Concurrent Data Structures in Software for SDRs

Kalyani Gadgil

Abstract

Keywords: Software Radios, Concurrent Data Structures, Buffers, Multithreading

1. Introduction

The Cognitive Radio Test System (CRTS) provides a flexible software framework for over the air test and evaluation of cognitive radio (CR) networks like the Cognitive Radio Network Testbed (CORNET) [1]. The project goals are to provide a framework to analyze and compare performance of different cognitive engines under noise, interference, fading and other effects. [2]. We are in the process of overhauling the old software to adapt to new radios, high-performance computers and a centralized allocation system similar to the ones used on high-performance clusters like ARC machines.

1.1. Objective

I am investigating existing software frameworks to help me understand design parameters required for software-defined radios (SDR). Software like GNURadio provide precompiled blocks that model signal processing components like resamplers, channelizers, fourier analysis etc. Blocks are arranged in a flowgraph to design, simulate and deploy real-world radio systems [3].

The objective is develop a scheduling scheme derived from the Thread Per Block(TPB) scheme of GNURadio. [4, 5, 6, 7].The TPB scheduler allows each block in a flowgraph to run in an independent thread, synchronizing only the buffers that connected adjacent-block threads [8]. CRTS incorporates the concepts of interconnected filters (similar to data-processing blocks in GNURadio) with a user-defined multithreading scheme. Users would decide which filter or set of filters run on which thread. This scheme adds significant design complexity for the user. In an attempt to improve this scheme, I studied the internals of the TPB scheduler and hypothesize that a change

in the concurrent data structures used between adjacent-block threads could improve performance.

In order to parallelize computations on mutlicore processors efficiently, the TPB scheduler starts one thread per block in a GNURadio flowgraph, in effect, distributing the load. Studying the internals of the TPB scheduler revealed that it consists of circular buffers that communicate between individual threads containing blocks. These circular buffers store data produced by the thread and when a certain capacity is reached, the consumer thread is woken up and the two threads can start processing concurrently.

A study of this buffer led to the observation that it consists of mutexes that lock a consumer out when a producer is adding data to the buffer. While lock-based buffers provide mutual exclusion and deadlock freedom, they suffer from performance issues in terms of Amdahl's law. In ring buffers or "magic" buffers as they are sometimes called, memory allocation takes place as follows : a page of memory is mapped to the process address space twice back-to-back in order to give us a contiguous chunk of memory that emulates a ring buffer.

The two-lock concurrent queue proposed by Michael and Scott Michael and Scott [9] could provide better performance than a lock-based ring buffer. Furthermore, a lockfree buffer could provide even better performance. This project tests out this hypothesis with a comparison of performance between ring buffer, partial queue and lockfree queue. Since this is a proof-of-concept, I test the data structures with artificial inputs. As a first step, I consider that an audio source block emits a stream of floats and so a stream of floats would make a good test.

C++ does not provide garbage collection and programmers need to take care of the memory that has been allocated. There are several solutions to memory reclamation in C++, here we have tried out reference counts to see if there is an improvement in the memory footprint.

2. Limitations and Opportunities

GNURadio's system is designed such that each buffer is connected to two threads of execution, each thread running one filter. One of these filter-thread pairs is the producer while the other is the consumer. GNURadio's circular buffer requires external control over samples being injected into it. Each filter block executes a certain processing function like sampling, encoding, tranforming data. Since these digital signal processing functions have varying

rates of operation, the GNURadio scheduler takes control of managing the flow of data through the buffers. Producers notify the scheduler the rate and amount at which they generate samples. Consumers notify the scheduler the rate and amount at which they will consume the samples. These values are coordinated through the filter block interface so that the data structure between threads does not explicitly manage input and output rates. Once the consumer has pulled data out of the circular buffer, the producer filter is notified that nbytes of space has been created for more samples to be pushed in. The chunking of data being pushed and pulled out of the queue is said to give better throughput at the cost of varying workload sizes GNU Radio Website [4].

CRTS implements the same circular buffer mechanism but leaves the thread allocation per filter block to the user. A scheduler has not been implemented yet because further study into efficient concurrent data structures needed to support applications for specialized digital signal processing have yet to be investigated.

In an effort to start this investigation, I look into the circular buffer implementation which is inherently single-producer, single-consumer. I implement lock-based and lockfree approaches to a queue which perform the same function as the circular buffer in that, both are first-in first-out.

3. Experimental Setup

I adapted the circular buffer code from GNURadio for testing purposes. For comparison, I implemented a two-lock blocking partial queue as described by Michael and Scott Michael and Scott [9], a single-producer single-consumer lockfree queue and a multi-producer multi-consumer lockfree queue with reference pointers. The current circular buffer code in CRTS closely mimics that of the GNURadio code so testing the one would be equivalent to testing the other.

Assuming an audio source block transmitting audio samples or a sound card outputing samples to a speaker. Most of these values would be floats or doubles with only complex radio applications requiring complex types. For the sake of experimentation, we include performance measurements for two types of data, integer and floating-point values.

Sequential time (sec)	Parallel time (sec)	Speedup
2.73E-04	2.36E-04	1.16E+00
6.08E-04	6.38E-04	9.53E-01
4.37E-04	2.25E-04	1.94E+00
	Average	1.35E+00

Table 1: Speedup for Circular Buffer

Sequential time (sec)	Parallel time (sec)	Speedup
6.26E-04	5.76E-04	1.09E+00
1.14E-03	1.04E-03	1.10E+00
1.07E-03	2.68E-03	3.98E-01
	Average	8.61E-01

Table 2: Speedup for two-lock bounded queue

4. Results

4.1. Speedup

I measured speedup of each of the data structures in terms of ratio of sequential time (time taken to execute producer and consumer operations separately) and parallel time (time taken to execute producer and consumer operations concurrently).

The results show that the lockfree single-producer single-consumer code has similar speedup to the circular buffer. The lock-based queue does not fare so well.

4.2. Throughput

Throughput is measured for 1024 items being enqueued and dequeued. These items are either integers(4 bytes) or doubles(8 bytes). GNURadio defines polymorphic types like complex(2 doubles) and vectors of types that

Sequential time (sec)	Parallel time (sec)	Speedup
8.42E-04	6.73E-04	1.25E+00
2.68E-03	1.75E-03	1.53E+00
1.97E-03	2.30E-03	8.60E-01
	Average	1.21E+00

Table 3: Speedup for lockfree queue

can be stored as items in the queue. The lockfree queue does not perform as well as the circular buffer for ints but is comparable for doubles.

Trials	Circular Buffer	Two-lock Bounded Queue	Lockfree_spsc
1	4.34E+06	1.78E+06	1.52E+06
2	1.60E+06	9.85E+05	5.85E+05
3	4.54E+06	3.82E+05	4.46E+05
Average	3.50E+06	1.05E+06	8.51E+05

Table 4: Throughput for 1024 integer values

Trials	Circular Buffer	Two-lock Bounded Queue	Lockfree$_spsc$
1	9.56E+05	5.47E+05	1.04E+06
2	2.30E+05	1.91E+05	6.81E+05
3	2.46E+05	2.73E+05	2.71E+05
Average	4.77E+05	3.37E+05	6.65E+05

Table 5: Throughput for 1024 float values

5. C++ memory model

The C++11 Standard introduced a new multithreading-aware memory model. Prior to C++11, multithreaded programs were written with the POSIX threads library written in C. Since C++ follows the object-oriented programming model, it was important to consider how objects could be dealt with in a thread-safe manner. Two threads accessing the same memory location leads to a data race which leads to undefined behaviour when the program is executed. Every object in a C++ program has a defined modification order in that, all threads of a program must write to the object's memory location in a mutually agree upon order. If the writes are not atomic, programmer has to make sure that there is sufficient synchronization between the threads to meet this requirement.

When an atomic operation is conducted by thread, it's effects are seen by other threads instantaneously. A non-atomic operation by one thread doesn't offer this guarantee; another thread might retrieve the initial value or the value stored after a modification. Standard atomic types in C++ do not have copy constructors or copy assignment operators.

C++11 memory model consists of 6 types of memory ordering.

- memory_order_relaxed
- memory_order_consume
- memory_order_acquire
- memory_order_release
- memory_order_acq_rel
- memory_order_seq_cst

These memory orders specify how regular non-atomic memory accesses are ordered around an atomic operation [10].

5.1. Memory model relations

5.1.1. synchronizes-with relation

The synchronizes-with operation can happen only between operations on atomic types. An atomic load and store will be ordered so that their time intervals will never overlap. Precedence is decided by program order and a mutually agree-upon order is imposed between threads.

5.2. happens-before relation

Regardless of threads, evaluation A simply happens before evaluation B if any of the following is true :

1. A is sequenced-before B. This means that within a thread, there is a precedence relationship between A and B such that A preceeds B in program order. $A \rightarrow B$.
2. A inter-thread happens before B. This means that between threads, A and B synchronize with each other directly or through other orderings imposed by the nature of atomic loads/stores.

Most of the code was written with the default memory order, sequential consistency which is defined from the C++11 standard 1.10.21 as "It can be shown that programs that correctly use mutexes and memory_order_seq_cst operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as 'sequential consistency'" [10].

6. Correctness Arguments

6.1. Lock-free Single-Producer Single-Consumer

There is a happens-before relationship between the enqueue and dequeue to ensure that its safe to retrieve the data. [11]. Section:5 describes the happens-before relation.

From the following listing we can see that there are one load and one store operation on the atomic *tail*. These two operations synchronize-with each other as shown in Fig.1. For an empty queue, if a dequeue operation happens before an enqueue, it will return an empty shared-pointer and a new node will attached when enqueue completes. For an empty queue and if an enqueue operation happens before a dequeue, a node will be enqueued and it will subsequently be dequeued. These operations are linearizable around the load and store operations on *tail*.

This code runs specifically for a single producer and single consumer and would give undefined behaviour if used for multiple threads.

```
1  std::atomic<Node*> head;
2  std::atomic<Node*> tail;
3
4  Node* deq_head(){
5          Node* const old_head = head.load();
6          if(old_head == tail.load()){
7                  return nullptr;
8          }
9          head.store(old_head->next);
10         return old_head;
11 }
12 Node* old_head = deq_head();
13         if(!old_head){
14                 return std::shared_ptr<T>();
15         }
16         std::shared_ptr<T> const res(old_head->value);
17         delete old_head;
18         return res;
19 }
20
21 void enq(T new_value){
22         std::shared_ptr<T> new_data (std::make_shared<T>(new_value));
23         Node* p = new Node;
24         Node* const old_tail = tail.load();
25         old_tail->value.swap(new_data);
26         old_tail->next = p;
```

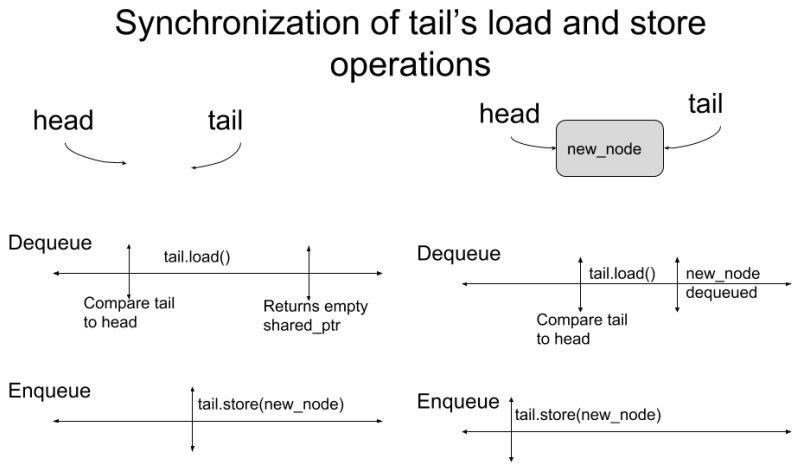

Figure 1: Linearization points for empty Lock-Free Single-Producer Single-Consumer Queue

```
27          tail.store(p);
28  }
```

6.2. Circular Buffer Single-Producer Single-Consumer

A major concern in circular buffers for streaming is whether data will be overwritten if a producer outruns a consumer. This is managed in GNURadio by other scheduler which provides input and output rates to each block. Hence, the data being produced will be at a specific rate and consumed at a specific rate. The circular buffer data structure in itself doesn't need to worry about this except with parameters to indicate how many bytes being pushed into it and how many consumed. GNURadio itself doesnt have clocks but connected devices like USRPs and RTL-SDRs have clocks.

7. Discussion of circular buffers

7.1. Ring buffers on general purpose platforms with MMU

Both DSP algorithms and processor architectures show features which are hardly present in general-purpose computing.Leupers [12]. The Memory

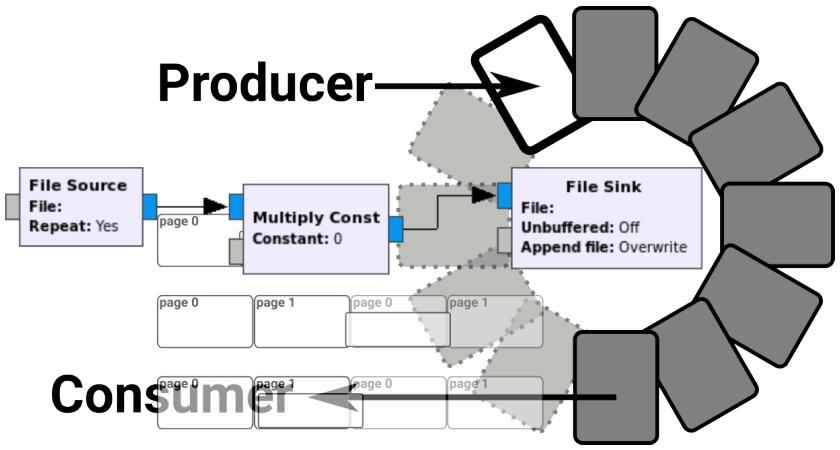

Figure 2: Circular Buffer in GNURadio

Management Unit (MMU) of modern operating systems provides each process it's own address space. Processes cannot share address spaces. Pages are mapped into this space from the linar address space and back. MMU converts physical RAM address ranges to one linear address space which is provided to the operating system. It also translates the pages in the linear address space to the virtual address space of the process and vice-versa. Segmentation faults occur when you try to access a memory address that wasn't mapped to the process's address space.

GNURadio takes advantage of these properties of the MMU to emulate a ring buffer as shown in Fig.2. By mapping the original pages twice, right behind the original, we can emulate the ring buffer.

7.2. GNURadio Circular Buffer

GNURadio doesn't support runtime buffer size determination but allocates a fixed size of 64kB for each buffer. Blocks typically consume all the samples in their input buffers to mitigate the overhead of context switching between threads. However, the ability to control the amount of data consumed in a single block execution can provide throughput/latency tradeoff. [13]

GNURadio use a runtime scheduler. When flowgraph execution starts, the scheduler runs blocks which have sufficient input data to proceed. Figure.2 shows cartoon of how a circular buffer uses one page mapped back to back to generate a circular queue.

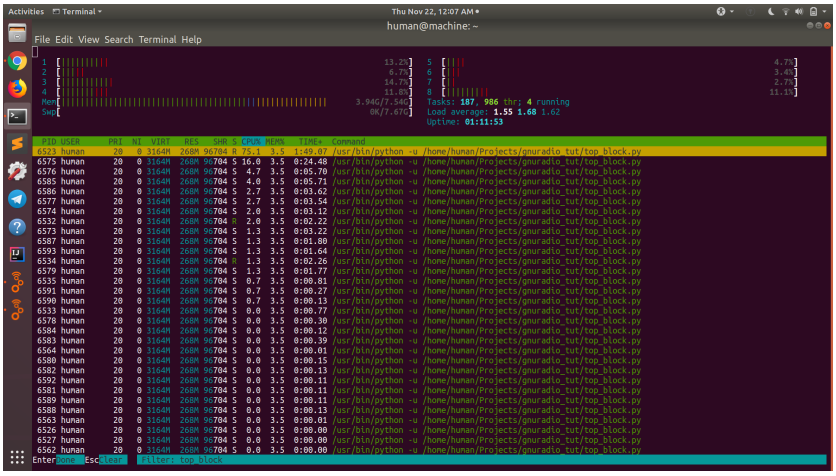

Figure 3: Caption

The GNURadio scheduler runs every block on one thread. This can be seen from the Figure.3 where each of the commands is a python block running on a separate thread.

Fig. 4 shows the performance report of GNURadio Companion while running the FM receiver block shown in Fig.6. These instructions were taken from GR Lists

7.3. Ring Buffers on DSPs with AGUs

A general view of digital signal processing is shown in Fig.5. Input is an analog signal like FM radio input. After A/D conversion and processing, the signal is converted back to an analog signal. A flowgraph of an FM receiver in GNURadio is shown in Fig.6. An important DSP function is that of digital filtering. The rational resampler block shown in Fig.6 contains standard antialiasing filters to resample audio input. Filters, show characteristics common to most DSP algorithms; predominance of arithmetic operations like multiple-accumulate (MAC)computations over control statements and the use of delay lines. Delay lines are often a special data structure. For a given period t, a delay line is characterized by the sequence of signal values;

$$D_x = x(t), x(t-1), x(t-2), ..., x(t-N) \tag{1}$$

where $x(t)$ denotes value of signal at t^{th} time period of an infinite-loop DSP program and $x(t-i), 1 <= i <= N$ represents the value of x from $i -$

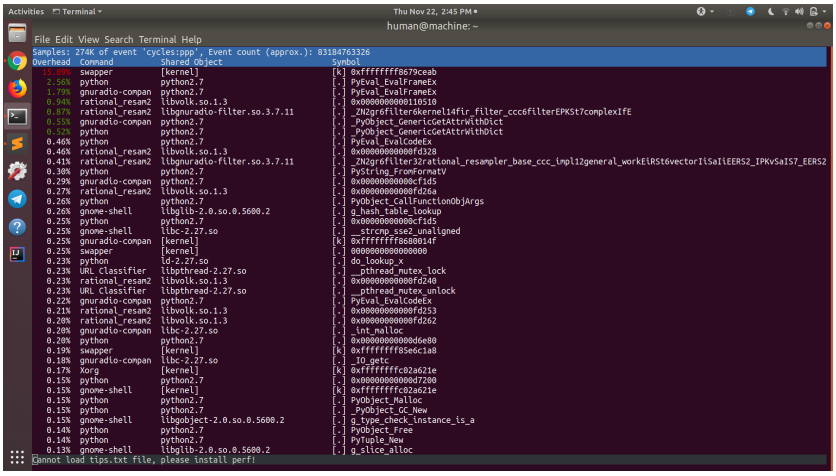

Figure 4: Performance Report using Linux Tools Perf

th previous period relative to t. Delay lines should ideally be placed in contiguous address spaces in memory. Efficient realization of delay lines in software demands for an appropriate layout of data in memory. Delay lines can implemented using fixed mapping of elements with data moves. But this creates read/write overhead. Ring buffers are useful in this aspect because instead of moving values between memory, we update the pointer to the memory. In digital signal processing units, Address Generation Units (AGUs) perform modulo addressing to provide these circular buffers.

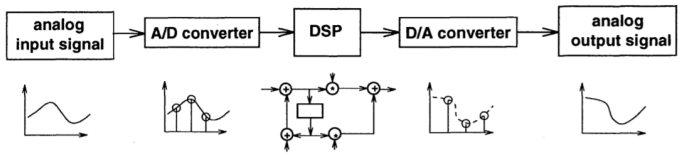

Figure 1.5 General digital signal processing system

Figure 5: [12]

[1] T. R. Newman, S. M. S. Hasan, D. Depoy, T. Bose, J. H. Reed, Designing and deploying a building-wide cognitive radio network testbed, IEEE Communications Magazine 48 (2010) 106–112.

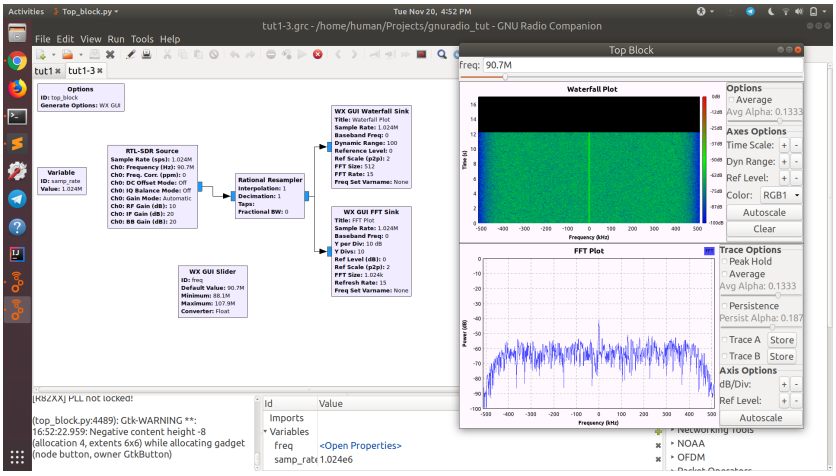

Figure 6: Flowgraph of FM Receiver

[2] E. Sollenberger, F. Romano, C. Dietrich, Test & evaluation of cognitive and dynamic spectrum access radios using the cognitive radio test system, in: Vehicular Technology Conference (VTC Fall), 2015 IEEE 82nd, IEEE, pp. 1–2.

[3] GNU Radio Website, Gnu radio manual and c++ api reference 3.7.13.4, accessed December 2018.

[4] GNU Radio Website, Behind the veil: A peek at gnu radio's buffer architecture, accessed December 2018.

[5] Principal Investigator, Tom Rondeau's Website, Scheduler details, accessed December 2018.

[6] Github commit, runtime: merge scheduler and scheduler_tpb 1278, accessed December 2018.

[7] Marcus Müller, Discuss-gnuradio: questions about the gnuradio scheduler, accessed December 2018.

[8] T. J. OShea, T. W. Rondeau, A data-synchronous event model for gnu radio.

[9] M. M. Michael, M. L. Scott, Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, in: Proceedings of the fifteenth

annual ACM symposium on Principles of distributed computing, ACM, pp. 267–275.

[10] ISO/IEC 14882:2017, Programming languages – C++, Standard, International Organization for Standardization, Geneva, CH, 2017.

[11] A. Williams, C++ concurrency in action, London, 2012.

[12] R. Leupers, Retargetable Code Generation for Digital Signal Processors, Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[13] A. S. Fayez, Design Space Decomposition for Cognitive and Software Defined Radios, Ph.D. thesis, Virginia Tech, 2013.