HawkEye: Efficient Fine-grained OS Support for Huge Pages

Ashish Panwar
Indian Institute of Science
ashishpanwar@iisc.ac.in

Sorav Bansal
Indian Institute of Technology Delhi
sbansal@iitd.ac.in

K. Gopinath
Indian Institute of Science
gopi@iisc.ac.in

Abstract

Effective huge page management in operating systems is necessary for mitigation of address translation overheads. However, this continues to remain a difficult area in OS design. Recent work on Ingens [55] uncovered some interesting pitfalls in current huge page management strategies. Using both page access patterns discovered by the OS kernel and fine-grained data from hardware performance counters, we expose problematic aspects of current huge page management strategies. In our system, called HawkEye/Linux, we demonstrate alternate ways to address issues related to performance, page fault latency and memory bloat; the primary ideas behind HawkEye management algorithms are async page pre-zeroing, de-duplication of zero-filled pages, fine-grained page access tracking and measurement of address translation overheads through hardware performance counters. Our evaluation shows that HawkEye is more performant, robust and better-suited to handle diverse workloads when compared with current state-of-the-art systems.

CCS Concepts • **Software and its engineering → Operating systems**; **Virtual memory**;

Keywords Virtual memory; huge pages; hardware counters

ACM Reference Format:
Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304064

1 Introduction

Modern applications with large memory footprints have put address translation overheads in general-purpose processors into focus [32, 49, 59, 61, 63]. Modern architectures implementing large multi-level TLBs and page-walk caches, all supporting multiple page sizes [35, 40], require careful OS design to determine suitable page sizes for different workloads [47, 55, 57, 64]. The problem becomes more severe with virtual machines where two layers of address translation cause additional MMU overheads [34, 46, 62].

Despite robust hardware support available across architectures [15], huge pages have provided unsatisfactory performance on important applications [6, 9, 10, 12, 23–25]. These performance issues are often due to inadequate OS-based huge page management algorithms [5, 7, 11].

OS-based huge page management algorithms need to balance complex trade-offs between address translation overheads (aka MMU overheads), memory bloat, page fault latency, fairness and the overheads of the algorithm itself. In this paper, we discuss some subtleties related to huge pages and expose important weaknesses of current approaches, and propose a new set of algorithms to address them. We begin with a brief overview of the three representative state-of-the-art systems: Linux, FreeBSD, and a recent research paper by Kwon et al., i.e., Ingens [55].

Linux: Linux's transparent huge page (THP) employs huge pages through two mechanisms: (1) either it allocates a huge page at the time of page fault if contiguous memory is available, or (2) it promotes base pages to huge pages, by optionally compacting memory [39], in a background kernel thread called khugepaged. Linux triggers background promotion when fragmentation is high and huge pages are difficult to allocate at the time of page fault. While promoting huge pages, khugepaged selects processes in first-come-first-serve (FCFS) order and promotes all huge pages in a process before selecting the next process. For security, a page is zeroed synchronously (except for copy-on-write pages) before getting mapped into the user process' page table.

FreeBSD: FreeBSD supports multiple huge page sizes [57]. Unlike Linux, FreeBSD *reserves* a contiguous region of physical memory in the page fault handler but defers the promotion of baseline pages until all base pages in a huge page sized region are allocated by the application. If the reserved memory region is only partially mapped, its unused pages are returned back to the page allocator when memory pressure increases. This way, FreeBSD manages memory contiguity more efficiently than Linux, at the potential expense of a

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304064

higher number of page faults and higher MMU overheads due to multiple TLB entries, one per baseline page.

Ingens: In the Ingens paper [55], the authors point out pitfalls in the huge page management policies of both Linux and FreeBSD and present a policy that is better at handling the associated trade-offs. In summary: (1) Ingens uses an adaptive strategy to balance address translation overhead and memory bloat: it uses conservative utilization-threshold based huge page allocation to prevent memory bloat under high memory pressure but relaxes the threshold to allocate huge pages aggressively under no memory pressure, to try and achieve the best of both worlds. (2) To avoid high page fault latency associated with synchronous page-zeroing, Ingens employs asynchronous huge page allocation with a dedicated kernel thread. (3) To maintain fairness across multiple processes, Ingens treats memory contiguity as a resource and employs a share-based policy to allocate huge pages fairly.

The Ingens paper highlights that current OSs deal with huge page management issues through "spot fixes", and motivates the need for a principled approach. While Ingens proposes a more sophisticated strategy than previous work, we show that its static configuration based and heuristic-driven approaches are suboptimal: conflicting performance objectives and inadequate knowledge of MMU overheads of running applications can limit its effectiveness. Several aspects of an OS-based huge page management system can thus benefit from a dynamic data-driven approach.

This paper presents HawkEye, an automated OS-level solution for huge page management. HawkEye proposes a set of simple-yet-effective algorithms to: (1) balance the tradeoffs between memory bloat, address translation overheads, and page fault latency, (2) allocate huge pages to applications with highest expected performance improvement due to the allocation, and (3) improve memory sharing behaviour in virtualized systems. We also show that the actual address-translation overheads, as measured through performance counters, can be sometimes quite different from the expected/estimated overheads. To demonstrate this, we also evaluate a variant of HawkEye that relies on hardware performance counters for making huge page allocation decisions, and compare results. Our evaluation involves workloads with a diverse set of requirements vis-a-vis huge page management, and demonstrates that HawkEye can achieve considerable performance improvement over existing systems while adding negligible overhead (≈ 3.4% single-core overhead in the worst-case).

2 Motivation

In this section, we discuss different tradeoffs involved in OS-level huge page management and how current solutions handle them. We also provide a glimpse of the results achieved through HawkEye which is discussed in detail in §3.

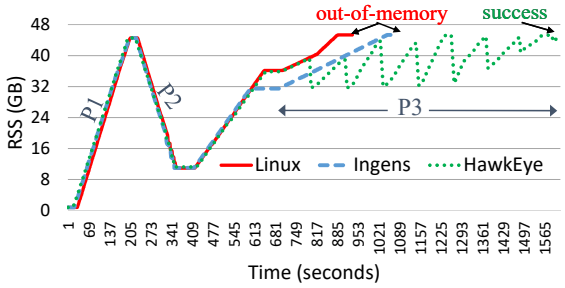


Figure 1. Resident Set Size (RSS) of `Redis` server across 3 phases: P1 (insert), P2(delete) and P3(insert).

2.1 Address Translation Overhead vs. Memory Bloat

One of the most important tradeoffs associated with huge pages is between address translation overheads and memory bloat. While Linux's synchronous huge page allocation is aimed at minimizing MMU overheads, it can often lead to memory bloat when an application uses only a fraction of the allocated huge page. FreeBSD promotes only after all base pages in a huge page region are allocated. FreeBSD's conservative approach tackles memory bloat but sacrifices performance by delaying the mapping of huge pages.

Ingens' strategy is adaptive, in that it promotes huge pages aggressively to minimize MMU overheads when memory fragmentation is low. To measure fragmentation, it uses the Free Memory Fragmentation Index (FMFI [50]): when FMFI < 0.5 (low fragmentation), Ingens behaves like Linux, promoting allocated pages to huge pages at the first available opportunity; when FMFI > 0.5 (high fragmentation), Ingens uses a conservative utilization-based strategy (i.e., promote only after a certain fraction of base pages, say 90%, are allocated) to mitigate memory bloat. While Ingens appears to capture the best of both Linux and FreeBSD, we show that this adaptive policy is far from a complete solution because *memory bloat generated in the aggressive phase remains unrecovered*. This property is also true for Linux and we demonstrate this problem through a simple experiment with the `Redis` key-value store [38] running on a 48GB memory system (see Figure 1).

We execute a client to generate memory bloat and measure the interaction of different huge page policies with `Redis`. For exposition, we divide our workload into three phases: (P1) The client inserts 11 million key-value pairs of size (10B, 4KB) for an in-memory dataset of 45GB. (P2) The client deletes 80% randomly selected keys leaving `Redis` with a sparsely populated address space. (P3) After some time gap, the client tries to insert 17K (10B, 2MB) key-value pairs so that the dataset again reaches 45GB. While this workload is used for clear exposition of the problem, it resembles realistic workload patterns that involve a mix of allocation and deallocation, and leave the address space fragmented. An ideal system should recover elegantly from memory bloat to avoid disruptions under high memory pressure.

In phase P2, when `Redis` releases pages back to the OS through `madvise` system call [16], the corresponding huge page mappings are broken by the kernel and the resident-set size (RSS) reduces to around 11GB. At this point, the kernel's `khugepaged` thread promotes the regions containing the deallocated pages back to huge pages. As more allocations happen in phase P3, Ingens promotes huge pages aggressively until the RSS reaches 32GB (or fragmentation is low); after that, Ingens employs conservative promotion to avoid any further bloat. This is evident in Figure 1 where the rate of RSS growth decreases, compared to Linux. However both Linux and Ingens reach memory limit (out-of-memory OOM exception is thrown) at significantly lower memory utilization. While Linux generates a bloat of 28GB (i.e., only 20GB useful data), Ingens generates a bloat of 20GB (i.e., 28GB useful data). Although Ingens tries to prevent bloat at high memory utilization, it is unable to recover from bloat generated at low memory utilization.

We note that it is possible to avoid memory bloat in Ingens by configuring it to use only conservative promotion. However, as also explained in the Ingens paper, this strategy risks losing performance due to high MMU overheads in low memory pressure situations. An ideal strategy should promote huge pages aggressively but should also be able to recover from bloat in an elegant fashion. Figure 1 shows HawkEye's behaviour for this workload, which is able to effectively recover from memory bloat, even with an aggressive promotion strategy.

2.2 Page fault latency vs. Number of page faults

The OS often needs to clear (zero) a page before mapping it into the process address space to prevent insecure information flow. Clearing a page is significantly more expensive for huge pages: in Linux on our experimental system, clearing a base page takes about 25% of total page fault time which increases to 97% for huge pages! High page fault latency often leads to high user-perceived latencies, jeopardizing performance for interactive applications. Ingens avoids this problem by allocating only base pages in the page fault handler and relegating the promotion task to an asynchronous thread `khugepaged`. It prioritizes the promotion of recently faulted regions over older allocations.

While Ingens reduces page allocation latency, it nullifies an important advantage of huge pages, namely fewer page faults for access patterns exhibiting high spatial locality [26]. Several applications that allocate and initialize a large memory region sequentially exhibit this type of access pattern and their performance degrades due to higher number of page faults if only base pages are allocated. We demonstrate the severity of this problem through a custom microbenchmark that allocates a 10GB buffer, touching one byte in every base page and later frees the buffer. Table 1 shows the cumulative performance summary for 10 runs of this workload.

Event	sync page-zeroing		async promotion	no page-zeroing	
	Linux 4KB	Linux 2MB	Ingens 90%	Linux 4KB	Linux 2MB
# Page faults	26.2M	51.5K	26.2M	26.2M	51.5K
Total page fault time (secs)	92.6	23.9	92.8	69.5	0.7
Avg page fault time (μs)	3.5	465	3.5	2.65	13
System time (secs)	102	24	104	79	1.3
Total time (secs)	106	24.9	116	83	4.4

Table 1. Page faults, allocation latency and performance for a microbenchmark with ≈100GB memory allocation.

Linux with THP support (Linux-2MB with sync page-zeroing in Table 1) reduces the number of page faults by more than 500× over Linux without THP support (Linux-4KB) and leads to more than 4× performance improvement for this workload, despite being 133× worse on average page fault latency (465μs vs. 3.5μs). Ingens considerably reduces latency compared to Linux-2MB but does not reduce the number of page faults for this workload because asynchronous promotion is activated only after 90% base pages are allocated from a region; copying these pages to a contiguous memory block takes more time than the time taken by this workload to generate the remaining 10% page faults in the same region. Consequently, the overall performance of this workload degrades in Ingens due to excessive page faults. If it were possible to allocate pages without having to zero them at page fault time, we could achieve *both* low page fault latency and fewer page faults resulting in higher overall performance over all existing approaches (last two columns in Table 1). HawkEye implements rate-limited asynchronous page pre-zeroing (§3.1) to achieve this in the common case.

2.3 Huge page allocation across multiple processes

Under memory fragmentation, the OS must allocate huge pages among multiple processes *fairly*. Ingens authors tackled this problem by defining fairness through a *proportional huge page promotion metric* wherein they consider "memory contiguity as a resource" and try and be equitable in distributing it (through huge pages) among applications. Ingens penalizes applications that have been allocated huge pages but are not accessing them frequently (i.e., have idle huge pages). Idleness of a page is estimated through the access-bit present in the page table entries which is maintained by the hardware and periodically cleared by the OS. Ingens employs an *idleness penalty factor* whereby an application is penalized for its idle, or *cold*, huge pages during the computation of the proportional huge page promotion metric.

Ingens' fairness policy has an important weakness — two processes may have similar huge page requirements but one of them (say P1) may have significantly higher TLB pressure than the other (say P2). This can happen, for example, if P1's accesses are spread across many base pages within its huge page regions, while P2's accesses are concentrated in one (or a few) base pages within its huge page regions. In this scenario, promotion of huge pages in P1 is more desirable than P2 but Ingens would treat them equally.

Benchmark Suite	Number of applications	
	Total	TLB sensitive applications
SPEC CPU2006_int	12	4 (mcf, astar, omnetpp, xalancbmk)
SPEC CPU2006_fp	19	3 (zeusmp, GemsFDTD, cactusADM)
PARSEC	13	2 (canneal, dedup)
SPLASH-2	10	0
Biobench	9	2 (tigr, mummer)
NPB	9	2 (cg, bt)
CloudSuite	7	2 (graph-analytics, data-analytics)
Total	79	15

Table 2. Number of TLB sensitive applications in popular benchmark suites.

Table 2 provides some empirical evidence that different applications usually behave quite differently vis-a-vis huge pages. For example, less than 20% of the applications in popular benchmark suites experience noticeable performance improvement (> 3%) with huge pages.

We posit that instead of considering "memory contiguity as a resource" and granting it equally among processes, it is more effective to consider "MMU overheads as a system overhead" and try and ensure that it is distributed equally across processes. A fair algorithm should attempt to equalize MMU overheads across all applications, e.g., if two processes P1 and P2 experience 30% and 10% MMU overheads resp., then huge pages should be allocated to P1 until its overhead also reaches 10%. In doing so, it should be acceptable if P1 has to be allocated more huge pages than P2. Such a policy would additionally yield the best overall system performance by helping the most afflicted processes first.

Further, in Ingens, huge page promotion is triggered in response to a few page-faults in the aggressive (non-fragmented) phase. These huge pages are not necessarily frequently accessed by the application; yet they contribute to a process's allocation quota of huge pages. Under memory pressure, these idle huge pages lower the promotion metric of a process, potentially preventing the OS from allocating more huge pages to it. This would increase MMU overheads if the process had other *hot* (frequently accessed) memory regions that required huge page promotion. This behaviour where previously-allocated cold huge pages can prevent an application from being allocated huge pages for its hot regions seems sub-optimal and avoidable.

Finally, within a process, Linux and Ingens promote huge pages through a *sequential* scan from lower to higher VAs. This approach is unfair to processes whose hot regions lie in the higher VAs. Because different applications would usually contain hot regions in different parts of their VA spaces (see Figure 6), this scheme is likely to cause unfairness in practice.

2.4 How to capture address translation overheads?

It is common to estimate MMU overheads based on working-set size (WSS) [56]: bigger WSS should entail higher MMU and performance overheads. We find that this is often not true for modern hardware where access patterns play an important role in determining MMU overheads, e.g., sequential access patterns allow prefetching to hide TLB miss latencies. Further, different TLB miss requests can experience high latency variations: a translation may be present anywhere in the multi-level page walk caches, multi-level regular data caches or in main memory. For these reasons, WSS is often not a good indicator of MMU overheads. Table 3 demonstrates this with the NPB benchmark suite [30]: a workload with large WSS (e.g., `mg.D`) can have low MMU overheads compared to one with a smaller WSS (e.g., `cg.D`).

Workload	RSS	WSS	% TLB-misses (native-4KB)	% cycles		speedup	
				4KB	2MB	native	virtual
bt.D	10GB	7-10 GB	0.45	6.4	1.31	1.05	1.15
sp.D	12GB	8-12 GB	0.48	4.7	0.25	1.01	1.06
lu.D	8GB	8 GB	0.06	3.3	0.18	1.0	1.01
mg.D	26GB	24 GB	0.03	1.04	0.04	1.01	1.11
cg.D	16GB	7-8 GB	28.57	39	0.02	1.62	2.7
ft.D	78 GB	7-35 GB	0.21	3.9	2.14	1.01	1.04
ua.D	9.6 GB	5-7 GB	0.01	0.8	0.03	1.01	1.03

Table 3. Memory characteristics, address translation overheads and speedup huge pages provide over base pages for NPB workloads.

	Performance Counter
C1	DTLB_LOAD_MISSES_WALK_DURATION
C2	DTLB_STORE_MISSES_WALK_DURATION
C3	CPU_CLK_UNHALTED
MMU Overhead = ((C1 + C2) * 100) / C3	

Table 4. Methodology used to measure MMU Overhead [54].

It is not clear to us how an OS can reliably capture MMU overheads: these are dependent on complex interactions between applications and the underlying hardware architecture, and we show that the actual address translation overheads of a workload can be quite different from what can be estimated through its memory access pattern (e.g., working-set size). Hence, we propose directly measuring TLB overheads through hardware performance counters when available (see Table 4 for methodology). This approach enables a more efficient solution at the cost of portability as the required performance counters may not be available on all platforms (e.g., most hypervisors have not yet virtualized TLB-related performance counters). To overcome this challenge, we present two variants of our algorithm/implementation in HawkEye/Linux, one where the MMU overheads are *measured* through hardware performance counters (HawkEye-PMU), and another where MMU overheads are *estimated* through the memory access pattern (HawkEye-G). We compare both approaches in §4.

3 Design and Implementation

Figure 2 shows our high-level design objectives. Our solution is based on four primary observations: (1) high page fault latency for huge pages can be avoided by asynchronously pre-zeroing free pages; (2) memory bloat can be tackled by identifying and de-duplicating zero-filled baseline pages present within allocated huge pages; (3) promotion decisions should be based on finer-grained access tracking of huge page sized regions, and should include recency, frequency,

Figure 2. Design objectives in HawkEye.

and *access-coverage* (i.e., how many baseline pages are accessed inside a huge page) measurements; and (4) fairness should be based on estimation of MMU overheads.

3.1 Asynchronous page pre-zeroing

We propose that page zeroing of available free pages should be performed asynchronously in a rate-limited background kernel thread to eliminate high latency allocation. We call this scheme *async pre-zeroing*. Async pre-zeroing is a rather old idea and had been discussed extensively among kernel developers in early 2000s [1, 2, 14, 41][1]. Linux developers opined that async pre-zeroing is not an overall performance win for two main reasons. We think that it is time to revisit these opinions.

First, the async pre-zeroing thread might interfere with primary workloads by polluting the cache [1]. In particular, async pre-zeroing suffers from the "double cache miss" problem because it causes the same datum to be accessed twice with a large re-use distance: first for pre-zeroing, and then for the actual access by the application. These extra cache misses are expected to degrade overall performance in general. However, these problems are partially solvable on modern hardware that support memory writes with non-temporal hints: non-temporal hints instruct the hardware to bypass caches during memory load/store instructions [43]. We find that using non-temporal hints during pre-zeroing significantly reduces both cache contention and the double cache miss problem.

Second, there was no consensus or empirical evidence to demonstrate the benefits of page pre-zeroing for real workloads [1, 42]. We note that the early discussions on page pre-zeroing were evaluating trade-offs with baseline 4KB pages. Our experiments corroborate the kernel developers' observation that despite reducing the page fault overhead by 25%, pre-zeroing does not necessarily enable high performance with 4KB pages. At the same time, we also show that it enables non-negligible performance improvements (e.g., 14× faster VM boot-time) with huge pages, due to much higher reduction (97%) in page fault overheads. Since huge pages (and huge-huge pages) are supported by most general-purpose processors today [15, 27], pre-zeroing pages is an important optimization that is worth revisiting.

Pre-zeroing offers another advantage for virtualized systems: it increases the number of zero pages in the guest's physical address (GPA) space enabling opportunities for content-based page-sharing at the virtualization host. We evaluate this aspect in §4.

To implement async pre-zeroing, HawkEye manages free pages in the Linux buddy allocator through two lists: `zero` and `non-zero`. Pages released by applications are first added to the `non-zero` list while the `zero` list is preferentially used for allocation. A rate-limited thread periodically transfers pages from `non-zero` to `zero` lists after zero-filling them using non-temporal writes. Because pre-zeroing involves sequential memory accesses, non-temporal store instructions provide performance similar to regular (caching) store instructions, but without polluting the cache [3]. Finally, we note that for copy-on-write or filesystem-backed memory regions, pre-zeroing may sometimes be unnecessary and wasteful. This problem is avoidable by preferentially allocating pages for these memory regions from the `non-zero` list.

Overall, we believe that async pre-zeroing is a compelling idea for modern workload requirements and modern hardware support. In our evaluation, we provide some early evidence to corroborate this claim with different workloads.

3.2 Managing bloat vs. performance

We observe that the fundamental tension between MMU overheads and memory bloat can be resolved. Our approach stems from the insight that most allocations in large-memory workloads are typically "zero-filled page allocations"; the remaining are either filesystem-backed (e.g., through `mmap`) or copy-on-write (COW) pages. However, huge pages in modern scale-out workloads are primarily used for "anonymous" pages that are initially zero-filled by the kernel [54], e.g., Linux supports huge pages only for anonymous memory. This property of typical workloads and Linux allows automatic recovery of bloat under memory pressure.

To state our approach succinctly: we allocate huge pages at the time of first page-fault; but under memory pressure, to recover unused memory, we scan existing huge pages to identify zero-filled baseline pages within them. If the number of zero-filled baseline pages inside a huge page is significant (i.e., beyond a threshold), we break the huge page into its constituent baseline pages and de-duplicate the zero-filled baseline pages to a canonical zero-page through standard COW page management techniques [37]. In this approach, it is possible for applications' in-use zero-pages to also get de-duplicated. While this can result in a marginally higher number of COW page faults in rare situations, this does not compromise correctness.

To trigger recovery from memory bloat, HawkEye uses two watermarks on the amount of allocated memory in the system: `high` and `low`. When the amount of allocated memory exceeds `high` (85% in our prototype), a rate-limited `bloat-recovery` thread is activated which executes periodically until the allocated memory falls below `low` (70% in our prototype). At each step, the `bloat-recovery` thread

[1] Windows and FreeBSD implement async pre-zeroing in a limited fashion [4, 19]. However, their huge page policies do not allow high latency allocations. This idea is more relevant for Linux due to its synchronous huge page allocation behaviour.

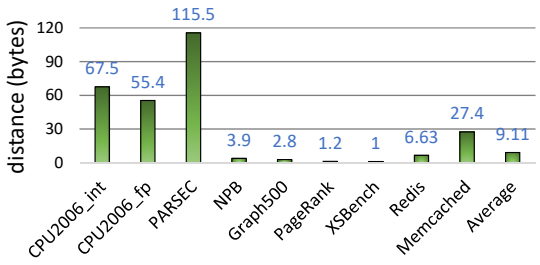


Figure 3. Average distance to the first non-zero byte in baseline (4KB) pages. First four bars represent the average of all workloads in the respective benchmark suite.

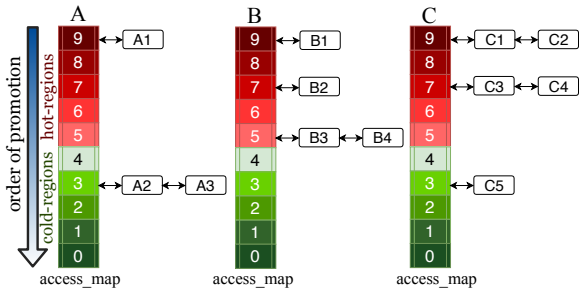


Figure 4. A sample representation of `access_map` for three processes A, B and C.

chooses the application whose huge page allocations need to be scanned (and potentially demoted) based on the estimated MMU overheads of that application: the application with the *lowest* estimated MMU overheads is chosen first for scanning. This strategy ensures that the application that least requires huge pages is considered first — this is consistent with our huge page allocation strategy (§2.3).

While scanning a baseline page to verify if it is zero-filled, we stop on encountering the first non-zero byte in it. In practice, the number of bytes that need to be scanned per in-use (not zero-filled) page before a non-zero byte is encountered is very small: we measured this over a total of 56 diverse workloads, and found that the average distance of the first non-zero byte in a 4KB page is only 9.11 (see Figure 3). Hence, only ten bytes need to be scanned on average per in-use application page. For bloat pages however, all 4096 bytes need to be scanned. This implies that the overheads of our `bloat-recovery` thread are largely proportional to the number of bloat pages in the system, and *not* to the total size of the allocated memory. This is an important property that allows our method to scale to large memory systems.

We note that our `bloat-recovery` procedure has many similarities with the standard de-duplication kernel threads used for content-based page sharing for virtual machines [67], e.g., the kernel same-page merging (`ksm`) thread in Linux. Unfortunately, in current kernels, the huge page management logic (e.g., `khugepaged`) and the content-based page sharing logic (e.g., `ksm`) are unconnected and can often interact in counter-productive ways [51]. Ingens and SmartMD [52] proposed coordinated mechanisms to avoid such conflicts: Ingens demotes only infrequently-accessed huge pages through `ksm` while SmartMD demotes pages based on access-frequency and repetition rate (i.e., the number of shareable pages within a huge page). These techniques are useful for in-use pages and our `bloat-recovery` proposal complements them by identifying unused zero-filled pages, which can execute much faster than typical same-page merging logic.

3.3 Fine-grained huge page promotion

An efficient huge page promotion strategy should try to maximize performance with a minimal number of huge page promotions. Current systems promote pages through a *sequential* scan from lower to higher VAs which is inefficient for applications whose hot regions are not in lower VA space. Our approach makes promotion decisions based on memory access patterns. First we define an important metric used in HawkEye to improve the efficiency of huge page promotions. **Access-coverage** denotes the number of base pages that are accessed from a huge page sized region in short intervals. We sample the page table access bits at regular intervals and maintain the exponential moving average (EMA) of access-coverage across different samples. More precisely, we clear the page table access bits and test the number of *set* bits after 1 second to check how many pages are accessed. This process is repeated once every 30 seconds.

The access-coverage of a region potentially indicates its TLB space requirement (number of base page entries required in the TLB), which we use to *estimate* the profitability of promoting it to a huge page. A region with high access-coverage is likely to exhibit high contention on TLB and hence likely to benefit more from promotion.

HawkEye implements access-coverage based promotion using a per-process data structure, called `access_map`, which is an array of buckets: a bucket contains huge page regions with similar access-coverage. On x86 with 2MB huge pages, the value of access-coverage is in the range of 0-512. In our prototype, we maintain ten buckets in the `access_map` which provides the necessary resolution across access-coverage values at relatively low overheads: regions with access-coverage of 0-49 are placed in bucket 0, regions with access-coverage of 50 – 99 are placed in bucket 1, and so on. Figure 4 shows an example state of the `access_map` of three processes A, B and C. Regions can move up or down in their respective arrays after every sampling period, depending on their newly computed EMA-based access-coverage. If a region moves up in `access_map`, it is added to the head of its bucket. If a region moves down, it is added to the tail. Within a bucket, pages are promoted from head to tail. This strategy helps in prioritizing recently accessed regions within an index.

HawkEye promotes regions from higher to lower indices in `access_map`. Notice that our approach captures both recency and frequency of accesses: a region that has not been

accessed or accessed with low access-coverage in recent sampling periods is likely to shift towards a lower index in access_map or towards the tail in its current index. Promotion of cold regions is thus automatically deferred to prioritize hot regions.

We note that HawkEye's access_map is somewhat similar to population_map and access_bitvector used in FreeBSD and Ingens resp.; these data structures are primarily used to capture utilization or page access related metadata at huge page granularity. HawkEye's access_map additionally provides the index of hotness of memory regions and enables fine-grained huge page promotion.

3.4 Huge page allocation across multiple processes

In our access-coverage based strategy, regions belonging to a process with the highest expected MMU overheads are promoted before others. This is also consistent with our notion of fairness §2.3, as pages are promoted first from regions/processes with high expected TLB pressure (and consequently high expected MMU overheads).

Our HawkEye variant that does not rely on hardware performance counters (i.e., HawkEye-G), promotes regions from the non-empty *highest access-coverage index* in all processes. It is possible that multiple processes have non-empty buckets in the (globally) highest non-empty index. In this case, round-robin is used to ensure fairness among such processes. We explain this further with an example. Consider three applications A, B and C with their VA regions arranged in the access_map as shown in Figure 4. HawkEye-G promotes regions in the following order in this example:

A1,B1,C1,C2,B2,C3,C4,B3,B4,A2,C5,A3

Recall however that MMU overheads may not necessarily be correlated with our access-coverage based estimation, and may depend on other more complex features of the access pattern (§2.4). To capture this in HawkEye-PMU, we first choose the process with the highest measured MMU overheads, and then choose regions from higher to lower indices in selected process's access_map. Among processes with similar highest MMU overheads, round-robin is used.

3.5 Limitations and discussion

We briefly outline a few issues that are related to huge page management but are currently unhandled in HawkEye.

1) Thresholds for identifying memory pressure: We measure the extent of memory pressure with statically configured values for low and high watermarks (i.e., 70% and 85% of total system memory) while dealing with memory bloat. Any strategy that relies on static thresholds faces the risk of being conservative or overly aggressive when memory pressure consistently fluctuates. An ideal solution should adjust these thresholds dynamically to prevent unintended system behavior. The approach proposed by Guo et. al. [51] in the context of memory deduplication for virtualized environments is relevant in this context.

2) Huge page starvation: While we believe our approach of allocating huge pages based on MMU overheads optimizes the system as a whole, unbounded huge page allocations to a single process can be thought of as a *starvation* problem for other processes. An adverserial application can also potentially monopolize HawkEye to get more huge pages or prevent other applications from getting a fair share of huge pages. Preliminary investigations show that our approach is reasonable even if the memory footprint of workloads differ by more than an order of magnitude. However, if limiting huge page allocations is still desirable, it seems reasonable to integrate a policy with existing resource limiting/monitoring tools, such as Linux's cgroups [18].

3) Other algorithms: We do not discuss some parts of the management algorithms, such as rate limiting khugepaged to reduce promotion overheads, demotion based on low utilization to enable better sharing through same-page merging, and techniques for minimizing the overhead of page-table access-bit tracking and compaction algorithms. Much of this material has been discussed and evaluated extensively in the literature [32, 55, 59, 68], and we have not contributed significantly new approaches in these areas.

4 Evaluation

We now evaluate our algorithms in more detail. Our experimental platform is an Intel Haswell-EP based E5-2690 v3 server system running CentOS v7.4, on which 96GB memory and 48 cores (with hyperthreading enabled) running at 2.3GHz are partitioned on two sockets: we bind each workload to a single socket to avoid NUMA effects. The L1 TLB contains 64 and 8 entries for 4KB and 2MB pages respectively while the L2 TLB contains 1024 entries for both 4KB and 2MB pages. The size of L1, L2 and shared L3 cache is 768KB, 3MB and 30MB resp. A 96GB SSD-backed swap partition is used to evaluate performance in an overcommitted system. We evaluate HawkEye with a diverse set of workloads ranging from HPC, graph algorithms, in-memory databases, genomics and machine learning [24, 30, 33, 36, 45, 53, 65, 66]. HawkEye is implemented in Linux kernel v4.3.

We evaluate (a) the improvements due to our fine-grained promotion strategy based on access-coverage in both performance and fairness for single, multiple homogeneous, and multiple heterogeneous workloads; (b) the bloat-vs-performance tradeoff; (c) the impact of low latency page faults; (d) cache interference caused by asynchronous pre-zeroing thread; and (e) the impact of memory efficiency enabled by asynchronous page pre-zeroing.

Performance advantages of fine-grained huge page promotion: We first evaluate the effectiveness of our access-coverage based promotion strategy. For this, we measure the time required for our algorithm to recover from a fragmented state with high address translation overheads to a state with

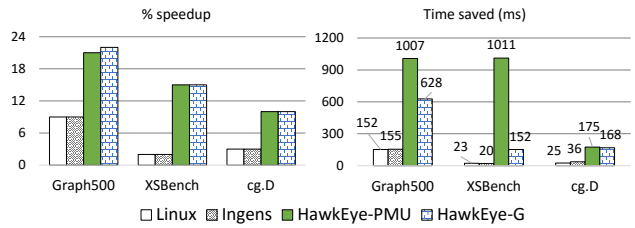


Figure 5. Performance speedup and time saved per huge page promotion over baseline pages.

Workload	Execution Time (Seconds)				
	Linux-4KB	Linux-2MB	Ingens	HawkEye-PMU	HawkEye-G
Graph500-1	2270	2145(1.06)	2243(1.01)	1987(1.14)	2007(1.13)
Graph500-2	2289	2252(1.02)	2253(1.02)	1994(1.15)	2013(1.14)
Graph500-3	2293	2293(1.0)	2299(1.00)	2012(1.14)	2018(1.14)
Average	2284	2230(1.02)	2265(1.01)	1998(1.14)	2013(1.13)
XSBench-1	2427	2415(1.0)	2392(1.01)	2108(1.15)	2098(1.15)
XSBench-2	2437	2427(1.0)	2415(1.01)	2109(1.15)	2110(1.15)
XSBench-3	2443	2455(1.0)	2456(1.00)	2133(1.15)	2143(1.14)
Average	2436	2432(1.0)	2421(1.00)	2117(1.15)	2117(1.15)

Table 5. Execution time of 3 instances of `Graph500` and `XSBench` when executed simultaneously. Values in parentheses represent speedup over baseline pages.

low address translation overheads. Recall that HawkEye promotes pages based on access-coverage while previous approaches promote pages in VA order (from low VAs to high). We fragment the memory initially by reading several files in memory; our test workloads are started in the fragmented state and we measure the time taken for the system to recover from high MMU overheads. This experimental setup simulates expected realistic situations where the memory fragmentation in the system fluctuates over time. Without promotion, our test workloads would keep incurring high address translation overheads; however HawkEye is quickly able to recover from these overheads through appropriate huge page allocations. Figure 5 (left) shows the performance improvement obtained by HawkEye (over a strategy that never promotes the pages) for three workloads: `Graph500`, `XSBench` and `cg.D`. These workloads allocate all their required memory in the beginning, i.e., in the fragmented state of the system. For these workloads, the speedups due to effective huge page management through HawkEye are as high as 22%. Compared to Linux and Ingens, access-coverage based huge page promotion strategy of HawkEye improves performance by 13%, 12% and 6% over both Linux and Ingens.

To understand this more clearly, Figure 6 shows the access pattern, MMU overheads and the number of allocated huge pages over time for `Graph500` and `XSBench`. We find that hot-spots in these applications are concentrated in the high VAs and any sequential-scanning based promotion is likely to be sub-optimal. The second and third columns corroborate this observation: for example, both HawkEye variants take ≈300 seconds to eliminate MMU overheads of `XSBench` while Linux and Ingens have high overheads even after 1000 seconds.

To quantify the cost-benefit analysis of huge page promotions further, we propose a new metric: the average execution time saved (over using only baseline pages) per huge page promotion. A scheme that maximizes this metric would be most effective in reducing MMU overheads. Figure 5 (right) shows that HawkEye performs significantly better than Linux and Ingens on this metric. The difference between the efficiency of HawkEye-PMU and HawkEye-G is also evident: HawkEye-PMU is more efficient as it stops promoting huge pages when MMU overheads are below a certain threshold (2% in our experiments). In summary, HawkEye-G and HawkEye-PMU are up to 6.7× and 44× more efficient (for `XSBench`) than Linux in terms of time saved per huge page promotion. For workloads whose access patterns are spread uniformly over the VA space, HawkEye delivers performance similar to Linux and Ingens.

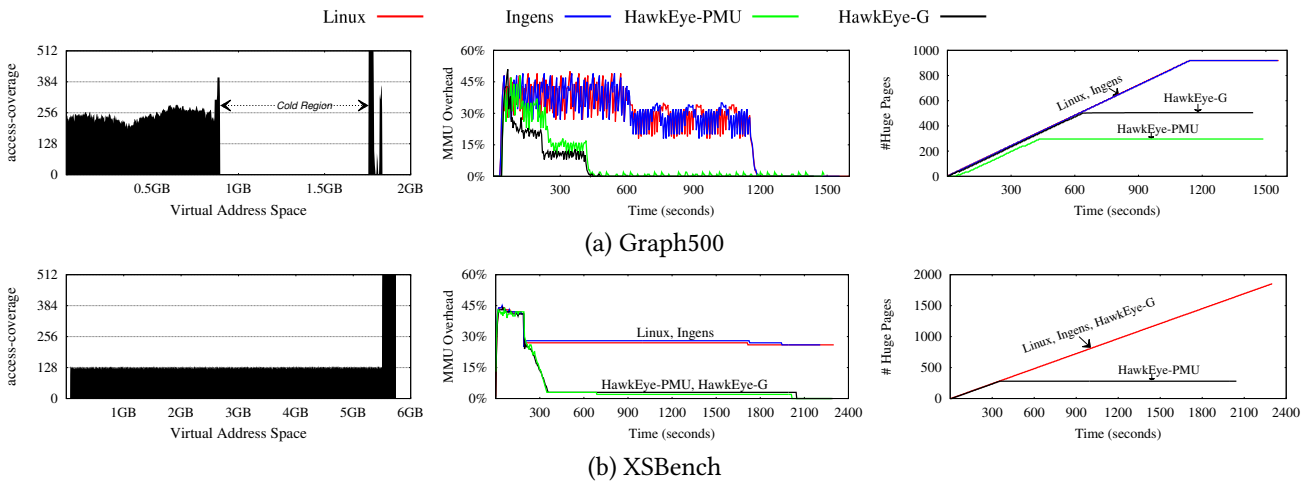
Fairness advantages of fine-grained huge page promotion: We next experiment with multiple applications to study both performance and fairness, first with identical applications, and then with heterogeneous applications, running simultaneously.

Identical workloads: Figure 7 shows the MMU overheads and huge page allocations when 3 instances of `Graph500` and `XSBench` (in separate runs) are executed concurrently after fragmenting the system (see Table 5 for execution time).

While Linux creates performance imbalance by promoting huge pages in one process at a time, HawkEye ensures fairness by judiciously distributing huge pages across all workload instances. Overall, HawkEye-PMU and HawkEye-G achieve 1.14× and 1.13× speedup for `Graph500` and 1.15× speedup for `XSBench` over Linux on average. Unlike Linux, Ingens promotes huge pages proportionally in all 3 instances. However, it fails to improve the performance of these workloads. In fact, it may lead to poorer performance than Linux. We explain this behaviour with an example.

Linux selects an address from `Graph500-1`'s address space and promotes all pages above it in around 10 minutes. Even though this strategy is unfair, it improves the performance of `Graph500-1` by promoting its hot-regions. Linux then selects `Graph500-2` whose MMU overheads decrease after ≈20 minutes. In contrast, Ingens promotes huge pages from lower VAs in each instance of `Graph500`. Thus it takes even longer for Ingens to promote suitable (hot) regions of the respective processes which leads to 9% performance degradation over Linux for `Graph500`. For `XSBench`, the performance of both Ingens and Linux is similar (but inferior to HawkEye) because both fail to promote the application's hot regions before the application finishes.

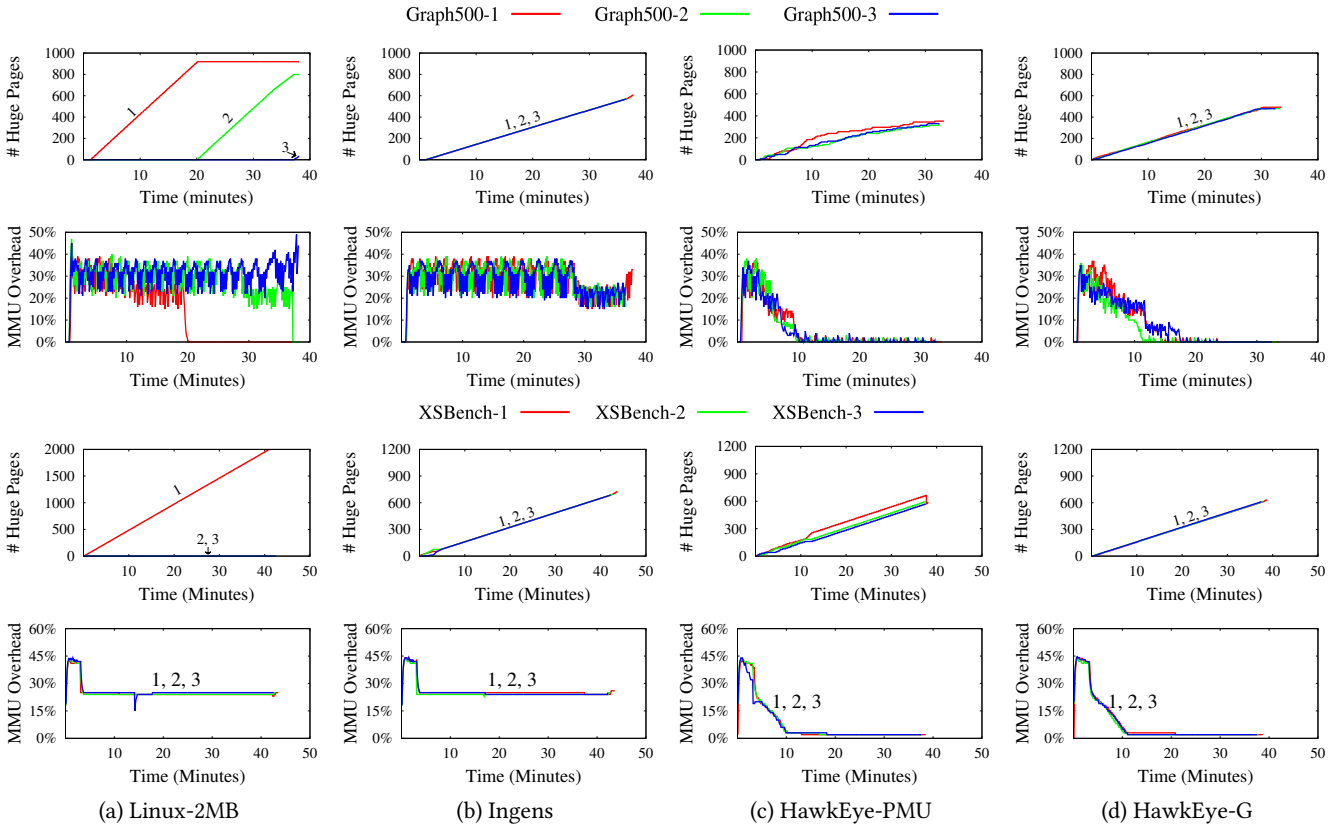
Heterogeneous workloads: To measure the efficacy of different strategies for heterogeneous workloads, we execute workloads by grouping them into sets where a set contains one TLB sensitive and one TLB insensitive application. Each



(a) Graph500

(b) XSBench

Figure 6. Access-coverage in application VA, MMU overhead and huge page promotions for `Graph500` and `XSBench`. HawkEye is more efficient in promoting huge pages due to fine-grained decision making.



(a) Linux-2MB (b) Ingens (c) HawkEye-PMU (d) HawkEye-G

Figure 7. Huge page promotions and their impact on TLB overhead over time for Linux, Ingens and HawkEye for three instances of `Graph500` and `XSBench` each.

set is executed twice, after fragmenting the system, to assess the impact of huge page policies on the order of execution. For the TLB insensitive workload, we execute a lightly loaded `Redis` server with 1KB-sized 40M keys servicing 10K requests per-second. Figure 8 shows the performance speedup over 4KB pages. The (Before) and (After) configurations show whether the TLB sensitive workload is launched before `Redis` or after. Because Linux promotes huge pages in process launch order, the two configurations (Before) and (After) are intended to cover its two different behaviours. Ingens and HawkEye are agnostic to process creation order.

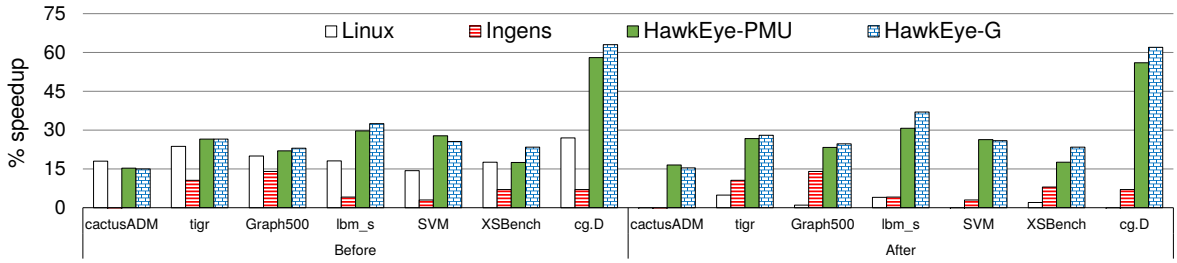


Figure 8. Performance speedup over baseline pages of TLB sensitive applications when they are executed alongside a lightly stressed `Redis` server in different orders.

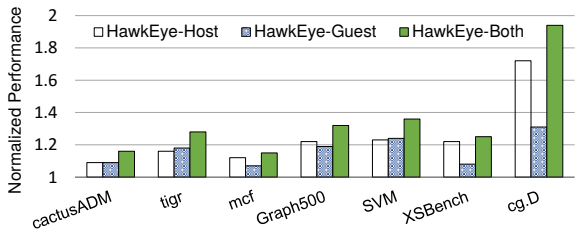


Figure 9. Performance compared to Linux in a virtualized system when HawkEye is applied at the host, guest and both layers.

The impact of execution order is clearly visible for Linux. In the (Before) case, Linux with THP support improves performance over baseline pages by promoting huge pages in TLB sensitive applications. However, in the (After) case, it promotes huge pages in `Redis` resulting in poor performance for TLB sensitive workloads. While the execution order does not matter in Ingens, its proportional huge page promotion strategy favors large memory workloads (`Redis` in this case). Further, our client requests randomly selected keys which makes the `Redis` server access all its huge pages uniformly. Consequently, Ingens promotes most huge pages in `Redis` leading to suboptimal performance of TLB sensitive workloads. In contrast, HawkEye promotes huge pages based on MMU overhead measurements (HawkEye-PMU) or access-coverage based estimation of MMU overheads (HawkEye-G). This leads to 15–60% performance improvement over 4KB pages irrespective of the order of execution.

Performance in virtualized systems: For a virtualized system, we use the KVM hypervisor running with Ubuntu16.04 and fragment the system prior to running the workloads. Evaluation is presented for three different configurations, i.e., when HawkEye is deployed at the host, guest and both the layers. Each of these configurations requires running the same set of applications differently (see Table 6 for details).

Figure 9 shows that HawkEye provides 18–90% speedup compared to Linux in virtual environments. Notice that in some cases (e.g., `cg.D`), performance improvement is higher in a virtualized system compared to bare-metal. This is due to high MMU overheads involved in traversing nested page tables while servicing TLB misses of guest applications, which presents higher opportunities to be exploited by HawkEye.

Configuration	Details
HawkEye-Host	Two VMs each with 24 vCPUs and 48GB memory. VM-1 runs `Redis`. VM-2 runs TLB sensitive workloads.
HawkEye-Guest	Single VM with 48 vCPUs and 80GB memory running `Redis` and TLB sensitive workloads
HawkEye-Both	Two VMs each with 24 vCPUs. VM-1 (30GB) runs `Redis`. VM-2 (60GB) runs both `Redis` and TLB sensitive workloads

Table 6. Experimental setup for configurations used to evaluate a virtualized system.

Bloat vs. performance: We next study the relationship between memory bloat and performance; we revisit an experiment similar to the one summarized in §2.1. We populate a `Redis` instance with 8M (10B key, 4KB value) key-value pairs and then delete 60% randomly selected keys (see Table 7). While Linux-4KB (no THP) is memory efficient (no bloat), its performance is low (7% lower throughput than Linux-2MB, i.e., with THP). Linux-2MB delivers high performance but consumes more memory which remains unavailable to the rest of system even when memory pressure increases. Ingens can be configured to balance the memory vs. performance tradeoff. For example, Ingens-90% (Ingens with 90% utilization threshold) is memory efficient while Ingens-50% (Ingens with 50% utilization threshold) favors performance and allows more memory bloat. In either case, it is unable to recover from bloat that may have already been generated (e.g., during its aggressive phase). By switching from aggressive huge page allocations under low memory pressure to memory conserving strategy at runtime, HawkEye is able to achieve both low MMU overheads and high memory efficiency depending on the state of the system.

Fast page faults: Table 8 shows the performance of different strategies; for the workloads chosen for these experiments, their performance depends on the efficiency of the OS page fault handler. We measure `Redis` throughput when 2MB-sized values are inserted. `SparseHash` [13] is a hash-map library in C++ while `HACC-IO` [8] is a parallel IO benchmark used with an in-memory file system. We also measure the spin-up time of two virtual machines: KVM and a Java Virtual Machine (JVM) where both are configured to allocate all memory during initialization.

Performance benefits of async page pre-zeroing with base pages (HawkEye-4KB) are modest: in this case, the other page

Kernel	Self-tuning	Memory Size	Throughput
Linux-4KB	No	16.2GB	106.1K
Linux-2MB	No	33.2GB	113.8K
Ingens-90%	No	16.3GB	106.8K
Ingens-50%	No	33.1GB	113.4K
HawkEye (no mem. pressure)	Yes	33.2GB	113.6K
HawkEye (mem. pressure)	Yes	16.2GB	105.8K

Table 7. `Redis` memory consumption and throughput.

Workload	OS Configuration				
	Linux 4KB	Linux 2MB	Ingens 90%	HawkEye 4KB	HawkEye 2MB
Redis (45GB)	233	437	192	236	551
SparseHash (36GB)	50.1	17.2	51.5	46.6	10.6
HACC-IO (6GB)	6.5	4.5	6.6	6.5	4.2
JVM Spin-up (36GB)	37.7	18.6	52.7	29.8	1.37
KVM Spin-up (36GB)	40.6	9.7	41.8	30.2	0.70

Table 8. Performance implications of asynchronous page zeroing. Values for `Redis` represent throughput (higher is better); all other values represent time in seconds (lower is better).

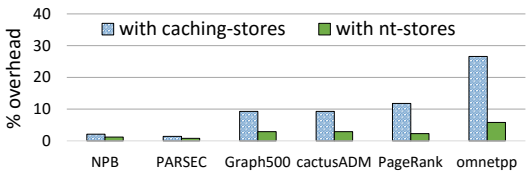


Figure 10. Performance overhead of async pre-zeroing with and without caching instructions. The first two bars (NPB and Parsec) represent the average of all workloads in the respective benchmark suite.

fault overheads (apart from zeroing) dominate performance. However, page-zeroing cost becomes significant with huge pages. Consequently, HawkEye with huge pages (HawkEye-2MB) improves the performance of `Redis` and `SparseHash` by 1.26× and 1.62× over Linux. The spin-up time of virtual machines is purely dominated by page faults. Hence we observe a dramatic reduction in the spin-up time of VMs with async pre-zeroing of huge pages: 13.8× and 13.6× over Linux. Notice that without huge pages (only base pages), this reduction is only 1.34× and 1.26×. All these workloads have high spatial locality of page faults and hence the Ingens strategy of utilization-based promotion has a negative impact on performance due to a higher number of page faults.

Async pre-zeroing enables memory sharing in virtualized environments: Finally, we note that async pre-zeroing within VMs enables memory sharing across VMs: the free memory of a VM returns to the host through pre-zeroing in the VM and same-page merging at the host. This can have the same net effect as ballooning, as the free memory in a VM gets shared with other VMs. In our experiments with memory over-committed systems, we have confirmed this behaviour: the overall performance of an overcommitted system, where the VMs are running HawkEye, matches the

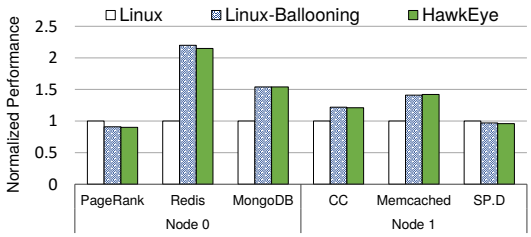


Figure 11. Performance normalized to the case of no ballooning in an overcommitted virtualized system.

performance achieved through para-virtual interfaces like memory balloon drivers. For example, in an experiment involving a mix of latency-sensitive key-value stores and HPC workloads (see Figure 11) with a total peak memory consumption of around 150GB (1.5× of total memory), HawkEye-G provides 2.3× and 1.42× higher throughput for `Redis` and `MongoDB` along with significant tail latency reduction. For `PageRank`, performance degrades slightly due to a higher number of COW page faults due to unnecessary same-page merging. These results are very close to the results obtained when memory ballooning is enabled on *all* VMs. We believe that this is a potential positive of our design and may offer an alternative method to efficiently manage memory in overcommitted systems. It is well-known that balloon-drivers and other para-virtual interfaces are riddled with compatibility problems [29], and a fully-virtual solution to this problem would be highly desirable. While our experiments show early promise in this direction, we leave an extensive evaluation for future work.

Performance overheads of HawkEye: We evaluated HawkEye with non-fragmented systems and with several workloads that don't benefit from huge pages, and find that the performance with HawkEye is very similar to performance with Linux or Ingens in these scenarios, confirming that HawkEye's algorithms for access-bit tracking, asynchronous huge page promotion and memory de-duplication add negligible overheads over existing mechanisms in Linux. However, the async pre-zeroing thread requires special attention as it may become a source of noticeable performance overhead for certain types of workloads, as discussed next.

Overheads of async pre-zeroing: A primary concern with async pre-zeroing is its potential detrimental effect due to memory and cache interference (§3.1). Recall that we employ memory stores with non-temporal hints to avoid these effects. To measure the worst-case cache effects of async pre-zeroing, we run our workloads while simultaneously zero-filling pages on a separate core sharing the same L3 cache (i.e., on the same socket) at a high rate of 0.25M pages per second (1GBps) with and without non-temporal memory stores. Figure 10 shows that using non-temporal hints significantly brings down the overhead of async pre-zeroing (e.g., from 27% to 6% for `omnetpp`): the remaining overhead is due to additional memory traffic generated by the pre-zeroing thread. Further we note that this experiment presents

Workload	MMU Overhead	Time (seconds)		
		4KB	HawkEye-PMU	HawkEye-G
random(4GB)	60%	582	328(1.77×)	413(1.41×)
sequential(4GB)	< 1%	517	535	532
Total		1099	863(1.27×)	945(1.16×)
cg.D(16GB)	39%	1952	1202(1.62×)	1450(1.35×)
mg.D(24GB)	< 1%	1363	1364	1377
Total		3315	2566(1.29×)	2827(1.17×)

Table 9. Comparison between HawkEye-PMU and HawkEye-G. Values in parentheses represent speedup over baseline pages.

a highly-exaggerated behavior of async pre-zeroing on worst-case workloads. In practice, the zeroing thread is rate-limited (e.g., at most 10k pages per second), and the expected cache-interference overheads would be proportionally smaller.

Comparison between Hawk-PMU and HawkEye-G: Even though HawkEye-G is based on simple and approximated estimations of TLB contention, it is reasonably accurate in identifying TLB sensitive processes and memory regions in most cases. However, HawkEye-PMU performs better than HawkEye-G in some cases. Table 9 demonstrates two such examples where four workloads, all with high access-coverage but different MMU overheads, are executed in two sets: each set contains one TLB sensitive and one TLB insensitive workload. The 4KB column represents the case where no huge pages are promoted. As discussed earlier (§2.4), MMU overheads depend on the complex interaction between the hardware and access pattern. In this case, despite having high access-coverage in VA regions, `sequential` and `mg.D` have negligible MMU overheads (i.e., they are TLB insensitive). While HawkEye-PMU correctly identifies the process with higher MMU overheads for huge page allocation, HawkEye-G treats them similarly (due to imprecise estimations) and may allocate huge pages to the TLB insensitive process. Consequently, HawkEye-PMU may perform up to 36% better than HawkEye-G. Bridging this gap between the two approaches in a hardware independent manner is an interesting future work.

5 Related Work

MMU overheads have been a topic of much research in recent years and several solutions have been proposed including solutions that are architecture-based, OS-based and/or compiler-based [28, 34, 49, 56, 58, 62–64].

Hardware Support: Most proposals for hardware support aim to reduce TLB-miss frequency or accelerate TLB-miss processing. Large multi-level TLBs and support for multiple page sizes to minimize the number of TLB-misses is already common in general-purpose processors today [15, 20, 21, 40]. Further, modern processors employ page-walk-caches to avoid memory lookups in the TLB-miss path [31, 35]. POM-TLB services a TLB miss using a single memory lookup and further leverages regular data caches to speed up address translation [63]. Direct segments [32, 46] were proposed to completely avoid TLB-miss processing cost through special segmentation hardware. OS-level challenges of memory fragmentation have also been considered in hardware designs: CoLT, or Coalesced-Large-reach TLBs [61], were initially proposed to increase TLB reach using base pages, based on the observation that OSs naturally provide contiguous mappings at smaller granularity. This approach was further extended to page-walk caches and huge pages [35, 40, 60]. Techniques to support huge pages in non-contiguous memory have also been proposed [44]. Hardware optimizations are important but complementary to HawkEye.

Software Support: FreeBSD's huge page management is largely influenced by previous work on superpages [57]. Carrefour-LP [47] showed that ad-hoc usage of huge pages can degrade performance in NUMA systems due to additional remote memory accesses and traffic imbalance across node interconnects, and proposed hardware-profiling based techniques to overcome these problems. Anti-fragmentation techniques for clustering pages based on mobility type of pages proposed by Gorman et al. [48] have been adopted by Linux to support the allocation of huge pages in long-running systems. Illuminator [59] highlighted critical drawbacks of Linux's fragmentation mitigation techniques and their implications on performance and latency, and proposed techniques to solve these problems. Mechanisms of dealing with physical memory fragmentation and NUMA systems are important but orthogonal problems, and their proposed solutions can be adopted to improve the robustness of HawkEye. Compiler or application hints can also be used to assist OSs in prioritizing huge page mapping for certain parts of the address space [27, 56], for which an interface is already provided by Linux through the `madvise` system call [16].

An alternative approach for supporting huge pages has also been explored via `libhugetlbfs` [22] where the user is provided more control on huge page allocation. However, such an approach requires manual intervention for reserving huge pages in advance and considers each application in isolation. Windows and OS X support huge pages only through this reservation-based approach to avoid issues associated with transparent huge page management [17, 55, 59]. We believe that insights from HawkEye can be leveraged to improve huge page support in these important systems.

6 Conclusions

To summarize, transparent huge page management is essential but complex. Effective and efficient algorithms are required in the OS to automatically balance the tradeoffs and provide performant and stable system behavior. We expose some important subtleties related to huge page management in existing proposals, and propose a new set of algorithms to address them in HawkEye.

References

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

[2] 2000. Linux: Page Zeroing Strategy. https://yarchive.net/comp/linux/page_zeroing_strategy.html.

[3] 2007. Memory part 5: What programmers can do. https://lwn.net/Articles/255364/.

[4] 2010. Mysteries of Windows Memory Management Revealed: Part Two. https://blogs.msdn.microsoft.com/tims/2010/10/29/pdc10-mysteries-of-windows-memory-management-revealed-part-two/.

[5] 2012. khugepaged eating 100% CPU. https://bugzilla.redhat.com/show_bug.cgi?id=879801.

[6] 2012. Recommendation to disable huge pages for Hadoop. https://developer.amd.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf.

[7] 2014. Arch Linux becomes unresponsive from khugepaged. http://unix.stackexchange.com/questions/161858/arch-linux-becomes-unresponsive-from-khugepaged.

[8] 2014. CORAL Benchmark Codes. https://asc.llnl.gov/CORAL-benchmarks/#hacc.

[9] 2014. Recommendation to disable huge pages for NuoDB. http://www.nuodb.com/techblog/linux-transparent-huge-pages-jemalloc-and-nuodb.

[10] 2014. Why TokuDB Hates Transparent HugePages. https://www.percona.com/blog/2014/07/23/why-tokudb-hates-transparent-hugepages/.

[11] 2015. The Black Magic Of Systematically Reducing Linux OS Jitter. http://highscalability.com/blog/2015/4/8/the-black-magic-of-systematically-reducing-linux-os-jitter.html.

[12] 2015. Tales from the Field: Taming Transparent Huge Pages on Linux. https://www.perforce.com/blog/151016/tales-field-taming-transparent-huge-pages-linux.

[13] 2016. C++ associative containers. https://github.com/sparsehash/sparsehash.

[14] 2016. Remove PG_ZERO and zeroidle (page-zeroing) entirely. https://news.ycombinator.com/item?id=12227874.

[15] 2017. Hugepages. https://wiki.debian.org/Hugepages.

[16] 2017. MADVISE: Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/madvise.2.html.

[17] 2018. Large-Page Support in Windows. https://docs.microsoft.com/en-gb/windows/desktop/Memory/large-page-support.

[18] 2019. CGROUPS: Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[19] 2019. FreeBSD: Pre-Faulting and Zeroing Optimizations. https://www.freebsd.org/doc/en_US.ISO8859-1/articles/vm-design/prefault-optimizations.html.

[20] 2019. Intel Haswell. http://www.7-cpu.com/cpu/Haswell.html.

[21] 2019. Intel Skylake. http://www.7-cpu.com/cpu/Skylake.html.

[22] 2019. Libhugetlbfs: Linux man page. https://linux.die.net/man/7/libhugetlbfs.

[23] 2019. Recommendation to disable huge pages for MongoDB. https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/.

[24] 2019. Recommendation to disable huge pages for Redis. http://redis.io/topics/latency.

[25] 2019. Recommendation to disable huge pages for VoltDB. https://docs.voltdb.com/AdminGuide/adminmemmgt.php.

[26] 2019. Transparent Hugepage Support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[27] Mohammad Agbarya, Idan Yaniv, and Dan Tsafrir. 2018. Memomania: From Huge to Huge-Huge Pages. In *Proceedings of the 11th ACM International Systems and Storage Conference (SYSTOR '18)*. ACM, New York, NY, USA, 112–112. https://doi.org/10.1145/3211890.3211918

[28] 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

[29] Nadav Amit, Dan Tsafrir, and Assaf Schuster. 2014. VSwapper: A Memory Swapper for Virtualized Environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 349–366. https://doi.org/10.1145/2541940.2541969

[30] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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

[36] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[37] Daniel Bovet and Marco Cesati. 2005. *Understanding The Linux Kernel*. Oreilly & Associates Inc.

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

[39] Jonathan Corbet. 2010. Memory compaction. https://lwn.net/Articles/368869/.

[40] 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

[41] Cort Dougan, Paul Mackerras, and Victor Yodaiken. 1999. Optimizing the Idle Task and Other MMU Tricks. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, Berkeley, CA, USA, 229–237. http://dl.acm.org/citation.cfm?id=296806.296833

[42] Niall Douglas. 2011. User Mode Memory Page Allocation: A Silver Bullet For Memory Allocation? *CoRR* abs/1105.1811 (2011). arXiv:1105.1811 http://arxiv.org/abs/1105.1811

[43] Ulrich Drepper. 2007. What Every Programmer Should Know About Memory.

[44] 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.

[45] M. Franklin, D. Yeung, n. Xue Wu, A. Jaleel, K. Albayraktaroglu, B. Jacob, and n. Chau-Wen Tseng. 2005. BioBench: A Benchmark Suite of Bioinformatics Applications. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.(ISPASS)*, Vol. 00. 2–9. https://doi.org/10.1109/ISPASS.2005.1430554

[46] 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, Washington, DC, USA, 178–189. https://doi.org/10.1109/MICRO.2014.37

[47] 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

[48] Mel Gorman and Patrick Healy. 2008. Supporting Superpage Allocation Without Additional Hardware Support. In *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/1375634.1375641

[49] 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_2$4

[50] Mel Gorman and Andy Whitcroft. 2006. The What, The Why and the Where To of Anti-Fragmentation. In *Linux Symposium*. 141.

[51] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 2015. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*. ACM, New York, NY, USA, 39–51. https://doi.org/10.1145/2731186.2731187

[52] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John C. S. Lui. 2017. SmartMD: A High Performance Deduplication Engine with Mixed Pages. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 733–744. http://dl.acm.org/citation.cfm?id=3154690.3154759

[53] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[54] Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrián Cristal, and Michael M. Swift. 2014. Performance analysis of the memory management unit under scale-out workloads. *2014 IEEE International Symposium on Workload Characterization (IISWC)* (2014), 1–12.

[55] 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, Berkeley, CA, USA, 705–721. http://dl.acm.org/citation.cfm?id=3026877.3026931

[56] 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

[57] 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

[58] Ashish Panwar, Naman Patel, and K. Gopinath. 2016. A Case for Protecting Huge Pages from the Kernel. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '16)*. ACM, New York, NY, USA, Article 15, 8 pages. https://doi.org/10.1145/2967360.2967371

[59] 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

[60] 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.

[61] 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

[62] 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

[63] 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

[64] Indira Subramanian, Clifford Mather, Kurt Peterson, and Balakrishna Raghunath. 1998. Implementation of Multiple Pagesize Support in HP-UX.. In *USENIX Annual Technical Conference*. 105–119.

[65] 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.

[66] Koji Ueno and Toyotaro Suzumura. 2012. Highly Scalable Graph Search for the Graph500 Benchmark. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '12)*. ACM, New York, NY, USA, 149–160. https://doi.org/10.1145/2287076.2287104

[67] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 181–194. https://doi.org/10.1145/844128.844146

[68] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards Practical Page Coloring-based Multicore Cache Management. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM, New York, NY, USA, 89–102. https://doi.org/10.1145/1519065.1519076