FlexHeap: Dynamic I/O-Aware Heap Resizing for Managed Applications

IACOVOS G. KOLOKASIS*, FORTH-ICS and University of Crete, Greece
SHOAIB AKRAM, Australian National University, Australia
FOIVOS S. ZAKKAK, Red Hat Ltd, Ireland
POLYVIOS PRATIKAKIS*, FORTH-ICS and University of Crete, Greece
ANGELOS BILAS*, FORTH-ICS and University of Crete, Greece

Popular JVM-based search and analytics systems, such as Elasticsearch and Spark, rely on the OS page cache (I/O cache) to accelerate storage access. However, dividing memory between the JVM heap and the I/O cache creates a trade-off: enlarging the heap reduces garbage collection (GC) overhead but starves the I/O cache, while shrinking it improves I/O performance but raises GC cost. Existing heap resizing mechanisms ignore I/O and thus fail to address this trade-off, resulting in inefficient memory utilization and degraded performance.

In this paper, we propose FlexHeap, a heap resizing mechanism for Garbage First (G1), the default OpenJDK garbage collector, that dynamically partitions a fixed DRAM budget between the JVM heap and the I/O cache. Between GC intervals, it estimates the CPU time lost to GC and to I/O stalls and repartitions DRAM to reduce their combined cost. FlexHeap relies on three concepts: (1) It makes resizing decisions using G1 collection boundaries. (2) It uses a history-based approach to estimate the cost of GC and I/O stalls for the future intervals. (3) It uses an adaptive resizing step that scales with changes in the combined cost.

We implement FlexHeap in OpenJDK 21's G1 garbage collector and evaluate it on two widely used systems: the Elasticsearch search engine and the Spark analytic framework. Compared to the G1 heap resizing mechanism, FlexHeap improves performance by an average of 30% in Elasticsearch and by an average of 33% in Spark. It outperforms Vertical G1, a state-of-the-art enhancement to the default G1 heap resizing mechanism, that returns unused memory to the OS eagerly, by 50% on average in throughput, demonstrating that JVM heap resizing needs to consider I/O overhead in search and analytics applications.

CCS Concepts: • **Software and its engineering** → **Memory management**; **Garbage collection**; • **Computer systems organization** → *Cloud computing*.

Additional Key Words and Phrases: Java Virtual Machine (JVM), Memory Management, Garbage Collection, G1 Garbage Collector, Application Heap and Kernel Page I/O Cache Resizing, eBPF

ACM Reference Format:
Iacovos G. Kolokasis, Shoaib Akram, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2026. FlexHeap: Dynamic I/O-Aware Heap Resizing for Managed Applications. *Proc. ACM Program. Lang.* 10, PLDI, Article 169 (June 2026), 24 pages. https://doi.org/10.1145/3808247

*Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS), Greece

Authors' Contact Information: Iacovos G. Kolokasis, FORTH-ICS and University of Crete, Greece, kolokasis@ics.forth.gr; Shoaib Akram, Australian National University, Australia, shoaib.akram@anu.edu.au; Foivos S. Zakkak, Red Hat Ltd, Ireland, fzakkak@redhat.com; Polyvios Pratikakis, FORTH-ICS and University of Crete, Greece, polyvios@ics.forth.gr; Angelos Bilas, FORTH-ICS and University of Crete, Greece, bilas@ics.forth.gr.

ACM 2475-1421/2026/6-ART169
https://doi.org/10.1145/3808247

1 Introduction

Widely used search and analytics systems [1, 26, 30, 53] require substantial data movement to and from storage devices. Such applications typically run on managed runtimes, particularly the Java Virtual Machine (JVM). To mitigate the performance impact of I/O, these systems rely heavily on the OS page cache (I/O cache) in DRAM [16, 19, 29, 36, 37], which buffers frequently accessed data to reduce access latency and I/O traffic. However, dividing DRAM between the JVM heap and the I/O cache has become increasingly challenging as applications process larger datasets while DRAM capacity scaling slows down [32, 38, 47, 50]. As a result, runtime adjustments to DRAM partitioning between the heap and the I/O cache can have a substantial impact on performance: allocating more memory to the heap can starve the I/O cache and increase I/O stalls, while allocating less increases garbage collection (GC) overhead [20].

Existing heap resizing mechanisms [5, 44, 45, 49] improve GC efficiency but remain agnostic to the interaction between GC and I/O performance. They aim to reduce GC-related costs, such as pause time or GC CPU overhead, by adjusting heap size rather than managing the trade-off between heap and I/O cache usage. For example, Garbage First (G1) [12], the default OpenJDK collector, adjusts heap size based on GC behavior: it grows the heap to reduce GC frequency and releases memory when observed pauses stay below user thresholds. To overcome the lack of I/O awareness, users often cap the maximum heap at a fixed fraction of DRAM, leaving the remainder to the I/O cache. For instance, Cassandra [26] and Elasticsearch [1] recommend assigning about half of the memory to the heap and half to the I/O cache [3, 14]. The collector's resizing mechanism still operates within these bounds. However, the heap cannot grow beyond the cap even when application demand rises, increasing GC overhead during phases with high allocation rates that require more heap space (Section §3.1).

Previous work [7, 33, 34, 49] has also examined how to reduce memory usage in containerized deployments by returning unused committed heap memory to the OS eagerly. Vertical G1 (VG1) [7, 34] preserves G1's heuristics and adds periodic triggers to uncommit memory to meet memory usage targets, but it does not incorporate I/O cost in its decisions. Consequently, such occupancy-based resizing mechanisms fail to adapt during application phases where heap and I/O demands evolve together (Section §3.2).

In this work, we propose FlexHeap, the first heap resizing mechanism, to the best of our knowledge, that accounts for both GC and I/O overheads in dynamic memory partitioning. We implement FlexHeap in G1, the default OpenJDK garbage collector, by replacing its heap resizing mechanism. FlexHeap continuously monitors how GC and I/O stalls affect application performance and mitigates their impact by either growing the heap to reclaim memory from the I/O cache or shrinking it to release memory back to the I/O cache. Our design tackles three key challenges:

Estimating GC cost and I/O stalls. GC cost and I/O stalls manifest differently: GC includes stop-the-world (STW) pauses, where all application (mutator) threads are suspended, as well as concurrent phases, where GC threads run alongside mutators; I/O stalls occur when mutator threads block waiting for data from storage [42]. To estimate their impact, FlexHeap quantifies both costs in terms of lost compute cycles. FlexHeap estimates CPU time differently for STW and concurrent phases of GC to reflect interference with mutator execution better. To estimate the I/O stall of mutator threads, FlexHeap uses an Extended Berkeley Packet Filter (eBPF) [39] program to track the time mutator threads spend in uninterruptible sleep [42] state in the Linux kernel.

Estimating the impact of heap resizing on GC cost and I/O stalls. GC cost and I/O stalls both depend on the amount of memory used by the respective operations. However, an amount of memory may not necessarily have the same benefit for GC and I/O. The relative effects of reassigning

memory between the heap and the I/O cache depend on workload and system characteristics, such as access patterns, working-set size, and storage devices. As a result, fixed weights for GC and I/O costs are difficult to choose, and a fixed analytical model of the GC–I/O relationship may not generalize across workloads and environments. FlexHeap therefore adopts a model-free design and bases its resizing decisions by monitoring the cycles lost for performing GC and for I/O stalls. When the combined cost increases, FlexHeap identifies which cost is responsible for this increase and guides subsequent resizing decisions. In particular, FlexHeap considers as the dominant overhead not simply the largest component in absolute terms, but the one most sensitive to recent resizing decisions.

Deciding the amount of the resizing step. Resizing steps should be aggressive when the combined cost of GC and I/O stalls increases, and conservative near steady state. In addition, shrink steps must respect G1's pause time goal, which depends on maintaining enough eden space for the allocation of new objects. FlexHeap sizes each step in two stages. First, it computes a safe budget under G1 constraints: for growth, it limits the step to the available headroom below the maximum heap size, and for shrinking, it uses only the free space beyond the eden space required for the pause time goal. Then, FlexHeap sizes the step according to how the total cost has changed in the last interval: larger increases trigger larger adjustments, small changes lead to small steps, and bounds prevent overshoot.

We implement FlexHeap in the G1 garbage collector of OpenJDK 21, a widely adopted long-term support release of the OpenJDK. We evaluate FlexHeap using two widely used systems: Elasticsearch and the analytic framework Spark. FlexHeap improves performance on average by 30% for Elasticsearch and by 33% for Spark. Compared to VG1, FlexHeap achieves on average 50% higher performance, showing that memory repartitioning must also account for I/O-induced stalls in such workloads.

Overall, our paper makes the following contributions:

- We examine the trade-off between application DRAM demands for heap and I/O cache and show the potential for improving overall performance by dynamically dividing DRAM between the two uses.
- We propose an *application-agnostic and adaptive* mechanism which reassigns memory between heap and I/O cache by considering the combined overhead of GC and I/O stalls during execution.
- We evaluate FlexHeap on widely used search and analytics systems and show that it outperforms prior GC-based and occupancy-based heap resizing mechanisms.

2 Background

This section provides background on the Garbage-First (G1) garbage collector, Vertical G1 (VG1) resizing mechanism, and the buffered I/O in Linux kernel.

G1. It is the default garbage collector of OpenJDK designed to balance high application throughput with predictable pause times. G1 divides the heap into equally sized regions and follows a generational model with two generations: young and old. The young generation consists of eden and survivor regions. Mutator threads allocate new objects in eden regions, and each young GC cycle reclaims eden space by evacuating live objects to survivor regions. Objects that survive multiple young GCs are promoted to the old generation.

G1 performs both STW and concurrent GCs. It begins with repeated young GCs (STW) that target only the young generation and pause all mutator threads. As the old generation fills and its occupancy crosses a configurable threshold, G1 triggers a concurrent marking GC to identify

live objects across the heap without stopping mutators. After concurrent marking completes, G1 performs one or more mixed GCs (STW) that reclaim memory from both young and selected old regions with a high fraction of garbage. Finally, full GCs (STW) are rare and serve as a fallback mechanism, pausing all mutators to reclaim the entire heap when incremental GCs cannot keep up with allocation demand.

G1 also relies on mutator write barriers to track cross-region references. When a mutator updates an object reference, G1 marks the corresponding entry in a card table, a byte array in which each byte represents a 512 B segment of the heap. A marked card indicates that the corresponding segment may contain updated references that must later be examined by the garbage collector. Concurrent refinement threads run alongside mutator threads and process these dirty cards while the application executes, identifying cross-region references and updating the remembered sets of the corresponding target regions. By doing so, they move much of remembered-set maintenance out of the STW pause, thereby reducing GC pause time. During young or mixed GC, when G1 collects a region, it uses that region's remembered set to locate incoming cross-region references and update them to point to the objects' new locations, without scanning the entire heap.

VG1. It is a time-driven, occupancy-based mechanism that preserves G1's default heap resizing heuristics while returning memory to the OS more aggressively. It triggers a concurrent mark cycle if no collection occurs within a user-configured interval (`-XX:G1PeriodicGCInterval`). In G1, each concurrent marking cycle concludes with a remark phase, a short STW pause in which G1 finalizes the marking process to ensure that all live objects are correctly identified before mixed GCs. At the end of the remark phase, VG1 makes heap resizing decisions based on heap occupancy. If the gap between committed memory and the memory occupied by live objects exceeds a threshold, VG1 returns memory to the OS.

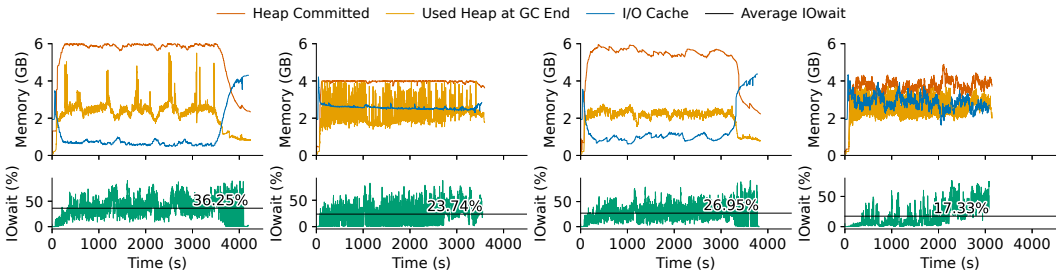
Buffered I/O. Buffered I/O is the default file-I/O mode in most operating systems. In Linux, it uses the kernel's I/O cache, which stores file data in memory in fixed-size pages (typically 4 KB), to serve reads and temporarily buffer writes before they are flushed to storage. This allows recently accessed file data to be served from DRAM instead of the storage device. Buffered I/O also allows the OS to prefetch file pages into the I/O cache in anticipation of future accesses. As a result, applications can issue file accesses at arbitrary offsets and sizes without managing low-level device requirements directly. In contrast, direct I/O bypasses the kernel's I/O cache and transfers data directly between the application and the storage device. Direct I/O typically requires applications to satisfy device-specific alignment and I/O-size constraints. Consequently, applications using direct I/O do not benefit from additional I/O cache capacity.

3 Motivation

We motivate the need for a dynamic heap-resizing mechanism that accounts for both GC and I/O stalls. Specifically, we show that existing heap-resizing mechanisms fail to adapt to the phase behavior of applications, whose heap and I/O cache demands vary over time. To illustrate this limitation, we use Elasticsearch, a widely used production-grade search engine deployed at scale for log analytics, full-text search, and observability workloads. We run the Wikipedia workload that performs concurrently search and document ingestion. During ingestion, incoming documents are indexed and made searchable. Section § 5 describes the full experimental setup.

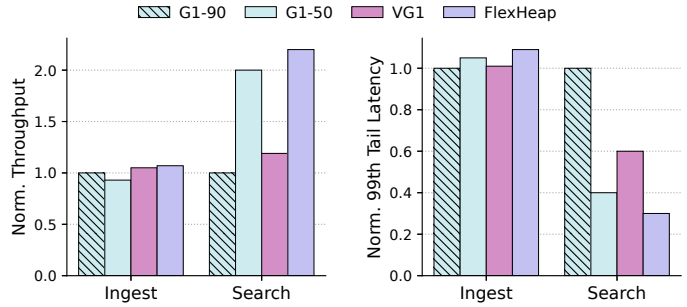
3.1 GC-Based Heap Resizing Mechanisms Prioritize the Heap Over the I/O Cache

G1's heap resizing mechanism cannot adapt to workloads that simultaneously stress the heap and the I/O cache because it relies solely on GC metrics. In Figure 1(a), we use G1 with the heap capped at 90% of the available system DRAM (G1-90) on Elasticsearch running the Wikipedia workload.



(a) G1-90 (b) G1-50 (c) VG1 (d) FlexHeap

Fig. 1. Heap usage, I/O cache occupancy, and I/O wait over time for the Wikipedia workload in Elasticsearch: (a) G1-90 (with a maximum heap cap of 90% of DRAM), (b) G1-50 (with a maximum heap cap of 50% of DRAM), (c) VG1, and (d) FlexHeap.



(a) Throughput (b) Latency

Fig. 2. Normalized document ingestion and query search (a) throughput, and (b) p99 latency for the Wikipedia workload in Elasticsearch under G1-90 (with a maximum heap cap of 90% of DRAM), G1-50 (with a maximum heap cap of 50% of DRAM), VG1, and FlexHeap. Results are normalized to G1-90.

Document ingestion and search run in parallel from the start until about 3500 s, after which only search continues. During this mixed phase, G1 continues to expand the committed heap until it reaches the configured cap. Because its decisions consider only GC cost, G1 assigns nearly all available DRAM to the heap, leaving little space for the I/O cache; as a result, I/O wait remains high (about 36% on average) for most of the execution. After ingestion ends, heap usage drops to approximately 1 GB, and the collector gradually returns memory to the OS; therefore, shrinking occurs only once heap pressure subsides.

Following Elasticsearch guidelines [14], we cap the heap at 50% of DRAM (G1-50) and run the same workload. G1's dynamic resizing remains active but is constrained by this cap. Using G1-50, the execution time decreases by 15% compared to G1-90. This improvement is mainly due to the reduction of the average I/O wait by 24% (a 33% improvement over G1-90) because the I/O cache remains larger (approximately 3 GB). However, the cap restricts heap growth during document ingestion and consequently drives up GC activity: 54 full GCs instead of 1 in G1-90, plus 2,530 additional young and mixed GCs overall. Under G1-50, document ingestion tasks run until the end of execution, resulting in 8% lower document ingestion throughput compared to G1-90 (Figure 2(a)). Conversely, the consistently larger I/O cache under G1-50 improves query performance: Figure 2(a) shows that search throughput is 2× higher than under G1-90, and Figure 2(b) shows that p99

latency is 62% lower. This improvement primarily stems from a 2.4× reduction in device read traffic compared to G1-90.

3.2 Occupancy-Based Heap Resizing Mechanisms Ignore I/O Cost

Next, we examine whether time-driven, occupancy-based mechanisms, such as VG1, that return memory to the OS more aggressively can reduce I/O overhead. As shown in Figure 1(c), VG1 improves execution time by 5% upon G1-90 but is insufficient without I/O awareness: during mixed phases, the heap remains near the cap and the I/O cache is constrained. VG1 improves performance relative to G1-90 by reducing average I/O wait from 36% to 27% and increasing document ingestion throughput by 5%. However, it remains inferior compared to G1-50. As VG1 does not adapt within the mixed ingestion–search phase, it continues to allocate most DRAM to the heap rather than the I/O cache when needed. Consequently, Figure 2(a) shows that search throughput is approximately 40% lower than G1-50, which maintains a larger I/O cache by capping the heap at 50% of DRAM (at the cost of increased GC activity and lower document ingestion throughput).

3.3 Considering Both GC and I/O Costs for Heap Resizing

Figure 1(d) shows FlexHeap, which resizes the heap using both GC and I/O feedback. During the mixed phase, it prevents the heap from monopolizing DRAM, leaving sufficient room for the I/O cache. These results indicate that considering both GC and I/O stalls enables more efficient DRAM use and higher end-to-end performance. We present detailed results in the Section §6.

4 FlexHeap Design

4.1 Overview

The goal of FlexHeap is to improve application performance by effectively repartitioning memory between the heap and the I/O cache for applications that demand both. We first outline how G1's resizing mechanism currently works, then show how FlexHeap replaces it with a mechanism that considers both GC and I/O stalls.

Figure 3(a) depicts G1's heap resizing mechanism in OpenJDK 26. Resizing criteria depend on the type of collection. Young and mixed GCs occur more frequently than full GC, so they drive most resizing decisions. In this path, heap resizing is based on the ratio of time spent in the GC versus the execution of mutator threads (GC time ratio). If the GC time ratio consistently exceeds a threshold (`-XX:GCTimeRatio`), G1 decides to grow the heap. If the ratio is consistently low, it decides to shrink the heap. G1 uses a fraction of the uncommitted memory as the grow step, scaled based on the recent change in the GC time ratio. The shrink step starts from the free committed bytes left after reserving the eden space target size, then scales that budget based on the recent change in the GC time ratio. Then, based on the decision, G1 either commits additional memory or uncommits memory back to the OS. On the other hand, after a full GC, which is rare compared to young and mixed GCs, G1 compares the post GC free fraction to `-XX:MinHeapFreeRatio` and `-XX:MaxHeapFreeRatio` and grows or shrinks the heap to keep free space within these bounds. After heap resizing, G1 also sizes the eden space using a pause time model so that GC pauses meet the user-defined goal (`-XX:MaxGCPauseMillis`).

Figure 3(b) illustrates the heap resizing mechanism of FlexHeap in G1. We use a unified resizing mechanism regardless of the GC type. At each decision point, FlexHeap (i) estimates the cost of GC (Section § 4.2) and I/O stalls (Section § 4.3), (ii) it uses the combined GC and I/O cost to grow or shrink the heap (Section § 4.4), and (iii) estimates the amount of memory to reassign (Section § 4.5). Next, we discuss each mechanism of FlexHeap in more detail.

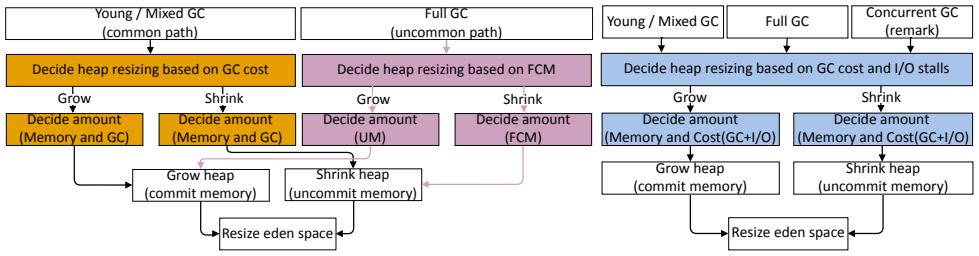


Fig. 3. Overview of (a) G1 and (b) FlexHeap heap resizing mechanisms (acronyms: **UM**=**U**ncommitted **M**emory, **FCM**=**F**ree **C**ommitted **M**emory

4.2 Estimating Impact of GC

GC activity is the dominant overhead for heap operations. We estimate the GC CPU cycles spent in the current interval, defined as the time from the end of STW_{i-1} to the end of STW_i. Within this interval, we account for all GC work: the concurrent GC executed between pauses and the subsequent STW collection (young, mixed, or full GCs). This concurrent activity includes the concurrent marking GC cycle, and the work of concurrent refinement threads that update per-region remembered sets. As the concurrent GC overlaps with mutators, wall-clock time overestimates actual CPU usage. Thus, we measure GC cost in CPU time, following prior work [45]. The total GC CPU time of the current interval is the sum of CPU time spent in the subsequent STW GCs and in concurrent GCs:

$$C_{gc} = C_{stw} + C_{conc} \tag{1}$$

STW GC CPU time. During an STW GC, all mutator threads are suspended, so the system loses the computation they could have performed during the pause time (T_{stw}). The lost computation depends on the minimum of the available CPU cores (N) and the number of mutator threads (M). The challenge is knowing M at the pause boundary, as the mutator threads start and terminate over time, and some may already be waiting on locks. FlexHeap therefore uses a practical upper bound: it maintains an atomic count of active Java mutator threads (excluding VM/GC/internal threads), incremented on thread start and decremented on termination, and reads this count at each STW boundary. This may overestimate lost work because some counted threads could be non-runnable, but filtering them out would require a pre-safepoint runnability snapshot and per-transition instrumentation, adding prohibitive overhead. Thus, we compute the STW cost as:

$$C_{stw} = T_{stw} \cdot \min(N, M) \tag{2}$$

Concurrent GC CPU time. During concurrent GC, both mutator and GC threads execute in parallel and contend for CPU resources. We estimate the portion of G concurrent GC threads that displace work from M mutator threads, resulting in lost mutator progress. G includes both concurrent marking threads and refinement threads. For this purpose, FlexHeap compares the total number of active threads ($M+G$) to the number of available cores N for the specific interval. When $M+G \leq N$, there are idle cores, and concurrent GC can proceed without cost. When $M+G > N$, some GC threads preempt mutators, and only the excess CPU time beyond idle capacity contribute to overhead. The concurrent cost is therefore:

$$C_{conc} = \begin{cases} \sum_{g \in G} T_{\text{CPU}}(g) - \max\big(0,\ (N - M) \cdot T_{mut}\big), & M + G > N \\ 0, & M + G \leq N \end{cases} \tag{3}$$

Here, T_{mut} is the time between the end of one STW phase and the start of the next, representing the interval during which mutator and concurrent GC threads overlap. For example, on an 8-core system, suppose that during a given interval there are $M = 6$ mutators and $G = 4$ concurrent GC threads. With $N = 8$, only $N - M = 2$ cores are available for concurrent GC, so $G - (N - M) = 2$ concurrent GC threads have no idle cores to run on; their CPU cycles in that interval are counted as overhead. In addition to competing for cores, concurrent GC threads can also interfere with mutators through contention for other resources, such as cache and memory. Precisely attributing that interference would require fine-grained hardware profiling which adds non-negligible overhead and complexity.

4.3 Estimating Impact of I/O Stalls

The memory available to the I/O cache directly determines how much time mutator threads spend stalled on I/O. A larger I/O cache allows more frequently accessed data to be served from DRAM, reducing page faults that cause storage access and keeping mutators from getting context switched. However, this benefit depends on the access locality: workloads with little or no reuse derive limited advantage from additional cache space. Conversely, when the I/O cache is small or under pressure, a larger fraction of reads and writes must be served from the storage device, causing mutator threads to block in the kernel I/O path until the data are fetched into memory. These stalls translate directly into CPU time that could otherwise be used for application processing, reducing performance. To capture this effect, FlexHeap estimates the CPU time lost due to I/O stalls, using it as a measure of the performance cost of limited I/O cache space to guide resizing decisions.

Algorithm 1 eBPF Handler for `sched_switch`.

```
1:  procedure ON_SCHED_SWITCH(prev_tid, next_tid, prev_state)
2:      ts ← current_time()
3:      if prev_tid ∈ mutators and prev_state = D then
4:          iostart[prev_tid] ← ts
5:      end if
6:      if next_tid ∈ mutators and next_tid ∈ iostart then
7:          dt ← ts − iostart[next_tid]
8:          if next_tid ∈ iowait then
9:              iowait[next_tid] ← iowait[next_tid] + dt
10:         else
11:             iowait[next_tid] ← dt
12:         end if
13:         delete iostart[next_tid]
14:     end if
15: end procedure
```

To quantify this loss, we measure the time mutator threads spend in the uninterruptible (D) state [42], during which they are blocked on I/O and cannot make progress. Even when other threads continue to use the CPUs, these stalled mutators still represent lost opportunities for

mutator progress. C_{io} therefore captures mutator progress lost while threads are blocked on I/O. This differs from C_{gc}, which estimates the loss caused by GC displacing runnable mutators from the CPU. Measuring C_{io} only as idle-core-equivalent stall time would therefore underestimate the impact of I/O-bound phases, particularly in services where blocked mutators contribute to queue buildup and tail latency.

Existing standard kernel interfaces do not directly expose the time per mutator thread lost to I/O stalls. The `iowait` metric in `/proc/stat` is a CPU-level aggregate metric. It reports that a CPU is idle while at least one task is waiting for I/O [28], rather than how much time a particular thread is stalled on I/O. For example, a mutator thread may be blocked on a lock while an unrelated thread on the same CPU waits for I/O, so treating the recorded `iowait` as the mutator's I/O stall time would be incorrect. More fundamentally, kernel interfaces do not directly expose per-thread I/O stall measurements, because scheduler semantics decouple I/O completion from rescheduling. For example, a thread's I/O may complete before the scheduler places the thread back on a CPU, making it unclear whether the intervening delay should be counted as I/O stall time.

FlexHeap implements an eBPF [39] program attached to the `sched_switch` tracepoint to obtain an approximation of the per mutator thread I/O stall time, as shown in Algorithm 1. The eBPF program maintains two maps: `iostart`, which stores the timestamp at which a mutator thread enters the D state, and `iowait`, which accumulates the total time that the thread has spent blocked on I/O. On each context switch, if a tracked mutator thread is scheduled out in the D state, we record the current timestamp in `iostart`; when it is scheduled back in, we compute the elapsed time since that timestamp and add it to `iowait`. We use the accumulated value in `iowait` as the mutator CPU time lost to I/O stalls.

To ensure the estimate respects hardware limits, the computed I/O-induced CPU time loss must not exceed the machine's compute capacity over the measurement interval. When the number of mutator threads exceeds the available CPU cores and all are simultaneously stalled on I/O, the total lost CPU time cannot be greater than the interval duration multiplied by the number of cores. We therefore cap the measured stall time to the product of the interval duration and the number of CPU cores, as shown in Eq. 4:

$$C_{io} = \min\Big(\sum_{m \in M} T_{\text{io}}(m), T_{mut} \cdot N\Big) \tag{4}$$

4.4 Deciding to Grow or Shrink the Heap

After each GC cycle, FlexHeap determines whether to grow the heap (`growHeap`) or shrink the heap (`shrinkHeap`) to release memory for the I/O cache. For the decision-making process, FlexHeap compares, across intervals, the sum of CPU time consumed by GC and I/O stalls, which together capture lost mutator progress. To reduce short-term noise, FlexHeap averages the sum of C_{gc} and C_{io} over the last ten intervals, following G1's use of a ten-interval moving average. An alternative approach would be to balance these two costs, aiming for equal contribution from both. However, this does not necessarily minimize total overhead. Equalizing the two costs may still yield suboptimal performance. For example, in scan-heavy workloads with low I/O cache hit rates, prioritizing I/O cost could enlarge the I/O cache without improving throughput, while simultaneously inflating GC overhead. This counterproductive behavior shows that balancing costs is insufficient—FlexHeap instead seeks to reduce their combined sum.

If a `shrinkHeap` (or `growHeap`) action reduces the total cost ($C_{tot} = C_{gc} + C_{io}$), FlexHeap continues applying the same action. However, whenever FlexHeap applies a resize action, it must account for the delay before the effects of the previous resize are reflected in heap usage or I/O cache growth. Without waiting for the repurposed memory to be absorbed by the heap or the I/O cache,

Predicate	Predicate Definition	Complementary Predicate
costDown	$C_{\text{tot}}[t] \leq C_{\text{tot}}[t-1]$	costUp
growSlack	$S_{\text{used}} < S_{\text{used\_before\_grow}} + S_{\text{grow}}$	growUsed
ioSlack	$S_{\text{io}} < S_{\text{io\_before\_shrink}} + S_{\text{shrink}}$	ioFull
gcBias	$\Delta^{\text{rel}} C_{\text{gc}}[t] \geq \Delta^{\text{rel}} C_{\text{io}}[t]$	ioBias

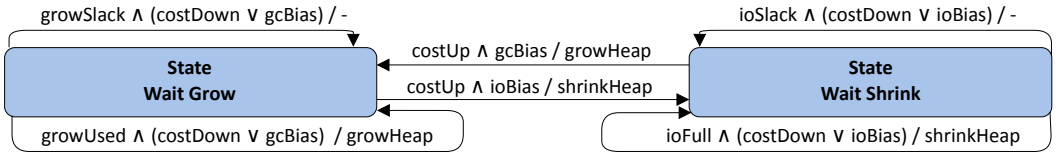


Fig. 4. Finite-state machine of the FlexHeap resizing controller. Each transition label has the form *condition* / *action*. The table defines the predicates used in transition guards, their formal definitions, and the complementary predicate names. For example, growUsed is the complement of growSlack and denotes the case where the heap has exhausted the space added by the previous grow step. In the table, for the definition of predicates, S denotes a size-related quantity.

FlexHeap could issue premature actions, resulting in unnecessary or even harmful resizing. This is particularly important when memory released from the heap does not improve the I/O cache, for example because the application uses direct I/O, or because, in a multitenant container, the freed memory is consumed by sibling processes instead.

To handle delayed effects, FlexHeap employs *wait states*, which defer further resizing until the impact of the previous action becomes observable, as shown in Figure 4. After a growHeap action, FlexHeap enters in the state *Wait Grow*. If C_{tot} decreases and the space added by the previous grow step has not yet been exhausted (growSlack), the controller waits; if that space has been exhausted (growUsed), it issues another growHeap action. Symmetrically, after a shrinkHeap action, FlexHeap enters in the state *Wait Shrink*. If C_{tot} decreases and the memory released by the previous shrink step has not yet been fully consumed by the I/O cache (ioSlack), the controller waits; otherwise, it issues another shrinkHeap action.

Conversely, if C_{tot} increases, FlexHeap determines the next direction by comparing the relative change of the two cost components, denoted by $\Delta^{\text{rel}} C_x[t] = (C_x[t] - C_x[t-1])/C_x[t-1]$. If the relative change in C_{gc} is greater than or equal to that in C_{io} (gcBias), the controller infers that GC was affected more adversely by the last resizing step and issues growHeap; otherwise, it issues shrinkHeap. We use relative rather than absolute changes because C_{gc} and C_{io} may differ by orders of magnitude and their dominance may shift across workloads and phases. Comparing absolute values would bias the controller toward whichever component is currently larger, even when the other is more sensitive to heap resizing. By normalizing each cost to its previous value, FlexHeap makes scale-invariant decisions based on which component changed more sharply in response to the last action.

A simpler alternative would be to immediately undo the previous resizing action whenever C_{tot} increases. We do not adopt this policy because such an immediate reversal may be premature and can induce oscillation. C_{gc} and C_{io} respond differently to memory reassignment: additional heap memory generally reduces GC pressure, whereas releasing memory to the I/O cache does not always reduce I/O cost, depending on the workload's working-set size and access pattern.

Proactive GC triggering during mutator I/O stalls. Mutator threads can become blocked on I/O for an extended period, significantly delaying the next GC cycle. When this happens, heap

resizing decisions are also delayed, leading to feedback lag. To maintain responsiveness in these cases, FlexHeap triggers a GC proactively when mutator threads stall as follows.

A lightweight VM thread runs periodically every second (similar to VG1) and monitors two runtime metrics: allocation rate and mutator I/O stall time. FlexHeap samples the total bytes allocated by mutator threads every second. It maintains a sliding window of recent allocation measurements and computes a moving average over the last five intervals. If the current allocation rate drops by half, FlexHeap marks the interval as a low-activity period. To avoid reacting to transient changes, it tracks how many of the last few intervals have shown low activity. Only if multiple consecutive samples indicate reduced allocation rate, does it consider the system to be stalled. For this period, FlexHeap also monitors I/O stall cycles. If high stall time coincides with low allocation activity, FlexHeap concludes that threads are waiting on I/O rather than idle, and triggers a GC. To avoid disrupting normal GC scheduling, FlexHeap inserts the GC request through standard internal JVM hooks, and ensures that no GC is already in progress.

4.5 Estimating the Resizing Step

The goal of the growing and shrinking steps is to respond quickly to changes in application behavior and to respect G1's pause time goals. For responsiveness, FlexHeap scales each resize step by the recent change in the combined cost. For pause protection, FlexHeap relies on G1's eden sizing model and leaves it unchanged. This model is not part of our resizing mechanism. After each young or mixed GCs, G1 measures the pause overhead. It uses this metric to predict the next young-GC pause and then chooses the largest size for eden space that meets the user pause target within available free memory. We use this model only to bound shrinking: the shrink step returns memory only from free regions beyond the computed target size of eden space. Similar to G1's resizing mechanism, we do not use the pause-time model for the growth step. Growing only adds free regions, and G1's eden sizing model still selects exactly as many eden regions as needed to meet the pause target. Growth therefore follows G1's bounded expansion mechanism, scaled by our total cost metric, while remaining aligned to region size and limited by uncommitted headroom.

Fig. 5. Committed and uncommitted heap memory.

Growing step. As shown in Figure 5, the heap has committed memory (backed) and uncommitted memory (reserved up to maximum heap size). A grow step, calculated by Algorithm 2, cannot exceed the uncommitted headroom. FlexHeap forms a base budget as the smaller of (i) a fixed percentage of the uncommitted headroom (the same percentage used by native G1) and (ii) the current committed size (Line 7). This avoids oversized jumps that would reduce GC frequency and risk longer pauses. FlexHeap then scales the normalized cost change ΔC_{tot} using a step scale with a logistic (S-shaped) function (Line 8). ΔC_{tot} is the change between intervals: larger increases yield larger steps, small changes yield conservative steps. We adopt the same logistic form used by G1's resizing mechanism, keeping tuning behavior consistent. Finally, FlexHeap returns the grow bytes constrained to the admissible range. The resize amount does not exceed the available uncommitted headroom and is not less than the minimum expansion.

Algorithm 2 Growing step

```
1:  procedure GROW_AMOUNT(ΔC_tot)
2:      reserved ← max_capacity()
3:      committed ← capacity()
4:      uncommitted ← reserved − committed
5:      expand_via_pct ← G1ExpandByPercentOfAvailable × uncommitted
6:      min_expand ← min(RegionSize, uncommitted)
7:      resize_bytes ← min(expand_via_pct, committed)
8:      scale_factor ← scale_total_cost_delta(ΔC_tot, MinScaleFactor, MaxScaleFactor)
9:      resize_bytes ← resize_bytes × scale_factor
10:     return max( min(resize_bytes, uncommitted), min_expand)
11: end procedure
```

Shrinking step. Algorithm 3 describes the computation for the shrink step. We first use G1's pause-time model to compute the number of regions needed to meet the eden target length (Line 3). Then, we subtract it from the current number of free regions to obtain the shrinkable budget (Line 4). This respects G1's pause time goal by not reducing eden below the target G1 selects. We then ,similar to growing step, we scales the normalized cost change ΔC_{tot} using a step scale with a logistic (S-shaped) function (Line 5). Finally, we convert the shrinkable regions to bytes, and multiply by the scale factor to obtain the shrink amount.

Algorithm 3 Shrinking step

```
1:  procedure SHRINK_AMOUNT(ΔC_tot)
2:      regions_to_shrink ← num_free_regions()
3:      regions_for_eden_space ← eden_target_length()
4:      regions_to_shrink −= regions_for_eden_space
5:      scale_factor ← scale_total_cost_delta(ΔC_tot, MinScaleFactor, MaxScaleFactor)
6:      resize_bytes ← regions_to_shrink · RegionSize · scale_factor
7:      return resize_bytes
8:  end procedure
```

5 Methodology

Our evaluation focuses on four questions:

(1) How much does FlexHeap improve application performance for I/O intensive applications?
(2) How do FlexHeap's resizing decisions affect the tail latency?
(3) How sensitive is FlexHeap to the (a) application phase changes, (b) storage device latency, and (c) resizing step?

For this purpose, we use the following methodology, applications and benchmarks.

Experimental platform. We use a single server equipped with two Intel(R) Xeon(R) CPU E5-2630 running at 2.4 GHz, with 16 physical cores with two threads per core. It has 256 GB of DDR4 DRAM and runs Ubuntu 20.04 with Linux kernel 5.4. The server has one 2 TB Samsung PCIe NVMe SSD and two 256 GB Samsung 850 SATA SSD. Unless otherwise stated, all experiments use the NVMe SSD. The SATA SSDs are used only in the experiment that studies the impact of higher-latency storage devices. To reduce variability, we disable swap and set the CPU scaling governor to

Table 1. Configurations of Elasticsearch workloads for each DRAM budget. G1-50 (G1 default heap resizing policy with maximum heap capped at 50% of DRAM), VG1 (Vertical G1), and FH (FlexHeap) indicate the heap size allocated to each setup under the given DRAM budget.

Dataset	Index (GB)	#Documents	Configuration A				Configuration B			
			DRAM (GB)	G1-50 (GB)	VG1 (GB)	FH (GB)	DRAM (GB)	G1-50 (GB)	VG1 (GB)	FH (GB)
Wikipedia	68	22,986,169	8	4	7	7	12	6	11	11
Cohere Vector	500	30,000,000	25	13	22	22	50	25	45	45
Dense Vector	19.5	10,000,000	4	2	3	3	6	3	5	5
OpenAI Vector	109.1	5,361,922	4	2	3	3	7	3	6	6

performance. When necessary for each experiment, we limit the available DRAM capacity using *cgroups*. We run all experiments on OpenJDK 21, since it is the latest long-term-support version supported by both Elasticsearch and Spark.

Elasticsearch. We use Elasticsearch version 9.1.1, a widely deployed search engine, and evaluate it using the ESRally benchmarking suite [15]. Our evaluation focuses on mixed workloads that perform concurrently search and document ingestion. This concurrency creates time-varying demands for heap and I/O cache that do not form distinct execution phases. ESRally runs on a separate server to avoid interference with Elasticsearch. Each experiment is repeated five times, and we report average throughput and 99th-percentile tail latency. ESRally measures ingestion throughput in documents per second, while search throughput is measured in queries per second. The Elasticsearch server is configured to use all CPU cores. We let the JVM ergonomics automatically choose the number of STW and concurrent GC threads [35]. To reflect realistic deployments, we set the DRAM budget to 5% and 10% of the total index size, following guidance that production DRAM is typically a small fraction of storage capacity due to cost and scalability [13]. For Wikipedia, Dense Vector, and OpenAI Vector workloads, we instead select the minimum DRAM that yields successful runs. Table 1 summarizes the workloads and DRAM budgets.

Spark. We use Spark [53] version 4.0.1, which is the state-of-the-art analytic framework, running the TPC-DS benchmark that contains 103 queries. Spark runs with a single executor and eight worker threads. On the server, we enable only 8 CPU cores and disable the rest to avoid interference. We allow HotSpot ergonomics to choose the number of STW and concurrent GC threads. We use 16 GB and 32 GB to cover two distinct regimes: a constrained budget that creates meaningful pressure on both the heap and the I/O cache, and a roomier budget that reduces pressure and tests whether benefits persist when more DRAM is available.

Lucene microbenchmarks. To evaluate the sensitivity of FlexHeap to the application phase behavior, storage latency, and resizing step, we design a set of controlled Lucene [30] microbenchmarks that isolate the impact of I/O and heap pressure. Lucene is the core indexing and search library behind Elasticsearch, making it a natural choice for constructing focused, fine-grained benchmarks that reflect real workload characteristics while offering greater experimental control. These workloads allow us to trigger predictable shifts in resource usage (between I/O cache and JVM heap demands) so we can observe FlexHeap's responsiveness to dynamic memory requirements.

We use a 300 GB real-world dataset [10] that contains 52.6 billion words. To build our queries, we first perform a word count to determine the exact number of frequency for each word. Then, we remove stop-words, single-character words, and non-English words. Figure 6 shows the cumulative distribution of words (after selection) by their exact frequency.

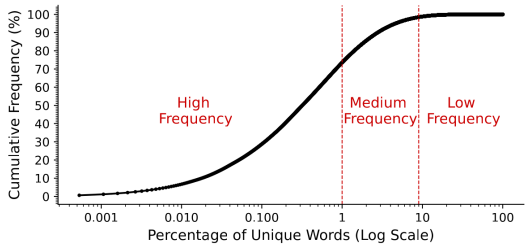


Fig. 6. Distribution of words by their exact frequency in the dataset used for Lucene microbenchmarks.

Based on these counts, we classify the words into three categories: high-frequency, medium-frequency, and low-frequency. The top 1% of words are classified as high-frequency, accounting for 80% of total occurrences. In contrast, 91% of words are low-frequency, representing only 1% of total occurrences. The remaining words fall into the medium-frequency category. These frequency thresholds help us simulate different levels of I/O load during query processing. High-frequency words appear in more documents within the Lucene index, resulting in higher I/O due to the need to access more documents. In contrast, low-frequency words generate less I/O. By using these frequency categories, we can effectively simulate varying I/O loads and analyze their impact on query performance.

We create small and large queries that fetch the top 50 or top 500 000 results, respectively. Unlike small queries, large queries require more memory in the managed heap because they create more memory pressure. On the one hand, the frequency dimension (i.e., high, medium, and low) affects I/O. Conversely, the number of results affects pressure in the managed heap. Table 2 shows the performance implications of DRAM allocation between the managed heap and the I/O cache under different query loads. Note that we omit running low-frequency queries with 500 000 results since their results are typically fewer than 1000.

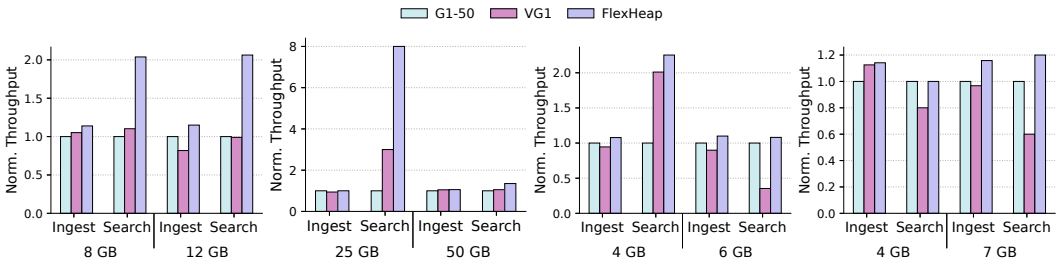
Table 2. Description of the mixes of Lucene queries.

	Top 50			Top 500,000	
	High (H50)	Medium (M50)	Low (L50)	High (H500k)	Medium (M500k)
Number of Queries	40,000	80,000	240,000	300	6,000
GC overhead (%)	0.1	0.2	0.7	21	30
I/O per query (MB)	30	10	3	31	2

Table 3. Configuration of Lucene workloads.

	Id	Queries description
Sequential	M1	H50-M500k-L50-H500k-M50
Concurrent	M2	H50-M500k-H500k
	M3	M50-M500k-H500k
	M4	L50-M500k-H500k

As shown in Table 3, we construct four experimental configurations using the query categories described in Table 2. M1 executes queries of different types sequentially without mixing them. All queries in H50 (high frequency top 50 results) run before any of the M500k queries, etc. Queries in each type are executed from 32 clients. For the last three configurations (concurrent), we use 16 clients to execute the small queries, and another 16 clients execute the large queries concurrently. The large queries arrive in batches based on a Poisson distribution, which aims to reflect the random nature of real-world query arrival rates. This approach helps us understand how the system handles a diverse and concurrent query load, reflecting real-world usage more accurately. To investigate future setups where the index size is significantly larger than the available DRAM, we use two different DRAM budgets: one where the DRAM budget equals 5% and another where it equals 10% of the index size as stored on the disk. We allow OpenJDK ergonomics to set up the number of STW and concurrent GC threads.



(a) Wikipedia (b) Cohere Vector (c) Dense Vector (d) OpenAI Vector

Fig. 7. Ingestion and search throughput for Elasticsearch workloads under G1-50 (maximum heap cap of 50% of DRAM), VG1, and FlexHeap.

Baselines. We compare FlexHeap against two classes of dynamic heap resizing: G1 (GC-based) and VG1 (occupancy-based, time-driven). To compare against the latest G1 heap resizing mechanism, we port its resizing policy from OpenJDK 26 to OpenJDK 21. VG1 remains unchanged across JDK versions. For Elasticsearch, we follow the official guidelines [14] and cap G1 at 50% of DRAM (G1-50), leaving the remainder to the I/O cache. We omit G1-90 from the main Elasticsearch evaluation because, as shown in Section § 3, it consistently results in longer execution time than G1-50. Accordingly, G1-50 serves as the strongest G1 baseline recommended for deployment. For Spark, we cap the maximum heap at 75% of DRAM (G1-75), following the guidelines of Spark 4.0.1 [4]. We configure VG1 to trigger a GC every second to resize the heap and release memory back to the OS, as in prior work [7]. In Elasticsearch, VG1's maximum heap is capped at 90% of the DRAM budget, whereas in Spark it is capped at 75%. FlexHeap uses the same maximum heap cap as VG1 for each workload. We note that in Spark, all systems use the same maximum heap cap, therefore, any differences are due to the heap resizing policy.

6 Evaluation

6.1 Performance Compared to GC-Based Resizing Mechanisms

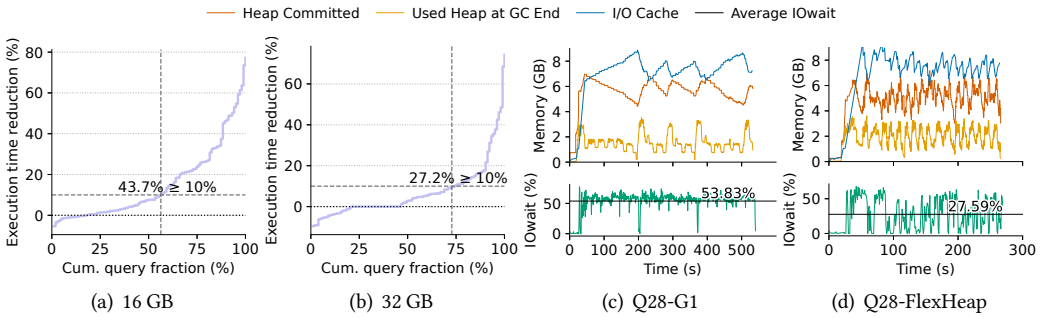
Figure 7 depicts ingestion and search throughput for Elasticsearch under G1-50 and FlexHeap, corresponding to the first and third bars in each group. The results are normalized to G1-50 (higher is better). FlexHeap improves ingestion throughput by 5–20% over G1-50 by allowing more DRAM to flow to the heap during ingestion phases with high allocation pressure, which reduces GC frequency and avoids full GCs caused by the 50% cap. For example, in the Wikipedia workload with an 8 GB DRAM budget, G1-50 incurs 17 full GCs during ingestion whereas FlexHeap incurs none. FlexHeap also improves search throughput by up to 7×, driven by lower device I/O, with read traffic reduced by up to 30% in the dense vector workload, as shown in Table 4. For example, in the Wikipedia workload the average I/O wait falls from 23% (G1-50) to 17% with FlexHeap, allowing the I/O cache to serve more requests and sustain a higher query rate.

Next, we evaluate FlexHeap against G1 in Spark. Figures 8(a) and 8(b) report per-query speedups on TPC-DS with 16 GB and 32 GB DRAM budgets. FlexHeap delivers more than 10% speedups on roughly 40% of queries with a 16 GB DRAM budget and on 27% of queries with 32 GB. The gains are smaller at 32 GB because the larger DRAM budget leaves a larger I/O cache even under G1, reducing the headroom for FlexHeap to improve I/O performance. The gains correlate with lower device pressure: read I/O drops by up to 30%, because FlexHeap reassigns DRAM from the heap to the I/O cache during I/O-intensive phases and grows the heap when allocation pressure rises. As a concrete example highlighted in Figures 8(c) and (d), q28 improves by 50% when FlexHeap

Table 4. Read and write I/O traffic for Elasticsearch workloads under G1-50 (maximum heap cap of 50% of DRAM), VG1, and FlexHeap (FH) in small DRAM budget configurations.

	Read Traffic					Write Traffic				
	G1-50 (GB)	VG1 (GB)	FH (GB)	Diff FH vs G1-50	Diff FH vs VG1	G1-50 (GB)	VG1 (GB)	FH (GB)	Diff FH vs G1-50	Diff FH vs VG1
Wikipedia	1677	3442	1444	**-14%**	**-57%**	637	587	605	**-5%**	+3%
Cohere Vector	29862	27100	23636	**-21%**	**-13%**	331	336	334	+0.9%	**-0.6%**
Dense Vector	189	468	134	**-29%**	**-71%**	79	79	85	+7%	+7%
OpenAI	716	700	638	**-10%**	**-8%**	467	435	432	**-7%**	**-0.7%**

maintains a smaller committed heap throughout the run, which preserves a larger I/O cache and cuts average I/O wait by about 30%. For queries where FlexHeap matches G1, the traces show little or no I/O bottleneck, so I/O cache size is not the limiting factor and both mechanisms exhibit similar I/O traffic.



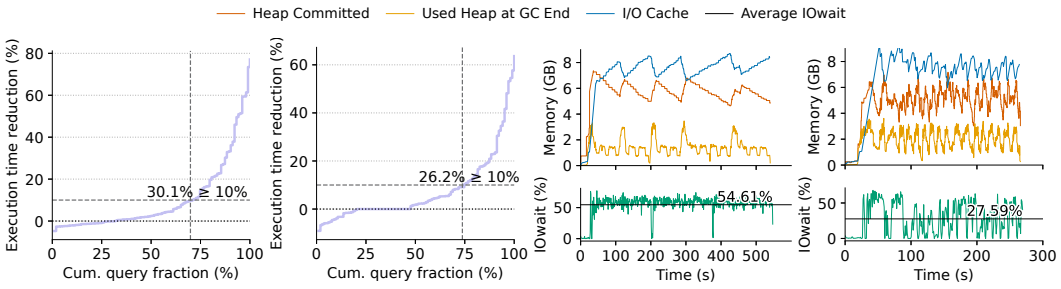
(a) 16 GB (b) 32 GB (c) Q28-G1 (d) Q28-FlexHeap

Fig. 8. Performance improvement of FlexHeap over G1-75 (maximum heap cap 75% of DRAM) heap resizing in Spark-SQL on TPC-DS.

6.2 Performance Compared to Occupancy-Based Resizing Mechanisms

We investigate the performance of FlexHeap compared to VG1 in Elasticsearch. As shown in Figure 7 (second and third bars in each group), across all four workloads, ingestion throughput under FlexHeap is higher than or comparable to VG1. For Wikipedia and OpenAI Vector, ingestion throughput is similar between the two policies. For Dense Vector and Cohere Vector, FlexHeap outperforms VG1 in ingestion by up to 30%. These workloads perform updates to the index during ingestion, which include not only writes but also reads of existing segments; these updates benefit from a larger I/O cache. For search operations, FlexHeap delivers substantially higher throughput compared to VG1, by up to 2× in Wikipedia, and by up to 5× in Cohere Vector. During the period when search runs alongside ingestion, FlexHeap shrinks the heap in response to I/O pressure and reserves more DRAM for the I/O cache, raising cache hit rates and reducing I/O wait for queries. For example, as shown in Table 4, the read traffic to the device improves up to 71% compared to FlexHeap. VG1 keeps the heap near its maximum heap size in this phase (as shown in our motivation), which starves the I/O cache and limits search throughput despite similar ingestion rates.

Next, we show how FlexHeap affects the performance of TPC-DS queries in Spark over VG1 (Figures 9(a, b)). Using FlexHeap, about 30% and 26% of queries have execution time reduction compared to VG1 for 16 GB and 32 GB DRAM budgets, respectively. Most of the improvements



(a) 16 GB (b) 32 GB (c) Q28-VG1 (d) Q28-FlexHeap

Fig. 9. Performance improvement of FlexHeap compared to VG1 in Spark-SQL on TPC-DS.

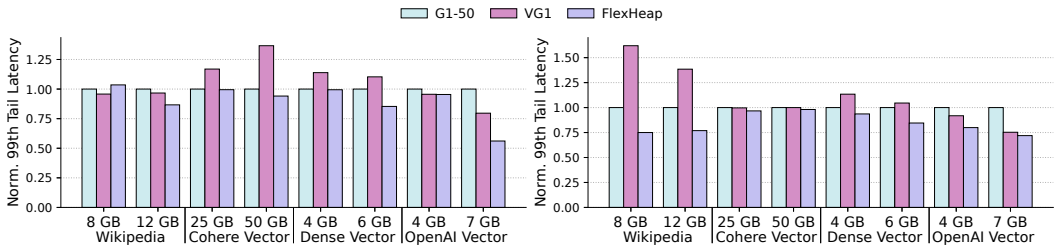
come from reducing I/O wait time. To explain this pattern, we show what happens during execution of q28 (Figures 9(c, d)). Under VG1, illustrated in Figure 9(c), the heap committed memory increases and stays high, the I/O cache is small with a sawtooth pattern, and I/O-wait stays elevated across the run. Under FlexHeap, depicted in Figure 9(d), the heap is kept lower and adjusted more often, the I/O cache is larger across the run, and I/O-wait collapses to short bursts. These decisions reduce average I/O wait time from 54% to 27%, reducing the query's wall time by roughly 2×.

6.3 Effects on Tail-Latency for Elasticsearch

We investigate how the decisions of FlexHeap affect the p99 tail-latency of Elasticsearch queries compared to G1 and VG1. Figure 10(a, b) shows the p99 latency of ingestion and search tasks for all the workloads, respectively.

For ingestion tasks, with larger DRAM budgets, FlexHeap lowers p99 latency compared to G1-50 because it can grow the heap (up to 90% of DRAM) when allocation pressure spikes, avoiding queueing and cap-induced full GCs. On Cohere-Vector, FlexHeap reduces p99 by about 10% compared to G1-50 and by up to 30% compared to VG1. VG1's periodic reclamation rarely triggers during ingestion (the heap is actively used) so its latency typically sits near or slightly above G1's. With smaller budgets, FlexHeap still reduces p99 by 5–10% compared to G1, chiefly by cutting GC pressure (in our traces, young/mixed GC counts often drop by about 2×) and eliminating cap-driven full GCs. Moreover, because VG1 keeps the heap near its maximum heap size and leaves less DRAM for the I/O cache, Elasticsearch's segment flushes during ingestion run against a smaller write-back buffer and take longer. By preserving more I/O cache, FlexHeap reduces flushes and further reducing tail latency.

For search tasks, FlexHeap lowers p99 across all DRAM budgets because it reacts to I/O stalls by shrinking the heap and enlarging the I/O cache, which increases cache residency, reduces device read traffic, and shortens queueing at the storage layer. The effect is strongest with small budgets: on Wikipedia at 8 GB, VG1's p99 is about 1.6× higher than G1 because it keeps the heap near the cap and starves the cache, while FlexHeap is about 25% below G1. At 12 GB, FlexHeap remains lower than both mechanisms. On OpenAI Vector, VG1 sometimes improves latency relative to G1 because it returns memory to the OS more often after the ingestion phase. FlexHeap improves performance further because its shrink decisions consider I/O stalls rather than occupancy targets, resulting in bigger steps of memory release.

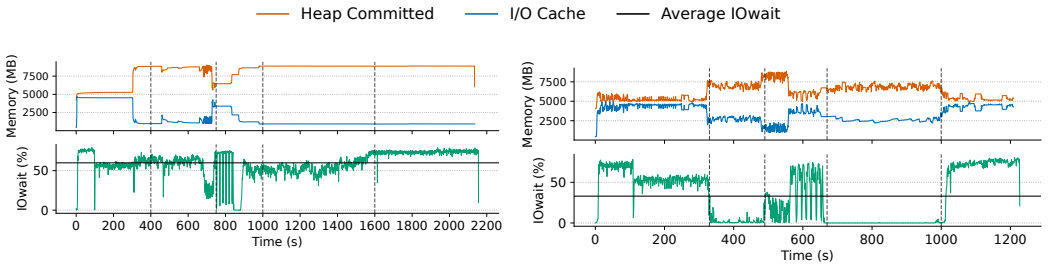


(a) Ingestion (b) Search

Fig. 10. P99 latency of (a) ingestion and (b) search tasks for Elasticsearch with G1-50 (maximum heap cap 50% of DRAM), VG1, and FlexHeap.

6.4 Sensitivity Analysis

Sensitivity to application phase changes. Here, we investigate how FlexHeap adapts to application phases using M1 workload of Lucene. Compared to G1 (Figure 11(a)), FlexHeap (Figure 11(b)) tracks the application's five phases much more closely by continuously rebalancing heap and I/O cache memory. Under G1, the heap quickly grows near its maximum and stays there, pushing the I/O cache down and keeping it low, whereas FlexHeap shrinks and regrows the heap at each phase boundary, leaving several additional gigabytes available in the I/O cache for I/O. This adaptation shortens multiple phases: the first phase finishes in about 350 s with FlexHeap versus 400 s with G1, and the second phase is dramatically shorter (150 s versus 350 s). The long final phase is also reduced from roughly 600 s with G1 to about 200 s with FlexHeap. By maintaining a significantly larger I/O cache and reacting quickly to phase changes, FlexHeap reduces I/O stalls and GC overhead, which directly translates into a substantially lower execution time by 40%.

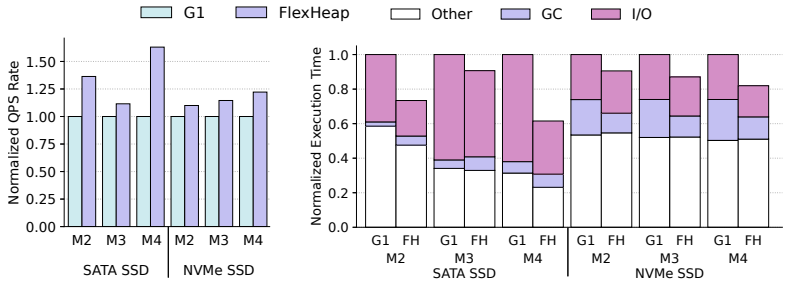


(a) G1-90 (b) FlexHeap

Fig. 11. Execution of M1 workload over time using (a) G1-90 and (b) FlexHeap with maximum heap cap 90% of DRAM budget.

Sensitivity to device latency. We investigate the benefits of using FlexHeap when Lucene utilizes SATA SSDs, which have higher latency than NVMe SSDs but are more cost-effective for storing Lucene's index. SATA SSDs differ from NVMe SSDs in terms of interface and speed. NVMe SSDs use the PCIe interface, which provides faster data transfer speed than SATA SSDs that use the older SATA interface. Also, the read and write throughput of our NVMe SSD is 6× and 4× higher compared to SATA SSD, respectively.

Figures 12(a, b) illustrate the throughput and the GC and I/O wait times for Lucene using SATA SSD and NVMe SSD with a 5 GB DRAM budget, respectively. FlexHeap consistently outperforms



(a) Throughput (b) GC and IO wait time

Fig. 12. Lucene (a) throughput and (b) execution time breakdown, comparing G1 and FlexHeap (FH) on SATA SSD and NVMe SSD using 5 GB DRAM budget.

G1 across all workloads for both types of SSDs. Notably, the throughput improvement is more pronounced with SATA SSDs, where FlexHeap increases throughput by up to 60% compared to G1, highlighting its effectiveness in environments with higher latency storage. As shown in Figure 12(b), in experiments with SATA SSD, FlexHeap increases GC time by up to 2.1× compared to G1 because it prioritizes reducing I/O overhead due to the higher latency of the device. However, with the reduces latency of NVMe SSDs, the GC overhead becomes a more significant bottleneck compared to I/O. Consequently, FlexHeap shifts focus to targeting GC overhead, reducing GC time by up to 46% compared to G1. By dynamically targeting both I/O and GC overheads, FlexHeap adapts effectively to the changing device latency, increasing performance.

Sensitivity to resizing step. We evaluate FlexHeap's sensitivity to reallocation step sizes using the M1 workload of Lucene with 5 GB, 10 GB, 18 GB, and 27 GB of DRAM (Figure 13). The optimal static step varies across DRAM budgets, making offline configuration challenging. In contrast, the adaptive approach consistently outperforms the best static steps in most cases, with only a 5% slowdown in the 10 GB configuration. A static 80% step results in up to 7.7× higher GC time, highlighting the efficiency of adaptive reallocation.

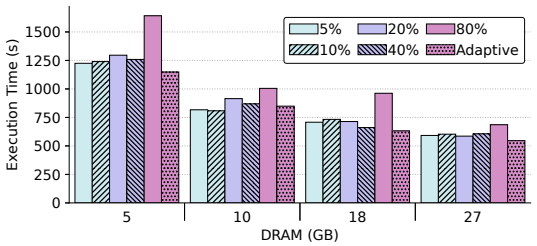


Fig. 13. Execution time of M1 workload in Lucene using static and adaptive reallocation steps for different DRAM budgets.

7 Related Work

Heuristic heap sizing. Brecht et al. [6] propose a heuristic based resizing mechanism for expanding the heap in Boehm-Demers-Weiser GC [5]. They aim to reduce GC cost while avoiding paging. The heap grows by different amounts, depending on its current size in relations to a set of predefined hand-tuned threshold values. PAMM [54] exploits repetition in program phase

behavior to manage memory adaptively. However, their technique does not apply to complex real-life big data frameworks. IslaVista [17] proposes a heap resizing policy to avoid GC-induced paging using OS information. Heap grows linearly as there are not allocation stalls and shrinks aggressively when allocation stalls are detected. Bruno et al. [7] propose a sizing policy enabling JVM to dynamically grow and shrink the heap according to the cloud application requirements. Cook et al. [9] and Simão et al. [40] study how the GC heap resizing policy affects the performance of the application. Tavakolisomeh et al. [45] propose the heap size adjustment based on the CPU time spent on GC. This approach focuses on minimizing CPU overhead for concurrent collectors, ensuring efficient memory use without stalling application performance. In contrast, FlexHeap tackles a broader challenge by dynamically partitioning DRAM between the heap and I/O cache, reducing the combined overhead of GC time and I/O wait time.

Explicit model-based heap sizing. Yang et al. [52] proposed an analytical model to adjust heap size in multi-tenant environments. The model monitors application memory allocation and footprint, periodically changing heap size to minimize GC cost and paging. However, their approach needs OS modifications. Sun et al. [44] tackle the issue of memory management for many applications operating under a single JVM by proposing segregated, per-application heaplets. They present an analytical model that modifies heaplet sizes to equilibrate GC frequency among all applications, so preventing any single application from unduly impacting the system's overall performance. Tay et al. [46] formulate a page fault equation that correlates the frequency of page faults with heap size and resident heap size, predicated on established GC and OS rules for a certain program input. They propose a heap sizing rule that reduces page faults. However, alterations in input or system load necessitate parameter recalibration, hence constraining its adaptability. Vengerov [48] creates a mathematical model to optimize throughput in generational heaps by tuning GC parameters. Singer et al. [41] apply microeconomic supply and demand theory to model GC behavior, using elasticity to guide heap size expansion. However, they require from users to set a target elasticity value which is not intuitive. MemBalancer [21] proposes a model-based heap resizing policy for V8 that derives a compositional square-root heap-limit rule. Its key idea is to coordinate heap allocations across multiple concurrent runtimes under a shared memory budget so as to optimize the trade-off between total GC time and memory usage. Unlike these approaches, FlexHeap does not rely on an explicit analytical or predictive model of heap behavior. Instead, it uses a model-free, feedback-driven controller based on observed changes in GC and I/O cost, which avoids assuming a fixed heap–I/O relationship across workloads and storage environments.

Control-theoretic heap sizing. White et al. [51] suggest a Proportional-Integral-Derivative (PID) controller that monitors the GC overhead and modifies the heap resize ratio to uphold a user-defined target GC goal. Storm et al. [43] examine autonomic database configuration through control theory to create a self-tuning memory manager that dynamically modifies heap sizes in databases. Their methodology employs separate heaps for different memory-intensive operations, including SQL cache and buffer pool. The suggested strategy employs a cost/benefit estimation model to adjust the size of each heap, aiming to equilibrate the cost/benefit metrics among all heaps. They utilize a multi-input multi-output (MIMO) controller with integral control for tuning, highlighting the advantages of this controller-based approach, which encompasses rapid convergence, swift adaptability, and stable responses to noise.

Cross-runtime heap sizing. Prior work [2, 8, 18, 49] has studied dynamic heap sizing in shared environments, adjusting heap sizes across multiple managed processes or VMs to reduce paging or improve overall throughput. ElasticMem [49], for example, reclaims heap memory from data analytics applications and returns it to the OS for use by other processes. Our work differs in both

scope and abstraction. Prior systems operate at the JVM abstraction boundary, modeling heap size as a function of throughput without distinguishing the causes of performance degradation. FlexHeap, in contrast, explicitly measures I/O stall time and separates GC displacement from blocked mutator progress. This cross-layer attribution lies outside the black-box interface assumed by prior multi-runtime allocators, and is important because heap–I/O interactions are mediated by I/O cache dynamics rather than by heap size alone.

I/O cache and application coordination. P2Cache [27] proposes an application-specific I/O cache that enables developers to build a custom I/O cache that aligns with I/O characteristics of their application. Symbiosis [11] dynamically configures the size of user-space block caches in persistent key-value stores with the I/O cache to improve their performance, requiring offline profiling. Unlike these systems, FlexHeap focuses on the dynamic DRAM division between the JVM heap and I/O cache without application knowledge or offline profiling. Also, FlexHeap could be applicable for systems [22–25, 31] that use I/O cache to extend the JVM heap over slower memory tiers, such as fast storage devices or remote memory.

8 Conclusions

Popular search engines and analytic frameworks running on JVM must carefully balance memory between the JVM heap and the I/O cache to achieve high performance. Existing heap resizing mechanisms focus solely on reducing GC overhead, neglecting the cost of I/O stalls. In this paper, we propose FlexHeap, a heap resizing mechanism for G1 that dynamically partitions memory between the heap and the I/O cache. Between GC intervals, FlexHeap estimates the CPU time lost to GC and to I/O stalls, and repartitions DRAM to reduce their combined cost. It makes frequent decisions at G1 collection boundaries, relies on a history-based model to project future GC and I/O stall costs, and adapts its resizing step proportional to the magnitude of changes in that combined cost. Our evaluation shows that FlexHeap improves throughput in Elasticsearch and Spark by an average 30% and 33%, respectively. Also, FlexHeap outperforms VG1 by on average 50%. These results demonstrate that I/O-aware heap resizing is important for improving performance in modern JVM-based search and analytic applications.

9 Data Availability Statement

The artifact containing the code and the benchmarks of this paper is available at https://github.com/CARV-ICS-FORTH/flexheap or alternatively a static version is available at https://doi.org/10.5281/zenodo.19183591.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback, which helped improve the final version of this paper. This work was partially supported by the European Union project AERO (grant agreement No. 101092850), the EUPEX project (grant agreement No. 101033975) through the European High-Performance Computing Joint Undertaking (JU)[1], and VMware's University Research Fund.

References

[1] 2024. *Elasticsearch: The Official Distributed Search & Analytics Engine | Elastic.* (Accessed: January 2025).

[2] Raphael Alonso and Andrew W. Appel. 1990. An advisor for flexible working sets. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Univ. of Colorado, Boulder, Colorado, USA) *(SIGMETRICS '90)*. Association for Computing Machinery, New York, NY, USA, 153–162. doi:10.1145/98457.98753

[1]The JU receives support from the European Union's Horizon 2020 research and innovation programme, as well as from France, Germany, Italy, Greece, the United Kingdom, Czech Republic, and Croatia.

[3] Apache Cassandra. 2025. Hardware Choices | Apache Cassandra Documentation. https://cassandra.apache.org/doc/latest/cassandra/managing/operating/hardware.html. Accessed: April 2026.

[4] Apache Spark. 2025. Hardware Provisioning - Spark 4.0.1 Documentation. https://spark.apache.org/docs/latest/hardware-provisioning.html. Accessed: April 2026.

[5] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Software: Practice and Experience* 18, 9 (1988), 807–820. doi:10.1002/spe.4380180902

[6] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. 2006. Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications. *ACM Trans. Program. Lang. Syst.* 28, 5 (Sept. 2006), 908–941. doi:10.1145/1152649.1152652

[7] Rodrigo Bruno, Paulo Ferreira, Ruslan Synytsky, Tetiana Fydorenchyk, Jia Rao, Hang Huang, and Song Wu. 2018. Dynamic Vertical Memory Scalability for OpenJDK Cloud Applications. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management* (Philadelphia, PA, USA) *(ISMM '18)*. Association for Computing Machinery, New York, NY, USA, 59–70. doi:10.1145/3210563.3210567

[8] Callum Cameron, Jeremy Singer, and David Vengerov. 2015. The judgment of forseti: economic utility for dynamic heap sizing of multiple runtimes. In *Proceedings of the 2015 International Symposium on Memory Management* (Portland, OR, USA) *(ISMM '15)*. Association for Computing Machinery, New York, NY, USA, 143–156. doi:10.1145/2754169.2754180

[9] Jonathan E. Cook, Artur W. Klauser, Alexander L. Wolf, and Benjamin G. Zorn. 1996. Semi-Automatic, Self-Adaptive Control of Garbage Collection Rates in Object Databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada) *(SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 377–388. doi:10.1145/233269.233354

[10] Common Crawl. 2022. Common Crawl - Blog - January 2022 crawl archive now available. https://commoncrawl.org/blog/january-2022-crawl-archive-now-available. (Accessed: January 2025).

[11] Yifan Dai, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2024. Symbiosis: The Art of Application and Kernel Cache Cooperation. In *22nd USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST '24)*. USENIX Association, Berkeley, CA, USA, 51–69. https://www.usenix.org/conference/fast24/presentation/dai

[12] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 37–48. doi:10.1145/1029873.1029879

[13] Elastic Cloud. 2018. Sizing Hot-Warm Architectures for Logging and Metrics in the Elasticsearch Service on Elastic Cloud. https://www.elastic.co/blog/sizing-hot-warm-architectures-for-logging-and-metrics-in-the-elasticsearch-service-on-elastic-cloud. Accessed: April 2026.

[14] Elasticsearch. 2025. JVM Settings | Reference. https://www.elastic.co/docs/reference/elasticsearch/jvm-settings. Accessed: April 2026.

[15] ESRally. 2025. Rally 2.12.0 Documentation. https://esrally.readthedocs.io/en/stable/. Accessed: April 2026.

[16] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. 2022. TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs. In *16th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI '22)*. USENIX Association, Berkeley, CA, USA, 395–411. https://www.usenix.org/conference/osdi22/presentation/feng

[17] Chris Grzegorczyk, Sunil Soman, Chandra Krintz, and Rich Wolski. 2007. Isla Vista Heap Sizing: Using Feedback to Avoid Paging. In *Proceedings of the International Symposium on Code Generation and Optimization* (San Jose, California, USA) *(CGO '07)*. IEEE Computer Society, Washington, DC, USA, 325–340. doi:10.1109/CGO.2007.20

[18] Matthew Hertz, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Chen Ding, Xiaoming Gu, and Jonathan E. Bard. 2011. Waste not, want not: resource-based garbage collection in a shared environment. In *Proceedings of the International Symposium on Memory Management* (San Jose, California, USA) *(ISMM '11)*. Association for Computing Machinery, New York, NY, USA, 65–76. doi:10.1145/1993478.1993487

[19] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. 2015. Unified Address Translation for Memory-mapped SSDs with FlashMap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 580–591. doi:10.1145/2749469.2750420

[20] Richard Jones, Antony Hosking, and Eliot Moss. 2023. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (2nd ed.). Chapman & Hall/CRC, Boca Raton, FL, USA. doi:10.1201/9781003276142

[21] Marisa Kirisame, Pranav Shenoy, and Pavel Panchekha. 2022. Optimal heap limits for reducing browser memory use. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 160 (Oct. 2022), 21 pages. doi:10.1145/3563323

[22] Iacovos G. Kolokasis, Konstantinos Delis, Shoaib Akram, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2025. SmartSweep: Efficient Space Reclamation in Tiered Managed Heaps. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Singapore, Singapore) *(MPLR '25)*. Association for Computing Machinery, New York, NY, USA, 71–78. doi:10.1145/3759426.3760981

[23] Iacovos G. Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2023. 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, Volume 3* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 694–709. doi:10.1145/3582016.3582045

[24] Iacovos G. Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2024. TeraHeap: Exploiting Flash Storage for Mitigating DRAM Pressure in Managed Big Data Frameworks. *ACM Trans. Program. Lang. Syst.* 46, 4, Article 12 (Dec. 2024), 37 pages. doi:10.1145/3700593

[25] Iacovos G. Kolokasis, Anastasios Papagiannis, Polyvios Pratikakis, Angelos Bilas, and Foivos Zakkak. 2020. Say Goodbye to Off-Heap Caches! On-Heap Caches Using Memory-Mapped I/O. In *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage '20)*. USENIX Association, Berkeley, CA, USA, Article 4, 1 pages. https://www.usenix.org/conference/hotstorage20/presentation/kolokasis

[26] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. doi:10.1145/1773912.1773922

[27] Dusol Lee, Inhyuk Choi, Chanyoung Lee, Sungjin Lee, and Jihong Kim. 2023. P2Cache: An Application-Directed Page Cache for Improving Performance of Data-Intensive Applications. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage '23)*. Association for Computing Machinery, New York, NY, USA, 31–36. doi:10.1145/3599691.3603408

[28] Linux Kernel. [n. d.]. Documentation/filesystems/proc.rst. https://kernel.googlesource.com/pub/scm/linux/kernel/git/stable/linux-stable/+/refs/tags/v5.8.4/Documentation/filesystems/proc.rst. [Accessed: April 2026].

[29] Ioannis Malliotakis, Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2020. HugeMap: Optimizing Memory-Mapped I/O with Huge Pages for Fast Storage. In *Euro-Par 2020: Parallel Processing Workshops: Euro-Par 2020 International Workshops, Revised Selected Papers* (Warsaw, Poland) *(Euro-Par '20)*. Springer-Verlag, Berlin, Heidelberg, 344–355. doi:10.1007/978-3-030-71593-9_27

[30] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. 2010. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* Manning Publications Co., USA.

[31] Dat Nguyen and Khanh Nguyen. 2024. Polar: A Managed Runtime with Hotness-Segregated Heap for Far Memory. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems* (Kyoto, Japan) *(APSys '24)*. Association for Computing Machinery, New York, NY, USA, 15–22. doi:10.1145/3678015.3680490

[32] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to Zombieland: Practical and Energy-Efficient Memory Disaggregation in a Datacenter. In *Proceedings of the Thirteenth European Conference on Computer Systems* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3190508.3190537

[33] OpenJDK. 2019. JEP 351: ZGC: Uncommit Unused Memory (Experimental). https://bugs.openjdk.org/browse/JDK-8220347. Accessed: April 2026.

[34] OpenJDK. 2023. JEP 346: Promptly Return Unused Committed Memory from G1. https://openjdk.org/jeps/346. Accessed: April 2026.

[35] Oracle. 2025. Garbage-First Garbage Collector Tuning. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc_tuning.html. Accessed: April 2026.

[36] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2021. Memory-Mapped I/O on Steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 277—-293. doi:10.1145/3447786.3456242

[37] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-Mapped I/O for Fast Storage Devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '20)*. USENIX Association, Berkeley, CA, USA, 813–824. https://www.usenix.org/conference/atc20/presentation/papagiannis

[38] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. 2022. DAOS: Data Access-Aware Operating System. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 4–15. doi:10.1145/3502181.3531466

[39] Liz Rice. 2023. *Learning eBPF.* " O'Reilly Media, Inc.".

[40] José Simão and Luís Veiga. 2013. Adaptability driven by quality of execution in high level virtual machines for shared cloud environments. *International Journal of Computer Systems Science & Engineering* 28, 6, Article 1 (Nov. 2013), 14 pages.

[41] Jeremy Singer, Richard E. Jones, Gavin Brown, and Mikel Luján. 2010. The Economics of Garbage Collection. In *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada) *(ISMM '10)*. Association for Computing Machinery, New York, NY, USA, 103–112. doi:10.1145/1806651.1806669

[42] Kedar Sovani. 2005. Kernel korner: sleeping in the kernel. *Linux Journal* 2005, 137 (2005), 11.

[43] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive Self-tuning Memory in DB2. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, Los Angeles, CA, USA, 1081–1092.

[44] Kewei Sun, Ying Li, Matt Hogstrom, and Ying Chen. 2006. Sizing Multi-space in Heap for Application Isolation. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 647–648. doi:10.1145/1176617.1176654

[45] Sanaz Tavakolisomeh, Marina Shimchenko, Erik Österlund, Rodrigo Bruno, Paulo Ferreira, and Tobias Wrigstad. 2023. Heap Size Adjustment with CPU Control. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Cascais, Portugal) *(MPLR '23)*. Association for Computing Machinery, New York, NY, USA, 114–128. doi:10.1145/3617651.3622988

[46] Y.C. Tay and X.R. Zong. 2010. A Page Fault Equation for Dynamic Heap Sizing. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering* (San Jose, California, USA) *(WOSP/SIPEW '10)*. Association for Computing Machinery, New York, NY, USA, 201–206. doi:10.1145/1712605.1712636

[47] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, 14 pages. doi:10.1145/3342195.3387517

[48] David Vengerov. 2009. Modeling, Analysis and Throughput Optimization of a Generational Garbage Collector. In *Proceedings of the 2009 International Symposium on Memory Management* (Dublin, Ireland) *(ISMM '09)*. Association for Computing Machinery, New York, NY, USA, 1–9. doi:10.1145/1542431.1542433

[49] Jingjing Wang and Magdalena Balazinska. 2017. Elastic Memory Management for Cloud Data Analytics. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 745–758. https://www.usenix.org/conference/atc17/technical-sessions/presentation/wang

[50] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanee, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. doi:10.1145/3503222.3507731

[51] David R. White, Jeremy Singer, Jonathan M. Aitken, and Richard E. Jones. 2018. Control Theory for Principled Heap Sizing. In *Proceedings of the 2013 International Symposium on Memory Management* (Seattle, Washington, USA) *(ISMM '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/2464157.2466481

[52] Ting Yang, Matthew Hertz, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2004. Automatic Heap Sizing: Taking Real Memory into Account. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 61–72. doi:10.1145/1029873.1029881

[53] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA, USA) *(HotCloud '10)*. USENIX Association, Berkeley, CA, USA, Article 10, 10 pages. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[54] Chengliang Zhang, Kirk Kelsey, Xipeng Shen, Chen Ding, Matthew Hertz, and Mitsunori Ogihara. 2006. Program-Level Adaptive Memory Management. In *Proceedings of the 5th International Symposium on Memory Management* (Ottawa, Ontario, Canada) *(ISMM '06)*. Association for Computing Machinery, New York, NY, USA, 174–183. doi:10.1145/1133956.1133979

Received 2025-11-13; accepted 2026-04-03