LMAX

Disruptor:

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

Martin Thompson
Dave Farley
Michael Barker
Patricia Gee
Andrew Stewart

May-2011

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

1 Abstract

LMAX was established to create a very high performance financial exchange. As part of our work to accomplish this goal we have evaluated several approaches to the design of such a system, but as we began to measure these we ran into some fundamental limits with conventional approaches.

Many applications depend on queues to exchange data between processing stages. Our performance testing showed that the latency costs, when using queues in this way, were in the same order of magnitude as the cost of IO operations to disk (RAID or SSD based disk system) – dramatically slow. If there are multiple queues in an end-to-end operation, this will add hundreds of microseconds to the overall latency. There is clearly room for optimisation.

Further investigation and a focus on the computer science made us realise that the conflation of concerns inherent in conventional approaches, (e.g. queues and processing nodes) leads to contention in multi-threaded implementations, suggesting that there may be a better approach.

Thinking about how modern CPUs work, something we like to call “mechanical sympathy”, using good design practices with a strong focus on teasing apart the concerns, we came up with a data structure and a pattern of use that we have called the Disruptor.

Testing has shown that the mean latency using the Disruptor for a three-stage pipeline is 3 orders of magnitude lower than an equivalent queue-based approach. In addition, the Disruptor handles approximately 8 times more throughput for the same configuration.

These performance improvements represent a step change in the thinking around concurrent programming. This new pattern is an ideal foundation for any asynchronous event processing architecture where high-throughput and low-latency is required.

At LMAX we have built an order matching engine, real-time risk management, and a highly available in-memory transaction processing system all on this pattern to great success. Each of these systems has set new performance standards that, as far as we can tell, are unsurpassed.

However this is not a specialist solution that is only of relevance in the Finance industry. The Disruptor is a general-purpose mechanism that solves a complex problem in concurrent programming in a way that maximizes performance, and that is simple to implement. Although some of the concepts may seem unusual it has been our experience that systems built to this pattern are significantly simpler to implement than comparable mechanisms.

The Disruptor has significantly less write contention, a lower concurrency overhead and is more cache friendly than comparable approaches, all of which results in greater throughput with less jitter at lower latency. On processors at moderate clock rates we have seen over 25 million messages per second and latencies lower than 50 nanoseconds. This performance is a significant improvement compared to any other implementation that we have seen. This is very close to the theoretical limit of a modern processor to exchange data between cores.

2 Overview

The Disruptor is the result of our efforts to build the world's highest performance financial exchange at LMAX. Early designs focused on architectures derived from SEDA[1] and Actors[2] using pipelines for throughput. After profiling various implementations it became evident that the queuing of events between stages in the pipeline was dominating the costs. We found that queues also introduced latency and high levels of jitter. We expended significant effort 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 their data storage. The Disruptor is the result of our work to build a concurrent structure that cleanly separates these concerns.

3 The Complexities of Concurrency

In the context of this document, and computer science in general, concurrency means not only that two or more tasks happen in parallel, but also that they contend on access to resources. The contended resource may be a database, file, socket or even a location in memory.

Concurrent execution of code is about two things, mutual exclusion and visibility of change. Mutual exclusion is about managing contended updates to some resource. Visibility of change is about controlling when such changes are made visible to other threads. It is possible to avoid the need for mutual exclusion if you can eliminate the need for contended updates. If your algorithm can guarantee that any given resource is modified by only one thread, then mutual exclusion is unnecessary. Read and write operations require that all changes are made visible to other threads. However only contended write operations require the mutual exclusion of the changes.

The most costly operation in any concurrent environment is a contended write access. To have multiple threads write to the same resource requires complex and expensive coordination. Typically this is achieved by employing a locking strategy of some kind.

3.1 The Cost of Locks

Locks provide mutual exclusion and ensure that the visibility of change occurs 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. During such a context switch, as well as releasing control to the operating system which may decide to do other house-keeping tasks while it has control, execution context can lose previously cached data and instructions. This can have a serious performance impact on modern processors. Fast user mode locks can be employed but these are only of any real benefit when not contended.

We will illustrate the cost of locks with a simple demonstration. The focus of this 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 results will be similar across all languages with the same basic primitives.

Once a lock is introduced to provide mutual exclusion, even when the lock is as yet un-contended, the cost goes up significantly. The cost increases again, by orders of magnitude, when two or more threads begin to contend. The results of this simple experiment are shown in the table below:

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
Single thread with volatile write	4,700

Table 1 - Comparative costs of contention

3.2 The Costs of "CAS"

A more efficient alternative to the use of locks can be employed for updating memory when the target of the update is a single word. These alternatives are based upon the atomic, or interlocked, instructions implemented in modern

processors. 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 to its new incremented value. The old and new values are provided as parameters to this instruction. If, when the operation is executed, the value of the counter matches the supplied expected value, the counter is updated with the new value. If, on the other hand, the value is not as expected, the CAS operation will fail. It is then up to the thread attempting to perform the change to retry, re-reading the counter incrementing from that value and so on until the change succeeds. This CAS approach is significantly more efficient than locks because it does not require a context switch to the kernel 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 in Java by using the *java.util.concurrent.Atomic** classes.

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 to orchestrate the contention. Developing concurrent programs using locks is difficult; developing lock-free algorithms using CAS operations and memory barriers is many times more complex and it is very difficult to prove that they are 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.

3.3 Memory Barriers

Modern processors perform out-of-order execution of instructions and out-of-order loads and stores of data between memory and execution units for performance reasons. The processors need only guarantee that program logic produces the same results regardless of execution order. This is not an issue for single-threaded programs. However, when threads share state it is important that all memory changes appear in order, at the point required, for the data exchange to be successful. Memory barriers are used by processors to indicate sections of code where the ordering of memory updates is important. They are the means by which hardware ordering and visibility of change is achieved between threads. Compilers can put in place complimentary software barriers to ensure the ordering of compiled code, such software memory barriers are in addition to the hardware barriers used by the processors themselves.

Modern CPUs are now much faster than the current generation of 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 that the cache coherency protocols can acknowledge invalidation messages quickly for efficiency when a write is about to happen.

What this means for data is that the latest version of any value could, at any stage after being written, be in a register, a store buffer, one of many layers of cache, or in main memory. If threads are to share this value, it needs to be made visible in an ordered fashion and this is achieved through the coordinated exchange of cache coherency messages. The timely generation of these messages can be controlled by memory barriers.

A read memory barrier orders load instructions on the CPU that executes it by marking a point in 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 marking a point in 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 these three are sufficient to understand the complexities of what is involved. In the Java memory model the read and write of a *volatile* field implements the read and write barriers respectively. This was made explicit in the Java Memory Model[3] as defined with the release of Java 5.

3.4 Cache Lines

The way in which caching is used in modern processors is of immense importance to successful high performance operation. Such processors are enormously efficient at churning through data and instructions held in cache and yet, comparatively, are massively inefficient when a cache miss occurs.

Our hardware does not move memory around in bytes or words. For efficiency, caches are organised into cache-lines that are typically 32-256 bytes in size, the most common cache-line being 64 bytes. This is the level of granularity at which cache coherency protocols operate. This means that if two variables are in the same cache line, and they are written to by different threads, then they present the same problems of write contention as if they were a single variable. This is a concept know as "false sharing". For high performance then, it is important to ensure that independent, but concurrently written, variables do not share the same cache-line if contention is to be minimised.

When accessing memory in a predictable manner CPUs are able to hide the latency cost of accessing main memory by predicting which memory is likely to be accessed next and pre-fetching it into the cache in the background. This only works if the processors can detect a pattern of access such as walking memory with a predictable "stride". When iterating over the contents of an array the stride is predictable and so memory will be pre-fetched in cache lines, maximizing the efficiency of the access. Strides typically have to be less than 2048 bytes in either direction to be noticed by the processor. However, data structures like linked lists and trees tend to have nodes that are more widely distributed in memory with no predictable stride of access. The lack of a consistent pattern in memory constrains the ability of the system to pre-fetch cache-lines, resulting in main memory accesses which can be more than 2 orders of magnitude less efficient.

3.5 The Problems of Queues

Queues typically use either linked-lists or arrays for the underlying storage of elements. If an in-memory queue is allowed to be unbounded then for many classes of problem it can grow unchecked until it reaches the point of catastrophic failure by exhausting memory. This happens when producers outpace the consumers. Unbounded queues can be useful in systems where the producers are guaranteed not to outpace the consumers and memory is a precious resource, but there is always a risk if this assumption doesn't hold and queue grows without limit. To avoid this catastrophic outcome, queues are commonly constrained in size (bounded). Keeping a queue bounded requires that it is either array-backed or that the size is actively tracked.

Queue implementations tend to have write contention on the head, tail, and size variables. When in use, queues are typically always close to full or close to empty due to the differences in pace between consumers and producers. They very rarely operate in a balanced middle ground where the rate of production and consumption is evenly matched. This propensity to be always full or always empty results in high levels of contention and/or expensive cache coherence. The problem is that even when the head and tail mechanisms are separated using different concurrent objects such as locks or CAS variables, they generally occupy the same cache-line.

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 implementations very complex to manage beyond using a single large-grain lock on the queue. Large grain locks on the whole queue for *put* and *take* operations are simple to implement but represent a significant bottleneck to throughput. If the concurrent concerns are teased apart within the semantics of a queue then the implementations become very complex for anything other than a single producer – single consumer implementation.

In Java there is a further problem with the use of queues, as they are significant sources of garbage. Firstly, objects have to be allocated and placed in the queue. Secondly, if linked-list backed, objects have to be allocated representing the nodes of the list. When no longer referenced, all these objects allocated to support the queue implementation need to be re-claimed.

3.6 Pipelines and Graphs

For many classes of problem it makes sense to wire together several processing stages into pipelines. Such pipelines often have parallel paths, being organised into graph-like topologies. The links between each stage are often implemented by queues with each stage having its own thread.

This approach is not cheap - at each stage we have to incur the cost of en-queuing and de-queuing units of work. The number of targets multiplies this cost when the path must fork, and incurs an inevitable cost of contention when it must re-join after such a fork.

It would be ideal if the graph of dependencies could be expressed without incurring the cost of putting the queues between stages.

4 The Design of the LMAX disruptor

While trying to address the problems described above, a design emerged through a rigorous separation of the concerns that we saw as being conflated in queues. This approach was combined with a focus on ensuring that any data should be owned by only one thread for write access, therefore eliminating write contention. That design became known as the "Disruptor". It was so named because it had elements of similarity for dealing with graphs of dependencies to the concept of "Phasers"[4] in Java 7, introduced to support Fork-Join.

The LMAX disruptor is designed to address all of the issues outlined above in an attempt to maximize the efficiency of memory allocation, and operate in a cache-friendly manner so that it will perform optimally on modern hardware.

At the heart of the disruptor mechanism sits a pre-allocated bounded data structure in the form of a ring-buffer. Data is added to the ring buffer through one or more producers and processed by one or more consumers.

4.1 Memory Allocation

All memory for the ring buffer is pre-allocated on start up. A ring-buffer can store either an array of pointers to entries or an array of structures representing the entries. The limitations of the Java language mean that entries are associated with the ring-buffer as pointers to objects. Each of these entries is typically not the data being passed itself, but a container for it. This pre-allocation of entries eliminates issues in languages that support garbage collection, since the entries will be re-used and live for the duration of the Disruptor instance. The memory for these entries is allocated at the same time and it is highly likely that it will be laid out contiguously in main memory and so support cache striding. There is a proposal by John Rose to introduce "value types"[5] to the Java language which would allow arrays of tuples, like other languages such as C, and so ensure that memory would be allocated contiguously and avoid the pointer indirection.

Garbage collection can be problematic when developing low-latency systems in a managed runtime environment like Java. The more memory that is allocated the greater the burden this puts on the garbage collector. Garbage collectors work at their best when objects are either very short-lived or effectively immortal. The pre-allocation of entries in the ring buffer means that it is immortal as far as garbage collector is concerned and so represents little burden.

Under heavy load queue-based systems can back up, which can lead to a reduction in the rate of processing, and results in the allocated objects surviving longer than they should, thus being promoted beyond the young generation with generational garbage collectors. This has two implications: first, the objects have to be copied between generations which cause latency jitter; second, these objects have to be collected from the old generation which is typically a much more expensive operation and increases the likelihood of "stop the world" pauses that result when the fragmented memory space requires compaction. In large memory heaps this can cause pauses of seconds per GB in duration.

4.2 Teasing Apart the Concerns

We saw the following concerns as being conflated in all queue implementations, to the extent that this collection of distinct behaviours tend to define the interfaces that queues implement:

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

When designing a financial exchange in a language that uses garbage collection, too much memory allocation can be problematic. So, as we have described linked-list backed queues are a not a good approach. Garbage collection is minimized if the entire storage for the exchange of data between processing stages can be pre-allocated. Further, if this allocation can be performed in a uniform chunk, then traversal of that data will be done in a manner that is very friendly to the caching strategies employed by modern processors. A data-structure that meets this requirement is an array with all the slots pre-filled. On creation of the ring buffer the Disruptor utilises the abstract factory pattern to pre-allocate the entries. When an entry is claimed, a producer can copy its data into the pre-allocated structure.

On most processors there is a very high cost for the remainder calculation on the sequence number, which determines the slot in the ring. This cost can be greatly reduced by making the ring size a power of 2. A bit mask of size minus one can be used to perform the remainder operation efficiently.

As we described earlier bounded queues suffer from contention at the head and tail of the queue. The ring buffer data structure is free from this contention and concurrency primitives because these concerns have been teased out into producer and consumer barriers through which the ring buffer must be accessed. The logic for these barriers is described below.

In most common usages of the Disruptor there is usually only one producer. Typical producers are file readers or network listeners. In cases where there is a single producer there is no contention on sequence/entry allocation.

In more unusual usages where there are multiple producers, producers will race one another to claim the next entry in the ring-buffer. Contention on claiming the next available entry can be managed with a simple CAS operation on the sequence number for that slot.

Once a producer has copied the relevant data to the claimed entry it can make it public to consumers by committing the sequence. This can be done without CAS by a simple busy spin until the other producers have reached this sequence in their own commit. Then this producer can advance the cursor signifying the next available entry for consumption. 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. Lock free multi-producer – multi-consumer queues do exist but they require multiple CAS operations on the head, tail, size counters. The Disruptor does not suffer this CAS contention.

4.3 Sequencing

Sequencing is the core concept to how the concurrency is managed in the Disruptor. Each producer and consumer works off a strict sequencing concept for how it interacts with the ring buffer. Producers claim the next slot in sequence when claiming an entry in the ring. This sequence of the next available slot can be a simple counter in the case of only one 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 claiming producer. When the producer has finished updating the entry it can commit the changes by updating a separate counter which represents the cursor on the ring buffer for the latest entry available to consumers. The ring buffer cursor can be read and written in a busy spin by the producers using memory barrier without requiring a CAS operation as below.

```
long expectedSequence = claimedSequence - 1;
while (cursor != expectedSequence)
{
    // busy spin
}

cursor = claimedSequence;
```

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 from wrapping. Consumer sequences also allow consumers to coordinate work on the same entry in an ordered manner

In the case of 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.

4.4 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 lagging consumer 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. Based on our observations, this effect results in a close to constant time for latency regardless of load, up until the memory sub-system is saturated, and then the profile is linear following Little's Law[6]. This is very different to the "J" curve effect on latency we have observed with queues as load increases.

4.5 Dependency Graphs

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

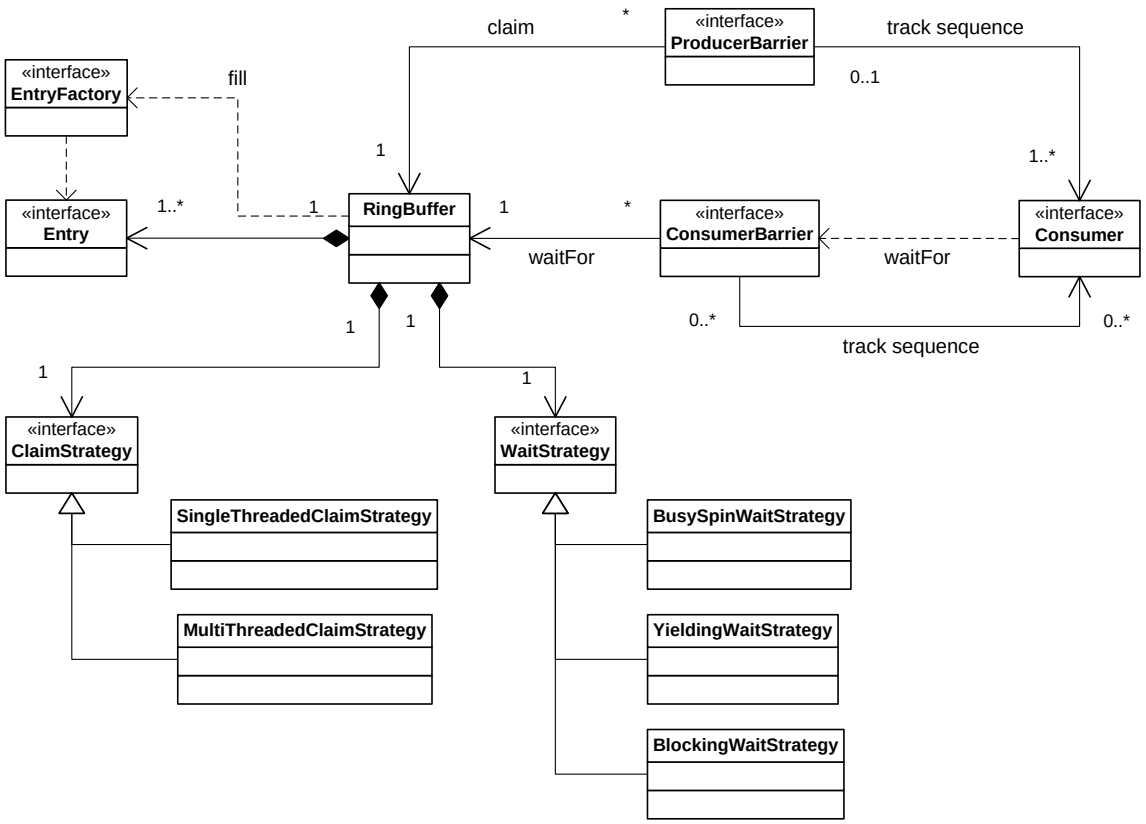
Because the producer and consumer concerns are separated with the Disruptor pattern, it is possible to represent a complex graph of dependencies between consumers while only using a single ring buffer at the core. This results in greatly reduced fixed costs of execution thus increasing throughput while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

A single ring buffer can be used to store entries with a complex structure representing the whole workflow in a cohesive place. Care must be taken in the design of such a structure so that the state written by independent consumers does not result in false sharing of cache lines.

4.6 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.

Separating the concerns normally conflated in queue implementations allows for a more flexible design. A *RingBuffer* exists at the core of the Disruptor pattern providing storage for data exchange without contention. The concurrency concerns are separated out for the producers and consumers interacting with the *RingBuffer*. The *ProducerBarrier* manages any concurrency concerns associated with claiming slots in the ring buffer, while tracking dependant consumers to prevent the ring from wrapping. The *ConsumerBarrier* notifies consumers when new entries are available, and *Consumers* can be constructed into a graph of dependencies representing multiple stages in a processing pipeline.



4.7 Code Example

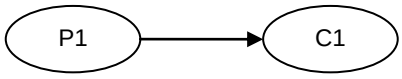
The code below is an example of a single producer and single consumer using the convenience interface *BatchHandler* for implementing a consumer. The consumer runs on a separate thread receiving entries as they become available.

```
// Callback handler which can be implemented by consumers
final BatchHandler<ValueEntry> batchHandler = new BatchHandler<ValueEntry>()
{
    public void onAvailable(final ValueEntry entry) throws Exception
    {
        // process a new entry as it becomes available.
    }

    public void onEndOfBatch() throws Exception
    {
        // useful for flushing results to an IO device if necessary.
    }

    public void onCompletion()
    {
        // do any necessary clean up before shutdown
    }
};

RingBuffer<ValueEntry> ringBuffer =
    new RingBuffer<ValueEntry>(ValueEntry.ENTRY_FACTORY, SIZE,
                               ClaimStrategy.Option.SINGLE_THREADED,
                               WaitStrategy.Option.YIELDING);
ConsumerBarrier<ValueEntry> consumerBarrier = ringBuffer.createConsumerBarrier();
BatchConsumer<ValueEntry> batchConsumer =
    new BatchConsumer<ValueEntry>(consumerBarrier, batchHandler);
ProducerBarrier<ValueEntry> producerBarrier = ringBuffer.createProducerBarrier(batchConsumer);

// Each consumer can run on a separate thread
EXECUTOR.submit(batchConsumer);
```

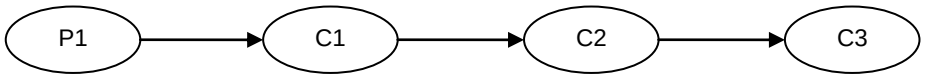
```
// Producers claim entries in sequence
ValueEntry entry = producerBarrier.nextEntry();

// copy data into the entry container

// make the entry available to consumers
producerBarrier.commit(entry);
```

5 Throughput Performance Testing

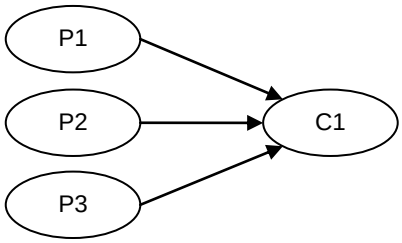
As a reference we choose Doug Lea's excellent *java.util.concurrent.ArrayBlockingQueue*[7] which has the highest performance of any bounded queue based on our testing. The tests are conducted in a blocking programming style to match that of the Disruptor. The tests cases detailed below are available in the Disruptor open source project. **Note**: running the tests requires a system capable of executing at least 4 threads in parallel.



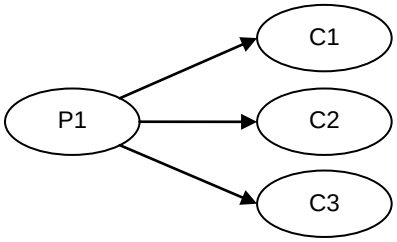
Unicast: 1P – 1C



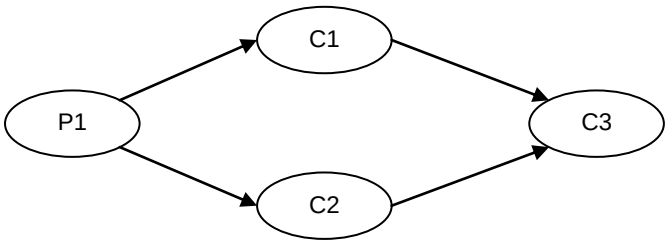
Three Step Pipeline: 1P – 3C



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

Table 2 - Comparative throughput (in ops per sec)

6 Latency Performance Testing

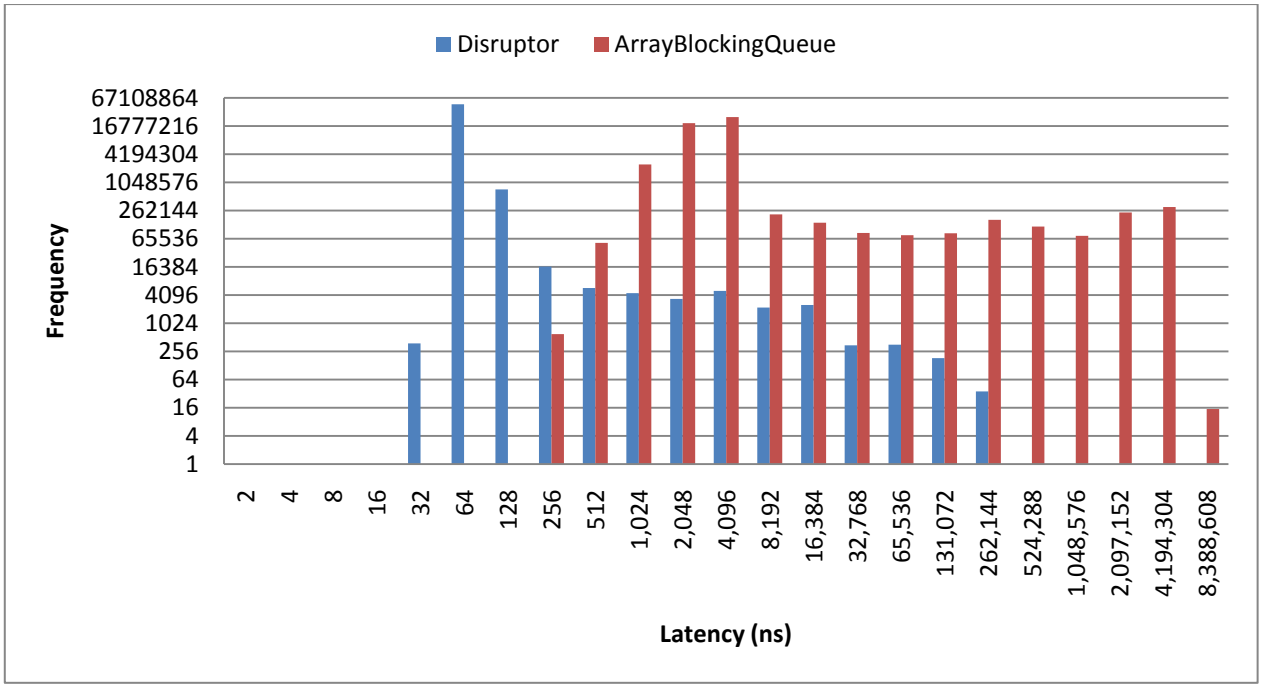
To measure latency we take the three stage pipeline and generate events at less than saturation. This is achieved by waiting 1 microsecond after injecting an event before injecting the next and repeating 50 million times. To time at this level of precision it is necessary to use time stamp counters from the CPU. We chose CPUs with an invariant TSC because older processors suffer from changing frequency due to power saving and sleep states. Intel Nehalem and later processors use an invariant TSC which can be accessed by the latest Oracle 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*[8] which is likely to give better results but we want to use a bounded 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.

Mean latency per hop for the Disruptor comes out at 52 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)
Min Latency	145	29
Mean Latency	32,757	52
99% observations less than	2,097,152	128
99.99% observations less than	4,194,304	8,192
Max Latency	5,069,086	175,567

Table 3 - Comparative Latency in three stage pipeline



7 Conclusion

The Disruptor is a major step forward for increasing throughput, reducing latency between concurrent execution contexts and ensuring predictable latency, an important consideration in many applications. Our testing shows that it out-performs comparable approaches for exchanging data between threads. We believe that this is the highest performance mechanism for such data exchange. By concentrating on a clean separation of the concerns involved in cross-thread data exchange, by eliminating write contention, minimizing read contention and ensuring that the code worked well with the caching employed by modern processors, we have created a highly efficient mechanism for exchanging data between threads in any application.

The batching effect that allows consumers to process entries up to a given threshold, without any contention, introduces a new characteristic in high performance systems. For most systems, as load and contention increase there is an exponential increase in latency, the characteristic “J” curve. As load increases on the Disruptor, latency remains almost flat until saturation occurs of the memory sub-system.

We believe that the Disruptor establishes a new benchmark for high-performance computing and is very well placed to continue to take advantage of current trends in processor and computer design.

[1] Staged Event Driven Architecture – http://www.eecs.harvard.edu/~mdw/proj/seda/
[2] Actor model – http://dspace.mit.edu/handle/1721.1/6952
[3] Java Memory Model - http://www.ibm.com/developerworks/library/j-jtp02244/index.html
[4] Phasers - http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/Phaser.html
[5] Value Types - http://blogs.oracle.com/jrose/entry/tuples_in_the_vm
[6] Little’s Law - http://en.wikipedia.org/wiki/Little%27s_law
[7] ArrayBlockingQueue - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ArrayBlockingQueue.html
[8] ConcurrentLinkedQueue - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html