A Data-Synchronous Event Model for GNU Radio

Timothy J. O’Shea
oshea@umd.edu
University of Maryland

Thomas W. Rondeau
tom@trondeau.com
Rondeau Research

Abstract—In this paper, we present a synchronous event stream model overlaid on the existing GNU Radio streaming dataflow model. GNU Radio has long utilized a traditional static streaming dataflow model to interconnect modular signal processing blocks. While this model fits many radio and signal processing applications well, GNU Radio and the applied signal processing community face several pressing needs which require extensions to this model. The event stream model exploits the inherent data parallelism present in many multi-access wireless protocols, greatly enhancing performance on modern concurrent hardware. It also greatly reduces the complexity necessary to implement modern communications protocols and greatly improves code and component re-use by removing the need to co-locate state machines and in-band signaling within primitive GNU Radio blocks. These features ultimately contribute to the agility and rapid development cycle targeted by GNU Radio.

We describe the basic components of the event stream model. Through this discussion, we show how the model works as a system to provide additional opportunities for parallelism, creates simplified complex waveform design, promotes rapidly reconfigurable behavior, and encourages good modular design practices. We discuss a number of critical performance issues and considerations to take into account when the creating of this system. We further describe how it interoperates with many existing GNU Radio subsystems, including existing streaming blocks, hierarchical blocks, stream tags, and asynchronous message handling.

I. INTRODUCTION

GNU Radio has long relied on a fixed dataflow graph for the control flow of data and work functions through a series of primitive signal processing "blocks". At its core, each block has a work function that consumes data from a set of upstream input circular buffers and generates output values in a set of downstream output circular buffers. When a modem can be defined by a constant sequence of work functions operating on uniformly spaced or arbitrary buffer increments this model works well.

The two major areas where problems with this model arise are:

1) Due to increasing wireless bandwidth and diversity requirements, software modems must be able to scale to meet very challenging computational throughput. At the same time, due to the power-driven processor clock speed ceiling and the abundance of available dye space due to geometry reduction, virtually all microprocessors are becoming increasingly parallel at both the instruction level and at the thread level. Therefore, the first challenge is to continue to achieve higher software radio performance by leveraging increasingly concurrent software design.

2) Due to the increasing complexity and rapid adaptation present in modern multi-user channel access techniques and channel rate-matching techniques, modern protocols require increasingly capable adaptive modems implementations. This often means the modem will be required to change modulation techniques, forward error correction techniques, or time-frequency allocations at runtime on the order of microseconds or milliseconds and at very precisely timed time instants which are often negotiated with a remote radio endpoint. The prevalence of these modem techniques leads to the second challenge to support complex modem operations within GNU Radio while maintaining good, modular code re-use and providing tools which help with rapid prototyping.

The event-stream processing model presented in this paper offers a technique which aims to simultaneously address both of these problems over a broad scope of modem technologies.

A. Achieving Performance Through Concurrency

To enable high performance software modems in GNU Radio, we must leverage concurrency from all available sources. These generally consist of the following:

1) Instruction level task parallelism
2) Instruction level data parallelism
3) Thread level task parallelism
4) Thread level data parallelism

GNU Radio currently leverages each of these to varying extents explained below. Our primary focus in this paper is on the last of these: thread level data parallelism. Algorithm optimization is of course a key parameter in high performance software modem design as well, but it is beyond the scope of this paper as it is generally domain specific.

1) Instruction Level Task Parallelism: This level of concurrency refers to multiple-issue architectures, which can execute a number of instructions simultaneously such as load/store operations and an arithmetic or logic operation. It is typically handled below the scope of GNU Radio. On a modern superscalar architecture, instruction and micro-op scheduling occurs in hardware and can generally be achieved without doing anything special. Use of a good compiler can generally ensure that instructions will keep each micro-op pipeline as full as
possible given the underlying algorithm. In extreme cases, kernels can be hand-tuned to ensure optimal operation, but in most cases going to this level of optimization is not desirable as the gain is often not worth the time invested. In the case of non-super-scalar architectures which are multiple issue (such as very long instruction word, or VLIW, architectures), this responsibility rests solely on the compiler.

2) *Instruction Level Data Parallelism:* This aspect refers to the use of Single Instruction Multiple Data (SIMD) instructions provided by many processors. On x86 and x86-64 architectures, this consists of Streaming SIMD Extensions (SSE) and a number of similar subsequent extensions. On ARM processors, these are the NEON extensions. Compilers provide varying levels of support for automated SIMD pluralization of loops, but they often miss the desired mark in critical signal processing inner loops. Since GNU Radio strives to leverage low-level native SIMD performance and also maintain architecture portability, a library called the Vector Optimized Library of Kernels, or VOLK [1], was introduced to allow for GNU Radio blocks to call kernel functions identified by their well-defined operations. The VOLK library then selects the fastest native SIMD implementation available to execute, achieving both design goals. With this abstraction in place, GNU Radio is well positioned to leverage this form of parallelism.

3) *Thread Level Task Parallelism:* Since a modem typically consists of a number of different underlying algorithms, such as framing data PDUs, channel encoding them, mapping to symbols, up-converting, filtering, etc, one of the most obvious things to do is to divide these well-defined, independent tasks into a separate threads of execution, which may be executed across any one of the available hardware cores.

To address this issue, in 2008 Eric Blossom introduced the Thread Per Block (TPB) scheduler to GNU Radio. This scheduler allows each block in a flowgraph to run in an independent thread, synchronizing only around the input and output buffers with adjacent-block threads. This effort resulted in instant performance improvements to new and existing GNU Radio applications. Given that $N$ blocks are present, each block does $1/N$ of the work, the host machine contains $N$ cores, and various other resource contention issues are not large factors, we should expect to immediately achieve a direct $N$-fold speedup or throughput increase with virtually no additional work! This was truly a great advancement as it provided speedup and allowed GNU Radio block developers to generally not worry about the complexities of multi-threaded development.

However, there are several problems with this model. First, the processing load is typically not uniformly distributed over all blocks in a flowgraph; often there are one or two heavy lifting blocks such as FEC encoders/decoders or tracking-loop type synchronization blocks. When the load is heavily centralized in any one block, the maximum speedup of the graph and thus the highest available throughput or channel bandwidth achievable is bound by these blocks, which can quickly put an end to any gains achieved by pipeline parallelism. For instance given a pipeline of 10 blocks where one block comprises 50% of total CPU time on a 10-core system, the greatest speedup we could expect to achieve using pipeline parallelism is only 2x.

In general, GNU Radio does a good job of achieving low-hanging thread-level task parallelism but runs into issues achieving sufficient concurrency when algorithms are hard to sub-divide into separate independent blocks comprising their sub-tasks.

4) *Thread Level Data Parallelism:* In information retrieval and many other areas, macro-data parallelism is an extremely obvious source of concurrency. A common example of thread level data parallelism involves a web-server where two clients may issue two separate documents or search queries simultaneously. The clear thing to do in this case is to distribute each of these queries to a worker thread to service. These threads can independently execute each query or return the requested document to each client, and unless the clients are changing the state of the web-server or an associated database, the same task can execute simultaneously for each user on independent data.

This macro-data parallelism can be thought of at several levels. Inter-modem data parallelism is fairly easy to achieve in GNU Radio. For instance, a base station may transmit or receive on several different frequencies, so each of the independent modems instantiated in software may run its own set of threads for each. However, real challenges come when looking at intra-modem data parallelism. Specifically, if we look at the algorithmic data dependencies in a TDMA or OFDMA system, we see gross data parallelism. Given that a transmitter knows the next 10 PDUs he wishes to transmit, he can generally modulate each of these independently and concurrently rather than serially. Likewise a receiver which has 10 receive bursts buffered, can independently synchronize and receive each of these bursts concurrently rather than serially. The problem is that in GNU Radio, there is no straightforward way to do this, which is where the event-stream model begins.

In the transmitter case, in general, we wish to convert a sequence of logical events such as “transmit this PDU” into samples in the outgoing sample stream to the DAC. In the receiver case, we wish to perform the inverse operation and convert the flow of information from the ADC sample stream into a sequence of logical software events.

Figure 1 illustrates how these basic primitive operations fit into an existing GNU Radio transceiver.

Given that each of these events is independent, it may now be serviced by event-handlers concurrently. The goal of this event-stream model is to provide a set of primitive tools to assist in the translation between events and continuous streams of information, such that it can be easily and rapidly leveraged for a variety of tasks in GNU Radio.

**B. Implementing Complex Multi-Access and Adaptive Techniques**

The other driving motivation for the introduction of this event-stream model into GNU Radio is that of complex multi-
access and adaptation techniques widely used in modern day protocols. GNU Radio has long been effective at implementing static modems well. In general, a modem in GNU Radio instantiates a set of blocks, connects them together and then starts running. Several years ago the ability to support runtime flowgraph modifications was introduced.

In theory, the runtime reconfiguration should allow a developer to, for instance, disconnect a QPSK mapper block from a graph and switch it to a BPSK mapper at run time. However, in practice, the calls to lock() or stop() which enable the safe modifications of a GNU Radio flowgraph (they pause the execution of block worker threads behind the scenes) are asynchronous! This means that we simply do not know how many samples will have progressed through each block when we call it. The timing precision over which we can stop and adapt may be on the order of many thousands or millions of samples. Since most wireless protocols require precise timing as to the instant adaptation or multi-user access occurs, this timing uncertainty is clearly too high for this to be an acceptable solution for such rapid adaptation.

To address this need for rapid adaptation, we again introduce the concept of the event-stream model. In the most general form, to achieve well-timed transitions in our modem flowgraph state, we simply need to execute different processes, or handlers, such as a transmitterQPSK over time \((t_0, \Delta t_0)\) and transmitterBPSK over time \((t_1, \Delta t_1)\). In this case, we can simply schedule events across orthogonal time intervals with our transmit messages to be sent in each modulation. The job of the event-stream system will then be to execute these tasks concurrently and ensure that the samples are serialized into the output information stream. The reverse shall be true in the radio receiver, where the event-stream system shall extract the serial events and provide them to the appropriate handler for each time region \((t_i, \Delta t_i)\). Using these simple primitives to produce and consume information streams it is now possible to build complex, precisely timed multi-mode radios using existing GNU Radio components.

II. PRIMITIVES BLOCKS IN THE EVENT-STREAM SUBSYSTEM

The event-stream system, at its core, provides two simple GNU Radio blocks which provide translation between the traditional GNU Radio stream processing model and the new data-synchronous event processing model, where the events are synchronously timed over a specific region \((t_i, \Delta t_i)\).

- The Event-Stream Source block, which produces a sample stream and executes a series of events defined by a list of triplets \((t_i, \Delta t_i, h_j)\) specifying the associated handler function \(h_j\), and the time region over which samples will be produced by the handler during this event instance \((t_i, \Delta t_i)\).
- The Event-Stream Sink block, which consumes a sample stream and executes a series of events defined by a list of triplets \((t_i, \Delta t_i, h_j)\) specifying the associated handler function \(h_j\), and the time region over which samples will be consumed by the handler during this event instance \((t_i, \Delta t_i)\).

These blocks have an associated event queue that defines the current list of event triplets. Typically, an insertion sort is performed to order the triplets by their \(t_i\) to allow for time-ordered event assumptions in our source and sink blocks.

Similar to GNU Radio’s validation of block connections and item-sizes, queues maintain a list of allowed event types, \(e_k\). They then contain a list of bindings which map event types to event handlers \(e_k \rightarrow h_j\). This allows for a layer of abstraction between event types and handlers as is commonly used in many event models such as Qt [2]. Typically an event consisting of \((t_i, \Delta t_i, e_k)\) will be handed to the queue, and the corresponding \((t_i, \Delta t_i, h_j)\) triplets will be represented in the underlying queue. This gives us an extra layer of abstraction by allowing the appropriate handler to generate and handle re-usable event types depending on the modem.

It should be noted that when using an Event-Stream source block, some form of back-pressure should be applied to limit its production rate. Typically, this should take the form of the DAC clock on either the USRP [3] Sink or Audio Sink, or a throttle block if running fully simulated. Additionally, since the source operates at a distance defined by full buffers between the source block and the block providing backpressure, if lower latency to this sink is desired, the maximum buffers sizes for these intermediate buffers should be reduced.

III. EVENT TYPES

Correspondingly there are three general types of events which may be inserted into the queue:

- Source Events, which expect their handlers to generate information into the buffer provided over a specific time window.
- Sink Events, which expect their handlers to consume information from the buffer provided over a specific time window.
- Free Standing Events, which neither consume data from a stream buffer window or produce data to a stream buffer window but may occur at a specific instance, \(t_i\), as to provide a convenient mechanism for specifying the order of execution of events.

Under the hood, these events are represented using the GNU Radio Polymorphic Type (PMT) and may contain any number of additional parameters in the form of a dictionary.
Event-Stream Source and Sink blocks simply execute a list of events, marrying up the event-stream-window-handler pairings with the associated handler function. However, until now, we have not discussed how these event triplets get into the event queue to begin with. They are placed there either manually through an insertion call or through one of the classes of triggers, which insert events into the queue. These triggers include:

- **Upstream Triggers** - GNU Radio blocks or event handlers upstream of the source or sink block which detect that a new event has occurred. One common example of this might be a matched filter block correlating with a preamble at the beginning of a transmission, which would insert an event triggering the receive handler for that transmission at the stream-time that it occurs.

- **Downstream Triggers** - GNU Radio blocks or event handlers downstream of the source or sink block which detect that a new event has occurred. One common example of this might be if a short transmission already received by a handler specifies the presence of another transmission. In this case, it may reach back upstream to insert the event into the event queue.

- **Side Triggers** - Side triggers are essentially any kind of trigger which occurs outside the scope of the flowgraph or event subsystem of the immediate transmit or receive pathway. This could be anything from a higher layer software event to transmit a burst of voice or data to a radio response which may schedule an acknowledgment event after receiving a message. Through a side trigger, the existing asynchronous messaging interface could directly schedule events or interface with a slot scheduler appropriate for a specific wireless access technique.

The use of these trigger classes is illustrated in Figure 2.

### Common Triggers

Several common triggers are provided with the event-stream module which may be commonly used in a wide range of applications. These consist of:

- A Rising-Edge Threshold Trigger, which inserts an event when a certain threshold has been hit. This can be used with an external correlator or averaging blocks to provide flexibility to detect events in streams without implementing any new primitive GNU Radio blocks.

- A Sample-Timer Trigger, which inserts an event periodically every $N$th sample with the precision of the DAC clock, where the samples are scheduled exactly at a sample periodicity of $N$ and typically at some forward insertion distance.

- A System-Trigger, which sits disconnected to any flowgraph and simply inserts an event every $M$ seconds at the earliest available sample. Since GNU Radio buffer consumption rates across all blocks are not typically uniform, this will jitter around the periodicity $M$ depending on the buffer pointers at each insertion instant. For this reason, the use of a sample-timer trigger is usually desirable.

- A Keyboard Input Trigger has been provided for demonstration purposes that sits outside the flowgraph and inserts key events on a keyboard pressed event. The key pressed is tagged as an event parameter, and in the DTMF demonstration, the event handler maps this key to frequencies. This demonstration is illustrated in Figure 3.
C. Buffer Consideration for Downstream Triggers

Downstream triggers also carry with them some important considerations. Since a running sink event may decide to re-schedule itself, or schedule another event very near its own time window, the event scheduler keep at list of “Live” event times, which is separate from the list of triples in the event queue. This list essentially specifies the time windows over the events currently being handled in concurrent threads. Given this list of Live-times, the scheduler can ensure that as it consumes samples from the incoming sample-stream. It will pause until the event dies and it is sure that it will not need the time-region it operated on again. It can then consume the next Live-event. It should also be noted for downstream triggers of a sink block, such as the Sampler-Timer Trigger, the forward insertion distance should be set such that it is greater than the length of the intermediate buffers.

V. IMPLEMENTING EVENT HANDLERS

The heart of the event-stream system lies in the event handlers. When an event is scheduled, it is the specific handler which is run on the corresponding buffer window which produces useful work. So we need to have both very high performance and very convenient to use handlers, and we need to consider the impact of each upon module re-use.

A. C++ Handlers

At its most basic form, a handler is a C++ class which implements the function:

\[ \text{handler(pmt\_t: msg, gr\_vector\_void\_star: buf)} \]

This function contains a reference to the event PMT which contains additional details about the event (including the maximum length) in the form of a dictionary \((\text{msg})\), and a reference to the buffer data itself \((\text{buf})\). Typically, a sink event handler function will read from the buffer, perhaps parametrized by values in \text{msg} and then take some action, and a source event handler function will write some data to the buffer based on the contents of \text{msg}. This function prototype has been designed to loosely mirror that of the \text{gr\_block::work()} function.

This is the most basic, but least convenient way to implement an event handler. Since it is simply a C++ routine, all the work within it must typically be done manually or by using various underlying C++ algorithm implementations from this function. For this reason, this is perhaps the poorest choice for re-use as it does not allow the use of \text{gr\_block’s} to compute output. It is however the lowest overhead and allows for the most efficient handlers.

B. Python Flowgraph Handlers

To promote code re-use and to leverage the existing stream and block conventions within GNU Radio, we have implemented a specific handler called a Python flowgraph_handler for convenience. This handler allows for the definition of new handler types without ever writing a line of C++. Essentially, this allows the developer to create a normal streaming flowgraph out of existing GNU Radio blocks that will be executed only on the data within the event’s time window. This greatly improves re-use by leveraging existing blocks and by encouraging the continued development of new blocks into the existing application programming interface (API).

A Python flowgraph handler is defined by passing a Python function to the class during creation. Additional Python function handles called \text{pre\_hook} and \text{post\_hook} may be passed that allow for simple Python routine-oriented code to manipulate the flowgraph before and after execution. It should be noted that a factory function is passed instead of a hierarchical block because the handler maintains a pool of instantiated flowgraphs under the hood. Since our goal here is concurrency, and ownership of a flowgraph must be exclusive at any one time, we instantiate a pool big enough such that we can achieve the desired level of concurrency for each handler function.

C. GRC and the Hierarchical Event Handler

Lastly, we extend the Python flowgraph handler all the way to the GNU Radio Companion (GRC) graphical user interface (GUI) for building GNU Radio flowgraphs. By doing so, the developer is now able to define a hierarchical event by graphically dropping in GNU Radio blocks, making connections, defining the input/output signature of the event, and then optionally defining \text{pre} and/or \text{post} hooks to be run before and/or after the flowgraph execution. This is similar to the existing concept of a GNU Radio hierarchical block, except that a finite well controlled window of event data will be passed through the underlying flowgraph instead of an unbounded stream. Likewise the hierarchical event must be scheduled by some trigger in an event-stream source or sink block’s queue instead of being connected directly to a flowgraph. This model allows for rapid development of precisely timed function in a highly optimized manner as has never been possible before in GNU Radio. This also allows for clean abstraction of underlying events as a way to help deal with modern complexity. The dual sinusoide generating event used in the DTMF example shown using GRC in Figure 4.
We can either schedule both events immediately or wait for a transmission estimate for the time of fact. We can easily divide and conquer time series into chunks quickly we can insert new events into the system. There are a number of sink blocks, the bottleneck in the system is simply how only by the width of the buffer range available to the source A. Speculative Event Scheduling

sions which must be taken into account in the Event-Stream subsystem. There are a number of additional performance considerations which must be taken into account in the Event-Stream subsystem.

A. Speculative Event Scheduling

Since we are now processing events concurrently, bounded only by the width of the buffer range available to the source and sink blocks, the bottleneck in the system is simply how quickly we can insert new events into the system. There are a number of optimization which can now be made to exploit this fact. We can easily divide and conquer time series into chunks for detection tasks, or we can speculate as to the occurrence of future events or event parameters based on expected values and locations. This is particularly useful when we wish to exploit some periodic structure and believe that we have fairly stable timing estimates without significant clock drift.

There is a great new trade space between concurrent processing performance by relaxing data-dependencies and that of minimizing the serial work load by leveraging many best-estimate data dependencies. To add to this trade-space, since events may be scheduled both forwards and backwards in time within some range, incorrect speculative events can often be corrected after the fact with a new event with no loss of information. For instance, given a periodic transmission with an estimate of the time of $transmission_1$, we now have an estimate for the time of $transmission_1$ and $transmission_2$. We can either schedule both events immediately or wait for a better estimate after the reception of $transmission_1$, thereby introducing a new data-dependence and reducing available concurrency. But more interestingly with this new model, we can simply schedule both speculatively, and then in case of failure in $transmission_2$, re-introduce the data dependency on $transmission_1$ and insert the event again with a better seed estimate. This sort of operation can achieve extremely high receive throughput or latency on average by making otherwise unsafe assumptions while still allowing for outlier compensation with no loss of fidelity by paying some extra compute or data dependency cost during these low-likelihood miss cases;

B. Minimizing Coordination in the Arbiter

While the goal of developing any highly concurrent system generally should be to minimize coordinating communications as much as possible, sometimes it is unavoidable. An interface called the arbiter is provided which represents a shared memory space between events in which they can keep track of shared state of the modem. A reference to this is passed via the event $msg$ dictionary and any accesses to it that involve writing must be handled with extreme caution to thread safety. Several ideas have been discussed surrounding this, including adding hierarchical tree-locking mechanisms or making the tree update operation inherently atomic and thread safe.

C. Acceleration Thread Coordination

A significant amount of high rate thread coordination must occur to dispatch event+handler tasks to worker threads to execute. In order to accomplish this, we use a variety of optimizations to try to accelerate this critical region. We make heavy use of lock-free queues [4] and atomic variables to avoid the penalties of traditional lock based thread synchronization. This has produced significant gains in experiments with highly concurrent systems and applications.

VIII. Conclusion

By introducing the event-stream subsystem to GNU Radio, we have explained how we are able to greatly increase the amount of available task level data parallelism as well as enable a model which should encompass a broad range of applications requiring precisely timed coordinated events. This capability greatly expands the scope of GNU Radio, it helps to avoid the monolithic frame/synchronizer block that has existed in the community for some time, and it equips GNU Radio to be able to rapidly prototype and implement a number of complex multi-user access techniques and adaptive behaviors as used in many modern wireless protocols.

IX. Future Work

A. Stream Tag to Event Parameter Translation

Stream Tags [5] are the mechanism in GNU Radio for carrying annotations along with normal sample streams. Typically they are low update rate and propagated downstream sample-data-synchronously. Recently, the USRP has started tagging such statistics as receive time and receive frequency onto
its graphs, and ideally the Event-Stream sink block should be able to translate these stream-tags into event parameters. Since both utilize the polymorphic types (PMTs), this should be straightforward in its implementation and will allow for various global and time sensitive properties such as GPS time and RSSI to be propagated effortlessly between flowgraph and event domains.

Stream Tags have also been used as an alternative mechanism to provide a form of synchronous burst tagging. However in comparison to this model, only events from upstream triggers are generally capable of triggering events without specialized interfaces. Likewise, the rather specialized form of both the tagging blocks and the downstream receiving blocks may lead to many high application specific tagging systems with little ability to re-use and interoperate components between waveforms. Lastly the job of extracting a time region given the time tags is extremely tedious and error prone. Given that this is a well defined and commonly used task, we prefer a block dedicated to this purpose which has been both well tested and well optimized so that it need not be re-implemented for every special purpose event-block.

B. Heap Management Optimizations

Since events are defined using polymorphic types (PMTs), which allow loose typing and access to event parameters from both C++ and Python, additional considerations should be made as to the performance thereof. Recently, capabilities have been introduced into certain GNU Radio branches [6] to support different PMT allocators. Since events occur at a rate corresponding to events in the underlying information stream which may vary in arrival time from the order of seconds to microseconds, we need to carefully consider the data structures we use for performance. Using the concept of a pmt-manager, we can alleviate the penalty of allocating and deallocating heap space through a managed pool and still allow for highly variable and unstructured data-dependent sequences of events.

C. Concurrent Memory Access Optimizations

Lastly, as processors develop more and more cores, non-uniform memory architectures tend to become increasingly important to performance. Because GNU Radio and Event-Stream generally assume a single large global memory space, this model raises some concerns about memory bandwidth and bus contention when scaling to highly concurrent systems. Looking forward, it may be greatly beneficial to ensure certain flowgraph handlers, managed PMT pools, flowgraph buffers, and hardware-software core affinities be set to help try to segment independent execution areas into different regions such as allowed by NUMA (non-uniform memory access) machines. Looking towards many-core systems, this may provide significant improvement and gains from rather general optimizations within the GNU Radio architecture.

REFERENCES


