CMIG: Concurrent Java Object Migration In Tiered Memory Systems

Tong Zhou

Abstract— **Memory technologies, such as non-volatile memory and MCDRAM are on the horizon. This work explores a data migration strategy in a managed language runtime that exploits the fact that most objects are short-lived. We propose a system that migrates objects from higher memory tier to lower tier after the objects are no longer active but before the next GC cycle to improve the throughput of the higher tier memory.**

I. Introduction

Various work [7] (todo: add more) has been done to explore data placement policies in a hybrid memory system based on offline profiling, mostly in the OS level. But none of them exploits objects lifetime and access pattern instead of profiling in managed languages that run in a VM such as Java and C#. Empirically we know that most Java objects are short-lived, that is the assumption generational GC has based on. Shaham [1] shows that the potential memory savings for SPECjvm98 range from 23% to 74% if dragged objects can be reclaimed early. Our previous experiments [2] have shown that 95% of the objects are only used in the first 100ms after creation for DaCapo [8] benchmark.

In a hybrid memory system, the large, cheap tier-two memory provides the opportunity to migrate objects after they are no longer actively used but not yet reclaimed. Unlike disk storage, tier-two memory also provides an acceptable fall-through penalty. CMIG is a system that implements the idea in Java.

II. RELATED WORK

Related works can be categorized to Java object lifetime analysis and dynamic data migration in a NUMA system (including hybrid memory system). Jones et al. [3] showed that (i) sites allocate objects with lifetimes in only a small number of narrow ranges, and (ii) sites cluster strongly with respect to the lifetime distributions of the objects they allocate. Furthermore, (iii) these clusterings are robust against the size of the input given to the program and (iv) are likely to allocate objects that are live only in particular phases of the program's execution. (v) allocation site alone is not always sufficient as a predictor of object lifetime distribution but one further level of stack context suffices. Pretenuring [4] is another direction to utilize the access pattern of Java objects but they focus on the long-lived objects.

On the other hand, dynamic page placement policy in NUMA systems is an old topic. Many data placement policies [5], [6] have been proposed in the OS level to improve data locality and avoid accessing data from a remote NUMA node.

todo: add more hybrid system work

Because hybrid system is an emerging technique, to our best knowledge no similar work exists so far that moves data from tier to tier in a hybrid memory NUMA system and in a managed language runtime such as JVM.

III. Methodology

CMIG partitions the original eden space into a migration space and a normal (non-migration) space. Objects will be selectively allocated to either of them depending on some heuristics to predict if the objects are going to be short-lived. The normal space is mapped to tier-one physical memory and the migration space is further divided into two spaces, a small tier-one space and a large tier-two space which we L1-migration space and L2-migration space. In such a system, we keep migrating data from L1-migration space to L2-migration space to utilize the short-liveness of objects. Now we first introduce a naive version of CMIG that does stop-the-world migration.

In the naive implementation, we let the whole L1-migration space fills up completely and then triggers a global safepoint. We also limits GCs behavior to simplify interaction, such as disabling the tenured space and adaptive heap resizing. A detailed configuration is given in the next section. In regards to CMIGs algorithm, the following steps are completed in order during the safepoint:

- The VM thread copies the whole L1-migration space to L2-migration space, which is assumed to be much larger so it will not fill up very quickly.
- Iterate the heap (including all the spaces) to fix pointers. Specifically, for each object, we iterate all its fields. If a field is a reference to an object, namely an oop, and the referent resides previously in L1-migration space, we update this field to point to the new copy of the referent in L2-migration space. Iterate and update the root sets.
- Resume the application and start reusing L1-migration space again.

In terms of object selection heuristics, we justs filtered a set of objects that are obviously long-lived by always letting threads of some certain kind allocate in the normal space, such as the thread that does heap initialization(the Main thread in HotSpot VM). Mirror objects for primitive types and well known classes(such as `Java.lang.Object`) are an example of such data.

To avoid the stop-the-world overhead, we now consider a concurrent version. Apart from having to bring to a global safepoint, the naive strategy also might migrate newly allocated objects that are likely to be still active. To tackle this issue, we divide the L1-migration space into two subspaces and defines the following time stamps in a concurrent migration setting:

- T1: the time when subspace 1 fills up. The subsequent allocation requests will be satisfied by subspace 2.
- T2: the time when subspace 2 is half-full. This is the time when a migration should be initiated that copies everything before T1.
- T3: the time when subspace 2 is completely full. At this point, if subspace 1 has finished migration, then subspace 1 starts to meet the allocation request. Likewise, when subspace 1 is half-full, another migration should be initiated to migrate subspace 2. If subspace 1 migration is still ongoing, bring the application to a global safepoint and wait for the migration to finish.

With such a scheme, the newly allocated objects are not immediately migrated and the issue could be avoided.

The concurrent migration algorithm is similar to a concurrent replication copying collection algorithm [6]. It consists of the following four phases:

- phase1: copy the migrated space to L2-migration space, this phase runs concurrently with the mutators.
- phase2: take a screenshot, make all TLABs at this point parsable but still allow the rest of the L1 migration space to satisfy allocation requests, this phase stops the world.
- phase3: atomically fix pointers in the parsable space, this phase is concurrent.
- phase4: atomically fix pointers in newly allocated space and the root sets, this phase stops the world.

This algorithm would require a write barrier that catches all the writes to objects that have been copied to L2-migration space by the mutators. The pointer flipping phase depends on the parsability of the heap, which has to be done during a safepoint. In the first iteration of the project, we assume the object selection heuristic is able to only select read-only objects so that the write barrier is not necessary so far.

IV. Implementation

In order to evaluate the feasibility of the idea, we fully implemented the naive version in the `ParallelGC` collector of HotSpot VM and for the concurrent version we only measured the elapsed time between one subspace fills up and the initialization of the migration as well as the time it takes to finish all migration phases, to get a sense of how quickly the migration should be.

Note that the heap is agnostic of the migration space. Upon the startup and right after each GC, the VM allocates a large contiguous space from the eden space and constructs a migration space object to manage the requested space. From the standpoint of heap, that is just like allocating a big TLAB. From this point on, any subsequent object allocation request will be handled by the migration space object. Our system does not change the organization of the heap and all the management objects are allocated in the native memory. Therefore when a GC is triggered, the heap will be iterated the same way as it normally is. But the whole migration space needs to be made parsable before the heap is iterated, because this space is managed by CMIG and is not associated with any thread as a TLAB. Also note that when non-migration space fills up, a minor GC is triggered immediately.

Also note that this algorithm assumes TLABs are enabled, but it can be easily adapted to a version that does not use TLABs.

V. Experiments

In this section, we first show the experiments setting we employed and a preliminary results.

A. Hardware

We used an Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz CPU with 2 processor sockets and 2 NUMA nodes with each node having 32 GB RAM.

B. Command Line Options

We categorize the command line options to the following categories: We used `numactl --cpunodebind=0` to simulating lower-tier memory by accessing a remote NUMA node. However, we later found out that accessing a remote NUMA node does not actually have longer latency than accessing the local node on this machine. But that does not affect the correctness of the algorithm.

- `-XX:+UseParallelGC`
  - The algorithm is implemented in the parallel scavenging collector
- `-XX:MaxNewSize=524288000`
  - The maximum size of the young generation
- `-XX:NewSize=524288000`
  - The size of the young generation
- `-XX:SurvivorRatio=8`
  - The ratio of size of eden space to one survivor space

The combination of `NewSize` and `SurvivorRatio` forces the eden space to be a fixed size of 400 MB and 50 MB for each survivor space.

Besides, we also turned on the following options to make the garbage collector's behavior more predictable:

- `-XX:+PreventTenuring`
- `-XX:-UseAdaptiveSizePolicy`
- `-XX:-UsePSAdaptiveSurvivorSizePolicy`

C. Preliminary Results

We run our customized HotSpot with the configuration above with DaCapo benchmark suite and timed each step in the naive implementation. The results showed that for all benchmarks updating pointers in the entire heap (step two) has too much overhead, which totally defeats the benefits of moving data around. Although we iterate the heap completely sequentially, which should generate pretty cache hits (as opposed to traversing a random graph), it still takes too much time. For instance, for **fop** it took 0.4 seconds to fill up another semi-migration space, but pointer fix-up phase alone needs 1.4 seconds to finish, let alone updating the root sets.

So we basically stopped here, and may continue to explore this with OS's support.

VI. Future Work

We may improve the migration algorithm to avoid iterating the whole heap.

We may explore an OS-runtime hybrid approach. Instead of scanning the entire heap to fix pointers and doing the job in the VM, if the OS could migrate data in pages and keep the virtual space unchanged, the overhead will be much less. But OS needs to have the facility to flexibly remap a virtual page to a physical page at a given memory tier.

References

[1] Ran Shaham, Elliot K. Kolodner, Mooly Sagiv Tel-Aviv: ”On effectiveness of GC in Java”, ISMM ’00 Proceedings of the 2nd international symposium on Memory management

[2] Tong Zhou: ”Read-only Analysis For Java Objects”, http://web.eecs.utk.edu/˜tzhou9/2017/java_object_analysis.pdf

[3] Richard Jones, Chris Ryder: ”A Study of Java Object Demographics”, ISMM ’08 Proceedings of the 7th international symposium on Memory management

[4] Stephen M. Blackburn et al. : ”Pretenuring for Java”, OOPSLA ’01 Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications

[5] Zoltan Majo, Thomas R. Gross: ”Matching Memory Access Patterns and Data Placement for NUMA Systems”, CGO ’12 Proceedings of the Tenth International Symposium on Code Generation and Optimization

[6] Paul R. Wilson: ”Uniprocessor Garbage Collection Techniques”, IWMM ’92 Proceedings of the International Workshop on Memory Management

[7] Kshitij Sudan et al. :”Increasing dram efficiency with locality-aware data placement”, Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems

[8] The DaCapo Benchmark Suite: http://www.dacapobench.org/