An Essay on Garbage Collection

Advanced Topics in Programming Languages

Garbage collectors (GCs) are often designed with the goal of minimizing latency which manifests in the pursuit of short pause times. I shall argue that instead we should refocus our efforts on reducing the overheads provoked by the concurrency and defragmentation needed for low-latency applications.

Bacon, Cheng and Rajan in *A Real-time Garbage Collector with Low Overhead and Consistent Utilization,* demonstrate that emphasizing short pause times will lead to poorer overall mutator performance. However, the limitations imposed by real-time garbage collection result in a collector with concurrency and defragmentation overheads. By aiming to eliminate these, Zhao, Blackburn and McKinley in *Low-Latency, High- Throughput Garbage Collection* achieve an optimal GC with high throughput and low latency.

In the pursuit of short pause times

Low-latency GCs are desired for multiple purposes: real-time applications require a hard upper-bound on the length of time that an application can be paused collecting garbage. Applications with latency-sensitive workloads must respond to the client before any timeouts. In order to meet these latency targets, GCs have seeked to minimize pause times.

In their paper, Bacon, et al. show analytically and empirically that the scheduling of a real-time GC (i.e. when the collector should preempt the mutator) should be done on a time basis. This means that rather than scheduling the collector after a certain amount of memory has been allocated by the mutator, we should schedule it after a set amount of time has passed. This allows us to establish a minimum mutator utilization rate that can guarantee our application is always progressing.

We can liken a scheduler that tries to minimize pause times, to a work-based scheduler since short pause times are achieved by limiting the amount of memory that the collector needs to process. Bacon, et al.'s analysis shows that the efficacy of work-based scheduling is highly dependent on the mutator's allocation patterns. So, by ensuring that individual pause times are kept low, high-allocation rates will plummet mutator utilization as the collector is constantly scheduled to keep up with demand. The application consequently gets stuck and the actual latency of any requests being processed increases substantially.

Thus, Bacon, et al. show how focusing solely on low pause times can lead to pessimal performance. However, for time-based scheduling to be effective for low-latency applications, the corresponding GC must be efficient enough to guarantee a minimum mutator utilization. We must therefore design a GC that minimizes the overheads resulting from pursuing low latencies.

Tripping over low latency

The main overheads for low-latency GCs stem from the need for concurrency and defragmentation.

In order to reduce the time spent collecting garbage, we spread the collector's work over small incremental spans of time. This complicates garbage collection as we now need to ensure that the collector and mutators are synchronized properly to maintain a consistent view of the heap. We can think of garbage collection as a wavefront of reachable marked objects that expands towards all the reachable objects. A mutator running alongside such a GC could potentially move an unmarked object's reference behind the wavefront. Since the collector does not rescan objects behind the wavefront, this unmarked object is believed to be unreachable, and will subsequently be incorrectly collected. Therefore, read and write barriers must be introduced to ensure consistency whenever the heap is modified. Naturally, these impose performance drawbacks.

Most applications make use of objects of various sizes, and as these are freed by the GC they result in fragmentation. Thus, we must employ some form of compaction to defragment the heap which inevitably requires copying – a slow process. This is naively done through a copying collector, but these require at least twice as much virtual memory as other collectors, and often more. A further inefficiency emerges as copying collectors must process all the bytes of each object rather than just traversing references. Copying itself is also significantly slower than marking an object or incrementing a reference count, especially when copying large data arrays. Modern GCs, thus aim to limit the amount of copying done.

Catching up to real-time

Bacon, et al. propose a real-time garbage collector for uniprocessor embedded systems in their paper. They design an incremental mark-and-sweep GC (later named 'Metronome') that is non-copying and implements a more efficient read barrier to reduce the costs of defragmentation and incremental collection. These optimizations are necessary to yield an effective real-time GC.

For their defragmentation scheme, Bacon, et al. divide the heap into pages composed of sequential blocks of the same size. An object is then stored in a page whose corresponding block size matches its size most closely. Thus, by locality of size, objects allocated close in time are located next to each other, minimizing fragmentation by removing any gaps between blocks.

Nevertheless, fragmentation can still occur as an application deallocates objects, leaving behind half-empty pages. If these empty slots do not match the block size required by the next object we want to allocate, then we may need to compact our data to allocate new pages. Metronome chooses the least occupied pages to compact in order to minimize the amount of data that needs to be copied, thus tackling defragmentation without suffering the costs that a full copying collector would have otherwise.

Similarly to how concurrent GCs require synchronization, as Metronome interleaves collection with the mutator, we need to ensure that when an object has been copied, we always read or write from the new copy (i.e. the to-space invariant). Otherwise, the mutator's and collector's view of the heap will be inconsistent. To maintain this invariant, Metronome uses eager read barriers to forward any reads done on the stale copy to the new copy. This necessitates all references pointing to an object be updated when it is moved, a rather inefficient process. Nevertheless, Bacon, et al. show that compiler optimizations can reduce the mean cost of a read barrier to a 4% overhead versus a system without barriers.

These optimizations result in a time-based garbage collector that manages to achieve a minimum mutator utilization of 45% with a heap 1.6-2.5 times the actual maximum memory used by the application. This result would not be acceptable nowadays, especially outside the restrictions of real-time applications. A time-based scheduling approach is only useful in the context of garbage collection on a uniprocessor, where concurrent collection is unfeasible.

Moreover, Bacon, et al. make the assumption that to be functional for low-latency applications, we cannot have stop-the-world collection as the pauses generated would be too long. However, Zhao, et al. show that these pauses can be minimized by optimizing a reference counting GC.

Stopping to go faster

Guaranteeing a minimum mutator utilization rate makes sense when you have real-time guarantees to meet, but will necessitate an incremental/concurrent approach. However, concurrent collectors have the performance overheads discussed in the second section, and so we may be able to achieve better average latency and throughput through other means.

Zhao, et al. introduce LXR, a stop-the-world garbage collector that uses reference counting (RC). LXR runs with lower latency than concurrent GCs that focus on short pause times (C4, Shenandoah, and ZGC), and with higher throughput than G1, the OpenJDK default optimized for speed. As we will see, this is done by tackling defragmentation and concurrency overheads.

LXR uses coalescing reference counting with stop-the-world pauses. This scheme recognizes that between pauses the only changes to a pointer's reference count that matter are the first and the last. We can therefore store the overwritten starting reference in a decrement buffer, and increment the last referent of the modified pointer at the start of the stop-the-world pause. By coalescing the decrements and increments inside the stop-the-world pauses, we can parallelize this work to yield much faster garbage identification.

However, like concurrent tracing collectors, RC requires traversing long chains of pointers, although it only needs to do this at the point of object collection, rather than for every collection cycle. Performing this operation within the stop-the-world pauses would result in unacceptable worst-case latency. Instead, LXR uses lazy-decrements to free objects whose reference count has reached zero concurrently with the mutator. Thus, we reduce the necessary pause times without introducing concurrent tracing's overheads.

By having stop-the-world pauses we can also eliminate all read barriers. This we identified was a major source of overheads for concurrent garbage collection, and plagues systems like Metronome. LXR uses an inexpensive write barrier that keeps track of which pointers are overwritten to later perform the corresponding RC increments and decrements during the stop-the-world pauses. This barrier is around five times cheaper than the alternatives' read barriers, further reducing the concurrency overheads that we identified earlier.

Finally, Zhao, et al. design LXR with the observation that concurrent copying is intrinsically expensive. By seeking low pause times, modern GCs have relied on concurrent evacuation, which necessitates the expensive read barrier. LXR uses the young-generational hypothesis, which assumes that all new objects will become dead by the time the first stop-the-world pause arrives. By allocating objects sequentially, we need only trace through this newly allocated region of memory. Thus, rather than compacting at the scale of the entire heap, LXR gets rid of most fragmentation solely through compacting young objects, minimizing the negative effects of copying.

In conclusion, by focusing on reducing the overheads imposed by defragmentation, and using concurrency minimally, Zhao, et al. design a garbage collector that outperforms its contemporaries across a diverse set of benchmarks. LXR delivers great latency under heavy

memory allocations. Although this comes at the cost of GC pause times, Bacon, et al. demonstrate that these result in unacceptable mutator performance under high memory allocation rates. Thus, the papers discussed empirically show that we should instead focus on reducing the performance overheads imposed by concurrency and defragmentation, to produce more effective low-latency garbage collectors.

Word Count: 1630