# EMD: Fair and Efficient Dynamic Memory De-bloating of Transparent Huge Pages

Parth Gangar
Fujitsu Research
Bengaluru, India

Ashish Panwar
Microsoft Research
Bengaluru, India

K. Gopinath
Rishihood University
Sonipat, India

## Abstract

Recent processors rely on huge pages to reduce the cost of virtual-to-physical address translation. However, huge pages are notorious for creating memory bloat – a phenomenon wherein the OS ends up allocating more physical memory to an application than its actual requirement. This extra memory can be reclaimed by the OS via de-bloating at runtime. However, we find that current OS-level solutions either lack support for dynamic memory de-bloating, or suffer from performance and fairness pathologies while de-bloating.

We address these issues with EMD (Efficient Memory De-bloating). The key insight in EMD is that different regions in an application's address space exhibit different amounts of memory bloat. Consequently, the tradeoff between memory efficiency and performance varies significantly within a given application e.g., we find that memory bloat is typically concentrated in specific regions, and de-bloating them leads to minimal performance impact. Hinged on this insight, EMD employs a prioritization scheme for fine-grained, efficient, and fair reclamation of memory bloat. EMD improves performance by up to 69% compared to HawkEye — a state-of-the-art OS-based huge page management system. EMD also eliminates fairness concerns associated with dynamic memory de-bloating.

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

***Keywords:*** Virtual memory; huge pages; memory bloat

**ACM Reference Format:**
Parth Gangar, Ashish Panwar, and K. Gopinath. 2025. EMD: Fair and Efficient Dynamic Memory De-bloating of Transparent Huge Pages. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management (ISMM '25), June 17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3735950.3735952



*ISMM '25, Seoul, Republic of Korea*

ACM ISBN 979-8-4007-1610-2/25/06
https://doi.org/10.1145/3735950.3735952

## 1 Introduction

The amount of data being generated and consumed is increasing exponentially [41]. Therefore, modern applications depend on accessing vast amounts of data from the physical memory [29]. In particular, for each memory access, virtual memory requires the processor to translate an (application-visible) virtual address into a (system-visible) physical address. Since this virtual-to-physical address translation lies in the critical path of execution, it can cause a significant slowdown. For example, Google recently reported that many of their data-center workloads spend 20-30% of total CPU cycles in address translation alone [24].

To accelerate this virtual-to-physical address translation, processors cache recently used addresses in multi-level hardware structures called Translation Lookaside Buffers (TLBs). A hit in the TLB is fast but processing of TLB misses is very expensive (as demonstrated by prior works [8, 10, 26, 31, 32, 35, 39, 42]), accounting for 10–40% of total memory access time [7, 25, 30, 37, 40]. Unfortunately, while the application's memory footprint is increasing at an unprecedented rate, it is not possible to scale up TLB sizes in the same proportion due to fundamental hardware limitations [12, 38].

Modern processors employ huge pages to alleviate the cost of virtual-to-physical address translation [20, 26, 28, 31, 32, 38, 44]. A huge page is a larger memory allocation unit, typically 2MB or 1GB, compared to the standard 4KB base page. Huge pages enable a single TLB entry to map significantly larger memory regions. For instance, a 2MB huge page maps 512 times more memory than a 4KB base page, thereby increasing the TLB's effective reach. Additionally, huge pages reduce the depth of page table traversal. On x86-64 systems, resolving a memory access for a 2MB huge page typically requires only three levels of page table walks instead of four, significantly lowering TLB miss costs. By reducing the frequency and cost of TLB misses, huge pages can boost application performance substantially [26, 31, 32]. Workloads such as large-scale machine learning models, HPC simulations, and in-memory databases have demonstrated performance improvements of 20–50% when huge pages are effectively utilized [26, 38].

However, the efficient utilization of huge pages demands robust support from the OS, which remains inadequate in current systems. This is because while huge pages alleviate address translation overheads, they introduce several memory management challenges such as fragmentation, latency

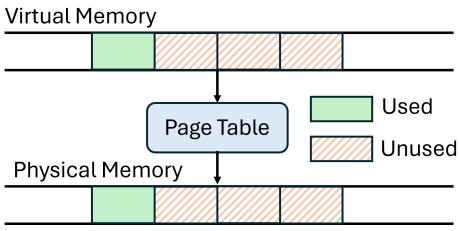



**Figure 1.** Illustration of memory bloat with huge pages assuming that a huge page consists of 4 base pages. If an application accesses only one of the base pages in a huge page region (left most block in green), the OS ends up allocating 4 base pages worth of physical memory resulting in 3 unused pages (rightmost boxes).

spikes, fairness concerns in multi-tenant environments, and memory bloat [2, 4, 15]. Troubleshooting these issues often leads to disabling huge pages, undermining their potential benefits. While prior work[26, 31] has tackled various aspects of huge page management, we find that no existing system has satisfactorily addressed the challenge of higher physical memory requirement of huge pages. Therefore, in this paper, we aim to tackle this issue, focusing on the interplay of memory efficiency and performance.

**Memory bloat:** We define memory bloat as the unnecessary physical memory mapped in an application's page tables due to the allocation of huge pages (see Figure 1). Memory bloat can significantly increase an application's total memory footprint, especially when memory access patterns are sparse or when internal fragmentation leaves portions of huge pages unused. Consequently, huge pages can increase the cost of deployment, lead to out-of-memory errors or cause unnecessary swapping.

In this paper, our objective is to mitigate memory bloat in a way that allows an application to benefit from huge pages without running into (avoidable) out-of-memory situations. In particular, our goal is to make sure that an application enjoys the performance benefits of huge pages as long as enough physical memory is available. However, when physical memory is scarce, the OS should be able to reclaim unused physical memory from huge pages with minimal impact on performance. Therefore, we focus on managing the tradeoff between memory bloat and performance at runtime. Towards this goal, we make the following major contributions:

We first present a broad characterization of existing systems into three categories based on how they balance huge page trade-offs. Some systems use coarse-grained measures relying on end-users to enable or disable huge pages at the system level - transparent huge pages in Linux is an example of this category. The second category of systems allow users to configure a threshold to limit the maximum amount of permissible memory bloat – Ingens [26] and FreeBSD [27] represent this category. The third category represents systems that deal with memory bloat dynamically e.g., HawkEye [31]. Overall, HawkEye is the only reported solution capable of managing memory bloat depending on the availability of physical memory at runtime.

We then show that while HawkEye enables dynamic memory de-bloating, its strategy is suboptimal and unfair. Specifically, we find that HawkEye performs de-bloating sequentially with an application's address space, without considering the distribution of memory bloat across different regions. This is suboptimal from a performance standpoint because it can break more huge pages than required. We also show that HawkEye's de-bloating strategy can be unfair wherein it penalizes some applications more than others.

Finally, we present EMD (Efficient Dynamic Memory Debloating), a new approach that enables fair and efficient dynamic memory de-bloating. EMD is based on our key insight that different regions in an application's address space exhibit different amounts of memory bloat. Consequently, the tradeoff between memory efficiency and performance varies significantly even within a given application e.g., we find that memory bloat is typically concentrated in certain regions of an application address space, and de-bloating such regions first can free up physical memory with little impact on performance. Based on this insight, EMD employs a prioritization scheme for fine-grained, efficient, and fair dynamic memory de-bloating. We implement EMD in the Linux kernel and demonstrate that it improves performance by up to 69% compared to HawkEye. Additionally, EMD ensures fairness in de-bloating when multiple applications are running concurrently.

## 2 Background

### 2.1 Huge Pages and Associated Challenges

**2.1.1 Huge Pages.** Huge page is a hardware-software codesign approach to reduce the overhead of address translation. They are used to reduce the number of TLB misses and make page tables shorter, improving performance [26, 31, 38, 44]. Huge pages are commonly deployed via transparent huge pages (commonly referred to as THP) [6]. In this method, the OS allocates huge pages transparently to user applications, as and when possible. This approach is flexible, does not require user involvement and hence is the most preferred way of deploying huge pages in real-world systems. Therefore, we focus purely on transparent huge pages in this paper.

**2.1.2 Challenges with Transparent Huge Pages.** Transparent huge pages have led to severe performance issues in several cases e.g., latency spikes, high CPU usage, increased memory pressure and fairness, etc [3, 5, 15]. These issues primarily stem from two fundamental requirements. First, a huge page must be mapped in contiguous physical memory. However, memory contiguity is difficult for an OS to maintain in a long-running systems due to fragmentation.

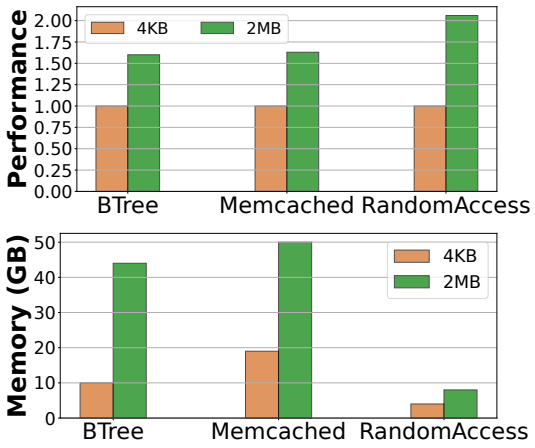



**Figure 2.** Tradeoff between performance and memory utilization. Huge pages improve performance but at the expense of more memory compared to base pages.

Consequently, efforts to de-fragment physical memory lead to spikes in latency and CPU usage and the lack of contiguity creates fairness issues [16, 26]. Secondly, a physical memory page must be cleared (zero-filled) before it can be allocated to an application. Since huge pages are larger, clearing them also takes longer which also increases the risk of latency spikes [31, 44]. Recent solutions have attacked several of these challenges. However, we find that the issue of increased memory pressure due to huge pages has received less attention.

#### 2.1.3 Tradeoff between Performance and Memory Efficiency.

Huge pages can lead to increased memory pressure (commonly known as bloat) in an application because they allocate larger chunks of physical memory at once, typically 2MB or more, compared to the standard 4KB pages. When a page fault occurs, the OS must allocate an entire page, even if the application only needs a small portion of it. The untouched portions of a page represents wasted memory. Since huge pages are larger, the memory waste can also be proportionally higher especially for workloads with sparse or unpredictable memory access patterns. As a result, applications may experience increased memory footprint, reduced availability of free memory, and potential performance degradation due to excessive memory consumption.

Therefore, huge pages lead to a complex tradeoff between memory usage and performance: *huge pages can improve performance but at the potential expense of additional physical memory usage*. Figure 2 highlights this tradeoff with three representative applications: BTree, Memcached and RandomAccess. For these applications, huge pages improve performance by as much 1.6× but at the cost of up to 4.3× higher memory footprint. Ideally, the OS should be able to navigate this tradeoff automatically (i.e., without user involvement) and at runtime (i.e., depending on the availability of physical memory). For example, if enough free memory is available, the OS should allocate huge pages aggressively to maximize performance. However, when memory is limited, it must prioritize eliminating memory waste to avoid unintended side-effects e.g., out-of-memory errors.

### 2.2 Bloat Management in Current Systems

Based on our study of prior work, we divide current systems into three categories based on how they deal with the issue of memory bloat due to huge pages:

#### 2.2.1 Category-1 (No Control Over Bloat).

These systems enable or disable the use of huge pages for the whole system i.e., either all the applications are eligible to use huge pages or none of them are (e.g., Linux THP) [6]. Intuitively, these coarse-grained strategies fail to adapt to the properties of individual applications and the availability of physical memory at runtime. For example, if memory requirement of two co-running applications exceed physical memory capacity due to memory bloat, then the user is forced to disable huge pages for both applications.

#### 2.2.2 Category-2 (Coarse-Grained, Static).

These systems employ more fine-grained strategies than Linux THP. For example, FreeBSD and Ingens let a user configure a threshold to limit the maximum amount of permissible memory bloat by letting them decide when a virtual memory range can be backed by a huge page [26, 44]. For example, suppose a user wants to limit memory bloat to a maximum of 10%. In that case, she can configure the threshold such that a 2MB virtual memory range will be allocated a 2MB huge page only if at least 90% of the 512 base pages (4KB each) within that range have been accessed at least once by the application. While such a strategy is better than the coarse-grained system-wide policies of Linux, it is still static and suboptimal. More importantly, memory wasted due to bloat is permanently wasted from a system point of view i.e., the OS cannot reclaim the unused pages to satisfy other allocations if it runs out of memory.

#### 2.2.3 Category-3 (Fine-Grained, Dynamic).

The third category represents systems that deal with memory bloat dynamically e.g., HawkEye [31]. For example, HawkEye guarantees that if a page has been allocated by the system to an application but the application has not accessed it, then it is zero-filled. Based on this observation, physical memory pages wasted due to memory bloat can be reclaimed at runtime in HawkEye by de-bloating e.g., by breaking a huge page mapping into smaller page mappings and de-duplicating the constituent zero-filled base pages. For example, if a 2MB huge page contains 100 zero-filled 4KB pages and 412 in-use pages, then HawkEye maps the 100 pages into a single zero-filled 4KB page while mapping the other pages as individual 4KB page mappings in the process page tables. The mechanism of de-bloating is illustrated in Figure 3. The ability to

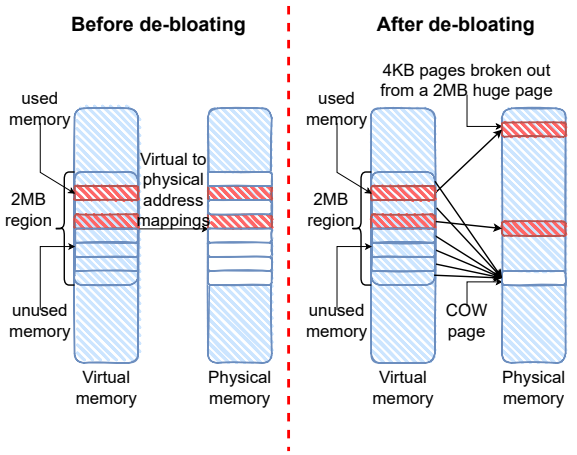



**Figure 3.** An example of de-bloating, which reduces memory bloat in the system by reclaiming unused memory (zero-filled base pages) from a huge page.

de-bloat enables one to trade performance and memory efficiency dynamically depending on the availability of physical memory at runtime.

## 3 Motivation

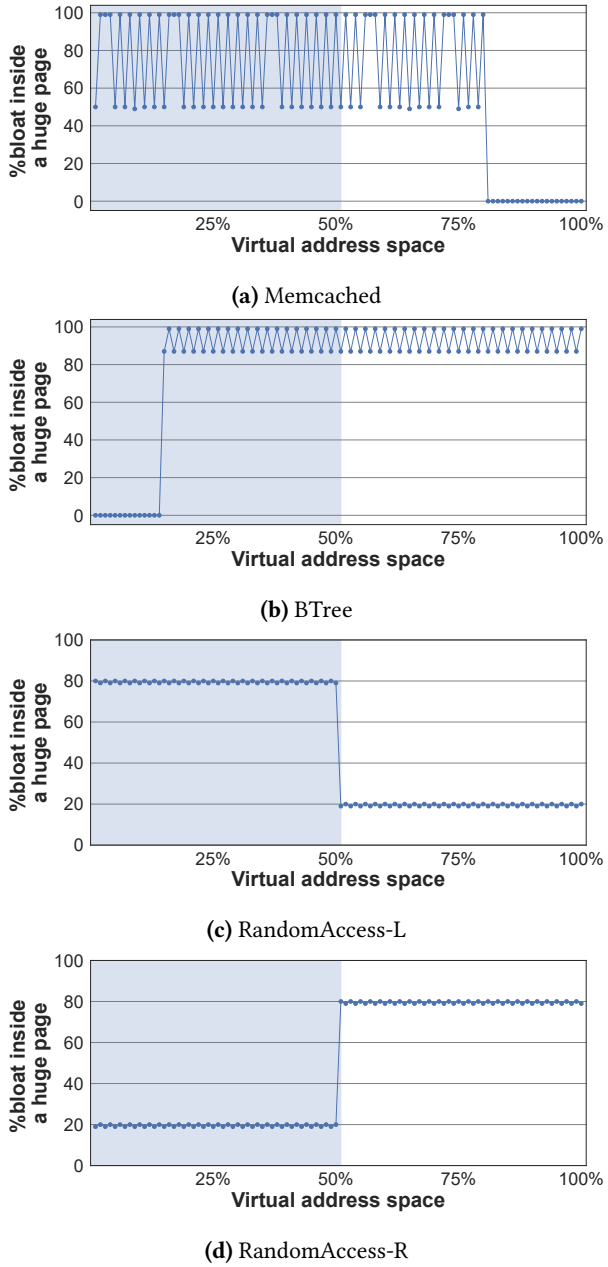
We find that among all the existing systems, HawkEye is the only one that provides support for dynamic memory de-bloating. However, we find that in doing so, it uses policies that can be inefficient or unfair. In this section, we elaborate on these shortcomings of HawkEye.

### 3.1 Inefficient De-bloating (Intra-process Scenario)

HawkEye de-bloats huge pages by sequentially scanning the address space of a process. In doing so, it examines the contents of base (e.g., 4KB) pages within each huge (e.g., 2MB) page. If a huge page contains one or more zero-filled base pages, HawkEye de-bloats it as discussed in §2.2. It then scans (and de-bloats, where applicable) the next set of huge pages in the same sequential order.

This approach works well when memory bloat is uniformly distributed across different huge pages. However, we find that *memory bloat is concentrated in specific memory regions* for some applications. For example, Figure 4(a) shows that Memcached has three kinds of huge pages: 1) nearly 100% memory bloat, 2) about 50% memory bloat and 3) no memory bloat (towards the end of the address space). Similarly, Btree also has three kinds of huge pages as shown in Figure 4(b). This happens particularly when some regions corresponding to different nodes in a tree are more populated than the others (BTree), or when different keys are clustered in specific patterns (Memcached).

The issue with the sequential scanning based method is that *it ignores the relative contribution of each huge page towards memory bloat.* In general, it makes sense to first de-bloat huge pages that constitute the maximum amount of memory bloat. Doing so would enable memory savings while breaking the minimum number of huge pages. However, due to ignoring the relative contribution of different huge pages, HawkEye may end up breaking huge pages with very little bloat. If this happens, huge pages are broken unnecessarily leading to a significant impact on performance.




**(a)** Memcached

**(b)** BTree

**(c)** RandomAccess-L

**(d)** RandomAccess-R

**Figure 4.** Distribution of bloat across the address space for BTree, Memcached, RandomAccess-L, and RandomAccess-R. Shaded regions represent the first half of the address space.

**Table 1.** Huge page broken and performance impact while de-bloating two benchmarks that are identical except for how memory bloat is distributed in their address space.

| | # Huge pages broken | % slowdown |
|---|---|---|
| RandomAccess-L | 611 | < 2% |
| RandomAccess-R | 2146 | 65% |

**Experimental Validation.** To quantitatively demonstrate this issue, we conducted a simple experiment with two variants of a simple micro-benchmark. The micro-benchmark allocates a large buffer of 8GB and accesses it randomly, leading to high TLB misses if only base pages are allocated. Therefore, using huge pages improves performance. We configure the access patterns of this benchmark such that there is a fixed pre-determined amount of memory bloat in each huge page. Using this property, we configure the first variant of the benchmark (called RandomAccess-L) such that each huge page in the first half of its address space has 80% memory bloat whereas huge pages in the second half of the address space have only 20% memory bloat each. In the second variant called RandomAccess-R, huge pages in the first half have 20% bloat whereas those in the second half have 80% bloat each. The bloat distribution of these two variants of the micro-benchmark is shown in Figure 4(c)-(d). With these two variants, we expose the inefficiency of HawkEye while reclaiming 2GB of unused physical memory from the benchmark.

We argue that an ideal system should break minimum number of huge pages to reclaim unused physical memory, irrespective of how memory bloat is distributed in the address space. HawkEye fails to meet this property. For RandomAccess-L wherein the first half of the address space has more bloat, HawkEye breaks only 611 huge pages to reclaim 2GB unused physical memory. However, in RandomAccess-R wherein the first half of the address space has far less bloat, HawkEye ends up breaking 2146 huge pages to reclaim the same amount of memory. This is because in RandomAccess-R, when HawkEye scans its address space sequentially, it keeps on breaking huge pages in the first half of the address space even though de-bloating them frees up small amount of memory each. The effect of breaking huge pages can be seen in Table 1 wherein RandomAccess-R suffers 65% slowdown compared to no de-bloating, whereas RandomAccess-L observes a negligible impact on performance. Therefore, it is clear that HawkEye's performance is highly dependent on how memory bloat is distributed in an application.

### 3.2 Unfair De-bloating (Inter-process Scenario)

In cases where multiple applications are running concurrently, HawkEye de-bloats applications in the first-come-first-serve (FCFS) order. Specifically, HawkEye first identifies a process to be de-bloated, de-bloats it fully and then proceeds to the next process in the same FCFS order, if required. This strategy is intuitively unfair because some processes end up giving up most of their huge pages while some may not be affected by de-bloating. Moreover, similar to the issue discussed in §3.1, it ignores the relative contribution of each process towards the total memory bloat present in the system. Therefore, there is a need to incorporate both fairness and efficiency challenges while de-bloating multiple applications.

## 4 EMD: Design and Implementation

This section begins with a high-level description of our design goals and principles. We then present the design and implementation of EMD.

### 4.1 Design Goals and Principles

Among all the existing OS-level solutions, HawkEye is the only system that provides the ability to de-bloat physical memory allocated to huge pages. However, as established in §3, HawkEye's de-bloating policy is suboptimal and unfair. Our goal in EMD is to enable an OS to automatically find a sweet spot in the trade-off space between memory bloat and performance. Thus, at a high level, our design goals and principles are as follows:

**1. Minimal Performance Impact while De-bloating:** De-bloating should have minimal impact on application performance. We achieve this goal through finer-grained tracking of huge page regions and measuring bloat-density (i.e., the number of zero-filled base pages within a huge page).

**2. Fairness across Concurrent Processes:** Fairness is based on the principle that de-bloating should not unfairly penalize one or more applications in the presence of multiple concurrently running workloads. For example, if two processes have identical bloat distributions, a fair de-bloating policy ensures the ratio of huge pages broken is approximately equal for both processes. We achieve fairness with a fine-grained prioritization scheme.

Overall, EMD prioritizes memory reclamation in the order of memory bloat, i.e., huge pages with maximum bloat are de-bloated first. The advantages of this are two-fold: 1) de-bloating this way requires breaking the fewest huge pages to reclaim a certain amount of unused memory, 2) note that if a huge page contains mostly zero-filled unused pages, it is also likely to be referenced less frequently compared to more densely populated huge pages. This means that the contribution of highly bloated pages will likely be low in the overall TLB misses for a given application. Therefore, reclaiming memory from the most bloated pages first also minimizes the impact on runtime performance.

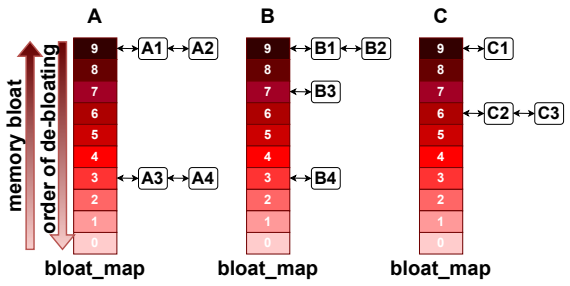



**Figure 5.** A sample representation of bloat_map for three processes A, B, and C.

### 4.2 Efficient Fine Grained De-bloating

To achieve our first goal of efficient de-bloating, EMD considers the distribution of bloat before reclaiming unused physical memory from an application. EMD is based on our key insight that memory bloat is not uniformly distributed across all regions, and that the trade-offs between memory efficiency and performance vary across memory regions.

We first define an important metric that EMD uses for efficient de-bloating: *bloat-density*. Given a huge page, its bloat-density represents how many of its base pages are zero-filled. Therefore, a high value of bloat-density represents a higher amount of memory bloat. EMD prioritizes reclaiming bloat from huge pages with high bloat-density. This helps in reclaiming the maximum amount of memory bloat while breaking a minimum number of huge pages.

An ideal implementation of the above strategy would require sorting all huge pages based on their bloat-density. This can be prohibitively expensive on large memory systems. Instead, we propose a simpler method to achieve the benefit of de-bloating with minimal processing overheads. We implement a per-process data structure called the **bloat_map**[1], which is an array of 10 buckets: a bucket contains huge page regions with similar bloat-density values. On x86 systems with 2MB huge pages, the value of bloat-density is in the range of 0-512. In our prototype, we maintain ten buckets in the per-process bloat_map. Huge pages with bloat-density of 0-49 (i.e., those with less than 10% bloat) are placed in bucket index 0, regions with bloat-density of 50-99 (10-20% bloat) are placed in bucket index 1, and so on. EMD prioritizes de-bloating regions from higher-index buckets first, gradually de-bloating regions from higher indices to lower indices in the bloat_map. Figure 5 illustrates an example bloat_map for three processes: A, B, and C and Figure 6 shows how EMD de-bloats a given process.

[1]Inspired by the access_map data structure of HawkEye [31], population_map of FreeBSD [27], and access_bitvector of Ingens [26].

```
function debloat_process(mm_struct, num_target_pages)
    num_recovered_pages = 0
    density_threshold = 450
    // Loop until meeting target
    while num_recovered_pages < num_target_pages do
        for each vma in mm_struct do
            for each huge_page in vma do
                // compute bloat-density
                bloat_density = COUNT_ZERO_PAGES(hp)
               // skip or de-bloat?
                if bloat_density < density_threshold then
                    continue
                end if
                // de-bloat the huge page
                num_recovered_pages += RMV_ZERO_PG(hp)
               if num_recovered_pages>=num_target_pages then
                    goto out
                end if
            end for
        end for
        // adjust threshold to de-bloat next bucket
        density_threshold = density_threshold - 50
    end while
out:
    return num_recovered_pages
end function
```

**Figure 6.** EMD algorithm for de-bloating huge pages with a given process's address space.

### 4.3 Fairness across All Processes while De-bloating

We extend EMD's fine-grained de-bloating to also align with our notion of fairness. When multiple processes have non-empty buckets at the highest non-empty index, EMD uses a round-robin approach to ensure fairness among these processes. Note that within a given process, de-bloating in the order of higher index buckets to lower index buckets ensures efficiency (i.e., minimal performance impact). Therefore, EMD provides both fairness and efficiency.

In the round-robin approach, we use *quantum* as a means to control the number of huge pages de-bloated with a process in a given iteration. EMD statically sets the quantum to 100 huge page regions (configurable by the user). This ensures that at most 100 huge pages are de-bloated from a process at a given time, before considering other processes for de-bloating. This strikes a balance between fairness and the processing overhead.

To illustrate the fairness policy, consider three applications: A, B, and C. Their virtual address (VA) regions are organized in the bloat-map as shown in Figure 5. EMD de-bloats the regions in the following sequence:

A1, B1, C1, A2, B2, B3, C2, C3, A3, B4, A4.

Overall, EMD de-bloats memory regions starting from the highest index (representing the highest bloat-density) in the bloat map and works toward lower indices. Further, among similar index in the bloat_map of different processes, it uses round-robin scheduling to ensure fairness.

**Table 2.** Specification of the benchmarks.

| Name | Mem footprint 4KB | Mem footprint 2MB | Description |
|---|---|---|---|
| BTree | 10GB | 44GB | Random lookups in a B+ tree [43]. |
| Memcached | 19GB | 50GB | In-memory key-value caching store [18]. |
| RandomAccess-L (RA-L) | 4GB | 8GB | A custom micro-benchmark with high memory bloat in left half. |
| RandomAccess-R (RA-R) | 4GB | 8GB | A custom micro-benchmark with high memory bloat in right half. |

### 4.4 EMD Overhead

Note that applications allocate and de-allocate memory at runtime. Hence, the state of free/used memory and the distribution of memory bloat can change significantly in short periods of time. Therefore, EMD needs to scan a process address space every time before de-bloating. However, the overhead of scanning the address space is not very high in most cases. This has already been established by HawkEye [31][2], which shows that only a small fraction of memory needs to be examined to detect memory bloat in huge pages.

### 4.5 Implementation Notes

We implement EMD as a loadable kernel module designed to optimize memory management on Linux systems. The module includes functionalities to detect and manage zero-filled base pages within huge pages. EMD allows for dynamic loading and unloading, providing flexibility in deployment and operational management without necessitating system downtime. This capability facilitates real-time adaptation to varying system conditions and operational requirements. To trigger recovery from memory bloat, EMD uses two watermarks on the amount of allocated memory in the system: high and low. When the amount of memory in the system exceeds high (80% in our prototype), EMD is activated, which executes periodically until the memory falls below low (65% in our prototype). More sophisticated policies can be used to activate/deactivate de-bloating based on user requirements.

## 5 Evaluation

### 5.1 Experimental Setup and Workloads

**Hardware Setup.** Our experimental setup is based on an AMD EPYC 7401 processor running at 2.0 GHz, featuring 24 cores (48 threads) per socket. The system is equipped with a multi-level TLB hierarchy, including separate instruction and data L1 TLBs (L1-iTLB and L1-dTLB). The L1-iTLB supports 4KB pages with 128 entries (8-way set associative) and 2MB pages with 8 fully associative entries. Similarly, the L1-dTLB accommodates 4KB pages (64 entries, 4-way set associative), 2MB pages (32 entries, 4-way set associative), and 1GB pages (4 fully associative entries). The L2 TLB provides broader coverage with 1536 entries for 4KB/2MB pages (12-way set associative) and 16 entries for 1GB pages (4-way set associative). The system also features a three-level cache hierarchy, including 64KB L1 instruction cache, 32KB L1 data cache, 512KB L2 cache, and an 8MB L3 cache. The platform is configured with 256GB of main memory and runs Ubuntu with Linux kernel version 4.3.

**Workloads.** We evaluate EMD with a mix of real-world workloads and custom microbenchmarks. We evaluate the improvements due to our fine-grained de-bloating strategy based on bloat-density in both performance and fairness for single, multiple homogeneous, and multiple heterogeneous workloads. Table 2 provides the details of the workloads.

### 5.2 Evaluating Intra-process De-bloating

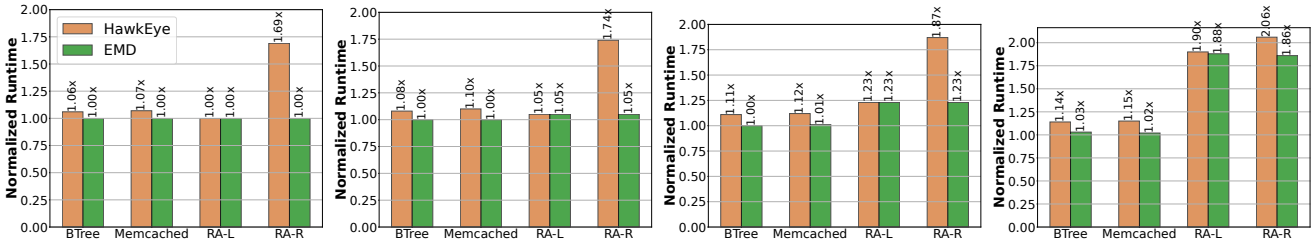
**Goal.** We first evaluate the effectiveness of our bloat-density-based de-bloating strategy. Specifically, we measure the impact of de-bloating on application runtime performance when memory de-bloating is performed using HawkEye versus EMD, comparing both methods against a huge page only baseline (i.e., no de-bloating). This case serves as the baseline for best-case runtime performance.

**Methodology.** To analyze the effects of de-bloating under different scenarios, we reclaim different proportions of unused memory using de-bloating i.e., 20%, 30%, 40% and 50%. For instance, if an application's total memory bloat is 8GB and we aim to de-bloat 20% of the physical memory, we effectively reclaim 1.6GB worth of zero-filled 4KB pages. In general, a system that breaks the least number of huge page mappings to reclaim a certain amount of memory is the most efficient.

**Results.** Figure 7 illustrates the performance impact of HawkEye and EMD across four workloads: BTree, Memcached, RandomAccess-L (RA-L), and RandomAccess-R (RA-R). HawkEye de-bloats memory sequentially in virtual address space order (low to high) without considering that bloat distribution is often non-uniform across the address space. Therefore, all four workloads experience a significant increase in runtime when de-bloating is performed using HawkEye. For example, RandomAccess-R has low bloat in the left half and higher bloat in the right half. As a result, HawkEye breaks more huge pages during de-bloating, leading to a runtime increase of more than 2× in the worst case (at 50% de-bloating). The average runtime overhead for RandomAccess-R with HawkEye is 1.84×. In contrast, EMD's fine-grained de-bloating strategy, which accounts for bloat distribution, reduces this overhead to an average of 1.28×.

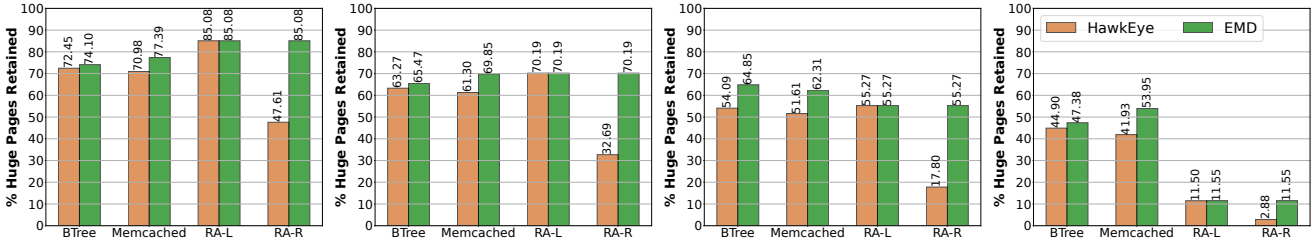
For Memcached and BTree, a similar trend is observed. HawkEye results in an average runtime increase of 1.1× for both workloads, whereas EMD incurs an overhead of

[2]"HawkEye showed that scanning each in-use page requires ~10 bytes on average, whereas bloat pages require scanning all 4096 bytes. Hence, overheads are proportional to the number of bloat pages, not total allocated memory.



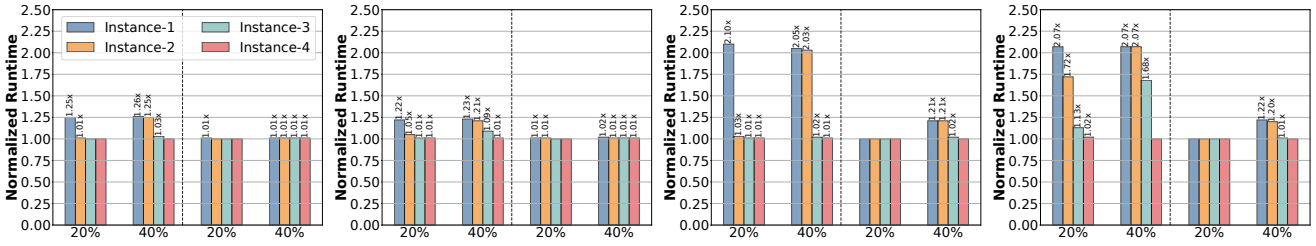

**(a)** 20% de-bloating. **(b)** 30% de-bloating. **(c)** 40% de-bloating. **(d)** 50% de-bloating.

**Figure 7.** Performance impact compared to no de-bloating in different scenarios (lower is better).



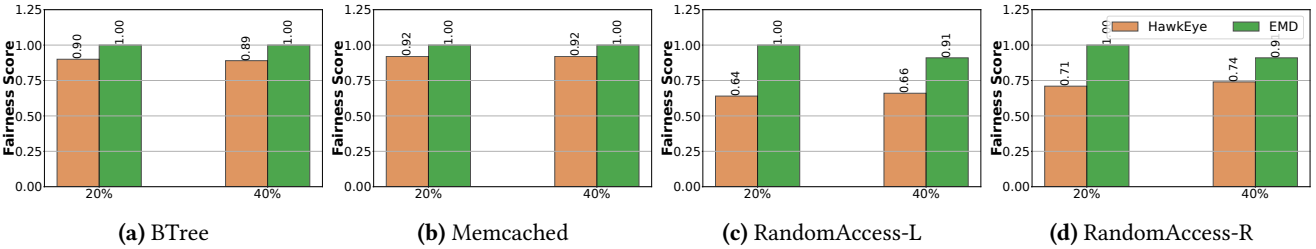

**(a)** 20% de-bloating. **(b)** 30% de-bloating. **(c)** 40% de-bloating. **(d)** 50% de-bloating.

**Figure 8.** Number of huge pages retained in different scenarios (higher is better).




**(a)** BTree **(b)** Memcached **(c)** RandomAccess-L **(d)** RandomAccess-R

**Figure 9.** Performance impact on four instances of the same workload under different de-bloating scenarios (lower is better). The vertical dashed line in each plot separates HawkEye (left) and EMD (right) results.




**(a)** BTree **(b)** Memcached **(c)** RandomAccess-L **(d)** RandomAccess-R

**Figure 10.** Fairness scores for four homogeneous instances of each workload under different de-bloating scenarios (higher is better).

less than 1.01×. HawkEye and EMD perform optimally for RandomAccess-R since, in this case, the optimal de-bloating order is naturally the sequential order that HawkEye uses.

Both HawkEye and EMD experience roughly similar performance at 50% de-bloating. This is because under aggressive de-bloating, most huge pages are broken (more than 75%), causing the application to rely predominantly on base

pages. Consequently, application runtime approaches that of a native execution without huge pages. Thus, at higher de-bloating levels, both HawkEye and EMD converge to roughly similar performance.

To explain EMD's superior performance, Figure 8 presents the number of huge pages retained by both systems at different de-bloating levels. Since HawkEye de-bloats memory without considering non-uniform bloat distribution, it breaks significantly more huge pages than EMD. For example, at 40% de-bloating, HawkEye retains only about 18% of the huge pages initially allocated to RadomAccess-R whereas EMD retains more than 50% of the huge pages. Similar trend is also seen for other applications wherein EMD retains 20% more huge pages than HawkEye in many cases. For RandomAccess-L, the number of huge pages retained by both systems remains the same. These results show that by breaking fewer huge pages and performing fine-grained de-bloating, EMD significantly minimizes the impact of de-bloating on application performance, compared to HawkEye.

### 5.3 Evaluating Inter-process De-bloating

**5.3.1 Defining Fairness.** We quantify fairness based on the work by Craeynest et al. [17]. Concretely, let

$$S = \{S_1, S_2, \ldots, S_N\}$$

be the set of slowdowns experienced by each of the $N$ instances under a given de-bloating policy, where

$$S_i = \frac{T_i^{\text{debloat}}}{T_i^{\text{baseline}}}$$

Here, $T_i^{\text{baseline}}$ is the runtime of instance $i$ with no de-bloating (ideal case), and $T_i^{\text{debloat}}$ is its runtime under the de-bloating scheme being evaluated. We then compute the mean slowdown

$$\mu_S = \frac{1}{N}\sum_{i=1}^{N} S_i$$

and the standard deviation

$$\sigma_S = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(S_i - \mu_S)^2}$$

The coefficient of variation $CV$ is defined as $\frac{\sigma_S}{\mu_S}$. It measures the variability in slowdowns relative to the average slowdown across all instances. It captures how much individual instances deviate from a fair baseline. Thus, $CV$ serves as a measure of unfairness. Consequently, we define fairness as follows.

$$\text{Fairness} = 1 - \frac{\sigma_S}{\mu_S}$$

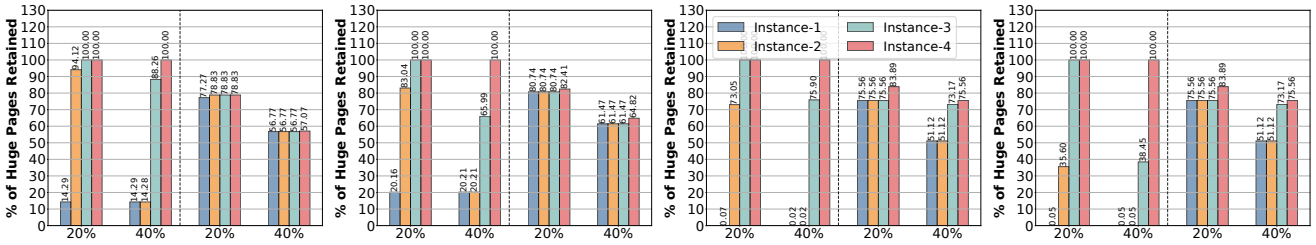
Since $CV$ is always non-negative, the fairness score ranges between 0 and 1. A value of 1 indicates perfectly equal slowdowns across all instances (i.e., fully fair de-bloating), while values approaching 0 indicate increasing levels of unfairness. This statistically grounded metric offers several advantages for our context. First, it is application-agnostic, relying only on per-instance runtimes, which the OS can observe without special instrumentation or hardware counters. Second, by normalizing the variability with respect to the average slowdown, this metric captures fairness independently of the absolute performance drop. For instance, if all instances slow down equally due to de-bloating, the system is still considered fair—even if the slowdown is large—because no instance is disproportionately affected. This normalization ensures that fairness is judged based on how balanced the impact is across instances, rather than the magnitude of slowdown alone. Finally, using a single scalar in $[0, 1]$ makes it straightforward to compare the fairness of different de-bloating policies.

We next experiment with multiple applications running concurrently to study both fairness and performance, first with identical instances of a single application, and then with heterogeneous applications running simultaneously.

**5.3.2 Homogeneous Workloads.** To understand the effect of different de-bloating policies on identical instances, we perform four experiments. In the first experiment, we execute four identical instances of BTree, launching them one after the other with a few seconds of delay in between. The other experiments replicate this with Memcached, RandomAccess-L, and RandomAccess-R. To distinguish different instances of the same workload, we label them based on the process creation order. For example, Instance-1 denotes the first instance while Instance-2, Instance-3, and Instance-4 denote the second, third, and fourth instances, respectively, of a particular workload. We investigate the effect of de-bloating on the performance of each instance while reclaiming 20% and 40% of the unused memory across all instances.

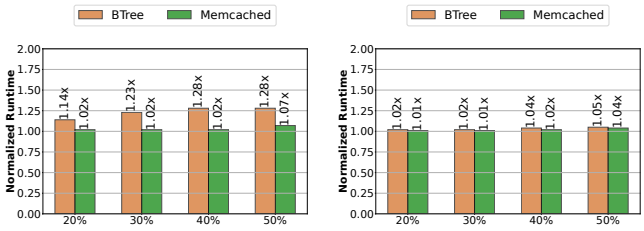
Unfair de-bloating leads to unfair impact among concurrently running applications. To understand this more clearly, Figure 9 shows the runtime of all the instances of BTree. With HawkEye, at all de-bloating levels, all instances finish at different times. For example, in de-bloating 40% of the memory, the first instance has a 1.26× increase in runtime, the second instance has a 1.25× increase in runtime but the third and fourth instances are nearly unaffected. On the other hand, with EMD, all instances finish at almost the same time, ensuring fair runtime performance. This pattern is observed for Memcached, RandomAccess-L, and RandomAccess-R as well (see sub-figures (b), (c) and (d) of Figure 9).

This pattern is also evident when we compute the fairness score which ranges from 0 (completely unfair) to 1 (perfectly fair). Figure 10 shows that EMD consistently outperforms HawkEye. Under both 20% and 40% de-bloating, EMD achieves a perfect score of 1.0 for BTree and Memcached, indicating fair de-bloating across all instances. For the microbenchmarks, HawkEye attains only average




**(a)** BTree **(b)** Memcached **(c)** RandomAccess-L **(d)** RandomAccess-R

**Figure 11.** Number of huge pages retained on four instances of the same workload under different de-bloating scenarios (higher is better). The vertical dashed line in each plot separates HawkEye (left) and EMD (right) results.




**(a)** HawkEye **(b)** EMD

**Figure 12.** Runtime compared to no de-bloating scenario with one instance of BTree and one instance of Memcached running simultaneously (lower is better).

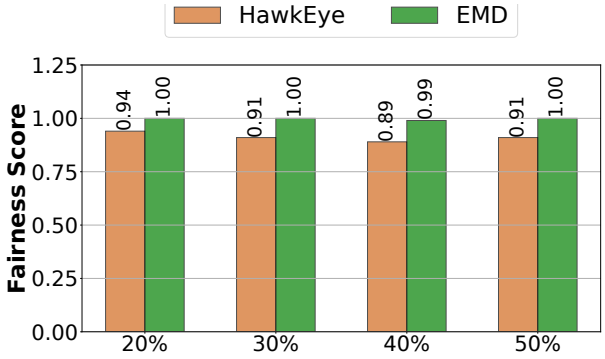
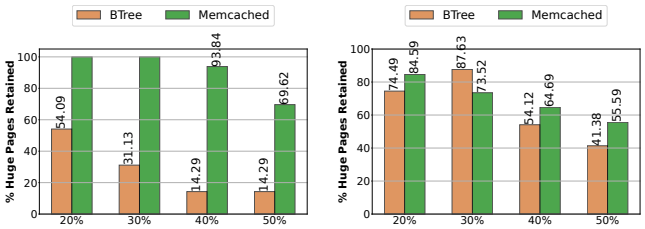



**Figure 13.** Fairness scores for heterogeneous instances comprising one BTree and one Memcached instance running simultaneously under different de-bloating scenarios (higher is better).




**(a)** HawkEye **(b)** EMD

**Figure 14.** Number of huge pages retained with one instance of BTree and one instance of Memcached running simultaneously (higher is better).

fairness scores of 0.65 for RandomAccess-L and 0.72 for RandomAccess-R, while EMD maintains a high score of 0.95 for both microbenchmarks.

HawkEye is unfair because it breaks huge pages one instance at a time based on the process arrival order i.e., it starts de-bloating memory from the first instance and moves on to the next instance only after first instance. In contrast, EMD ensures fairness by judiciously breaking huge pages across all instances. It identifies the most bloated regions across all instances and de-bloats them. This is evident from Figure 11 which show the number of huge pages retained by HawkEye and EMD for BTree, Memcached, RandomAccess-L, and RandomAccess-R, respectively. For example, when de-bloating 40% of memory from BTree, HawkEye retains only about 14% of the huge pages from Instance-1 and Instance-2 whereas Instance-3 and Instance-4 get to retain almost all of their huge pages. In contrast, with EMD, all instances retain nearly 56% of their huge pages. Similar patterns are observed for Memcached, RandomAccess-L, and RandomAccess-R. These results confirm that EMD is fair in the presence of multiple concurrently running applications.

**5.3.3 Heterogeneous Workloads.** To evaluate the efficacy of different de-bloating strategies for heterogeneous instances, we grouped workloads into sets, each containing one instance of BTree and one instance of Memcached. The BTree instance was launched a few seconds before the Memcached instance to capture the effect of process creation order. We assessed the impact of various de-bloating strategies while reclaiming 20%, 30%, 40% and 50% of the unused memory of both applications. We normalize the runtime of each application against that of huge pages without de-bloating.

Figure 12 illustrates the runtime of both BTree and Memcached under all de-bloating scenarios. With HawkEye, BTree instances experiences a 1.28× increase in runtime

**Table 3.** Comparison against 4KB pages

| Benchmark | Linux-4KB | EMD | |
|---|---|---|---|
| | | High Bloat | Low Bloat |
| BTree | 1733 | 1070 | 1301 |
| Memcached | 1633 | 991 | 1192 |
| RandomAccess-L | 480 | 234 | 434 |
| RandomAccess-R | 482 | 231 | 431 |

in the worst case (1.22× on average), whereas the performance of Memcached is nearly unaffected. In contrast, with EMD, both applications complete their execution at nearly the same time with minimal impact on performance compared to the best case.

We now evaluate the fairness score between heterogeneous instances. As shown in Figure 13, EMD achieves a near-perfect fairness score of 1.0 (as one score is 0.99), across all de-bloating levels, demonstrating fully balanced de-bloating between the BTree and Memcached instances. In contrast, HawkEye averages a fairness score of 0.91, reflecting its tendency to complete de-bloating for the BTree instance before proceeding to Memcached. This imbalance reinforces the performance disparities observed earlier and foreshadows the uneven number of huge pages reclaimed from each instance.

Figure 14 shows the number of huge pages retained by HawkEye and EMD for the same experiment. As the figure shows, EMD ensures a fair reclamation of huge pages among BTree and Memcached, whereas HawkEye prioritizes de-bloating BTree before moving on to Memcached (note that this is because we launch BTree before Memcached in our experiments). For instance, when de-bloating 40% of the unused memory, HawkEye retains only about 14% of the huge pages of BTree whereas Memcached retains almost all of its huge pages. In contrast, EMD retains huge pages nearly fairly i.e., 54% in BTree and 64% in Memcached, ensuring a more balanced de-bloating approach. This fairness in de-bloating translates into fair runtime performance.

## 6 Discussion

**Performance Comparison against Base Pages:** EMD navigates the memory-performance trade-off fairly and efficiently. Table 3 shows that EMD consistently matches or outperforms Linux configured with 4KB base pages across all evaluated workloads. The runtimes, measured in seconds, clearly demonstrate the inefficiency of using only base pages—4KB runtimes are significantly higher for all benchmarks. By leveraging huge pages and selectively de-bloating only the zero-filled base pages, EMD preserves low address translation overhead while reducing unnecessary memory usage.

**Performance Evaluation when There Is No Bloat:** Not all workloads exhibit the memory bloat that EMD is designed to reclaim. Figure 15 presents several benchmarks that show zero bloat when running with Linux-THP, yet still achieve substantial performance benefits from Linux-THP [1, 9, 13, 14]. To test whether EMD inadvertently introduces overhead in such "bloat-free" scenarios, we explicitly invoked EMD's de-bloating mechanism on these workloads. As shown in Figure 15, the performance impact is negligible—the average overhead across these benchmarks is 1%, indicating that EMD is effectively non-intrusive when no bloat is present.

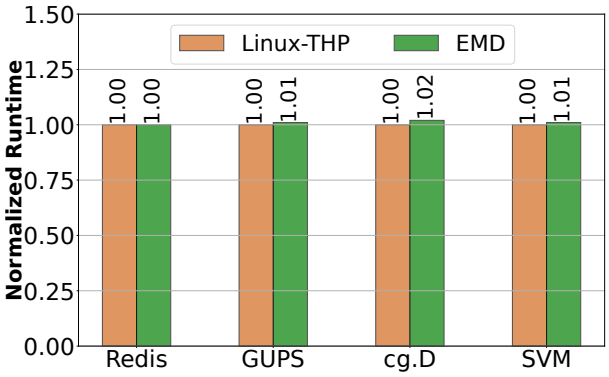



**Figure 15.** Normalized runtime of applications that are not subject to memory bloat.

**Table 4.** Memory footprint of BTree and Memcached in Linux kernel version 6.7.2.

| | Memory (4KB) | Memory (2MB) |
|---|---|---|
| BTree | 10GB | 44GB |
| Memcached | 19GB | 50GB |

**Memory Bloat in Recent Linux Kernels:** Although Linux has significantly improved THP support, particularly in addressing latency, external fragmentation, NUMA, and swapping-related issues, memory bloat remains a persistent challenge. To validate this, we measured the memory footprint of Memcached and BTree on Linux 6.7.2 (see Table 4). These results clearly show that memory bloat continues to exist. Since our solution is not tied to any specific kernel version, we expect our findings to generalize well across newer Linux versions.

**Support for 1GB Pages:** EMD is designed around the key insight that memory bloat can be mitigated dynamically by identifying and reclaiming unused, zero-filled base pages within huge pages. This de-bloating mechanism is independent of the specific huge page size, making EMD conceptually applicable not only to 2MB pages but also to larger page sizes such as 1GB. However, we do not include a quantitative evaluation using 1GB pages because neither the Linux kernel nor HawkEye currently support 1GB pages under THP.

**Other Policies for Fair De-bloating:** While our bloat-map–based policy prioritizes de-bloating based on the relative amount of zero-filled pages per process, one could

also achieve fairness using other policies e.g., based on application-level performance metrics or TLB hit (or miss) ratio. However, application-metric based or hardware-counters–based fairness brings practical challenges. The OS kernel lacks fine-grained insight into application internal performance metrics, making it difficult to translate raw throughput or latency into de-bloating decisions. Likewise, relying on TLB hit statistics assumes easy access to performance counters—a luxury absent in many virtualized or secure environments. In contrast, our bloat-map heuristic depends only on readily available page-table metadata (zero-page counts), enabling a light-weight, portable, and low-overhead fairness mechanism that works uniformly on all environments.

## 7 Related Work

**Hardware Approaches:** Several techniques aim to reduce TLB misses and improve address translation. Direct Segments [11], later extended to virtualized [19] and heterogeneous systems [22] map large virtual regions to contiguous physical memory. Other methods like SpecTLB [23], GLUE [36], and CoLT [35] use speculation or coalescing to increase TLB reach. Mosaic Pages [21] and Perforated Pages [34] enhance TLB efficiency using compressed or sparse mappings. DyLeCT [33] dynamically adapts page length for compressed memory. While effective, these solutions require hardware changes. In contrast, EMD is a pure software approach and can be readily adopted in existing systems.

**Software Approaches:** OS-level techniques improve huge page usage without hardware support. Navarro et al. [28] proposed contiguity-aware allocation, influencing the work on FreeBSD [27]. Ingens [26] adds dynamic promotion based on access patterns but lacks de-bloating. HawkEye [31], most similar to EMD, uses fine-grained tracking but performs sequential de-bloating without fairness. Quicksilver [44] and Trident [38] enhance huge page lifecycle and extend support to 1GB pages. EMD complements these efforts by introducing fair and efficient runtime de-bloating with no hardware or application changes.

## 8 Conclusion

Huge pages promise to boost the performance of memory-hungry applications, albeit at the expense of consuming extra physical memory known as bloat. In this paper, we show that current OS-level solutions either lack support for dynamically mitigating memory bloat or suffer from avoidable performance and fairness issues. We address these shortcomings with EMD (Efficient Memory De-bloating), an OS-level solution that employs a prioritization scheme to reclaim unused memory fairly and efficiently at runtime. EMD offers a promising solution to improve the efficiency and fairness of large page management in operating systems.

## References


[1] 2006. GUPS: HPCC RandomAccess benchmark. https://github.com/alexandermerritt/gups.

[2] 2024. Recommendation for disabling huge pages for Hadoop. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf.

[3] 2024. Recommendation for disabling huge pages for MongoDB. https://docs.mongodb.org/manual/tutorial/transparent-huge-pages/.

[4] 2024. Recommendation for disabling huge pages for NuoDB. http://www.nuodb.com/techblog/linux-transparent-huge-pages-jemalloc-and-nuodb.

[5] 2024. Recommendation for disabling huge pages for VoltDB. https://docs.voltdb.com/AdminGuide/adminmemmgt.php.

[6] 2025. Transparent Hugepage Support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

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

[8] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 273–287. doi:10.1109/PACT.2017.38

[9] 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* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. ACM, New York, NY, USA, 158–165. doi:10.1145/125826.125925

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

[11] 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* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 237–248. doi:10.1145/2485922.2485943

[12] Abhishek Bhattacharjee and Daniel Lustig. 2017. *Architectural and Operating System Support for Virtual Memory*. Morgan & Claypool Publishers. doi:10.2200/S00795ED1V01Y201708CAC042

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

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

[15] Jonathan Corbet. [n. d.]. Huge pages, slow drives, and long delays. https://lwn.net/Articles/467328/.

[16] 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* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 435–448. doi:10.1145/3037697.3037704

[17] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. 2013. Fairness-Aware Scheduling on Single-ISA Heterogeneous Multi-Cores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE Press, GA, 177–188. https://ieeexplore.ieee.org/document/6618815

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

[19] 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* (Cambridge, United Kingdom) *(MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 178–189. doi:10.1109/MICRO.2014.37

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

[21] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N. Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A. Bender, and Abhishek Bhattacharjee. 2023. Mosaic Pages: Big TLB Reach with Small Pages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ACM, GA, 443–448. https://doi.org/10.1145/3582016.3582021

[22] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. *SIGPLAN Not.* 53, 2 (mar 2018), 637–650. doi:10.1145/3296957.3173194

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

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

[25] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. 2016. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 631–643. doi:10.1109/HPCA.2016.7446100

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

[27] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. 2014. *The Design and Implementation of the FreeBSD Operating System* (2nd ed.). Addison-Wesley Professional.

[28] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. http://www.usenix.org/events/osdi02/tech/navarro.html

[29] Akash Panda, Ashish Panwar, and Arkaprava Basu. 2021. nuKSM: NUMA-aware Memory De-duplication on Multi-socket Servers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 258–273. doi:10.1109/PACT52795.2021.00026

[30] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. 2021. *Fast Local Page-Tables for Virtualized NUMA Servers with VMitosis*. Association for Computing Machinery, New York, NY, USA, 194–210. https://doi.org/10.1145/3445814.3446709

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

[32] 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* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 679–692. doi:10.1145/3173162.3173203

[33] Gagandeep Panwar, Muhammad Laghari, Esha Choukse†, and Xun Jian. 2024. DyLeCT: Achieving Huge-page-like Translation Performance for Hardware-compressed Memo. In *ISCA '24: Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ACM, GA.

[34] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. 2020. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *ISCA '20: Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ACM, GA, 913–925. https://doi.org/10.1109/ISCA45697.2020.0007

[35] 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* (Vancouver, B.C., CANADA) *(MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 258–269. doi:10.1109/MICRO.2012.32

[36] 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* (Waikiki, Hawaii) *(MICRO-48)*. ACM, New York, NY, USA, 1–12. doi:10.1145/2830772.2830773

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

[38] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. 2021. Trident: Harnessing Architectural Resources for All Page Sizes in X86 Processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1106–1120. doi:10.1145/3466752.3480062

[39] 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* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 469–480. doi:10.1145/3079856.3080210

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

[41] Statista. [n. d.]. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. https://www.statista.com/statistics/871513/worldwide-data-created/.

[42] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. 2011. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 340–349. doi:10.1109/PACT.2011.65

[43] Mitosis workload BTree. 2019. Open Source Code Repository. https://github.com/mitosis-project/mitosis-workload-btree.

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