**Disruptor:**

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

高性能可替代有界队列，用于在并发线程之间交换数据

Martin Thompson  
Dave Farley  
Michael Barker  
Patricia Gee  
Andrew Stewart

May-2011

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

# 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.

建立了一个非常高效的金融交易所。作为实现这一目标的工作的一部分，我们评估了设计这样一个系统的几种方法，但当我们开始衡量这些方法时，我们遇到了传统方法的一些基本限制。

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.

许多应用程序依赖队列在处理阶段之间交换数据。我们的性能测试表明，当以这种方式使用队列时，延迟成本与对磁盘(基于RAID或SSD的磁盘系统)进行IO操作的成本处于同一量级——显著地降低。如果端到端操作中有多个队列，那么总延迟将增加数百微秒。显然存在优化的空间。

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.

进一步的研究和对计算机科学的关注使我们意识到，传统方法(例如队列和处理节点)中固有的关注点的合并会导致多线程实现中的争用，这表明可能有更好的方法。

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.

考虑到现代cpu的工作方式，我们称之为“机械同情”(mechanical sympathy)，利用良好的设计实践，将重点放在梳理关注点上，我们提出了一种数据结构和一种使用模式，我们将其称为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.

测试表明，使用Disruptor进行三级管道的平均延迟比等效的基于队列的方法低3个数量级。此外，对于相同的配置，Disruptor处理大约8倍以上的吞吐量。

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.

这些性能改进代表了对并发编程思想的一步改变。对于需要高吞吐量和低延迟的任何异步事件处理体系结构，这种新模式都是理想的基础.

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.

然而，这不是一个专业的解决方案，只有在金融行业相关。Disruptor是一种通用机制，以一种最大化性能的方式解决并发编程中的复杂问题，而且实现起来很简单。尽管有些概念可能看起来不太寻常，但根据我们的经验，按照这种模式构建的系统比可比机制的实现要简单得多。

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.

与同类方法相比，Disruptor具有更少的写争用、更低的并发开销和更友好的缓存，所有这些都导致更大的吞吐量、更少的抖动和更低的延迟。在中等时钟速率的处理器上，我们看到每秒超过2500万条消息，延迟低于50纳秒。与我们所见过的任何其他实现相比，此性能是一个显著的改进。这非常接近现代处理器在内核之间交换数据的理论极限。

# 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]](#endnote-0) and Actors[[2]](#endnote-1) 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.

# 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.

在本文的上下文中，以及一般的计算机科学中，并发性不仅意味着两个或多个任务并行发生，而且还意味着它们争夺对资源的访问。争用资源可以是数据库、文件、套接字，甚至是内存中的某个位置。

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.

代码的并发执行涉及到两件事，互斥和变更的可见性。互斥是关于管理对某些资源的争用更新。更改的可见性是指控制这些更改何时对其他线程可见。如果可以消除争用更新的需要，就可以避免互斥的需要。如果你的算法可以保证任何给定的资源只被一个线程修改，那么互斥就没有必要了。读写操作要求所有的修改对其他线程都是可见的。然而，只有争用的写操作才需要互斥更改。

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.

在任何并发环境中，代价最高的操作是争用写访问。要让多个线程写入相同的资源，需要复杂而昂贵的协调。这通常是通过使用某种锁定策略来实现的。

## 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.

锁提供互斥，并确保更改的可见性以有序的方式发生。锁非常昂贵，因为它们在争用时需要仲裁。这个仲裁是通过切换到操作系统内核来实现的，内核将挂起线程等待锁，直到锁被释放。在这样的上下文切换过程中，以及将控制权释放给操作系统(操作系统有控制权时可能决定执行其他内务任务)时，执行上下文可能会丢失之前缓存的数据和指令。这可能会对现代处理器产生严重的性能影响。可以使用快速用户模式锁，但只有在没有争用的情况下才有真正的好处。

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.

我们将通过一个简单的演示来说明锁的成本。这个实验的重点是调用一个函数，在一个循环中递增一个64位计数器5亿次。这可以由一个线程在2.4Ghz Intel Westmere EP上执行，如果用Java编写，只需300毫秒。这种语言对这个实验并不重要，所有的语言都具有相同的基本基元，其结果是相似的。

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:

一旦引入锁以提供互斥，即使锁还没有竞争，成本也会显著增加。当两个或多个线程开始竞争时，成本又会按数量级增加。这个简单实验的结果如下表所示:

|  |  |
| --- | --- |
| **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

## 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.

当更新的目标是一个单词时，可以使用一种比使用锁更有效的方法来更新内存。这些替代方案是基于现代处理器中实现的原子指令或连锁指令。这些操作通常被称为CAS (Compare And Swap)操作，例如x86上的“lock cmpxchg”。CAS操作是一种特殊的机器代码指令，它允许将内存中的一个字有条件地设置为一个原子操作。对于“增加计数器实验”，每个线程可以循环读取计数器，然后尝试原子地将其设置为新的增加的值。旧值和新值作为这个指令的参数提供。如果在执行操作时，计数器的值与所提供的期望值相匹配，则计数器将被更新为新值。如果该值不符合要求，则CAS操作将失败。然后由试图执行更改的线程重试，重新读取从该值递增的计数器，以此类推，直到更改成功。这种CAS方法比锁的效率高得多，因为它不需要上下文切换到内核进行仲裁。然而，CAS操作并不是免费的。处理器必须锁定它的指令管道，以确保原子性，并使用内存屏障使更改对其他线程可见。在Java中，可以通过使用Java .util.concurrent来使用CAS操作。*java.util.concurrent.Atomic\**类。

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.

如果程序的临界部分比计数器的简单增量更复杂，则可能需要使用复杂的状态机使用多个CAS操作来协调争用。使用锁开发并发程序很困难;使用CAS操作和内存屏障开发无锁算法要复杂很多倍，而且很难证明它们是正确的。

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.

理想的算法是只有一个线程拥有对单个资源的所有写操作，而其他线程读取结果。要在多处理器环境中读取结果，需要内存屏障来让在其他处理器上运行的线程可以看到更改。

## 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.

由于性能原因，现代处理器在内存和执行单元之间执行指令的无序执行以及数据的无序加载和存储。处理器只需要保证无论执行顺序如何，程序逻辑都能产生相同的结果。这对于单线程程序来说不是问题。但是，当线程处于共享状态时，重要的是所有内存更改必须按顺序出现，以便成功进行数据交换。内存屏障被处理器用来指示代码段中内存更新的顺序是重要的。它们是实现线程之间更改的硬件排序和可见性的方法。编译器可以设置互补的软件屏障来确保编译代码的顺序，这种软件内存屏障是在处理器本身使用的硬件屏障之外的。

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.

现在的cpu比现在的内存系统要快得多。为了跨越这种划分，cpu使用了复杂的缓存系统，这种系统是快速有效的硬件哈希表，没有链接。这些缓存通过消息传递协议与其他处理器缓存系统保持一致。此外，处理器有“存储缓冲区(*store buffers*)”来卸载对这些缓存的写操作，以及“使队列失效(*invalidate queues*)”，这样当写操作即将发生时，缓存一致性协议可以快速确认失效消息以提高效率。

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.

对于数据来说，这意味着任何值的最新版本，在写入后的任何阶段，都可以在寄存器、存储缓冲区、多个缓存层之一或主存中。如果线程要共享这个值，则需要以一种有序的方式使其可见，这是通过协调交换缓存一致性消息来实现的。这些消息的及时生成可以由内存屏障控制。

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.

读内存屏障命令通过在invalidate queue中标记一个点来给CPU执行指令。这使它对写操作在读屏障之前的顺序有一个一致的看法。

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.

写内存屏障命令将指令存储在CPU上，CPU通过在store buffer中标记一个点来执行指令，这样就可以通过它的cache刷新写操作。这个barrier为写barrier之前发生的存储操作提供了一个有序的视图。

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

一个完整的内存屏障命令加载和存储，但只在执行它的CPU上。

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]](#endnote-2) as defined with the release of Java 5.

除了这三个原语之外，有些cpu还有更多的变体，但这三个变体足以理解所涉及的复杂性。在Java内存模型中，volatile字段的读写分别实现了读写屏障。这在Java内存模型[Java内存模型- http://www.ibm.com/developerworks/library/j-jtp02244/index.html]中被明确地定义为Java 5版本。

## 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.

高速缓存在现代处理器中的使用方式对于成功的高性能操作至关重要。这样的处理器在处理缓存中的数据和指令时效率非常高，但是，相对而言，当缓存丢失发生时效率非常低。

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.

我们的硬件不以字节或字的形式移动内存。为了提高效率，缓存通常被组织成32-256字节大小的缓存行，最常见的缓存行是64字节。这是缓存一致性协议操作的粒度级别。这意味着，如果两个变量在同一个缓存行中，并且它们被不同的线程写入，那么它们就会出现写争用的问题，就像它们是单个变量一样。这就是所谓的“虚假共享”。为了提高性能，如果要尽量减少争用，确保独立但并发写入的变量不共享相同的缓存行是很重要的。

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.

当以一种可预测的方式访问内存时，cpu能够通过预测下一个可能被访问的内存并将其预取到后台的缓存中来隐藏访问主存的延迟成本。只有当处理器能够检测到一种访问模式，例如以可预测的“步幅”行走内存时，这才会起作用。当遍历数组的内容时，“步幅”是可预测的，因此内存将在缓存行中预取，以最大限度地提高访问效率。处理器注意到的跨距通常必须在两个方向上小于2048字节。然而，像链表和树这样的数据结构往往具有分布在内存中更广泛的节点，并且没有可预测的访问步长。内存中缺乏一致的模式限制了系统预取缓存线的能力，导致主存访问的效率降低了两个数量级以上。

## 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.

队列通常使用链表或数组作为元素的底层存储。如果允许内存中的队列无限制，那么对于许多类的问题来说，它会无限制地增长，直到它到达耗尽内存的灾难性故障点。当生产者的步伐超过消费者时，就会出现这种情况。无界队列在保证生产者的速度不会超过消费者的系统中是有用的，而且内存是一种宝贵的资源，但是如果这个假设不成立，并且队列的增长没有限制，那么总是有风险的。为了避免这种灾难性的结果，队列通常在大小上受到限制(有界)。保持队列有界要求它要么支持数组，要么主动跟踪队列大小。

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.

队列实现倾向于在头、尾和大小变量上产生写争用。在使用时，由于消费者和生产者之间的速度差异，队列通常总是接近满或接近空。它们很少能在生产和消费速度相等的平衡的中间地带运作。这种总是满的或总是空的倾向导致了高水平的争用和/或昂贵的缓存一致性。问题是，即使头和尾机制使用不同的并发对象(如锁或CAS变量)分离，它们通常也占用相同的缓存线。

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.

管理生产者声明队列的头部、消费者声明队列的尾部以及两者之间的节点存储的问题，使得并发实现的设计非常复杂，难以管理，无法在队列上使用单个大粒度锁。整个队列上用于put和take操作的大粒度锁实现起来很简单，但却是吞吐量的一大瓶颈。如果并行关注在队列的语义中被分开，那么实现就会变得非常复杂，而不是单一生产者-单一消费者的实现.

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.

在Java中，队列的使用还有一个进一步的问题，因为它们是垃圾的重要来源。首先，必须分配对象并将其放入队列中。其次，如果支持链表，则必须分配表示链表节点的对象。当不再被引用时，所有为支持队列实现而分配的对象都需要重新声明。

## 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.

对于许多类型的问题，将几个处理阶段连接到管道中是有意义的。这样的管道通常有平行的路径，被组织成类似图形的拓扑。每个阶段之间的链接通常由队列实现，每个阶段都有自己的线程。这种方法并不便宜——在每一个阶段，我们都要付出排队和排队退出工作单元的代价。当路径必须分叉时，目标的数量会增加这个开销，当它在这样一个分叉之后必须重新加入时，就会不可避免地产生争用开销。如果能够表达依赖关系图而不产生在阶段之间放置队列的成本，那将是理想的。

# 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]](#endnote-3) in Java 7, introduced to support Fork-Join.

在试图解决上述问题时，通过严格分离我们看到的合并在队列中的关注点，出现了一种设计。这种方法结合了一个重点，即确保任何数据都应该只由一个线程拥有，以便进行写访问，因此消除了写争用。这个设计被称为“颠覆者”。它之所以这样命名，是因为它在处理依赖关系图时具有与Java 7中的“Phasers”[Phasers - http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/Phaser.html]概念相似的元素，而“Phasers”是为了支持Fork-Join而引入的。

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.

LMAX disruptor 旨在解决上述所有问题，试图最大化内存分配的效率，并以缓存友好的方式操作，使其在现代硬件上表现最佳。

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.

在中断器机制的核心位置有一个预先分配的有界数据结构，其形式为环形缓冲区。数据通过一个或多个生产者添加到循环缓冲区，并由一个或多个消费者处理。

## 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]](#endnote-4) 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.

所有用于环形数组的内存在启动时预先分配。环形数组既可以存储指向条目的指针数组，也可以存储表示条目的结构数组。Java语言的限制意味着条目作为对象的指针与循环缓冲区相关联。这些条目通常不是传递的数据本身，而是传递数据的容器。这种条目的预分配消除了支持垃圾收集的语言中的问题，因为条目将被重用，并在Disruptor实例的持续时间内有效。这些条目的内存是同时分配的，它很可能会连续地分布在主存中，因此支持缓存跨越。有一个提议由John Rose引入“值类型”(值类型——http://blogs.oracle.com/jrose/entry/tuples\_in\_the\_vm)的Java语言将使数组的元组,像其他语言(如C,所以确保连续内存分配,避免间接的指针。

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.

在Java等托管运行时环境中开发低延迟系统时，垃圾收集可能会有问题。分配的内存越多，给垃圾收集器带来的负担就越大。当对象要么寿命很短，要么实际上是不朽的时候，垃圾收集器的工作状态最佳。循环缓冲区中条目的预分配意味着，就垃圾收集器而言，它是不朽的，因此负担很小。

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.

在高负载下，基于队列的系统可能会进行备份，这可能会导致处理速度下降，并导致分配的对象存活时间长于它们应该存活的时间，从而通过代垃圾收集器提升到年轻代之外。这有两个含义:首先，对象必须在代与代之间复制，这会导致延迟抖动;其次，这些对象必须从旧的代中收集，这通常是一个更昂贵的操作，并增加了“stw”停顿的可能性，当碎片内存空间需要压缩时，这种停顿会导致“stw”停顿。在大内存堆中，这可能导致每GB持续几秒的暂停。

## 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

在所有队列实现中，我们看到以下问题被合并在一起，这一系列不同的行为倾向于定义队列实现的接口:

1. 交换对象的储存
2. 要求交换下一个序列的生产者的协调

3.协调被告知有新项目可用的消费者

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.

在用使用垃圾收集的语言设计金融交换时，过多的内存分配可能会产生问题。因此，正如我们所描述的，链表支持队列并不是一种好方法。如果可以预先分配处理阶段之间交换数据的整个存储空间，则垃圾收集将最小化。此外，如果这种分配可以在一个统一的块中执行，那么该数据的遍历将以一种对现代处理器所采用的缓存策略非常友好的方式进行。满足这一要求的数据结构是一个预先填充了所有槽的数组。在创建环形缓冲区时，Disruptor利用抽象工厂模式来预分配条目。当一个条目被声明时，生产者可以将它的数据复制到预先分配的结构中。

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.

在大多数处理器上，对序列号进行剩余计算的成本非常高，序列号决定了环中的槽。这一成本可以大大降低，使ring的大小为幂2。一个大小为- 1的位掩码可以有效地执行余数运算。

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.

正如我们前面所描述的，有界队列在队列的头和尾遭受争用。环缓冲区数据结构不受这种争用和并发原语的影响，因为这些问题被划分为生产者和消费者屏障，环缓冲区必须通过这些屏障进行访问。这些障碍的逻辑如下所述。

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.

在大多数Disruptor 的常见用法中，通常只有一个生产者。典型的生产者是 file readers或network listeners。在只有一个生产者的情况下，在序列/条目分配上没有争用。

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.

在更不寻常的情况下，如果有多个生产者，生产者将会在环形队列中竞争下一个条目。声明下一个可用条目时的争用可以通过对该槽的序列号进行简单的CAS操作来管理。

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.

一旦生产者将相关数据复制到声明的条目中，它就可以通过提交序列将其公开给消费者。这可以在没有CAS的情况下通过简单的do..while(true)来完成，直到其他生产者在他们自己的提交中达到这个顺序。然后这个生产者可以向前移动光标，表示下一个可供消费的条目。生产者可以通过跟踪消费者的序列来避免包装环在他们写入环缓冲区之前，作为一个简单的读操作。

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.

消费者在读取条目之前，会等待一个序列在循环缓冲区中可用。在等待的过程中可以使用各种策略。如果CPU资源是宝贵的，他们可以等待锁中的条件变量得到生产者的信号。这显然是一个争用点，并且只在CPU资源比延迟或吞吐量更重要时才使用。使用者还可以循环检查表示循环缓冲区中当前可用序列的游标。这可以通过用CPU资源交换延迟来完成，也可以不使用线程yield。这可以很好地扩展，因为如果我们不使用锁和条件变量，就会打破生产者和消费者之间的争用依赖关系。无锁的多生产者-多消费者队列确实存在，但它们需要在头、尾、大小计数器上进行多次CAS操作。干扰者不会遭受这种CAS争用。

## 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.

Sequencing 是如何在Disruptor中管理并发的核心概念。每个生产者和消费者都有一个严格的排序概念，来决定它是如何与环形缓冲区交互的。当在擂台上申请参赛时，生产者会按顺序申请下一个位置。下一个可用槽的序列在只有一个生产者的情况下可以是一个简单的计数器，在有多个生产者的情况下可以是使用CAS操作更新的原子计数器。一旦序列值被声明，循环缓冲区中的这个条目现在就可以被声明生成器写入。当生产者完成对条目的更新后，它可以通过更新一个单独的计数器来提交更改，该计数器表示消费者可以使用的最新条目在循环缓冲区上的游标。生产者可以使用内存屏障在繁忙的旋转中读写环缓冲游标，而不需要下面的CAS操作。

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.

消费者通过使用内存屏障读取游标来等待给定序列变为可用。一旦游标被更新，内存屏障将确保对环缓冲区中条目的更改对等待游标前进的消费者是可见的。

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

每个消费者都包含自己的序列，它们在处理来自循环缓冲区的条目时更新这些序列。这些消费者序列允许生产者跟踪消费者，以防止环被包装。消费者序列还允许消费者以有序的方式协调对同一条目的工作

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.

在只有一个生产者的情况下，无论消费者图的复杂性如何，都不需要锁或CAS操作。只需对所关注的序列设置内存屏障，就可以实现整个并发协调

## 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]](#endnote-5). This is very different to the “J” curve effect on latency we have observed with queues as load increases.

当消费者在循环缓冲区中等待前进的游标序列时，会出现一个有趣的机会，而队列是不可能做到这一点的。如果使用者发现循环缓冲区游标比上次检查时前进了许多步，那么它可以在不涉及并发机制的情况下处理该序列。这导致落后的消费者迅速恢复与生产者的步伐，当生产者突破，从而平衡系统。这种批处理增加了吞吐量，同时减少和平滑了延迟。根据我们的观察，无论负载如何，这种效应都会导致一个接近恒定的延迟时间，直到内存子系统饱和，然后概要遵循利特尔定律[利特尔定律- http://en.wikipedia.org/wiki/Little%27s\_law]为线性。这与我们观察到的随着负载增加的队列对延迟的“J”曲线效应非常不同。

## 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.

队列表示生产者和消费者之间简单的一步管道依赖关系。如果消费者形成了一个链或图状的依赖结构，那么图的每个阶段之间都需要队列。这在依赖阶段图中多次引起队列的固定成本。在设计LMAX金融交易所时，我们的分析显示，采用基于队列的方法会导致处理事务的总执行成本主要由队列成本决定。

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.

因为生产者和消费者的关注点与Disruptor模式分离，所以可以在核心只使用一个循环缓冲区的情况下表示消费者之间依赖关系的复杂图形。这大大降低了执行的固定成本，从而增加了吞吐量，同时减少了延迟。

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.

单个循环缓冲区可以用于存储具有复杂结构的条目，该结构在一个内聚的地方表示整个工作流。在设计这样的结构时必须小心，这样独立的消费者写的状态不会导致缓存线的错误共享。

## 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.

在下面的类图中描述了Disruptor框架中的核心关系。这个图省略了一些方便的类，这些类可以用来简化编程模型。在构建了依赖图之后，编程模型就变得简单了。生产者通过ProducerBarrier顺序声明条目，将他们的更改写入声明的条目中，然后通过ProducerBarrier提交该条目，使其可以被使用。作为一个消费者，我们所需要做的就是提供一个BatchHandler实现，当一个新条目可用时，该实现接收回调。这个生成的编程模型是基于事件的，与Actor模型有很多相似之处。

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.

将通常合并在队列实现中的关注点分开，可以实现更灵活的设计。RingBuffer存在于Disruptor模式的核心，为数据交换提供存储而不产生争用。对于与RingBuffer交互的生产者和消费者来说，并发问题是分开的。ProducerBarrier管理与在环缓冲区中声明插槽相关的任何并发问题，同时跟踪依赖的消费者，以防止环包装。当有新的条目可用时，ConsumerBarrier会通知消费者，消费者可以被构造成一个表示处理管道中多个阶段的依赖关系图。



## 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.

下面的代码是一个单一生产者和单一消费者的例子，它使用方便的接口BatchHandler来实现消费者。消费者在一个单独的线程上运行，接收可用的条目。

// 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);

# Throughput Performance Testing

As a reference we choose Doug Lea’s excellent *java.util.concurrent.ArrayBlockingQueue*[[7]](#endnote-6) 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.

作为参考，我们选择Doug Lea的优秀的java.util.concurrent。ArrayBlockingQueue[ArrayBlockingQueue - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ArrayBlockingQueue.html]根据我们的测试，它是所有有界队列中性能最高的。测试以拦截编程风格进行，以匹配Disruptor的风格。下面详细介绍的测试用例可在Disruptor开源项目中获得。注意:运行这些测试需要一个能够并行执行至少4个线程的系统。

P1

C1

**Unicast: 1P – 1C**

P1

C1

C2

C3

**Three Step Pipeline: 1P – 3C**

**Sequencer: 3P – 1C**

P1

C1

P2

P3

**Multicast: 1P – 3C**

C1

P1

C2

C3

**Diamond: 1P – 3C**

C1

P1

C3

C2

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.

对于上述配置，ArrayBlockingQueue应用于每个数据流弧线，而不是使用Disruptor的屏障配置。下表显示了使用Java 1.6.0\_25 64位Sun JVM、Windows 7、Intel Core i7 860 @ 2.8 GHz(无HT)、Intel Core i7- 2720qm、Ubuntu 11.04时的每秒操作的性能结果，并在处理5亿条消息时采用3种运行方式的最佳结果。不同的JVM执行的结果可能有很大的不同，下面的数字并不是我们观察到的最高的。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **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)

# 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.

为了测量延迟，我们采用了三个阶段的管道，并在低于饱和的情况下生成事件。这是通过在注入一个事件后等待1微秒，然后再注入下一个事件并重复5000万次来实现的。要达到这种精度，必须从CPU使用时间戳计数器。我们选择了具有不变TSC的cpu，因为由于省电和休眠状态，旧处理器的频率会发生变化。Intel Nehalem和以后的处理器使用不变TSC，它可以被Ubuntu 11.04上运行的最新Oracle jvm访问。这个测试没有使用CPU绑定。

For comparison we use the *ArrayBlockingQueue* once again. We could have used *ConcurrentLinkedQueue*[[8]](#endnote-7) 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.

为了比较，我们再次使用ArrayBlockingQueue。我们可以使用ConcurrentLinkedQueue [ConcurrentLinkedQueue - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html]可能给更好的结果,但是我们希望使用一个有限队列实现,确保生产者不超过消费者通过创建背压。下面的结果是在Ubuntu 11.04上运行Java 1.6.0\_25 64位的2.2Ghz Core i7-2720QM上。

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*.

Disruptor的平均每跳延迟是52纳秒，而ArrayBlockingQueue的平均每跳延迟是32757纳秒。分析表明，锁的使用和通过条件变量发送信号是ArrayBlockingQueue延迟的主要原因。

|  |  |  |
| --- | --- | --- |
|  | **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

# 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.

Disruptor是提高吞吐量、减少并发执行上下文之间的延迟和确保可预测延迟的重要一步，这是许多应用程序的重要考虑因素。我们的测试表明，在线程之间交换数据时，它的性能优于类似的方法。我们认为这是此类数据交换的最高性能机制。通过专注于对跨线程数据交换所涉及的关注点进行清晰的分离，消除写争用，最小化读争用，并确保代码与现代处理器使用的缓存工作良好，我们已经为任何应用程序中的线程间交换数据创建了一种高效的机制。

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.

批处理效应允许使用者在给定阈值前处理条目，而不产生任何争用，这在高性能系统中引入了一个新特性。对于大多数系统，随着负载和争用的增加，延迟呈指数增长，即特征“J”曲线。随着负载在破坏者上的增加，延迟几乎保持平坦，直到饱和的内存子系统发生。

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.

我们相信，Disruptor为高性能计算建立了一个新的基准，并处于非常有利的位置，可以继续利用当前处理器和计算机设计的趋势。

1. Staged Event Driven Architecture – http://www.eecs.harvard.edu/~mdw/proj/seda/ [↑](#endnote-ref-0)
2. Actor model – http://dspace.mit.edu/handle/1721.1/6952 [↑](#endnote-ref-1)
3. Java Memory Model - http://www.ibm.com/developerworks/library/j-jtp02244/index.html [↑](#endnote-ref-2)
4. Phasers - http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/Phaser.html [↑](#endnote-ref-3)
5. Value Types - http://blogs.oracle.com/jrose/entry/tuples\_in\_the\_vm [↑](#endnote-ref-4)
6. Little’s Law - http://en.wikipedia.org/wiki/Little%27s\_law [↑](#endnote-ref-5)
7. ArrayBlockingQueue - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ArrayBlockingQueue.html [↑](#endnote-ref-6)
8. ConcurrentLinkedQueue - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html [↑](#endnote-ref-7)