High Performance OFDM PHY in C++

M.S. Thesis

Thomas Gadek

Supervised by:

Dr. Bruno Adriamanalimanana

May 2022

Master of Science in Computer and Information Sciences

Department of Computer Science

State University of New York Polytechnic Institute
High Performance OFDM PHY in C++, a thesis by Thomas Gadek (U00290793), is approved and recommended for acceptance as a final project in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences, SUNY Polytechnic Institute.

Date

Dr. Bruno Adriamanalimanana

Dr. Jorge Novillo

Dr. Scott Spetka

STUDENT DECLARATION

I declare that this project is my own work and has not been submitted in any form for another degree or diploma at any university or other institute or tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given

__________________________
Thomas Gadek
Abstract

Orthogonal Frequency Division Multiplexing (OFDM) is a popular modulation technique used in some of the most well known waveforms today such as 5G and Wi-Fi. Almost all waveform implementations are currently being performed on hardware and FPGA firmware due to the high performance these techniques allow, with a significant trade off being the time and cost to develop with these methods. It seems that almost all software based research and development is being done using GNU Radio, which while a very quick and easy environment to test with, has nowhere near the performance capabilities of a pure C++ implementation. This work aims to investigate software optimization techniques that can be used in C++ to allow for quick and high performance applications to be created on general purpose processors (GPPs), and set a benchmark for what can be achieved on some common platforms, like a laptop and a PC. The results show that sample rates and bandwidths of well over 1000MHz can be achieved. To the best of my knowledge, the contributions presented in this paper have resulted in the highest performing implementation of a completely software based OFDM PHY in terms of sample rate, bandwidth, and subcarrier count.
Contents

1 Introduction 5

2 Background 9
   2.1 Industry Standard ........................................... 9
   2.2 Hardware vs Software ....................................... 11
   2.3 SDR Development ............................................. 12
   2.4 Waveform Basics ............................................. 13

3 Research 16
   3.1 Literature Review ........................................... 16
   3.2 OFDM .................................................... 20

4 Software Development 27
   4.1 Contributions .............................................. 27
   4.2 Asynchronous Buffer Manager ............................... 28
      4.2.1 Introduction ........................................... 28
      4.2.2 Implementation ....................................... 31
   4.3 CP-OFDM Workers .......................................... 35
      4.3.1 Setup .................................................. 35
      4.3.2 TX Worker ............................................. 36
      4.3.3 RX Worker ............................................. 38
      4.3.4 Other Considerations ................................. 40
   4.4 CP-OFDM Frame Detection ................................... 40
   4.5 OFDM Parameters ............................................ 43
   4.6 Other Implementations ..................................... 46

5 Results 48
   5.1 Signal Analysis ............................................. 48
### List of Figures

2.1 "Trade-off between reconfigurability and development time for FPGA, ASIC, DSP, GPP, and hybrid GPP/FPGA-centric SDR architectures" taken from [23] ........................................... 10

3.1 Diagram of Wi-Fi Subcarriers, courtesy of [17] .................. 20
3.2 QPSK Constellation with Arbitrary Bit Mapping ............... 22
3.3 Diagram of Cyclic Prefix Insertion .............................. 24

4.1 Samsung Galaxy S22 Ultra Chip Set ............................... 31
4.2 Buffer State Diagram ........................................... 33
4.3 Worker Thread Distribution Diagram ............................... 34
4.4 Worker Memory Distribution Diagram ............................. 36
4.5 TX PHY Worker Block Diagram .................................. 38
4.6 TX PHY Worker Block Diagram .................................. 39
4.7 Auto Correlation Optimization .................................. 42

5.1 Spectrum Analyzer Capture ...................................... 49
5.2 Spectrum Analyzer Capture Setup ................................ 50
5.3 Laptop Specs ...................................................... 50
5.4 CP-OFDM PHY Laptop Results .................................... 51
5.5 Custom PC Specs .................................................. 52
5.6 CP-OFDM PHY Custom PC Results ............................... 53
5.7 ABM vs Serial Manager Performance Difference ............. 60
5.8 ABM vs Serial Manager Performance Difference, ARM big.LITTLE Simulation .............................................. 61
Chapter 1

Introduction

Orthogonal Frequency Division Multiplexing (OFDM) is a popular modulation technique used in communications today, two of the most well known applications currently being 5G (and previous generations of the cellular network) along with Wi-Fi. Almost every electronic device these days has the hardware necessary to communicate with other devices using one (or both) of these waveforms. Top of the line cell phones today are equipped with the Snapdragon 8 Gen 1 Mobile Platform, which contains the latest 5G Modem-RF system. This chip set also contains the hardware to support Wi-Fi 6 and 6E. In both cases 5G and Wi-Fi support are directly tied to the chip set (i.e. the hardware), which means our devices will only ever support these versions it was developed for; an upgrade would require buying a new device with a new chip set built to support new versions. In almost all cases it is not possible to swap out or update the chip sets within these devices. Once 6G is released, or Wi-Fi 7, to take advantage of those new generations a new device with hardware built with these new generations in mind will need to be purchased. These waveforms aren’t updated very often, so upgrading isn’t crucial, but this hardware is tied to many more waveforms as well (ex. Bluetooth), meaning that chances are the hardware will be obsolete in some way fairly quickly when one of these waveforms are inevitably updated.

As every commercial implementation of these waveforms is done through hardware or FPGA firmware (that ultimately produces hardware), this thesis will explore what can be accomplished in terms of an OFDM Physical Layer with a software only C++ implementation. Software optimization techniques that can be applied specifically to OFDM signal processing will
be explored and the performance that can be accomplished on a typical General Purpose Processor (GPP) will be demonstrated.

The differences between hardware and software are common knowledge in the computer science community, hardware is superior when it comes to performance, but requires significantly more time and cost to develop. This is driven by the nature of hardware implementations, one component being the inability to make quick changes, or any changes once the hardware is released as a physical object is used to accomplish the task. This is where software based applications shine, software is significantly quicker and cheaper to develop with as there is no direct tie to any physical object; yes a physical processor is needed to execute the software, but an infinite amount of copies can be thrown onto any processor and executed. There is unlimited flexibility as changes can be made on the fly and updates can be pushed to devices just like every other app on a typical phone today. When it comes to waveform development, choosing which route to take (i.e. hardware vs software), all of these factors come into play based on requirements.

While it is known that it is not possible to reach the same performance as existing hardware, quantifying the performance that can be obtained in software is important as it can be used to determine if/which future OFDM based waveforms can be achieved entirely in software. It can also be used to determine what aspects of modes of existing waveforms, like 5G, can be implemented in software, along with the overall reusability and flexibility a software implementation allows. This also allows a GPP to run any number of waveforms that are downloaded onto the system, granted that the processing power is high enough, giving infinite possibilities. Another advantage of software based waveforms is they can be built and installed on the latest GPPs as they are released, meaning that simply waiting a few months could be all that is needed for a performance increase of the waveform, or that next step that was needed for further capabilities. Hardware on the other end, once it is built, that is what you are stuck with. Many years down the line the same old and, by then, very outdated hardware will be the only implementation you have, which could cause issues or difficulty when trying to interface with newer technology. Updating this waveform to the latest hardware would require significantly more work than the software based equivalent, meaning time and money are also saved in the long run.
The contributions made in this work are the techniques that were created and used to accomplish the high performance of an OFDM PHY, specifically CP-OFDM motivated by 5GNR. While many details for the implementation are small, yet still important in reaching high performance, there are three main and significant techniques/contributions that were developed to achieve this high performance.

The first main contribution that will be showcased in this thesis has been named the "Asynchronous Buffer Manager", in reality this consists of one manager on each side of a worker thread distribution. This buffer management strategy allows for the worker threads to be resistant to workload variation, allowing for maximum efficiency of the worker threads. Another significant advantage to this contribution is that the buffer management strategy itself is not designed around OFDM, meaning that it can be applied to any application that is built to use worker threading. Other situations where this management strategy can be exploited will also be explored, specifically ARM big.LITTLE architecture, the primary architecture used in mobile devices. This architecture was specifically chosen as the devices mostly used today that use 5G and Wi-Fi are cell phones, so creating optimization techniques that take into account the layout of this architecture are important as this can motivate future work relating to higher performance software based waveforms on cell phones or other portable devices that use ARM architecture; along with any other software applications that will need to use high performance worker based architectures. The implementation of software based waveforms on low cost battery powered devices could be very attractive in some applications or environments.

The second significant contribution is the CP-OFDM workers themselves. There are two workers that have been developed, one that does the transmitter (TX) end processing, turning data coming in from a higher layer into a CP-OFDM signal, and then the opposite case, the receiver (RX) end where a worker takes in a CP-OFDM signal and processes it back into data. While the worker threaded architecture results in a significant performance boost, ensuring the workers themselves are highly efficient is even more important as performance increases in the order of magnitudes can be achieved, rather than just linear improvements (if even) by throwing more workers in the mix.

The third main contribution is an implementation for CP-OFDM Frame
Detection, used to obtain the framing of the signal on the receiver end. It takes advantage of how frame detection is done on a CP-OFDM signal to minimize the computational load needed to process the signal in software. This technique can be applied to other signals if they fit a certain set of parameters, allowing for software reuse, another major advantage to software based development.

With these contributions, and along with all other software development, an 11th Generation i7 Intel Laptop can, under specific parameters, run as a CP-OFDM transceiver at sample rates of over 400MHz. A custom PC with an i9 Intel GPP is capable of running as a CP-OFDM transceiver at over 1500M samples per second. These results well exceeded the initial goal of reaching the hardware limitations of an Ettus X300 Software Defined Radio (SDR), which has a maximum sample rate of 200MHz, paving the way for future work and research to be done on extremely high end SDRs such as the Motorola NS-1 [3,22].
Chapter 2

Background

2.1 Industry Standard

Waveform development is commonly accomplished on hardware, one method being Field Programmable Gate Array (FPGA) development. Another method is to mix FPGA development with software, implementing the most computationally expensive tasks on the FPGA. In fact, the Ettus X300 SDR comes with an FPGA, specifically designed to allow for this hybrid development if desired. The most extreme case for hardware development is application-specific integrated circuits (ASIC), which is the least flexible and most time consuming to implement. The hardware route (whether it be ASIC or reliance on FPGA) has been the default path for waveform development for a while as the performance trade off was simply not a reasonable decision to make even just a few years ago. A graph showing these trade-offs is given in [23] and is shown in figure 2.1 with GPP development having highest ability for rapid prototyping and reconfigurability. I would go further and say SDRs have more potential than simply prototyping and can be used for commercial waveform development, granted this paper is from 2016 so GPPs were significantly weaker back then, which may be why the authors specifically chose to say SDRs are mainly for creating prototypes, as that was the most appropriate use at the time. Over the years GPP performance has exponentially grown to the point where these default assumptions of needing hardware should be challenged, a significant amount of computational performance can be leveraged if software and modern processors are used to their advantages. A software based waveform implementation will commonly
Figure 2.1: "Trade-off between reconfigurability and development time for FPGA, ASIC, DSP, GPP, and hybrid GPP/FPGA-centric SDR architectures" taken from [23]
use a lower level programming language such as C++, although there are no limitations other than performance needs and what a specific language can achieve. The work presented in this thesis will be completed entirely in C++ as the API used to interface with the RF hardware used is also implemented in and made for C++ development. C++ offers the high performance needed to develop waveforms with significant performance needs.

ANDRO Computational Solutions has performed case studies and has found that software-only waveform development is 3-5x faster than FPGA development [29]. This increase in development speed paired with the significantly lower cost of entry due to expensive tools and licenses needed to develop on FPGA and other hardware makes software based development very attractive if it is known that the performance goals can be reached.

2.2 Hardware vs Software

These different development methods come with the obvious trade offs. For this section, hardware development will focus on FPGA development, but the points will still hold for all hardware based development in general. An FPGA based waveform will almost certainly have higher performance than a software based application if done correctly, there is no disputing that. The most crucial question to be asked when it comes to deciding if FPGA should or needs to be used is if that extra performance is needed. It is a very difficult question to answer if one does not have any data points to refer to in terms of this performance difference, which is why exploring this development and setting a benchmark is significant. These benchmarks are obviously irrelevant if the techniques used to get high performance in software are not also documented, which is why they should also be tied to this work. What can be accomplished in software, and how, are needed to make the best decisions. Due to the exponential growth of GPP performance, the answer to this question may shock FPGA and other hardware developers that have put the blinders on, along with potential customers that may be unaware of what can be accomplished with a software only approach on a modern GPP.

Software development is significantly quicker than hardware development, along with the valuable addition of flexibility and the ability to rapidly throw out prototypes throughout the development process. This allows for rapid
prototyping in software only approaches, which is valuable information for the development team and customers as project time moves forward. For an FPGA approach, it can take months before there is anything to show for the work, whereas software prototypes can be thrown out in the matter of weeks. Since software implementations are quicker and more flexible, it is also easier to implement a waveform specification iteratively and demonstrate progress and abilities with more resolution. If it is determined that the waveform performance needs can be met through software, then all the advantages go to the software development route. The only reason one might still choose to go with the FPGA or hardware route is inexperience with software based development, or a strict requirements to develop on FPGA or some other hardware.

### 2.3 SDR Development

Software based waveform development requires RF hardware that the software can interface with, this is where Software Defined Radios (SDRs) come into play. There are plenty of solutions available at many cost levels, ranging from under $50 to tens of thousands of dollars. While the upper range may seem very high, it still pales in comparison to the hardware based alternatives. A high performance SDR can be used for testing and development of any software based waveform, so there is plenty of reuse once the investment is made. The SDR solution will naturally be driven by cost and performance needs. For this work, an Ettus X300 equipped with a UBX-160 daughterboard will be used, an approximate cost of $8300 [24,25]. This SDR is considered a higher end radio, with a maximum sample rate of 200MHz and a maximum instantaneous bandwidth of 160MHz. For the sake of some comparison and context, a low cost solution from the same brand, Ettus, is the B205mini-i [26]. The B205mini-i has an approximate cost of $1300 and has a max sample rate and instantaneous bandwidth of 61.44MHz and 56MHz respectively. Ettus provides an open source driver for their radios, UHD [24, 27], which is used to interface with all of their SDRs. This is another great advantage to these radios and software development in general. The UHD API uses the same commands between all Ettus radios (apart for some very specific commands that only higher end SDRs support), meaning that once a waveform is built to support an Ettus SDR, you can for the most part, plug and play with RF hardware. The main considerations to make in
this case is to verify the requested sample rate, bandwidth, etc, are all supported by the SDR hardware you wish to use. For the sake of this research, since the X300 is the highest performance SDR that I have at my disposal for this work and the focus is high performance, the ability to swap any Ettus radio will not come into play, but it is an important ability automatically built-in when the software is created. When improved hardware is released (ex. Ettus releases a new radio that has higher performance than the X300, and a new processor is released), it will allow for the work presented here to be reused and possibly showcase even higher performance without requiring any changes to the software. This ability to swap/support numerous radios at multiple price points demonstrates the flexibility software allows.

The next question a skeptic may ask is about the reliance on Ettus products and the UHD driver. Since this is software, there is nothing stopping developers from adding support for multiple other radios into their software. Once the software for those new radios and drivers is developed, it can be reused in future waveform development and built up as more SDR applications become available, speeding up future work.

2.4 Waveform Basics

A waveform will consist of a MAC (Media Access Control) and a PHY (Physical Layer). This work will strictly focus on the PHY layer as that is where the application of OFDM occurs. This work assumes a developer already has a MAC or plans to create one, and is making a decision on the path they will take for PHY development (a possible consideration being a hybrid FPGA application where the PHY is implemented on an FPGA). Another reason for the focus on a PHY rather than a MAC is the PHY typically carries the most computational load, so this is a driving decision on whether or not a software implementation will be explored. The primary focus of this thesis is to research and determine what level of performance is possible in a software based OFDM PHY on a run of the mill GPP, in this case an Intel laptop. A higher performance PC was also tested on to provide more information on what performance can be expected on higher end systems, as that may be closer to the target platform in many circumstances. The MAC layer is above the PHY and performs the management. It will manage what data, synchronization sequences, and everything else the waveform needs to
transmit or process on the receiver end, and supplies the PHY layer with the
data or commands to perform those tasks. The PHY would then pick up this
data, processes it into an OFDM signal, and then transmit it. The reverse
order would happen on the receiver end, where an OFDM signal would be
detected, processed, and data would be sent to the MAC layer to be decoded
and processed. The MAC can also control what processes the PHY performs,
in this case the MAC would be able to tell the PHY to enter the initial search
for the OFDM frame detection, and once the signal is found, to switch to
tracking mode where the correlator can skip to the approximate location of
the next frame to operate more efficiently. This tracking stage is also used
to determine if we have lost the signal to allow the MAC to switch back to
initial search. The point here is the MAC is in control of all the decision
making for the waveform, and the PHY’s job is to perform the computationally
expensive signal processing.

In regards to the performance of this PHY, simply throwing together some
code will obviously not be sufficient for most applications, certainly not
enough to come to a conclusion on what performance is possible. A per-
fect example is if one were to use the Fourier Transform, it would be highly
inefficient and result in poor performance. A waveform developer would no
doubt use the FFT algorithm to perform the Fourier Transform, but there
is nothing stopping an newbie from fruitlessly attempting to implement the
Fourier Transform due to inexperience and lack of knowledge. This can be
applied for many other methods used in signal processing, or waveform pro-
cessing in general; the ability to accomplish tasks effectively in software is
crucial for creating high performance applications. There are a plethora of
references and software prototypes for almost any digital signal processing
technique you can think of, many of which may use inefficient methods, which
is fine for what they are typically designed for. While these sources are a
great teaching tool, there is much more needed to accomplish what this work
sets out to do. Real time software applications are not as common to find,
as the prototypes and resources are typically put together for educational
purposes, or proof of concept, so there is no goal to run this software in real
time, which would not be possible anyways in their form. It is possible to
find some applications, but they all tend to use low performance methods
that only allow for very basic waveforms to be run in real time, like an FM
receiver [20]. A low performance radio development toolkit exists that en-
geineers tend to use for their SDR development and testing, which will be
discussed and is precisely what I mean when referring to developers using inefficient methods.
Chapter 3

Research

3.1 Literature Review

5G, Wi-Fi, and OFDM overall are very well known, so references and information on these topics are easy to come by. Tutorials on 5G seem to be everywhere, with MATLAB even providing a toolbox for simulation and analysis [1]. All of these resources are for educational purposes, so while those are helpful for understanding the basics of 5G, they are not useful when it comes to creating a software only implementation. The MATLAB software on the other hand, while not being a high performance, or a real time system, does have its uses. This toolbox would be very useful for verification of the software based implementation. This seems like it would be a great tool to have if this work were to expand into a 5G specific PHY and MAC, which could potentially be future work.

One well known tool in the SDR community is GNU Radio [12]. An FM radio made with this tool was referenced earlier, with the point that this tool has very low performance and only very simple waveforms such as FM can be expected to run in real time. This is a rather obvious trade off you need to make when deciding to go with a plug and play sandbox, generalization tends to hurt performance. It is also difficult, if not impossible, to use optimization methods with this tool, such as creating a multi-threaded program. There is simply no way an OFDM application written with this tool will have good performance, I would categorize this under the education category as it is a great environment for beginners to familiarize themselves
with waveform design and signal processing.

There are many articles and papers that have investigated software based applications that use an OFDM PHY, primarily from the IEEE 802.11a standard, which is the specification for Wi-Fi. Some papers look at different parts of the 802.11 standard which are also fine as they all still require an OFDM PHY.

- **Reference [14]** GNU Radio Companion was used to create an IEEE 802.11a transceiver. A bandwidth of 20MHz was tested at a bit rate of 54Mbps. Performance for this work was done more from a signal analysis perspective than processing performance, documenting BER, Doppler shifts, and FFT plots under noise.

- **Reference [21]** This work also looked at performance of OFDM from a different perspective than I plan to. OFDM was compared to Generalized Frequency Division Multiplexing (GFDM). A comparison between both methods was made in terms of BER performance and latency, with OFDM found to perform better. For the OFDM processing, an FFT size of 128 was used with QPSK modulation and a sample rate of 1MHz. Once again GNU radio was used for this work.

- **Reference [4]** GNU Radio was used to implement the 802.11p standard. A maximum bandwidth of 20MHz was tested with an FFT size of 64. A "fully saturated" test was ran to see how the processing for all the components holds up. A sample stream of 1500 Byte frames with 64-QAM modulation and a 3/4 rate decoder was tested. It was found that the implementation was able to run in real time, with the Viterbi Decoder being the bottleneck of the system, taking up 34 percent of the overall processing. How much higher they could go is unknown as the performance results do not provide enough information to make an estimate.

- **Reference [16]** This is another study where performance was looked at from a signal analysis perspective, BER being their main metric along with graphing the constellations of the received signal. GNU Radio Companion was used for this work with an FFT size of 512 at a sample rate of 1MHz.
• **Reference [13]** GNU Radio Companion was once again used in this paper. The BER was studied under several different modulation schemes that can be used for the 802.11a standard. Since the standard was followed, the bandwidth used was 20MHz with a bit rate of 54Mbps.

• **Reference [15]** This paper looked at a transmitter only implementation. This was an FPGA based implementation that operated at a sample rate of 40MHz as that was the maximum rate of the RF hardware used. Unfortunately no performance metrics were used that can give us an idea of what the maximum performance of this application is.

• **Reference [19]** I also found a repository that is an SDR implementation of Wi-Fi 4. After looking through some of the information and instructions for this project it turns out they have went with a hybrid approach where some of the processing has been implemented on an FPGA. This means it is not a software only project as it does rely on some specific FPGA hardware. Regardless of using FPGA, the performance seems to be rather low, with the claim of 30.6Mbps and 21.5Mbps upload and download speed respectively. When I saw there was an FPGA in play I was expecting to see higher numbers to be honest. Wi-Fi 4 is supposed to support speeds up to 54Mbps. This is a perfect example proving that high performance is not guaranteed, the processing and implementation need to be done effectively.

• **Reference [9]** This is another GNU Radio implementation that reports an FFT size of 64 and a sample rate of 500KHz.

• **Reference [7]** This work used the GNU Radio Companion, it also reports an FFT size of 64 with a sample rate of 2MHz used.

As shown here, GNU Radio and the GNU Radio Companion application are very popular when it comes to software based SDR applications. The focus of these papers tended to use the mindset of "can this specific thing run in real time" versus my mindset of "what is the maximum that can be achieved". This means the papers relating to this type of work didn’t supply enough information to give an idea of what is truly possible in software. This, paired with small FFT sizes and sample rates, makes it clear why many people in the field may discount software based development. If waveform developers were to investigate performance for an OFDM PHY layer, these are the results that would be found, and the very low sample rates and FFT lengths
would immediately eliminate GPP development if they did not wish to conduct a performance evaluation for themselves using C++ instead of a toolkit. The use of GNU Radio is logical for a quick signal analysis when it comes to BER, Doppler shift and so on, but not for throughput performance; the papers whose focus was on these aforementioned analyses was an appropriate use of GNU Radio. In my opinion, this is where GNU Radio hits its limit in terms of being used for what its designed for.

While I am saying GNU Radio is not appropriate for measuring performance, it is clear why it was used for the work presented here. GNU Radio is the quickest way to throw together a prototype and get results; and since these results were successful, no harm no foul. This is obviously dependent on the performance goals being low enough for GNU Radio to support, which all of these tests were. I mentioned that the Ettus X300 can handle up to 160MHz of bandwidth, yet these tests only go up to 20MHz and 40MHz. These goals make sense for those tests as those bandwidths were near or at the limit of the hardware used for that specific testing. The hardware limitations and goals chosen are the main reasons why GNU Radio was an appropriate solution for those cases and why they were successful. The scope of my work is an extension of this, where I would ask "how much performance can be achieved?", meaning my next step would be to swap out the hardware they used for an Ettus X300 and see if the hardware limits of that platform can be reached; which I assume would not be possible. It is at this point that GNU Radio would be dropped and the switch to C++ would need to occur to reach the true potential of software. This is why I will be going straight into C++ development rather than starting with GNU Radio, as I need to give accurate results for what can be achieved in software, requiring me to use the method that results in the highest performance.

As with everything there will be a trade off for choosing C++ over GNU Radio. This development will take significantly longer than the GNU Radio equivalent as I will need to write all the signal processing from scratch. This requires me to understand the underlying math behind all this processing, something a GNU Radio developer (although they shouldn’t) can go without. Since this signal processing will be written from scratch, it will not only be personalized for this specific task, but all processing blocks will be designed and optimized around the other blocks, which allows for significantly higher performance. I will also have full control over how I optimize the architecture
of the software, something not possible in GNU Radio. With performance being my most important consideration, I need to take this trade off. Compared to the other waveform development methods (hardware based), this implementation time will still be significantly quicker, so in reality this trade off hardly matters in the grand scheme of things.

3.2 OFDM

Orthogonal Frequency Division Multiplexing (OFDM) is a modulation technique widely used in communications today. It is a specific form of frequency division multiplexing, which at the highest level means data is being split into multiple streams that will be transmitted at different frequencies. A simple example would be Wi-Fi, most people have numerous devices connected to their network, all sending and receiving data simultaneously. This is possible as each device can be allocated a specific frequency (or multiple if the device needs more bandwidth, i.e. more data transfer speed). These specific frequencies are called subcarriers, and the amount of subcarriers in a signal tells you how many frequencies/streams can be transmitted concurrently. A phone streaming 4k video may be using subcarriers 1-50, while a phone loading a text message may only be using subcarrier 51 as it does not require as much information and speed. A diagram of the Wi-Fi subcarriers, 3.1, is shown below, the X axis being the frequency domain. Note that subcarriers can be assigned different roles in a signal, while arbitrary these are specific
to the specifications of waveforms. More on this later.

This leaves the final distinction, orthogonal. In the simplest terms this means that the frequencies of the subcarriers are a specific distance, or delta, apart. That specific delta results in each carrier having an integer amount more complete cycles than the previous carrier within each OFDM symbol. This means the minimum distance between subcarriers is one cycle per symbol, so if one subcarrier has 3 periods in the symbol, the two adjacent subcarriers can have 2 and 4 periods within the symbol. Of course they can be any integer amount apart, meaning the adjacent subcarriers can also have 1 and 5 periods within the symbol, that would mean the subcarrier spacing is larger in the given signal. When an OFDM signal is described, the parameters (specifically the subcarrier spacing) will not be given in this way, so it is important to understand how to convert between the two. Every OFDM symbol has a specific length (or period), this can be given as either the symbol time, or the symbol rate. The subcarrier spacing will be given in terms of frequency (Hz). Using this frequency and the known symbol time/rate, the integer period spacing that was previously mentioned can be calculated. Below is the formula for this conversion:

$$Subcarrier\ Spacing = \frac{k}{Symbol\ Length}$$

Where:

- Subcarrier Spacing is in Hz
- $k$ is the integer amount of cycles delta between adjacent subcarriers
- Symbol Length is in seconds

If the symbol rate is given, the inverse of this value would result in the symbol time. Now that subcarriers have been described along with how they are laid out in an OFDM signal, the next step is knowing how to put information into these subcarriers. Each subcarrier can be filled with modulated data, which is commonly Phase Shift Keying (PSK) or Quadrature Amplitude Modulation (QAM). Other modulation techniques can be used, such as Frequency Shift Keying (FSK), but the focus of this paper will be using PSK, specifically QPSK. QPSK, the Q standing for Quadrature, means there are four unique symbols with different phases, typically referred to as
constellation points. Two bits of data are taken at a time, and mapped to one of the four constellation points, as there are four possible combinations that two bits can create. How bits are mapped to the constellation points is completely arbitrary as long as the constellation map depicting this mapping is supplied, in a specification for example. Each subcarrier will receive a QPSK symbol, which will later be recovered by the receiver, mapping the subcarrier symbols back to bits, and reassembling them back to useful data. That would be handled by the MAC, and again is arbitrary and depends on the waveform design, OFDM and the PHY do not care about this. The process of putting these QPSK symbols into subcarriers is typically called resource mapping. A QPSK constellation is shown in 3.2 with an arbitrary bit mapping.

Figure 3.2: QPSK Constellation with Arbitrary Bit Mapping
There are different types of OFDM that exist, the two types used in 5G are Cyclic Prefix OFDM (CP-OFDM) and Direct Fourier Transform spread OFDM (DFT-s-OFDM). For this work CP-OFDM was chosen as the target modulation, with its implementation in 5G as the motivation. CP-OFDM allows for spacing to be inserted between the OFDM symbols to help counter inter-symbol interference (ISI). ISI is a distortion of the signal where symbols interfere with subsequent symbols, which will result in a less reliable signal. Since this cyclic prefix is adding space between the symbols, it diminishes the effect a symbol will have on the next symbol, resulting in better signal quality. This prefix is scalable and is given as a ratio of the length of the symbol. While this ratio can be any arbitrary value, ratios such as 1/4, 1/8, 1/16, and 1/32 are common. To generate the cyclic prefix, the end of the OFDM symbol is prepended to the front of the symbol, meaning if the cyclic prefix is a size of 1/4, the last quarter of the OFDM symbol is copied and inserted before the front of that symbol. Note that the OFDM symbols are currently being referred to in the time domain, not the frequency domain. This conversion will be covered soon. The clear trade-off here is the more guard you insert between the symbols, the less ISI you will have, but the lower your overall data rate will be as more of your time is eaten up by spacing between the symbols. A diagram showing the insertion of the cyclic prefix is shown in 3.3. The symbol length in figure 3.3 is 5000 samples with a cyclic prefix length of 1/4. This results in inserting a copy of the last 1250 samples at the front of the symbol, resulting in 6250 total samples.

It must be noted, as it was confusing to me when I first investigated this, that the cyclic prefix should not be thought of as part of the symbol when it comes to symbol length, rate, and subcarrier spacing. This means that there will effectively be two symbol rates, one that considers just the symbols, and one that considers the symbols with the cyclic prefix inserted. This is because the subcarriers need to be orthogonal within the original symbol, as the cyclic prefix is not technically part of the symbol; it is a guard between the symbols. Again the point of the cyclic prefix is to reduce ISI, so if the prefix was a part of the symbol, nothing would be accomplished. I did not find any terminology for these two different symbol rates in my research, which may be the cause for my initial confusion. I will refer to the original symbol rate before cyclic prefix insertion as the symbol rate, and the symbol rate after cyclic prefix insertion as the overall or total symbol rate to avoid confusion. The total symbol rate will always be less than the symbol rate.
Figure 3.3: Diagram of Cyclic Prefix Insertion
To identify the framing of the OFDM symbols (i.e. where does each symbol begin) we can take advantage of knowing the cyclic prefix is a copy of the end of the symbol. An autocorrelation can be performed, with the delay being the symbol length to identify the cyclic prefix, and used to figure out where the symbol starts. If one was to run the autocorrelation on the entire signal they would see a pulse (or peak) at a rate of the overall symbol rate. This will come in useful when it comes to optimizing the correlator, as the approximate location of the next peak is known (i.e. when the next OFDM symbol begins).

While this is a high level overview of CP-OFDM, these are all the basic concepts needed to get started with creating a CP-OFDM based PHY. Further detailed explanations of how this was all accomplished efficiently in software will be given in the implementation details as they do not pertain to CP-OFDM itself.

The last piece of processing remaining is turning these subcarriers into an OFDM symbol, as this step was skipped to discuss the cyclic prefix insertion. Turning these subcarriers into an OFDM signal will translate the subcarriers from the frequency domain to the time domain. Not only does this conversion need to be made, but it must be done with orthogonality in mind. The most efficient way to do this, given the QPSK modulated subcarriers, is to use the Fast Fourier Transform (FFT) algorithm with a specific sample rate that automatically spaces each bin so it is orthogonal from its adjacent bins. The frequency delta between FFT bins is calculated with the following equation.

\[
\text{BinFrequencyDelta} = \frac{\text{SampleRate}}{\text{FFTSize}}
\]

Where:
- Bin Frequency Delta will be equivalent to the subcarrier spacing
- Sample Rate is the sample rate of the signal in samples per second
- FFT Size is the number of bins

After converting to the time domain using these parameters, this signal can
then be resampled to any other desired sample rate. To convert these subcarriers from the frequency domain to the time domain the Inverse FFT (IFFT) algorithm must actually be used. The input for the IFFT algorithm will also be the bins, each bin taking a subcarrier (i.e. a QPSK symbol in this case) and translate it into the time domain. It is at this point that the cyclic prefix insertion can take place. The result is a CP-OFDM symbol which can be added to a stream to create a CP-OFDM signal. On the receiver end, after identifying the framing of the signal, the signal can be resampled back to the sample rate used in the equation, and the FFT algorithm can be used to map the OFDM symbols back into the QPSK subcarrier symbols in the frequency domain. The resulting QPSK symbols will then be demodulated back to bits.

A full OFDM PHY will have more processing than this, specifically error correction. This will not be explored in this work as there is simply not enough time to develop and optimize an error correction component within the given time frame.
Chapter 4
Software Development

4.1 Contributions

The overall contribution for this thesis is the high performance implementation of an OFDM PHY, specifically CP-OFDM, along with the documentation of the performance on two different platforms. The optimization techniques developed and implemented to achieve this performance will be broken down into a few main contributions:

- **Asynchronous Buffer Manager** This is a thread safe buffer passing implementation designed to eliminate idle time under worker processing time variation.

- **OFDM Workers** The worker architecture needs workers, so the next contribution is a TX OFDM worker and RX OFDM worker. These are specifically designed to be CP-OFDM workers and their processing and work flow has been optimized for the highest possible performance.

- **CP-OFDM Frame Detector** This is the third and final main contribution. This frame detector performs auto correlation to detect the OFDM framing. The calculation has been optimized to reuse previous correlation results to allow it to run in real time on a GPP.

All development was done using C++ and built using the GCC 9.3.0 compiler. CMake was also used to manage the organization and building of the software, version 3.16.3 was used in development. This software was built for
and tested on systems with the Ubuntu 20.04LTS operating system. While in its current state it may not work on other operating systems or even different versions of Ubuntu, only small adjustments would be required to allow for this support. Testing was done using Ubuntu18.04LTS, GCC version 11.1.0, CMake version 3.20.3 and on another Ubuntu20.04LTS system, GCC version 11.2.0, CMake version 3.23.0 with no code changes needed on either platform, demonstrating that the code is relatively flexible in its current state. The ability to throw the software onto another system and run it right away is one of those desirable advantages of using software over hardware.

4.2 Asynchronous Buffer Manager

4.2.1 Introduction

This manager, referred to as ABM from here on, in reality consists of two managers, one on each side of a worker thread distribution. There is no limit as to how many workers must be in the distribution, it can be as little as one worker if a GPP is highly limited. Before any other details are given, the motivation behind this manager should be made clear. One well known C library designed for parallel/distributed computing is MPI, Message Passing Interface [18]. While this is a very useful and powerful library to create parallel and distributed applications, its generality and wide range of accommodations slows it down in certain situations. For example, the method in which buffer communication is accomplished is designed to accommodate distributed computing, which makes data passing within the same system needlessly intense. Passing from one thread to another requires a deep copy regardless of whether or not it would make sense in that situation. The ABM allocates all buffer memory during initialization and only passes references to optimize the data passing between threads, which is happening at a high rate and is why deep copies should be avoided at all costs. This reference passing means distributed computing across multiple systems is not possible, but that is not necessary for what I am trying to accomplish. Another issue with MPI is the limited amount of passing methods. Two of the most common passing methods are to pass a buffer to a specific worker ID, or to broadcast to all workers. Broadcasting does not fit into the OFDM PHY since each worker will work on its own specific data, leaving the passing of data to specific workers as the only option. Since MPI can only pass to a specific
worker, it must wait until that worker is ready and picks up the data before the data passing can proceed. In other words, all passing is done serially, or in a specific order. While this is a perfectly valid method, it is not optimal as other worker threads may be ready and idle while MPI waits for the next worker in line to pick up its data. This idle time is where efficiency is lost and is where the asynchronous aspect of the ABM comes in. By removing the need for the passing method to take a worker ID, or any specific order for passing, the concept of a job queue can be applied for the worker thread data passing management. Similar to how a computer will have jobs lined up and pass them to available cores, data buffers can be assembled for processing, allowing data to be sent to the next available worker. The main difference here is this implementation is more specific as the workers all have the same job and are waiting for information to process, rather than waiting for a specific task. This does not have to be the case as the ABM is a template, meaning a software developer can create a data structure with metadata to handle different jobs. Regardless, this removes the idle time issue by fulfilling requests from workers rather than waiting for a specific worker to be ready. It would be possible to create a workaround in MPI for asynchronous passing, but it would not solve the deep copy issue; also if a workaround needs to be developed, simply creating an optimal solution from scratch may make more sense.

This may seen unnecessary at first as the idle time can be very minimal and result in trivial performance gains. While this can be true, there is no reason to waste this idle time if it can be fixed by altering how data passing is managed. The reason I say it can be true is there are many factors that come into play that can increase the idle time, or the processing time a worker thread takes that may increase the idle time of other workers. When running applications on a GPP, you are at the mercy of the scheduler. This means some workers may take longer to accomplish their workload because they were interrupted for other system tasks. These interruptions become more apparent the more a system is pushed as there will be less cores available, if any, for the OS to run background tasks on, so it will have to interrupt workers to accomplish those tasks. Other factors that can affect processing time is the worker processing itself. If there are several branches within the worker or other conditions that must be met, that specific iteration may take longer to process than the previous. Branch predictability is one way in which the system tries to optimize its processing as much as possible. The system will predict a section of code that will need to be processed (think of
an if-else block) and start executing it ahead of time. If predicted correctly, once that branch is reached the processing will already be ahead of schedule, saving time. If predicted incorrectly, the system will need to restart the other branch from scratch, not saving us any time, which will feel like a net loss. If one worker predicts incorrectly, and the subsequent worker predicts correctly, in a serial passing architecture that time saved will turn into idle time as the second worker will have to wait for the previous worker to run through all of its processing again. How much of a factor branch predictability plays into the optimization of the workload depends on how predictable the branches are, if one specific branch condition is met 90% of the time, then the system will always predict that branch. This means that around 10% of the time the branch will be predicted incorrectly and a slowdown will be felt, introducing idle time. If the branch conditions are uniformly met, then this extra idle time will occur at the highest rate, as the branch is completely unpredictable. This will result in the most amount of idle time between workers.

If the worker is waiting for some condition to be met, depending on the process, it may be a significant slow down as well, which can play a factor in the deviation of workload. Another slowdown that can happen is a cache miss. While not an exhaustive list, it is clear that many factors are at play that can introduce idle time into the worker thread processing. Overall, the worker threads should be designed to try and mitigate the factor these play into the overall processing, and in turn the idle time, but it is not possible to eliminate all of them, especially interruptions from the operating system. This is why the thread passing should take this into account and do its best to eliminate the effect it has on overall processing speed.

The most important reason for the ABM has yet to be mentioned though. When people think about 5G and Wi-Fi, the first device that comes to mind will most likely be a cell phone, or some other portable device. These devices all have one major thing in common, they all use ARM architecture due to its power efficiency, which is crucial for long battery life. One proponent for this efficiency is the big.LITTLE architecture that ARM uses in these devices. When looking at the specifications for these devices, we will typically see the chip set split up into multiple cores with different clock frequencies. The cores with lower frequencies are the LITTLE cores, they have lower performance that allows them to be very efficient when it comes to battery life. The phone will try to use these cores as much as possible to increase bat-
tery life. The cores with higher frequencies are of course the big cores, they sacrifice battery efficiency for high performance, mainly saved for processor intensive tasks. The specs for the chip set of a Samsung Galaxy S22 Ultra are provided in 4.1 to provide some context for this architecture. There are eight total cores, with a blend of three different types of cores. At this point

<table>
<thead>
<tr>
<th>Core</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cortex-X2</td>
<td>3.0 GHz</td>
</tr>
<tr>
<td>Cortex-A710</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>Cortex-A510</td>
<td>1.7 GHz</td>
</tr>
<tr>
<td>Cortex-A510</td>
<td>1.7 GHz</td>
</tr>
</tbody>
</table>

Figure 4.1: Samsung Galaxy S22 Ultra Chip Set

it may already be apparent why the ABM needs to be asynchronous, these slower cores can not be allowed to drive down and cancel out the speed of the higher performance cores, as the performance difference in the cores will result in significant worker processing time variation. If serial passing were to be used, all the extra performance and processing speed from the big core(s) will only result in more idle time while waiting for the LITTLE cores to catch up with their iteration of processing.

4.2.2 Implementation

To allow for maximum flexibility, the ABM has been implemented as a template. This template applies to the data that will be passed between the threads, in the case of the transmitter PHY, the input (I will refer to this as “pick-up”) for these workers will be an array of integers. These integers are the bits that the MAC requests and sends to be processed into an OFDM signal. These bits will be processed into samples for this signal, the result being an array of complex 16 bit integers, the native data type used by UHD. This means the output of the transmitter PHY will use an array of 16 bit complex integers as the template for its output (I will refer to this as ”drop-off”). This makes pick-up and drop-off the two sides of the ABM that were previously mentioned, while the logic used within these two managers is similar, there are several crucial differences in how they manage the requests made to them by the workers based on how they will receive and pass out buffers. The receiver PHY will use the same template types in the reverse order, as it will take in samples and process them back into bits.
Upon initialization the two managers will both allocate a pool of buffers, consisting of the template type supplied. The size of the buffer pool is passed in as a parameter to the managers upon startup. Each buffer will also contain internal metadata that signifies the state of that buffer, the two possible states being read and write. When the manager is initialized each buffer will start out in the write state, as it is considered empty and needs data written to it. After data is written to this buffer the state transitions to the read state where it can be picked up and read from. Once the buffer is done being read from, it will transition back into the write state and begin the process over again. A state transition diagram is provided in 4.2 to show the process. The process begins by requesting a write buffer, a "Get Write" command. A reference to the buffer will be returned along with the manager locking the buffer. Once the buffer has been written to, a "Pass Write" command is given and the manager changes the state of the buffer to the read state. At this point the reference to the buffer on the write side should be discarded. The "Get Read" command will now be used to obtain the buffer, and once done with the buffer the "Pass Read" command is given to switch it back to write state. At this point the reference to the read buffer should be discarded. These buffers will all be passed using FIFO order as there may be multiple read or write buffers available. Since the managers are request based and asynchronous, they do not need to know the number of workers that will be sending requests, as it is irrelevant from the perspective of the managers.

It is at this point where these two managers begin to differ due to how they must handle the asynchronous requests. Figure 4.3 shows how these two managers will interface with the worker threads, along with the opposite ends of this worker distribution. At this point it may be more clear why the managers are named as they are. The names come from the perspective of the worker thread distribution, as they will all be picking up their buffers from the pick-up manager, and dropping off their results into the drop-off manager. In both cases the manager handles one side asynchronously based on FIFO requests, and the other side from only one thread. The side at which the asynchronous handling is done makes all the difference in how the manager must order and process the requests and buffers internally.

I will start with the pick-up manager, it is designed to take "Get Write"
and “Pass Write” from only one thread. As these buffers are written to and passed back, the manager will tie some metadata to each buffer, primarily the ID of the buffer. The ID of the buffers is how the order will be preserved, starting from an index of 1. Since these buffers will be passed into the manager, the handling is rather simple. When a worker requests a buffer, it will receive the next in line, meaning all the read buffers will be passed out in order as well. A mutex is used in several positions within the interactions to avoid any race conditions from occurring.

The drop-off manager is where the handling gets more complicated, as the buffers coming in will be out of order. To avoid a lockup from occurring, workers must inform the manager of the buffer ID they will be writing into the buffer pool. The manager will ensure that there will always be a location available for the next buffer in line just in case it happens to be running late.
This needs to be done as the buffer pool was allocated upon initialization and is finite. The buffer pool should be created with a size large enough to prevent this situation from occurring in the first place as it will add idle time into the system. If the buffer ID is deemed valid by the manager, it will accept it, which allows the workers to immediately move on to the next buffer on the pick-up end, making this passing method asynchronous. This write buffer will accept the metadata that was originally given from the pick-up manager, the buffer ID, and also add its own metadata on top, primarily the location of this buffer in the pool. Once the worker is done using the buffer, the worker will pass the buffer back to the manager, and the manager will use this metadata to keep track of what IDs are in the pool and where. Again, these interactions are all completed with a mutex strategically placed to avoid race conditions, but also allow for the manager to do as much computation as possible around this mutex to help with performance. On the other side of the drop-off manager, a thread will be requesting read buffers. The manager will know what the next read buffer ID needs to be to preserve the buffer order from the input of the pick-up manager. When the ”Get Read” request is made, the manager will already know what IDs are ready, as they are saved when the workers pass in their write buffers, so it can verify if the next buffer in line is ready. If so, it will obtain the location of this buffer and pass the reference to the requesting thread. If the next buffer in line has not entered the pool yet, the manager will stand by, wait for it, and then pass the buffer, making the interaction as seamless as possible for the requesting thread. Once the read buffer is done being used, the manager will throw it back into the write pool, keeping track of which buffer locations are designated as write buffers, just as the read side is managed. This allows the
manager to know which pool locations it should be passing to the workers when write requests are made.

4.3 CP-OFDM Workers

4.3.1 Setup

It has already been mentioned many times that a worker architecture was chosen for this development, which brings us to the details of how the workers were implemented and optimized. Each worker utilizes the same function, with the main difference being the ID tied to each individual worker, starting from ID 0. These IDs are important as the PHY will allocate memory for each worker, stored in vectors where their ID maps to the location where the memory was allocated within. Thus the worker with an ID of 0 will always use index 0 within these vectors to obtain its memory, allowing the function to be consistent and reused for all workers as they will pass in their ID to the memory blocks to obtain their specific piece of memory. These blocks of memory are only allocated once during initialization, just as in the ABM, as a method of optimization. These memory locations will be reused and overwritten every worker iteration. Figure 4.4 shows this worker and memory layout. While this does apply to the worker distributions in both the TX and RX PHY, the worker function is different between the two PHYs as the RX must apply the inverse operations to the OFDM symbols to obtain the original data. Once the memory is allocated, the PHY can start. To be clear, each PHY (TX and RX) are initialized separately and started separately to allow for TX or RX only operation if desired, but these details simply apply to both PHYs. When the PHY is instructed to start, it will whip up the workers, the worker count being one of the parameters that was supplied during initialization. Since both PHYs are currently created for software simulation they both take in a reference to a drop-off ABM. The TX PHY will write its results (an OFDM signal) to this buffer, and then the RX PHY will pick them up and process them. This allows me to simulate transceiver operation and also verify that the RX PHY processing is indeed correct, as it should give me the same bits back that were fed into the TX PHY. Both workers will use the FFTW3 library to perform the IFFT and FFT processing. The FFTW3 library is not completely thread safe so some considerations must be made when designing these workers to avoid race
4.3.2 TX Worker

Now we get into where these two workers differ, starting with the worker on the TX side. The input will be bits, already ordered in the way the MAC desired them to be mapped as far as the subcarriers are concerned. This order is arbitrary as long as the worker and MAC are on the same page, allowing for the MAC to accurately place its bits in the subcarrier domain. For the simulation done in this work, a MAC does not exist, so the PHY will generate its own bits randomly and feed them into the pick-up ABM where the MAC would. The workers will be on the other side of the pick-up ABM, obtaining the input bits for their next processing iteration. Each buffer will be filled with one complete OFDM symbol’s worth of bits, this amount is constant and calculated upon startup when the parameters for the waveform are read in; details for these parameters covered later.

When the worker first starts up it will create its plan for FFTW3; this plan tells FFTW3 which FFT direction to compute, the input and output pointers, and the length of the operation. This is one of those FFTW3 functions that is not thread safe, so a mutex is thrown around this call to ensure these plans for the workers do not cause a race condition. When the processing is complete and the worker is exiting, it will destroy this plan. After plan
creation processing will begin as follows:

- **Get Read**  Request a read buffer from the pick-up manager. The pick-up manager will pass back a reference to the read buffer along with the ID of that buffer. This read buffer will contain an array of bits equivalent to the amount of subcarriers times the amount of bits per subcarrier.

- **QPSK Modulate and Resource Mapping**  The worker will QPSK modulate these bits, inserting them into the correct subcarrier locations for the OFDM conversion when the modulation takes place. This includes accounting for the frequency guards in the FFT process, the size of which is calculated upon startup based on the waveform parameters. This guard was also used to calculate exactly how many bits will be used for each OFDM symbol, and subsequently each read buffer. Little things like this, where components are specifically created to format their output for the waveform implementation, is what helps us get the most from the GPP.

- **Pass Read**  The read buffer is immediately passed back once the worker has converted all bits into QPSK subcarriers. This allows the MAC (or whoever else is writing to this buffer) to get this buffer back as soon as possible and continue its processing.

- **Create OFDM Symbol**  At this point, the QPSK symbols can be converted to the time domain, resulting in an OFDM symbol. The QPSK symbols are fed into the IFFT function in the FFTW3 library. In reality, since the FFTW3 plan has already been created, the worker is just telling FFTW3 to execute the plan. The plan knows the input and output locations, whether to run FFT or IFFT (IFFT in this case), and the optimization strategy. FFTW_ESTIMATE was the strategy used. This plan execution is one of the only functions in FFTW3 that is thread safe, which is absolutely necessary and crucial for performance.

- **Get Write**  We are now approaching the last stage of processing, so the worker will request a write buffer from the drop-off manager.

- **CP Insertion**  The last processing step is cyclic prefix insertion. A section from the end of the OFDM symbol will be taken and copied to the front of the write buffer. Once again this amount of samples will be known as it will have been calculated upon startup based on the waveform
parameters provided. The rest of the symbol will then be copied into this write buffer. This is another place where time is saved as there are no explicit steps where data is copied, as buffer copying is integrated into the processing itself.

- **Pass Write** The worker will now pass the write buffer back to the drop-off manager. At this point it will be picked up by the RX PHY and processed. In a real implementation a radio transmission thread will pick these samples up and transmit them over the air.

Figure 4.5 shows this process.

![Figure 4.5: TX PHY Worker Block Diagram](image)

### 4.3.3 RX Worker

The RX worker process is similar to the TX worker, with the implementations and processing being the inverse and applied in the reverse order. The same optimization strategies and thought process were applied when planning and developing this worker, so those details will be omitted. The details for each of the blocks in the worker are as follows:

- **Get Read** Request a read buffer from the pick-up manager. The pick-up manager will pass back a reference to the read buffer along with the ID of that buffer. This read buffer will contain an array with one complete OFDM symbol. The data type will be 16-bit integer samples, the native data type for UHD.
• **Conversion** FFTW3 only works on float or double data types, so the 16-bit integer samples are converted to float samples. This conversion writes directly into the input location that the FFTW3 plan has been set to.

• **Pass Read** The read buffer is immediately passed back since it is no longer needed.

• **Obtain Subcarriers** The worker now tells FFTW3 to execute the plan. This time the FFT operation will be performed to convert the symbol back into the frequency domain.

• **Get Write** We are now approaching the last stage of processing, so the worker will request a write buffer from the drop-off manager.

• **Resource Demapping and Demodulate QPSK** The last processing step is mapping the subcarrier QPSK symbols back into bits, these bits will be written directly to the write buffer as they are demodulated.

• **Pass Write** The worker will now pass the write buffer back to the drop-off manager. At this point it would be picked up by the receiver side of the MAC, where it will deal with the data however it needs to. In simulation these bits are simply discarded, although there is a mode enabled in the code where the bits can be compared to ensure the processing end-to-end is correct. This test successfully passes when enabled.

The block diagram of the RX worker is given in figure 4.6.

![Diagram](image)

**Figure 4.6: TX PHY Worker Block Diagram**
4.3.4 Other Considerations

It should be noted that these workers are setup to be adjusted, as not all OFDM PHYs will use this exact process. For example, only QPSK subcarrier modulation was used in these workers even though other methods were mentioned in the beginning. This modulation function can be swapped out by a developer if a different modulation technique is needed, they would simply replace the function call with their own, making sure to use the same input and output format. The resource mapping can also be changed when this function is replaced. Very similar performance is expected from other modulation techniques as the majority of the processing time is used by the IFFT and FFT computation. While these are CP-OFDM workers, this framing technique can also be changed if needed, this may affect the processing speed as cyclic prefix insertion seems less computationally intense than the other methods given that this only requires the copying of existing data.

4.4 CP-OFDM Frame Detection

Running a receiver in real time requires significantly more processing power than a transmitter thanks to all the additional processing that is needed to recover the signal. The first step specifically pertaining to OFDM signal processing would be obtaining the framing of the signal, i.e. the start of each cyclic prefix. This can also be referred to as synchronization. The reason I say ”specifically pertaining to OFDM” is because typically some signal resampling or filtering is done to the received signal first, this PHY will not contain that processing due to time constraints and the scope of work. This frame detection, especially initial frame search, is typically one of the most intense stages of waveform operation, so the optimization of this component is crucial. To put it into perspective, my first implementation for this component ran in the magnitude of tens of kilo-samples per second. This implementation was a basic implementation of an auto correlator, exactly how someone might try to implement this if they understood the basic math on how correlation works. A starting sample is chosen along with a delay, and a complex correlation is performed. The delay would be the length of the OFDM symbol, and the length of the correlation would be the length of the cyclic prefix. If the result does not reach some threshold, you move to the next sample and repeat the calculation. This initial processing rate is
nowhere close to allowing a receiver to run in real time.

The first idea I had was to save a snapshot of the signal and do the slow correlation on this saved snapshot. If the framing was identified, I could use the time of the initial frame found here and try to approximate where it would line up on the live signal. If the framing was not found a new snapshot would be taken and tried again. This would require a stage where samples are discarded while waiting for the frame detection to slowly identify the alignment, which could get complicated. It would also greatly increase the time it takes to achieve synchronization, so I decided to keep this idea on the back burner as a last resort.

After some more brainstorming, I figured out a way to greatly simplify the calculations needed for each correlation step after the first. Since the delay is the same throughout processing, this means every sample will be correlated against the same delayed sample for each iteration until it falls out of the length of the correlator. If I can properly keep track of some values and save them between iterations, I could take advantage of them to save myself from doing the same sample correlations multiple times. The first correlation will still need to be the full calculation, but once this full calculation is performed once, the calculations for each subsequent iteration are greatly simplified until the framing is identified. The correlation total will be saved along with the normalization factor after each iteration, since I decided to go for a normalized correlator. When moving to the next iteration, the sample correlation that falls out of the correlator can be subtracted from the totals, and then the new sample entering the correlator can be added to these totals. A similar concept is explained in [30]. The main difference is the shifting explained in that reference shifts the delay value, which will be constant in my case. Instead I will be shifting the location of the correlation, and adjusting the math performed based on what a shift would entail when used on the location of the correlation rather than the delay. The total can then be normalized, resulting in the next correlation value. Using this method, the speed of the correlator went to over one billion samples per second, plenty enough for a real time signal. Figure 4.7 shows this mathematical optimization. The red colored pieces in the figure are calculations used in the first correlation, the blue pieces are calculations used in the subsequent calculation. Calculations that are shared between the two iterations are colored purple. All but the first and last samples in the correlation are not shared
between two adjacent correlation calculations, meaning the results from a previous correlation iteration can be reused and, instead of redoing the full correlation, when compensated to account for the sample shift.

Once the initial framing is found, the frame detection can switch to the tracking stage. This is significantly less computationally expensive as the processing can now skip to the next approximate location of the next OFDM symbol and correlate around that area to find the true location. By scanning around the area, offsets such as the Doppler Effect can be accounted for, with larger scans resulting in the ability to recover larger offsets. This ability to skip most of the calculations and move right to the next approximate location is where all of the performance is gained. Now, based on the nature of the optimization, the processing speed over that time will actually be slower than the initial framing detection as the overhead of the first correlation will account for more of the processing time. This doesn’t matter in the grand scheme of things, and is still orders of magnitude faster than the initial implementation would be during this tracking stage, but it is important to note as this will come into effect when measuring the performance of this frame detector in the results section.

Figure 4.7: Auto Correlation Optimization
Based on the nature of this correlator, this code can easily be reused for any detection that requires auto correlation on a constant delay. For implementations where different delays are scanned, for example in applications where distances may be measured, the delay shift method would need to be used instead. The inputs for this correlator are the length of the cyclic prefix and the signal delay, which in this implementation is the length of the OFDM symbol. Since these values are calculated at startup, they are used as inputs for the frame detection, meaning this implementation is already designed to be flexible and fit other signals where auto correlation is used. This means this implementation and technique has more uses than CP-OFDM frame detection, which makes this contribution more significant as there are more applications where this can be used in software.

4.5 OFDM Parameters

Up to this point I have been referring to values and parameters that are calculated upon startup without any explanation as to where or how this is accomplished. In a full implementation a JavaScript Object Notation (JSON) file would be supplied to the software containing all of the parameters needed for operation. This would allow users to create configuration files for the software without needing to recompile or adjust the code. Given the scope and time constraints, this input method was not used, instead a header file with global constants was created. Changing the parameters involves opening this file and adjusting some of the values within and recompiling the code. This file also includes some toggles, one of which is a toggle that switches the simulation to UHD transmit mode, where an Ettus Radio will be initialized and begin transmitting the signal over the air. The complete list of toggles and parameters is as follows:

- **USING_UHD** This is a toggle that tells the code to initialize only the transmitter, and links the drop-off manager to a radio transmit thread. This was created for two reasons, the first as a proof of concept that the signal can be transmitted by an SDR. The second is to capture the spectrum of the transmitted signal and verify the signal looks as expected. For example, if I set the bandwidth of the IFFT to 160MHz, and specify that I want a 5MHz frequency guard on each side of the spectrum, a 150MHz signal on the spectrum analyzer is expected.

43
If this is toggled off, the drop-off manager on the TX PHY will be linked to the RX PHY and run in transceiver software-only simulation.

- **COMPARE** This toggle will enable the bit comparison to verify that the bits going into the TX PHY are what is recovered by the RX PHY, validating that the processing is correct.

- **ADD_VAR** This toggle expands the buffer sizes for the OFDM signal to allow for the testing of the CP-OFDM frame detection. Since the buffers are larger than the OFDM symbol, the transmitter will place it in some specific spot within the buffer and the receiver will search for this location. This validates that the frame detection works and is identifying the correct location of the CP-OFDM symbol.

- **SAMPLE_RATE** This parameter is the sample rate of the signal.

- **SYMBOL_TIME_IN_SAMPLES** This parameter is the length of the OFDM symbol. It also is the length of the IFFT and FFT computation, as those coincide.

- **BITS_PER_SYMBOL** This parameter tells us how many bits each sub-carrier symbol will contain. This is one way to tell what kind of modulation is being used for the subcarriers. Since this implementation only supports QPSK as it stands, this value will always be 2. If BPSK support was added we would be able to change this value down to 1.

- **FREQUENCY_OFFSET_GUARD** This parameter is the frequency guard that will be inserted on each end of the OFDM symbol. This is what will allow us to perform frequency correction, as using the entire range of the IFFT and FFT would mean that any frequency shift would go out of the range of those calculations (and potentially the bandwidth of the hardware). This guard needs to account for the sample rate and not the bandwidth supplied, as they may differ and the bin offset is based on the sample rate.

- **GUARD_INTERVAL** This parameter tells us the cyclic prefix length as a ratio of the OFDM symbol.

- **VARIATION** This parameter tells the software how much extra space to add to the buffers when ADD_VAR is enabled.
This completes the list of parameters that the user is allowed to adjust and set. The following are parameters contained in the same header file that are calculated based on the parameters that were supplied:

- **OFFSET_BINS**  This is the number of bins on each side that will be allocated for the frequency guard.

  \[ OFFSET_BINS = \text{FREQUENCY_OFFSET_GUARD} \times \frac{A}{B} \]

  Where:
  
  A = SYMBOL\_TIME\_IN\_SAMPLES
  B = SAMPLE\_RATE

- **SUBCARRIERS**  This is the number of available subcarriers with the given frequency guard.

  \[ SUBCARRIERS = A \times (2 \times OFFSET_BINS) \]

  Where:
  
  A = SYMBOL\_TIME\_IN\_SAMPLES

- **BANDWIDTH**  This is the bandwidth of the signal given the subcarrier count.

  \[ BANDWIDTH = SUBCARRIERS \times \frac{A}{B} \]

  Where:
  
  A = SAMPLE\_RATE
  B = SYMBOL\_TIME\_IN\_SAMPLES

- **GUARD_INTERVAL_IN_SAMPLES**  This is the cyclic prefix length in samples.

  \[ GUARD_INTERVAL_IN_SAMPLES = A \times B \]
Where:
A = GUARD_INTERVAL
B = SYMBOL_TIME_IN_SAMPLES

• TOTAL_SYMBOL_IN_SAMPLES This is the total length of the CP-OFDM symbol once the cyclic prefix has been inserted.

\[ TOTAL\_SYMBOL\_IN\_SAMPLES = A + B \]

Where:
A = SYMBOL_TIME_IN_SAMPLES
B = GUARD_INTERVAL_IN_SAMPLES

• SYMBOL_RATE This is the overall symbol rate of the signal after the cyclic prefix has been added.

\[ SYMBOL\_RATE = \frac{SAMPLE\_RATE}{TOTAL\_SYMBOL\_IN\_SAMPLES} \]

4.6 Other Implementations

Apart from the main contributions, several other small measures were taken to ensure maximum performance for this PHY. The first important optimization was lowering the precision of the FFTW3 library. When building the library, the default precision (or data type) is double. When the library compiles, all the functions are designed around using doubles. Unless you dig through the documentation for FFTW3, you might not know the precision can be lowered to floats by using the –enable-float compiler flag. Once float precision is enabled, every FFTW3 function that starts with fftw_ can be changed to fftwf_, which tells the library to lower the precision of the computations, which in turn increases the performance of the library. Since the IFFT and FFT computations are the most expensive calculation in the worker threads, this simple adjustment is a big deal. This precision decrease also has little to no side effects as UHD works with 16-bit integers, meaning the 7 significant figures that a float can hold are already more than a 16-bit integer can handle. Another simple but important ”optimization” to the
code was using the -O3 compile flag in GCC. This tells the compiler to use the highest level of optimization when building the code. The compiler will do its best to find patterns in the code and use techniques to increase performance, one of which is using SIMD instructions. This typically results in a longer compile time and a larger executable, but these factors can hardly be considered downsides considering the performance gains such a simple toggle can give us.
Chapter 5

Results

5.1 Signal Analysis

The reliable and easy way to verify signal transmission is to use a spectrum analyzer. The main verification here is to make sure the spectrum looks as expected based on the waveform parameters supplied to the waveform. It is actually a good thing I did this as the first time I measured the spectrum it did not look correct, leading me to finding a bug in how I was setting the bins for the IFFT operation. This is not something that could have been found without this step as my FFT subcarrier recovery was built based off my incorrect implementation of the IFFT bin setting. Another issue found was that I was not scaling up the sample values high enough, i.e. not using enough of the DAC range for the radio, leading to a spike at the center frequency. This spike was presumed to be leakage from the amplifier from within the radio. Regardless of the true cause, using more of the DAC range by scaling the samples up fixed this issue as the power of the signal overtook this peak. Figure 5.1 shows the spectrum for one of the tests. The bandwidth for this setup was 160MHz with a 5MHz frequency guard on both sides, which should result in a 150MHz bandwidth measured on the spectrum analyzer. The capture shown clearly measures 149MHz, some roll off is not being included within the measurement which will account for the last 1MHz to get us to a bandwidth of 150MHz. This confirms that the CP-OFDM TX PHY can in fact correctly process and transmit an OFDM signal in real time on SDR hardware. The setup used to take this measurement is given in figure 5.2.
5.2 CP-OFDM PHY Performance

5.2.1 Introduction

A software only simulation was used to test the PHY in transceiver mode under multiple different conditions. Two systems were used to quantify what kind of performance can be expected with a regular (but higher end) laptop and on a high end custom PC. Multiple IFFT/FFT lengths were tested as a way to demonstrate the performance we can expect under different subcarrier counts. The exact amount of subcarriers that the IFFT/FFT length will map to depends on the frequency guards used. Since this frequency guard will not have a significant effect on performance I will not worry about it. The waveform designer can also choose to use however many of those bins within the FFT length as they please, which would also adjust the subcarrier count, and expect the same performance that was measured here, so subcarrier count will also not be considered in the testing. Since this is a worker based architecture the last parameter will be the worker count used by each PHY. This means a setup with 4 workers will have 4 TX PHY workers and 4 RX
PHY workers, a total of 8 workers in the end to end system. The average sample rate the software only simulation is able to process over some period of time is what will be used as a way to quantify performance. This tells us the maximum sample rate supported without doing any resampling. This seems to be the fairest and most consistent way to measure performance as needlessly upsampling a signal doesn’t properly represent what the GPP can truly process, as upsampling does not increase the throughput of the signal.

5.2.2 Laptop Performance

The first system that was tested on was an i7 laptop. The specs for this system are given in figure 5.3. Since three parameters were used for the

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPP</td>
<td>11th Gen Intel Core i7-11800H @ 2.3GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>Threads</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu20.04LTS</td>
</tr>
<tr>
<td>RAM</td>
<td>16GB</td>
</tr>
<tr>
<td>GCC Version</td>
<td>11.2.0</td>
</tr>
<tr>
<td>CMake Version</td>
<td>3.23.0</td>
</tr>
</tbody>
</table>

Figure 5.3: Laptop Specs
testing, the X axis will represent the worker count for each PHY, the Y axis will display the performance in samples per second, and the legend to sort each line by IFFT/FFT length. Note that "**" represents "to the power of", ex. $2^{**17}$ is a length of 131,072. Lengths with power of 2 were chosen as they are the most efficient sizes that the IFFT and FFT algorithm can work on as it is a divide and conquer algorithm. This is another one of those subtle ways performance was optimized. There is also a length that multiplies a power of 2 by 3, FFTW3 is able to also efficiently work on lengths that have a power of 2 factor along with factors of 3 and 5. That length was chosen to add more resolution as the maximum tested length was reached. Figure 5.4 shows the results. It is clear that even for the very large FFT length of 131,072 the laptop was still able to reach the maximum sample rate of an Ettus X300, with the smaller lengths exceeding more than double the maximum sample rate. This shows that even a GPP on a laptop can process

![Figure 5.4: CP-OFDM PHY Laptop Results](image)

51
OFDM signals with relative ease, with performance leveling at 4 workers per PHY. This makes sense based on the architecture of the system, where there are 8 physical cores hyper-threaded to 16 threads. While the 16 threads allow the system to be in the middle of up to 16 tasks all in parallel, it can only physically execute 8 threads in parallel. The best analogy here is a cashier and a checkout lane. The cashier would be the physical core, and the checkout lane is the thread. When hyper-threading each physical core to two threads, the cashier is receiving two lanes to process. While the cashier will now be checking out two customers at once, they can only physically work on one lane at a time, moving back and forth based on scheduling and demands.

### 5.2.3 Custom PC Performance

A custom PC was also tested to see what can be achieved on a high end system, the specs for this PC are given in figure 5.5. The same parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPP</td>
<td>Intel Core i9-10900KF @ 3.7GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>Threads</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu20.04LTS</td>
</tr>
<tr>
<td>RAM</td>
<td>16GB</td>
</tr>
<tr>
<td>GCC Version</td>
<td>11.1.0</td>
</tr>
<tr>
<td>CMake Version</td>
<td>3.20.0</td>
</tr>
</tbody>
</table>

Figure 5.5: Custom PC Specs

as the laptop test were used along with the method of measurement. The
The main difference here from the laptop results is more workers were tested as the system had more cores available. Interestingly the same pattern was not present on the PC, the performance did not cap once all the physical cores were used. This may be due to the PC being significantly higher performance than the laptop, but the exact reason is not of concern for the scope of this work. This PC performed significantly better than the laptop, reaching processing speeds over 1.5B samples per second. The lowest rate was obviously the longest FFT length, which is expected, but still well exceeded the capabilities of an Ettus X300 with a processing rate of 466M samples per second. This shows that a high performance GPP is very capable of running very high sample rates in a software only implementation.

For reference, the largest IFFT/FFT size used in 5GNR is 4096 (or 2**12), showing that this PHY is not only very capable at this maximum size, but it
can be pushed significantly further and still easily process high sample rates (relative to a high end SDR). This shows that custom OFDM waveforms with high subcarrier counts are feasible in a software only implementation.

This graph does have some clear gaps, the most obvious being the $2^{*12}$ results ending at 7 workers per PHY. When running through all these combinations, some issues within the software were exposed. For some reason these parameter combinations were resulting in segmentation faults. There may be some bug in the code initialization causing these parameter combinations to fail, every other combination that passed the initialization stage was stable. This will be further addressed in the conclusions.

5.3 CP-OFDM Frame Detection

The nature of the frame detection algorithm should be explained further before any results are presented. Since the first correlation requires the complete computation and subsequent correlations reuse calculations from previous results, as the span that the processing speed is averaged over increases, the performance will also seem to increase. In reality after the first expensive correlation, or overhead, the processing speed will remain extremely fast and consistent. Since this overhead is required every time the correlation begins it will be included in the measurements.

Since the initial processing will tend to be over significantly longer lengths of time to locate the framing of the signal, it will appear to run faster during this stage. It needs to be made clear that while yes, the algorithm itself will average a faster processing speed, it will not actually run faster in relation to the overall signal as the tracking will skip significant chunks of the signal and omit a large portion of the processing. This will mean the overhead will play a larger role in the tracking stage speed, but will overall still result in higher performance when the tracking stage processing is looked at in general.

Thanks to the nature of the algorithm, apart from the overhead time, the processing speed of this algorithm will be consistent regardless of the cyclic prefix (i.e. the length of the correlation). Regardless of the cyclic prefix length, one sample will always exit the correlation, and another sample will take its place, meaning the computation, and thus the processing speed, will
be consistent. Testing was performed over several cyclic prefix lengths to verify this theory and it held true. The following results will apply to all cyclic prefix lengths and OFDM symbol lengths, since that will drive the cyclic prefix length.

The following results were measured on the laptop described in figure 5.3. I first started by measuring the performance over a time span of 50 samples, which may be the time span used during the tracking stage. This time span was picked arbitrarily and gives us a decent estimate for how fast we can expect each tracking iteration to take. The overall processing speed in relation to the signal itself will be significantly faster as it will skip larger portions of the signal to get to the estimated location for the next OFDM symbol alignment; it will not be considered for these results as it will vary greatly based on symbol length. When the value is presented it will also be clear why the symbol length does not need to be considered, as the speed is already high enough to process a signal in real time.

The performance was then measured over a span of 60,000 samples. This will be used to give a better estimate as to what performance can be expected during the initial detection stage. Depending on where the frame detection begins in relation to the framing of the signal this time span can vary significantly. Again, due to the nature of this algorithm, the average processing speed will fall between these two values if the time span calculated over also falls between these two time spans. The speed is already sufficient so measuring larger or shorter time spans is unnecessary. The results are as follows:

- **Time span of 50 samples** This span was chosen to simulate processing speed during the tracking stage. Regardless of symbol length and cyclic prefix length the processing speed was consistently around 900M-1000M samples per second, or an overall time of around 50ns to process this time span. Given that the max sample rate of the Ettus X300 is 200M samples per second and the tracking stage will already be skipping most of the incoming samples, this processing speed is more than high enough to process the signal in real time.

- **Time span of 60,000 samples** This span was chosen to simulate processing speed during the initial frame detection stage. Regardless of
symbol length and cyclic prefix length the processing speed was consistently around 750B samples per second, or an overall time of around 80ns to process this time span. This is also more than sufficient to process the incoming signal in real time for an Ettus X300 at the highest possible sample rate.

Based on these results, this optimization has resulted in enough performance to run the frame detection in real time on very high sample rates. While the method for measuring the speed and the calculations do seem correct (as I have not been able to prove them wrong), I am in disbelief when it comes to the processing rate over 60,000 samples. Even if I ignore this result, the slower processing speed for a span of 50 samples is already more than enough, and we know that larger spans will have even faster processing rates. If we were to assume the initial frame detection stage also runs at around 1000M samples per second, based on the concept of duty cycle to measure CPU utilization, only 20 percent CPU utilization on one thread will be needed for initial frame sync on an X300 running at the max sample rate of 200MHz.

5.4 Asynchronous Buffer Manager

5.4.1 Introduction

In terms of the results for the ABM, the correct ordering of the buffers from end to end under many worker counts proves that the implementation has been successful from a logical and managerial standpoint. It now comes time to lay out the expectations for this manager when it comes to performance in relation to the problems it aims to solve, idle time introduced by serial buffer management. To make this comparison as fair as possible, a copy of the ABM was created and the management was refactored to change the buffer passing from asynchronous to serial. This keeps all internal computations and management identical, the main difference is the manager now keeps track of who is the next worker in line to receive a buffer and waits for that worker to make a request, just as the worker would do in the ABM.

This refactor requires one additional parameter upon startup, the worker count. The manager needs to know the worker count so it knows when to roll over the ID for the next processing iteration, and the worker must pass the ID in each interaction with the manager so the manager knows how to
handle the interaction. The worker must identify itself when it interacts with the manager because this manager is request based, so it needs to know who is making the request to correctly pass the buffers in a serial order. If a worker makes a request out of order, which is still allowed, the manager will use a mutex to block the request until it is that worker’s time in line. The blocking of requests so they are executed in order of ID is what changes the management from asynchronous to serial buffer passing, as the requests are now executed in serial order of worker ID rather than FIFO.

5.4.2 Expectations

The expectations are that the ABM will perform better than the serial manager as idle time increases in the serial manager. This idle time is created by variation in the worker processing times, if one worker finishes sooner than the worker preceding it, the faster worker will sit in an idle state until the preceding worker finishes and receives its next buffer first. In the asynchronous implementation this faster worker would send its request for a buffer before the preceding worker and have its request fulfilled since the requests are handled FIFO and not in a serial order, therefore eliminating this idle time.

The processing speed difference is clear if equations are created for the overall processing speed for the worker distribution based on the buffer passing method. I will first start with the asynchronous buffer manager. For the simplicity of these equations I will assume the interaction time between the workers is negligible and strictly focus on the worker processing itself. Since each worker will run independently and have its requests fulfilled immediately, each worker will have its own average processing speed and all the average processing speeds will be summed up to get the overall processing speed of the worker distribution. The formula for this equation is as follows:

$$Overall\, Speed = \sum_{n=0}^{W} p_n$$

Where:
W = the number of workers in the distribution
p = the average processing speed of the worker
Now, for the serial manager, this equation requires some more thinking behind it. Since all manager interactions are assumed to be negligible, this means all workers upon startup are fulfilled instantaneously. The next iteration of processing can only begin once all workers have finished their previous iteration of processing, meaning the slowest worker will drive this time. There will be some slight shift in where this holdup occurs based on which worker is the slowest, but the data passing will always get stuck and wait at this slowest worker, meaning each complete iteration turnover will take as long as the processing time for this worker. This leads us to the equation for this overall processing speed of the serial manager:

\[ \text{OverallSpeed} = \sum_{n=0}^{W} p_m \]

Where:
- \( W \) = the number of workers in the distribution
- \( p \) = the average processing speed of the worker
- \( m \) = the index of the slowest worker

As clearly shown in this equation, the slowest worker will eat up all the extra processing speed of the other workers, which is the idle time of the system. While these equations do not tell us exactly what the performance difference will be, it makes it clear which method is superior. The best case scenario for the serial manager is that the slowest worker in the distribution is equal to all other workers in the distribution. In this case, the equations would be equal, and asynchronous management would provide no performance increase; but it should be noted that it will also not provide any decrease in performance either.

### 5.4.3 Performance

Since the equations given above are theoretical estimations, real measurements should be performed to see how my expectations align with reality, as the interactions with the manager will of course play some (but I doubt much) factor into the processing speed difference along with other factors in the system that cannot be controlled thanks to the operating system. I will also test under different simulation circumstances to give some quantifications as to what the performance difference is in that specific case. As I
noted, I expect the performance to be identical when all the workers process at the same speed, that will be a benchmark for my measurements. This testing is also where the big.LITTLE ARM architecture will finally come back into play, as the ABM was designed with this exact architecture in mind.

The testing simulations will use sleep commands in the workers to simulate processing speed. These sleep times will be randomly generated with parameters adjusting the range of these sleep times. By using randomly generated sleep times, I can reset the seed to regenerate the same values and make measurements for the simulation. The main parameter for these times that will be presented is the coefficient of variation. The coefficient of variation was chosen as it best coincides with the idle time experienced in the serial manager. Since it is a normalization of the variation, it allows us to display what effect, if any, different average processing times will have. Figure 5.7 shows the performance difference between the ABM and the serial refactor. The X axis is the coefficient of variation and the Y axis is the performance difference between the ABM and the serial refactor. A value of 0 means no improvement, and a value of 100 would mean the ABM is 100 percent, or twice as fast as its serial counterpart. The legend is sorted by worker count and processing speed, 4 and 12 workers were tested here. Two different processing speeds were chosen to see what effect the average processing speed will play in terms of the performance difference. As expected, the performance was nearly identical between the two methods when there was no variation between the worker processing, i.e. all workers had the same processing speed. As more idle time is added, an increase of the coefficient of variation, the performance difference increases as predicted. Interestingly, distributions with more workers result in a larger performance difference, this was not something I thought would happen; but I do see it as a good surprise. Simply changing how requests are processed can result in over a 40 percent performance increase on workloads with high variation.

It was also observed that the average processing speed did not affect the performance improvement by much, but shorter processing speeds do offer less of an improvement. This meets my expectations as I figured the interactions with the manager would factor more into the overall processing speed and eat up some of the time saving. Regardless, the performance improvement is similar and can be significant. The most important note here is there is no performance decrease at any point, which makes this manage-
The next test was to simulate the ARM big.LITTLE architecture. This was done by adjusting one of the workers to be a "big" or "LITTLE" worker. For simplicity this means the processing speed, or sleep time, would be scaled based on which designation each worker was given. A "big" worker in this test was signified as a "strong" worker and the sleep time would be halved to simulate this worker being twice as fast as the other workers. A "LITTLE" worker would be signified as a "weak" worker in this test and the sleep time would be doubled to simulate this worker being twice as slow as the rest of the workers. Figure 5.8 shows these results. Since the previous testing demonstrated that the average processing speed did not affect the performance difference by much, it was eliminated from this test to simplify the presentation.

This testing is where the asynchronous management shows its strengths. Again the performance difference increases with the coefficient of variation.
Figure 5.8: ABM vs Serial Manager Performance Difference, ARM big.LITTLE Simulation

but now even no variation in workload sees a significant improvement as
the workers themselves add variation thanks to their differences in speed.
The largest performance difference is seen when there is one weak worker,
this worker will drive the serial management processing speed whereas it will
play no effect on the asynchronous management, to the tune of an immediate
50 percent improvement. The performance improvement is immediate when
there is one stronger worker in the distribution, a larger difference predictably
occurring with less cores.

These results show us why actual measurements are needed, as the results
for the strong worker, while still an improvement, are not as large as the
theoretical difference. The theoretical performance increase is 25 percent,
whereas the actual difference was only around 15 percent. The calculation
for this prediction is shown below:

$$\text{Increase} = \frac{x + x + x + 2x}{x + x + x + x} = \frac{5}{4} = .25$$
Where:
\( x \) = the processing speed of the workers
\( 2x \) = the processing speed of the strong worker

I expected this value to be less than 25 percent due to the manager interaction overhead time, but this was significantly more than expected. The prediction here is that the longer the processing speed is for each worker, due to the observation in figure 5.7, it is expected that this improvement will converge towards the theoretical value of 25 percent.

Despite the caveats and differences between the expected results and the actual results, it has been proven that in every test case, the ABM is at worst as good as serial management. The clear upside to using the ABM is it has been shown to result in significant performance improvements with a simulated ARM big.LITTLE architecture and high worker workload variation, to the tune of well over 50 percent faster. The increase in performance experienced will be based on the worker distribution and architecture it is implemented on, and can be predicted if some parameters of the cores and workers are known; we can do our best to map them to these results, or create a simulation here inputing the parameters for the specific system we want to get an estimation for.
Chapter 6

Conclusions

6.1 Accomplishments

This work has shown that a software only OFDM PHY implementation can boast very high performance on a GPP if done correctly, well exceeding 1B samples per second under certain parameters on a high performance PC. To the best of my knowledge this is the highest performance software only OFDM PHY transceiver. Given the architecture of today’s GPPs, a worker thread architecture is absolutely necessary along with several other techniques and optimizations to allow for such performance. It also shows how quick software development can be, as the majority of the time was spent researching and documenting my work, the actual software development took only around 4-5 weeks starting from scratch. Given my experience and the ability to reuse most of this code for any other OFDM based waveform, an OFDM PHY similar to this can easily be accomplished in 3 weeks or less, giving more time to develop the rest of the waveform components. This time span is significantly shorter than the months it would take to make an equivalent implementation in FPGA. The development also used all open source and free tools, meaning my cost to develop this software was $0.

Apart from my overall goal being reached of maxing out the hardware capabilities of an Ettus X300, my contributions are also general enough to be applied to many more software applications. The ABM can be used to optimize existing worker thread architectures, especially on ARM big.LITTLE architecture. If a developer needs to use different data types and buffers,
they can use whatever data structures they need as the ABM has been implemented as a template. If more than one buffer is needed, along with other metadata, a data structure can be created and passed in as the template.

The OFDM frame detection can also be implemented into any other waveform that requires auto correlation with constant delay. The input does not need to be an OFDM signal for this frame detection to work. As it stands, it does require the input to be the native UHD driver sample type, complex 16-bit integers. This can be converted to a template as future work if a developer requires a different data type. It should be noted that fixed point should be used for this correlator as rounding errors will add up if floating point is used. How much of an impact these rounding errors will have has not been determined.

The OFDM workers also have high reuse, as they can easily be adjusted if a waveform has slightly different specs from what was used in this work. Changing the QPSK modulation for the subcarriers has been used as an example several times now, the function calls to modulate and demodulate could be swapped out with a new modulation type. This goes for all other aspects of the workers as well, for example the cyclic prefix insertion can be removed and DFT-s-OFDM can be attempted instead.

### 6.2 Shortcomings

While my goals have been reached and the work overall has been successful, there are many shortcomings for this work. A complete PHY would contain more components than what was accomplished here. None of these directly relate to OFDM so they weren’t a concern when it comes to presenting the performance that was achievable. For example, resampling and filtering were mentioned, neither of these were implemented into the PHY, even though a complete PHY would certainly contain these two components. Frequency and phase correction were also not implemented in the PHY, two crucial components for recovering a received signal. Another component is error correction. Unfortunately the time frame for this work did not allow for any of these components to be researched and implemented.

While these components will affect the overall performance of the wave-
form, as new GPPs are released the performance this software will be able to achieve will end up crossing out the extra computation needed by those components. Since those PHY components are not specific to OFDM, they have also already been implemented in existing waveforms by ANDRO, meaning that it is known that these components can efficiently be implemented in software and executed in real time, as they already have been. The current performance is so much higher than the Ettus X300 hardware supports that there is plenty of processing power left for these components, so that is not a concern. Of course these should eventually be implemented and the performance should be officially documented, as it will be lower than the results presented here for a complete waveform; the exact amount is important to know.

Another shortcoming is some issues that still exist in the software. This was seen in some of the presented results, some combinations of parameters caused a segmentation fault, which would need to be resolved in a production release. There is also a double free crash that sometimes occurs when the waveform shuts down. This does not affect the waveform while it is running, but would also need to be resolved in an official release. My guess is there is a bug with some of the FFTW3 calls I made that are not thread safe, as it is a FFTW3 free call that causes this crash. Since neither of these issues affect the results once the software has started and is running, they were not mentioned in the results and are not a huge concern in relation to what this thesis wanted to accomplish and present.

Another shortcoming is the normalization of the CP-OFDM frame detection. The results are currently not being normalized properly, meaning the results are not between 0 and 1. This means I had to increase the threshold and analyze the values coming from the detection to create a reasonable threshold for detection. Regardless, the peak of the correlation is in the correct location, meaning it does identify the framing of the signal, which is what truly matters. Normalization is not specifically necessary for frame detection anyways, and many times not even used, but it is my goal as it makes it easier to define a threshold for the frame detection. A better implementation might be able to adjust this threshold dynamically, making this normalization unnecessary. This is another one of those issues that were not addressed due to the priority of the issue along with the given time frame.
6.3 Future Work

There is more future work that can be done for this project, this work has barely scratched the surface as it only gives a glimpse into what kind of performance is possible on a GPP. For one, all other components that would be included in a complete PHY would be implemented and used to remeasure the performance that can be achieved. The most difficult and computationally expensive component would be error correction, as identified in [4]. Some error correction algorithms are complicated enough that their software implementation and optimization could be a thesis in itself. Of course this depends on what error correction algorithm is used, one could spend forever trying to implement and test every possible algorithm. The bugs that still exist in the software that were mentioned in the previous section should also be troubleshot and fixed.

Relating to completing the PHYs, the receiver also would need to have a thread created to receive samples from the UHD driver to support an SDR. Only the transmitter has this implemented currently due to an easier implementation as samples are simply being pushed to the driver. To receive samples, additional buffering would need to be implemented to allow for the collection of samples and prevent the UHD driver from overrunning. Tying right into adding radio support to the receiver is adding more radio support to the PHYs in general. Right now this software is only built to support UHD devices. Additional software can be written to allow for interfacing between more sets of SDR hardware, making the software even more flexible. This is necessary if we wish to run signals at sample rates higher than possible on the X300. The Motorola NS-1 has its own driver that the software would need to interface with to test these higher rates.

The frame detection also needs to have its stages implemented. The testing and performance measurements were done with different sample spans rather than measuring the actual stages as the handling for that was not created. As far as measuring performance, the implementation of these states is not necessary, but to allow a MAC to control this frame detection when a signal is running through the receiver in real time, this control would need to exist. This implementation would make more sense once the receiver could process an incoming signal for a radio, it would not make sense to spend time on this work until that is completed first when it comes to the overall development.
There is also still more optimization that could be done to squeeze out more performance, mainly AVX SIMD optimization. This was not attempted as the -O3 flag does a decent enough job at optimizing the code already, but manually writing the SIMD code yourself, as you know the exact processing and patterns for what you are trying to accomplish, will typically yield higher performance than the compiler.

Since I have only looked at a software implementation of a PHY, the obvious next step would be to implement a MAC in software to control this PHY. This would allow us to run this software as a complete waveform and move data across it. A specification would need to be defined for this work as well.

Another test would be the replace the sleep times in the buffer management comparison test to use real processing and perform the test on different ARM systems to get real life results rather than attempting to make estimations through simulations. I attempted to build the test software on my phone to run this test but was not successful. This would be interesting to look further into, along with running the OFDM PHY on an ARM system to document what an ARM device is capable of.
Bibliography


Appendix A

Code Snippets

A.1 Main

Source Code

```
#include <iostream>
#include <chrono>
#include <cassert>
#include <cstdlib>
#include <tx.hpp>
#include <rx.hpp>
#include <constants.hpp>

#define ITERATIONS 30000

void printUsage()
{
    std::cout<<"Usage: ofdm [workers per frontend]\n";
}

int main(int argc, char* argv[])
{
    if(argc != 2)
    {
        printUsage();
```
return 0;
}
std::size_t workers = strtol(argv[1], NULL, 10);
std::cout << "Using \"workers\" workers\n";
std::cout << SUBCARRIERS << " subcarriers\n";
std::cout << "Symbol Rate: \"SYMBOL_RATE\"\n";
std::cout << "Bandwidth: \"BANDWIDTH\"\n";
MultiDropoffBuffer<OTABUFFER> ota(workers*3);

auto txPhy = Tx::create(workers, &ota);
#if not USING_UHD
auto rxPhy = Rx::create(workers, &ota);
#endif
auto start = std::chrono::steady_clock::now();
 txPhy->start();
#if not USING_UHD
 rxPhy->start();
#endif
 txPhy->wait();
#if not USING_UHD
 rxPhy->wait();
#endif
 auto end = std::chrono::steady_clock::now();

auto modRate = static_cast<double>((SYMBOL_TIME_IN_SAMPLES + GUARD_INTERVAL_IN_SAMPLES) * ITERATIONS) / (std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count() / 1e9);
std::cout << "Overall: \"modRate\"\n";
}

A.2 PHY Parameters

#ifndef constants_hpp
#define constants_hpp

299

73

#endif

#define constants_hpp


#include <cstdlib>
#include <mutex>

#define USING_UHD 0
#define COMPARE 0
#define ADD_VAR 0

const std::size_t SAMPLE_RATE = 200e6; // max rate of X300
const std::size_t SYMBOL_TIME_IN_SAMPLES = 32768;
const std::size_t BITS_PER_SYMBOL = 2; // QPSK
const std::size_t FREQUENCY_OFFSET_GUARD = 25e6; // 25e6 in both
directions, for ~150MHz BW, 160MHz is max on X300, giving
+5MHz guard for freq correction
constexpr std::size_t OFFSET_BINS =
    static_cast<double>(FREQUENCY_OFFSET_GUARD)/
    (static_cast<double>(SAMPLE_RATE)/
    static_cast<double>(SYMBOL_TIME_IN_SAMPLES));
constexpr std::size_t SUBCARRIERS =
    SYMBOL_TIME_IN_SAMPLES-2*OFFSET_BINS;
constexpr double BANDWIDTH =
    SUBCARRIERS*
    (static_cast<double>(SAMPLE_RATE)/
    static_cast<double>(SYMBOL_TIME_IN_SAMPLES));
constexpr double GUARD_INTERVAL = 1.0/32.0;
constexpr std::size_t GUARD_INTERVAL_IN_SAMPLES =
    static_cast<std::size_t>(
    static_cast<double>(SYMBOL_TIME_IN_SAMPLES)*GUARD_INTERVAL);
constexpr std::size_t TOTAL_SYMBOL_IN_SAMPLES =
    GUARD_INTERVAL_IN_SAMPLES+SYMBOL_TIME_IN_SAMPLES;
constexpr double SYMBOL_RATE =
    static_cast<double>(SAMPLE_RATE) /
    static_cast<double>(TOTAL_SYMBOL_IN_SAMPLES);

static std::mutex FFTW_LOCK;
const std::size_t VARIATION = 50;

#if ADD_VAR
    using OTABUFFER =
        std::array<std::complex<int16_t>,TOTAL_SYMBOL_IN_SAMPLES+VARIATION>;
#else

74
using OTABUFFER = 
    std::array<std::complex<int16_t>, TOTAL_SYMBOL_IN_SAMPLES>;
#endif
#endif

A.3 Asynchronous Buffer Manager

A.3.1 Pick-up Manager

Header File

#ifndef multiPickupBuffer_hpp
#define multiPickupBuffer_hpp

#include <mutex>
#include <vector>
#include <memory>
#include <atomic>
#include <cstdint>
#include <iostream>

/*
   Class that creates a block of buffers of type T that can be
   passed one way between threads
   Allows for thread safe data segment passing
*/

template <typename T>
class MultiPickupBuffer
{

public:
    struct Md {
        T& buffer;
        std::size_t id;
    };
    MultiPickupBuffer(uint8_t buffer_count);
    ~MultiPickupBuffer();

};

75
Md getRead(std::size_t workerId)
{
    getReadLock.lock();
    std::size_t current = currentGetReadBuffer%block.size();
    unlockId = workerId;
    workerMetaData[workerId] = currentGetReadBuffer;
    readLock[current]->lock();
    ++currentGetReadBuffer;
    --readyRead;
    return {block[current],currentGetReadBuffer-1};
}

void unlockRead(std::size_t workerId);
T& getWrite();
void passRead(std::size_t workerId);
void passWrite();
void passWrite(T& x);
typename std::vector<T>::iterator begin();
typename std::vector<T>::iterator end();
int64_t numReadyRead();
int64_t numReadyWrite();

private:
    using mutexPtr = std::unique_ptr<std::mutex>;
    std::vector<T> block;
    std::vector<mutexPtr> writeLock;
    std::vector<mutexPtr> readLock;
    std::mutex getReadLock, passReadLock;
    std::size_t currentGetReadBuffer = 0;
    std::size_t currentWriteBuffer = 0;
    std::size_t unlockId;
    uint8_t readyWrite;
    std::atomic<uint8_t> readyRead;
    std::vector<std::size_t> workerMetaData;
};

#include <template/multiPickupBuffer.cpp>

#endif
template <typename T>
MultiPickupBuffer<T>::MultiPickupBuffer(uint8_t bufferCount)
{
    workerMetaData.resize(bufferCount);
    block.resize(bufferCount);
    for(int16_t n=0;n<bufferCount;++n)
    {
        writeLock.push_back(mutexPtr(new std::mutex()));
        readLock.push_back(mutexPtr(new std::mutex()));
        readLock[n]->lock();
    }
}

template <typename T>
MultiPickupBuffer<T>::~MultiPickupBuffer(){}

template <typename T>
T& MultiPickupBuffer<T>::getWrite()
{
    writeLock[currentWriteBuffer]->lock();
    --readyWrite;
    return block[currentWriteBuffer];
}

template <typename T>
void MultiPickupBuffer<T>::unlockRead(std::size_t workerId)
{
    if(workerId == unlockId) getReadLock.unlock();
}

template <typename T>
void MultiPickupBuffer<T>::passRead(std::size_t workerId)
{
    passReadLock.lock();
    std::size_t current = workerMetaData[workerId]%block.size();
    writeLock[current]->unlock();
    ++readyWrite;
    passReadLock.unlock();
}
template<typename T>
void MultiPickupBuffer<T>::passWrite()
{
    readLock[currentWriteBuffer]->unlock();
    ++readyRead;
    currentWriteBuffer = (currentWriteBuffer+1)%block.size();
}

template<typename T>
void MultiPickupBuffer<T>::passWrite(T& x)
{
    auto& y = getWrite();
    std::swap(x,y);
    passWrite();
}

template<typename T>
typename std::vector<T>::iterator MultiPickupBuffer<T>::begin()
{
    return block.begin();
}

template<typename T>
typename std::vector<T>::iterator MultiPickupBuffer<T>::end()
{
    return block.end();
}

template<typename T>
int64_t MultiPickupBuffer<T>::numReadyRead()
{
    return readyRead;
}

template<typename T>
int64_t MultiPickupBuffer<T>::numReadyWrite()
{
    return readyWrite;
}
A.3.2 Drop-off Manager

Header File

ifndef multiDropoffBuffer_hpp
define multiDropoffBuffer_hpp

#include <mutex>
#include <vector>
#include <memory>
#include <atomic>
#include <cstdint>
#include <iostream>
#include <queue>
#include <thread>
#include <chrono>

/*
   Class that creates a block of buffers of type T that can be
   passed one way between threads
   Allows for thread safe data segment passing
*/

template <typename T>
class MultiDropoffBuffer
{

public:
    struct Md {
        T& buffer;
        std::size_t id;
        std::size_t bufferIndex; //to tie id to index
    };
    struct ReadMd {
        std::size_t index; //index tied to id
        std::size_t id;
    };
    MultiDropoffBuffer(uint8_t bufferCount);
    ~MultiDropoffBuffer();
    Md getRead();
};
while(!checkForNextId()){
    std::this_thread::sleep_for(std::chrono::microseconds(1));
}
currentReadIndex = consumeNextId();
readLock[currentReadIndex]->lock();
--readyRead;
return
    {block[currentReadIndex],nextReadId,currentReadIndex};
}
Md getWrite(std::size_t id)
{
    while (id >= currentMinimumId+block.size()){
        std::this_thread::sleep_for(std::chrono::microseconds(1));
    }
    getWriteLock.lock();
    while (availableWriteBufferIndicies.size() < 1)
    {
        std::this_thread::sleep_for(std::chrono::microseconds(1));
    }
    auto index = availableWriteBufferIndicies.front();
    unlockId = id;
    writeLock[index]->lock();
    --readyWrite;
    availableWriteBufferIndicies.pop();
    return {block[index],id,index};
}
void unlockWrite(std::size_t id);
void passRead();
void passWrite(std::size_t id, std::size_t index);
typename std::vector<T>::iterator begin();
typename std::vector<T>::iterator end();
int64_t numReadyRead();
in64_t numReadyWrite();

private:
    using mutex_ptr = std::unique_ptr<std::mutex>;
    std::size_t currentMinimumId = 0;
    std::size_t bufferCount;

    std::queue<std::size_t> availableWriteBufferIndicies;
std::vector<mutex_ptr> writeLock;
std::vector<mutex_ptr> readLock;

std::mutex readyReadBufferIndiciesLock;
std::vector<ReadMd> readyReadBufferIndicies;
std::size_t nextReadId = 0;
std::size_t unlockId;
std::size_t currentReadIndex;

std::mutex getWriteLock, passWriteLock;

std::vector<T> block;

uint8_t readyWrite;
std::atomic<uint8_t> readyRead;

bool checkForNextId();
std::size_t consumeNextId();
};

#include <template/multiDropoffBuffer.cpp>

#endif

Source Code

template <typename T>
MultiDropoffBuffer<T>::MultiDropoffBuffer(uint8_t bufferCount)
{
    block.resize(bufferCount);
    for(int16_t n=0;n<bufferCount;++n)
    {
        availableWriteBufferIndicies.push(n);
        writeLock.push_back(mutex_ptr(new std::mutex()));
        readLock.push_back(mutex_ptr(new std::mutex()));
        readLock[n]->lock();
    }
}

template <typename T>

MultiDropoffBuffer<T>::~MultiDropoffBuffer() {}

template <typename T>
bool MultiDropoffBuffer<T>::checkForNextId() {
    for(std::size_t n=0;n<readyReadBufferIndicies.size();++n) {
        if (readyReadBufferIndicies[n].id == nextReadId) {
            return true;
        }
    }
    return false;
}

template <typename T>
std::size_t MultiDropoffBuffer<T>::consumeNextId() {
    readyReadBufferIndiciesLock.lock();
    std::size_t ret = 0;
    for(std::size_t n=0;n<readyReadBufferIndicies.size();++n) {
        if (readyReadBufferIndicies[n].id == nextReadId) {
            ret = readyReadBufferIndicies[n].index;
            readyReadBufferIndicies.erase(readyReadBufferIndicies.begin()+n);
        }
    }
    readyReadBufferIndiciesLock.unlock();
    return ret;
}

template <typename T>
void MultiDropoffBuffer<T>::unlockWrite(std::size_t id) {
    if(id == unlockId) getWriteLock.unlock();
}

template <typename T>
void MultiDropoffBuffer<T>::passRead()
{  
    ++nextReadId;
    writeLock[currentReadIndex]->unlock();
    ++readyWrite;
    ++currentMinimumId;
    availableWriteBufferIndicies.push(currentReadIndex);
}

template <typename T>
void MultiDropoffBuffer<T>::passWrite(std::size_t id, std::size_t index)  
{  
    passWriteLock.lock();
    readLock[index]->unlock();
    readyReadBufferIndiciesLock.lock();
    readyReadBufferIndicies.push_back({index, id});
    readyReadBufferIndiciesLock.unlock();
    ++readyRead;
    passWriteLock.unlock();
}

template <typename T>
typename std::vector<T>::iterator MultiDropoffBuffer<T>::begin()  
{ return block.begin(); }

template <typename T>
typename std::vector<T>::iterator MultiDropoffBuffer<T>::end()  
{ return block.end(); }

template <typename T>
int64_t MultiDropoffBuffer<T>::numReadyRead()  
{  
    return readyRead;
}

template <typename T>
int64_t MultiDropoffBuffer<T>::numReadyWrite()  
{  
    return readyWrite;
}
A.3.3 Pick-up Manager Serial Refactor

Header File

```cpp
#ifndef serialPickupBuffer_hpp
#define serialPickupBuffer_hpp

#include <mutex>
#include <vector>
#include <memory>
#include <atomic>
#include <cstdint>
#include <iostream>

/*
 * Class that creates a block of buffers of type T that can be passed one way between threads
 * Allows for thread safe data segment passing
 */

template <typename T>
class SerialPickupBuffer
{

public:

  struct Md {
    T& buffer;
    std::size_t id;
  };

  SerialPickupBuffer(uint8_t buffer_count, std::size_t workerCount);
  ~SerialPickupBuffer();

  Md getRead(std::size_t workerId)
  {
    workerSerialAccess[workerId]->lock(); // must be your turn in line
    std::size_t current = currentGetReadBuffer%block.size();
    unlockId = workerId;
    workerMetaData[workerId] = currentGetReadBuffer;
    readLock[current]->lock(); // buffer must be ready
    ++currentGetReadBuffer;
  }

};
```

84
template <typename T>  
SerialPickupBuffer<T>::SerialPickupBuffer(uint8_t bufferCount,  
   std::size_t workerCount) :  
workerCount(workerCount)  
}
workerMetaData.resize(workerCount);
block.resize(bufferCount);
for(int16_t n=0;n<bufferCount;++n)
{
    writeLock.push_back(mutexPtr(new std::mutex()));
    readLock.push_back(mutexPtr(new std::mutex()));
    readLock[n]->lock();
}
for(std::size_t n=0;n<workerCount;++n)
{
    workerSerialAccess.push_back(mutexPtr(new std::mutex()));
    workerSerialAccess[n]->lock();
}
workerSerialAccess[0]->unlock();//first worker can now read

template <typename T>
SerialPickupBuffer<T>::~SerialPickupBuffer(){}

template <typename T>
T& SerialPickupBuffer<T>::getWrite()
{
    writeLock[currentWriteBuffer]->lock();
    --readyWrite;
    return block[currentWriteBuffer];
}

template <typename T>
void SerialPickupBuffer<T>::unlockRead(std::size_t workerId)
{
    //next worker now has access, hence what makes this serial
    if(workerId == unlockId)
        workerSerialAccess[(workerId+1)%workerCount]->unlock();
}

template <typename T>
void SerialPickupBuffer<T>::passRead(std::size_t workerId)
{
    passReadLock.lock();
}
std::size_t current = workerMetaData[workerId]%block.size();
writeLock[current]->unlock();
++readyWrite;
passReadLock.unlock();
}

template <typename T>
void SerialPickupBuffer<T>::passWrite()
{
    readLock[currentWriteBuffer]->unlock();
    ++readyRead;
    currentWriteBuffer = (currentWriteBuffer+1)%block.size();
}

template <typename T>
void SerialPickupBuffer<T>::passWrite(T& x)
{
    auto& y = getWrite();
    std::swap(x,y);
    passWrite();
}

template <typename T>
typename std::vector<T>::iterator SerialPickupBuffer<T>::begin()
{
    return block.begin();
}

template <typename T>
typename std::vector<T>::iterator SerialPickupBuffer<T>::end()
{
    return block.end();
}

template <typename T>
int64_t SerialPickupBuffer<T>::numReadyRead()
{
    return readyRead;
}

template <typename T>
int64_t SerialPickupBuffer<T>::numReadyWrite()
{
    return readyWrite;
}
A.3.4 Drop-off Manager Serial Refactor

Header File

```cpp
#ifndef serialDropoffBuffer_hpp
#define serialDropoffBuffer_hpp

#include <mutex>
#include <vector>
#include <memory>
#include <atomic>
#include <cstdint>
#include <iostream>
#include <queue>
#include <thread>
#include <chrono>

/*
   Class that creates a block of buffers of type T that can be
   passed one way between threads
   Allows for thread safe data segment passing
*/

template <typename T>
class SerialDropoffBuffer
{

public:
    struct Md {
        T& buffer;
        std::size_t id;
        std::size_t bufferIndex; //to tie id to index
    };
    struct ReadMd {
        std::size_t index; //index tied to id
        std::size_t id;
    };

};
```
SerialDropoffBuffer(uint8_t bufferCount, std::size_t
  workerCount);
~SerialDropoffBuffer();
Md getRead()
{
    while(!checkForNextId())
      std::this_thread::sleep_for(std::chrono::microseconds(1));
    currentReadIndex = consumeNextId();
    readLock[currentReadIndex]->lock();
    --readyRead;
    return
      {block[currentReadIndex],nextReadId,currentReadIndex};
}
Md getWrite(std::size_t workerId)
{
    workerSerialAccess[workerId]->lock(); //must be your turn
    in line
    getWriteLock.lock();
    unlockId = workerId;
    currentWriteBufferId++;
    while (availableWriteBufferIndicies.size() < 1)
      {
      std::this_thread::sleep_for(std::chrono::microseconds(1));
      }
    auto index = availableWriteBufferIndicies.front();
    writeLock[index]->lock();
    --readyWrite;
    availableWriteBufferIndicies.pop();
    getWriteLock.unlock();
    Md ret = {block[index],currentWriteBufferId-1,index};
    return ret;
}
void unlockWrite(std::size_t workerId);
void passRead();
void passWrite(std::size_t id, std::size_t index);
typename std::vector<T>::iterator begin();
typename std::vector<T>::iterator end();
typename std::vector<T>::iterator end();
int64_t numReadyRead();
int64_t numReadyWrite();

private:
using mutexPtr = std::unique_ptr<std::mutex>;
std::size_t currentMinimumId = 0;
std::size_t bufferCount;

std::queue<std::size_t> availableWriteBufferIndicies;
std::vector<mutexPtr> writeLock;
std::vector<mutexPtr> readLock;
std::vector<mutexPtr> workerSerialAccess;

std::mutex readyReadBufferIndiciesLock;
std::vector<ReadMd> readyReadBufferIndicies;
std::size_t nextReadId = 0;
std::size_t currentWriteBufferId = 0;
std::size_t workerCount;
std::size_t unlockId;
std::size_t currentReadIndex;

std::mutex getWriteLock, passWriteLock;

std::vector<T> block;

uint8_t readyWrite;
std::atomic<uint8_t> readyRead;

bool checkForNextId();
std::size_t consumeNextId();
};

#include <template/serialDropoffBuffer.cpp>

#endif

Source Code

template <typename T>
SerialDropoffBuffer<T>::SerialDropoffBuffer(uint8_t bufferCount,
    std::size_t workerCount) :
workerCount(workerCount)
{
    block.resize(bufferCount);
for(int16_t n=0;n<bufferCount;++n)
{
    availableWriteBufferIndicies.push(n);
    writeLock.push_back(mutexPtr(new std::mutex()));
    readLock.push_back(mutexPtr(new std::mutex()));
    readLock[n]->lock();
}
for(std::size_t n=0;n<workerCount;++n)
{
    workerSerialAccess.push_back(mutexPtr(new std::mutex()));
    workerSerialAccess[n]->lock();
}
workerSerialAccess[0]->unlock();//first worker can now write

template <typename T>
SerialDropoffBuffer<T>::~SerialDropoffBuffer(){}

template <typename T>
bool SerialDropoffBuffer<T>::checkForNextId()
{
    for(std::size_t n=0;n<readyReadBufferIndicies.size();++n)
    {
        if (readyReadBufferIndicies[n].id == nextReadId)
        {
            return true;
        }
    }
    return false;
}

template <typename T>
std::size_t SerialDropoffBuffer<T>::consumeNextId()
{
    readyReadBufferIndiciesLock.lock();
    std::size_t ret = 0;
    for(std::size_t n=0;n<readyReadBufferIndicies.size();++n)
    {
        if (readyReadBufferIndicies[n].id == nextReadId)
        {
            ret = n;
        }
    }
    return ret;
}
template <typename T>
void SerialDropoffBuffer<T>::passRead()
{
    ++nextReadId;
    writeLock[currentReadIndex]->unlock();
    ++readyWrite;
    ++currentMinimumId;
    availableWriteBufferIndicies.push(currentReadIndex);
}

template <typename T>
void SerialDropoffBuffer<T>::unlockWrite(std::size_t workerId)
{
    //next worker now has access, hence what makes this serial
    if(workerId == unlockId)
        workerSerialAccess[(workerId+1)%workerCount]->unlock();
}

template <typename T>
void SerialDropoffBuffer<T>::passWrite(std::size_t bufferId,
                               std::size_t index)
{
    passWriteLock.lock();
    readLock[index]->unlock();
    readyReadBufferIndiciesLock.lock();
    readyReadBufferIndicies.push_back({index,bufferId});
    readyReadBufferIndiciesLock.unlock();
    ++readyRead;
    passWriteLock.unlock();
}

template <typename T>
A.3.5 Performance Comparison Test

/*

Test to compare performance between async and serial buffer passing
Also allows you to simulate a "weak worker" to simulate what might
happen with big.LITTLE arm architecture
Where some cores are purposely low energy consumption - low
performance

*/

#include <iostream>
#include <chrono>
#include <cassert>
#include <cmath>

#include <multiPickupBuffer.hpp>
#include <multiDropoffBuffer.hpp>
#include <serialPickupBuffer.hpp>
#include <serialDropoffBuffer.hpp>

//#define ENABLE_WEAK_WORKER
//#define ENABLE_STRONG_WORKER
#define TEST_ITERATIONS 10000

std::size_t workerCount = 0;
std::size_t baseSpeed = 0;
std::size_t baseVariation = 0;
std::atomic<std::size_t> iterationCount = 0;

void printUsage()
{
  std::cout<<"Usage: bufferComp [workerCount] [baseSpeedUs]
               [maxVariationUs]\n";
}

void multiThreadFcn(std::size_t workerId,
                    MultiPickupBuffer<std::array<int,10>>& pickup,
                    MultiDropoffBuffer<std::array<int,10>>& dropoff)
{
  while((iterationCount++)<(TEST_ITERATIONS*workerCount))
  {
    auto pickMd = pickup.getRead(workerId);
    pickup.unlockRead(workerId);
    pickup.passRead(workerId);

    std::size_t amount = baseSpeed+rand()%baseVariation;
    #ifdef ENABLE_WEAK_WORKER
    if (workerId == 0) amount *= 2;
    #endif
    #ifdef ENABLE_STRONG_WORKER
    if (workerId == 0) amount /= 2;
    #endif
    std::this_thread::sleep_for(std::chrono::microseconds(amount));

    auto dropMd = dropoff.getWrite(pickMd.id);
    dropoff.unlockWrite(dropMd.id);
    dropoff.passWrite(dropMd.id, dropMd.bufferIndex);
void serialThreadFcn(std::size_t workerId,
    SerialPickupBuffer<std::array<int,10>>& pickup,
    SerialDropoffBuffer<std::array<int,10>>& dropoff)
{
    for(std::size_t n=0;n<TEST_ITERATIONS;++n)
    {
        auto pickMd = pickup.getRead(workerId);
        static_cast<void>(pickMd);
        pickup.unlockRead(workerId);
        pickup.passRead(workerId);

        std::size_t amount = baseSpeed+rand()%baseVariation;
        #ifdef ENABLE_WEAK_WORKER
            if (workerId == 0) amount *= 2;
        #endif
        #ifdef ENABLE_STRONG_WORKER
            if (workerId == 0) amount /= 2;
        #endif
        std::this_thread::sleep_for(std::chrono::microseconds(amount));

        auto dropMd = dropoff.getWrite(workerId);
        dropoff.unlockWrite(workerId);
        dropoff.passWrite(dropMd.id, dropMd.bufferIndex);
    }
}

void multiWrite(std::size_t workerCount,
    MultiPickupBuffer<std::array<int,10>>& pickup)
{
    for(std::size_t n=0;n<TEST_ITERATIONS*workerCount;++n)
    {
        pickup.getWrite();
        pickup.passWrite();
    }
}
void multiConsume(std::size_t workerCount,
    MultiDropoffBuffer<std::array<int,10>>& dropoff)
{
    for(std::size_t n=0;n<TEST_ITERATIONS*workerCount;++n)
    {
        dropoff.getRead();
        dropoff.passRead();
    }
}

void serialWrite(std::size_t workerCount,
    SerialPickupBuffer<std::array<int,10>>& pickup)
{
    for(std::size_t n=0;n<TEST_ITERATIONS*workerCount;++n)
    {
        pickup.getWrite();
        pickup.passWrite();
    }
}

void serialConsume(std::size_t workerCount,
    SerialDropoffBuffer<std::array<int,10>>& dropoff)
{
    for(std::size_t n=0;n<TEST_ITERATIONS*workerCount;++n)
    {
        dropoff.getRead();
        dropoff.passRead();
    }
}

int main(int argc, char* argv[])
{
    if(argc != 4)
    {
        printUsage();
        return 0;
    }
    workerCount = strtol(argv[1], NULL, 10);
    baseSpeed = strtol(argv[2], NULL, 10);
    baseVariation = strtol(argv[3], NULL, 10);
MultiPickupBuffer<std::array<int,10>> mpb(workerCount*3);
MultiDropoffBuffer<std::array<int,10>> mdb(workerCount*3);
SerialPickupBuffer<std::array<int,10>> spb(workerCount*3, workerCount);
SerialDropoffBuffer<std::array<int,10>> sdb(workerCount*3, workerCount);

srand(1); //ensure both tests get exactly the same sleep times
auto startM = std::chrono::steady_clock::now();
std::thread workersM[workerCount];
std::thread writerM(&multiWrite, workerCount, std::ref(mpb));
std::thread consumerM(&multiConsume, workerCount,
    std::ref(mdb));
for(std::size_t n=0; n<workerCount; ++n) {
    workersM[n] = std::thread(&multiThreadFcn, n,
        std::ref(mpb), std::ref(mdb));
}
writerM.join();
for(std::size_t n=0; n<workerCount; ++n) workersM[n].join();
consumerM.join();
auto endM = std::chrono::steady_clock::now();
auto timeM =
    std::chrono::duration_cast<std::chrono::milliseconds>(endM - startM).count();
std::cout << "Async Time: " << timeM << "\n";

srand(1); //ensure both tests get exactly the same sleep times
auto startS = std::chrono::steady_clock::now();
std::thread workersS[workerCount];
std::thread writerS(&serialWrite, workerCount, std::ref(spb));
std::thread consumerS(&serialConsume, workerCount,
    std::ref(sdb));
for(std::size_t n=0; n<workerCount; ++n) {
    workersS[n] = std::thread(&serialThreadFcn, n,
        std::ref(spb), std::ref(sdb));
}
writerS.join();
for(std::size_t n=0;n<workerCount;++n) workersS[n].join();
consumerS.join();
auto endS = std::chrono::steady_clock::now();
auto timeS =
    std::chrono::duration_cast<std::chrono::milliseconds>(endS
    - startS).count();
std::cout<<"Serial Time: "<<timeS<<"\n";

srand(1);
double dataSum = 0.0;
double squareDataSum = 0.0;
for(std::size_t n=0;n<TEST_ITERATIONS*workerCount;++n)
{
    double currentValue =
        static_cast<double>(baseSpeed+rand()%baseVariation);
    dataSum += currentValue;
    squareDataSum += currentValue * currentValue;
}
std::size_t length = TEST_ITERATIONS*workerCount;
double mean = dataSum/static_cast<double>(length);
double variance = ((length * squareDataSum) - (dataSum *
    dataSum)) / (length * (length - 1));
double stdDeviation = sqrt(variance);
std::cout<<"mean: "<<mean<<"\n";
std::cout<<"variance: "<<variance<<"\n";
std::cout<<"stdDeviation: "<<stdDeviation<<"\n";

A.3.6 Unit Test for Order Preservation

/*

Test to verify both all buffer managers correctly order the
buffers from end to end

*/

98
```cpp
#include <iostream>
#include <chrono>
#include <cassert>
#include <cmath>

#include <multiPickupBuffer.hpp>
#include <multiDropoffBuffer.hpp>
#include <serialPickupBuffer.hpp>
#include <serialDropoffBuffer.hpp>

//#define ENABLE_WEAK_WORKER
//#define ENABLE_STRONG_WORKER
#define TEST_ITERATIONS 10000

std::size_t workerCount = 0;
std::size_t baseSpeed = 0;
std::size_t baseVariation = 0;
std::atomic<std::size_t> iterationCount = 0;

void printUsage()
{
    std::cout << "Usage: bufferComp [workerCount] [baseSpeedUs] [maxVariationUs]\n"
                "\n";
}

void multiThreadFcn(std::size_t workerId,
                     MultiPickupBuffer<std::array<int,10>>& pickup,
                     MultiDropoffBuffer<std::array<int,10>>& dropoff)
{
    while((iterationCount++)<(TEST_ITERATIONS*workerCount))
    {
        auto pickMd = pickup.getRead(workerId);
        pickup.unlockRead(workerId);
        int number = pickMd.buffer[0];
        pickup.passRead(workerId);

        std::size_t amount = baseSpeed+rand()%baseVariation;
        #ifdef ENABLE_WEAK_WORKER
        if (workerId == 0) amount *= 2;
        #endif
        ...
    }
}
```
#ifdef ENABLE_STRONG_WORKER
if (workerId == 0) amount /= 2;
#endif
std::this_thread::sleep_for(std::chrono::microseconds(amount));

auto dropMd = dropoff.getWrite(pickMd.id);
dropoff.unlockWrite(dropMd.id);
dropMd.buffer[0] = number;
dropoff.passWrite(dropMd.id, dropMd.bufferIndex);
}
}

void serialThreadFcn(std::size_t workerId,
    SerialPickupBuffer<std::array<int,10>>& pickup,
    SerialDropoffBuffer<std::array<int,10>>& dropoff)
{
    for(std::size_t n=0;n<TEST_ITERATIONS;++n)
    {
        auto pickMd = pickup.getRead(workerId);
        static_cast<void>(pickMd);
        pickup.unlockRead(workerId);
        int number = pickMd.buffer[0];
pickup.passRead(workerId);

        std::size_t amount = baseSpeed+rand()%baseVariation;
        #ifdef ENABLE_WEAK_WORKER
if (workerId == 0) amount *= 2;
#endif
        #ifdef ENABLE_STRONG_WORKER
if (workerId == 0) amount /= 2;
#endif
        std::this_thread::sleep_for(std::chrono::microseconds(amount));

        auto dropMd = dropoff.getWrite(workerId);
dropoff.unlockWrite(workerId);
dropMd.buffer[0] = number;
dropoff.passWrite(dropMd.id, dropMd.bufferIndex);
    }
}
void multiWrite(std::size_t workerCount,
        MultiPickupBuffer<std::array<int,10>>& pickup)
{
    for(std::size_t n=0;n<TEST_ITERATIONS*workerCount;++n)
    {
        auto& buffer = pickup.getWrite();
        buffer[0] = n;
        pickup.passWrite();
    }
}

void multiConsume(std::size_t workerCount,
        MultiDropoffBuffer<std::array<int,10>>& dropoff)
{
    for(std::size_t n=0;n<TEST_ITERATIONS*workerCount;++n)
    {
        auto buffer = dropoff.getRead();
        if (static_cast<std::size_t>(buffer.buffer[0]) != n)
        {
            std::cout<<"ABM out of order\n";
            assert(false);
        }
        dropoff.passRead();
    }
    std::cout<<"ABM pass\n";
}

void serialWrite(std::size_t workerCount,
        SerialPickupBuffer<std::array<int,10>>& pickup)
{
    for(std::size_t n=0;n<TEST_ITERATIONS*workerCount;++n)
    {
        auto& buffer = pickup.getWrite();
        buffer[0] = n;
        pickup.passWrite();
    }
}

void serialConsume(std::size_t workerCount,
        SerialDropoffBuffer<std::array<int,10>>& dropoff)
for(std::size_t n=0;n<TEST_ITERATIONS*workerCount;++n)
{
    auto buffer = dropoff.getRead();
    if (static_cast<std::size_t>(buffer.buffer[0]) != n)
    {
        std::cout<<"Serial out of order\n";
        assert(false);
    }
    dropoff.passRead();
}
std::cout<<"Serial pass\n";

int main(int argc, char* argv[])
{
    if(argc != 4)
    {
        printUsage();
        return 0;
    }
    workerCount = strtol(argv[1], NULL, 10);
    baseSpeed = strtol(argv[2], NULL, 10);
    baseVariation = strtol(argv[3], NULL, 10);
    MultiPickupBuffer<std::array<int,10>> mpb(workerCount*3);
    MultiDropoffBuffer<std::array<int,10>> mdb(workerCount*3);
    SerialPickupBuffer<std::array<int,10>> spb(workerCount*3,
         workerCount);
    SerialDropoffBuffer<std::array<int,10>> sdb(workerCount*3,
         workerCount);
    srand(1); //ensure both tests get exactly the same sleep times assigned
    std::thread workersM[workerCount];
    std::thread writerM(&multiWrite, workerCount, std::ref(mpb));
    std::thread consumerM(&multiConsume, workerCount,
                            std::ref(mdb));
    for(std::size_t n=0;n<workerCount;++n)
    {

workersM[n] = std::thread(&multiThreadFcn, n,
                       std::ref(mpb), std::ref(mdb));
}
writerM.join();
for(std::size_t n=0;n<workerCount;++n) workersM[n].join();
consumerM.join();

srand(1); //ensure both tests get exactly the same sleep times
assigned
std::thread workersS[workerCount];
std::thread writerS(&serialWrite, workerCount, std::ref(spb));
std::thread consumerS(&serialConsume, workerCount,
                       std::ref(sdb));
for(std::size_t n=0;n<workerCount;++n)
{
    workersS[n] = std::thread(&serialThreadFcn, n,
                               std::ref(spb), std::ref(sdb));
}
writerS.join();
for(std::size_t n=0;n<workerCount;++n) workersS[n].join();
consumerS.join();

A.4  CP-OFDM Workers

A.4.1  TX PHY Worker

Header File

#ifdef tx_hpp
#define tx_hpp

#include <memory>
#include <complex>
#include <multiDropoffBuffer.hpp>
#include <constants.hpp>

class Tx
{

103
public:
    using Ptr = std::unique_ptr<Tx>;
    static Ptr create(std::size_t workers,
                      MultiDropoffBuffer<OTABUFFER>* ota);
    virtual ~Tx() = default;

    virtual void start() = 0;
    virtual void wait() = 0;
};

#endif

Source Code

#include <tx.hpp>
#include <qpsk.hpp>
#include <constants.hpp>
#include <multiPickupBuffer.hpp>
#include <multiDropoffBuffer.hpp>

#include <iostream>
#include <vector>
#include <thread>
#include <cstring>
#include <chrono>
#include <mutex>

#if USING_UHD
#include <uhd/usrp/multi_usrp.hpp>
#endif

struct TxImpl : public Tx
{
    public:
        TxImpl(std::size_t workers,
                MultiDropoffBuffer<OTABUFFER>* ota);
        ~TxImpl();
        void start();
        void wait();

    public:

private:
  static constexpr std::size_t size =
      (SYMBOL_TIME_IN_SAMPLES - 2*OFFSET_BINS) * BITS_PER_SYMBOL;
  std::vector<fftwf_complex*> in, out;
  MultiPickupBuffer<std::array<int, size>> mpb;
  MultiDropoffBuffer<OTABUFFER>* mdb;
  std::vector<std::thread> threads;

#if USING_UHD
  uhd::usrp::multi_usrp::sptr dev;
  uhd::tx_streamer::sptr txStream;
#endif

  std::size_t workers;
  std::size_t testIterations = 30000;
  std::atomic<std::size_t> iterationsCompleted = 0;
  void dataGenerator();
  void worker(std::size_t myId);
  void consumer();
  void cpofdm(fftwf_complex* in, OTABUFFER& out);
};

void
TxImpl::worker(std::size_t myId)
{
  fftwf_plan p;
  FFTW_LOCK.lock();
  p = fftwf_plan_dft_1d(SYMBOL_TIME_IN_SAMPLES,
             &in[myId][0],
             &out[myId][GUARD_INTERVAL_IN_SAMPLES],
             FFTW_BACKWARD,
             FFTW_ESTIMATE);
  FFTW_LOCK.unlock();
  while((iterationsCompleted++)<testIterations)
  {
    //Get Read Buffer
    auto bufferMd = mpb.getRead(myId);
    mpb.unlockRead(myId);
    //Modulate to QPSK and perform resource mapping

105
A.4.2 RX PHY Worker

Header File

```cpp
#ifndef rx_hpp
#define rx_hpp

#include <memory>
#include <complex>
#include <constants.hpp>
#include <multiDropoffBuffer.hpp>

class Rx
{
public:
    using Ptr = std::unique_ptr<Rx>;
    static Ptr create(std::size_t workers,
                      MultiDropoffBuffer<OTABUFFER>* ota);
    virtual ~Rx() = default;

```
virtual void start() = 0;
virtual void wait() = 0;

};

#endif

Source Code

struct corrMd {
    int64_t corrI = 0;
    int64_t corrQ = 0;
    int64_t norm = 0;
    std::size_t length = 0;
};

struct corrRet {
    int64_t time = 0;
    uint64_t amount = 0;
};

struct RxImpl : public Rx
{
    public:
        RxImpl(std::size_t workers,
                MultiDropoffBuffer<OTABUFFER>* ota);
        ~RxImpl();
        void start();
        void wait();

    private:
        static constexpr std::size_t size =
            (SYMBOL_TIME_IN_SAMPLES - 2*OFFSET_BINS)*BITS_PER_SYMBOL;
        std::vector<fftwf_complex*> in, out;
        MultiDropoffBuffer<OTABUFFER>* ota;
        MultiPickupBuffer<std::array<std::complex<int16_t>,
            SYMBOL_TIME_IN_SAMPLES>> mpb;
        MultiDropoffBuffer<std::array<int,size>> mdb;
        std::vector<std::thread> threads;
float threshold = 0.5;
std::size_t workers;
std::size_t testIterations = 30000;
std::atomic<std::size_t> iterationsCompleted = 0;
void signalGenerator();
void worker(std::size_t myId);
void consumer();
void toCFloat(std::array<std::complex<int16_t>,
              SYMBOL_TIME_IN_SAMPLES>& input,
              fftwf_complex* output);
corrMd correlate(std::complex<int16_t>* x,
                   std::complex<int16_t>* y,
                   std::size_t length);
double toNormCorr(corrMd in);
corrRet cpofdmCorrelator(std::vector<
                         std::complex<int16_t>>& input);
};

void
RxImpl::worker(std::size_t myId)
{
    fftwf_plan p;
    FFTW_LOCK.lock();
p = fftwf_plan_dft_1d(SYMBOL_TIME_IN_SAMPLES,
                         &in[myId][0],
                         &out[myId][0],
                         FFTW_FORWARD,
                         FFTW_ESTIMATE);
    FFTW_LOCK.unlock();
    while((iterationsCompleted++)<testIterations)
    {
        //Get Read Buffer
        auto bufferMd = mpb.getRead(myId);
        mpb.unlockRead(myId);
        //Convert Complex Int16 to Complex Float
        toCFloat(bufferMd.buffer, &in[myId][0]);
        //Pass Read Buffer
        mpb.passRead(myId);
        //Demodulate OFDM Symbol
        fftwf_execute(p);
    }
// Get Write Buffer
auto t = mdb.getWrite(bufferMd.id);
mdb.unlockWrite(t.id);
// Demodulate QPSK and remap carrier bits
fromQpsk(&out[myId][0],
    &t.buffer[0],
    t.buffer.size(),
    OFFSET_BINS);
// Pass Write Buffer
mdb.passWrite(t.id, t.bufferIndex);
}

fftwf_destroy_plan(p);

A.5 CP-OFDM Frame Detection

```cpp
struct corrRet {
    int64_t time = 0;
    uint64_t amount = 0;
};

corrRet
RxImpl::cpofdmCorrelator(std::vector<std::complex<int16_t>> & input) {
    double prev = 0, curr = 0, next = 0;
    corrMd currCorr, nextCorr;
    // get initial values
    auto start = std::chrono::steady_clock::now();
    nextCorr = correlate(&input[0],
        &input[SYMBOL_TIME_IN_SAMPLES],
        GUARD_INTERVAL_IN_SAMPLES);
    next = toNormCorr(nextCorr);
    // adjust only for next samples
    int total = 1;
    for(std::size_t n=1;
        n<input.size()-(SYMBOL_TIME_IN_SAMPLES+GUARD_INTERVAL_IN_SAMPLES);
        ++n)
    {
```
prev = curr;
curr = next;
currCorr = nextCorr;

//remove sample corr from prev iteration
nextCorr.corrI = currCorr.corrI -
    (static_cast<int32_t>(input[n-1].real())*
    static_cast<int32_t>(input[n-1+SAMPLE_TIME_IN_SAMPLES].real()) +
    static_cast<int32_t>(input[n-1].imag())*
    static_cast<int32_t>(input[n-1+SAMPLE_TIME_IN_SAMPLES].imag()));
nextCorr.corrQ = currCorr.corrQ -
    (static_cast<int32_t>(input[n-1].imag())*
    static_cast<int32_t>(input[n-1+SAMPLE_TIME_IN_SAMPLES].real()) -
    static_cast<int32_t>(input[n-1].real())*
    static_cast<int32_t>(input[n-1+SAMPLE_TIME_IN_SAMPLES].imag()));
nextCorr.norm = currCorr.norm -
    (static_cast<int32_t>(input[n-1].real())*
    static_cast<int32_t>(input[n-1].real()) +
    static_cast<int32_t>(input[n-1].imag())*
    static_cast<int32_t>(input[n-1].imag()));

//add sample corr for this iteration
nextCorr.corrI +=
    static_cast<int32_t>(input[n-1+GUARD_INTERVAL_IN_SAMPLES].real())*
    static_cast<int32_t>(input[n-1+SAMPLE_TIME_IN_SAMPLES+GUARD_INTERVAL_IN_SAMPLES].real()) +
    static_cast<int32_t>(input[n-1+GUARD_INTERVAL_IN_SAMPLES].imag())*
    static_cast<int32_t>(input[n-1+SAMPLE_TIME_IN_SAMPLES+GUARD_INTERVAL_IN_SAMPLES].imag());
nextCorr.corrQ +=
    static_cast<int32_t>(input[n-1+GUARD_INTERVAL_IN_SAMPLES].imag())*
    static_cast<int32_t>(input[n-1+SAMPLE_TIME_IN_SAMPLES+GUARD_INTERVAL_IN_SAMPLES].imag()) -
    static_cast<int32_t>(input[n-1+GUARD_INTERVAL_IN_SAMPLES].real())*
    static_cast<int32_t>(input[n-1+SAMPLE_TIME_IN_SAMPLES+GUARD_INTERVAL_IN_SAMPLES].real());
nextCorr.norm +=
    static_cast<int32_t>(input[n-1+GUARD_INTERVAL_IN_SAMPLES].real())*
    static_cast<int32_t>(input[n-1+GUARD_INTERVAL_IN_SAMPLES].real()) +
    static_cast<int32_t>(input[n-1+GUARD_INTERVAL_IN_SAMPLES].imag())*
    static_cast<int32_t>(input[n-1+GUARD_INTERVAL_IN_SAMPLES].imag());
next = toNormCorr(nextCorr);
if(curr > threshold)
{
    //commented out for performance timing
    //return n-1;
}
++total;
}
auto end = std::chrono::steady_clock::now();
auto corrTime =
    std::chrono::duration_cast<std::chrono::nanoseconds>(end -
    start).count();
//commented out for performance timing
//return -1
return {corrTime, total};
