DynaHeap: Dynamic Division of DRAM between Heterogeneous Managed Heaps

Iacovos G. Kolokasis†‡
FORTH-ICS, Greece (student)

Shoaib Akram†
ANU, Australia

Foivos S. Zakkak
Red Hat, Inc.

Polyvios Pratikakis‡
FORTH-ICS, Greece

Angelos Bilas‡
FORTH-ICS, Greece

1 Motivation

As data grows fast, managed big data frameworks like Spark [21], Giraph [12], and Flink [6], need to process larger datasets than server memory. However, the DRAM available within each single server scales slower than the data growth rate due to physical scaling limitations [11]. For this purpose, existing solutions extend the managed heap of big data applications over block-addressable fast storage devices (e.g., NVMe SSDs) [3, 4, 7, 8, 14], byte-addressable non-volatile memories (NVM) [1, 2, 9, 15, 18–20], or remote memory [10, 16, 17]. While these alternatives offer higher capacity than DRAM, they have higher latency and lower throughput, constituting a slower memory tier. Existing systems use three main strategies for organizing the managed heap over the fast and slow memory tiers: (1) uniform managed heap with caching, (2) partitioned managed heap without caching, and (3) partitioned managed heap with caching.

Systems in the first category [10, 16, 17], shown in Figure 1(a), allocate the managed heap over the slow tier and use the fast tier as a cache. The OS hides memory tiers’ heterogeneity and transparently fetches objects from the slow to the fast tier. This transparency eliminates the need for the JVM to maintain extra data structures to track objects’ locations within the memory hierarchy and to adjust object references during promotions or demotions between the tiers. However, this approach leads to high GC cost because the garbage collector scans objects in the slow tier, resulting in excessive swapping.

Systems in the second category [1, 2, 9, 15, 18, 19] partition the memory address space into fast and slow tiers, reducing swapping. They allocate the young generation and a portion of the old generation of the managed heap on the fast tier and the remaining on the slow tier (Figure 1(b)). Instead of page swapping, they explicitly move objects between the fast and the slow tiers, which requires updating their references. This reference adjustment becomes prohibitively expensive for frequent object relocation as the garbage collector must scan objects in the slow tier to update their references. Even using lazy reference adjustment with load reference barriers [18], application performance decreases [5, 13] due to the extra overhead on every load.

Systems in the last category [3, 4, 7, 8, 14] overcome reference adjustment overheads and avoid scanning the slow tier. They allocate a primary managed heap (H1) over the fast tier and a second managed heap (H2) over the slow tier (Figure 1(c)), reserving a portion of the fast tier as a cache for H2. The garbage collector maintains only cross-heap references while it avoids scanning objects in the slow tier, reducing GC time. However, these approaches divide the fast tier between H1 and the cache for H2 statically at JVM launch, leading to two main problems.

Problem #1: Requiring hand-tuning configuration. Finding which portion of the fast tier must be reserved as cache to yield good performance, requires iterative adjustments and experimentation. Users may need to search for a suitable configuration whenever they change dataset size or application. These experimentations are time-consuming and impractical in real-life deployments where applications and datasets change frequently.

Problem #2: Changing application behavior. Applications have dramatically different memory requirements at different periods. H1 should use enough space in the fast tier to avoid memory pressure and frequent GC cycles. However, increasing the cache for H2 results in faster access to objects on the slow tier. Thus, the static division of the fast tier between H1 and the cache for H2 cannot adapt to dynamic changing application behavior.

To support the point that DRAM should be divided dynamically, we perform a study using a state-of-the-art system, TeraHeap [7]. TeraHeap has a primary managed heap (H1) on DRAM and a second high-capacity managed heap (H2)
memory-mapped over an NVMe SSD. It partitions DRAM statically between H1 and the page cache for H2, at the initialization of the JVM. Figure 2(a) shows the occupancy of H1 and iowait time for an SVM workload on Spark. From 0–2500 s, SVM loads the dataset into memory and performs the algorithm’s first iteration. In this phase, the application generates a large number of objects, resulting in filling the old generation very frequently. The high peaks in iowait time between 0–2500 s are due to the transfers of objects to H2 during GC. After 2500 s, the algorithms perform iterations over the cached objects we moved to H2. Between 2500–4600 s, H1 utilization is low because SVM creates few objects; of these, 90% die in the young generation, and the remaining 10% is promoted to the old generation. However, during that time, the application demands a large page cache to reduce the I/O traffic to the device. On the other hand, as shown in Figure 2(b), CDLP running on Giraph has eight distinct periods where memory requirements vary for H1 and page cache.

2 DynaHeap Design Overview

We propose DynaHeap, a system that dynamically divides a fixed of DRAM budget between the primary heap (H1) and cache for the second heap (H2). DynaHeap treats applications as black boxes and defines distinct rules for memory tuning between H1 and the cache for H2. It determines whether to increase H1 or the cache for H2 at runtime by tracking GC and I/O time. Our design introduces three essential functions:

Tracking I/O and GC overheads. DynaHeap dynamically observes program behavior at fixed granularity intervals and repartitions DRAM between H1 and the cache for H2, to minimize the sum of overheads: GC time for H1 and I/O time for H2. It uses metrics already available by the corresponding subsystems; these are accessible at negligible overhead. We read the counters for GC and iowait time reported by the JVM and OS, respectively.

We divide time into intervals of duration \( T \). Although there is no explicit value for \( T \), \( T \) should correspond to a granularity that is required for adjusting DRAM between H1 and the cache for H2. Using a very large value for \( T \), such as tens of seconds, results in a noticeable delay in making decisions. For this purpose, we use \( T \) corresponding to the interval between each young (minor) GC cycle. Minor GC in generational garbage collectors happens when the JVM cannot allocate space for a new object. The allocation rate in big data frameworks is high, resulting in frequent minor GC cycles, on average every 5–10 s. However, taking actions at a too fine-grain granularity might lead to suboptimal decisions due to noise. Thus, DynaHeap takes decisions for DRAM repartition based on the metrics of the \( N \) last intervals.

DynaHeap calculates for each past interval the cost of the GC and I/O time. It starts by allocating almost all DRAM to H1. When GC is higher than I/O time, DynaHeap reduces GC cost by (1) moving objects to H2 or (2) growing the size of H1. In the first case, wherein a significant portion of objects within H1 are marked for relocation to H2 by the application, DynaHeap moves a subset of these objects to H2. Otherwise, DynaHeap tries to increase the capacity of H1 to postpone the next GC cycle. On the other hand, when I/O time is higher than GC time, then DynaHeap shrinks H1 to increase the size of the cache.

Calculating GC overhead. Although DynaHeap takes decisions in each interval \( T \), full GC does not happen in every interval. To compare GC cost with the I/O in each interval, DynaHeap amortizes the full GC cost over subsequent intervals, to apportion GC overheads to each interval. In short, DynaHeap estimates how many intervals will it take until the next GC cycle, and amortizes the last GC pause time over them. This amortized GC cost is then compared to current I/O time within each interval.

Tracking unused memory. Adjusting the fast tier memory between H1 and the cache for H2 may take a different amount of time for each of the two uses. That is, reducing the size of H1 to increase the cache for H2 will eventually reduce I/O overheads for H2 access. However, it may result in high memory pressure and frequent GC cycles before that happens. For this purpose, DynaHeap monitors how much of the available memory (after shrinking H1) is being used for the cache of H2. In case the caching system has not taken advantage of all the available memory yet, DynaHeap avoids extra shrinking of H1.

3 Conclusions

We implement DynaHeap on top of TeraHeap in OpenJDK8 and OpenJDK17. We evaluate DynaHeap compared to TeraHeap using two well-broad analytic frameworks, Spark and Giraph. Overall, DynaHeap performance is between 24% better and 2% worse than a hand-tuned baseline of TeraHeap.
References


[16] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Neetravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In 14th USENIX Symposium on Operating Systems Design and


