Concurrent Reference Counting

Or Ostrovsky

13/1/15

This summery will contain the major points of the lecture given at the 13/1/15 about the different techniques used to apply reference counting concurrently.

Simple Reference Counting

The most simple way to do concurrent reference counting to avoid concurrency completely. The way to achieve this is to use write and read barriers that *locks the object* so only one thread at a time may access it. That allows the reference count to be updated correctly. The major downside of this approach is that it hinders performance considerably, since all the thread may wait to access a single object, thus nullifying any advantage concurrency might present.

A slight improvement is to apply *atomic memory primitives* such as `CompareAndSet` to create a spin-lock that allows individual access to each of the object's fields. This provides a small increase in performance, since now the bottleneck is a field rather than an object. While a write barrier may be easily implemented that way, a problem arises when trying to implement the read barrier, since it requires *multiple atomic memory primitives*, which hard to come by.

Buffered Reference Counting

Another different approach is to buffer each modification locally (a buffer for each thread) and collect those modifications *on the fly* - stop one thread at a time and collect it's buffers. Afterward, applying each of the modifications on a single dedicated thread is relatively easy. An improvement for this method is to apply *deferred reference counting* and don't log operations involving roots and collect the state of each threads roots, when that thread is paused.

The great advantage of this method is that it provides barriers that create minimal pause time, and that the collection phase ensures that at any given time, at least one mutator thread is running. The problem with that method (and other methods) is that it ignores the subject of cycles completely.

Sliding View

This section will discuss the not a complete algorithm, but a concept that may be used in one.

First, an observation: most of the changes made to a reference are irrelevant. It it enough to sample each one periodically and update the reference count according the difference. This is called *coalesced refrence counting*. Using the logs gathered this way, requires to stop all of the mutators.

An improvement is to have each thread log the initial state of object changed by that thread at the beginning of each collection cycle. Those logs are to be collected on the fly and used to create a *sliding view*. At each cycle both the current sliding view and the next one (which is built as the collection progresses) are used to change the reference count of each object. This method creates a "spread" snapshot of the heap, which may be used by single threaded algorithm while the heap itself is changing.

An Efficient On-the-Fly Cycle Collection

An interesting way to use sliding views it to use a *generational garbage collector*, that uses *Mark & Sweep* on the young generation and reference counting on the old one, with *The Recycler* to handle cycles. Both of the collectors will use the sliding view gathered, because both the simple version of Mark & Sweep and The Recycler need a "fixed" heap to operate. This setup allows the M&S part to trace few live objects (due to the weak generational hypothesis), also it strives to have the Recycler trace as little objects as possible.