Dynamic Heap Resizer for G1GC TeraHeap

Dimitris Basakidis, csd4960

Abstract

Modern memory-managed runtimes must efficiently balance memory allocation across dynamically changing workloads, especially when operating under constrained DRAM environments. TeraHeap enhances the G1 Garbage Collector by logically the Java heap into two regions: H1, optimized for GC-managed short-lived objects, and H2, used for longer-lived data with reduced GC overheads. However, TeraHeap statically relies on a fixed total heap size and does not adapt to variations in application memory pressure or available system resources.

In this work, we extend TeraHeap with a dynamic heap resizer that adjusts the heap size at runtime based on total lost compute cycles, combining both GC-induced CPU contention and I/O wait delays. Our resizer uses a lightweight policy engine that monitors metrics such as GC pause times, concurrent GC thread CPU competition, and per-thread I/O wait time to decide whether to grow or shrink the heap. This dynamic resizing mechanism is fully integrated into the OpenJDK's G1 TeraHeap implementation and requires no manual tuning or application-specific knowledge.

We evaluate our system on Lucene, a widely used full-text search engine, under varied memory-constrained scenarios. Our results show that the dynamic heap resizer improves adaptability and performance stability across workloads. It effectively reclaims memory during idle periods to reduce system pressure and reallocates heap space under load to maintain throughput, outperforming the baseline TeraHeap configuration in both responsiveness and efficiency.

1 Introduction

Modern high-performance full-text search engines like Apache Lucene [**?**] are increasingly used in large-scale data processing and indexing. Lucene, which runs on the JVM, relies on DRAM to store indexing structures, field and query caches, query evaluation data to deliver low-latency performance. However, data volumes continues to expand at a rapid rate, also growing memory requirements accordingly. This presents a challenge, as improvements in DRAM capacity have not kept up with the rate of data growth in recent years [9, 11]. As a result, the memory footprint of Lucene applications often becomes a bottleneck, particularly working with large indexes or real-time search workloads. The JVM heap and related managed memory components grow more slowly than the datasets Lucene is expected to handle, leading to pressure on memory management and performance tuning. In our experiments, we use TeraHeap as the underlying managed runtime system. TeraHeap introduces a dual heap design that places the primary heap (H1) in DRAM and a secondary heap (H2) on slower, larger-capacity memory such as SSD-backed storage. This separation enables the system to confine garbage collection (GC) activity to H1, avoiding the costly scanning and compaction of objects stored in the slow tier (H2). TeraHeap also uses a portion of DRAM as an I/O cache to speed up accesses to H2, allowing frequently accessed objects to benefit from DRAM latency.

In our setup, we use TeraHeap [6] as the runtime system and focus exclusively on varying the size of H1, the primary heap located in DRAM and managed by the G1 Garbage Collector. By increasing or decreasing the size of H1 at runtime, we evaluate how memory pressure affects application performance and garbage collection behavior. Since DRAM is shared between H1 and the page cache used to accelerate H2 accesses, changing the size of H1 effectively changes the partitioning of DRAM between managed memory and the cache for the slow-tier heap (H2). This setup allows us to study the trade-off between GC-managed heap size and available cache capacity for H2, and its impact on overall system performance.

In this work, we introduce a Dynamic Heap Resizer, which is a system that automatically decides at runtime whether to adjust the size of H1 or the H2 page cache. A Dynamic Heap Resizer is the first approach that enables practical and efficient deployment of hybrid heaps. Its design, which continuously rebalances DRAM allocation between H1 and the H2 cache, addresses some challenges in hybrid memory management.

Hybrid heap systems face significant challenges in effectively dividing a fixed DRAM budget between H1, the GC-managed heap, and the H2 page cache, which accelerates access to slow-tier memory. First, static DRAM partitioning leads to imbalanced performance trade-offs: (1) allocating too little memory to H1 results in high garbage collection overhead, while a small H2 cache increases I/O latency for accessing off-heap objects. (2) Existing approaches lack adaptability, as they require prior knowledge of application behavior or manual tuning to set optimal heap sizes. This is impractical for real-world applications with dynamic memory demands that change over time. (3) Even when DRAM allocations are adjusted, the impact of these changes is delayed due to OS memory reclamation and page cache resizing latencies, making current systems slow to react to shifting workload phases and limiting their responsiveness.

With the use of a Dynamic Heap Resizer, by continuously monitoring both garbage collection and I/O costs, decisions are made to resize H1 or adjust the effective cache space for H2, optimizing performance across diverse workload conditions. It employs an advanced adaptation mechanism based on a finite state machine (FSM) with wait states and direct transitions, enabling fast and stable adjustments as application memory requirements evolve. This approach ensures that DRAM is allocated efficiently between managed heap and page cache without requiring application-specific knowledge or manual intervention, improving overall system performance and responsiveness.

2 Motivation

Lucene is a widely used text search engine library that relies on managed runtimes for indexing and querying large datasets efficiently. TeraHeap, a state-of-the-art memory system, extends the managed heap beyond DRAM by partitioning it into a primary heap (H1) in DRAM and a secondary heap (H2) mapped to a slower memory tier such as NVMe SSD. This design reduces garbage collection (GC) overhead by limiting GC operations to H1, while H2 provides additional capacity for less frequently accessed data.

However, TeraHeap statically divides DRAM between H1 and the page cache for H2 at JVM launch time, which introduces significant limitations. If H1 is too small, Lucene experiences high GC overhead due to frequent collections, impacting query and indexing latency. Conversely, if the DRAM page cache for H2 is too small, accessing index segments stored in H2 incurs high I/O latency, degrading search performance. Because Lucene workloads exhibit varying memory demands, such as bulk indexing phases which require large heaps, while read-heavy query phases benefit from a larger page cache. Moreover, a static DRAM division is not able adapt to these changes, leading to suboptimal performance.

These limitations motivate our approach: dynamically resizing the primary heap (H1) at runtime in TeraHeap to better match Lucene's changing memory needs. By adjusting H1 size based on GC and I/O costs, we enable Lucene to maintain low GC overhead during indexing and benefit from a larger page cache during query processing, ultimately improving overall performance under DRAM constraints.

The figure 1 illustrates the breakdown of Lucene's total execution time into GC time and non-GC (other) time under different page cache configurations. The x-axis represents the fraction of DRAM allocated to the page cache (10%, 20%, 30%, 40%, etc.), while the y-axis shows the average total execution time in seconds for each configuration.

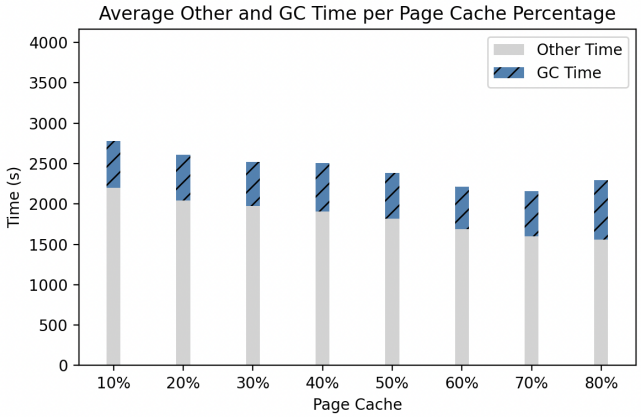

Figure 1: Graph of different H1/page cache partitionings

At low page cache allocations (10% to 30%), the GC time is minimal relative to total execution time. However, as the page cache share grows (40% to 60%), GC time steadily increases. This is because increasing page cache size reduces the available DRAM allocated to H1, resulting in a smaller primary heap. A smaller heap triggers more frequent garbage collections, leading to higher GC overhead. The grey part of each bar represents the time spent executing Lucene and system operations excluding GC. As the page cache share increases, the non-GC time decreases. This is because a larger page cache reduces I/O latency when accessing index segments stored in H2 (SSD) such as Lucene's query cache. The graph captures the tradeoff in TeraHeap's static DRAM partitioning:

- Allocating more DRAM to H1 results in a larger heap, reducing GC time but yielding a smaller page cache, which increases I/O latency and non-GC execution time.
- Allocating more DRAM to the page cache reduces I/O latency and non-GC execution time but increases GC overhead due to the smaller H1.

The graph indicates that there is potential for optimization by introducing a dynamic heap resizer that adjusts the H1 size at runtime based on workload phases and memory pressure. Such an approach could balance the trade-offs between configurations by reducing GC time during indexing while also minimizing I/O latency during query processing.

3 Design

3.1 Overview

The goal of our dynamic heap resizer is to adjust the size of the primary managed heap (H1) at runtime, based on the total CPU cycles lost to the concurrent GC threads plus STW pauses versus I/O delays ,aiming to improve performance within a fixed DRAM budget. In our setup, the JVM uses TeraHeap, where H1 resides in DRAM and H2 is memory-mapped to an NVMe SSD, with the Linux page cache serving as the cache for H2. Unlike static DRAM division, this approach dynamically changes the size of H1 without requiring any modifications to the application itself.

Operations within H1 are mainly impacted by garbage collection (GC) overhead. Increasing H1 provides more space for new objects, reducing memory pressure and thus decreasing GC frequency and pause times, thereby preserving compute cycles for mutator threads. However, allocating more DRAM to H1 also reduces the memory available for the page cache of H2, potentially increasing I/O cost due to more frequent and expensive page faults when accessing SSD-resident data.

To balance these trade-offs, the dynamic resizer collects the totall lost compute cycles from concurrent GC thread CPU usage and STW pauses, and I/O metrics at regular sampling intervals. Using these insights, the system determines how to partition DRAM between H1 and the page cache for H2 by deciding whether to grow or shrink H1. When H1 is shrunk, the freed physical memory is reclaimed by the operating system and becomes available for the H2 page cache.

Because resizing decisions do not have immediate effects – for example, growing H1 reclaims pages from the H2 page cache only when the JVM accesses them, and shrinking H1 frees memory for H2 cache only after page faults trigger it – the Dynamic Heap Resizer incorporates a finite state machine (FSM) to manage adaptation safely. This FSM introduces wait states after each resizing action, allowing the system to observe the impact before making further changes, and includes direct state transitions to improve responsiveness and avoid delays. Figure 2 illustrates the architecture and workflow of the Dynamic Heap Resizer within our setup.

Overall, by dynamically adapting the size of H1 at runtime, our system enables TeraHeap to maintain low GC overheads during memory-intensive phases while ensuring sufficient DRAM page cache capacity for I/O-heavy phases, leading to improved performance without modifying application logic or the underlying hybrid heap architecture.

3.2 GC overheads

The Dynamic Heap Resizer operates based on the total compute cycles lost to garbage collection (GC) versus I/O delays to determine how to partition DRAM between the primary managed heap (H1) and the page cache used for the secondary

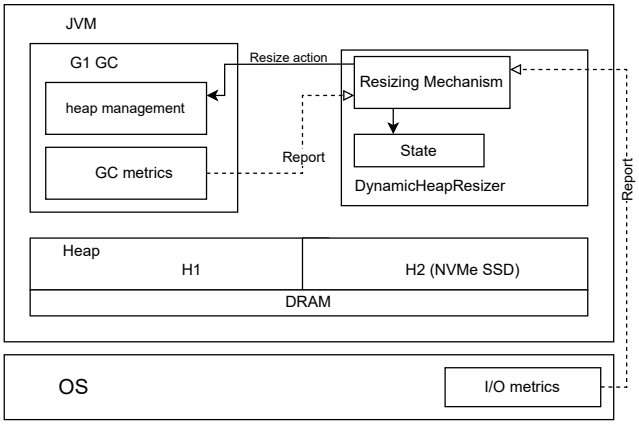

Figure 2: Overview of the Dynamic Heap Resizer architecture and workflow.

heap (H2). By dynamically resizing H1 at runtime, the system seeks to minimize overall application stall time. To understand this trade-off, we analyze the two key overhead components:

- **Stop-The-World (STW) Pauses:** STW pauses occur when the garbage collector halts all mutator threads to perform marking, evacuation, or compaction tasks. In G1GC, these pauses are triggered by allocation failures or periodic collection cycles. Longer STW pauses directly impact tail latency and user-facing performance, especially for latency-sensitive workloads like Lucene queries.

- **Concurrent GC Threads:** The performance impact of these threads depends on CPU allocation and workload conditions. When the total number of mutator and GC threads exceeds the available physical cores (*oversubscription*), GC threads compete with mutators for CPU time, increasing lost compute cycles due to context switching. Under full utilization (when mutator + GC threads equal the number of cores), GC threads do not directly preempt mutators, but still occupy cores that could have been available for additional mutator work. In partial competition scenarios (where the number of threads slightly exceeds the core count), a subset of GC threads preempt mutators, causing partial performance degradation.

- **Refinement Threads:** Refinement threads in G1 process write barriers and maintain the remembered sets required for incremental collection. These threads process card table updates asynchronously to reduce mutator overheads. However, high allocation or mutation rates (e.g. frequent index updates in Lucene) can overload refinement threads, leading to increased CPU utilization or buffer overflow stalls if write barrier queues become saturated.

Overall, GC overheads manifest as a combination of direct latency (pause times) and indirect CPU overhead (concurrent and refinement thread activity), both of which reduce application performance.

3.3 I/O Costs

The dynamic heap resizer evaluates I/O costs within sampling intervals between GC cycles. Minor GC cycles occur frequently in memory-intensive applications such as Lucene, making them natural boundaries for performance assessment.

To measure I/O cost accurately, we deploy an eBPF hook inside the Linux kernel that records the CPU iowait time by each mutator thread individually. This approach enables us to precisely measure each thread's non-GC/application work. At the end of each sampling interval, these exclusive per-thread iowait measurements are aggregated to calculate the total I/O cost for that interval.

While it is also possible to count major page faults directly, such counts fail to represent true I/O overhead because page fault durations vary significantly, depending on SSD latency, queue depth, and caching state. In contrast, time-based per-thread iowait captures the direct impact of storage delays on application progress.

This unified and precise I/O cost metric is used alongside GC cost estimates to inform resizing decisions, enabling the system to balance H1 heap size against page cache capacity effectively, thereby minimizing overall stall time experienced by mutator threads.

3.4 Interval variations to calculate GC time

Figure 3 presents three interval patterns observed in G1GC, presenting how its design combines concurrent phases with stop-the-world pauses to manage memory efficiently [5]. These patterns highlight the interplay between mutator threads, concurrent GC threads, and refinement threads during different phases of garbage collection, each exhibiting distinct GC cost characteristics relevant to the dynamic heap resizer.

(a) Simple GC Interval. In this pattern, the entire GC time consists solely of a stop-the-world (STW) pause. This occurs when the GC cycle completes within the STW phase without involving any concurrent GC threads. Thus, GC overhead here equals the STW pause duration.

$$GC_{\text{time}} = STW_{\text{pause}} \tag{1}$$

(b) Normal Concurrent Mark Cycle. This interval includes both an STW pause and a concurrent marking phase (CMC). During the concurrent mark, concurrent GC threads traverse the object graph while mutator threads continue executing, while refinement threads process remembered set updates. The interval ends with the STW Remark and Cleanup phases. Therefore, total GC time here is the sum of the concurrent mark duration and the final STW pause.

$$GC_{\text{time}} = CMC + STW_{\text{pause}} \tag{2}$$

(c) Interrupted Concurrent Mark Cycle. In this more complex pattern, GC time spans an initial concurrent marking phase (CMC1), followed by an STW pause that interrupts it, and then resumes with a second concurrent marking phase (CMC2). This scenario occurs when an additional GC trigger happens during an ongoing mark cycle, forcing a STW pause. As a result, GC time here is the cumulative duration of CMC1, the STW pause, and CMC2.

$$GC_{\text{time}} = CMC_1 + STW_{\text{pause}} + CMC_2 \tag{3}$$

These interval variations illustrate different GC scenarios that are compared against I/O costs, enabling the FSM to make informed decisions about resizing the heap accordingly.

3.5 Decision Making via FSM

The dynamic heap resizer implements a Finite State Machine (FSM) as shown in figure 4 with three core states to make runtime decisions about growing or shrinking the primary heap (H1). Each state evaluates current GC and I/O metrics to determine the next resizing action, ensuring responsiveness to application memory and I/O demands.

1. State: S WAIT GROW. In the S WAIT GROW state, the dynamic heap resizer evaluates the effectiveness of a previous grow action to determine whether further heap expansion is beneficial. Upon entering this state, the system compares the current total lost compute cycles and I/O costs to the delay recorded in the previous interval. If the lost compute cycles have not decreased, this indicates that increasing the heap size did not improve performance. Consequently, the finite state machine transitions to the S WAIT SHRINK state and issues a shrink action to reduce the size of H1, freeing DRAM for the H2 page cache. Conversely, if the lost compute cycles have decreased, the resizer checks the current heap occupancy. When the occupancy is high (exceeding 70%) and the heap has not reached its maximum allowed size, it remains in S WAIT GROW, issuing another grow action to further expand H1. If occupancy is low, it stays in S WAIT GROW but issues a wait action, pausing further resizing decisions until the next evaluation interval. Overall, this state ensures that heap growth is only reinforced when it yields a measurable reduction in lost compute cycles, maintaining a balance between garbage collection efficiency and available page cache capacity.

2. State: S WAIT SHRINK. In the S WAIT SHRINK state, the dynamic heap resizer evaluates the impact of a previous shrink action to determine whether a further heap reduction is required. Upon entering this state, it compares the current total lost compute cycles (GC and I/O costs) to the previously recorded delay. If the lost compute cycles have not decreased,

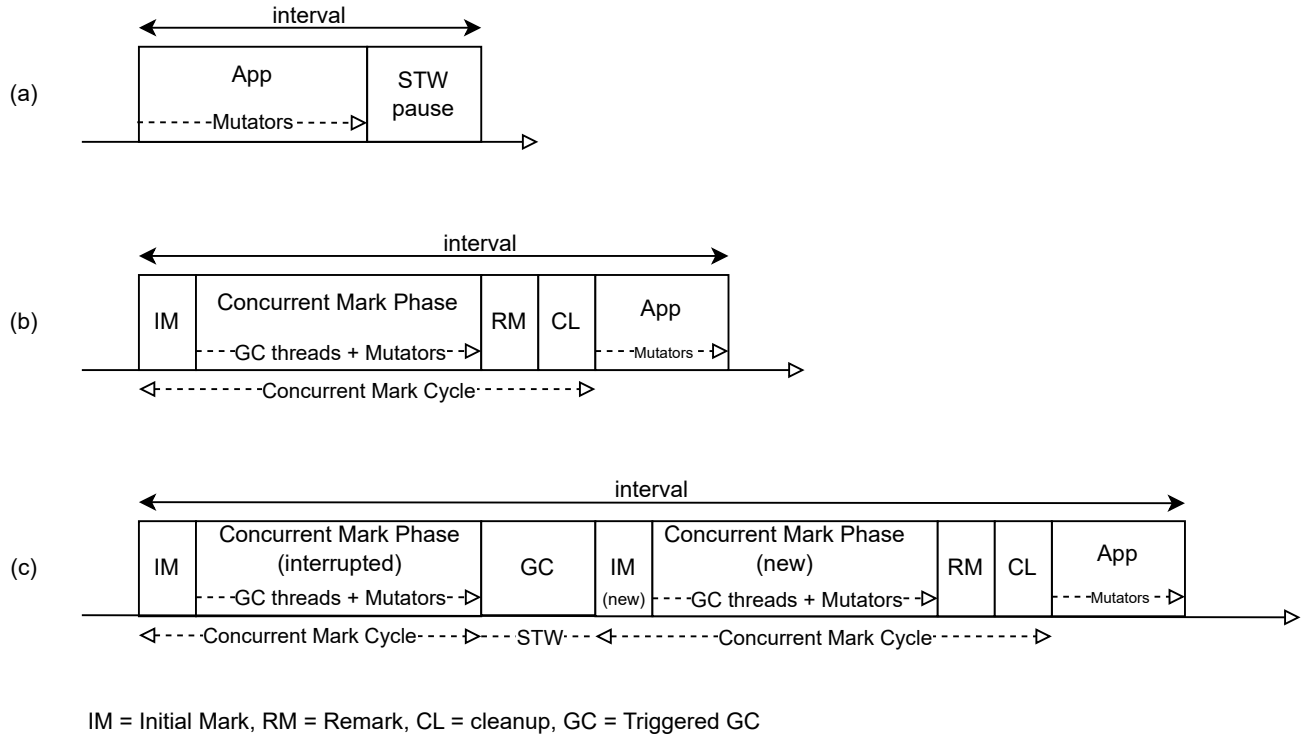

Figure 3: Interval variations for calculating GC cost.

indicating that shrinking H1 did not improve performance, the finite state machine transitions to the S WAIT GROW state and issues a grow action to increase the heap size, aiming to reduce GC overhead by partitioning more DRAM to H1. If the lost compute cycles have decreased, the resizer evaluates system memory slackness by checking whether the combined DRAM usage of the heap and page cache remains below 80% of the configured DRAM limit. If sufficient slack exists, it remains in S WAIT SHRINK, issuing an IOSLACK action to acknowledge available memory without invoking a resizing action. Otherwise, if slack is limited, the state remains in S WAIT SHRINK but issues an additional shrink action to continue freeing DRAM for the H2 page cache.

3. State: S NOACTION. In the S NO ACTION state, the dynamic heap resizer maintains its current heap configuration while continuously monitoring system performance. Upon entering this state, it compares the current total lost compute cycles, calculated as the combined GC and I/O costs, to the delay recorded before the last resizing action. If the lost compute cycles have decreased, indicating that the system performance has improved or remained stable without further resizing, the FSM remains in S NO ACTION, issuing no additional actions and preserving the current heap allocation. However, if the lost compute cycles have not decreased, suggesting that the current allocation is suboptimal, the resizer examines the last action executed. If the previous action was a heap shrink and the current heap size has not reached its configured maximum limit, the system transitions to S WAIT GROW and issues a grow action to increase H1, targeting reduced GC overhead. Conversely, if the last action was a grow, the FSM transitions to S WAIT SHRINK and issues a shrink action to reclaim DRAM for the page cache, aiming to reduce I/O costs. If neither condition is met, the FSM remains in S NO ACTION without issuing any resizing commands.

4 Evaluation

5 Related Work

Prior approaches fall into four categories: cached heaps, tiered heaps, hybrid heaps, and managed heap resizing techniques.

Cached heaps like Semeru [12], MemLiner [13], and Mako [8] allocate the managed heap entirely on remote memory while using local DRAM as a cache. These systems often require kernel modifications to implement remote paging and suffer from high I/O traffic and GC overheads due to frequent object scans and evacuations between memory tiers. For example, Semeru offloads GC scans to remote JVM instances to mitigate latency, while MemLiner reorganizes GC thread

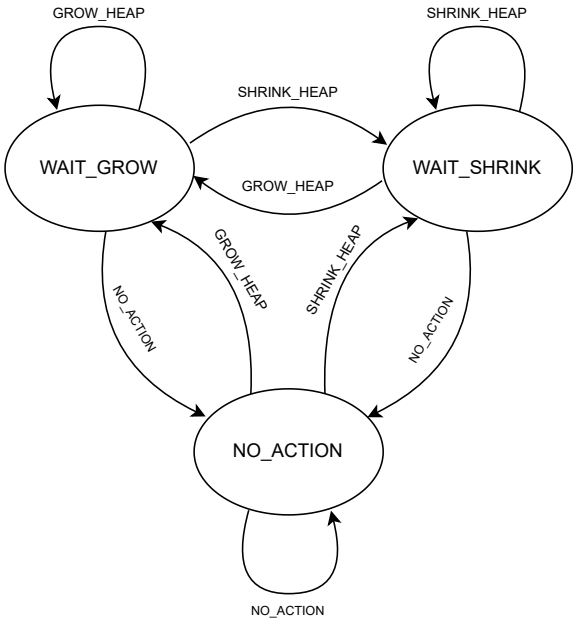

Figure 4: FSM for heap resizing decisions

memory access order to align with mutator access patterns. However, both still suffer from costly object transfers and rewriting between memory tiers. Mako further introduces a Heap Indirection Table to track object locations, which imposes load reference barriers on every access, adding CPU overhead. Friendly-NVM-GC [18] uses DRAM as a cache for a heap entirely placed on NVM but cannot avoid slow GC scans over NVM-resident objects.

Tiered heaps aim to reduce write amplification or NVM access latencies. Crystal Gazer [1, 2] and GCMove [7] frequently migrate objects between DRAM and NVM generations to limit writes but incur slow scans to update references. ThinGC [16] ensures mutators never directly access NVM-resident objects by moving them into DRAM upon access, avoiding immediate reference updates through lazy barriers, but this increases DRAM memory pressure and GC frequency. JPDHeap [19] requires programmers to annotate objects for placement across DRAM and NVM, again leading to expensive NVM scans for reference adjustments.

Hybrid heaps such as Melt [3] and LeakSurvivor [10] place hot objects in DRAM and cold objects on HDD or SSD, maintaining a user-space cache inside the JVM to track object offsets. While effective for HDD latency, they require a cache lookup for every access, adding CPU management overhead. TeraHeap [6], on the other hand, targets NVMe SSDs with memory-mapped I/O to reduce cache lookup costs, but its DRAM partition between the heap (H1) and page cache for the secondary heap (H2) is statically configured and does not adapt to workload changes.

Finally, managed heap resizing techniques focus on improving GC performance by dynamically adjusting heap sizes. Brecht et al. [4] propose aggressive heap expansion in Boehm GC to reduce execution time, while Yang et al. [17] develop an analytical model for heap resizing in multi-tenant environments. ElasticMem [14] reclaims heap memory for other processes in shared clusters to increase throughput. White et al. [15] use PID controllers to meet user-defined GC goals by tuning heap resize ratios. However, these approaches solely optimize GC behavior without considering I/O costs, limiting their applicability to hybrid heaps that require balanced DRAM allocation between managed memory and page cache for performance stability.

In contrast to these systems, our Dynamic Heap Resizer monitors runtime GC overhead and I/O cost, enabling dynamic partitioning of DRAM within hybrid managed heaps.

6 Conclusions

As shown in [**?**], modern systems require secure storage.

References

[1] Shoaib Akram, Jennifer Sartor, Kathryn McKinley, and Lieven Eeckhout. Crystal gazer: Profile-driven write-rationing garbage collection for hybrid memories. In *Proceedings of the ACM Measurement and Analysis of Computing Systems*, volume 3, pages 1–29. ACM, 2019.

[2] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*, pages 62–77. ACM, 2018.

[3] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA '08)*, pages 109–126. ACM, 2008.

[4] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. Controlling garbage collection and heap growth to reduce the execution time of java applications. *ACM Transactions on Programming Languages and Systems*, 28(5):908–941, 2006.

[5] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM)*, pages 37–48, Vancouver, British Columbia, Canada, 2004. ACM, ACM.

[6] Iacovos G. Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. TeraHeap: Reducing Memory Pressure in Managed Big Data Frameworks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 3 of *ASPLOS '23*, pages 694–709, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Zhe Li and Mingyu Wu. Transparent and lightweight object placement for managed workloads atop hybrid memories. In *18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '22)*, pages 72–80. ACM, 2022.

[8] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*, pages 92–107. ACM, 2022.

[9] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. DAOS: Data access-aware operating system. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22)*, pages 4–15, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Yan Tang, Qi Gao, and Feng Qin. Leaksurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *2008 USENIX Annual Technical Conference (USENIX ATC '08)*, pages 307–320. USENIX Association, 2008.

[11] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 261–280. USENIX Association, 2020.

[13] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, pages 35–53. USENIX Association, 2022.

[14] Jingjing Wang and Magdalena Balazinska. Elastic memory management for cloud data analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 745–758. USENIX Association, 2017.

[15] David R. White, Jeremy Singer, Jonathan M. Aitken, and Richard E. Jones. Control theory for principled heap sizing. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*, pages 27–38. ACM, 2013.

[16] Albert Mingkun Yang, Erik Österlund, Jesper Wilhelmsson, Hanna Nyblom, and Tobias Wrigstad. Thingc: Complete isolation with marginal overhead. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management (ISMM '20)*, pages 74–86. ACM, 2020.

[17] Ting Yang, Matthew Hertz, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*, pages 61–72. ACM, 2004.

[18] Yanfei Yang, Mingyu Wu, Haibo Chen, and Binyu Zang. Bridging the performance gap for copy-based garbage collectors atop non-volatile memory. In *Sixteenth European Conference on Computer Systems (EuroSys '21)*, pages 343–358. ACM, 2021.

[19] Litong You, Tianxiao Gu, Shengan Zheng, Jianmei Guo, Sanhong Li, Yuting Chen, and Linpeng Huang. Jpdheap: A jvm heap design for pm-dram memories. In *58th ACM/IEEE Design Automation Conference (DAC '21)*, pages 31–36. IEEE, 2021.