HawkEye: Efficient Fine-grained OS Support for Huge Pages
ASPLOS Submission #449– Confidential Draft – Do Not Distribute!

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 [56] uncovered some interesting pitfalls associated with current huge page management strategies. Our paper exposes more nuances involved in OS-based huge page management and proposes a new set of algorithms to address them in a principled manner. We have implemented our algorithms in a system called HawkEye inside the Linux kernel, and show that HawkEye is more performant, robust and fairer when compared with current state-of-the-art systems.

1. Introduction

Modern applications with large memory footprints have put address translation overheads in general-purpose processors into focus [33, 50, 60, 62, 64]. Modern architectures implementing large multi-level TLBs and page-walk caches, all supporting multiple page sizes [36, 41], entails more decision making by the OS designers who need efficient policies to automatically choose suitable page sizes for different workloads [48, 56, 58, 65]. The problem becomes more severe with virtual machines where two layers of address translation cause additional MMU overheads [35, 47, 63].

Despite promising hardware support available across architectures [8], huge pages have been a source of unsatisfactory performance with important applications [18, 19, 20, 21, 22, 24, 26]. These performance issues are often due to inadequate OS-based huge page management algorithms [1, 2, 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 Youngjin et al., i.e., Ingens [56].

Linux: Linux employs huge pages through two mechanisms: (1) either it allocates a huge page at the time of page fault if memory contiguity is available, or (2) it promotes base pages to huge pages, by optionally compacting memory [40], 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 [58]. 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 bloat and contiguity more efficiently than Linux, at the potential expense of a higher number of page faults and higher address translation overheads due to multiple TLB entries, one per baseline page.

Ingens: In the Ingens paper [56], 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, by switching from an aggressive Linux-like policy at low memory pressure to a utilization-based FreeBSD-like strategy at high memory pressure to try and achieve the best of both worlds. (2) To avoid high page fault latency associated with synchronous page-zeroing, Ingens avoids allocating huge pages in the page fault handler; instead it always defers huge page promotion to an asynchronous background 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 the current systems (prior to Ingens) are dealing with huge page management issues through “spot fixes”, and motivates the need for a principled strategy. While the Ingens strategy is more sophisticated than previous works, we show that even Ingens behaviour is less than desirable and fails to meet the expectations of diverse modern workloads. Also, while Ingens offers some flexibility in managing the tradeoffs at runtime, it delegates critical decision-making to system users through configuration parameters. We show that system behavior in Ingens is often highly sensitive to these parameters, which is something that is not desirable of a robust stable system.

We present HawkEye, an automated OS-level solution for huge page management that effectively manages the tradeoffs between page fault latency, application performance, memory bloat and fairness. HawkEye is significantly different in its design from prior work and we empirically demonstrate that it outperforms current state-of-the-art. HawkEye design is based
on sound principles and its algorithms automatically adapt to workload changes, while being relatively insensitive to configuration parameters. Our evaluation involves several workloads with a diverse set of requirements vis-a-vis huge page management, and convincingly demonstrates the improvements achieved by HawkEye over previous work.

2. Motivation: Tradeoffs and Associated Pitfalls

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 solves the problems associated with previous approaches. HawkEye is discussed in detail in §3.

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 aggressive 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 uses Linux-like aggressive promotion policy to minimize MMU overheads when memory fragmentation is low. To measure fragmentation, it uses the Free Memory Fragmentation Index (FMFI [51]): 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 [39] that is running on a machine with 48GB memory (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 a total 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 allocation and subsequent deallocation, and leave the address-space fragmented. Ideally, a good huge page manager should recover elegantly from memory bloat to avoid disruptions under high memory pressure situations.

In phase P2, when Redis releases pages back to the OS through madvise system call [15], 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 mimics Linux’s aggressive behavior 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 tuning the configuration parameters appropriately and using only conservative promotion. However, as also explained in the Ingens paper, this strategy risks losing performance due to high MMU overheads under no (or low) memory pressure.

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, 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

![Figure 1: Resident Set Size (RSS) of Redis server across 3 phases: P1 (insert), P2(delete) and P3(insert).](image-url)
recently faulted regions to mimic Linux’s aggressive behavior. 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 [25]. 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 this through a microbenchmark that allocates a 10GB buffer, touching one byte in every base page and later frees the buffer. This workload resembles allocation and access patterns of memory-intensive applications that allocate, initialize, use, and discard large memory regions. Table 1 shows the performance summary for 10 runs of this workload.

Linux with huge pages (Linux-2MB) reduces the number of page faults by more than 500 \times over Linux with base pages (Linux-4KB) and leads to more than 4 \times performance improvement for this workload, despite being 133 \times worse on average page fault latency. Ingens reduces latency 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, Ingens degrades the overall performance due to excessive page faults for this workload. HawkEye, on the other hand, is able to achieve both low page fault latency and fewer page faults resulting in higher overall performance over all existing approaches.

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux 4KB</td>
<td>Linux 2MB</td>
</tr>
<tr>
<td># Page faults</td>
<td>26.2M</td>
</tr>
<tr>
<td>Total page fault time (secs)</td>
<td>92.6</td>
</tr>
<tr>
<td>Avg page fault time (µs)</td>
<td>3.5</td>
</tr>
<tr>
<td>System time (secs)</td>
<td>102</td>
</tr>
<tr>
<td>Total time (secs)</td>
<td>106</td>
</tr>
</tbody>
</table>

Table 1: Page faults, allocation latency and performance for a microbenchmark with \( \approx 100\) GB memory allocation.

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 [25]. 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 this through a microbenchmark that allocates a 10GB buffer, touching one byte in every base page and later frees the buffer. This workload resembles allocation and access patterns of memory-intensive applications that allocate, initialize, use, and discard large memory regions. Table 1 shows the performance summary for 10 runs of this workload.

Linux with huge pages (Linux-2MB) reduces the number of page faults by more than 500 \times over Linux with base pages (Linux-4KB) and leads to more than 4 \times performance improvement for this workload, despite being 133 \times worse on average page fault latency. Ingens reduces latency 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, Ingens degrades the overall performance due to excessive page faults for this workload. HawkEye, on the other hand, is able to achieve both low page fault latency and fewer page faults resulting in higher overall performance over all existing approaches.

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux 4KB</td>
<td>Linux 2MB</td>
</tr>
<tr>
<td># Page faults</td>
<td>26.2M</td>
</tr>
<tr>
<td>Total page fault time (secs)</td>
<td>92.6</td>
</tr>
<tr>
<td>Avg page fault time (µs)</td>
<td>3.5</td>
</tr>
<tr>
<td>System time (secs)</td>
<td>102</td>
</tr>
<tr>
<td>Total time (secs)</td>
<td>106</td>
</tr>
</tbody>
</table>

Table 1: Page faults, allocation latency and performance for a microbenchmark with \( \approx 100\) GB memory allocation.

We argue that considering huge pages (or memory contiguity) as a contended resource is misleading — 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 the Ingens definition of fairness would treat them equally.

We posit that instead of considering “memory contiguity as a resource” and trying to grant it equally among processes, it is more appropriate 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. This definition of fairness is philosophically different from Ingens. Distributing MMU overheads proportionally across processes appears closer to the spirit of true virtualization. We show that this approach also happens to usually yield the best performance by helping the most afflicted processes first.

Further, in Ingens, huge pages get allocated to applications at the time of the first page fault 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.

Further, within a process, Ingens promotes 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) [57]: 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

2.3. What is fair behaviour?

Under memory fragmentation, the OS must allocate huge pages among multiple processes fairly. We find that there is a fundamental ambiguity in the definition of fairness in this setting. Ingens authors define fairness through a proportional huge page promotion metric wherein they consider “memory contiguity as a resource” and try and be equitable in distributing contiguous memory regions (through huge pages) among applications. Further, 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 access bit present in the page table entries which is maintained by the hardware and periodically cleared by the OS. Ingens, during the computation of the proportional huge page promotion metric, employs an idleness penalty fac-
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 2 demonstrates this with the NPB benchmark suite [31]: 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).

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. We instead propose directly measuring TLB overheads through hardware performance counters when available (see Table 3). Because hardware counters may not be available for all platforms, we demonstrate two versions of our algorithm: one that directly measures MMU overheads through hardware counters, and the other estimates these overheads through access patterns. We compare both approaches in our experiments.

### 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 the pages in a background thread; (2) memory bloat can be tackled by identifying and de-duplicating the zero-filled baseline pages present within an allocated huge page; (3) promotion decisions should be based on finer-grained access tracking of huge page sized regions, and should include recency, frequency, 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. We have implemented our algorithms inside HawkEye/Linux; there are two variants of our algorithm/implementation, 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 of a process (HawkEye-G).

#### 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 asympre-zeroing. Async pre-zeroing is a rather old idea and had been discussed extensively among kernel developers in early 2000s [5, 14, 23, 42]. Linux developers opined that async pre-zeroing is not an overall performance win for two main reasons. We argue that it is time to revisit these opinions.

First, the async pre-zeroing thread might interfere with primary workloads, especially on cache space [5]. 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 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 [44]. This optimization eliminates 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 [5, 43]. 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 [8, 28], 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

---

1Windows and FreeBSD implement async pre-zeroing in a limited fashion [7, 17]. However, their huge page policies do not allow high latency allocations. This idea is more relevant for Linux due to its aggressive huge page allocation behaviour.
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 [55], 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 aggressively; 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 [38]. 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 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 definition of fairness.

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 interplay in counter-productive ways [53]. 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 sharable 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.

### 3.2. 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 [55], 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 aggressively; 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 [38]. 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 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 definition of fairness.

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 interplay in counter-productive ways [53]. 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 sharable 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. Promotion based on fine-grained access-bit tracking

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.

![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.](image)
Access-coverage: It 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 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. This way, promotion of cold regions is automatically deferred to prioritize hot regions.

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

3.4. Fairness

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 definition of fairness, as pages are promoted first from regions/processes with high expected TLB pressure (and consequently high expected MMU overheads).

In HawkEye-G, regions from the highest access-coverage index across all processes are promoted first. 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. Discussion

We now briefly outline a few remaining issues related to huge page management that are out of scope for this paper.

1) Adjusting thresholds: 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. This strategy can be too aggressive if memory pressure is consistently fluctuating. An ideal solution should adjust these thresholds dynamically to prevent aggressive kernel behavior. The approach proposed by Guo et. al. [53] in the context of memory deduplication in virtualized environments can be beneficial in this context.

2) 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. Our preliminary investigations show that our approach based on estimated MMU overheads is reasonable even if workloads differ by more than an order of magnitude in memory size. However, if limiting huge pages is still required, perhaps it is best to handle it through resource limiting/monitoring tools, such as Linux’s cgroups [4].

3) Other algorithms: For space reasons, we do not discuss some parts of the management algorithms, such as rate limiting of promotion overheads, demotion based on low utilization to enable better sharing through same-page merging, and compaction algorithms. Much of this material has been discussed...
and evaluated in detail in prior work (Ingens [56]) and we do not contribute 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 on which 96GB memory and 48 cores (with hyperthreading enabled) running at 2.3GHz are equally divided on two sockets. The L1 TLB contains 64 and 8 entries for 4KB and 2MB pages resp. while the shared 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 over a wide range of workloads ranging from HPC, graph algorithms, in-memory databases, genomics and machine learning [21, 27, 31, 34, 37, 46, 54, 66]. HawkEye is implemented in Linux kernel v4.3 (same as Ingens).

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 in bare-metal and virtualized environments; (b) the bloat-vs-performance tradeoff; (c) the impact of low latency page faults; (d) the negligible cache interference caused by asynchronous pre-zeroing thread due to non-temporal memory stores; (e) and the memory efficiency enabled by asynchronous page pre-zeroing.

Performance advantages of fine-grained access-coverage based 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 low address translation overheads through a judicious promotion to huge pages. Recall that while previous approaches would promote pages in VA order (from low VAs to high), we promote based on access-coverage. To fragment memory initially, we create a helper workload that allocates several 4KB pages (to create memory fragmentation) before freeing them (to restore memory to the unfragmented state); our test workloads are started in the fragmented state and we measure the time taken for the system to recover from high address translation overheads in this scenario. 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 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 almost all their required memory in the beginning (in the fragmented state of the system). HawkEye’s fine-grained huge page promotion improves performance by 13%, 12% and 6% for these workloads over Linux and Ingens. Evidently, HawkEye’s promotion strategy is much more effective as it identifies the most profitable promotions first.

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 observe 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, HawkEye takes ∼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 shows that HawkEye perform 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 throughout the VA space, HawkEye delivers performance similar to Linux and Ingens.

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

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.

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 4 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 around 20 minutes. In contrast, Ingens promotes huge pages from lower VAs in each instance of Graph500. Thus it takes even longer for Ingens before it can promote the hot regions of the respective processes, leading to 9% performance degradation over Linux on average 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 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 workloads are 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. Both Ingens and HawkEye are agnostic to process creation order.

The impact of execution order is clearly visible for Linux. In the (Before) case, Linux promotes huge pages in TLB sensitive applications and improves their performance. 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 (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 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 5 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 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.

**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, 4KB) sized key-value pairs and then delete 60% randomly selected keys (see Table 6). While Linux-4KB is memory efficient (no bloat), its performance is low (7% lower throughput than Linux-2MB). Linux-2MB delivers high performance but consumes more memory which remains unavailable to the rest of system even when memory
Table 6: Redis memory consumption and throughput.

<table>
<thead>
<tr>
<th>Workload</th>
<th>OS Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linux 4KB</td>
</tr>
<tr>
<td>Redis (45GB)</td>
<td></td>
</tr>
<tr>
<td>SparseHash (36GB)</td>
<td>30.1</td>
</tr>
<tr>
<td>HACC-IO (65GB)</td>
<td>6.5</td>
</tr>
<tr>
<td>JVM Spin-up (36GB)</td>
<td>37.7</td>
</tr>
<tr>
<td>KVM Spin-up (36GB)</td>
<td>40.6</td>
</tr>
</tbody>
</table>

Table 7: Performance implications of asynchronous page zeroing. Numbers for Redis represent throughput (higher is better). All other numbers represent time in seconds (lower is better).

Fast page faults: Table 7 shows the performance of different strategies with multiple workloads whose performance depend on the efficiency of the OS page fault handler. We measure Redis throughput when 2MB-sized values are inserted. SparseHash [3] is a hash-map library in C++ while HACC-IO [6] is a parallel IO benchmark used with an in-memory file system. We also measure the spin-up time of virtual machines: KVM and a Java Virtual Machine (JVM) where both are configured to allocate all memory during initialization.

Performance benefits of asynchronous page-zeroing with base pages (HawkEye-4KB) are modest: in this case, the other page fault overheads (apart from zeroing) dominate performance. However, page-zeroing cost becomes significant with huge pages. Consequently, HawkEye-2MB improves the performance of Redis and SparseHash by 1.26× and 1.62× resp. over Linux-2MB. 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-2MB. Notice that the reduction is only 1.34× and 1.26× with base pages. All these workloads have high spatial locality of page faults and hence Ingens strategy of utilization-based promotion degrades performance over Linux due to a higher number of page faults.

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 in this experiment). This experiment shows the highly-exaggerated behavior of async pre-zeroing. 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 and negligible in practice.

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 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. For PageRank, performance degrades slightly due to a higher number of COW page faults due to unnecessary same-page merging. HawkEye also reduces the mean latency by 45% for Redis, and 28% for MongoDB and Memcached compared to stock Linux. Tail latency reductions are even higher at 77%, 35% and 73% reduction for 99th percentile latency for...
While our experiments show early promise in this direction, with high access-coverage but different MMU overheads, we leave an extensive evaluation for future work.

Comparison between HawkEye-PMU and HawkEye-G. Table 8:

<table>
<thead>
<tr>
<th>Workload</th>
<th>MMU Overhead</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4KB HawkEye-PMU</td>
<td>HawkEye-G</td>
</tr>
<tr>
<td>random(4GB)</td>
<td>60%</td>
<td>582</td>
</tr>
<tr>
<td>sequential(4GB)</td>
<td>&lt; 1%</td>
<td>517</td>
</tr>
<tr>
<td>Total</td>
<td>1099</td>
<td>863(1.27×)</td>
</tr>
<tr>
<td>cg.D(16GB)</td>
<td>39%</td>
<td>1952</td>
</tr>
<tr>
<td>mg.D(24GB)</td>
<td>&lt; 1%</td>
<td>1363</td>
</tr>
<tr>
<td>Total</td>
<td>3315</td>
<td>2566(1.39×)</td>
</tr>
</tbody>
</table>

Table 8: Comparison between HawkEye-PMU and HawkEye-G. Parenthesis represent memory size and speedup over base pages.

Redis, MongoDB and Memcached respectively. 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 over-committed systems. It is well-known that balloon-drivers and other para-virtual interfaces are riddled with compatibility problems [30], 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.

Hawk-PMU vs. HawkEye-G: Even though HawkEye-G uses 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 8 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. 4KB 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 patterns. In this case, mg.D and sequential have negligible MMU overheads despite having high access-coverage in all huge page regions. While HawkEye-PMU correctly identifies the process with higher MMU overheads, HawkEye-G treats both workloads in each set identically, due to imprecise estimations. Consequently, HawkEye-PMU performs up to 36% better than HawkEye-G. Bridging the gap between the two approaches in a hardware independent manner is 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 [29, 35, 50, 57, 59, 63, 64, 65].

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 [8, 9, 10, 41]. Further, modern processors employ page-walk-caches to avoid memory lookups in the TLB-miss path [32, 36]. POM-TLB services a TLB miss using a single memory lookup and further leverages regular data caches to speed up address translation [64]. Direct segments [33, 47] 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 (coalesced-large-reach TLBs [62]) 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 [36, 41, 61]. Techniques to support huge pages in non-contiguous memory have also been proposed [45]. Hardware optimizations are important but complementary to HawkEye.

Software Support: Different ways of supporting huge pages and associated challenges have been explored in OS designs. FreeBSD’s huge page management is largely influenced by [58]. Carrefour-HP [48] showed that ad-hoc usage of huge pages can degrade performance in NUMA systems due to increased 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. [49] have been adopted by Linux to support the allocation of huge pages in long-running systems. Illuminator [60] 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 driven hints can also be used to assist the OS in prioritizing huge page mapping for certain parts of the address space [28, 57], for which an interface is already provided by Linux through the madvise system call [15].

An alternative approach for supporting huge pages has also been explored via libhugetlbfs [13] 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 using only this reservation-based approach to avoid issues associated with transparent huge page support [12, 56, 60]. We believe 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 behaviour. 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.


