Taming Hot Bloat Under Virtualization with HugeScope

Chuandong Li[1 2] *Sai Sha*[1 3] *Yangqing Zeng*[1] *Xiran Yang*[1]
Yingwei Luo[1 2] *Xiaolin Wang*[1 2] *Zhenlin Wang*[4] *Diyu Zhou*[1 5]

[1] National Key Laboratory for Multimedia Information Processing, School of CS, Peking University
[2] Zhongguancun Laboratory, [3] Beijing Huawei Digital Technologies, [4] Michigan Tech, [5] EPFL

Abstract

Huge pages are effective in reducing the address translation overhead under virtualization. However, huge pages suffer from the hot bloat problem, where accesses to a huge page are skewed towards a few base pages (*i.e.*, 4KB page), making the hypervisor (mistakenly) classify the whole huge page as hot. Hot bloat renders several critical techniques used in virtualization ineffective, including tiered memory and page sharing. Prior work addressing hot bloat either requires hardware modification or targets a specific scenario and is not applicable to a hypervisor.

This paper presents HugeScope, a lightweight, effective and generic system that addresses the hot bloat problem under virtualization based on commodity hardware. HugeScope includes an efficient and precise page tracking mechanism, leveraging the other level of indirect memory translation in the hypervisor. HugeScope provides a generic framework to support page splitting and coalescing policies, considering the memory pressure, as well as the recency, frequency, and skewness of page access. Moreover, HugeScope is general and modular, thereby can be easily applied to various scenarios concerning hot bloat, including tiered memory management (HS-TMM) and page sharing (HS-Share). Evaluation shows that HugeScope incurs less than 4% overhead, and by addressing hot bloat, HS-TMM improves performance by up to 61% over vTMM while HS-Share saves 41% more memory than Ingens while offering comparable performance.

1 Introduction

As the memory footprint of virtual machines (VMs) increases [15, 22], huge pages are employed to alleviate the pressure of TLB [12, 14, 24, 33, 34, 37]. This brings significant performance improvements (11% - 53% across a wide range of benchmarks as reported in prior work [24]) due to two reasons: (1) it increases the TLB coverage, reducing the TLB miss rate; (2) it reduces the overhead of page walking.

Unfortunately, as discovered by prior reasearch [4, 7, 26], huge pages lead to the hot bloat problem, where accesses to a huge page are skewed toward a few base pages (*i.e.*, 4KB page). We refer to such huge pages as unbalanced huge pages. With the conventional approaches to track memory accesses, hot bloat misleads the hypervisor to believe the whole huge page is hot, thereby making suboptimal decisions.

Several critical techniques in virtualization, such as tiered memory management [2, 35, 39, 46] and page sharing [13], must carefully tackle hot bloat to be effective. Our evaluation reveals that hot bloat causes performance degradation of tiered memory management system up to 50% since cold base pages in unbalanced huge pages occupy the precious fast memory (§2.2). Furthermore, hot bloat causes the current huge page sharing mechanism to fail to resolve the tradeoff between memory saving and performance (§2.3),

Few prior systems focus on the hot bloat problem, and those that do have various limitations (§2.5). In particular, in virtualized environments on commodity hardware, no previous work can accurately track memory accesses within a huge page at a reasonable cost. This limitation prevents the hypervisor from identifying huge pages with skewed accesses effectively, thereby hindering the ability to address the hot bloat problem.

To address hot bloat, we performed an extensive evaluation of the relevant systems. Our evaluation reveals two new findings: (i) page sharing systems can make a better tradeoff between performance and memory saving by considering hot bloat; (ii) sampling access to a small portion of huge pages, as used by prior work [2], is highly inaccurate.

We present HugeScope, an **effective**, **lightweight**, **generic**, and **modular** solution to the hot bloat problem in a hypervisor without requiring guest OS and hardware modifications. Our key observation behind HugeScope is to exploit the other level of address indirection brought by virtualization. Specifically, the current hypervisor rarely modifies a VM's extended page table (EPT) entry after the VM has booted up. This enables HugeScope to bypass most (94%) of the overhead incurred by page splitting and coalescing,

simply by splitting the page table to enable memory access tracking at the base page granularity.

HugeScope proposes a comprehensive page splitting/ coalescing policy for hot bloat. To choose page candidates for splitting/coalescing, HugeScope carefully considers i) the condition to trigger page splitting/coalescing and ii) the recency, frequency, and skewness of memory access. This policy makes HugeScope much more effective than existing mechanisms.

Finally, we design HugeScope to be a general and modular solution for hot bloat, exposing a set of flexible interfaces. We integrate HugeScope into a tiered memory management system (HS-TMM) and a page sharing system (HS-Share). Our evaluation shows that HugeScope incurs less than 4% overhead, and by addressing hot bloat, HS-TMM improves performance by up to 61% while HS-Share saves 41% more memory while offering comparable performance.

In summary, we make the following contributions:

- **Analysis.** Our analysis reveals new findings: (i) page sharing system should consider hot bloat to be effective; (ii) access sampling approach is highly ineffective.
- **HugeScope.** We propose HugeScope, an effective framework for hot bloat under virtualization.
- **HS-TMM and HS-Share.** Using HugeScope, we build a tiered memory management system (HS-TMM) and a page sharing system (HS-Share) to address hot bloat.

2 Background and Motivation

This section discusses the hot bloat phenomenon (§2.1), its impact on two important techniques: tiered memory management (§2.2) and page sharing (§2.3), a motivation for tackling hot bloat within a hypervisor (§2.4), and finally explains why existing techniques are insufficient (§2.5).

2.1 The hot bloat phenomenon

For many workloads, during a period of its execution, access to base pages (*i.e.*, 4KB pages) of a huge page are highly skewed [4, 7, 26, 44]; some base pages are frequently accessed, while others are rarely accessed or not at all. To our knowledge, PRISM [4] includes the most extensive study to date on this phenomenon. For a set of 35 benchmarks including a complete set of PARSEC, SPEC, three GPU benchmarks, and several server benchmarks, 69% exhibits such access skewness, demonstrating its wide existence.

Prior work defines this phenomenon with different terms, such as "poor page metadata fidelity" [4], "hotness fragmentation" [7]. We refer to this phenomenon as "hot bloat" to capture its essential detrimental effects on systems that rely on memory access information to make decisions; the systems will mistakenly classify the whole huge page as hot due to skewed access to a few base pages.

Hot bloat vs. memory bloat. In addition to the reason mentioned above, we call this phenomenon "hot bloat" to emphasize both the similarity and the difference between it and memory bloat [24, 32] caused by huge pages.

Memory bloat is the internal fragmentation caused by huge pages, where the huge page uses 2MB of memory even if many 4KB pages within it are freed. In essence, hot bloat reflects the skewness of *memory accesses*, resulting in an overestimation of the *hot page set*, while memory bloat reflects the skewness of *memory utilization*, resulting in an overestimation of *footprint*.

2.2 Hot bloat with Tiered Memory Systems

Tiered memory systems [2, 21, 29, 35, 39, 46] scale up the precious memory capacity. To maximize performance, frequently accessed pages are placed in fast memory (*e.g.*, DRAM), while infrequently accessed pages go to slow memory (*e.g.*, NVM [3, 25, 43]).

Prior studies [26, 44] have pointed out that hot bloat severely degrades the performance of tiered memory systems. This is because, for a huge page with skewed accesses, the underlying system is likely to classify the huge page as a hot one, thereby moving it to the fast memory. However, in such a case, the page's rarely accessed base pages waste the precious fast memory space, resulting in more accesses to the slow memory in the end.

Experimental setups. We confirm this finding in a virtualization environment with *vTMM* [39], a recent tiered memory system specifically designed for virtualization. We conduct the experiments on a virtual machine allocated with 8GB DRAM as fast memory and 120GB Optane persistent memory [18] as slow memory. To demonstrate how different ratios of unbalanced huge pages (i.e., huge pages with highly skewed access) may impact performance, we use a microbenchmark as the workload. (§4.5 presents the evaluation of real-world applications.) The microbenchmark initially allocates and touches 40GB of memory, but subsequently only accesses 4GB of memory. We adjust the portion of unbalanced huge pages by changing the distribution of the 4GB memory. We evaluate vTMM with two setups: vTMM-Huge and vTMM-base, where the microbenchmark is mapped with 2MB huge pages and 4KB base pages, respectively.

Results. Figure 1 shows the results. When there is no unbalanced huge page, vTMM-Huge achieves the best performance, due to the lower address translation overhead enabled by huge pages. However, as the portion of unbalanced huge pages increases, vTMM-Huge's performance quickly degrades, due to the reason mentioned above. HS-TMM addresses the hot bloat, achieving the best performance in different ratios of unbalanced huge pages.

Confirmation #1. **Hot bloat severely damages the performance of a tiered memory system.**

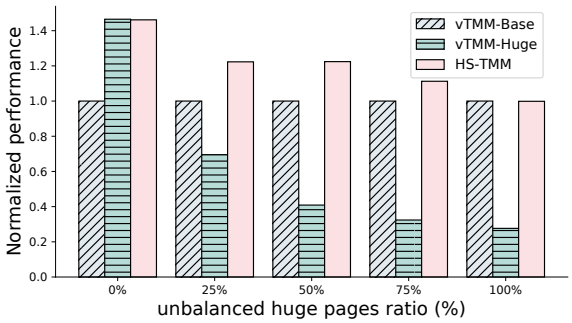

Figure 1: Impact of hot bloat on tiered memory systems (higher is better).

Table 1: Page sharing results under different policies

Policy	Memory saving	Avg. Performance	Huge page ratio
Huge page only	6MB (<1%)	1	100%
Base page only	8590MB (46.6%)	0.704	0%
KSM	8568MB (46.4%)	0.727	3%
Ingens	952MB (5.2%)	0.981	69%
THP Shrinker	152MB (0.82%)	0.994	99%
HugeScope	5448MB (29.6%)	0.943	21%

Table 2: Comparison of hot bloat solutions

System	Keeping most huge page performace	Commodity hardware	Generic scenario	Applicable to a hypervisor
PRISM [4]	✓	×	✓	✓
RainBow [44]	×	×	×	✓
HotBox [7]	×	✓	×	✓
Memtis [26]	✓	✓	×	×
HugeScope	✓	✓	✓	✓

2.3 Hot bloat with Page Sharing

Page sharing is a lightweight mechanism in a virtualization environment to improve memory utilization. With page sharing, a hypervisor first identifies the pages with the same content across all the VMs and then deduplicates these pages by mapping a single copy in memory, protected with copy-on-write. Page sharing is widely deployed in production systems [13, 19, 36] and studied by prior research [16, 24, 34, 42].

Existing page sharing mechanisms. Prior work has studied the impact of huge pages on page sharing and reveals a difficult trade-off between performance and memory saving. Specifically, compared to base pages, huge pages are much more likely to have different content, reducing the effectiveness of page sharing [34]. As a result, one approach (Linux Kernel Same-page Merging or KSM [13]) aggressively splits a huge page whenever it finds a base page within the huge page that can be deduplicated, incurring high performance overhead. Another approach (THP Shrinker [48]) splits only huge pages that contain zero-filled base pages. Recent systems (LPageBreak [16], Ingens [24]) splits only cold (i.e., infrequently accessed) huge pages for page sharing.

New observation. Our new observation is that, by considering hot bloat, the aforementioned trade-off between performance and space-saving can be better resolved (by splitting the *unbalanced huge pages* as detailed in §2.5).

Evaluation setups. We confirm our new observation by running an intuitional microbenchmark on two VMs, respectively, which in total utilize 18GB of memory (§4.6 presents the evaluation on real-world applications). Each VM initializes a buffer with the same data, but issues different access requests. Within the buffer, each VM accesses 1) 25% of the huge pages with high frequency and low skewness towards its base pages; 2) 50% of the huge pages with high frequency and high skewness towards their base pages; and 3) the remaining 25% of the huge pages with low frequency.

As shown in Table 1, if one does not split any huge pages (`Huge page only`), the memory saving is low. As the other extreme, `Linux KSM` splits most huge pages. Although this saves the most memory, it also incurs the highest performance overhead. `Ingens` and `THP Shrinker` split a few huge pages, subsequently achieving little space saving. Considering hot bloat, our solution, HS-Share, achieves a better trade-off between performance and memory savings.

***Finding #1.* Page sharing systems need to consider the hot bloat problem.**

2.4 Why tackling hot bloat in a hypervisor

We tackle the hot bloat issue at the hypervisor level because it has a more severe impact in virtualized environments than in native ones. In native environments, the tiered memory system is the only scenario reported in previous studies [26, 44]. This scenario also applies to virtualized environments [2, 39]. In addition, virtualized enviroments also face hot bloat in page sharing between VMs [24] and VM migration [31]. While there also exist process migration and page sharing among processes in native enviroments, these technologies are much more common and important with virtualization.

2.5 Related work

Table 2 compares HugeScope with existing solutions for hot bloat. Among them, PRISM [4] and RainBow [44] are architectural solutions that require hardware modifications. HotBox [7] completely avoids the use of huge pages, and therefore missing its performance benefits. The closest work to HugeScope is Memtis [26]. However, Memtis tracks memory accesses using processor event-based sampling (PEBS), which does not apply to a hypervisor (as detailed in §2.5.2). In addition, Memtis only targets tiered memory and is not general. It splits a huge page by evaluating whether doing so results in better overall performance by estimating the improved hit rate and the performance difference between the fast and slow memory tiers. It is unclear how to generalize this policy for other scenarios, such as page sharing.

2.5.1 Keys to addressing hot bloat

The key point in addressing hot bloat is, first, tracking memory access to obtain (1) the frequency and recency of mem-

Table 3: Comparison of address tracking techniques

System	Mechanism	Lightweight	Subpage	Applicable to virtualization
TPP [29]	Page fault(PF)	×	✓	✓
Ingens [24]	A/D bits	✓	×	✓
HeMem [35]	PEBS	✓	×	×
Memtis [26]	PEBS	✓	✓	×
Thermostat [2]	Sampling PF	×	✓	✓
HugeScope	Two-phase scanning	✓	✓	✓

ory accesses to classify hot and cold pages; and (2) access skewness within huge pages. Based on this information, the system can decide the base pages to coalesce into huge pages and the huge pages to split into base pages. Next, we explain why existing memory access tracking approaches are insufficient (§2.5.2).

2.5.2 Memory access tracking technique

Table 3 summarizes the memory access tracking techniques.

Page table-based memory access tracking. The most straightforward approach to tracking memory access is through page faults [2, 23, 29]. However, handling a page fault is expensive (and it gets more expensive in a virtualized environment [38, 39]), so page fault-based approaches are limited to tracking a small sample of memory accesses.

Modern hardware includes an access/dirty (A/D) bit in each page table entry to facilitate memory access tracking. Unfortunately, this approach [17, 24, 39] does not enable tracking accesses to the base page within a huge page, and thus cannot obtain access skewness information.

Processor event-based sampling. Recent systems [26, 35] use a hardware feature named processor event-based sampling (PEBS) to track memory accesses. PEBS takes as input a trigger condition (*e.g.*, every 1000 LLC misses) and a PEBS buffer, specified by a virtual address stored in a model-specific register called IA32_DS_AREA. Whenever the trigger condition is met, PEBS refers to IA32_DS_AREA to obtain the virtual address of the PEBS buffer, writes the process ID and the accessed virtual address to the PEBS buffer.

Unfortunately, with the current hardware, it is not possible for a hypervisor to monitor guests using PEBS [30, 45, 47]. This limitation is because there is only one IA32_DS_AREA register per CPU, which is shared by both the VM and the hypervisor. When the VM execution triggers the specified condition, the hardware logs to the virtual address of the VM (at the address stored in IA32_DS_AREA) instead of the hypervisorfis address space. This could potentially corrupt the VM (since the VM does not expect such writes) and makes it difficult for the hypervisor to access the results since they reside in the VM's address space.

Thermostat. Thermostat [2] samples a small fraction (5%) of huge pages, splits them into base pages, tracks memory accesses within the base pages, and coalesces them back. The paper does not explain why choosing such a small fraction of sampling, and we suspect this is due to the overhead of splitting/coalescing pages (as detailed in §3.3.1).

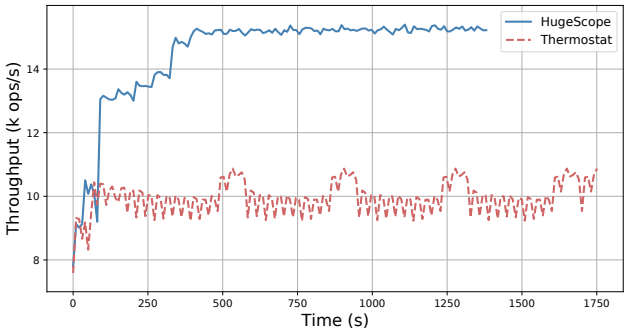

Figure 2: Performance impact with HugeScope vs. Thermostat (higher is better).

Sampling always leads to inaccuracies, which in turn results in systems making suboptimal decisions. To demonstrate this, we implement the sampling approach used in Thermostat and applied it to a tiered memory system (HS-TMM-sampling). We compare this with another tiered memory system with accurate memory access tracking (HS-TMM). The VM accesses a 20GB Redis database with a hotspot distribution, with 8GB of fast memory, and 120GB of slow memory.

As shown in Figure 2, HS-TMM, based on accurate memory access tracking, quickly reaches peak performance. HS-TMM-sampling reduces monitoring overhead but its inaccuracy leads to suboptimal page placement decisions, leading to, on average, a performance decrease of 26%.

***Finding #2.* Sampling a small portion of huge pages, as used by Thermostat, is highly ineffective.**

3 The HugeScope System

We presents HugeScope, a lightweight, effective, and generic solution to the hot bloat problem under virtualization. This section presents the design goal of HugeScope (§3.1), an overview of HugeScope (§3.2), the design of each individual component (§3.3, § 3.4, §3.5), and concludes with its implementation (§3.6).

3.1 Design Goals and Non-goals

We design HugeScope to meet the following design goals.

- **Lightweight.** HugeScope's runtime overhead must be minimal and must not offset the performance advantages enabled by huge pages. Otherwise, HugeScope is not useful, since one can trivially just use base pages to resolve the hot bloat problem (as in HotBox [7]).
- **Accurate and complete memory access information for informed decision making.** As demonstrated in§ 2.5.2, HugeScope must provide accurate and complete memory access information to enable smart and well-informed decision-making, in determining whether to split or preserve huge pages.

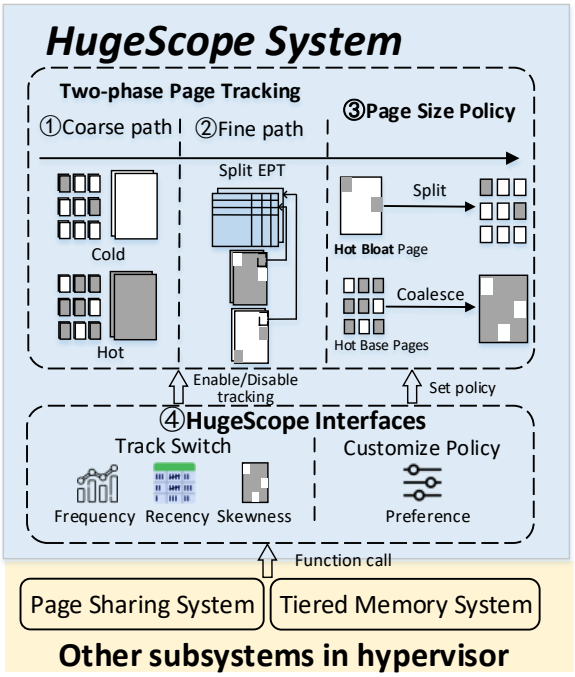

Figure 3: The overview of HugeScope.

- **Generality.** The design of several important mechanisms needs to consider hot bloat (§2.4). We aim to design HugeScope as a one-stop solution for the hot bloat problem; HugeScope should be general enough to be applicable across a wide range of scenarios.
- **Modularity.** To enable its easy integration with other systems, HugeScope should be an isolated and self-contained module, and expose a set of well-defined and flexible interfaces for interaction.

Non-goals. HugeScope is specifically designed for tackling hot bloat. Other issues incurred by huge pages, such as external memory fragmentation and unfair allocation of huge pages across different VMs [24, 32], are out of scope. As further discussed in §3.4, HugeScope only provides mechanisms to address hot bloat, and does not concern with particular policies (*e.g.*, the threshold to determine whether a page is hot). Such policies are and, as evident in §3.6, should be determined outside HugeScope.

3.2 Overview

Figure 3 depicts the architecture and workflow of HugeScope. ① Initially, HugeScope enters the first phase of memory access tracking by monitoring all system pages without any page splitting or coalescing. ② Afterwards, HugeScope enters the second phase, where it applies lightweight page table splitting and coalescing to obtain the access skewness of hot huge pages. ③ Next, HugeScope addresses the hot bloat problem with a default page size policy that considers page hotness and skewness. ④ HugeScope offers flexible and modular interfaces to facilitate seamless

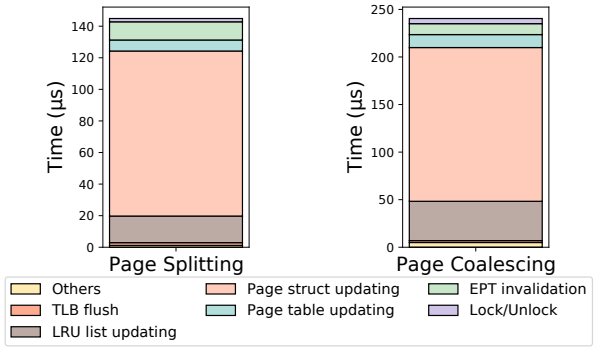

Figure 4: Overhead breakdown of page splitting (split_huge_page()) and coalescing (collapse_huge_page()) in KVM.

enhancements of other subsystems (*e.g.*tiered memory management and page sharing).

3.3 Lightweight and precise access tracking

This subsection first presents our observation regarding the challenges (§3.3.1) and opportunities (§3.3.2) under virtualization for access tracking. Next, we present how HugeScope overcomes the challenges and exploits the opportunities with a two-phase page tracking approach (§3.3.3), that is both lightweight and precise. We conclude by discussing the correctness of our methods (§3.3.4).

3.3.1 Challenges under virtualization

Challenges: prohibitive overhead for page splitting and coalescing. As discussed in §2.5.2, all the existing techniques for access tracking, including the PEBS-based ones, do not work for the scenario HugeScope targets. With the commodity hardware, HugeScope must resort to a paging-based mechanism for access tracking.

Intuitively, to obtain the access skewness within a huge page, one can split the huge page into base pages, and track memory accesses to these base pages with a paging-based mechanism. After the access tracking period ends, the base pages can be coalesced back into a huge page to continue enjoying the performance advantages of huge pages.

Unfortunately, as shown in Figure 4, we find that the overhead of performing splitting and coalescing is unacceptable, costing 145 and 240 μs, respectively. Furthermore, both operations require invalidating the EPT of the VMs. This causes expensive VM exits to rebuild EPT when these pages are accessed again (just in the near future for those active pages).

We note that the reported number already reflects an optimization we perform to save a large portion of the time for page coalescing, which originally takes 665 μs. Specifically, the original page coalescing function provided by Linux (even on the latest 6.7 version) always allocates a new huge page and copies the content of the base pages to this new huge page, even if the base pages are contiguous and aligned

Table 4: Modifications to EPT and GPT during the whole execution

Benchmark	EPT modifications	GPT modifications
429.mcf	4	835
657.xz	12	11201
GAPBS-pr	2	31049
GAPBS-bc	3	30169
Graph500	4	9680
Redis	4	10435
Mongodb	2	12765

on the huge page boundary. Our optimization performs an in-place merging whenever possible.

Overhead analysis. The overhead incurred by splitting the page table (*i.e.*, page table updating plus TLB flush), which is the only required operation for tracking memory access, is only 6.0% and 6.5% for splitting and coalescing, respectively. Instead, most of the overhead in page splitting/coalescing is due to modification to kernel metadata, especially the page descriptor (*i.e.*, page struct) and the page reclamation structures (*i.e.*, LRU list). Thus, a key challenge HugeScope must overcome to enable a lightweight, precise, and base page granularity memory access tracking, in the presence of prohibitive page splitting/coalescing overhead.

3.3.2 Opportunities with virtualization

HugeScope is enabled by two characteristics that exist in both Type 1 and and Type 2 hypervisors.

***Opportunity #1*: A hypervisor rarely accesses and modifies extended page tables.** To support virtualization, in addition to guest page tables, modern hardware provides another level of address mapping (which maps guest-physical addresses to host-physical addresses) with extended page tables (EPTs), that are maintained by hypervisors [20]. This additional level of address mapping introduces unique opportunities for tracking memory accesses.

First, after an extended page table entry (EPTE) is built, a hypervisor rarely modifies it. This is because the guest page tables (GPTs) absorb all page table modification requests from applications (*e.g.*, mapping and unmapping pages, changing permission bits). The access of EPT will only appear in the hypervisor management request. Table 4 shows the results that confirm the above finding with our workloads in KVM. We do not evaluate this on a type 1 hypervisor (*e.g.*, Xen [5]), but prior work reports similar findings [27].

***Opportunity #2*: Even if a hypervisor accesses/ modifies EPTs, it always does so through a few stable interfaces.** We inspect the latest version of the KVM and Xen and find that 1) they access/modify EPTs through a small number of functions (*e.g.*, 11 in KVM). 2) This set of interfaces is stable. For KVM and Xen, no functions have been added to/removed from this set for the past 9 and 7 years, respectively.

3.3.3 Two-phase page tracking

Observing the challenges and opportunities discussed above, following Rainbow [44], HugeScope employs a two-phase page tracking approach, consisting of a coarse phase and a fine phase. Unlike Raindow, HugeScope's memory tracking approach does not require hardware modification while is still lightweight and accurate.

Coarse phase: classifying hot/cold pages without page splitting and coalescing. To resolve the hot bloat problem, one should (1) split the hot unbalanced huge pages and (2) coalesce the base pages that have even access. Therefore, in the initial coarse phase, HugeScope (1) obtains the access frequency and recency of all pages (both huge pages and base pages) (2) classifies huge pages into hot and cold. The above requirements can be met without the expensive page splitting and coalescing operations.

In the coarse phase, HugeScope uses the A/D bits in the EPTEs of the pages to obtain the required information. Specifically, when an iteration of memory access tracking starts, HugeScope clears the A/D bits of the EPTEs and flushes the TLB. When a TLB miss occurs, the hardware MMU sets the A/D bits of the corresponding EPT entries. Upon the completion of the iteration, HugeScope scans the A/D bits of the PTEs to obtain the memory access tracking information.

Fine phase: obtaining access skewness of hot huge pages with lightweight page splitting and coalescing. In the next fine phase, thanks to the classification in the coarse path stage, HugeScope only needs to obtain the access skewness of hot huge pages. HugeScope splits all the hot huge pages when the fine phase starts and coalesces the corresponding base pages back into huge pages when the stage ends (so that the system can continue to benefit from the performance advantages of huge pages). To further reduce performance overhead, HugeScope adopts a page splitting/coalescing approach that avoids costly modifications to hypervisor metadata.

As discussed in §3.3.1, the overhead incurred by the current page splitting/coalescing is unacceptable, due to costly modifications to hypervisor metadata. To overcome this challenge, in the fine path stage, HugeScope only modifies the EPT (specifically, entries in the last-level page table) of a huge page under memory access tracking to perform both splitting and coalescing operations, and does not perform any modifications to the metadata structures. This approach causes a temporary inconsistency between the EPT and the hypervisor metadata; the inconsistency occurs when HugeScope splits the page table (at the beginning of the fine path stage) and disappears when HugeScope coalesces the page table (at the end of the fine path page).

This approach is effective, since a hypervisor rarely accesses and modifies EPTs (§3.3.2). As a result, in most cases, the hypervisor will never observe such inconsistency (since it does not access or modify EPTs). Hence, in almost all cases,

HugeScope does not need to employ the special fallback mechanism to address the inconsistency.

3.3.4 Ensuring Correctness

To ensure correctness in the rare cases (where a hypervisor does observe the inconsistency by accessing or modifying EPTs during the address tracking period), HugeScope leverages the observation that a hypervisor only accesses EPTs through a few stable interfaces (§3.3.2). Therefore, HugeScope intercepts these few functions by adding a hook in them to detect if they access or modify EPTs that may expose the inconsistency. If so, HugeScope coalesces the page table back to its original form, eliminating the inconsistency. This approach minimizes the intrusion into the rest of the hypervisor (since only a few functions are modified) and is easy to maintain (since the interfaces are stable).

A downside of this approach is that, in the aforementioned rare cases, HugeScope cannot track access skewness within the hot huge pages since it can no longer be split due to inconsistency. We do not observe that this causes an issue in our workload (since the rare case hardly occurs). If this is a concern, one could fix this issue by splitting the page tables of the huge pages back to base pages again, after the hypervisor finishes accessing the EPTs, at the cost of more intrusive code modifications to the hypervisor.

In addition, to ensure correctness, HugeScope reuses the page splitting/coalescing code in KVM / Linux (except that HugeScope does not modify the kernel metadata as these functions). The code splits or merges the page table entries only after the new entries have been properly set up, and the code performs an atomic memory write to make the page table use the new entries. This ensures, e.g., the page table walker to always see a valid entry.

Finally, we note that HugeScope does not change the mapping between the guest physical address and the host physical address. This, together with the aforementioned approaches to resolve metadata inconsistencies and split/coalesce pages, ensures HugeScope operate correctly.

3.4 Page Size Policy

The page size policy for handling hot bloat must consider the memory pressure, hotness, and skewness of huge pages. The default policy of HugeScope fully considers these metrics to address hot bloat, *i.e.*, to split hot bloat huge pages. Additionally, with generality in mind, HugeScope also supports the modification of this policy by backend subsystems (*e.g.*, tiered memory management and page sharing) to fulfill a variety of requirements (as detailed in §3.5).

When using the default policy, the backend needs to provide huge hot pages and hot page sizes (N_{hot}, to be determined by the backend after the coarse path), as well as the memory hot page waterline (N_{line}, indicating the expected size of hot pages by the backend). We then introduce the hot page pressure metric (HP) to describe the degree to which the actual size of VM's hot pages exceeds the memory hot page waterline. Essentially, HP represents the trade-off between the usage of huge pages and the memory pressure. The HP is calculated as follows:

$$HP = \begin{cases} N_{hot} - N_{line} & initialization \\ HP - PSR_i \times S_{huge} & splitting\ huge\ page\ i \\ HP + PSR_i \times S_{huge} & coalescing\ huge\ region\ i \end{cases}$$

Following [7], we use PSR to quantify the access skewness within a huge page. PSR is defined as follows: $PSR = 1 - \frac{N_b}{N_t}$, where N_b is the number of base pages accessed, and N_t is the total number of base pages (N_t is 512 for 2MB huge pages). PSR_i denotes the PSR of the huge page i and S_{huge} represents the size of a huge page.

After fine path monitoring, HP is initialized. The HP adjusts along with page splitting and coalescing. Specifically, page splitting can expose the cold base page regions within a hot huge page. As those cold pages are no longer treated as hot, the HP should be decreased. In contrast, page coalescing may cause the cold base pages to coalesce into a hot huge page, and the HP should be increased.

When $HP > 0$, a series of huge pages should be selected to split. We prioritize splitting the pages based on the PSR, from large to small. This is because a high PSR implies fewer visited base page regions and, thus, lesser speedup in address translation. When $HP < 0$, a series of huge page regions should be coalesced. we prioritize coalescing pages based on the PSR from small to large, giving priority to the huge page region containing more accessed base pages.

3.5 Flexible and Modular Interface

As a a general solution, HugeScope is not coupled with any specific scenario, but instead provides a flexible interface for other memory management subsystems to call. Figure 5 shows the pseudo-code of HS-TMM (§3.6) to illustrate the use of HugeScope interfaces.

For two-phase page tracking, HugeScope exposes switch interfaces for both coarse and fine paths (i.e., enable/disable_coarse/fine_path). HugeScope places the access data of each page in the heap space, and each page's access data are represented by a 64-bit entry (access entry). The first 32 bits of this entry represent the dirty bit information of the page in the last 32 cycles, and the last 32 bits represent the access bit information of the page in the last 32 cycles. Each base page region in a huge page also has an access entry. Thus, for a huge page, we can obtain its recency, frequency, and skewness information through access entries.

For the page splitting and coalescing policy, HugeScope provides a default strategy (invoked by hs_page_size_policy) that can split and balance huge pages reasonably. The backend subsystems need to provide HugeScope with the hot

```c
/* vm_desc is the VM descriptor */
void hs_tmm(struct VM* vm_desc, int fast_mem_size,
            int profiling_times, int coarse_path_sleeping_interval,
            int fine_path_sleeping_interval){
    // Coarse phase tracking
    for(i = 0; i < profiling_times; i++){
        // `hs_start_coarse_path` clears A/D bits for all VM pages.
        hs_start_coarse_path(vm_desc);
        msleep(coarse_path_sleeping_interval);
        // `hs_end_coarse_path` checks A/D bits for all VM pages
        //  and increases a per-page counter array in vm_desc
        hs_end_coarse_path(vm_desc);
    }

    size_t* hot_huge_pages, hot_base_pages;
    // vTmm iterates the per-page counter array in vm_desc
    // to obtain hot pages for huge and base pages.
    vtmm_get_hot_pages(vm_desc, hot_base_pages, hot_huge_pages);

    // `hs_start_fine_path` split EPTs for each huge page in
    //  hot_huge_pages to track accesses to their base pages
    hs_start_fine_path(vm_desc, hot_huge_pages);
    msleep(fine_path_sleeping_interval);
    // `hs_end_fine_path` checks the A/D bits of each base page
    //  and increases a per-page counter array in vm_desc
    hs_end_fine_path(vm_desc);

    // Invoke the default HugeScope paging policy
    hs_page_size_policy(vm_desc, hot_huge_pages, fast_mem_size);

    // vTMM updates page placement in fast/slow memory for the VM.
    vtmm_page_placement(vm_desc, hot_huge_pages, hot_base_pages);
}
```

Figure 5: Pseudo code of HS-TMM. All the functions starting with 'hs_' are HugeScope interfaces.

pages and memory hot page waterline as input to this default strategy. In addition, this interface also supports customized strategies from the backend. The backend can specify the candidate page set for page splitting and coalescing, which will occur only for pages in this set.

3.6 Implementation

We implement HugeScope as a kernel module on top of KVM [8] (and thus QEMU 3.1.0 [9]) in the 5.4.142 Linux kernel with 2000 lines of C codes. This is the version where vTMM [39] (a tiered memory management system previously developed by us) is built and we implemented HugeScope on the same version to ensure fair eveluation.

We believe that the implementation of HugeScope depends little on the hypervisor and host OS versions. We have also ported HugeScope to Qemu 7.1.0; the porting does not require modifying any code. We also check the code of the latest Linux (6.9) and expect little porting effort (since the interfaces to access/modify EPTs do not change).

Access Entry Placement. HugeScope organizes the access information of the VMs into a page table structure called the access metadata table (AMT). HugeScope uses the host physical address (HPA) to index the AMT, as it can directly obtain the HPA of a page during page table scanning, avoiding additional overhead and allowing multiple VMs to share the same AMT. This implementation effectively reduces the space overhead of the HugeScope metadata. The walking of AMT is similar to that of a regular page table, with the only difference being that for a huge page, we need to save the metadata page index for its next-level base pages and also save the AMT entry for the huge page itself. Therefore, the PMD page for AMT occupies 8KB, and each PMD entry includes the address index for the next level and the AMT entry for the huge page.

Virtualization-Friendly Page Splitting and Coalescing. HugeScope needs to split huge pages or coalesce base pages. However, the implementation of these functionalities under KVM / Linux is not designed for virtualized environment. Specifically, the implementation invalidates EPT entries. As a result, the next access to these pages causes a VM exit, which severely damages the performance of the VMs. To minimize such overhead, HugeScope proposes a virtualization-friendly page splitting and collapsing mechanism. Upon page splitting (or coalescing), HugeScope actively refills the EPT of base pages (or the huge page) after changes are performed to the VM process page table.

Tiered memory management with HugeScope (HS-TMM). vTMM is a virtualization-customized tied memory management system, which efficiently utilizes fast memory through page tracking, page classification, and page migration. However, vTMM does not address the hot bloat problem and is only applicable to either base page systems or huge page systems. Following vTMM, we implemented HS-TMM using the interfaces provided by HugeScope. In the page tracking phase, HS-TMM directly uses the coarse path interface for page table tracking. After page classification, HS-TMM uses the fine path interface to monitor hot huge pages at the base page granularity. Then, HS-TMM employs HugeScope's default policy and sets the memory hot page waterline to the size of fast memory, ensuring that hot bloat will not cause hot pages to be placed in slow memory and maximizing the proportion of huge pages in fast memory. HS-TMM then performs the correct page placement, placing hot pages in fast memory and cold pages in slow memory.

Page Sharing with HugeScope (HS-Share). Ingens attempts to balance the usage of huge pages and page sharing by distinguishing between frequently and infrequently accessed huge pages based on their A/D bits. Only infrequently accessed huge pages are split for page sharing. Inspired by Ingens, we implemented HS-Share using the interfaces provided by HugeScope. Similarly to HS-TMM, HS-Share first obtains the hotness and skewness information of the huge pages through the coarse path and fine path interfaces of HugeScope. However, the default policy in HugeScope (that is, splitting all unbalanced huge pages) needs to be slightly modified. This is because, in page sharing, both cold huge pages with mergeable areas and unbalanced huge pages need to be considered as splitting candidates. To customize the split policy, HS-Share only needs to disable the default policy and set the candidate page set through the policy interface provided by HugeScope.

Table 5: Performance improvement of huge pages to base pages in virtualization

Benchmarks	Local DRAM	NVM
429.mcf	10.50%	7.88%
657.xz	15.50%	5.75%
GUPS-bc	102.06%	10.42%
GUPS-pr	227.06%	9.36%
Redis	11.94%	5.54%
MongoDB	9.73%	4.92%
Graph500	19.35%	7.29%

4 Evaluation

Our evaluation answers the following questions:

- What is the cost and accuracy of the two-phase page tracking used by HugeScope? (§4.2)
- How much performance can improve with virtualization-friendly page splitting/ coalescing? (§4.3)
- How much performance improvement is brought by each HugeScope component? (§4.4)
- What improvement can be achieved using HugeScope in scenarios such as tiered memory management and page sharing? (§4.5, §4.6)

4.1 Experimental Setup

We evaluate HugeScope on a dual-socket Intel Cascade Lake-SP server with 24 cores/48 threads per socket at 2.2GHz. The VMs are configured with 8 cores. The host and guest OS run Ubuntu 18.04 with Linux kernel 5.4.142. We use local DRAM as fast memory and Intel Optane DC PMem as slow memory for the tiered memory system. We choose benchmarks with varying access patterns, including: 429.mcf from SPEC CPU 2006 [41] and 657.xz from SPEC CPU 2017 [40]; Graph 500 [11] running SSSP and BFS algorithms on a graph containing 2^{25} points, with a 20GB footprint; GAPBS [6] running BC and PR algorithms on a Kronecker graph with 2^{25} points and a 20GB footprint; Redis [28] and MongoDB [1] with YCSB [10] test suites, both configured with 20GB data, a 4KB value size and operations follow the hotspot distribution with a 1:1 read-update ratio. Unless otherwise specified, Redis and MongoDB use throughput as the performance metric. As shown in Table 5, all the evaluated benchmarks benefit from huge pages.

4.2 Page Tracking Efficiency

In this section, we compare HugeScope's page tracking mechanism with with the methods mentioned in 2.5.2. *Split scan* refers to using the Linux interface to split and coalesce pages. *Sampling scan* refers to sampling 5% of huge pages for splitting to track each iteration as Thermostat. For the page table scan-based methods, we set the interval for clearing A/D bits to once per second. *Zero scan* just scanning zero base pages as HawkEye. PEBS controls the frequency of hardware counter-overflow by adjusting the sampling period. We choose 50 (PEBS-50), 500 (PEBS-500) and 5000 (PEBS-5000) for testing. These values are selected to comprehensively illustrate the trade-off between the precision and the overhead of PEBS. We chose the sampling period being 50 to show that, with our workloads, even such a small sampling period is much less accurate than HugeScope. We do not further increase the sampling period beyond 5000 since a frequency of 5000 incurs less than 1% overhead. Since PEBS is not supported by virtualization, we test it in native mode.

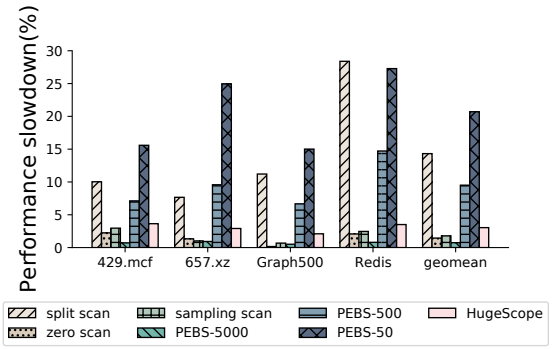

Figure 6: Performance slowdown of different monitors (lower is better).

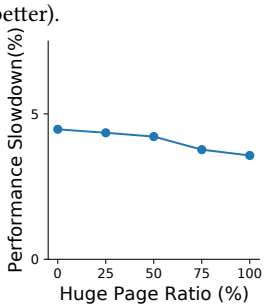

Figure 7: Performance slowdown of monitoring Redis under different huge page ratios

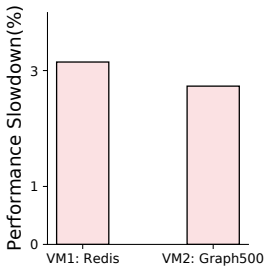

Figure 8: Performance slowdown of monitoring two co-run VMs.

4.2.1 Page Monitoring Overhead

To assess the performance impact caused by these monitoring mechanisms, we enabled monitoring for all benchmarks for 10 out of 20 seconds.

The results are shown in Figure 6. Due to space limitations, the unshown results are similar to those in the figure. The split scan incurs a much higher overhead due to costly splitting and coalescing operations. The sampling scan has an average performance penalty of 1.78% as only 5% of the huge pages are selected for monitoring. The overhead of zero page scanning is only 1.45%, as most pages are non-zero pages. The average performance overheads of PEBS-5000, PEBS-500, and PEBS-50 are 0.7%, 9.5%, and 20.7%, respectively. As precision increases, the overhead also increases. HugeScope tracks at the granularity of the base page with an average cost of 3.04%.

We then show the performance of HugeScope with re-

Figure 9: Performance slowdown caused by Linux splitting/coalescing and HugeScope fine path (lower is better).

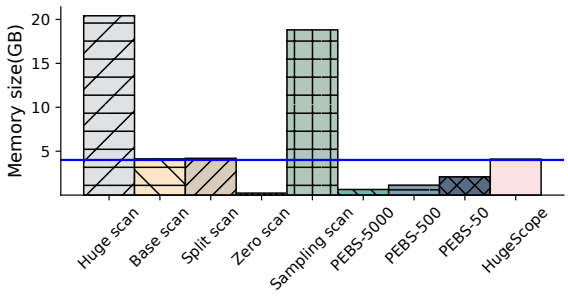

Figure 10: Real-time monitoring results on `Redis`. The blue horizontal line is the exact memory access size.

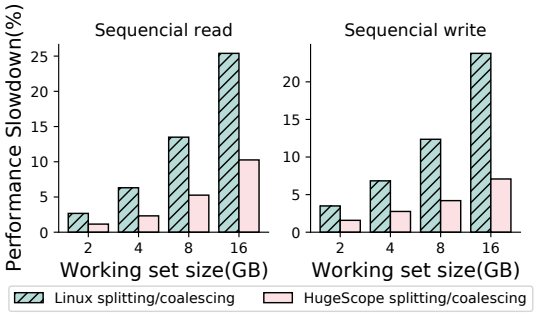

Figure 11: Performance slowdown due to page splitting/coalescing (lower is better).

spect to the number of huge pages and different numbers of VMs. Figure 7 shows the performance degradation caused by using HugeScope to monitor Redis with different huge page ratios. When there are less huge pages, monitoring will cause more overhead. This is because more PTEs of base pages need to be scanned. Figure 8 shows the performance slowdown of monitoring two co-run VMs. Two VM instances run Redis and Graph500 respectively, and HugeScope monitors them at the same time. The results show that HugeScope does not cause more overhead in multi-VM scenarios.

We then delve further into the performance benefits brought about by the fine path of HugeScope. We compare the fine path with the traditional Linux interfaces. To illustrate the relationship between the overhead and the number of pages, we used a sequential access microbenchmark with a 1:1 read/write ratio and varied the WSS from 2 GB to 16 GB. We activate the Linux page splitting/coalescing and HugeScope fine path every 10 out of 20 seconds. As shown in Figure 9, the performance degradation of traditional interfaces is proportional to WSS. A 16GB program causes a performance degradation of up to 25.39%. In contrast, the HugeScope fine path performs only a light change in EPT, with a performance overhead of less than 1.5%.

At the beginning and end of the fine-path monitoring, TLB shootdowns are performed due to page table entries modification. Our experiments show that the performance impact is almost negligible for all applications (<0.5%).

4.2.2 Page Monitoring Accuracy

To assess the tracking accuracy, we use database benchmarks as workloads due to their predictable RSS. We test real-time monitoring of the workload. Real-time monitoring involves monitoring every 10 out of 20 seconds and taking the average value of the memory accessed. We only show the result of `Redis` in Figure 10. The result of `MongoDB` is similar to `Redis`. Base scan and huge scan, respectively, refer to the system building only base page or huge page mappings.

Due to hot bloat, the results of the huge scan imply that most memory is accessed, which is inaccurate. The sampling scan also suffers from hot bloat, as it only demotes a small partition of huge pages for monitoring. In contrast, the split scan and HugeScope can accurately identify the accessed memory regions as the base scan. PEBS always has a gap with the baseline because it is based on sampling. The zero scan only counts zero pages, which does not provide an accurate reflection of the VM's exact memory access behavior.

4.3 Page Splitting and Coalescing

HugeScope uses a virtualization-friendly page splitting and coalescing mechanism after policy decision which proactively refills EPT entries to reduce the number of VM exits (§3.6). To illustrate the relationship between the overhead and the number of pages, we use a sequential read microbenchmark with varying WSS. We split and collapse all pages of the workload every 20s. As shown in Figure 11, Linux interfaces cause a slowdown of up to 25.39% for the 16GB program. With refilling, it is reduced to 10.26%. The performance slowdown is proportional to the WSS, as increasing the WSS results in more VM exits and more page states to modify. Table 6 shows that the VM exits caused by the virtualization-friendly page splitting and collapsing of HugeScope are much lower than those caused by Linux interfaces.

The refill operation incurs less than 1% overhead for a 16GB program, but insteain turn d signficantly improves overall performance, as discussed above. The refill overhead increases linearly with the memory size since the work performed for each page (i.e., refilling an EPT entry) does not increase with memory size.

Table 6: VM exits caused by page splitting and coalescing

WSS(GB)	Linux		HugeScope	
	Read	Write	Read	Write
2	545715	547324	1197	825
4	1093350	1083551	1215	943
8	2185320	2163738	966	977
16	4355483	4285644	915	915

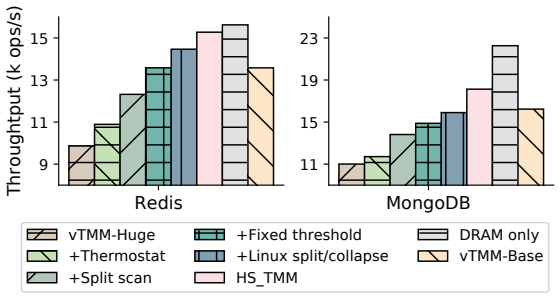

Figure 12: Ablation study of HugeScope (higher is better).

4.4 Ablation Study

Figure 12 shows a breakdown of the performance gains of three components (*i.e.*, two-phase page tracking, page size policy, and virtualization-friendly page splitting and coalescing) of HugeScope. We replace them with split scanning, Thermostat (sampling scanning), a fixed threshold policy, and Linux splitting/coalescing interfaces separately. We show the results in a tiered memory system (*HS-TMM*) and use *vTMM-Huge* as the baseline. DRAM only indicates that the VM's memory consists entirely of fast memory to demonstrate optimal performance metrics. All other configurations use 12GB (60% of WSS) and 38Gb of fast and slow memory, respectively.

HS-TMM outperforms *+Thermostat* as the sampling scanning cannot obtain accurate real-time access information, leading to the hypervisor placing infrequently accessed base page regions into fast memory. *HS-TMM* outperforms *+Split scanning* as a fine path reduces VM exits and metadata modifying overheads to a minimum. *+Fixed threshold* selects a fixed threshold, where a huge page ensures that more than 256 base page regions are accessed, but it cannot achieve the optimal tradeoff between address translation overhead and huge page utilization. *+Linux split/collapse* produces VM-exits linearly correlated to the number of pages, resulting in performance degradation. HS-TMM results in minimum VM exits and achieving performances matching those of DRAM only.

4.5 Case Study 1: Tiered Memory

For the tiered memory system, we compare HS-TMM with the state of the art, vTMM, of pure huge and base page management (vTMM-Huge and vTMM-Base). We run a VM with a total memory of 50GB, which exceeds the RSS of all benchmarks. We employ the same configuration as vTMM, that is, monitoring and adjustment of pages every 60 seconds. We adjust the size of the fast memory to demonstrate performance improvement under different memory pressures.

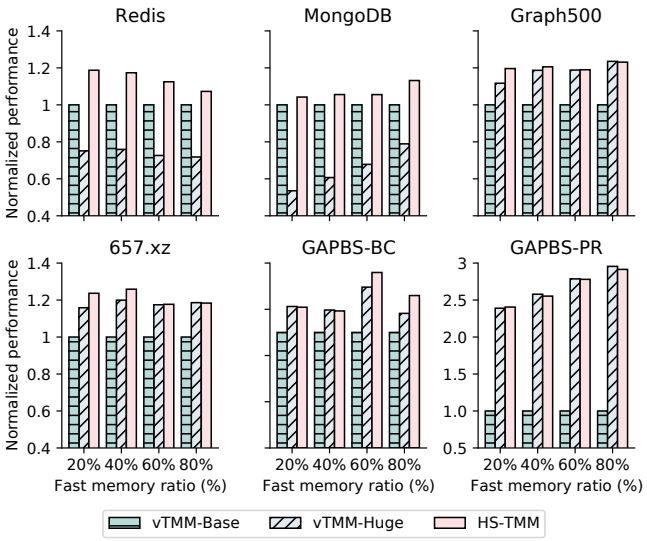

Figure 13: The normalized performance of different fast memory ratios for vTMM-Base, vTMM-Huge, and HS-TMM (higher is better). The x-axis represents the ratio of fast memory to the WSS.

As shown in Figure 13, `Redis` and `MongoDB` suffer serious hot bloat, so the performance of vTMM-Huge is lower than that of vTMM-Base. HS-TMM achieves optimal performance, as it addresses hot bloat while retaining balanced huge pages for faster address translation. `Graph500` and `657.xz` are slightly affected by hot bloat. The performance of vTMM-Huge is the same as that of HS-TMM when the DRAM ratio exceeded 60% since the fast memory can accommodate all hot pages. hot bloat occurs when the DRAM ratio is less than 40%. HS-TMM achieves the best performance. vTMM-Huge offers performance improvement because the address translation overhead overcomes the fast memory utilization. For `GAPBS-BC`, HS-TMM outperforms vTMM-Huge when the fast memory is more than 60%, and is the same as vTMM-Huge when the fast memory is less than 40%. This is because the most hot huge pages are balanced huge pages. When increasing fast memory, an unbalanced huge page has the opportunity to harm the utilization of fast memory. `GAPBS-PR` always has no hot bloat, so the performance of HS-TMM is close to that of vTMM-Huge.

Figure 14 shows the latency metric of Redis. Same as throughput metric, HS-TMM consistently achieves the best average latency across all memory configurations.

4.6 Case Study 2: Page Sharing System

To evaluate the efficiency of the sharing system, we perform experiments on three VMs that run database benchmarks, collectively utilizing 60 GB of memory, as shown in Table 7. Initially, all Redis instances store the same content, but we feed them with different requests. For HS-Share and Ingens, we set the frequency of page monitoring and page classifica-

Figure 14: Latency Comparison for Redis with a 1:1 READ and UPDATE Ratio (lower is better). The x-axis represents the ratio of fast memory to the WSS.

Table 7: Memory savings and performance of `Redis`

Policy	Memory saving	Avg. Performance	Huge page ratio
Huge page Only	321MB (0.52%)	1	100%
Base page Only	28916MB (47.1%)	0.889	0%
KSM	28742MB (46.8%)	0.902	4%
Ingens	785MB (1.28%)	0.990	98%
THP Shrinker	384MB (0.56%)	0.986	96%
HS-Share	25981MB (42.3%)	0.964	25%

tion to 10 seconds every 30 seconds. We record the maximum shared memory size for each method while also measuring the average database throughput after enabling each shared memory mechanism. Huge page share can hardly share memory but has the highest performance because none of the huge pages is demoted to share. Linux KSM can share all base pages with identical content, saving 46.8% of total memory, but all three VMs have a ∼10% performance penalty. Ingens suffers from hot bloat and only saves 1.28% of memory. THP Shrinker can hardly share zero pages, indicating that the running application rarely initializes the data to zero and then no longer accesses it. HS-Share achieves a better trade-off between address translation overhead and memory savings, and ends up sharing 42.3% of the memory, with an average performance loss of 3.6%.

5 Discussion

PEBS vs. HugeScope. While the PEBS issue discussed in §2.5.2 is fixable, it is difficult to predict when/if the hardware vendor will fix it. Furthermore, performing the fix would be non-trivial given the complexity of hardware virtualization support and performance monitoring mechanisms.

In addition, with our workloads, our experience suggests that the two-phase mechanism works better than PEBS. We report the overhead (§4.2.1) and accuracy (§4.2.2) of using PEBS to track base pages on Redis (in a native environment). We found that even with a sampling period of 50, PEBS is much less accurate than HugeScope (Figure 10), and it incurs an overhead of 20.7% (vs 3.04% with HugeScope)

Limitations. When the physical memory is insufficient, the hypervisor uses swapping and ballooning to reclaim memory from VMs. These techniques involve modifications to the EPT of the VM. The activation of the fine path can potentially cause many inconsistencies in such scenarios. However, our mechanism aims to enhance overall memory utilization to obviate the need for costly swapping and ballooning operations. So in cases where the physical memory remains insufficient and swapping and ballooning are imminent, we just selectively filter pages for management, and focus only monitor the pages not being swapped out.

6 Conclusion

This paper presents HugeScope, a lightweight, effective, generic, and modular system to manage huge pages to address hot bloat in a hypervisor. A key insight in our design is to exploit the other level of address indirection brought by virtualization. This level of address indirection makes EPT highly stable, thereby enabling HugeScope. Specifically, HugeScope uses two-phase page tracking to achieve fine-grained access monitoring and splitting/coalescing pages based on memory pressure as well as recency, frequency and skewness of memory accesses. HugeScope provides flexible interfaces and we adopt it on a tiered memory management system (HS-TMM) and page sharing system (HS-Share). Our evaluation shows that HugeScope incurs less than 4% overhead, and By addressing hot bloat, HS-TMM improves performance by up to 61% over vTMM while HS-Share saves 41% more memory than Ingens while offering comparable performance. Our artifact is publicly available at https://github.com/TELOS-syslab/hugescope-atc24-ae.

Our experience additionally reveals two findings that could apply beyond hot bloat. The first is the mechanism to efficiently split/merge the page table entries EPT with inconsistency resolution. We envision that future research could benefit from this technique in solving other problems that involve huge pages in a virtualized scenario. The second is the observation or confirmation that the interfaces to access EPT are stable. This provides the basis for future work to fihookfi on these interfaces to solve problems.

Acknowledgment

We thank the anonymous reviewers for their insightful and constructive comments. The research is supported in part by the National Key R&D Program of China under Grant No. 2022YFB4500701, and by the National Science Foundation of China (Nos. 62032008, 62032001,62372011). Diyu Zhou is the corresponding author.

References

[1] M. administration documentation. Mongodb. https://www.mongodb.com/, 2023.

[2] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2017.

[3] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, et al. Spin-transfer torque magnetic random access memory (stt-mram). *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):1–35, 2013.

[4] R. Ausavarungnirun, T. Merrifield, J. Gandhi, and C. J. Rossbach. PRISM: architectural support for variable-granularity memory metadata. In V. Sarkar and H. Kim, editors, *PACT '20: International Conference on Parallel Architectures and Compilation Techniques, Virtual Event, GA, USA, October 3-7, 2020*, pages 441–454. ACM, 2020. doi: 10.1145/3410463.3414630. URL https://doi.org/10.1145/3410463.3414630.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In M. L. Scott and L. L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177. ACM, 2003. doi: 10.1145/945445.945462. URL https://doi.org/10.1145/945445.945462.

[6] S. Beamer, K. Asanovi, and D. Patterson. The gap benchmark suite. *arXiv e-prints*, 2015. http://arxiv.org/abs/1508.03619.

[7] S. Bergman, P. Faldu, B. Grot, L. Vilanova, and M. Silberstein. Reconsidering OS memory optimizations in the presence of disaggregated memory. In M. Lippautz and D. Chisnall, editors, *ISMM '22: ACM SIGPLAN International Symposium on Memory Management, San Diego, CA, USA, 14 June 2022*, pages 1–14. ACM, 2022. doi: 10.1145/3520263.3534650. URL https://doi.org/10.1145/3520263.3534650.

[8] K. community. Kernel virtual machine. https://www.linux-kvm.org/page/Main_Page, 2023.

[9] Q. community. Qemu. https://www.qemu.org/, 2023.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, 2010.

[11] G. developers. Graph500. http://graph500.org/, 2023.

[12] L. developers. Transparent hugepage support. https://www.kernel.org/doc/Documentation/admin-guide/mm/transhuge.rst, 2023.

[13] L. developers. Kernel samepage merging. https://www.kernel.org/doc/Documentation/admin-guide/mm/ksm.rst, 2023.

[14] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. G. Melhem. Supporting superpages in non-contiguous physical memory. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 223–234. IEEE Computer Society, 2015. doi: 10.1109/HPCA.2015.7056035. URL https://doi.org/10.1109/HPCA.2015.7056035.

[15] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *SIGARCH Comput. Archit. News*, 40(1):37–48, 2012.

[16] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee. Proactively breaking large pages to improve memory overcommitment performance in vmware esxi. In A. Gavrilovska, A. D. Brown, and B. Steensgaard, editors, *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Istanbul, Turkey, March 14-15, 2015*, pages 39–51. ACM, 2015. doi: 10.1145/2731186.2731187. URL https://doi.org/10.1145/2731186.2731187.

[17] T. Hirofuchi and R. Takano. Raminate: Hypervisor-based virtualization for hybrid main memory systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 112–125, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450345255. doi: 10.1145/2987550.2987570. URL https://doi.org/10.1145/2987550.2987570.

[18] I. Inc. Intel optaneffl persistent memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html, 2023.

[19] Q. Inc. Using linux as hypervisor with kvm. https://indico.cern.ch/event/39755/attachments/797208/1092716/slides.pdf, 2023.

[20] Intel-Developers. Intel extended page table. https://www.intel.com.au/content/www/au/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html, 2023.

[21] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. Heteroos: OS design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 521–534. ACM, 2017. URL https://dl.acm.org/citation.cfm?id=3080245.

[22] K. Keeton. The machine: An architecture for memory-centric computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336062. doi: 10.1145/2768405.2768406. URL https://doi.org/10.1145/2768405.2768406.

[23] J. Kim, W. Choe, and J. Ahn. Exploring the design space of page management for multi-tiered memory systems. In I. Calciu and G. Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 715–728. USENIX Association, 2021. URL https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon.

[24] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 705–721, USA, 2016. USENIX Association. ISBN 9781931971331.

[25] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, 2009.

[26] T. Lee, S. K. Monga, C. Min, and Y. I. Eom. MEMTIS: efficient memory tiering with dynamic page classification and page size determination. In J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 17–34. ACM, 2023. doi: 10.1145/3600006.3613167. URL https://doi.org/10.1145/3600006.3613167.

[27] T. K. Lengyel. Stealthy monitoring with xen altp2m. https://xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m/, 2016.

[28] R. Ltd. Redis. https://redis.io/, 2023.

[29] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. O. Kanaujia, and P. Chauhan. TPP: transparent page placement for cxl-enabled tiered-memory. In T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 742–755. ACM, 2023. doi: 10.1145/3582016.3582063. URL https://doi.org/10.1145/3582016.3582063.

[30] G. Natapov. disable pebs on a guest entry. https://www.uwsg.indiana.edu/hypermail/linux/kernel/1208.1/02365.html, 2023.

[31] Y. Ozawa and T. Shinagawa. Exploiting sub-page write protection for VM live migration. In C. A. Ardagna, C. K. Chang, E. Daminai, R. Ranjan, Z. Wang, R. Ward, J. Zhang, and W. Zhang, editors, *14th IEEE International Conference on Cloud Computing, CLOUD 2021, Chicago, IL, USA, September 5-10, 2021*, pages 484–490. IEEE, 2021. doi: 10.1109/CLOUD53861.2021.00063. URL https://doi.org/10.1109/CLOUD53861.2021.00063.

[32] A. Panwar, S. Bansal, and K. Gopinath. Hawkeye: Efficient fine-grained OS support for huge pages. In I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 347–360. ACM, 2019. doi: 10.1145/3297858.3304064. URL https://doi.org/10.1145/3297858.3304064.

[33] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh. Perforated page: Supporting fragmented memory allocation for large pages. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages 913–925. IEEE, 2020. doi: 10.1109/ISCA45697.2020.00079. URL https://doi.org/10.1109/ISCA45697.2020.00079.

[34] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 1–12, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340342. doi: 10.1145/2830772.2830773. URL https://doi.org/10.1145/2830772.2830773.

[35] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.

[36] S. Roesch. Kernel samepage merging (ksm) usage at meta and future improvements. https://lpc.events/event/17/contributions/1625/attachments/1320/2649/KSM.pdf, 2023.

[37] S. Sha, J. Hu, Y. Luo, X. Wang, and Z. Wang. Huge page friendly virtualized memory management. *J. Comput. Sci. Technol.*, 35(2):433–452, 2020. doi: 10.1007/s11390-020-9693-0. URL https://doi.org/10.1007/s11390-020-9693-0.

[38] S. Sha, Y. Zhang, Y. Luo, X. Wang, and Z. Wang. Swift shadow paging (SSP): no write-protection but following TLB flushing. In B. L. Titzer, H. Xu, and I. Zhang, editors, *VEE '21: 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Virtual USA, April 16, 2021*, pages 29–42. ACM, 2021. doi: 10.1145/3453933.3454012. URL https://doi.org/10.1145/3453933.3454012.

[39] S. Sha, C. Li, Y. Luo, X. Wang, and Z. Wang. vtmm: Tiered memory management for virtual machines. In G. A. D. Luna, L. Querzoni, A. Fedorova, and D. Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, pages 283–297. ACM, 2023. doi: 10.1145/3552326.3587449. URL https://doi.org/10.1145/3552326.3587449.

[40] Standard Performance Evaluation Corp. Spec cpu 2017 benchmarks. http://www.spec.org/cpu2017, 2023.

[41] Standard Performance Evaluation Corporation. Spec cpu 2006 benchmarks. http://www.spec.org/cpu2006, 2023.

[42] C. A. Waldspurger. Memory resource management in vmware ESX server. In D. E. Culler and P. Druschel, editors, *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. USENIX Association, 2002. URL http://www.usenix.org/events/osdi02/tech/waldspurger.html.

[43] X. Wang, X. Liao, H. Liu, and H. Jin. Big data oriented hybrid memory systems. *Big Data Research*, 2018.

[44] X. Wang, H. Liu, X. Liao, J. Chen, H. Jin, Y. Zhang, L. Zheng, B. He, and S. Jiang. Supporting superpages and lightweight page migration in hybrid memory systems. *ACM Trans. Archit. Code Optim.*, 16(2): 11:1–11:26, 2019. doi: 10.1145/3310133. URL https://doi.org/10.1145/3310133.

[45] L. Xu. Prevent any host user from enabling pebs for profiling guest. https://lore.kernel.org/all/6c4bd247-1f81-4b43-9e21-012f831d26b8@linux.intel.com/T/, 2023.

[46] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.

[47] Y.-S. Yang. Avoid unnecessary work in guest filtering. https://lists.ubuntu.com/archives/kernel-team/2019-November/105645.html, 2023.

[48] A. Zhu. Linux thp shrinker. https://lwn.net/Articles/910993/, 2023.