DISRUPTOR - USING HIGH PERFORMANCE, LOW LATENCY TECHNOLOGY IN THE CERN CONTROL SYSTEM

M. Gabriel, R. Gorbonosov, CERN, Geneva, Switzerland

Abstract

Accelerator control systems process thousands of concurrent events per second, which adds complexity to their implementation. The Disruptor library provides an innovative single-threaded approach, which combines high performance event processing with a simplified software design, implementation and maintenance. This open-source library was originally developed by a financial company to build a low latency trading exchange. In 2014 the high-level control system for CERN experimental areas (CESAR) was renovated. CESAR calculates the states of thousands of devices by processing more than 2500 asynchronous event streams. The Disruptor was used as an event-processing engine. This allowed the code to be greatly simplified by removing the concurrency concerns. This paper discusses the benefits of the programming model encouraged by the Disruptor (simplification of the code base, performance, determinism), the design challenges faced while integrating the Disruptor into CESAR as well as the limitations it implies on the architecture.

INTRODUCTION

CESAR is the high level software used to control CERN experimental areas. The experimental areas are composed of eleven beam lines used by experimental physicists for fixed target research and detectors tests. Four beam lines are located in the East Area, using a beam extracted from the PS ring, and seven are in the North Area, using a beam extracted from the SPS ring. The core of CESAR is responsible for the data acquisition of all the devices controlling these beam lines. While refactoring this data acquisition part of Cesar, we decided to use the Disruptor library in order to simplify the design of the code handling the 2500 asynchronous event streams coming from these devices.

In the last decade, the actors of the world of finance and high frequency trading have been involved in an arms race to build exchanges and trading robots that can operate at the nanosecond scale. From time to time, some technologies created by the massive investments in this field are shared with the community [1].The Disruptor library was created by LMAX [2] -a London-based financial company- in order to develop a low-latency forex [3] trading venue [4]. In the early design phase, they tried different approaches: functional programming, Actors, SEDA [5], and noticed that they could not achieve the required latency because the cost of queuing was higher than the time spent executing the business logic. They finally settled on an innovative design and decided to open source it.

THEORETICAL BACKGROUND

The base idea around the Disruptor is to make the most of the available CPU resources, following a concept that its creators call ‘mechanical sympathy’. This term coming from the car racing world is used to describe software working in harmony with the hardware design, similar to a driver understanding how a car works in order to achieve the best performance. Since the appearance of multicore CPUs, we have heard expressions like “the free lunch is over” [6] and there is a general belief that CPUs are not getting any faster. Although their clock speed is not getting higher, modern CPUs have brought significant performance improvements. Regrettably, the progresses made in hardware are often lost by software designs that do not consider how modern processors work.

Feeding the Core

The most important aspect to consider in order to use a processor efficiently is to feed it correctly. Processors are very fast, but this speed is of little use if they spend most of their time waiting for the data they need to process. Looking at a simplified view of the memory architecture of a modern CPU such as Intel’s Sandy Bridge (Fig. 1), we see that the cores read data from several cache layers (L1, L2, and L3).

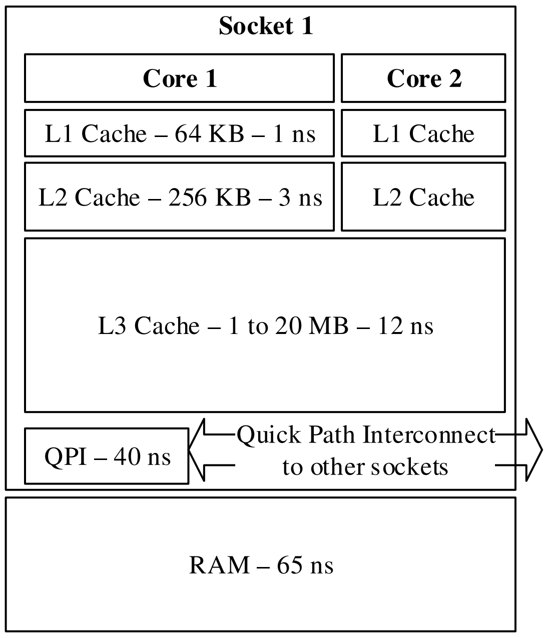

Figure 1: Memory Hierarchy

ISBN 978-3-95450-148-9

When a core needs to process data, it looks down the chain in the L1, L2 and L3 caches. If the data is not found in any cache, it is fetched from the main memory. As we physically move away from the core, each memory layer has an increased capacity, but is also several orders of magnitude slower than the previous one. In order to use the core at its maximum throughput, a program should work on data that is available within the caches.

Developers do not have to do this manually, because the processor is able to automatically prefetch the data in its caches. However, the prefetcher has a limitation: it only works if the memory is accessed with a predictable pattern: this means that the program has to walk through memory in a predictable stride (either forward or backward) [7]. This works very well while iterating on data structures that use memory allocated contiguously, such as arrays, but cannot be used with linked lists or trees because the prefetcher is unable to recognize an access pattern on these complex structures. This mechanism is not limited to one thread; on modern Intel processors, up to 32 streams of data can be prefetched concurrently.

Padding Cache Lines

Caches are composed of cache lines: these are memory blocks of fixed size (64 bytes on modern x86). Cache lines are the atomic units of memory used in caches: the prefetcher always loads full cache lines, and when a variable is written in a cache line, the full cache line is considered as being modified. The phenomenon known as 'false sharing' happens when two unrelated variables share the same cache line, and are written concurrently by two threads running on two different cores (Fig. 2). These threads constantly fight for the ownership of the cache line. Since each write of a variable results in the invalidation of the same cache line on the other core, the data needs to be reloaded. If the two cores are on the same socket, it can be reloaded from the L3 cache. If they are on different sockets, this battle is fought through the QPI bus (Fig. 1), adding even more latency.

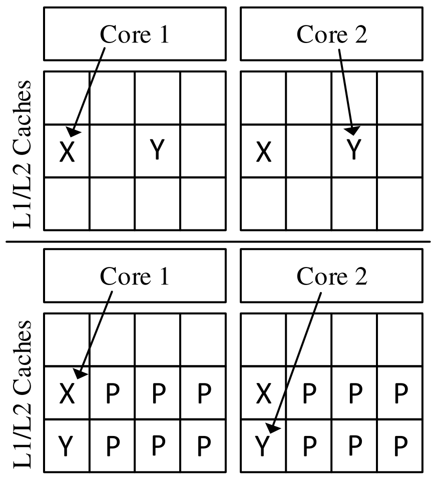

Figure 2: Top: variables X and Y sharing the same cache line. Bottom: cache lines padded (represented as P) to avoid false sharing.

Although false sharing is often overlooked, it can significantly and silently degrade the performance of concurrent code. The simplest example is a thread incrementing a long variable in a loop. Running two of these threads in parallel on a multi-core system should in theory yield the same performance figures as running a single thread, since each one should run on its own core and increment its own independent variable. In reality, when both variables share the same cache line, the threads will need more than twice the time to complete, and this ratio will increase as we add more threads [8]. In order to solve this issue, the variables which are the most susceptible to write contention can be padded in order to fill the full cache line (Fig. 2). The padding forces the variables to be placed in different cache lines.

DISRUPTOR ARCHITECTURE

At the heart of the Disruptor is the ring buffer (Fig. 3). This data structure is used to pass messages between producers and consumers. It has a bounded size, in order to apply backpressure if the consumers are not able to keep up with the producers.

It is backed by an array, which is initialised up front in order to be fully allocated in contiguous blocks of memory. The array structure and the contiguous memory make it cache friendly because the CPU will be able to detect that a consumer is walking through the memory in a predictable pattern and will prefetch the memory into its caches.

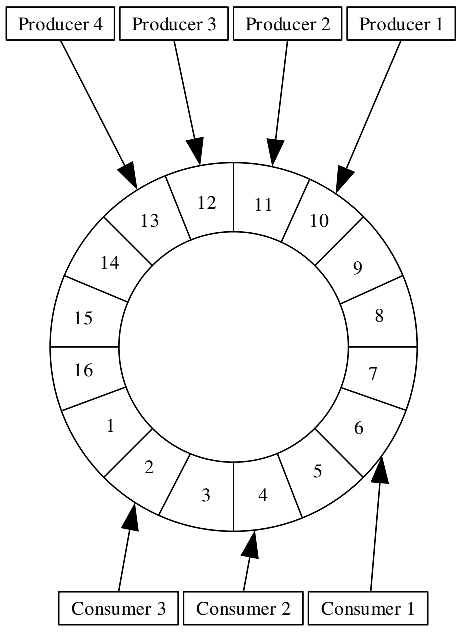

Figure 3: Ring Buffer

The elements in the ring buffer are mutable. When the producers reach the end of the buffer and wrap to the start, they reuse existing entries and overwrite them. This means that this data structure does not generate any work for the garbage collector.

ISBN 978-3-95450-148-9

The progress of producers and consumers on the ring buffer is tracked by sequence numbers. A sequence number is a 64 bit long number, which is padded to fill a full cache line. Since these variables are frequently read and updated from concurrent threads, they could be a source of contention. The padding removes the risk of false sharing.

Memory Visibility

The memory visibility of the data exchanged between producers and consumers relies on the sequence numbers. Updating a sequence number is similar to writing a Java *AtomicLong*: it is translated into a *compareAndSwap* CPU instruction. The CPU memory model guarantees that the memory modified before a *compareAndSwap* will be visible by other threads (this is true for the most common CPU architectures, in the other case the compiler will add an additional synchronization) [9]. This allows the Disruptor framework to be lock-free, thus eliminating the main contention point that comes with traditional implementations of queues.

Batching

In most producer/consumer architectures, the producers regularly outperform the consumers. The Disruptor framework offers a smart way to catch up for consumers: if several items are waiting on the queue, consumers can process the full batch of available items at once instead of processing one item at a time. Batching is usually more efficient, especially when it involves I/O.

Consumer Dependencies

For a simple usage, the Disruptor can be used as a queue between producers and consumers. In addition, the API allows to declare a dependency graph between consumers (Fig. 4).

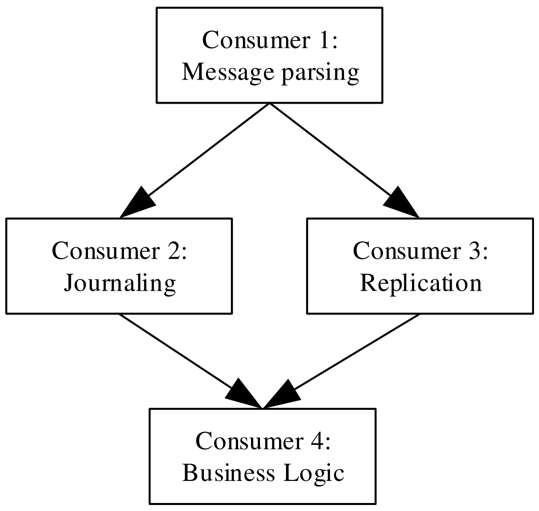

Figure 4: Example of consumer dependency

Real world applications often have several processing layers connected by queues. With this feature, the Disruptor can replace multiple layers of queues. The data will be exchanged between the dependant consumers over the ring buffer, eliminating the contention and delays brought by the traditional queuing approach.

All these concurrent programming concepts are complex and very error prone. They are a common source of errors in real world applications and often difficult to troubleshoot.

The Disruptor architecture actually encourages code simplification by writing the business logic in a single thread. Since the framework takes care of the synchronization, the business logic can be uncluttered of the concurrency concerns. As a result, it is easier to reason about it, as well as to test and maintain it.

DISRUPTOR USAGE IN CESAR

The 1300 devices controlling the beam lines generate multiple streams of data. The CESAR server is in charge of acquiring this data in order to compute an overall state for each device. The first implementation was using manual locking on the data structures in order to synchronize the concurrent data streams. The main reason to use the Disruptor is to simplify the overall device state calculation.

In the new design, the messages coming from the devices are stored on the ring buffer. Then a single Disruptor thread updates simple buffers dedicated to keep the last value for each data stream, and based on these last values computes the overall device state that is published to clients (Fig. 5).

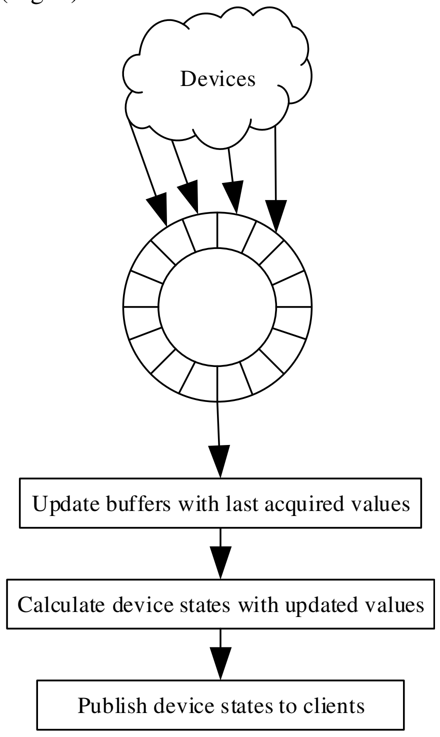
Devices
Update buffers with last acquired values
Calculate device states with updated values
Publish device states to clients

Figure 5: State computation in CESAR

Benefits

This architecture has many benefits. The threads delivering messages from devices never encounter a lock and are released very quickly. If multiple messages are

ISBN 978-3-95450-148-9

available on the ring buffer, they are processed as a batch. This is a good fit for the experimental areas, where most detectors publish their state simultaneously after being triggered by a timing event. The batching mechanism allows CESAR to process such bursts of messages more efficiently.

The state computations are completely deterministic: since a single thread processes messages in a known FIFO order, we can easily understand why a given state was calculated by looking at the message log. When a race condition causes a bug in concurrent code, developers are often left wondering why something that seemed impossible actually happened. This usually occurs at the worst possible times, when the system is heavily loaded and when the applications logs offer little help to understand in which order things really happened. The single threaded design eliminates this class of problems while keeping an excellent throughput.

Since we can rely on the fact that the business logic code runs on a single thread, we can write lock-free code that is more efficient and simpler. This also reduces the overall cost of maintenance, because the business logic code has a high chance of being modified during the software's lifetime.

The main concern when moving to a single thread is to know if it will be fast enough. Considering that the Disruptor was designed for high performance, we speculated that it would easily cover our needs. We indeed measured that our code, without any optimization effort, was able to process around 1 million messages per second. This is more than enough for our current needs and gives us some room to grow, as other Disruptor users reach more than 10 million messages per second with optimized code.

Limitations

During our refactoring we found that integrating the Disruptor in an existing architecture is reasonably simple, and less invasive than other approaches like actor frameworks. There are nonetheless some aspects to consider carefully before using this model.

The ring buffer is designed to apply backpressure. This is usually a good design choice to fail gracefully under load, but one should evaluate if the other parts of the application are compatible with that approach. For instance it might be a business choice to define if messages can be lost when the buffer is full.

As the computations run on a single thread, developers must make sure that this thread is never blocked. Any kind of blocking I/O such as an interaction with a traditional database should be avoided. The most common logging frameworks also make use of locks, which could add contention to this thread and reduce the performance dramatically [10]. After executing the business logic, a common use case is to publish the result of the computation. If this publication is using potentially blocking I/O, such as a remote message broker, the design should handle a network or broker failure without blocking the processing. In our case, we publish the calculated device states over a JMS broker, and decided to add an additional layer of buffering that keeps only the latest device states if the publication cannot keep up. This is acceptable because our GUI only needs the latest updates.

The Disruptor architecture is inherently asynchronous. This is a natural design choice for control applications that handle streams of data. At the same time, an extra effort is necessary if it is required to support synchronous operations as well. In our case, CESAR is required to support a synchronous refresh for the overall device states, based on the current hardware information. In order to create a synchronous functionality based on asynchronous services, we had to carefully analyse the different scenarios that could happen in the asynchronous world (timeouts, concurrent requests, etc.).

CONCLUSION AND OUTLOOK

The new CESAR architecture based on the Disruptor has been used operationally for over a year and proved to be very stable. The simplification brought by the separation of concerns between the concurrency aspects and the business logic allows for easy maintenance and extension of the code base. We believe that a similar architecture can be used for other control systems and can be particularly beneficial for the ones that need to handle large amounts of events with low latency.

A possible area of improvement for CESAR would be to follow the design adopted by the creator of the library [11]: journal and replicate all messages and base the server state on these messages only. This would bring hot-swappable servers and an easy way to reproduce operational scenarios on a developer's machine.

REFERENCES

[1] For example Goldman Sachs collections: https://github.com/goldmansachs/gs-collections
[2] http://www.lmax.com/exchange
[3] https://en.wikipedia.org/wiki/Foreign_exchange_Market
[4] https://en.wikipedia.org/wiki/Multilateral_trading_facility
[5] https://en.wikipedia.org/wiki/Staged_event-driven_architecture
[6] http://www.gotw.ca/publications/concurrency-ddj.htm
[7] http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html Section 2.2.5.4 Data Prefetching
[8] http://mechanical-sympathy.blogspot.ch/2011/07/false-sharing.html
[9] http://www.azulsystems.com/blog/cliff/2010-07-24-unsafe-compareandswap
[10] http://www.grobmeier.de/log4j-2-performance-close-to-insane-20072013.html
[11] http://martinfowler.com/articles/lmax.html

ISBN 978-3-95450-148-9