The Use of Java in Online Event Building and Recording at Jefferson Lab

Carl Timmer, David Abbott, William Gu, Vardan Gyurjyan, Graham Heyes, Edward Jastrzembski, and Bryan Moffit[1]

Abstract—**The CODA data acquisition software at Jefferson Lab uses Java extensively. In particular, we use Java to code the distributed Event Builder and Event Recorder. Although Java is not generally regarded as real-time software, we have taken advantage of several techniques that allow the Event Builder to handle the data rates being produced by the front end- up to 2GB/s. In this paper, we describe these techniques and discuss the relative merits of using the Java language in this context**.

***Index Terms*—Java, event building, event recording, data acquisition, software.**

I. Why Java?

OVER a decade ago we began the design and development of a new CODA version for use with the upgraded detectors of the 12 GeV accelerator at the Thomas Jefferson National Accelerator Facility (JLab). One goal was to develop a highly portable data acquisition (DAQ) suite that would be supported over a wide range of platforms. By far, the most successful language that has good portability is Java, which was deliberately designed for that purpose. It has achieved this by standardizing not only the language, but also the platform environment through its use of the Java Virtual Machine (JVM).

An advantage of Java over its alternative, C++, is the shorter development time due to its memory management, simplicity, and extensive libraries. Looking at the speed of development as a whole, one must consider not only how long it takes to write code, but also how long it takes to find a solution to the problem at hand - usually finding a bug or adding a feature. Java's comprehensibility and maintainability again come out ahead.

Part of the short development time scale is due to the availability of excellent, no-cost tools such as the IntelliJ and Eclipse IDEs. These have debugging built in and they partner well with Oracle's latest profiling tool, Java Mission Control or jmc. Jmc does an excellent job while having minimal effect on a running program.

With this in mind, we chose Java as the primary language for experiment control, inter-process communication, and user interfaces. Initial experience with Java convinced us that the performance was also acceptable as the primary language for major system components such as the Event Builder (EB) and Event Recorder (ER) that involve moving bulk data but very little compute intensive code. The data acquisition group at JLab is small but the use of Java allowed rapid coding progress.

II. Context

Of the four experimental halls at Jefferson Lab, Hall D running the Gluex experiment has the most demanding online DAQ requirements. At peak rate, 1.5 GB/s must be taken from the front end Readout Controllers (ROCs) and eventually stored on disk. The Fig. 1shows the data flow.

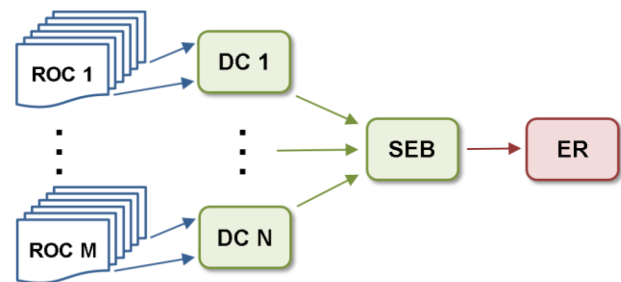

Fig. 1. Online data flow for Hall D. A typical configuration will have over 60 ROCs, 5 Data Concentrators (DC or first level EB), 1 SEB (secondary EB), and one ER.

The data-producing ROCs are running on VME single board computers with a Linux operating system and are programmed in C. The EBs and ER are Java programs running on Linux. We have taken a divide-and-conquer approach by having two-tiered event building. The first level EB is a Data Concentrator (DC) each of which builds events obtained directly from a subset of ROCs. The output of all DCs go into a Secondary Event Builder (SEB). The resulting events are sent to the ER for storage in a file. The network is 40 Gb infiniband.

Using round numbers, with 60 ROCs, each producing 25MB/s of data, the rate into each of 5 DCs will be 300 MB/s. The input into the SEB will be 5 streams of 300MB/s each or 1.5GB/s.

[1] This work was supported by the U.S. Department of Energy under Contract No. DE-AC05-06OR23177.

C. Timmer, D. Abbott, W. Gu, V. Gyurjyan, G. Heyes, E. Jastrzembski, and B. Moffit are with the Thomas Jefferson National Accelerator Facility (Jefferson Lab), MS-10, 12000 Jefferson Ave., Newport News, VA 23606 USA (emails: timmer@jlab.org, abbottd@jlab.org, jgu@jlab.org, gurjyan@jlab.org, heyes@jlab.org, jastrzem@jlab, moffit@jlab.org).

III. EB & ER DESIGN

The EB and ER are part of a more general framework, the Event Management Unit (EMU) within which all components handling the data are created and with the ability to communicate with run control built in. It is simple with three basic components in each EMU. There are input channels that read incoming data, parse it into individual events, and place these events on a queue. A module reads the data from the input queues, processes it, and then writes it to output queues. Finally, output channels take the processed data from the output queues and send it to the next EMU. See Fig. 2 below.

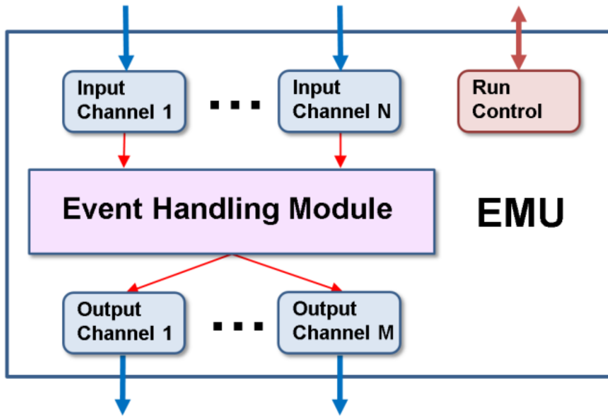

Fig. 2. The Event Management Unit's basic structure.

In CODA, there are different channel types that communicate using different protocols. Most communicate over the network in some fashion while one type in particular writes to files. With communication separated from the event handling logic, the modules become simpler to program. There are two types of modules, one being the EB and the other the ER. The ER module is quite simple since most of the work is done in the communication channels. It is the EB that is the main challenge to program.

IV. JAVA PERFORMANCE FACTORS

Most readers will be familiar with the saying that, "the devil is in the details", and so it was with our Event Builder. One member of our group said that, "The EB is like the bagpipes - easy to build but the devil to tune." After initially writing the EB, its speed was disappointingly slow. This lead to an investigation into finding ways to improve both the performance of specific classes as well as the programming in general.

Profiling tests revealed that the code was spending over 40% of its time putting items on and removing items from the internal queues. We were initially using Java's (highest performing) ArrayBlockingQueue class to store events. These were replaced with ring buffers from a software package called the "Disruptor" programmed by Martin Thompson *et al.* [1] originally for use in high performance financial exchange.

The authors of the Disruptor state that its ring buffer's mean latency in a three-stage pipeline is 1000 times lower than an equivalent queue-based approach. It has less write contention, lower concurrency overhead and is more cache friendly. They made measurements comparing the ArrayBlockingQueue with their ring buffer seen in Table I below.

TABLE I
LATENCY COMPARISON BETWEEN QUEUE AND RING BUFFER

	Array Blocking Queue (ns)	Disruptor (ns)
Min Latency	145	29
Mean Latency	33,000	52
99% less than	2,100,000	128
99.99% less than	4,200,000	8,200
Max Latency	5,100,000	176,000

As the reader can see, the Disruptor's performance is orders of magnitude better than available with Java's built in queues. These performance benefits were achieved through a number of means. Examining these is not only instructive but also acts as a guide in how to efficiently program Java in general.

A. Locks

Locks are critical for providing mutual exclusion and ensuring visibility of change in an orderly way. The problem is that locks, when contended, require a context switch to the kernel which suspends the threads we want operating at peak speed. During this time the kernel may choose to run other tasks which, in turn, may result in loss of cached data and instructions.

A more efficient alternative to using locks is a Compare And Swap (CAS) operation which can be performed on a single word using a single instruction on today's processors. It does not context switch to the kernel but still must use a memory barrier to make changes visible to other threads.

In Table II below from [1], a function which increments a 64-bit counter in a loop 500M times is executed in different conditions that illustrate the effect of locks and CAS operations. The reader can quickly see that locks are best avoided if at all possible even if uncontested.

TABLE II
TIME TO EXECUTE FUNCTION

Method	Time (ms)
Single thread	300
Single thread with volatile write	4,700
Single thread with CAS	5,700
Two threads with CAS	30,000
Single thread with lock	10,000
Two threads with lock	224,000

B. Cache Lines and Memory Access

In modern processors, caches are written to in cache-lines (64 bytes in Linux) for efficiency. If two variables are in the same cache-line and are written to by different threads, we face the identical problem as when writing to the same variable. This is known as "false sharing". Thus, for best performance, independent but concurrently written variables must not be stored in the same cache-line. It is possible to do this in Java with proper technique.

If processors can find a pattern of memory access with a fixed and therefore predictable stride, they will efficiently pre-fetch memory that will soon be used. This works well with arrays, but not generally with structures like linked lists or trees in which nodes are too far and irregularly apart in memory to be pre-fetched. When possible, use arrays.

C. Queues in a Quandary

A queue, though convenient to use, is not ideal for speed. In normal operation a queue provides buffering between a producer and consumers (ignoring multiple producers). If the consumers are faster, the head and tail will be the same leading to contention between the producer and all consumers. If the producer is faster, there will still be contention among the multiple consumers. Since we are using a bounded queue, its size will be updated with each operation, also leading to more contention.

Other inefficiencies include that head, tail, and size often occupy the same cache-line. If a queue is backed by a linked list, it takes locking (and time) for each entry to be inserted into or removed from that list. In addition to each entry having to be allocated, it must have an associated object representing that node created, resulting in a significant load on the garbage collector.

D. Garbage Collection

The single feature that makes Java so easy to use, automatic memory management, is also one that can cause troublesome performance issues. When objects no longer have a reference, the garbage collector reclaims their memory. The larger the number of objects produced, the longer the garbage collector needs to stop the program to perform its duties. Programmers can address this by minimizing the number of objects created or by reusing them.

Since Java works best with either very short-lived or very long-lived objects, code accordingly. In a single, young generation memory block, objects living beyond a short time are copied and moved out of it to an older generation space while those that do not are left as is. Since the objects that are left have no reference, the whole block is reclaimed without having to handle objects individually.

Here, queues have another disadvantage. Under heavy load, they can fill up leading to a reduced rate of consumption. This can result in objects living longer than necessary causing them to be copied to an older generation space. Being collected from that old generation space is an expensive operation and increases the chance of having a “stop-the-world” pause resulting from the need to compact its fragmented memory.

V. Disruptor

As we stated earlier, our solution was to use the Disruptor’s ring buffers that use a very clever design to sidestep all the difficulties queues run into. Each ring is essentially an array of objects each of which is a container for the data of interest. These objects are never added to or removed from the ring. All the memory for a ring is allocated upon creation, making it likely to be laid out contiguously in memory and therefore friendly toward caching strategies and making garbage collection unnecessary.

Access to and bookkeeping of the ring itself is controlled through objects called a SequenceBarrier (see Fig. 3). In the case of one producer per ring (which is what we limited ourselves to), making this separation allows the ring to be accessed free of contention with no locks or CAS operations. Critical parts of the code are written in a manner that eliminates false sharing.

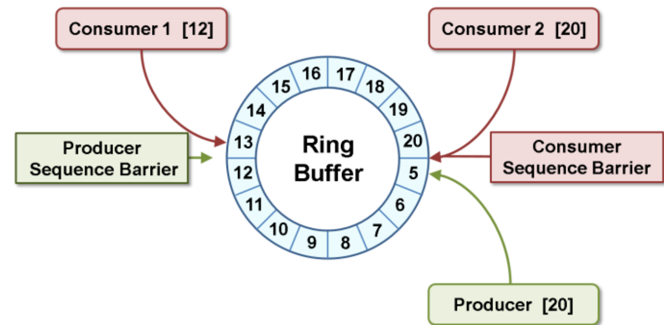

Fig. 3. The Disruptor’s ring buffer and users.

Each producer or consumer tracks its own current position on the ring or “sequence” as it proceeds from one entry to the next. This can be seen by the numbers in brackets in Fig.3. When finished with an entry, the producer makes it available by setting its own sequence using a memory barrier and notifying all consumers. Similarly, it avoids wrapping the ring by monitoring the sequences of consumers. The consumers, on the other hand, ask their SequenceBarrier which sequences are available for consumption. When finished processing an available entry, they set their own sequence with the memory barrier ensuring all changes are visible to the producer.

In Fig. 3 you can see that the producer is currently finished producing entry (sequence =) 20, has claimed the next free entry (5) and when finished, it will update its sequence to 21- making it available to all consumers. The producer’s barrier tracks the sequences of both consumers, allowing the producer to claim up to sequence 12 – the sequence of the slowest consumer. Meanwhile, consumer 2, the fastest, has consumed everything and is waiting for the producer. Consumer 1, however, is slower and is finished with 12, is working on 13 and has up to 20 available. Although not pictured here, the user can create multiple consumer sequence barriers. This allows one to distinguish between different groups of consumers and have them ordered with respect to each other.

An additional benefit of Disruptor design, not available in queues, is that while consumers are waiting for access to a specific sequence, they can be notified of all the sequences available. Access to each of these sequences requires no further involvement of concurrency mechanisms. For example, while consumer 2 is waiting to get the next entry (21), the producer may quickly produce up to entry 28. The consumer barrier returns from the “get” function call with sequence 28 meaning all entries up to and including 28 can be accessed from the ring directly without any bookkeeping. This batching increases throughput while simultaneously reducing latency.

VI. WAIT STRATEGIES

As is often the case, consumers may be faster than the producer. In such circumstances, they must wait for the next available entry. In the Disruptor, each ring is created with one of several possible waiting strategies. The software allows for choosing between: 1) spinning, 2) spinning then yielding, 3) cyclically spinning, yielding, then sleeping, 4) blocking then spinning upon waking up, 5) timing out, and 6) spinning for given time, yielding for a given time, then switching to different strategy (phased backoff). It is also a very simple matter to create another strategy since all source code is available. We created one that first spins for a given number of iterations, then blocks, then spins upon waking up (spin-block). After extensive testing, our spin-block strategy performed the best in our system.

The performance of our system had a huge dependency on which strategy we chose. The strategies that spin or spin-yield, though fast, consumed a tremendous amount of cpu time. In one simulation of 11 ROCs, each sending at 32 MB/s to one DC, spin-yield waiting resulted in the DC using 15.8 cores – most of the cores of an 8-core hyper-threaded machine. Switching to spin-block waiting reduced that to 2.7 cores and it had better performance!

VII. PERFORMANCE

We took a number of steps to improve the performance of the EB. All queues were replaced with ring buffers having a single producer. A good ring buffer wait strategy was selected. Unnecessary use of objects was eliminated. Sections of code needed a continuous supply of ByteBuffers in which to place incoming data over an input channel or to place built events. In light of that, we created a fast object pool of reusable ByteBuffer objects with one-time allocation based on the Disruptor. Locks were removed whenever possible.

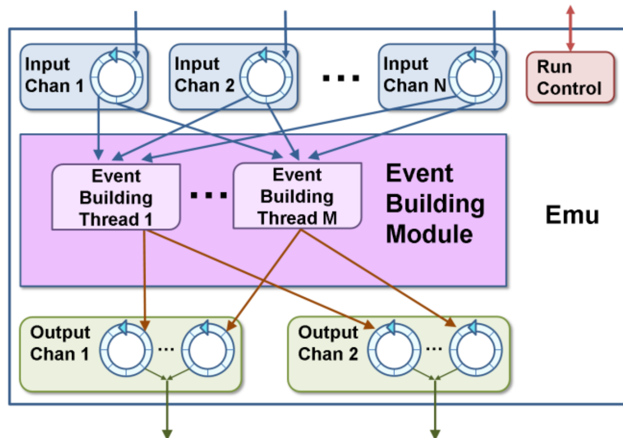

Fig. 4. The Event Builder's basic structure. Circular objects are the ring buffers.

The final configuration of the EB looks like Fig. 4. Each input channel has one ring buffer. Multiple build threads can simultaneously build events, which we found necessary to keep the throughput up at high input rates We also found that one lock was necessary with multiple build threads in order to preserve output event order. Notice that each output channel has one ring per event building thread. This was necessary to ensure only one producer per ring and so to remove contention.

Originally, each input channel ring had 3 consumer barriers. The first was used by an event pre-processing thread, the second by the building thread, and the third by a resource releasing thread designed to keep output events in order. We found that the spin-block waiting strategy worked poorly with more than one barrier, leading to excessive spinning and doubling the cpu usage. Consolidating the pre-processing and build threads and using a lock instead of the last thread, we were able to get better performance while simultaneously reducing cpu usage by over a factor of 2.

Once we made these improvements, we once again profiled the EB. With a configuration of 5 inputs at 300MB/s each, we now spend no measureable time in ring buffer internals and 0.13% of the time in waiting for an empty entry from the input channel ring so it can be filled with parsed events from incoming data. The biggest toll is the 0.47% of the time in the EB's event building method waiting for entries to be read from input channel rings. By any standard, this is a vast improvement over the previous 40%.

In order to assure ourselves that we will be able to handle the data rates of the Gluex experiment, we did two simulations of the experimental conditions. In the first, we approximated the DC with 12 ROC inputs, each producing data at 25 MB/s. It not only handled this, but we were able to push each input up to 100 MB/s with no problems while using 4 cores.

For the second simulation, we had 5 inputs producing at 300 MB/s for a total rate into an SEB of 1.5GB/s. It not only handled this, but we were able to successfully push each input up to 350 MB/s for a total rate of 1.75 GB/s while using less than 6 cores.

At this point, with all the afore mentioned changes made and many bottlenecks removed, the largest consumer of cpu time (40%) is, as it should be, the method used to take raw data from several ROCs and build them into a single event. Our next step is to make this as efficient as possible.

VIII. CONCLUSION

While certainly a challenge, it is possible to program Java applications to handle multiple input streams of data for an aggregate rate approaching 2GB/s.

REFERENCES

[1] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, "Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads" May, 2012. [Online]. Available: http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf