and selecting huge page candidates, outperforming prior software-only solutions.

3.1 Characterizing Page-Level Reuse

To understand how virtual memory performance is impacted by different page sizes, we first study page access patterns and the corresponding TLB hit rates. We study 8 applications (further described in Sec. 4) that vary in memory demands to capture a wide spectrum of datacenter workloads. We analyze *page reuse distance*, i.e. the number of accesses to other pages between two accesses to a given page (4KB or 2MB), which reveals that for many applications there are three distinct categories of memory reuse occurring within the virtual memory hierarchy.

(a) TLB-Friendly Accesses: These accesses exhibit high spatial and/or temporal locality, e.g. sequential accesses. The corresponding pages can have low or average reuse distances when the page size is 4KB, but all have high TLB hit rates. Because the base page size already achieves good TLB performance, there is little additional benefit of using huge pages.

(2) High-Reuse TLB-Sensitive Accesses: These accesses exhibit low spatial and temporal locality with 4KB base pages but exhibit good locality at the 2MB granularity. This behavior often manifests in sparse data access patterns that have reuse. For example, the frequency of pointer indirect accesses to vertex neighbor data in graph applications correlates with vertex degree [14, 36]. These accesses are TLB-unfriendly at base page level (page reuse distance is too high for pages to be retained in the existing TLB hierarchy) yet they have good/low reuse at the huge page level (their huge page translations will be well cached). Thus, these regions are

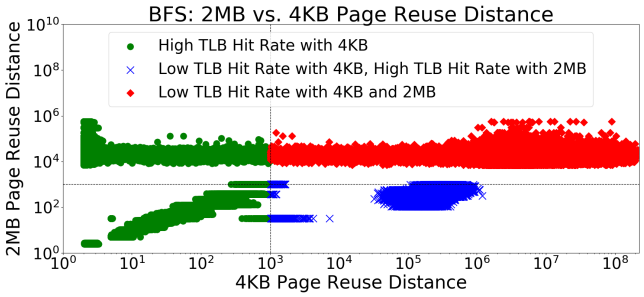


Figure 2: Characterization of data accessed based on page reuse distance with 2MB vs. 4KB pages, running BFS on a Kronecker network. *HUBs* (blue) have low TLB hit rates with 4KB pages, but high TLB hit rates with 2MB.

the best promotion candidates because they are likely to achieve higher TLB hit rates when backed by huge pages. We refer to these regions as *HUBs (High-reUse TLB-sensitive data)* and these pages are the ones we attempt to identify with our proposed solution.

(3) Low-Reuse TLB-Sensitive Accesses: These accesses exhibit such little locality that even with huge pages, accesses are still too infrequent to benefit from the additional coverage provided. Page reuse distance remains high regardless of the page size. Consequently, the OS should not prioritize these pages for promotion.

Fig. 2 visualizes these three types of access patterns by measuring the average reuse distance of each page accessed when running Breadth First Search on a power-law network using 4KB base pages and 2MB pages respectively. For each 4KB page, we plot the reuse distance of the 2MB region it falls in (y-axis) against its 4KB reuse distance (x-axis). In this example, we consider a "low" reuse distance as less than 1024, a common number of entries in a CPU's second-level TLB [23]. Pages with reuse distances lower than this are likely retained in the TLB hierarchy because of their temporal locality. Thus, page-size agnostic TLB-friendly accesses (green) appear on the left of the scatterplot (low 4KB page reuse distances), *HUBs* (blue) appear in the lower right, and low-reuse accesses (red) appear in the upper right.

Based on these observations, it is clear that an ideal promotion candidate tracking structure should ignore pages for which there is already good TLB locality, and differentiate between the remaining pages to identify *HUBs*, data for which a huge page upgrade will improve the TLB hit rate.

3.2 Capturing Reuse Distance in Hardware

In practice, calculating page reuse distance requires recording all page accesses. Software cannot obtain the information without incurring significant overheads, e.g. making every memory access trigger a software fault. Thus, prior works approximate this by sampling pages using hardware access bits, but even sampling has a nontrivial amount of overhead. We propose using hardware to directly capture page reuse information, addressing the major shortcomings of prior work. This hardware has two primary features.

First, while prior approaches track page information for all pages in the system, our design avoids tracking pages with TLB-friendly accesses to significantly reduce the amount of page data to track and scan. Our approach filters out these TLB-friendly accesses (green in Fig. 2) because it only checks page accesses missed in the whole TLB hierarchy, i.e. page accesses that result in page table walks. We place our hardware after the last-level TLBs to achieve this.

Second, many prior works rely on huge page region utilization (the number of base pages accessed within a huge page region) to guide promotion decisions. This incurs significant storage overhead. Instead, we use per-huge page region page table walk frequency as this metric can distinguish *HUBs* (blue) from pages with low reuse in 4KB that also have low reuse in 2MB (red).

We propose a *promotion candidate cache (PCC)* to track pages at the 2MB/1GB granularity and identify huge page regions frequently associated with page table walks as these regions are most likely to benefit from promotion. The PCC is designed to be a small, fully-associative structure whose sole responsibility is to assist the OS with page size promotion decisions. It operates in parallel with page table walks and the retrieval of page table mappings to fill the TLBs, not on an application execution's critical path.

The left side of Fig. 3 details the overall virtual memory hardware datapath including the PCC. When a memory request to a 4KB page from the CPU misses in the TLB hierarchy, it triggers a hardware walk through the hierarchical page tables **(1)**. Intel CPUs set an access bit in each page table hierarchy entry from L4 to L1 when any page from the sub-tree pointed by the entry is accessed [25]. Our approach leverages these access bits to identify page table walks incurred by cold TLB misses. To avoid polluting the PCC with cold misses (and potentially filling the PCC with data accessed in a TLB-friendly manner), our design checks the access bit(s) when a page table walk occurs and inserts data into the PCC only if the bit(s) are already set. If the hardware supports 1GB pages, the page table walker first reaches the Page Global Directory (PGD) **(2)**, checks the access bit of the 1GB region accessed **(3)**, and updates the 1GB PCC (Sec. 3.2.3 describes this in detail) **(4)**. Once the page table walk reaches the Page Middle Directory (PMD) **(5)**, the access bit is checked **(6)**. If the bit is already set, then the PCC is accessed with the tag of the 2MB virtual address prefix **(7)**, as this 2MB region can be permitted into the PCC.

3.2.1 PCC Structure, Function, and Overheads. The right side of Fig. 3 details PCC operation. When the PCC is accessed with a 2MB virtual address prefix, its frequency, stored as an N-bit saturating counter is updated. If the access results in a hit, then the frequency simply increments. Otherwise, a victim is evicted if the PCC is full, and the new 2MB virtual address prefix (a huge page promotion candidate) is inserted with frequency 0. The PCC implements a decay function for frequency; whenever the frequency counter saturates for any given tag, all counters are halved to maintain their relative order. Note that in the PCC design, the frequency is the data field for each tag.

PCC Replacement Policy: Ideally all *HUBs* are backed by huge pages, however, due to the costly nature of huge page creation, the OS can only create so many huge pages at a given time. Thus, promotion candidates ultimately need to be ranked in a priority list so that huge page regions that will eliminate the most TLB misses are promoted first. The most intuitive replacement policy to is to use a Least Frequently Used (LFU) ranking of the PCC data. Performing this sorting within the PCC can save the OS from a

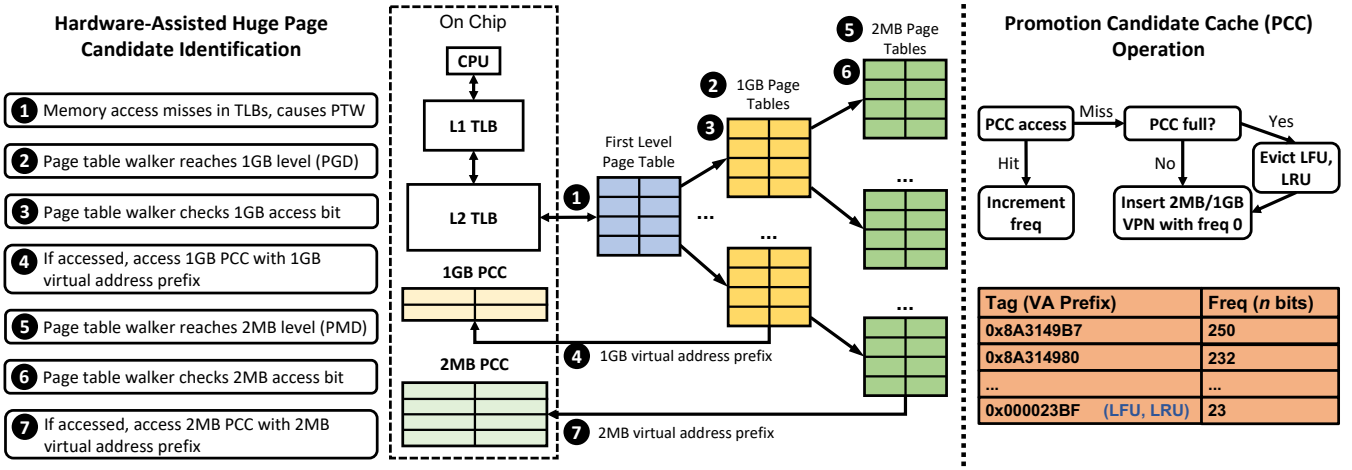


Figure 3: Overview of PCC operation. Left: Page table walks trigger accesses to the PCC(s) when the huge page region for a memory request has been accessed at least once. Each PCC's operation runs in parallel to page table walks. Right: Each PCC tracks the frequency of page table walks per huge page (2MB or 1GB) region.

significant effort. However, a perfect implementation of this policy, which must scan and reorder all PCC data, complicates the PCC design and implementation compared to a simpler policy, such as Least Recently Used (LRU), which may not as accurately sort promotion candidates based on their frequency.

In our experiments we did not find replacement policy changes to have significant impact on performance because the PCC size is sufficiently large to capture the *HUBs* that have a large impact on application runtime. In fact, there are often many entries in the PCC with a frequency of 0 and thus LFU (with LRU as a tiebreaker) and LRU select the same victims for eviction. Thus, we opt for the simpler policy. However, in scenarios where the footprint of the *HUBs* far exceeds the PCC capacity and causes thrashing, the PCC still tremendously helps huge page promotion by providing local optimal candidates instead of global ones until all candidates are promoted. An optimized replacement policy as well support for huge page demotion (see Sec. 3.3.3) can mitigate this. We leave this exploration for future work.

PCC Overheads: Each entry in the 2MB PCC stores a 40-bit tag (2MB virtual address prefix) and an 8-bit frequency, totaling 6B. With a 128-entry 2MB PCC, this results in 768B of storage for the tags and data combined. For an 8-entry 1GB PCC (see details in Sec. 3.2.3), each entry stores a 31-bit tag (1GB virtual address prefix) and 8-bit frequency, totaling 40B. Note that the total amount of PCC storage, i.e. 808B, would only afford 50 additional TLB entries, assuming 16B per TLB entry (8B for the virtual address + 8B for the physical address). With a 1024-entry L2 TLB, this only increases TLB coverage by 5%, whereas spending the same amount of hardware to identify pages worth promoting can enable 64K (128×512) 4KB pages to be identified as promotion candidates into 2MB pages, a much better value proposition.

We use Cacti 7.0 [41] to obtain upper bound estimates for area, power, and timing. We model each PCC as a 768B fully-associative cache and measure the area to be 0.0019mm^2 (less than 1% of the L1 data cache area), the dynamic energy per PCC access as 0.0105nJ (13% of the L1 data cache access energy), and the access latency of the PCC as 0.5ns (37% of the L1 data cache access latency), which equates to about 2 cycles on our evaluation machine. We also note that the PCC is only accessed following a page table walk. For a given core, walk latencies are serialized and can span hundreds of cycles, which separate consecutive accesses to the PCC. Thus, PCC operation latencies are negligible and the PCC can afford full associativity to avoid all conflict misses.

3.2.2 Per Core vs Shared PCCs. In a multicore setting, multiple threads may perform memory accesses within the same (single process) or different (multiprocess) virtual address spaces. Because the number of huge pages depends on the amount and state of physical memory, the OS must be able to manage these different address spaces in order to create huge page mappings that maximize application performance. To assist with this process, there are two choices for the placement of the PCC design.

A single *global* PCC shared across all cores in the system can track and organize the globally most frequently accessed *HUBs*. This effectively places the responsibility of aggregating data from multiple cores on the hardware. While this prevents the OS from needing to perform any aggregation or sorting, this design decision introduces additional PCC hardware complexity. First, multiple page table walks can result in simultaneous updates to a single structure. Second, this requires the PCC to be much larger in order to support the memory footprint of the *HUBs* for all applications running on the system. Third, a single centralized structure must be accessed across growing chip footprints. Finally, a user or the OS may want to bias certain processes, i.e. prioritize data for specific applications for page size promotion. If the hardware manages the global ranking of all huge page candidate data, then it needs to be aware of such bias, which also involves additional complexity.

Conversely, using a *local* PCC per core significantly reduces hardware complexity. Each core has its own TLB hierarchy so the corresponding PCC tracks the huge page regions that incur page table walks the most. This design choice allows PCCs to remain

fairly small, since a PCC only needs to capture the *HUBs* for a single core and large application memory footprints are typically divided amongst multiple cores. The OS then becomes responsible for aggregating and sorting the promotion candidate information from each core before performing promotions.

3.2.3 1GB Page Support. As modern datasets continue to grow in size, 1GB pages are becoming increasingly important and gaining support in modern hardware [54, 56]. There are two ways the PCC design can be extended to support 1GB pages (or another page size) with the addition of a smaller PCC that tracks the frequency of page table walks at the 1GB granularity. These page table walks can arise from 4KB or 2MB page accesses, where the latter indicates that the 2MB page size is not sufficiently preventing last-level TLB misses.

To determine whether to promote 4KB pages that comprise a 1GB region with a high collective page table walk frequency, the OS can compare the 2MB and 1GB PCC frequencies. That is, if the frequency in a 2MB PCC entry is at least 512× less than the frequency in the corresponding entry in the 1GB PCC, then 1GB is a more suitable page size, as it would likely eliminate more page table walks. If a 1GB page promotion candidate comprises both 4KB and 2MB regions, then the entire region is collectively promoted into a 1GB page.

Alternatively, this smaller 1GB PCC can track only 2MB huge pages (data that have already been promoted) that frequently incur page table walks at the 1GB granularity. Such data are not only poorly served by the 2MB page size (frequent page table walks), but also exhibit high temporal locality at the 1GB granularity and thus can further benefit from another promotion. This is a direct extension of determining when to promote 4KB pages into 2MB and requires the hardware to have knowledge of which 2MB regions tracked in the PCC correspond to data that have already been promoted.

3.3 Operating System Integration

Operating systems manage memory resources, including huge page promotion and demotion, but they lack application page utilization information to make optimal page size decisions. Prior work has shown that software-only page monitoring introduces significant runtime overheads. Our proposed PCC design is able to minimize these overheads. The PCC assists the OS with promotion decisions, removing the burden of tracking and scanning from the OS while still providing flexibility about when and what high-utility pages to promote. In our OS integration with the PCC, we aim to optimize huge page utility, that is, promote as few huge pages while eliminating as many TLB misses as possible.

OS Behavior: Fig. 4 presents an overview of how the OS leverages the PCC to select huge page promotion candidates. The contents of the PCC can be periodically written into a designated region of physical memory accessible by a device driver or read directly on demand. This data is written in the same order as the priority list maintained by the PCC. When the PCC is full or after a preset time period, the CPU dumps PCC content to the designated memory region and initiates a software interrupt. The OS then reads the data from the memory region **(A)** and selects N huge page candidates from the data to promote **(B)**. Unlike prior works where the OS

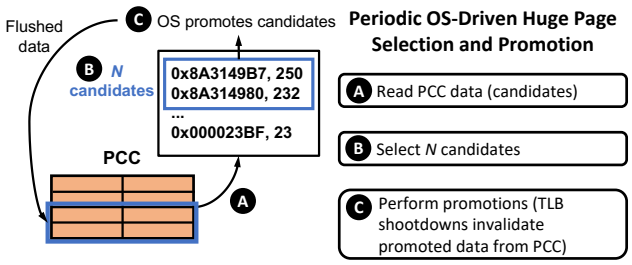


Figure 4: Overview of how the OS periodically reads PCC data to select N candidates for huge page promotion. The N candidates are invalidated from the PCC by TLB shootdowns during promotions.

may need to scan several GB of page data, the PCC aggregates page data together in a single, smaller region of memory that is simpler and quicker for the OS to scan. OS developers can design their own policies to inform how the OS selects the N candidates. Last, the OS performs the promotions, which cause TLB shootdowns **(C)**. When *any* TLB shootdown occurs, whether due to promotion or another process such as page migration, the corresponding huge page region, if it exists in the PCC, is invalidated from the PCC.

Normally, a TLB shootdown invalidates page table entries and the OS does not promote regions with invalid entries. Promotion can occur before invalidations (serialized by the page table lock), wasting effort, but this is rare. Either (1) a TLB shootdown invalidates PCC entries or (2) the OS promotion thread triggers a TLB shootdown that invalidates PCC entries (promoted candidates). Since there is only one OS promotion thread (or multiple with a PCC lock for exclusive candidate access), no stale promotion candidate can exist.

3.3.1 Promotion Frequency. The OS has direct control over when and which pages to promote. It can operate as frequently as desired, as long as the time to perform promotions does not exceed the promotion interval. The PCC determines the maximum number of pages the OS can promote at a given time. Given that most modern machines have around 32 cores and each PCC can store up to $C = 128$ candidates (256MB of coverage per core), the OS is not limited in practice. Our policy by default has the OS promote C pages per interval (shared across all PCCs), where C is the individual PCC size. However, the user can write to a kernel parameter, `regions_to_promote`, to control the number of promotions occuring per interval. In our empirical evaluations, we find this promotion interval to be sufficient (detailed in Sec. 5). However, this interval can be tuned, either by a user or automatically by the OS, based on application memory consumption or system memory pressure.

3.3.2 Multiple Threads and Process Fairness. We choose to place the responsibility of promotion candidate data aggregation on the OS. With many PCCs, the OS has a variety of options with respect to how it prioritizes candidates for promotion. The most fair policy is a round-robin policy. However, this policy assumes that all threads have similar memory usage and benefit equally from huge pages, which is often not the case. A potentially higher performing alternative is to prioritize the candidates with the highest frequency

values, as these candidates have incurred the most page table walks. This is more expensive for the OS (comparing up to C frequencies at once, where the system has C cores), but may pay off in terms of performance. We evaluate both policies in Sec. 5. To bias the promotion process, the user can write a value (0 = round-robin, 1 = highest PCC frequency) to a kernel parameter, `promotion_policy`.

Allowing Process Bias: A user or the OS often wants to prioritize the performance of a specific application or process. In this case, the OS should also prioritize huge pages for the process by promoting data for it until there are no more promotion candidates left (as tracked in the PCC(s)), before trying to promote pages for other processes. The user can write to a kernel parameter, `promotion_bias_process`, the process ID(s) to prioritize. The OS, which knows about process information and associated PCC data, can then adjust its promotion priority list accordingly, e.g. highest PCC frequency or round robin amongst the biased processes if there is more than one. In the future this mechanism should be standardized with OS control and resource groups.

3.3.3 Page Demotion. When memory pressure is high, huge page resources become very limited for new huge page creation. In such scenarios, these resources should be utilized for the data that benefits the most from promotions. If existing huge pages no longer benefit the data they back, e.g. the data becomes cold, it is necessary to demote them to free up contiguous memory regions and promote candidates that can offer more performance improvement when backed with huge pages.

The PCC can help identify demotion candidates by searching for huge pages (data already promoted) that incur more page table walks than others. If a 2MB page incurs page table walks with high frequency, then the page size is not suitable and either the page should be promoted to 1GB or demoted to 4KB if promotion is not possible and promoting another 2MB region can yield better performance. The OS has knowledge of which data have been promoted and when scanning the data from the PCC, it locates poorly used 2MB pages. When memory pressure is high and 1GB page promotion is infeasible, it can compare this page with other promotion candidates to decide whether to demote the page or not.

Furthermore, if a 2MB huge page is rarely accessed, the PCC might not see this page and could lose demotion opportunities. In this case, the OS needs to assist in determining demotion candidates with huge page access information. For example, the existing Linux multi-generation LRU algorithm can identify cold pages [34]. We leave OS-assisted demotion to future work.

Application Phases: For some applications, data may be frequently accessed in one phase and infrequently in another. Pages that were promoted before a program phase change may later no longer need to exist as huge pages, in which case it can be particularly valuable to demote cold data. In our workloads this phasing is not significant. We evaluate demotion in the context of memory pressure, but leave more intelligent demotion candidate selection that benefits multi-phase applications for future work.

4 EXPERIMENTAL METHODOLOGY

This work evaluates a variety of applications that exhibit varying TLB sensitivity, including irregular graph algorithms Breadth First Search, Single Source Shortest Paths, and PageRank based on implementations from the GAP benchmark suite [4], as well as memory-bound workloads from the PARSEC [5] and SPEC2017 benchmark suites [6, 30, 61]. Because graph application behavior is typically dataset-dependent, we evaluate each graph workload on synthetic power-law (Kronecker 25), real-world social (Twitter), and real-world web (Sd1 Arc) networks. Data preprocessing, particularly degree-based grouping (DBG) [14], has been shown to improve baseline on-chip cache and TLB performance by coalescing frequently accessed data to occupy the same cachelines or TLB entries [15, 37]. We report results for each of our 3 graph workloads as the geomean performance of both sorted and unsorted networks, totalling 6 datasets for each graph workload. Table 1 summarizes the applications, datasets, and workload memory footprints we evaluate.

Table 1: Evaluation Applications and Inputs

Application	Input	Nodes	Edges	Footprint
Breadth First Search (BFS)	Kronecker 25	34M	1.05B	10GB
	Twitter	53M	1.94B	17GB
	Sd1 Web	95M	1.96B	19GB
Single Source Shortest Paths (SSSP)	Kronecker 25	34M	1.05B	19GB
	Twitter	53M	1.94B	34GB
	Sd1 Web	95M	1.96B	38GB
PageRank (PR)	Kronecker 25	34M	1.05B	10GB
	Twitter	53M	1.94B	17GB
	Sd1 Web	95M	1.96B	19GB
Suite	**Application**	**Input**	**Size**	**Footprint**
PARSEC	**canneal**	native	98MB	860MB
	dedup		672MB	838MB
SPEC2017	**mcf**	native	3.2MB	5GB
	omnetpp		18MB	252MB
	xalancbmk		56MB	427MB

Table 2: Evaluation System Parameters

Processor	Intel Xeon CPU E5-2667 v3 @ 3.20 GHz 2 sockets, 8 cores/socket, 2 threads/core
OS	CentOS 7 - Linux v5.15
L1 D-TLB	4KB: 64 entries, 4-way set associativity 2MB: 32 entries, 4-way set associativity 1GB: 4 entries, 4-way set associativity
L1 I-TLB	4KB: 64 entries, 8-way set associativity 2MB: 8 entries, full set associativity
L2 TLB	4KB&2MB: 1024 entries, 8-way set associativity
Memory	64GB DDR4 (per socket), 2 NUMA nodes
2MB PCC	Per Core: 128 entries, full set associativity 40-bit tags, 8-bit frequency counters Up to 128 promotions every 30 seconds

Simulation and Real-System Infrastructures: To accurately capture PCC hardware behavior, our evaluation is a two-step process consisting of first, offline hardware simulation to capture the 4KB virtual address regions that incur the most page table walks and model promotion of these regions, and second, online real-system performance evaluation that includes OS behavior and overheads.

In the first step, we model and simulate the behavior of the CPU's data TLB hierarchy (L1 and L2 TLBs), so that the PCC receives all page table walk information. We use Intel's Pin tool [26] to extract memory accesses during application execution and input these accesses into the simulated TLBs. The PCC tracks 2MB-aligned regions (where the tags are 2MB virtual address prefixes) that miss in the last-level TLB, as described in Sec. 3.2. Within the simulation, a periodic "promotion" process takes place every 30 seconds (calibrated based on number of memory accesses per second observed in each application), extracting promotion candidates from the PCC and removing them as if they have been promoted. We use the same time interval as HawkEye [44] for a fair performance comparison. During TLB+PCC simulation, the PCC candidate addresses as well as the time when they are promoted are recorded in a trace file. Tab. 2 summarizes details about the PCC parameters used in simulation.

In the second step, the OS reads from the promotion candidate address trace as if real hardware provided the data. The candidate addresses identified by the PCC are used by the OS promotion logic at the correct time during workload execution. A low-overhead background thread performs userspace promotion system calls from a Linux v5.15 kernel that we modified to support *synchronous* huge page promotion after pages fault in. The system call we introduced takes as input a range of data (formatted as a base address and offset) to synchronously promote, i.e. try to promote immediately. This is invoked in a similar fashion to `madvise()`, but is distinct in how the kernel operates; the kernel asynchronously scans ranges of data provided by `madvise()` and may not attempt to promote the data immediately after the system call is invoked. This new support is similar to a recent kernel patchset from Google allowing userspace promotions for experimental purposes [58]. The simulation does not need to consider physical memory, but promoted virtual addresses must match in simulation and real-system evaluation. We set the `randomize_va_space` kernel parameter to 0 to guarantee deterministic virtual address assignment [2] and ensure real-system evaluations operate on the same regions captured in simulation.

This two-step process emulates a system setup with a hardware PCC identifying profitable promotion candidates and the OS periodically consuming the candidate information to perform promotions. This demonstrates the real-system effect of these promotions, including all page promotion overheads.

We perform our experiments on a 2-socket machine using Intel Xeon processors and 128GB of total RAM, while running Linux kernel v5.15. Table 2 provides detailed information about the system and simulation parameters. Memory access latency can differ when accessing local vs. remote NUMA nodes and Linux's default allocation policy can result in variable application runtimes for the same huge page configuration. To eliminate this effect and minimize non-determinism in our results, we use the `numactl` library and invoke the `–membind` and `–physcpubind` flags [31, 32] to bind the process(es) to CPU(s) on NUMA node 1 and all memory allocations within that NUMA node. We also store graph data in *tmpfs* on the remote NUMA node instead of loading from disk to eliminate the NUMA impact on the page cache. We measure runtime performance of an application as the workload's elapsed wall clock time after one full execution, and report the results of the geomean of 3 executions. Collectively, all of these measures eliminate experimental noise. Observed differences in performance from existing works may arise from evaluation setup, e.g. executing the application only once instead of repeatedly, where performance overheads may be amortized.

Performance Utility Curve: To measure how well our approach optimizes huge page selection across different huge page availability in a system, we evaluate performance while limiting the number of huge pages used to back N% of the total application memory footprint, where N ranges from 0 (baseline), 1, 2, 4, ..., 64, ~100%, totaling 9 data points per utility curve. The ~100% configuration represents promotions occurring until 100% of all huge page candidates tracked in the PCC are promoted (the most aggressive case for our approach). However, the PCC might not have visibility of 100% of the total application footprint since TLB-friendly accesses may never experience page table walks. The utility curve demonstrates the effect of memory pressure or fragmentation limiting huge page resources and shows the effective utility of promoting additional huge pages in each application.

5 RESULTS AND EVALUATION

This section presents our real-system performance evaluation of promoting pages recommended by the PCC in single-threaded, multi-threaded, and multiprocess settings. In all experiments, the baseline configuration uses 4KB base pages only, while the ideal performance is achieved by allocating all application memory with huge pages (no memory pressure).

5.1 Single Thread Performance

We first present evaluation results where the PCC is dedicated to a single thread application running on one core.

Runtime Performance: Fig. 5 presents a comparison of the performance utility curves with our PCC approach (blue) and HawkEye (red), the state of the art in software-based huge page management, for 8 irregular applications (described in Sec. 4). The top figure shows workload speedup as the number of huge pages used ranges from 0, 1, 2, ..., 64, ~100% of the application memory footprint (as explained in Sec. 4). The black line above each curve represents the peak achievable performance with huge pages (all application data backed by huge pages), while the dotted green and orange lines represent realistic performance evaluations of Linux's policy when memory is 50% and 90% fragmented, respectively.

For graph applications with larger memory footprints, our approach achieves 1.19-1.33× speedups over 4KB base pages (69-77% of the ideal performance) when backing just 1-4% of the application footprint with huge pages. The remaining applications are less TLB-sensitive. *dedup* and *mcf*, which are optimized for low cache miss rates, exhibit negligible sensitivity. However, our approach does not hurt TLB-insensitive applications and due to its negligible overheads, it can be enabled in a general-purpose system.

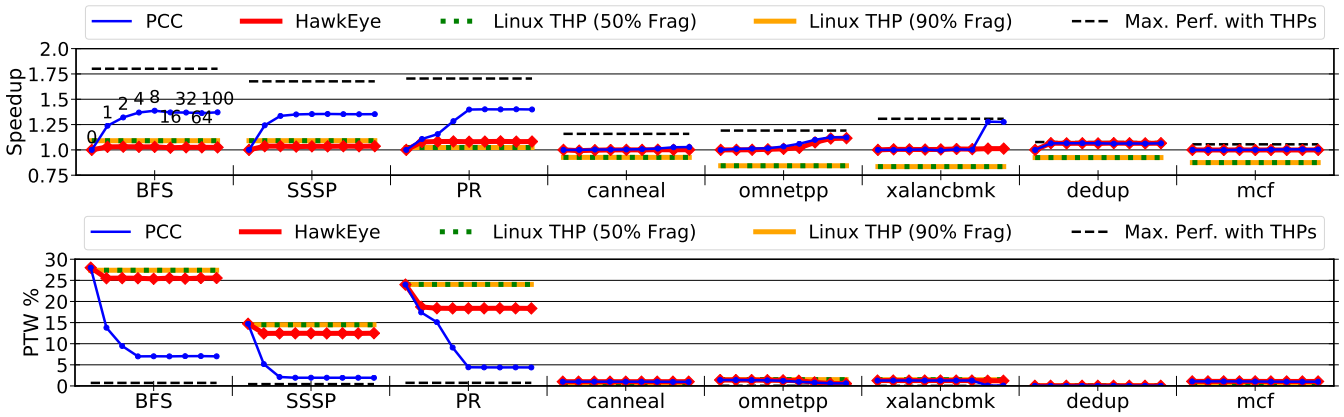


Figure 5: Single-thread runtime performance (top) and PTW rate (bottom) utility curve comparison between using a PCC (blue) and HawkEye (red). Each line plots performance when huge pages back 0, 1, 2, 4, ..., 64, ~100% of the application footprint. For all applications, the PCC approach outperforms HawkEye due to its hardware assistance.

For all applications our approach outperforms HawkEye because HawkEye faces two key issues. First, while its access coverage metric should favor *HUBs* (multiple sparse accesses can span across a single huge page region within a page scanning period), this fully-software approach cannot identify *HUBs* quickly enough because it is limited by the number of pages it can scan during each period. More specifically, within a scanning interval HawkEye scans the same number of pages as *khugepaged*, i.e. 4096 (covering 8 huge page regions), and therefore cannot perform as many promotions as the PCC (up to 128) during a given interval.

Second, HawkEye's metric loses out on performance opportunity because it only records whether or not a base page is used (no access or TLB miss frequency) and only considers the spatial distribution of accesses across a huge page region. Regardless of how distributed page accesses or TLB misses are within a huge page region, the collective page table walk frequency of the region is a good indicator of how many TLB misses can be saved from promotion. For example, a huge page region may only be 25% utilized, but a significant number of TLB misses leading to thousands of page table walk (PTW) cycles may occur to these 25% of base pages. Promoting this huge page region can avoid the TLB misses and PTW cycles. HawkEye ignores this opportunity since the utilization of the huge page region is below its promotion threshold. The PCC approach can identify such regions using PTW frequency information. Furthermore, promotion frequency or the OS policy can be tuned based on workload demands to determine the number of PCC entries to promote based on collective frequencies. We observe the greatest discrepancy in HawkEye vs. PCC candidate selection for PageRank, where the PCC identifies *HUBs* faster and better.

Page Table Walk Rates: The bottom of Fig. 5 presents the percentage of accesses that result in PTWs, i.e. miss in both TLB levels, for the PCC approach and HawkEye. In general, PTW rate reduction strongly correlates with application speedup. Graph applications exhibit a 18% reduction in PTW rate on average. The plateauing of PTW rates, which often occurs when 4% of the application footprint has been promoted, indicates where performance improvements plateau.

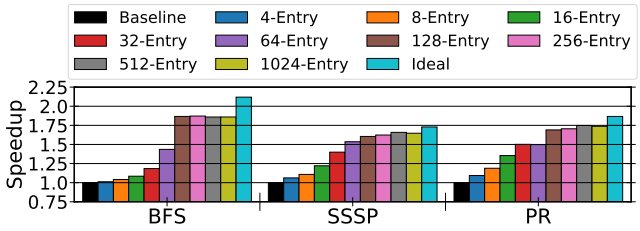


Figure 6: Sensitivity analysis of the impact of promotion candidate cache size, ranging from 4 to 1024 entries, on application runtime performance for graph applications. Bars from left to right represent the performance of the baseline, 4-entry PCC, 8-entry PCC, ..., 1024-entry PCC, and the ideal scenario where all data is backed by huge pages.

Sensitivity Analysis: PCC Size Fig. 6 studies the effect of PCC size on graph application speedups. Our analysis focuses on these workloads running on the Kronecker network due to their huge page sensitivity; the PCC size has little to no impact on irregular applications that have low TLB miss rates with 4KB pages. In Fig. 6 the PCC size ranges from 4 to 1024 entries (in powers of 2) while the promotion footprint limit is set to 32% of the application footprint. For all applications, runtime speedup steadily increases as the PCC size increases from 4 to 32 entries. Each application has slightly different diminishing return points with respect to performance improvements, but for all applications, a 128-entry PCC achieves the bulk of possible performance improvement. TLB miss rates also plateau at this size.

5.1.1 Realistic Memory Conditions. Memory fragmentation is a serious and common cause of memory pressure in datacenters [67]. To demonstrate how the utility curves translate to a realistic scenario, e.g. where memory compaction often takes place to form contiguous physical memory regions for promotions, we evaluate the PCC approach when system memory is 50% and 90% fragmented. We fragment memory by allocating one non-movable page in every

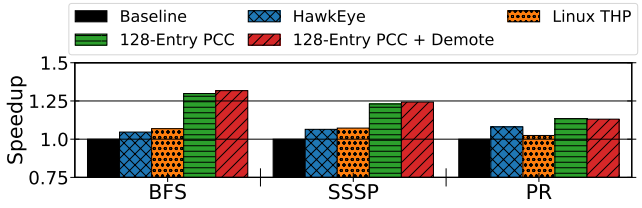


Figure 7: Speedup comparisons between 4KB pages, HawkEye, and the PCC approach with and without demotion when system memory is 90% fragmented.

2MB-aligned region. Fig. 7 presents speedup comparisons between 4KB pages, HawkEye, Linux's greedy THP policy, and the PCC approach with and without PCC-driven demotion for the graph applications when memory is 90% fragmented. The PCC approach achieves the best performance, demonstrating 1.22×, 1.15×, and 1.16× speedups over the baseline, HawkEye, and Linux, respectively. The PCC identifies the few promotion candidates that eliminate the most TLB misses and therefore even when there are limited huge page resources, this approach is able to find high-utility candidates. Conversely, Linux's greedy policy causes it to allocate huge pages to data that are not TLB-sensitive and thus it runs out of huge page resources before it can promote *HUBs*. HawkEye again is limited by the number of 4KB pages it can scan during its measurement period. We observe the same performance trends when memory is just 50% fragmented, indicating that THP performance gains are fairly sensitive to fragmentation.

Demotion: We evaluate page demotion using the PCC as described in Sec. 3.3.3 to identify huge pages that are no longer well served by the 2MB size. We observe negligible performance improvements with demotion in place because the PCC identifies high-utility promotion candidates early in application execution that experience high data reuse throughout execution. This is exemplified in Fig. 5, where there are negligible performance gains when backing all of the application memory with huge pages as opposed to only 4%. Thus, even though demotion makes room for new huge page creations when system memory is fragmented, the new huge pages do not have as high utility in these applications.

5.2 Single Process, Multithread Performance

We evaluate our approach on multi-threaded applications, where all threads belong to the same process and each thread runs on a different core with individual per-core PCCs. In this case, the OS gathers promotion information from multiple PCCs and makes huge page promotion decisions for a single process because all threads share the same address space. We show how OS policy can affect performance.

Fig. 8 presents utility curve comparisons of our multi-threaded graph applications running with 2-8 threads each. We focus on the graph analytic workloads because these are particularly sensitive to huge page usage. We compare two different OS policies when selecting huge page candidates from multiple PCCs: **Highest PCC Frequency** selects promotion candidates with the globally highest PCC frequencies (blue). **Round Robin** selects candidates so that huge pages are distributed equally across the threads (red). Unless

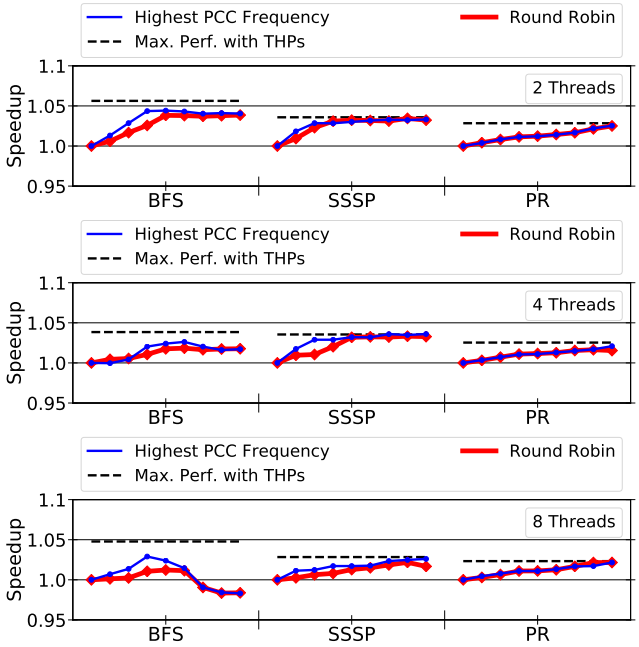
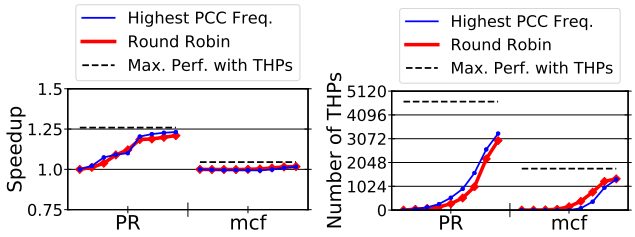


Figure 8: Runtime performance comparisons of parallelized graph applications running with multiple (2-8) threads (1 per core). The OS either uses a highest PCC frequency first (blue) or round-robin policy (red) to select candidates from the many PCCs.

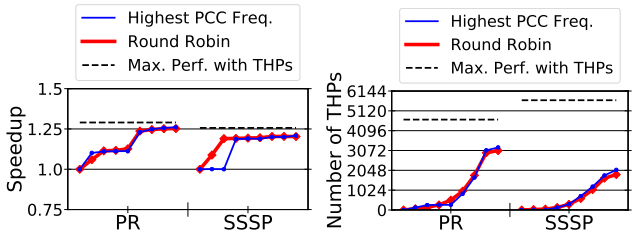
a thread runs out of candidates in its PCC, huge pages will always be distributed evenly amongst threads.

Overall, selecting candidates based on PCC frequency is slightly more performant than opting for fairness via a round robin policy. This is due to load imbalance; some threads incur more PTWs and consequently reap more benefit from huge pages. When backing 1-4% of the application memory footprint with huge pages, our approach achieves speedups of 1.07-1.18× (85-94% of ideal) with 2 threads, 1.04-1.13× (85-93% of ideal) with 4 threads, and 1.04-1.12× (86-92% of ideal) with 8 threads. These speedups are lower than single-thread gains for two reasons. (1) Due to parallelization, the probability of memory compaction touching a specific page at the same time as one of the threads, and total TLB shootdown rate due to promotions, increases. (2) Atomic operations used in multithread implementations cause serialization and with more threads this becomes a larger part of the execution time that cannot be sped up by reducing TLB misses.

Data undergoing promotion can conflict with application accesses and cause execution to stall for a very short period of time. However, the probability of such conflicts increases when more threads are executing and thus the OS must be careful not to promote too much data. Such overheads are often negligible, but can become more apparent when the application runtime is very short, as is the case when BFS runs with 8 threads and more than 16% of the application footprint is promoted.



(a) 1 TLB-sensitive and 1 TLB-insensitive application



(b) 2 TLB-sensitive applications

Figure 9: Runtime performance (left) and huge page usage (right) when running two applications simultaneously. The OS prioritizes highest PCC frequency first (blue) or operates in a round-robin fashion (red) to select candidates.

5.3 Multiprocess Performance

When multiple processes run in the system, the OS is responsible for distributing huge pages across them fairly. To understand how the PCC approach affects multiprocess performance, we performed two case studies where we ran two single-threaded applications on two cores running in parallel. Each application accesses its own PCC, but huge pages are a system-wide resource that are shared amongst the processes.

The first study focuses on the TLB-sensitive graph application PageRank (PR) running alongside *mcf*. Fig. 9(a) presents curves for performance (utility) and huge page usage, i.e. number of promotions when using the same two OS policies studied in the multi-thread setting. With the round-robin policy, both applications receive the same number of huge pages until *mcf*'s execution finishes first and the OS only provisions huge pages to PageRank. The PCC frequency-based policy offers slightly better performance because it heavily biases PageRank, which is much more TLB-sensitive. However, more *HUB* promotions do not occur until later in execution when 16% of the memory footprint is promoted. Both policies identify candidates where their promotions will significantly improve performance. PageRank achieves more than 95% of the peak performance possible when huge pages are limited to 8% of the total memory footprint of the 2 applications, resulting in a speedup of 1.1×. Meanwhile *mcf* performance remains unaffected.

The second study focuses on running two TLB-sensitive graph applications, PageRank and SSSP, alongside each other. Fig. 9(b) presents the performance and huge page usage curves. When backing 1-2% of the application footprint with huge pages, the PCC frequency-based OS policy favors PageRank. However, once huge pages cover 4% of the footprint, the OS provisions huge pages to SSSP, whose PCC entries have accumulated more frequency, and both policies perform similarly. Although SSSP has a larger memory footprint, it has less TLB-sensitive data (thus fewer promotion candidates in its PCC) than PageRank. When all PCC data is promoted, more huge pages end up provisioned to PageRank. Both PageRank and SSSP achieve 95% of their best possible performances when huge pages are limited to only 16% of their combined memory footprint, yielding 1.23× and 1.19× speedups over using 4KB base pages alone.

In summary, when a TLB-sensitive application runs alongside an application that has little sensitivity to huge pages, the PCC frequency-based policy is slightly more performant since one application benefits much more from huge pages. The two policies perform similarly when the TLB-sensitive application has many more PCC accesses as the other application eventually runs out of candidates. When more than 1 TLB-sensitive application runs simultaneously, the round robin OS policy for huge page selection performs best. This is because with the PCC frequency-based policy it is possible for one application to starve the other application of huge page resources due to higher PTW frequency counts that can be phase-dependent.

5.4 Discussion

5.4.1 Design Alternatives. The PCC could be implemented in software, which would increase portability and enable flexibility for architectures with support for more page sizes, such as RISC-V [59]. However, this implementation would incur high overheads to update and manage PCC data as well as the priority list for evictions. Every TLB miss would trap into the OS for PCC updates, which would interrupt the CPU pipeline and would be prohibitively expensive, much like how software page table walkers have been replaced by hardware walkers on most platforms. A hardware PCC does not involve such complexities and works well with the hardware page table walker.

Another design alternative would be to augment Page Walk Caches (PWCs), which cache partial PTWs to shorten their latencies and do so quite effectively. They can minimize PTW overhead to 1.1-1.4 references/walk, while any leaf PTE requires a single access [25]. An infinite-sized PWC will approach a single reference/walk so there is not significant room for improvement[1]. While the PWC and PCC share similarities at first glance, residing in the same location in the translation hierarchy, they track very different information. PWCs do not reduce TLB miss rates, so applications still suffer from multiple TLB lookups and miss cycles on the processor's critical path in addition to the residual walk overhead that caching cannot eliminate. For example, a PWC entry caching a L4-L3 page structure can point to a L2 page table page containing entries for both 2MB huge pages and an L1 PTE. Because the PWC has no knowledge about the page size accessed by the CPU, it cannot attribute access frequency for any individual page, or well-defined groups of pages, like the PCC does. Thus the PWC cannot (easily) be repurposed to perform the PCC's function.

The necessary changes to PWCs (and potentially the page table walker) include adding page sizes of data accessed by the CPU to

[1]True measurement of this requires prohibitively long simulation times given the memory footprints and execution times of the applications we evaluate.

PWC entries and requests to the PWC (key PCC information to identify promotion vs. demotion candidates) as well as per-huge page region PTW frequency counters (to better guide promotion decisions). Augmenting the PWC would save more storage space compared to a standalone PCC, since the PWC is also indexed by virtual address prefixes like the PCC. Evaluating such a design is beyond the scope of this work, as not all system effects, from the TLB miss to the page table walk, can be accurately modeled via simulation. We leave it as future work to develop such an evaluation methodology and further optimize the PCC implementation.

Finally, a victim cache for the L2 TLB could capture *HUBs* as huge page regions evicted due to TLB capacity constraints. However, a cache too small cannot sufficiently track and rank promotion candidates and would get polluted with other data that is too sparsely accessed to benefit from promotion. As the cache grows larger, it approaches the PCC size.

5.4.2 Huge Page Allocation. In our experiments, all page data is initially allocated as 4KB base pages and the PCC dynamically selects promotion candidates. However, compiler or programmer analysis can identify *HUBs* before workload execution and this knowledge can guide the allocation of huge pages in lieu of dynamic promotion. Aggressive allocations of huge pages can prevent TLB misses if promotions occur at low frequency, but we note that the PCC can identify *HUBs* within a few seconds and when memory pressure is high, aggressive policies should be avoided.

5.4.3 Virtualization. In a virtualized environment, the guest OS and hypervisor must coordinate for efficient huge page management. If only the guest OS promotes a huge page, the hypervisor might still back the guest huge page with base pages, resulting in no performance improvement, since the TLB does not use 2MB entries for the translation [52]. Thus, both the guest OS and hypervisor need to promote guest and host pages, respectively and together. The PCC can recommend guest virtual address regions for the guest OS to promote to improve performance. The guest OS can then initiate the promotion and a hypercall (from guest OS to hypervisor) can invoke the hypervisor to promote the host pages as well [54]. Such a design can be implemented by using an additional bit to tag PCC entries as corresponding to guest vs. host pages, or utilizing additional, smaller PCCs for guest page data.

6 ADDITIONAL RELATED WORK

Memory Management: Efficient memory management is an area with much potential for improving huge page performance. To avoid imbalanced data access issues for applications using huge pages in NUMA systems, prior work proposes NUMA-aware page placement algorithms that sample page accesses and track the nodes they access to inform future placement decisions [9, 11, 17]. A group of works also aim to optimize the placement and migration of pages and huge pages in tiered and hybrid memory systems by monitoring page access patterns and usage [1, 9, 17, 38, 55, 64]. Userspace memory allocation libraries are also often made aware of huge pages for better huge page utilization [22].

TLB Improvements: Improved hardware support for virtual memory translation has largely focused on isolated changes to the TLB that can be performed without requiring OS involvement. Existing hardware only supports discrete page sizes (4KB, 2MB, and 1GB on x86) [23, 24, 33, 35], limiting the number of possible huge page sizes. Prior works propose various new TLB concepts, such as intermediate page sizes via entry coalescing [47, 51], any power of two page sizes [19], a range of virtual addresses [16, 28], or enabling holes in contiguous physical memory regions that are part of huge page mappings [46]. For applications with a small number of large contiguous memory ranges, direct segment proposes to use memory segments in the OS to provision the ranges and [base, size] format in the TLB to cache translations of these segments and lower translation overheads even when huge pages are not present [3].

Software techniques to group base pages strive to achieve the same performance goal as huge pages [3, 16, 19, 54, 66]. Midgard attacks the TLB coverage issue from a different angle. By introducing a new address space of virtual memory areas (an OS concept of/data structure for representing contiguous virtual addresses with the same permissions) as a new layer of translation, it significantly reduces the number of translations to cache since virtual memory areas are usually much larger than normal page sizes and much smaller in numbers [18]. Overall, TLB optimization is orthogonal to page size management; these approaches complement OS-driven techniques for huge page management.

7 CONCLUSION

Deploying huge pages is an effective way of reducing TLB misses incurred in the virtual memory translation system. However, naïve application of huge pages can squander the availability of this limited set of pages. This work proposes augmenting the OS with hardware-based page promotion candidate identification, a novel approach that frees the OS from the storage overheads and runtime complexity needed in prior solutions. Our experimental results show that promotion of pages recommended by the PCC yields 1.19-1.33× application speedups (69-77% of the peak achievable performance) on a real machine, while backing just 1-4% of the application memory footprint with huge pages. Our approach demonstrates that explicit architectural support for what has historically been OS-only functionality can make OS-driven huge page management both practical and performant.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This research was supported in part by the DARPA Software-Defined Hardware Program under agreement No. FA8650-18-2-7862 and NSF Award No. 1763838. Aninda Manocha was also supported by the NSF Graduate Research Fellowship. The views and conclusions contained herein should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or NSF. Prof. Aragón was also supported by Grants TED2021-130233B-C33 and PID2022-136315OB-I00 funded by MCIN/AEI/10.13039/501100011033, the EU NextGenerationEU/PRTR, and "ERDF A way of making Europe", EU.

REFERENCES

[1] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Xi'an, China). ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/3037697.3037706

[2] Linux Audit. 2018. Configuring ASLR with randomize_va_space. https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/ [Retrieved September 2022].

[3] 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 International Symposium on Computer Architecture (ISCA)* (Tel-Aviv, Israel). ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[4] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. http://gap.cs.berkeley.edu/benchmark.html [Retrieved October 2022].

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

[6] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU 2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE).* Association for Computing Machinery, New York, NY, USA, 41–42.

[7] Kevin Burke. 2017. Unusually large (but constant) memory usage with transparent huge pages enabled. https://github.com/nodejs/node/issues/11077 [Retrieved October 2022].

[8] Jonathan Corbet. 2011. Transparent huge pages in 2.6.38. https://lwn.net/Articles/423584/ [Retrieved October 2022].

[9] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. http://lwn.net/Articles/488709/ [Retrieved October 2022].

[10] Couchbase. 2022. Disabling Transparent Huge Pages (THP). https://docs.couchbase.com/server/current/install/thp-disable.html [Retrieved October 2022].

[11] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, Texas, USA). ACM, New York, NY, USA, 381–394.

[12] Yannick Deville and Jean Gobert. 1992. A Class of Replacement Policies for Medium and High-Associativity Structures. *SIGARCH Computer Architecture News* 20, 1 (March 1992), 55–64. https://doi.org/10.1145/130823.130827

[13] DigitalOcean. 2015. Transparent Huge Pages and Alternative Memory Allocators: A Cautionary Tale. https://www.digitalocean.com/blog/transparent-huge-pages-and-alternative-memory-allocators [Retrieved October 2022].

[14] Priyank Faldu, Jeff Diamond, and Boris Grot. 2019. A Closer Look at Lightweight Graph Reordering. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC).* IEEE, USA, 1–13.

[15] Priyank Faldu, Jeff Diamond, and Boris Grot. 2020. Domain-Specialized Cache Management for Graph Analytics. In *Proceedings of the 26th International Symposium on High-Performance Computer Architecture (HPCA).* IEEE, USA, 234–248.

[16] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Ünsal. 2016. Range Translations for Fast Virtual Memory. *IEEE Micro* 36, 3 (May-June 2016), 118–126. https://doi.org/10.1109/MM.2016.10

[17] 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)* (Philadelphia, PA). USENIX Association, USA, 231–242. http://dl.acm.org/citation.cfm?id=2643634.2643659

[18] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. 2021. Rebooting Virtual Memory with Midgard. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA).* IEEE, USA, 512–525.

[19] Faruk Guvenilir and Yale N. Patt. 2020. Tailored Page Sizes. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)* (Virtual Event). IEEE, USA, 900–912. https://doi.org/10.1109/ISCA45697.2020.00078

[20] Hadoop. 2012. Recommendation to disable huge pages for Hadoop. https://developer.amd.com/wordpress/media/2012/10/HadoopTuningGuide-Version5.pdf [Retrieved October 2021].

[21] Red Hat. 2019. Memory allocation latency in RHEL. https://access.redhat.com/solutions/2607721 [Retrieved July 2023].

[22] Andrew Hamilton Hunter, Chris Kennelly, Darryl Gove, Parthasarathy Ranganathan, Paul Jack Turner, and Tipp James Moseley. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *Proceedings of the 15th Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, USA, 257–273.

[23] Intel. 2013. Intel Haswell. http://www.7-cpu.com/cpu/Haswell.html [Retrieved October 2022].

[24] Intel. 2015. Intel Skylake. http://www.7-cpu.com/cpu/Skylake.html [Retrieved October 2022].

[25] Intel. 2016. Intel® 64 and IA-32 Architectures Developer's Manual. Volume 3A: System Programming Guide, Part 1.

[26] Intel. 2022. Pin - A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html [Retrieved October 2022].

[27] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA).* IEEE, USA, 78–89.

[28] 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).* ACM, New York, NY, USA, 66–78.

[29] 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 Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, USA, 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[30] Ankur Limaye and Tosiron Adegbija. 2018. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, USA, 149–158.

[31] Linux Kernel Documentation. 2004. numactl - Control NUMA policy for processes or shared memory. https://linux.die.net/man/8/numactl [Retrieved February 2022].

[32] Linux Kernel Documentation. 2007. numa(3) — Linux manual page. https://man7.org/linux/man-pages/man3/numa.3.html [Retrieved February 2022].

[33] Linux Kernel Documentation. 2021. hugetlb page. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt [Retrieved February 2021].

[34] Linux Kernel Documentation. 2021. Multi-Gen LRU. https://docs.kernel.org/admin-guide/mm/multigen_lru.html [Retrieved March 2022].

[35] Linux Kernel Documentation. 2021. Transparent Hugepage Support. https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html [Retrieved February 2021].

[36] Aninda Manocha, Juan Luis Aragón, and Margaret Martonosi. 2022. Graphfire: Synergizing Fetch, Insertion, and Replacement Policies for Graph Analytics. *IEEE Trans. Comput.* 72, 1 (March 2022), 291–304.

[37] Aninda Manocha, Zi Yan, Esin Tureci, Juan Luis Aragón, David Nellans, and Margaret Martonosi. 2022. The Implications of Page Size Management on Graph Analytics. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC).* IEEE, USA, 199–214. https://doi.org/10.1109/IISWC55918.2022.00026

[38] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, J ohn Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous Memory Architectures: A HW/SW Approach For Mixing Die-stacked And Off-package Memories. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA).* IEEE, USA, 126–136.

[39] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. 2019. MEGA: Overcoming Traditional Problems with OS Huge Page Management. In *Proceedings of the 12th International Conference on Systems and Storage (SYSTOR)* (Haifa, Israel). ACM, New York, NY, USA, 121–131. https://doi.org/10.1145/3319647.3325839

[40] MongoDB. 2013. Disable Transparent Huge Pages (THP). https://www.mongodb.com/docs/manual/tutorial/transparent-huge-pages/ [Retrieved October 2022].

[41] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Labs* 27 (2009), 1–24.

[42] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. 1993. Design Tradeoffs for Software-Managed TLBs. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)* (San Diego, California, USA). ACM, New York, NY, USA, 27–38.

[43] NuoDB. 2022. Recommendation to disable huge pages for NuoDB. http://www.nuodb.com/techblog/linux-transparent-huge-pagesjemalloc-and-nuodb [Retrieved October 2022].

[44] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, RI, USA). ACM, New York, NY, USA, 347–360. https://doi.org/10.1145/3297858.3304064

[45] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* ACM, New York, NY, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[46] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. 2020. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA).* IEEE, USA, 913–925.

[47] 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 International Symposium on Computer Architecture (ISCA)* (Toronto, ON, Canada). ACM, New York, NY, USA, 444–456.

[48] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every Walk's a Hit: Making Page Walks Single-Access Cache Hits. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Lausanne, Switzerland). ACM, New York, NY, USA, 128–141.

[49] Percona. 2014. Why TokuDB Hates Transparent HugePages. https://www.percona.com/blog/2014/07/23/why-tokudb-hates-transparent-hugepages/ [Retrieved October 2022].

[50] Perforce. 2015. Tales from the Field: Taming Transparent Huge Pages on Linux. https://www.perforce.com/blog/151016/tales-field-tamingtransparent-huge-pages-linux [Retrieved October 2015].

[51] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th International Symposium on Microarchitecture (MICRO)* (Vancouver, B.C., CANADA). IEEE, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[52] 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)*. ACM, New York, NY, USA, 1–12.

[53] PingCAP. 2020. Transparent Huge Pages: Why We Disable It for Databases. https://www.pingcap.com/blog/transparent-huge-pages-why-we-disable-it-for-databases/ [Retrieved October 2022].

[54] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. 2021. Trident: Harnessing Architectural Resources for All Page Sizes in X86 Processors. In *Proceedings of the 54th International Symposium on Microarchitecture (MICRO)* (Virtual Event, Greece). ACM, New York, NY, USA, 1106–1120. https://doi.org/10.1145/3466752.3480062

[55] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the 25th International Conference on Supercomputing (ICS)* (Tucson, Arizona, USA). ACM, New York, NY, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[56] RedHat. 2023. 8.3.3.3. Enabling 1 GB huge pages for guests at boot or runtime. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-memory-huge_pages-1gb-runtime [Retrieved April 2023].

[57] Redis. 2022. Recommendation to disable huge pages for Redis. http://redis.io/topics/latency [Retrieved October 2022].

[58] David Rientjes. 2021. [RFC] Hugepage collapse in process context. https://lore.kernel.org/linux-mm/d098c392-273a-36a4-1a29-59731cdf5d3d@google.com/ [Retrieved February 2022].

[59] RISC-V. 2020. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 1.12-draft.* Technical Report. EECS Department, University of California, Berkeley.

[60] Theodore H. Romer, Wayne H. Ohlrich, Anna R. Karlin, and Brian N. Bershad. 1995. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, USA, 176–187.

[61] Standard Performance Evaluation Corporation (SPEC). 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/ [Retrieved October 2022].

[62] Splunk. 2022. Transparent huge memory pages and Splunk performance. https://docs.splunk.com/Documentation/Splunk/7.3.2/ReleaseNotes/SplunkandTHP [Retrieved October 2022].

[63] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in Supporting Two Page Sizes. *SIGARCH Computer Architecture News* 20, 2 (April 1992), 415–424. https://doi.org/10.1145/146628.140406

[64] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2008. Hardware Monitors for Dynamic Page Migration. *J. Parallel and Distrib. Comput.* 68, 9 (September 2008), 1186–1200. https://doi.org/10.1016/j.jpdc.2008.05.006

[65] VoltDB. 2022. Recommendation to disable huge pages for VoltDB. https://docs.voltdb.com/AdminGuide/adminmemmgt.php [Retrieved October 2022].

[66] 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)* (Phoenix, Arizona). ACM, New York, NY, USA, 698–710. https://doi.org/10.1145/3307650.3322223

[67] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. 2023. Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, USA, 1–15.

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