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 overall Java heap size at runtime based on GC behavior and I/O performance trends. Our resizer uses a lightweight policy engine that periodically monitors metrics such as GC pause times and page cache pressure 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 [3, 4]. 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 [2] 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 effec-

tively 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 without modifying TeraHeap's underlying architecture.

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 GC time compared to the I/O, 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, as increasing H1 provides more space for new objects, reducing memory pressure and thus the frequency and cost of GC cycles. 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 GC and I/O metrics at regular sampling intervals. GC costs are estimated using a model that distributes upcoming major GC pause times across future intervals, while I/O costs are directly measured based on page fault activity. 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 1 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.

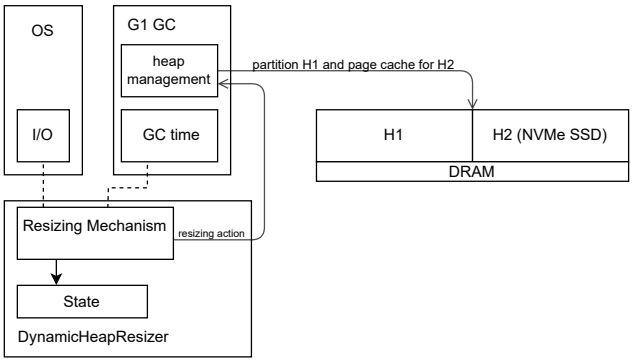

Figure 1: Overview of the Dynamic Heap Resizer architecture and workflow.

3.2 GC overheads

The Dynamic Heap Resizer operates by continuously comparing garbage collection (GC) costs and I/O costs to determine how to partition DRAM between the primary managed heap (H1) and the page cache used for the secondary 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:** G1GC employs multiple concurrent GC threads to perform background tasks such as concurrent marking, remembered set rebuilding, and cleanup phases. Although concurrent threads reduce the reliance on long STW pauses, they consume CPU resources that would otherwise be available for application threads. Excessive concurrent GC thread utilization can lead to CPU contention, reducing effective throughput.

- **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 2 presents three interval patterns observed in G1GC, presenting how its design combines concurrent phases with stop-the-world pauses to manage memory efficiently [1]. 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

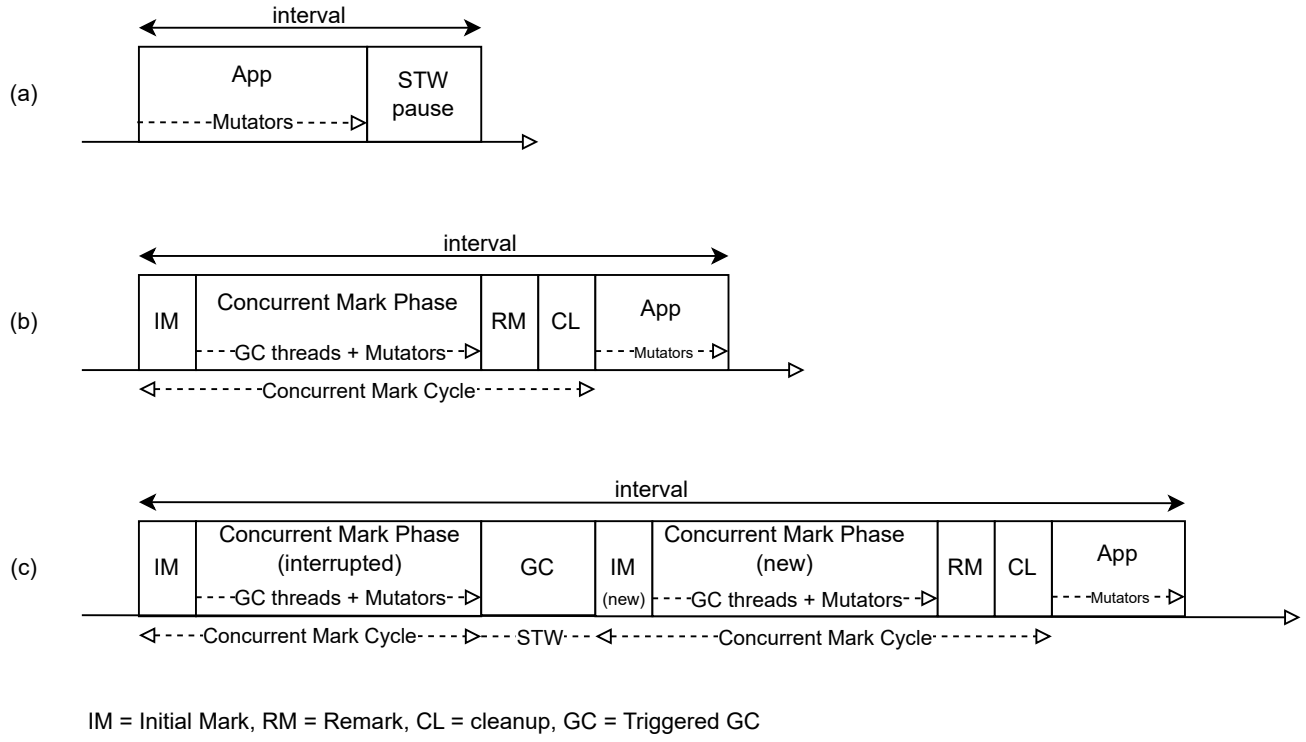

Figure 2: Interval variations for calculating GC cost.

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) 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.
2. State: S WAIT SHRINK.
3. State: S NOACTION.

Overall, this FSM design enables the dynamic heap resizer to:

- Adapt heap size based on observed performance effects.
- Prevent continuous grow or shrink oscillations through wait states.
- Respond efficiently to changing application memory pressure and I/O demands.

The clear separation of states ensures safe adaptation without introducing instability into the heap management policy, maintaining a balance between GC overhead reduction and sufficient DRAM availability for page caching.

4 Evaluation

5 Related Work

6 Conclusions

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

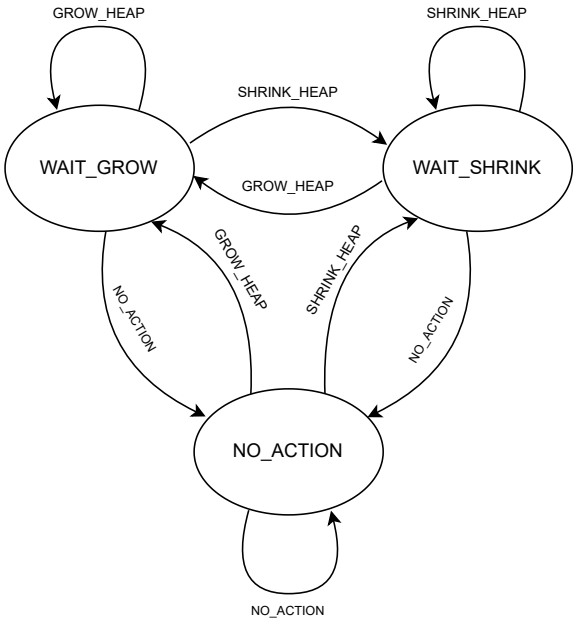

Figure 3: FSM for heap resizing decisions

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.