DynaHeap: Dynamic Division of DRAM between Heterogeneous Managed Heaps

Iacovos G. Kolokasis†‡ FORTH-ICS, Greece
Shoaib Akram‡ AU, Australia
Foivos S. Zakkak Red Hat, Inc.
Polyvios Pratikakis†‡ FORTH-ICS, Greece
Angelos Bilas†‡ FORTH-ICS, Greece

1 Motivation

With the high growth of data, managed big data frameworks, such as 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] 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 garbage collection (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. Instead of page swapping, they explicitly move objects between the fast and 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].

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, 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 a 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.

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.

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 H2 cache. It determines whether to increase H1 or the H2 cache at runtime by tracking GC and I/O time. DynaHeap uses an adaptation mechanism that takes decisions to repartition DRAM in each minor GC cycle when mutator threads are stopped to avoid extra pauses and synchronization overheads. Although OS memory reclamation leads to memory movement between H1 and H2 cache, this is performed on-demand, introducing an inherent delay in observing the resizing action impact. For this purpose, we extend DynaHeap adaptation mechanism to a finite state machine that includes wait states, used to stop making new decisions until their effect occurs.

†This work is partially funded by a grant from the European Union’s Horizon 2020 programme under grant agreement No 10048318 (AERO).
References


