**Disruptor:**

High performance alternative to bounded queues for exchanging data between concurrent threads

Martin Thompson (CTO)

Dave Farley (Head of Software Development)

Michael Barker (Lead Developer)

29-May-2011

<http://code.google.com/p/disruptor/>

## Abstract

The Disruptor is a new concurrent programming framework for exchanging data between concurrent execution contexts in a shared memory system. It is an alternative mechanism to bounded queues for exchanging data between threads. The implementation teases apart the concerns for data storage, producer synchronisation, and consumer synchronisation that are normally conflated with queue based implementations. By avoiding this conflation, the concerns can be managed independently resulting in greater flexibility of design and establishing a new performance standard. The resulting implementation manifests this performance as greater throughput with lower and more predictable latency. The Disruptor is available as an open-source Java implementation on Google Code.

## Overview

The Disruptor is the result of efforts at LMAX building the World’s highest performance financial exchange. Early designs focused on architectures derived from SEDA[[1]](#endnote-1) and Actors[[2]](#endnote-2) using pipeline designs to for throughput. After profiling various implementations it became evident that the queuing of events between stages in the pipeline where dominating the costs. Queues also introduced latency and high levels of jitter. Significant effort was expended on developing new queue implementations with better performance. However it became evident that queues as a fundamental data structure are limited due to the conflation of design concerns for the producers, consumers and data storage. The Disruptor is the emergent design from teasing these concerns apart combined with the teams desire to introduce parallelism where possible.

## Write Contention

In a concurrent environment the fundamental issue to avoid is contended write access to any resource. The resource could be a database, file, socket or even a location in memory. To have multiple threads write to the same resource requires complex and expensive coordination. Typically this would be achieved by employing a locking strategy. Locks provide mutual exclusion and ensure the visibility of changes occur in an ordered manner. Locks are incredibly expensive because they require arbitration when contended. This arbitration is achieved by a context switch to the operating system kernel which will suspend threads waiting on a lock until it is released. Fast user mode locks can be employed when not contended but these are ultimately just an optimisation when not under contention, this is usually not the case in high throughput pipelined system.

To illustrate the cost of locks we will demonstrate with a simple experiment. The experiment is to call a function which increments a 64-bit counter in a loop 500 million times. This can be executed by a single thread on a 2.4Ghz Intel Westmere EP in just 300ms if written in Java. The language is unimportant to this experiment and will be similar across languages with the same basic primitives.

Once a lock is introduced to provide mutual exclusion the cost goes up significantly and even more so when two or more threads contend as the table below shows.

|  |  |
| --- | --- |
| **Method** | **Time (ms)** |
| Single thread | 300 |
| Single thread with lock | 10,000 |
| Two threads with lock | 224,000 |
| Single thread with CAS | 5,700 |
| Two threads with CAS | 30,000 |

An alternative to locks can be employed for updating single words know as atomic or interlocked instructions. These are commonly known as CAS (Compare And Swap) operations, e.g. “*lock cmpxchg*” on x86. A CAS operation is a special machine code instruction that allows a word in memory to be conditionally set as an atomic operation. For the “increment a counter experiment” each thread can spin in a loop reading the counter then try to atomically set it with an incremented value. If the value has changed the CAS operation will fail and the thread witll try to repeat the increment until is succeeds. The CAS approach is more efficient than locks because no context switch to the kernel is required for arbitration. However CAS operations are not free of cost. The processor must lock its instruction pipeline to ensure atomicity and employ a memory barrier to make the changes visible to other threads. CAS operations are available via *java.util.concurrent.Atomic\** classes from Java.

If the critical section of the program is more complex than a simple increment of a counter it may take a complex state machine using multiple CAS operations. Developing concurrent programs using locks is difficult; developing lock-free algorithms using CAS operations and memory barriers is many times more complex and very difficult to prove correct.

The ideal algorithm would be one with only a single thread owning all writes to a single resource with other threads reading the results. To read the results in a multi-processor environment requires memory barriers to make the changes visible to threads running on other processors.

## Memory Barriers

Modern processors perform out-of-order execution of instructions and memory loads/stores for performance reasons. The processors need only guarantee that program logic produces the same results regardless of execution order. While this is not an issue for single threaded programs it is important that when threads share state that all memory changes appear in order at the point where they need to exchange data. Memory barriers are the means by which hardware ordering and visibility of change is achieved between threads. These barriers are in addition to the software barriers compilers put in place to ensure ordering of compiled code.

Modern CPUs are now much faster than modern memory systems. To bridge this divide CPUs use complex cache systems which are effectively fast hardware hash tables without chaining. These caches are kept coherent with other processor cache systems via message passing protocols. In addition processors have store buffers to offload writes to these caches and invalidate queues so the cache coherency protocols work efficiently.

What this means for data is the latest version of any value could at any stage after being written be in a register, store buffer, one of many layers of cache, or in main memory. If threads are to exchange this value it needs to be made visible in an ordered fashion and this is achieved via memory barriers.

A read memory barrier orders load instructions on the CPU that executes it by effectively marking the invalidate queue for changes coming into its cache. This gives it a consistent view of the world for write operations ordered before the read barrier.

A write barrier orders store instructions on the CPU that executes it by effectively marking the store buffer thus flushing writes out via its cache. This barrier gives an ordered view to the world of what store operations happen before the write barrier.

A full memory barrier orders both loads and stores but only on the CPU that executes it.

Some CPUs have more variants in addition to these three primitives but they are sufficient to understand the complexities of what is involved. In the Java memory model a read and write of a *volatile* field implements the read and write barriers respectively. This was made explicit in the Java memory model as defined in Java 5.

## Cache Lines

The unit of currency when moving memory is not bytes or words. For efficiency caches are organised into cache lines that are typically 32-256 bytes in size with the normal being 64 bytes. These are the units passed around by the cache coherency protocols. What this means for data contention is that if two variables are in the same cache line and they are written by different threads then you effectively have write contention on each variable by nature of being in the same cache line. It is therefore important to ensue independent but concurrently written variables do not sure the same cache line.

When accessing memory in a predictable manner CPUs hide the main memory latency by pre-fetching in the background if they can see a pattern of access such a walking memory with a predictable “stride”. When walking an array the stride is predictable and thus memory will be pre-fetched in cache lines. Linked lists and trees tend to have nodes randomly allocated in memory with no predictable stride of access.

## Memory Allocation

All the memory for the ring buffer is pre-allocated on start up. This can be as either an array of pointers to entries containing the fields for re-use or as an array of structures representing the entries. In Java it is only possible at present to have an array of pointers to objects. If the memory is allocated at the same time it is highly likely to be laid out contagiously in main memory to support cache striding. John Rose has a proposal for “value types”[[3]](#endnote-3) which would allow arrays of tuples like other languages such as C.

Garbage collection can be the enemy when developing a low-latency system in a managed runtime environment such as Java. The more memory that is allocated the more burdens this puts on the garbage collector. Garbage collectors work at their best when memory is very short lived or effectively immortal. The pre-allocation of memory in the ring buffer falls in the immortal category thus removing the burden. Under heavy load a queue based system can back up with the allocated objects in the queues surviving beyond the young generation. This has two implications. Firstly the objects have to copied between generation which happens on a collection and thus not an amortised cost which causes latency jitter. Secondly these objects have to be collected from the old generation which is typically more expensive and increases the likelihood of “stop the world” pauses that can take many seconds on a large heap when compaction has to occur.

## Queues

Queues are typically implemented as either linked-lists or array backed. If an in-memory queue is allowed to be unbounded then for many classes of problem it can grow unchecked until catastrophic failure if the producers outpace the consumers. Unbounded queues can be useful in systems where the consumers are guaranteed to outpace the producers and memory is a precious resource. To keep a queue bounded it needs to be array backed or the size must be tracked.

Queue implementations by design tend to have write contention on the head, tail, and size variables. It should also be noted that in a running system queues are typically always close to full or empty due to the differences in pace between consumers and producers. This can result in compounding the contention when the head and tail mechanisms are separated using different concurrent objects such as locks or CAS’ed variables. This can be because the head and tail of an array backed queue share the same cache line for writes, or head and tail pointers plus chaining links pointing to the same node in a linked-list queue.

The concerns of managing producers claiming the head of a queue, consumers claiming the tail, and the storage of nodes in between make the designs of concurrent implementation very complex to manage beyond using a single large-grain lock on the queue. A large grain lock on the whole queue for *puts* and *takes* is simple to implement but ultimately the bottleneck to throughput. If the concurrent concerns are teased apart within the semantics of a queue implementation the implementations become very complex for anything other than a single producer – single consumer implementation.

In Java queues are also significant sources of garbage due to allocating objects to place in the queue, and additionally if linked-list backed then the nodes themselves become garbage.

Most array backed implementations resort to locks which introduce jitter to the latency or use link-list backed implementations that have a state machine to track the CAS operations for patching in a node.

## Pipelines and Graphs

For many classes of problem a number of processing stages have to be wired together in pipelines that can also have parallel paths to give graph like topologies. The links between each stage can be implemented by queues when each stage has its own thread.

For the links between each stage we have to incur the cost of en-queuing and de-queuing the units of work. This cost is multiplied by the number of targets when the path must fork.

It would be ideal if the graph of dependencies could be expressed without incurring the cost of putting the queues between stages. Trying to address this issue provide the major inspiration for the Disruptor.

## Birth of the Disruptor

While trying to address the problem domain described above a design emerged from teasing apart the conflated concerns in queues, combined with pushing towards the principle that any data should owned by only one thread for write access. That design came to be known as the “Disruptor”. It was named so because it had elements of similarity for dealing with graphs of dependencies like “Phasers”[[4]](#endnote-4) in Java 7 introduced to support Fork-Join compounded by the Star Trek weapons link, and because we wanted to disrupt current thinking about concurrent programming.

## Teasing Apart the Concerns

It became evident that a queue is a conflation of 3 concerns when exchanging a sequence of items:

1. Storage of items being exchanged
2. Coordination of producers claiming the next sequence for exchange
3. Coordination of consumers being notified that a new item is available for exchange

Queued items are normally stored in array slots or as nodes in a linked list. When designing a financial exchange too much memory allocation is the enemy because of the resulting garbage collection therefore linked-lists are not the best option. It is best if the entire storage for the exchange of data can be pre-allocated in a uniform chunk within the data structure employed. This means no further memory allocation and it is laid out in memory enabling cache line pre-fetching. A structure that fits this requirement is a ring buffer implemented as an array with all the slots pre-filled. When an entry is claimed the data can be copied into the pre-allocated structure by the producer.

Producers race with each other to claim the next entry in sequence. With many data exchange usecases there is only one producer so sequence allocation is not contended. Typical producers are a file reader or network listener. When many producers exist the next sequence can be claimed with a simple CAS operation. Once the producer has copied the relevant data to the claimed entry they can make it public to the consumers by committing the sequence. This can be done without CAS by a simple busy spin until the other producers can have reached this sequence in their own commit so this producer then advances the cursor signifying the available entry in the ring buffer. Producers can avoid wrapping the ring by tracking the sequence of consumers as a simple read operation before they write to the ring buffer.

Consumers wait for a sequence to become available in the ring buffer before they read the entry. Various strategies can be employed while waiting. If CPU resource is precious they can wait on a condition variable within a lock that gets signalled by the producers. This obviously is a point of contention and only to be used when CPU resource is more important than latency or throughput. The consumers can also loop checking the cursor which represents the currently available sequence in the ring buffer. This could be done with or without a thread yield by trading CPU resource against latency. This scales very well as we have broken the contended dependency between the producers and consumers if we do not use a lock and condition variable.

## Sequencing

Sequencing is the core concept to how the concurrency is managed in the Disruptor. Each producer and consumer works of a strict sequencing concept for how it interacts with the ring buffer. Producers claim the next slot in sequence when putting an entry in the ring. This sequence of the next available slot can be a simple counter in the case of only producer or an atomic counter updated using CAS operations in the case of multiple producers. Once a sequence value is claimed this entry in the ring buffer is now available to be written to by the producers. When the producer has finished updating the entry is can commit the changes by updating a counter which represents the cursor on the ring buffer for what is the latest available entry to be consumed. This is a different counter for the next claim sequence used by the producers. The ring buffer cursor can be read and written in a bust spin by the producers using memory barrier without requiring a CAS operation.

Consumers wait for a given sequence to become available by using a memory barrier to read the cursor. Once the cursor has been updated the memory barriers ensure the changes to the entries in the ring buffer are visible to the consumers who have waited on the cursor advancing.

Consumers each contain their own sequence which they update as they process entries from the ring buffer. These consumer sequences allow the producers to track consumers to prevent the ring wrapping. These consumer sequences also allow consumers to depend on other consumers completing their work on a given entry before proceeding to do work on the entry themselves.

In the case on having only one producer, and regardless of the complexity of the consumer graph, no locks or CAS operations are required. The whole concurrency coordination can be achieved with just memory barriers on the discussed sequences.

## Batching Effect

When consumers are waiting on an advancing cursor sequence in the ring buffer an interesting opportunity arises that is not possible with queues. If the consumer finds the ring buffer cursor has advanced a number of steps since it last checked it can process up to that sequence without getting involved in the concurrency mechanisms. This results in the consumers quickly regaining pace with the producers when the producers burst ahead thus balancing the system. This type of batching increases throughput while reducing and smoothing latency at the same time.

## Dependency Graphs

A queue represents the simple one stage pipeline dependency between producers and consumers. If the consumers form a chain or graph like structure of dependencies then queues are required between the dependent stages of the graph. This incurs the fixed costs of queues been applied many times within the graph of dependent stages. When designing the LMAX financial exchange our profiling showed how when taking a queue based approach resulted in queuing costs dominating the execution costs for processing an order representing a financial transaction.

Because the producer and consumer concerns are separated with the Disruptor pattern it is possible to have a complex graph of dependencies between consumers represented yet only using a single ring buffer at the core. This results in greatly reduced fixed costs of execution thus increasing throughput while reducing latency.

If a single ring buffer is used to sort entries in a complicated manner the entry can have a complex structure representing the whole workflow in a single cohesive place. Care much be taken in the design of such a structure so that the same state is not worked on by multiple consumers or write contention will occur. This is not as costly as concurrent write contention but still has an impact on throughput.

## Disruptor Class Diagram

The core relationships in the Disruptor framework are depicted in the class diagram below. This diagram leaves out the convenience classes which can be used to simplify the programming model. After the dependency graph is constructed the programming model is simple. Producers claim entries in sequence via a *ProducerBarrier*, write their changes into the claimed entry, then commit that entry back via the *ProducerBarrier* making them available for consumption. As a consumer all one needs do is provide a *BatchHandler* implementation that receives call backs when a new entry is available. This resulting programming model is event based having a lot of similarities to the Actor Model.

The *BatchHandler* call back implementation encapsulates the batching effect described earlier. This is very effective when dealing with IO devices such as network and storage where changes can be batched and then flushed to the device at the end of a batch.



## Throughput Performance Testing

As a reference we choose Doug Lea’s excellent *java.util.concurrent.ArrayBlockingQueue*[[5]](#endnote-5) which has the highest performance of any bounded queue based on our testing. The tests are conducted in a blocking programming style to match that of the Disruptor. The tests cases detailed below are available in the Disruptor open source project. Note: running the tests requires a system capable of executing at least 4 threads in parallel.

**Unicast: 1P – 1C**

**Three Step Pipeline: 1P – 3C**

**Sequencer: 3P – 1C**

**Multicast: 1P – 3C**

**Diamond: 1P – 3C**

For the above configurations an *ArrayBlockingQueue* was applied for each arc of data flow compared to barrier configuration with the Disruptor. The following table shows the performance results in operations per second using a Java 1.6.0\_25 64-bit Sun JVM, Windows 7, Intel Core i7 860 @ 2.8 GHz without HT and Intel Core i7-2720QM, Ubuntu 11.04, and taking the best of 3 runs when processing 500 million messages. Results can vary substantially across different JVM executions and the figures below are not the highest we have observed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Nehalem 2.8Ghz – Windows 7 SP1 64-bit** | | **Sandy Bridge 2.2Ghz – Linux 2.6.38 64-bit** | |
|  | **ABQ** | **Disruptor** | **ABQ** | **Disruptor** |
| Unicast: 1P – 1C | 5,339,256 | 25,998,336 | 4,057,453 | 22,381,378 |
| Pipeline: 1P – 3C | 2,128,918 | 16,806,157 | 2,006,903 | 15,857,913 |
| Sequencer: 3P – 1C | 5,539,531 | 13,403,268 | 2,056,118 | 14,540,519 |
| Multicast: 1P – 3C | 1,077,384 | 9,377,871 | 260,733 | 10,860,121 |
| Diamond: 1P – 3C | 2,113,941 | 16,143,613 | 2,082,725 | 15,295,197 |

## Latency Performance Testing

To measure latency we take the 3 step pipeline and generate events at less than saturation. This is achieved by waiting 1 microsecond after injecting an event before injecting the next and repeating 50 million times. To time at this level of precision it is necessary to use time stamp counters from the CPU. We choose CPUs with an invariant TSC because older processors suffered from changing frequency due to power saving and sleep states. Intel Nehalem and later processors use an invariant TSC which can be accessed by he latest JVMs running on Ubuntu 11.04. No CPU binding has been employed for this test.

For comparison we use the *ArrayBlockingQueue* once again. We could have used *ConcurrentLinkedQueue*[[6]](#endnote-6) which is likely to give better results but we want to use a bound queue implementation to ensure producers do not outpace consumers by creating back pressure. The results below are for 2.2Ghz Core i7-2720QM running Java 1.6.0\_25 64-bit on Ubuntu 11.04.

Average latency for the Disruptor comes out at 51 nanoseconds compared to 32,757 nanoseconds for *ArrayBlockingQueue*. Profiling shows the use of locks and signalling via a condition variable are the main cause of latency for the *ArrayBlockingQueue*.

|  |  |  |
| --- | --- | --- |
|  | **Array Blocking Queue (ns)** | **Disruptor (ns)** |
| Average Latency | 32,757 | 51 |
| 99% Max Latency | 314,510 | 128 |
| 99.99% Max Latency | 2,097,152 | 2,228 |

## Conclusion

The Disruptor is a major step forward for both increasing throughput and reducing latency in a concurrent environment. By separating the concerns normally conflated into queue implementations it allows for more flexible configurations when a graph of dependencies exist for processing a source of data via multiple execution stages.

Our testing shows that the batching effect described can introduce a new characteristic in high performance systems which usually suffer greater contention and increased latency under burst conditions is inverted with latency staying low until saturation occurs rather than the usual J curve experienced with queue based approaches.

1. Staged Event Driven Architecture – http://www.eecs.harvard.edu/~mdw/proj/seda/ [↑](#endnote-ref-1)
2. Actor model – http://dspace.mit.edu/handle/1721.1/6952 [↑](#endnote-ref-2)
3. Value Types - http://blogs.oracle.com/jrose/entry/tuples\_in\_the\_vm [↑](#endnote-ref-3)
4. Phasers - http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/Phaser.html [↑](#endnote-ref-4)
5. ArrayBlockingQueue - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ArrayBlockingQueue.html [↑](#endnote-ref-5)
6. ConcurrentLinkedQueue - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html [↑](#endnote-ref-6)