LMAX Disruptor

Jamie Allen
jallen@chariotsolutions.com
@jamie_allen

August 10, 2011

There is nothing new here

- Why is the Disruptor pattern relevant?
- The "virtual" nature of our runtime and deployment environment has desensitized us as developers to the impact of our decisions
- It flies in the face of so many concurrency abstractions to show what can be accomplished when implementing code in the most optimized fashion

How Did They Arrive at the Disruptor

- Initiative started several years ago at Betfair, with the Flywheel and 100x projects, trying to glean more performance from their system
- Looked into J2EE, SEDA, Actors, etc - couldn't get the throughput they desired
- Neither project made it to production due to legacy integration issues
- Betfair spun off Tradefair into LMAX, Martin Thompson leaves Matt Youill, works with Mike Barker and the rest of the team on their clean slate Disruptor implementation
- So named because it has elements for dealing with graphs of dependencies to the Java7 Phaser concurrency type, introduced in support of ForkJoin
- Why the JVM? Why wasn't this done in C++ or another native implementation? Ola Bini talked to Martin Fowler, and they believe that the benefits of using the Java platform outweighed the potential for marginal throughput gains in C++. Note that Thompson is currently looking to port the code to C++.

Mechanical Sympathy

- Martin Thompson, the lead architect at LMAX, believes in the concept of Mechanical Sympathy, a term coined by former F1 champion Jackie Stewart, who believed that all great drivers had to understand at some level how their machine worked to derive the fastest time driving it
- "The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry." - Henry Peteroski, as quoted on Martin's blog

Keys to the Disruptor's Performance

- Never release the core to the kernel (thus maintaining your L3 cache)
- Avoid lock arbitration
- Minimize usage of memory barriers
- Reuse pre-allocated sequential memory to avoid GC and compaction and enable cache pre-fetching

Thread Management

- In a low latency, high throughput system with balanced flow, you can assign a thread to a core from the JVM by never releasing control of the thread (wait, ForkJoin)

Avoiding Locks

- Locks, extremely non-performant due to context switching in the kernel which suspend threads waiting on a lock until it is released
- Note that during a kernel context switch, the OS may decide to perform other tasks not related to your process, thus losing valuable cycles
- CAS semantics are much better, since no kernel context switch is required for lock arbitration, but the processor must still lock its instruction pipeline to ensure atomicity and introduce a memory barrier to ensure that changes are made visible to all threads
- Memory barriers are used by processors to indicate sections of code where the ordering of memory updates matters - all memory changes appear in order at the point required (note: compilers can add software MBs in addtion to the processor's, which is how Java's volatile keyword works)

Minimal Use of Memory Barriers

- Processors only need to guarantee that the execution of instructions gives the same result, regardless of order, and thus perform instructions out of order frequently to enhance performance
- Memory barriers (volatile) specify where no optimizations can be performed to ensure that ordering is correct at runtime

What's Wrong With Queues

- Unbounded queues use linked lists, which are not contiguous and therefore do not support striding (prefetching cache lines)
- Bounded queues use arrays, where the head (dequeuing) and tail (enqueuing) are the primary points of write contention, but also have a tendency to share a cache line
- In Java, queues are also a significant source of garbage - the memory for data must be allocated, and if unbounded, the linked list node must also be allocated; when dereferenced, all of these objects must be reclaimed

Memory Allocation

- What if you designed your system to utilize on-board caches for a core as effectively as possible?
- Allocate a large ring buffer of byte arrays on startup, and copy data into those arrays as received/handled
- Sequencing of data in main memory is also important - as the core understands your usage of data from memory, it can pre-load the cache with data it knows you need next. You need to understand how you are allocating data and what that means (see padding)
- The sequential nature of the ring buffer allows you to introduce dependencies between processes that need something to happen before they move on (how you compose Consumers by ConsumerBarrier)
- Maximizing the use of "cache lines" (64 bytes, 8 longs of 8 bytes each, avoiding misses and "false sharing" - which aids with...

Caching

- Processors are much faster than memory now, and to optimize their performance, they use varying levels of caches (registers, store buffers, L1, L2, L3 and main memory) to support the execution of instructions, and they are kept coherent via message passing protocols
- One of the most expensive operations for a process is a cache miss - when data is looked for in one of the caches and not found, so it must go to the next level
- Data is not moved in bytes or words, but in cache lines (32-256 bytes depending on the processor, usually 64)
- If two variables are on the same cache line and written to by different threads, they present the same problems of write contention as if they were one variable ("false sharing")
- For performance, you must ensure that independent but concurrently written data are on separate cache lines

Caching

- When data is accessed from main memory in a predictable fashion (such as walking the data in a predictable "stride"), the processor can optimize by pre-fetching data it expects will be needed shortly. This ring buffer has a predicatable pattern of access
- Note that data structures such as linked lists and trees tend to have nodes that are more widely distributed (non-contiguous) in memory and therefore no predictable strides for performance optimization, which forces the processor to perform main memory direct access more often at the time the data is needed at significant performance cost

Garbage Collection

- The reuse of array buffers guarantees no GC of mature objects, thus no global GC compaction to "stop the world" - limit GC to short- and long-lived objects, not those in the middle
- Latency occurs as those objects are copied between generations, which is minimized by reusing objects so that they only traverse the generations once *GC: Restart the machine every day to clear the heap, guarantee no compaction (can go 3-4 days without worrying about it, but this is a safety measure)
- How much time do we all waste trying to find the optimal GC strategy for a system?
- What if you were able to design your system so that you didn't have to think about it, because YOU CONTROL how much GC and compaction is taking place?

Sequential Data

- The "ring" characteristics of the array buffer promote resequencing - you never stop traversing the queue (sequence number increases, use MOD by ring size to get slot)

Implementation: Ring Buffer

- The Ring Buffer is a bounded, pre-allocated data structure, and the allocated data elements will exist for the life of the Disruptor instance
- Per Daniel Spiewak, was considered for a core data structure in Clojure before the bit-mapped vector trie was selected by Rich Hickey
- On most processors, there is a high cost for a remainder calculation on a sequence number which determines the slot in the ring, but it can be greatly reduced by making the ring size a power of 2; use a bit mask of ring size minus one to perform the remainder operation efficiently as compared to sequence number % size (600x faster?)
- The data elements are merely storage for the data to be handled, not the data itself
- Since the data is allocated all at once on startup, it is highly likely that the memory will be contiguous in main memory and will support effective striding for the caches
- When an entry in the ring buffer is claimed by a producer, it copies data into one of the pre-allocated elements

Implementation: Producers

- In most Disruptor usages, there is only one producer (network IO, file system reads, etc), which means no contention on sequence entry/allocation; if more than one producer, they can race each other for slots and use CAS on the sequence number for next available slot to use
- This two-phase operation - getting the sequence number, copying the data and then explicitly committing, separates the action of putting the data into a slot and making it visible
- It also helps to maintain two-phase semantics if more than one Disruptor is accessed by a producer
- Producers can check with Consumers to see where they are so they don't overwrite ring buffer slots still in use
- Producers copy data into the claimed element and make it public to consumers by "committing" the sequence to their ProducerBarrier

Implementation: Consumers

- Consumers wait for a sequence to become available in the ring buffer before they read the entry using a WaitStrategy defined in the ConsumerBarrier; note that various strategies exist for waiting, and the choice depends on the priority of CPU resource versus latency and throughput
- If CPU resource is more important, the consumer can wait on a condition variable protected by a lock that is signalled by a producer, which as mentioned before comes with a contention performance penalty
- Consumers loop, checking the cursor representing the current available sequence in the ring buffer, which can be done with or without thread yield by trading CPU resource against latency - no lock or CAS to slow it down
- Consumers that represent the same dependency share a ConsumerBarrier instance, but only one consumer per CB can have write access to any field in the entry

Sequencing

- Basic counter for single producer, atomic int/long for multiple producers (using CAS to protect the counter)
- When a producer finishes copying data to a ring buffer element, it "commits" the transaction by updating a separate counter used by consumers to find out the next available data to use
- Consumers merely provide a BatchHandler implementation that receives callbacks when data is available for consumption
- Consumers can be constructed into a graph of dependencies representing multiple stages in a processing pipeline
- Read/Writes are minimized due to the performance cost of the volatile memory barrier

Batching Effect

- When a consumer falls behind due to latency, it has the ability to process all ring buffer elements up to the last committed by the producer, a capability not found in queues
- Lagging consumers can therefore "catch up", increasing throughput and reducing/smoothing latency; near constant time for latency regardless of load, until memory subsystem is saturated, at which point the profile is linear following Little's Law
- Producers also batch, and can write to the point in the ring buffer where the slowest consumer is currently working
- Producers also have to manage a wait strategy when there are multiples of them; no "commits" to the ring buffer occur until the current sequence number is the one before the claimed slot
- Compared to "J" curve effect on latency observed with queues as load increases

Dependency Graphs

- With a graph like model of producers and consumers (such as actors), queues are required to manage interactions between each of the elements
- The single ring buffer replaces this in a single data structure for all of the elements, resulting in greatly reduced fixed costs of execution, increasing throughput and reducing latency
- Care must be taken to ensure that state written by independent consumers doesn't result in the false sharing of cache lines

Event Sourcing

- Daily snapshot and restart to clear all memory
- Replay events from a snapshot to see what happened when something goes awry

Use cases for Disruptor

- Note that the key is BALANCED FLOW - if your flow is unbalanced, you need to weigh the cost of losing local L3 cache with the reuse of cores

Links

- Blog: Processing 1M TPS with Axon Framework and the Disruptor: http://blog.jteam.nl/2011/07/20/processing-1m-tps-with-axon-framework-and-the-disruptor/
- QCon presentation: http://www.infoq.com/presentations/LMAX
- Google Group: http://groups.google.com/group/lmax-disruptor
- Martin Fowler's Bliki post: http://martinfowler.com/articles/lmax.html
- Martin Thompson's Mechanical Sympathy blog: http://mechanical-sympathy.blogspot.com/
- Trisha Gee's Mechanitis Blog: http://mechanitis.blogspot.com/
- Disruptor Wizard (simplifying dependency wiring): http://github.com/ajsutton/disruptorWizard
- My Scala port: http://github.com/jamie-allen/sdisruptor

Presenting at JavaOne 2011