SURFER : ANY-CORE SOFTWARE DEFINED RADIO

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Michael L Dickens

____________________________________
J. Nicholas Laneman, Director

Graduate Program in Electrical Engineering
Notre Dame, Indiana
April 2012
Abstract

by

Michael L Dickens

This thesis presents a general-purpose software defined radio architecture, Surfer, which has been designed to allow for dynamic modification of system behavior during runtime. Surfer leverages commodity processing devices such as those on a typical laptop to allow heterogeneous processing. This heterogeneity is enabled through the use of one or more signal processing implementations, or flavors, per block. Each flavor can target a different processing device, and flavors can be swapped during runtime without interrupting data flow. Two waveform programming interfaces are provided that can be intermixed for waveform definition: traditional block-centric programming, and a buffer-centric approach such as that found in MATLAB. Within a waveform graph consisting of processing blocks, each connection is a buffer that holds the data being generated by one block and consumed by others. Each connection is paired with a threshold on the amount of data in the buffer, and is used for scheduling. Runtime statistics allow a system supervisor to assist in mapping and scheduling, dynamically during runtime, in order to meet waveform requirements. This work demonstrates that it is both possible and useful to implement dynamic reconfiguration, mapping, and scheduling during runtime on commodity hardware, entirely in software. These capabilities are demonstrated via an OFDM transmitter implementation.
To my grandfather, Milton Dickens, PhD; and my father, J. Kirk Dickens, PhD:

They always wanted one of their offspring to have a PhD.

And to Robert D. Preuss, PhD: You never truly knew the lessons I learned from you. Thank you for mentoring me; may you rest in peace, my friend.
ACKNOWLEDGMENTS

I would first and foremost like to thank my advisor, Dr. J. Nicolas Laneman, for his advice, encouragement, support, and – most of all – patience in tolerating the long and winding path I took that has culminated in this dissertation.

I am grateful to my committee – Drs. Poellabauer, Pratt, Rondeau, and Stevenson – for their time and valuable suggestions in this dissertation writing process as well as the papers and presentations leading up to it.

My advisor’s group has been supportive and encouraging the many ways. Thank you Alice, Brian, Ebrahim, Glenn, Joseph, Mostafa, Peyman, and Utsaw – we’ve had some great times together over the years.

I would not be where I am today with having been a GNU Radio developer, and the challenges and friendships made though that project.

My friends and Notre Dame colleagues from my Master’s days have also played an important role in keeping me sane. A big “shout out” especially to Betty, Rob, Sean, and Sami.

I thank my parents and grandparents for making education a high priority for me during my youth, and encouraging me to pursue higher education.

Finally, I recognize my partner, Jeanne, and son, Conrad, for loving and supporting me throughout this process.
# CONTENTS

ACKNOWLEDGMENTS ............................................ iii

FIGURES ....................................................... ix

CHAPTER 1: INTRODUCTION ................................. 1
  1.1 Problem Description ................................... 3
  1.2 Approach .............................................. 8
  1.3 Contributions ........................................ 10
  1.4 Thesis Layout ......................................... 13

CHAPTER 2: BACKGROUND ................................. 16
  2.1 A Brief History of Wireless Communications ....... 16
  2.2 A Brief History of Software Defined Radio ............ 18
  2.3 SDR Basics ............................................. 23
      2.3.1 Block Abstraction ................................ 24
      2.3.2 Waveform / Graph ................................ 25
      2.3.3 Data Flow ........................................ 27
      2.3.4 Buffering ......................................... 29
      2.3.5 Packets and Frames ............................. 31
      2.3.6 Computational Intensity ......................... 32
      2.3.7 Chunks, Throughput, and Latency ............ 33
      2.3.8 Signal Processing ................................ 34
      2.3.9 Scheduling and Mapping ......................... 36
      2.3.10 Reconfiguration ................................. 37
      2.3.11 OS Abstraction Layer ........................... 38
      2.3.12 Waveform Interfaces ............................ 39
  2.4 Other SDR implementations ............................ 40
  2.5 Other Related Work ................................... 42
      2.5.1 SDR Using GPUs .................................. 42
      2.5.2 GNU Radio ....................................... 43
      2.5.3 IRIS ............................................. 45
FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Conceptual role of the SDR runtime system: taking a waveform graph and list of processing devices, and performing mapping, buffering, scheduling, and signal processing.</td>
</tr>
<tr>
<td>1.2</td>
<td>Conceptual division of SDR processing hierarchy in an application, waveforms, and blocks.</td>
</tr>
<tr>
<td>2.1</td>
<td>Polyphase implementation of a downsample-by-L block.</td>
</tr>
<tr>
<td>2.2</td>
<td>Directed graph representing a narrowband FM demodulator.</td>
</tr>
<tr>
<td>3.1</td>
<td>OSI-like layers model applied to the Surfer SDR runtime implementation.</td>
</tr>
<tr>
<td>4.1</td>
<td>Example DynamicStructure implementation.</td>
</tr>
<tr>
<td>5.1</td>
<td>Class relationships for the Shark namespace implementation.</td>
</tr>
<tr>
<td>5.2</td>
<td>Potential threshold deadlock situation.</td>
</tr>
<tr>
<td>5.3</td>
<td>Threshold processing sequence.</td>
</tr>
<tr>
<td>5.4</td>
<td>Block runner sequence.</td>
</tr>
<tr>
<td>5.5</td>
<td>Example of buffer use, via a writer, master, the actual buffer, and multiple readers.</td>
</tr>
<tr>
<td>5.6</td>
<td>Internally kept buffer parameters.</td>
</tr>
<tr>
<td>6.1</td>
<td>Class relationships for the Spf namespace implementation.</td>
</tr>
<tr>
<td>6.2</td>
<td>General diagram of a SDR signal processing block.</td>
</tr>
<tr>
<td>6.3</td>
<td>Visualization of a kernel shared memory double-mapped buffer allocation: (a) An initial allocation of twice the necessary size is made, to retrieve a valid address, A, for containing the double mappings. (b) The mapping from (a) is removed, and a new one of the actual desired length created and then mapped twice in the same contiguous memory space.</td>
</tr>
<tr>
<td>6.4</td>
<td>Example reconfiguration using a switch.</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>6.5</td>
<td>Example hierarchy of flavors for the Sum block.</td>
</tr>
<tr>
<td>6.6</td>
<td>A narrowband FM demodulator, showing an example subgraph “downsample by L” block implementation.</td>
</tr>
<tr>
<td>6.7</td>
<td>Example flavor table and its relation to a signal processing block.</td>
</tr>
<tr>
<td>7.1</td>
<td>Generic hardware architecture assumed by OpenCL.</td>
</tr>
<tr>
<td>7.2</td>
<td>Universal Modeling Language diagram of OpenCL 1.1 classes.</td>
</tr>
<tr>
<td>7.3</td>
<td>Class relationships for the naCL framework implementation.</td>
</tr>
<tr>
<td>7.4</td>
<td>1-dimensional local and global work items example, showing the first 15 workers and each’s global, group, and local identifier.</td>
</tr>
<tr>
<td>8.1</td>
<td>Runtime optimization of N-1 2-way adders into a single N-way summation.</td>
</tr>
<tr>
<td>8.2</td>
<td>Visualization of data ordering for a block with N input streams, each of which contains M items.</td>
</tr>
<tr>
<td>9.1</td>
<td>Basic narrowband FM waveform with downsampler, showing available flavors for each block.</td>
</tr>
<tr>
<td>9.2</td>
<td>Narrowband FM supervisor sequence, demonstrating swapping flavors at a trigger point.</td>
</tr>
<tr>
<td>9.3</td>
<td>Narrowband FM waveform load graph over time.</td>
</tr>
<tr>
<td>9.4</td>
<td>Narrowband FM waveform execution time versus source threshold, different flavors, 1 to 4 block runners (#CPU, #naCL), no rate limit, with affinity only.</td>
</tr>
<tr>
<td>9.5</td>
<td>Narrowband FM waveform throughput versus source threshold, different flavors, 1 to 4 block runners (#CPU, #naCL), no rate limit, with affinity only.</td>
</tr>
<tr>
<td>9.6</td>
<td>Typical execution timing for the poorly mapped NBFM demo, viewed by block, with runner thread 0 targeting the local CPU and runner thread 1 targeting the GPU via naCL.</td>
</tr>
<tr>
<td>9.7</td>
<td>Typical execution timing for the poorly mapped NBFM demo, viewed by thread, with runner thread 0 targeting the local CPU and runner thread 1 targeting the GPU via naCL.</td>
</tr>
<tr>
<td>9.8</td>
<td>Typical execution timing for the better mapped NBFM demo, viewed by block, with runner thread 0 targeting the local CPU and runner thread 1 targeting the GPU via naCL.</td>
</tr>
<tr>
<td>9.9</td>
<td>OFDM transmitter waveform, including available flavors in which multiple are available.</td>
</tr>
</tbody>
</table>
9.10 OFDM receiver waveform created using the GNU Radio Companion.

9.11 Typical execution timing for the OFDM demo under default settings, consuming exactly one input packet’s worth of audio data.

9.12 Typical execution timing for the OFDM demo when the GNU Radio packet encoder input threshold is set to 10 packets worth of data.

9.13 Typical execution timing for the OFDM demo when the FFT Shift and later blocks’ input threshold is set to one OFDM symbol.
CHAPTER 1

INTRODUCTION

With the growing complexity and raw processing capabilities of general purpose processing devices (GPPs) – for example multi-core central processing units (CPUs) and many-core graphical processing units (GPUs) – the ability to efficiently manipulate different types of processing devices within the same project development framework during runtime has become a significant challenge. Traditional handling of heterogeneous processing has involved pre-determination of the mapping of the various tasks to be performed onto the various available processing devices, or runtime opportunistic execution where devices are used on a first come, first served basis. Providing the capability of changing runtime mappings raises new challenges in system and algorithm design due to the need for runtime statistics to allow for objective decision making on the part of a system supervisor.

This chapter motivates the essential features for designing a new software defined radio (SDR) by discussing current processing device trends and desirable operating characteristics for such an SDR runtime system. The high-level necessary runtime functionality of this SDR is shown in Figure [1.1] and each of the various components involved will be discussed in detail in this thesis. Using this figure as a starting point, our problem description is first provided, followed by the basics of our approach to handling runtime behavior. Next, we describe the contributions of this thesis, and, finally, preview the layout of the rest of this document.
Figure 1.1. Conceptual role of the SDR runtime system: taking a waveform graph and list of processing devices, and performing mapping, buffering, scheduling, and signal processing.
1.1 Problem Description

This thesis describes the design and implementation of a general purpose software defined radio framework – the collection of executables, libraries, and header, resource, and data files for a given project – called Surfer. The primary goals of Surfer are to implement a runtime system allowing for (1) mapping of signal processing blocks onto a heterogeneous collection of processors, (2) scheduling of the chunk size per block to process in order to set the throughput / latency processing point for each block as well as the overall waveform, and (3) controlling the whole system via a supervisor that can use a variety of statistics – such as system load and per block throughput and latency histories – to determine the block mapping and chunk size scheduling for processing. As Surfer is a fully functional SDR, much of this thesis is devoted to describing the software design process and project architecture rather than algorithms, though methodologies are provided where possible and applicable. Further, Surfer provides significant underlying structure and functionality beyond these primary goals, all of which are described when they are significantly different than that of known SDR systems.

The term software defined radio refers to a class of radios that can be reprogrammed and reconfigured via software [66]. Although SDR systems do require hardware, their goal is to move as much of the radio protocol handling for both transmit and receive chains from fixed hardware on to some type of reprogrammable and/or reconfigurable processing device [120]. A processing device can be a CPU, GPU, digital signal processor (DSP), system on a chip (SOC), field-programmable gate array (FPGA), arithmetic logic unit (ALU), application-specific integrated circuit (ASIC), or any other device that is either programmable or provides some specific signal processing oriented functionality. Processing de-
vices that are statically programmed can be used for performing computations only; these devices cannot generally host a SDR kernel – the runtime routines that control the radio. At least one processing device must be reprogrammable, by dynamically loading an operating system (OS) and applications from some sort of storage. Each processing device type has its unique strengths, and hence algorithm implementations will vary somewhat from device to device – for example a 2-core CPU is well suited for inherently sequential algorithms while a 100-core GPU is generally better for highly parallelizable algorithms.

The raw computation power of processing devices has been doubling every few years, whether via the addition of components placed onto the integrated circuit, or via the addition of more processing cores \[101\] – the term used for this doubling is called Moore’s Law. This law does not mean that processing capabilities will double during this period, but rather that the integrated circuit complexity will. Moore does not discuss processing devices with more than one core, nor how to make use of their raw computation power via parallel programming. In recent years, computer CPU speeds have saturated at around 3 GHz without specialized cooling systems; although this limitation will likely be overcome in the near future, it presented a “bump in the road” for CPU manufacturers. That bump spurred the growth of multi-core and many-core processors, and now almost every type of processing device comes in a variant providing multiple cores.

We use multi-core to describe a processing device containing typically no more than 16 cores; many-core means numerous cores – currently 64 to around 1024 in a single processing device. Multi-core processors tend to use a single bus to access global memory, which can become a bottleneck when all cores are actively executing. Many-core processors try to work around this issue by arranging cores
on a grid, in grids of multi-core clusters, or a pipeline of multi-core clusters – where each cluster has to independently deal with its own memory bottleneck. Both device architectures use multiple levels of memory and caching to reduce bus contention as well as speed up execution of commonly accessed data so long as it can fit into cache limits.

Because many of these processing devices are mainstream and general-purpose, today’s consumer-oriented computers provide significantly more raw computation power than needed by the typical consumer. Today’s portable computers – for example “smartphones” and tablets – provide a rich user experience and the raw processing power of consumer-oriented computers of maybe 5 years prior. These portable devices are possible because the energy efficiency of processing devices has been doubling at about the same rate \cite{55} as the overall processor complexity – the so-called “Koomey’s Law”.

SDR was originally conceived in order to move away from fixed hardware, so-as to allow the radio to be able to handle multiple protocols solely through reconfiguration, and to be able to fix bugs in current protocol implementations and/or add new ones without changing the hardware. The field of SDR is a mere 20 years old \cite{66}, and even at that time one of the underlying principles was to take advantage of Moore’s law – and, now, Koomey’s: If an SDR protocol implementation cannot function on current processing devices, just wait a few years until the raw processing power catches up. Or, if the protocol cannot function with the energy requirements of a portable computer, just wait few years until the energy usage has decreased enough.

As computers have become more heterogeneous in their use of processing devices – even a modern consumer-oriented laptop contains a multi-core CPU and
many-core GPU – there is a growing need to create applications that take advantage of all of these devices. For many application areas, it is enough to simply rewrite specific computationally intensive functions; these functions are executed only as needed, e.g., to add some effect to a photo. For code that must be repeatedly called, some broader functionality must be put in place to truly utilize the underlying hardware.

SDR presents a unique challenge in parallel programming and real-time signal processing, because it involves the processing of streaming data in real time, requiring that the utilized processing power must exceed that needed to keep up with all data rates. For example, if retrieving data from an audio source (e.g., microphone) at 44,100 samples per second, the average processing time per sample must take no more than $44,100^{-1}$ seconds in order to avoid data overflow. Assuming that the processing system is set up correctly (e.g., its buffers are large enough), the actual processing time for any specific sample or chunk of samples may be larger than this average so long as enough other samples are processed faster than the required average.

As shown in Figure 1.2, SDR processing breaks down radio protocols into parts, each called, in this thesis, a block. Each block generally performs a computation smaller than the overall protocol, e.g., adding together element by element two vectors of 32-bit big-endian integers, or performing a forward fast-Fourier transform (FFT) on a vector of 32-bit floating point samples. Chaining these simple blocks together into a graph allows for the creation of simple or complex protocols – e.g., a repeater, a high-definition television (HDTV) receiver, or an FM transmitter. Each overall source(s) to sink(s) graph, in SDR terms, representing a given radio protocol is called a waveform; an SDR application is made up of one
or more waveforms.

SDR is, broadly, another form of stream processing, where the streams data are grouped in chunks – frames or packets – and processed by a waveform graph. These chunks of data are buffered between blocks, and both the buffer and chunk size must be determined so-as to allow processing to keep up with the rate of incoming data and amount of generated data. We use the term *scheduling* to describe the chunk size determination, in the same manner as it is defined for the GNU Radio SDR project. Note that our use of the term *scheduling* here is somewhat different than that for OS kernels: “the allocation of tasks to processors, and the definition of the tasks’ execution order.” [126] Scheduling larger chunks generally results in greater average throughput in samples per second but also higher input to output latency, while processing smaller chunks generally results in lower average throughput but also lower latency. Scheduling can be performed
a-priori or during runtime.

The process of determining which block should be execute on which processing device is called *mapping*. All known SDR systems to date map each waveform’s block onto a specific processing device – a CPU or GPU, but not generally a specific core – and then do not change that setting. Some systems perform this mapping a-priori while others do it during runtime – e.g., in a first-come first-served, opportunistic fashion. SDR systems using a-priori mapping and scheduling cannot handle the dynamic addition and removal of arbitrary blocks or waveforms, while those performing mapping at runtime have thus far not been able to beneficially make use of the heterogenous nature of modern computers.

We view *Surfer* as an enhanced “mashup” of MATLAB [89], GNU Radio [52], and Trinity College’s IRIS [84] – the latter two of which are discussed in some depth in Section 2.5. *Surfer* provides the relatively simple interfaces of MATLAB and GNU Radio with the reconfigurability of IRIS, but augments these projects via heterogeneous processing on commodity, consumer-oriented hardware as well as through dynamic, runtime control of system behavior.

### 1.2 Approach

We started with SDR – both using and programming – via the open source GNU Radio project, and expanded our knowledge of SDR systems by reviewing standards such as the SCA, design and architecture oriented dissertations and papers, and, when available, actual project code. We wrote blocks and evaluated demonstration code for GNU Radio, and reviewed core code for Virginia Tech’s Open-Source SCA Implementation::Embedded (OSSIE) project [32] and IRIS. We compared and contrasted these SDR projects design and architecture.
This background work led us to the conclusion that there was room for another SDR framework, one replicating the best practices of these projects and providing enhanced flexibility in processing device utilization as well as functionality in runtime behavior. The end result, *Surfer*, is a flexible SDR framework capable of executing using fully a-priori or opportunistic mapping and scheduling, or a combination of the two. Its use of multiple heterogeneous processing devices make it *any*-core capable.

Instead of trying to augment an existing general-purpose SDR framework such as OSSIE or GNU Radio, we opted to make a fresh start for a variety of reasons. By starting anew, we could control the number of system dependencies from the up-start, keeping them at a minimum by implementing an in-project abstraction layer to handle the required functions. We also were not restricted to the already-in-place system runtime functionality as either required by a standard, or as implemented but not well documented. Although starting over meant “recreating some wheels”, by doing so we always had control over system functionality and could easily track down and fix bugs – because we were the ones who created them.

Starting fresh also provided the opportunity to utilize a truly object-oriented programming approach to signal processing, e.g., using C++ templates to describe processing blocks’ stream data types in order to provide a single class that can be instantiated for a number of different types (e.g., a single block class representing the “fft” algorithm that will work for the types “float” and “double”). Through this object-oriented approach, we determined that the traditional SDR blocks that contains all of the functionality required for its particular algorithm implementation can be split such that the algorithm implementation is selectable.
from within a table while the rest of the routines remain in the common class. We also found that using a chunk size as a threshold works effectively. Each threshold value represents the maximum amount of data to consume or generate in any single processing iteration; this value can be changed between processing iterations, or left alone hence becoming a constant. The use of an algorithm implementation table and thresholds provide almost a-priori mapping behavior yet still allow for opportunistic mapping and scheduling during runtime if desired.

No matter the way a waveform is defined – via a graphical and/or text-base interface – none of the above goals should impact the waveform definition methodology. All changes should be transparent to the waveform programmer – allowing casual developers to continue as before and advanced ones to take advantage of the additions. Thus, for example, the basic waveform graph should be independent of the underlying set of processing devices to be used for signal processing. The SDR system should be able to make use of information embedded in the waveform graph, along with user preferences, for setting buffer and chunk sizes as well as determining which processing device to handle each block’s processing. All of these factors went into the design of Surfer, and along the way a number of novel architectural features were implemented and demonstrated to others in the field.

1.3 Contributions

*Surfer* was inspired in various ways by GNU Radio, IRIS, MATLAB, and “the Nucleus Concept” [118], but designed and programmed entirely independently of those projects. Many issues are involved in the design and implementation of any complex computer program, and *Surfer* is no exception. Although *Surfer* currently
consists of some 100k lines of code, the scope of this thesis does not primarily include the implementation issues encountered by any complex computer program, whether mundane or otherwise. For example, we do not describe the actual algorithm implementations for different processing devices, e.g., a decimating FIR filter using a specific vector framework – though some are included as examples to show how to use those programming interfaces. We assume that such work is similar to that in other SDR frameworks: given a particular programming interface for Surfer or GNU Radio, one can write or port a given algorithm implementation. The extensive quality-assurance testing code provided by Surfer is also not described in this document. Rather, the main contributions of this thesis result from our investigating of any-core runtime system design and implementation issues which would be beneficial to the SDR community. This work demonstrates that it both possible and useful to implement dynamic reconfiguration, mapping, and scheduling during runtime on commodity hardware, entirely in software.

In particular, our explorations have lead to four primary contributions.

First, the thesis develops the use of thresholds on input and output buffers to determine when a block is ready to be processed. We show that using thresholds on both input and output buffers is often redundant and can lead to processing deadlock; instead, setting a single threshold on whichever connection has the highest nominal sample rate defines the other thresholds. Thresholds provide a way to set the throughput / latency operating point for each block, and hence for each waveform. We provide examples showing the throughput / latency profiles of some blocks. The use of a simple threshold results in near-constant overhead time, which means that, when compared with other runtime SDR systems determining these thresholds using a variable time algorithm, more CPU time is available for
performing actual signal processing.

The second contribution of this thesis is a mechanism for the collection and use of runtime statistics for a given block, set of blocks, whole waveform, individual Surfer threads, the overall Surfer process, or the OS. Examples of useful statistics include the average throughput or latency when doing processing, and CPU load. Other statistics can be collected when a means is available to do so – for example the energy usage of the host computer via a monitor. Runtime statistics will depend on the priority of the executing process and its threads, as well as the efficiency of the signal processing implementations. Statistics can be used by a supervisor to monitor the system as well as control the throughput / latency operating point by modifying threshold values.

The third contribution is a novel type of waveform reconfiguration targeting heterogeneous processing devices through the implementation of one or more signal processing flavors within a given block. Splitting the block programming interface into methods that are common to all flavors and a table containing available flavors, a single block definition can then be used to target multiple different processing devices. Thus, instead of having multiple blocks providing the same runtime interface but targeting different processing devices, a single block implementation is used, resulting in a single waveform graph with heterogeneous processing capabilities. This heterogeneous nature of flavors makes Surfer any-core capable. A necessary condition for this split is that the block state be transportable and interpretable across all processing devices, and we designed a dynamic structure class that provides such functionality. Some examples are provided showing the switching of flavors by a supervisor during runtime, using runtime statistics, while maintaining seamless data processing.
The last primary contribution is a novel MATLAB-programming style algebraic interface for data-flow processing. We show a duality when cycles are not allowed in the waveform graph between the traditional block-centric programming view and the MATLAB-style buffer-centric approach. The definition of the necessary types and operators required for defining such an interface leads to a simple example. Along the way, we show some interesting runtime optimizations that can be performed as the graph is being defined, under the correct circumstances.

1.4 Thesis Layout

Chapter 2 provides background information about the topics discussed in this thesis, including brief histories of wireless communications as well as SDR. We describe, briefly and concisely, the basics of a SDR runtime system – from the block abstraction and graphs, to packets and frames and how they flow through the graph as chunks of data, to scheduling and mapping. A listing of numerous other SDR implementations is provided, followed by other related and relevant work including more in-depth information regarding the runtime properties of the three primary SDR implementations currently in research use: GNU Radio, IRIS, and OSSIE. Appendix A contains many of the numerous acronyms and abbreviations, and much terminology, used in this thesis and the SDR world.

In Chapter 3, we include an overview of the Surfer SDR framework, including rationales for design decisions and the general system architecture used for assigning namespaces. The system architecture mirrors that of the OSI model, and provides a guideline for the location of classes depending on their functionality.

The abstraction layer used by Surfer is found in Chapter 4. This layer contains functionality common to other similar projects and the C++ standard library, as
well as new classes – all generally augmented and/or created for the specific needs of a general-purpose software radio.

Surfer’s runtime kernel, Shark, is described in Chapter 5, including the overarching goals for handling runtime changes dynamically. Classes for handling buffers and the base for blocks are mentioned in passing – for completeness – while those for block runners and statistics collection are provided in some detail because they are new or significantly different when compared with other SDR implementations.

Moving up to higher layers, Chapter 6 contains discussions of signal processing flavors, and the extensions to the basic runtime kernel needed to implement them. We describe the functionality that must be exposed from the SDR kernel in order to allow subgraph flavors.

We devote Chapter 7 to the OpenCL interface developed as a more object-oriented approach to providing support for heterogeneous processing. This chapter talks about OpenCL’s standard interface, the need for more object-oriented approach, and the actual implementation we came up with. Also included are sections discussing data-flow specific additions to provide repetitive tasks, and examples of using the runtime compiler for handling different stream types within a single OpenCL kernel. We describe the block runners needed to handle integration with our OpenCL implementation.

Chapter 8 provides the justification for, and C++-based implementation of, a MATLAB-programming style algebraic interface for data-flow processing. This set of classes requires specific C++ knowledge, the basics of which are found in Appendix B.

Examples and uses of the Surfer framework are Chapter 9 along with plots of the throughput / latency curves for some interesting blocks. Of particular interest
is the cross-over points for different signal processing flavors – the “break-even” point at which one implementation becomes more efficient – more throughput for a given latency – than another.

Finally, Chapter 10 summarizes the work described in this thesis, and offers future work possibilities.
CHAPTER 2

BACKGROUND

In this chapter, we motivate the use of SDR via a brief history of wireless communications and SDR itself. There are many terms, concepts, and definitions required to understand the whole of software defined radio; thus we provide a brief introduction to SDR with emphasis on those parts most relevant to this thesis. Finally, we discuss current related relevant work and SDR implementations, and the layout of the remainder of this thesis.

2.1 A Brief History of Wireless Communications

Since the first wireless transmissions in the 1890’s [13 [54 [11], radio transmission techniques have continually evolved, providing people with the ability to be connected with ever increasing speed and data rates [119]. We have gone through the golden age of radio in the mid-1930’s [9] – using limited bandwidth for analog audio communications. Next up was the golden age of broadcasting during the mid-1950’s [10] when analog video transmissions were commonplace, taking more bandwidth but providing a greater user experience. As computers became smaller and more powerful during the 1960’s, they became useful in providing communications over long distances, using both wired connectivity via the ARPANET [74] (later to become the current internet) as well as wireless via ALOHANET [2].
Cellular phones came into existence around this time as well \[44\], allowing users to communicate by voice wirelessly from the comfort of their home or vehicle, but the original cell phones were awkward at best to lug around given their bulk and weight. Many modern cell phones are now a mobile computer, providing both cellular and internet access, and can perform wireless communications at rates unimaginable a generation ago.

Wireless local area networks (WLAN; now commonly called “WiFi”) have their origins in the 1985 ruling by the US Federal Communications Commission allocating unlicensed spectrum in three different regions for use in Industry, Science, and Medication (ISM) application \[144\] – 902-928 MHz, 2400-2483.5 MHz and 5725-5850 MHz. The original IEEE standard for WLANs was released in 1997 \[63\] – about the same time as the first general-purpose software defined radio. A recent IEEE standard, 802.22 \[64\], was released in mid-2011 and provides for using the parts of the spectrum used by the analog video transmissions mentioned above – 50 years after their golden age – to provide wireless regional area networks to rural areas where wired access would be prohibitively expensive. We are now moving into an age of ubiquitous access to information, primarily accessed via portable wireless handheld devices.

An interesting, and potentially problematic, commonality of all of the above devices is that their radios and protocols are virtually entirely hardware-based; they are minimally reprogrammable or reconfigurable at least with respect to radio functionality. This lack of flexibility is problematic in that, if there is a bug in the hardware, firmware, or software then there is generally no reasonable way to fix it – such devices would be susceptible to hackers for the life of the product. Further, such devices are limited in functionality to their hardware capabilities,
and cannot generally be reconfigured to execute other wireless protocols beyond what the hardware provides. *Software defined radio* aims to provide a solution to many of these problems along with many other benefits – as discussed in the following sections.

### 2.2 A Brief History of Software Defined Radio

The concept of a *software radio* can be traced back to the defense industry of the United States and Europe to around 1980. In the early 1980’s, the Digital Signal Processing Laboratory division of E-Systems (now Raytheon) in Garland, TX, USA produced a product they called “the Software Radio” [130]. This project used an array of sixteen specialized digital computers and one input / output controller to implement a reprogrammable software-based digital receiver.

The term *software defined radio* is credited to Dr. Joseph Mitola III in the early 1990’s [66], and refers to a class of radios that can be reprogrammed and reconfigured via software. Mitola envisioned an *ideal software radio*, with radio hardware consisting of just an antenna and analog-to-digital converter (ADC) on the receive side, and digital-to-analog converter (DAC) and antenna on the transmit side, with all other processing being handled via reprogrammable processors. As this ideal is not yet realizable, and likely will not be for some time, the term *software defined radio* (SDR) is used to describe a realizable device – one that is primarily *defined* by software, but still includes significant hardware. With SDR, the analog hardware consists of one or more down- or up-conversion stages, such that some intermediate frequency is placed within the capabilities of any ADCs and DACs. On the software side, a field-programmable gate array (FPGA) and/or digital signal processor (DSP) with specialized instructions is used to further re-
duce or increase of the data sample rate and/or move the signal from baseband
to some intermediate frequency within the hardware’s capabilities.

SDR has evolved mostly separately from the flow-based programming (FBP)
techniques developed in the mid-1960’s [133][102], though they are quite similar in
design-functionality. In FBP, an information packet containing data and meta-data
(e.g., control information) is generated and flows through a graph of interconnected
components. Each component can manipulate the information packet before it is
passed on to the next component, whether just passing it on, changing it entirely,
or creating a new information packet and disposing of the older one. In FBP,
each component’s state is carried along with it as part of its information packet,
and hence FBP was useful for performing tasks on multi-processor computers of
that time where shared memory was not available. Although FBP and SDR are
interrelated in the use of concepts, terminology, and algorithm implementations,
they still handle many runtime operations differently.

A “tipping point” in the evolution from FBP to SDR came about in the mid-
1980’s with the formalization of synchronous data flow [77][78] (SDF) techniques
for digital signal processing. By this time, computers and applications on them
were capable of significant digital signal processing, e.g., via MATLAB [141][140]
by following the techniques provided in Oppenheim and Schafer’s seminal DSP
book [110]. That said, computers of the time had very limited memory and
storage resources, and hence programmers tried to make every bit count. Lee’s
work showed that by setting minimal restrictions on a then-popular data-flow
implementation called “large grain data flow”, his SDF system could provide low
runtime overhead by statically generating component execution at compile time.
Thus, his SDR system provided a user-friendly application programming interface
(API) while also making efficient use of computational resources by utilizing the fact that in most data-flows, the sample rate is known a-priori at most components.

The first working SDR, called SPEAKeasy [76], was developed for the US Army between 1991 and 1995. Unfortunately, the software was not portable beyond the hardware for which it was developed and the overall system filled up the entire rear of a transport vehicle [15]. SPEAKeasy II [146] was a follow-up to SPEAKeasy, and achieved much greater success via advances in wireless communications hardware, and modular and reusable programming. SPEAKeasy II went from contract award to demonstration in a mere 15 months, and provided enough software-based modules to allow for new waveform implementation in a matter of days. Unfortunately, this program was not truly “joint” across all of the US armed forces.

An offshoot of the SPEAKeasy projects, the “Modular Multifunction Information Transfer System Forum” was formed in 1996 with Mitola at the helm, for the purpose of advancing the development of software radio in the US Military. In 1998, the Forum changed its name to the “Software Defined Radio Forum” and expanded its scope to include commercial and international participation. In 2009, the Forum again changed its name to the “Wireless Innovation Forum” to better reflect its mission of not just traditional software defined radio, but also the growing multitude of wireless protocols and systems including specifically cognitive radio and dynamic spectrum access. The Forum is now a “non-profit ‘mutual benefit corporation’ dedicated to driving technology innovation in commercial, civil, and defense communications around the world” [1]. The Forum has been influential in the development, acceptance, and uptake of SDR in both commercial and military industries throughout the world, and its members and meetings work
to address all levels of wireless technology.

Staying on the US military side of SDR, the Joint Tactical Radio System (JTRS) has been in development for more than 13 years \[98,109\] and offers substantially more flexibility and portability than either SPEAKeasy via the use of the software communications architecture (SCA) \[129\] – an internationally recognized open source specification. The current JTRS standard specifies both a portable OS interface (POSIX) \[68\] compliant API to provide standard access to threads and other OS services, and the common object request broker (ORB) architecture (CORBA) to provide standardized client / server operations when using cross-platform distributed processing. The SCA has been implemented on many hardware platforms and operating systems: any hardware and software providing the required SCA interfaces can, in theory, be used for JTRS purposes. JTRS radios are intended to interoperate with existing “legacy” military radio systems as well as all other JTRS radios, while providing significantly enhanced capabilities such as remote or local video display, ad-hoc networking, and access to local maps and other visual data. That said, JTRS is not designed for execution on general-purpose commodity hardware, and hence it is included solely for completeness; Virginia Tech’s Open-Source SCA Implementation::Embedded (OSSIE) project \[32\] provides an SCA interface to such hardware, and will be discussed later in this chapter.

In the non-military community, SDR execution on a general-purpose computational resource became reality in the mid-1990’s via the “SpectrumWare” project at the Massachusetts Institute of Technology (MIT) \[16\]. The “Spectra” SDR framework created by this project was heavily influenced by the benefits and drawbacks learned from SPEAKeasy as well as commercial research into software
radios. Some primary project goals were to use portable object-oriented software and leverage commodity components where possible in order to leverage “Moore’s law” [101] for the data-processing CPU(s), and also conceptually for the rest of the hardware electronics [13]. Dr. Vanu Bose describes the Pspectra codebase and overall project in his dissertation; the code was eventually made open-source and forked in 2001 to form the GNU Radio project [52][12]. Dr. Bose went on to found Vanu, Inc. [145], which also made use of the Pspectra code base for their primary product: the Vanu Software Radio, an SDR-based cellular base station and the first FCC-approved software radio [75].

Verbal history has it that GNU Radio was formed to provide a means to bypass the, at that time, perceived threat of the United States (US) Federal Communications Commission (FCC) broadcast flag [40]. The flag was eventually approved by the FCC [81], and is simply an instruction to the copying or distributing device; it does not alter the content in any way. When this flag is set, all non-exempt devices are obliged by US law to obey it and disallow content copying. Because this flag was required on consumer devices only, GNU Radio and its related hardware, the Universal Software Radio Peripheral (USRP) – being non-consumer oriented general-purpose software and hardware – did not have to obey this flag. GNU Radio has evolved since those early days to be one of the primary open-source SDR frameworks available; it will be discussed in more detail later in this chapter.

About the same time that Vanu, Inc. was formed, a SDR group at Trinity College, Ireland was developing its own framework, called “Implementing Radio in Software” (IRIS) [84], focusing on providing dynamic reconfigurability for the nascent cognitive radio movement [65]. IRIS takes a different approach to signal processing from Pspectra or GNU Radio, and is more akin to JTRS / SCA in
many ways; it will be discussed in more detail later in this chapter.

SDR has found users in academia, amateur radio, industry, government, and the military, with applications in areas as diverse as prototype development, magnetic resonance force microscopy, aeronautical testing, testing and evaluation of multihop communications, mobile multimedia broadcast transmissions, cooperative diversity in wireless networks, wireless network cross-layer prototyping, quantum optical communication, and in particular cognitive radio research and development for dynamic spectrum access.

In order to understand this thesis, a basic understanding of SDR is helpful. Unfortunately, SDR encompasses many domains; we provide an overview of SDR basics most useful for interpreting this thesis.

2.3 SDR Basics

To fully understand SDR, one needs knowledge from many domains: analog hardware, firmware and FPGA-based languages, digital computer hardware, operating system (OS) device drivers / kernel extensions, OS thread models, real-time systems, object-oriented and graphical user interface (GUI) programming, and digital signal processing (DSP) techniques. Most SDR developers and users specialize in just a few of these domains. For example, this dissertation concentrates on SDR runtime implementation issues, and thus involves knowledge of computer hardware through object-oriented programming, as well as DSP algorithm implementations. In this section, we provide the basic SDR concepts and terminology needed to understand this dissertation; Appendix A provides a concise listing of acronyms, abbreviations, and definitions of various SDR-related
terms. For more in-depth and/or analytical treatment of these subjects, consult any of the growing number of books on the subjects [120][36][8][53].

2.3.1 Block Abstraction

Both FBP and SDR processing techniques break down algorithm implementations into “small” computations, called a block, component, process, actor, thread, or fiber, depending on particular framework and implementation; for this thesis we will use block. Each block generally performs a simple computation, such as adding together element by element two vectors of 32-bit integers. Some blocks perform more complicated algorithms, e.g., frequency and phase synchronization for an orthogonal-frequency division multiplexer (OFDM) receiver. Chaining blocks together allows for the creation of simple or complex applications, e.g., an analog repeater, an HDTV receiver, or a cellular base station.

In general, signal processing algorithm implementations will differ significantly between different processing device types. For example, a multi-core CPU excels at handling small amounts of data over a limited number of streams, while a many-core GPU is better suited for large amount data and/or a large number of streams. As an example, a finite-impulse response (FIR) decimation-by-L filter is more efficiently implemented on a many-core CPU via its polyphase representation [113] – splitting the filter taps into smaller chunks, and then spreading the filtering across all available cores. This multi-core version is actually a sub-graph, made up of a demultiplexer, FIR filters, and a synchronous adder, as shown in Figure 2.1. On a GPU, it is generally more efficient to have each core generate a set of output samples using all filter taps because the computation of each output sample is independent of the computation of all other output samples, and thus
the synchronous adder can be avoided and the implementation can be massively parallelized.

One area of active work is in defining the scope of each block’s computation – how “small” or ”large” its piece of the algorithm implementation is. Its definition has changed over the years, and now with the almost ubiquity of single-instruction multiple-data (SIMD) – e.g., SSE, Neon, and Altivec – and related specialized instructions – e.g., a single instruction fast-fourier transform (FFT) – that can be used to accelerate computations, the definition is on the verge of yet another change. Current SDR frameworks provide different blocks for the various implementations of the same or related algorithms – e.g., a block each providing N-way summation, written for SSE, Neon, Altivec, Apple’s Accelerate framework, and a generic, non-accelerated version for when none of the others are available. In this thesis, we call each algorithm’s implementation is called a flavor. Recent research has shown that defining a single block with different flavors provides for a more robust user experience than one flavor per block. This topic in the Surfer context will be investigated in Section 6.2 with regards to the actual flavor concept design and implementation.

2.3.2 Waveform / Graph

Modern SDR frameworks implement a user’s application by connecting a set of computation blocks together into a directed graph, where the connections between blocks show the direction of data movement. Figure 2.2 shows such a graph for a narrowband FM demodulator where the incoming data is at a somewhat faster rate than the outgoing audio data, and hence a downsampler is required to reduce the sample rate. In SDR terms, each separate interconnected set of
computations is called a *waveform* or sometimes *waveform graph*; an application is made up of one or more waveforms, typically along with other files related to those waveforms.

Although SDR-style data flow is performed slightly differently from that found in FBP, both use this block abstraction to represent the processing of data. Some frameworks generate each application indirectly, via a markup language that defines the block, connections, and other variables, while other frameworks generate each application directly using a standard API via a scripting or compiled language, e.g., Python or C++. The use of a graph to represent data processing provides the ability to process different blocks using different computational resources, e.g., one block might be processed using the host computer’s CPU while another is processed using the attached GPU. This processing can occur sequentially or concurrently, depending on the types of available computational resources,
Figure 2.2. Directed graph representing a narrowband FM demodulator.

user preferences, and the actual waveform graph.

2.3.3 Data Flow

Each connection between two blocks represents a buffer, which will be discussed in some detail in the next section. Each buffer can hold a finite amount of data, and hence chunks of data must be processed by a block, removed from its input buffers / queues, and the generated data added to its output buffers / queues. The data removed might be a pointer or handle to an information packet, or simply changing a read offset but not actually “removing” the data. When data is removed from or written to a buffer, any blocks connected to those buffers are notified of the change, so that they can determine if they are “ready for processing”. In this manner of processing, data moves through the waveform graph from sources to sinks through buffers and/or queues. When doing real-time SDR, each buffer’s average data throughput in items consumed and generated per time unit must be at least as great as the data sample rate at that point in the graph; if not, a data overflow or underrun will eventually occur.

In FBP, there is no synchronous data flow or blocks state, and thus all information packets are the equivalent of asynchronous and independent events. The move to SDF in the mid-1980’s changed the flow paradigm to allow for synchronous data
processing as well as block state. That said, data was still produced by sources and pushed through the waveform graph to sinks. This form of data movement is called *data driven processing or push processing*; each data push is the equivalent of an asynchronous FBP event, but, because the data rate is known, the average data flow will be synchronous for successful real-time processing.

In the mid-1990’s, a new data movement model was introduced such that sinks requested data from their buffers, and the request propagated through the waveform graph to sources. The sources then generated the requested amount of data, which then propagated through the graph fulfilling requests back to the sources. This form of data movement is called *pull processing*.

Both push and pull processing have their place. Due to the additional overhead of pull processing for keeping track of data generation requests, the latter is well suited to static waveforms, where the amount of raw processing power well exceeds that required to keep up with all data sample rates. The former is well suited to malleable waveforms, e.g., ones that can adapt their graph to the incoming data. When a waveform can change, it is easy to lose track of data requests, but simple to determine if the changed block is ready for processing by examining its input and output buffers. Although most modern SDR frameworks do not easily allow for malleable waveforms, this is an active area of research that we hope to address as future work for *Surfer*. Because the determination of “when” to process a block in push processing can be reduced to just checking input and output buffers for the amount of data in them, this method of data movement is generally lower in overhead than pull processing.
2.3.4 Buffering

Buffers are a critical part of the overall functionality of any SDR framework. They must be large enough compared with the amount of data consumed or generated such that for real-time processing there are no data overflows, yet small enough so-as to not be wasteful of resources. Data is buffered either as a continuous chunk, holding frames or inline packets, or via a queue of references – pointer or handle – to chunks.

A buffer physically resides in memory somewhere, e.g., in cache or on-board the processing device, or possibly in global memory shared among multiple processing devices. Transporting data between buffers introduces latency into the signal processing chain, and hence data affinity – the knowledge of where data actually resides – can be used in mapping blocks. If adjacent blocks are handled on different processing devices, then data must generally be transported between them to be made available as input to the consuming block, which has the potential to add latency and reduce computational intensity. This topic is discussed further in Section 5.4.3.

Depending on the type of system, and buffer implementation being used, multiple blocks might write into or read from a single buffer. The latter is used commonly for most SDR systems, while the latter requires careful programming in order to avoid unintended data overwriting. In practice, SDR systems do not generally allow multiple writers per buffer, but do allow multiple readers.

The overhead associated with buffering, broadly, depends on whether mapping is performed a-priori or during runtime. For SDR systems using a-priori mapping, buffer sizes are generally also be computed beforehand. These buffers are generally static in size, and, because all timing and mappings are known beforehand, can
be implemented such that there is nominal overhead associated with using them.

For runtime scheduling, where timing and mapping are not known beforehand, the abstraction of choice is a circular buffer \[128\] – where the buffer internally keeps track of its read and write pointers, and whether there is overflow or underrun. This buffering style generates a small amount of overhead, but it removes significant complexity from the code using it. The primary goal in utilizing a circular buffer of a desired size is to make sure that, no matter where the read or write pointer is located, a real or virtual sub-buffer of the desired size is always available – ignoring possible overflow or underrun.

In a circular buffer, when data is written to it, the write offset or pointer is advanced to the address immediately following the written data. Likewise, when data is read from it, the read offset or pointer is advanced to the address of the next valid data sample. When there are multiple readers for a single buffer, the movement of the read pointer is changed such that the pointer advance occurs only when all readers have accessed the buffer. SDR does not generally allow for multiple writers, though with careful programming this scenario could be made to work. Instead, a block is used to perform whatever computation would required by the multiple writers, e.g., a multiplexer or synchronous adder. FBP does generally allow for multiple writers because data is in the form of information packets that are written into a queue and hence ordering is not as important as it is with SDR.

There are three primary ways of implementing a circular buffer. No matter which one is chosen, the buffer implementation must keep track of the read and write offsets and wrap appropriately in order to maintain the condition that a sub-buffer of the desired size is always available for read or write. The three types are as follows.
1. Allocate exactly the desired size: Data must be regularly shuffled around in order to meet the condition. Hence this buffer style is used only when absolutely necessary because of the added overhead associated with it.

2. Allocate two or more times the desired size: Data shuffling only occurs when the read or write pointer moves to within one length of the end of the buffer. Hence this buffer type is a trade-off between extra resource allocation and lower overhead.

3. Allocate exactly the desired size, and then use a memory map to replicate the buffer immediately following the actual allocation; a variant is to use OS kernel memory for the original allocation, and then two adjacent memory maps – using hardware-based memory management to mirror one section of memory into another section. Data shuffling will not be necessary, because the buffer effectively wraps around on itself from end to start. This buffer method trades off system resources – the memory map(s) and/or kernel memory – for overhead. When possible, this buffer type is used for its simplicity and speed.

A buffer is allocated to hold a given maximum amount of data, which in turn limits the processed chunk size. If the chunk size grows to more than half of the buffer size, then only one chunk at a time may reside in the buffer – which increases the chance of a data overflow. SDR systems that allow for variable chunk sizes must carefully check to make sure that enough output data space is available to hold the number of generated items for any specific processing. These systems might also allow for buffers to be resized to meet the new demands – adding some overhead in the reallocation, but reducing overhead for verifying output space.

2.3.5 Packets and Frames

A stream of data is moved through the waveform graph in chunks – information/data packets, or frames. A frame can be thought of as a vector of items, and a stream being composed of multiple contiguous frames; each frame in a stream can be of the same or different length. A buffer contains a subset of the streaming
data – typically at least a few frames worth. The use of frames is well suited for synchronous data processing where data history is important – e.g., a FIR filter – because data is contiguous and hence just-used data is already in the buffer. The length of a frame of data will vary from block to block, connection to connection and over time; there is, in general, no reason for frame sizes to be constant. Data transport using frames is generally synchronous – with a known nominal sample rate – or isochronous – with a guaranteed amount of continuously-available bandwidth for handling data transfers, no matter the nominal or actual data sample rate. All forms of SDR to date perform data processing using data frames; some also do packet-based processing.

A packet contains structure that describes various properties of the enclosed data payload, e.g., a header with the packet length, payload length, and a checksum for verifying the integrity of the packet contents. Packet movement might be as a chunk of contiguous data, or a pointer or handle (pointer to pointer) to such a chunk. Packets are generally used for asynchronous, event-based processing, but are also often used when doing system to system communication where the receiving end wants to verify the integrity of the received data. Data transport using packets is generally asynchronous – with no guaranteed sample rate – or isochronous.

2.3.6 Computational Intensity

In any SDR system, there is overhead time spent performing the mundane tasks required to keep the system executing smoothly, along with time spent doing actual signal processing. In a typical SDR framework, overhead tasks include, among others, keeping track of the amount of data in buffers for determining
overflow, underrun, and when the block is ready for processing. Computational intensity in these terms is the ratio of the time spent doing signal processing versus that doing overhead – but does not include idle time.

2.3.7 Chunks, Throughput, and Latency

Because chunks of data are buffered between blocks, both the buffer and chunk size must be determined so-as to allow processing to keep up with the rate of incoming and generated data. Determining these sizes can be performed either during runtime or before execution – at compile time or via some external application and then the results provided to the SDR framework \[61\]. Runtime size determination is discussed with respect to GNU Radio later in this section, and for Surfer in Section 5.2.3.

The throughput of a given buffer is the number of items passing through the buffer per unit of time. The throughput of a given block is how quickly it can process a given set of input items. If the block’s throughput is smaller than the latency involved in acquiring the samples to process, then the thread handling processing can either handle some other block’s processing or sleep waiting.

Processing smaller chunks generally results in lower average throughput, but also lower latency. There is a minimum processing throughput and latency bound because of system overhead, and some blocks also have a minimum chunk size in order to perform any processing. Processing larger chunks generally results in greater average throughput in terms of samples per second, but also higher latency in terms of the time between when a data sample is generated by one block to when it is consumed by an immediate ancestor block. The actual processing throughput curve will peak either due to the complexity of the algorithm implementation,
or the memory hierarchy holding the data to be processed. Because modern processing devices generally contain multiple levels of data caches, the average processing throughput will diminish somewhat from its peak value once the chunks are larger than can be accommodated by the various caches due to the overhead of fetching data from the computer’s main memory. There is a trade-off between processing throughput and latency, with the possibility of optimizing one when provided with a bound on the other.

Computational intensity for block processing is generally a non-decreasing function of the chunk size up to a peak, and then becomes non-increasing. When processing small chunks of data, the average computational intensity will typically be at or below 1 – meaning that as much, or more, time is spent handling overhead as doing actual processing. When doing real-time processing, in order to reduce the chances of data overflow or underrun, the average computational intensity should be much greater than 1.

One condition for traditional, sample-by-sample style signal processing to meet this requirement is to disallow direct graph cycles – in graph theory terms, we restrict the waveform to a directed acyclic graph (DAG) [26]. Non-data-flow based feedback is still possible, e.g., via asynchronous messages. Removing this acyclic restriction to allow for feedback of variable size chunks is an area of future research.

2.3.8 Signal Processing

SDR signal processing can be divided into two broad categories: real-time and offline. The latter is applicable when doing processing where the amount of time the processing takes does not matter; processing can be faster or slower
than the nominal data samples rate, if any is known, because it is the end result that matters. An example of this category is when reading from and writing back to files, and doing some signal processing in between – sometimes called *batch processing*. This category of processing does not generally involve radio hardware – though the signal processing might involve radio protocols – and hence is more aptly called *streaming signal processing*.

The other category – real-time – generally involves actual hardware – radio or otherwise – where some fixed sample rate(s) must be met by the data processing system, in both the analog and digital parts. “Real-time” here means that the processing can at least on average keep up with the rate of incoming and outgoing data – and, possibly, be much faster. Signal processing can take place in hardware (e.g., the radio frequency (RF) front and back ends), firmware (e.g., a reprogrammable FPGA), and software (e.g., running on a local host-computer’s CPU or GPU). When performing real-time signal processing, the average throughput for any input or output buffer will match that buffer’s data rate. The instantaneous throughput of a given block must be somewhat higher than the nominal sample rate to accommodate time spent performing overhead tasks.

SDR does not necessary require “hard real-time” processing – meaning that overflow or underrun will occur immediately if processing is not fast enough to keep up data sample rates. Assuming buffers are large enough to hold a few frames or packets – which is advisable when not using an a-priori determined schedule – then the processing of any single frame or packet can be slower or faster than the required sample rate(s) so long as the average processing speed over it and its adjacent frames or packets is at least as fast as the required sample rates. Thus, for example, processing of one frame might be preempted by the OS and result
in slower-than necessary instantaneous throughput, but so long as the next few frames are processed faster than required to get back to the average sample rate then the system will not lose data.

2.3.9 Scheduling and Mapping

Scheduling, in SDR terms, is determination of the chunks’ sizes to process for a given block at a given time. The scheduler must determine this amount and make sure that enough input data is available in each input buffer for the given block, while simultaneously enough output space is available in all output buffers for the amount of data that the block will generate from a selected amount of input data. SDR schedulers have been created to execute during runtime or determined a-priori by a compiler [78] or data-flow interpreter [39] [61]. When determined beforehand, the resulting schedule is static, but buffer sizes, latency, throughput, and any other measurable parameters can easily be “optimized” because the scheduling is done offline.

A more recent area of research is runtime scheduling, not just of how much data to process – as with GNU Radio, which will covered in Section 2.5.2 – but of the actual runtime mapping of blocks to processing devices. Given a number of flavors per block, or simply blocks implementing different flavors, the issue is which implementation to use – either selected at initialization or via dynamic re-configuration. Runtime schedules can be dynamic if the SDR framework supports such behavior – as Surfer does through the use of multiple flavors per block, selectable dynamically during runtime. Surfer uses a supervisor that performs the task of a runtime scheduler along with controlling the overall system’s runtime behavior; the supervisor API will be described in Section 5.5.
2.3.10 Reconfiguration

Cognitive radio requires the ability to reconfigure parts, or all, of the receive and/or transmit graphs in response to changes in the communications channel. For example, when using OFDM the transmitter might detect that certain carriers are already occupied, and direct its OFDM encoder to not use those. Later, perhaps, they will be able to be used again. Similarly, if the channel changes then the OFDM encoder might change the length of the prefix appended before transmission. These changes, and others, are in current standards such as 802.22 [64].

Many SDR systems utilize an FPGA for data processing, as a trade-off between high speed and low programmability. Much research has been done as to how to partially reconfigure the FPGA – keeping the currently executing parts in place, while rewriting those being reconfigured [30]. Some advantages of using a reconfigurable FPGA versus a standard FPGA include reduced power consumption, better hardware reuse, and greater program flexibility. The primary issue is the speed of reconfiguration – often on the order of 1-10 ms.

Reconfiguration ideally would not involved stopping data processing, but rather could be executed dynamically and seamlessly – no matter if the reconfiguration is being applied to individual block, waveform, application, radio, or network, in whole or in part [115]. The IRIS system was designed from the ground up with cognitive radio and reconfigurability in mind [135]; it provides the most robust reconfiguration options thus far of any SDR framework, from parameter based to individual components, to whole waveforms via a switch in the graph.

The current SCA specification does not provide for dynamic reconfiguration; it primarily supports static, a-priori determined configurations. Research using OSSIE has shown that dynamic reconfiguration is possible, though costly in terms
of system latency during the reconfiguration process [27].

Surfer can handle runtime reconfiguration of individual blocks – swapping in a new block and disabling and removing an old block – or whole waveforms. This processing style was already demonstrated in IRIS, though the systems were developed independently and without knowledge of the other. We discuss these capabilities in the context of Surfer in Section 5.4.

In order to properly execute a reconfiguration, a controller of some sort is required [56]. IRIS uses different controllers to handle internally and externally generated triggers, allowing both the application or the user to reconfigure the system based on each’s observations. Surfer provides a supervisor that executes global control over the runtime system, as described further in Section 5.5.

Another form of reconfiguration is changing the block mapping onto processing devices. The general field of mapping – in OS terms called scheduling – has been around since the advent of multi-processor computers [122][42], and has remained an active area of research through the decades since then [38][126][131]. All current SDR frameworks instantiate each block for a specific processing device, which remains static thereafter. Surfer offers a novel implementation of this ability, providing usually seamless switching between each block’s various flavors – which might involved moving the actual signal processing between devices. This topic is addressed in Chapter 6.

2.3.11 OS Abstraction Layer

All SDR frameworks make use of an OS abstraction layer in some form or fashion. Such a layer provides the same API no matter the underlying OS. POSIX is a popular example of such a layer, though there are abstractions above that as well.
Some well known projects providing various functionality include Boost \([14][70]\), Apache Portable Runtime (APR) \([6]\), and Qt \([117][100]\). The C++ standard library (STD; “std:” in C++ programming terms) \([69]\) is a more generic set of classes as of the C++03 standard \([21]\) and provides mostly non-OS oriented functionality; Boost replicates parts of the STL, as well as providing OS abstractions. With the release of the new C++ standard, called C++11 \([22][20]\), the C++ standard library will be expanded to include more OS-oriented abstraction – e.g., threads, mutexes, locks, and conditions.

Most abstraction layers provide significantly enhanced functionality compared with that found in the native OS. That said, each such added *required* dependency makes the SDR framework more challenging to port; keeping dependencies optional is best when reasonably possible. In order to reduce external dependencies, *Surfer* provides its own OS abstraction layer, called Salt, which will be discussed further in Chapter \([4]\). *Surfer* currently requires a POSIX compliant OS for concurrent objects, and the GNU Autotools for building. Otherwise, it has no required dependencies on other projects, but it can use other frameworks and projects for different flavor implementations.

### 2.3.12 Waveform Interfaces

All current SDR frameworks provide waveform definition through some combination of graphical and/or text-based programming – e.g., markup languages, interpreted scripts such as Python or XML, and compiled C++. Graphical interfaces have been developed and used for signal processing purposes since the late 1960’s \([133]\), and are still in active development today \([107][99][4][39][106][90][47]\). *Surfer* does not provide a GUI yet, but we do plan on implementing one; this is
left as future work.

All SDR text-based interfaces use a block-centric approach to waveform definition, where blocks are instantiated and then ports are explicitly connected between blocks. This two-step procedure allows waveforms to be created from source(s) to sink(s), vice versa, or, really, in any order. Because of this arbitrary definition order, block-centric programming allows for cycles in the graph – if the system can utilize it. As already discussed in Section 2.3.7, SDR does not allow for cycles due to the low computational intensity such constructs would create.

Block-centric programming has been used since the origins of data-flow style processing [102], and has been carried over through the transitions and evolution of data-flow over the decades. Surfer allows for block-centric waveform definition, but also provides for the dual definition style – assuming no graph cycles – called buffer-centric programming, which will be described in Chapter 8.

2.4 Other SDR implementations

The purpose of this section is to briefly list various SDR and SDR-related implementations through the past three or so decades. This listing is not meant to be complete (and, really, cannot be); instead, it is meant to show the growing popularity of SDR. The implementations most relevant to this thesis will be discussed in more detail later in this chapter. Projects are listed roughly in chronological order, as best could be determined through publications and websites.

- The Software Radio (1980 - 1985) [130]
- National Instruments - LabVIEW (1986 - present) [105]
- Massachusetts Institute of Technology - SpectrumWare (1994 - 1999) [16]
There are a number of other projects and companies that are related to SDR in that they provide or allow data-flow or stream oriented processing, but are not defined directly as SDR. Examples include System-C, Agilent VEE, Mitov
Software OpenWire [99], NoFlo [107], ParC [114], Synopsys COSSAP [136], Synopsys System Studio [137], Cadence Signal Processing Workstation (now Synopsys SPW) [23], CoWare Signal Processing Designer (now owned by Synopsys) [28], and Data Analysis Workbench [29].

2.5 Other Related Work

In this section we discuss the other specific relevant work related to this thesis that does not fit into the SDR basics section. We are limiting the scope of discussion to design and implementation oriented work, to match the content of this thesis.

2.5.1 SDR Using GPUs

General purpose use of a GPU (GRGPU) is a relatively modern field, about the same age as general purpose SDR [112]. Efforts have been made to utilize a GPU for SDR purposes [95] [73], integrate a GPU into an existing framework such as OSSIE [95] and GNU Radio [127] [82] [49] [116], and Microsoft has even submitted a patent application in the US for an SDR based upon a GPU [31]. Although projects not using GNU Radio or OSSIE are software based, the projects’ intent is not for general-purpose SDR-style data processing; they are written specifically for the task at hand, and happen to be software-based and thus “SDR”. Thus, we concentrate on those projects using GNU Radio and OSSIE, since they are general-purpose.

The GPU implementations have been for the most part successful, though the non-general purpose ones show greater performance gains than the general purpose ones. Those using GNU Radio, in particular, have at best broken even with respect
to not using the GPU at all; the first official GPU branch used CUDA \cite{49}, and made an effort to port all of GNU Radio, for a given waveform, to using CUDA – from buffers to blocks. The other implementations show a break-even point of typically 500 - 1,500 items, depending on the block type (e.g., FIR filter, N-way summation, FFT). Although GPUs provide significant raw processing power, getting data to and from them is a significant bottleneck that adds latency and thus decreases throughput. We discuss our GPU implementation in Section \ref{section:gpu} and show that the break-even point is comparable with that of non-general purpose implementations.

2.5.2 GNU Radio

The GNU Radio project is fully an open source SDR framework that was specifically designed for execution on general-purpose commodity hardware. GNU Radio can be compiled and used under many versions of Linux, Mac OS X, and Windows, as well as various other UNIX types; it is written in C++, and provides APIs for developing applications in either C++ or Python.

Reconfiguration in GNU Radio is minimally provided for. The user can use a selectable switch implemented similar to that in IRIS, or stop data flow, reconfigure, and then restart processing. Each GNU Radio block is defined for specific types (e.g., complex \texttt{< float >}), and is reconfigurable in only a parameter sense. Connections are based on the size of the data type, not the type itself.

GNU Radio is designed to handle homogeneous processing, using the cores available on the host computer’s CPU. A branch using the CellBE was created in 2007 \cite{50}, but has since been removed from the project. A GPGPU branch was created in 2008 \cite{49} with the goal of porting all of the GNU Radio runtime to a
GPU via NVIDIA CUDA. Although this branch had basic success in block-level speed improvements, it never showed waveform-level speed improvements when compared with using just the host computer’s CPU; it was eventually abandoned. A more recent branch called GRGPU [116] aims to allow for true heterogenous use of both the host computer’s CPU and GPU – porting just those blocks that will benefit from being processed on a GPU. It is too early to tell if this branch will succeed, but its initial offering is not showing significant speedups due to the overhead of transferring data and kernels between the CPU and GPU.

GNU Radio provides two schedulers which determine when a block is ready for processing, and how much data to process. The amount of processed data can change between block processing; it may be constant, but does not have to be. Both schedulers function in roughly the same manner, trying to consume as much input data as possible per function call. Sinks and sources are handled differently from processing blocks with both inputs and outputs, but, generally, processing will take place so long as the amount of available output space exceeds that which will be used by the amount of data generated via processing the current number of input items. It is possible to control the maximum buffer size, and thus the maximum amount of data processed at any time, but the user cannot further control this amount. There is a small, variable amount of overhead time spent determining the amount of data to process.

The original scheduler implementation used a single thread to loop over all known blocks, keeping data flowing. This scheduler version used the pull processing data flow model, and would handle all blocks in a “round robin” style until either the application was quit or no more progress could be made – no blocks were requesting or generating data. A new multi-thread capable scheduler was
implemented in 2008 [51] that uses a single thread per block (TPB), with each
thread’s sole execution being the scheduler and thence the block’s signal process-
ing method. This TPB scheduler, when signaled by an adjacent block that new
input data is available, or that new output buffer space has been freed up, runs
through one iteration of the check above to determine if processing is in order.

GNU Radio provides a mechanism for handling instantiations of the same
block, through a registry. Each flavor-like variant adds itself to the registry at
application init time, before “main” is called; those that could not be compiled
are not in the registry. When the overall block is instantiated, the variant in
the registry with the highest priority of the correct CPU architecture will be
instantiated and used. This registry model is currently in use for audio source
and sink only.

2.5.3 IRIS

Implementing Radio in Software (IRIS) is a project out of Trinity College,
Ireland, started around 1999, that focuses on providing dynamic reconfigurability
for the nascent cognitive radio movement [84]. All versions of IRIS are written
primarily in C++ [132], and have been ported to multiple OSs including Linux
and Windows; IRIS is not open-source, though access to it can be requested.
IRIS was rewritten in 2008 to “provide increased flexibility, support higher layers
of the network stack, and leverage the parallel processing capabilities of emerging
processor platforms” [134]. IRIS is a working, general-purpose SDR framework
implementation that has been demonstrated at multiple conferences since 2007.
This project is still in active development and, if anything, its reach and use has
only been expanding over the years.
IRIS uses XML to define components, waveforms, and applications, as well as the various controllers and managers associated with the system. Connections are also defined via XML files, and made in the same basic way as other SDR frameworks: instantiate a block, and then explicitly connect to others blocks. The use of XML in IRIS is roughly comparable to the markup languages used in SCA.

Three levels of reconfigurability are provided in IRIS:

1. Internal Component: An API for one or more parameters is provided, and changing these parameters can result in the block significantly reconfiguring itself internally. For example, when using an OFDM mapper, to change the mapping from binary pulse shift keying (BPSK) to quadrature pulse shift keying (QPSK); this change is relatively minor, as it does not impact the input:output relative sample rate change. IRIS also allows for the sample rate to be changed, e.g., modifying the prefix length in an OFMD cyclic prefix component.

2. Component: Components can be added and removed during runtime. When replacing a component, a switch can be used to change the data-flow path.

3. Switch: A switch can be used, for example, when multiple independent waveforms are desired to be in place and accessible by some internal (to the waveform) or external (from the user or other currently executing application) trigger.

The latter two types of reconfiguration are handled by special managers – reconfiguration or flow-controller, depending on if the trigger is external or internal, respectively.

Heterogeneous processing is available in IRIS, via the use of different “software wrapper” components. XML files define the signal processing components as well as the target processing device, and then the software wrapper for that device is used to handle the interfacing to those components – changing parameters, data transfers, and the actual signal processing. In this manner, IRIS can use the host
computer’s CPU, or an attached FPGA or CellBE processor; other processing
devices can be added by creating new software wrappers. IRIS, in theory, allows
the switching processing devices during runtime, via a switch as described above.

Some of the various engines provided by IRIS allow for runtime changing of the
amount of data being processed. That said, any changes can result in significant
overhead while the component and/or waveform is being reconfigured.

2.5.4 OSSIE

Open-Source SCA Implementation::Embedded (OSSIE) is a project out of the
Virginia Polytechnic Institute and State University from the Mobile and Portable
Radio Research Group (MPRG), starting around 2005 [32]. The goal of OSSIE
is “to provide an open source framework and tools for developing and testing the
behavior of SDR waveforms and platforms”. [25]. OSSIE is written in a variety of
languages, from C++ to Python, C to JAVA; the critical runtime functionality is
implemented in C++ for speed. It is still in active development and, if anything,
its reach and use has only been expanding over the years. OSSIE is a working,
general-purpose SDR framework implementation that has been demonstrated at
multiple conferences since 2007.

OSSIE implements key elements that meet the JTRS SCA version 2.2.2 speci-
fication [129], but it is not fully SCA compliant. Although JTRS’ SCA implemen-
tation is not designed for general-purpose commodity hardware, OSSIE’s is. SCA,
in a general sense, is a specification for defining components, their interfaces, the
overall waveform, applications, and all related properties, for performing SDR-
style data processing. SCA programming is very precise in all parts of the wave-
form definition, which represents an added burden on the waveform developer.
SCA (and, thus, OSSIE) uses special markup languages to define components, waveforms, applications, connections, buffers, and anything else needed.

SCA version 2.2.2 does not discuss reconfiguration in any significant way; hence it is only recently that research in that direction has been performed. Reconfiguration using OSSIE has been demonstrated, though it is costly in terms of system latency when performing the reconfiguration [27]. The SCA-Next standard is more targeted towards cognitive radio platforms, and hence should allow for some forms of dynamic runtime reconfiguration.

OSSIE and SCA can do heterogenous signal processing through the use of components (blocks) developed for different target processing devices. All components must be registered with and available through a CORBA server, and are explicitly requested via the waveform’s scripts. Components cannot currently be moved between processing devices during runtime.

Neither SCA nor OSSIE provide a means to change the amount of data being processed – this amount is defined in the markup files that define the application, and is generally optimized offline for each application.

2.6 Summary

SDR has come a long way in its roughly 30 years of existence. We expect the use and development of SDR to continue to grow because the technology provides quicker time to market, more reusable programming, and after-sale expandability with new waveforms and bug fixes. The number of projects either using or developing SDR technologies are growing year upon year, though new implementations seem to have dropped off in the past few years. The next chapter provides an overview of a new implementation – our Surfer SDR framework.
Surfer is a complex project, involving 5 top-level namespaces and some 100,000 lines of C++ code. To split complexity into more manageable groupings, we developed a layers model that is well aligned with actual SDR functionality and applied it to the Surfer project during class and namespace development. A number of goals were developed and refined in the roughly 3 years of project work, with enough success to result in a fully functional general-purpose SDR framework. We recognize that there is always room for improvement, and have plenty of future work to do on Surfer—some of which we talk about in Section 10.1.

In this chapter, we provide an overview of the Surfer SDR framework architecture and implementation—which was designed to allow for experimentation on dynamic changes in runtime system behavior. The general system architecture and overarching model used for project development are shown and described in Section 3.1 and then a number of project goals are outlined in Section 3.2. We discuss the next few chapters, which discuss the actual software design and implementation, in Section 3.3.

3.1 System Model

The Open Systems Interconnect (OSI) layer model for networks provided a useful way to separate the various networking concepts into those for specific
purposes. A similar layer model for SDR is shown in Figure 3.1 on the left, with higher levels of abstraction at higher layers. In general, only higher layers can access lower layers. This SDR generic version was developed via research into other SDR projects, through reviewing papers, dissertations, and source code when possible. Although current SDR projects are not generally logically separated into layers, their actual programming creates a virtual set of layers. The SDR layers model shown here represent the logical or virtual division of programming.

The right side shows, roughly, the translation into Surfer namespaces. This model cannot and does not exactly reduce Surfer or any other SDR system perfectly; some of the layers do contain specific functionality that requires interactions in ways counter to this model. That said, the vast majority of each layer’s functionality does work within this model and hence it is a worthwhile guide for system development. A brief description of each layer is provided in the following sections.

3.1.1 Hardware

This thesis deals with software implementations only, but hardware is integral to the functioning of any SDR system. There are a number of single devices providing the “radio” parts of the hardware, including, but certainly not limited to, Ettus Research USRP, Lyrtech Small Form Factor SDR, Rice WARP, and FlexRadio SDR-1000.

SDR benefits from commoditization of computing devices, Moore’s law for the ever increasing capabilities of these devices, and Koomey’s law for the ever decreasing energy requirements of these devices. Modern computing devices have resulted in relatively inexpensive, computationally powerful desktop computers
and capable handheld, battery-powered portable computers.

### 3.1.2 Operating System

SDR also benefits from open-source software and OSs as well as commercial ones. SDR frameworks are typically designed to work on Linux, Mac OS X, and Windows since those are the three most used OSs. Luckily, open-source developer tools are available to make such cross-platform development easier – not simple, but also not nearly as difficult as it was even 5 years prior. Such tools include the GNU compiler collection (GCC), CMake, Qt, and Eclipse. SDR implementations using C++ can also leverage the C++ standard libraries including the standard template library. There are numerous standards, open-source projects, and OS-specific frameworks – such as OpenCL, FFTW, Accelerate – providing useful capabilities.
All of that said, for embedded systems these tools are not always available, or are difficult to maintain. Hence, an OS abstraction layer is often used to provide identical capabilities to higher layers when the underlying OS does not provide them, as well as to augment those already provided with multi-use project-specific functionality.

For Surfer, we created Salt and naCL to provide these abstractions. Salt has no formal dependencies beyond a C++03-compliant compiler and the overall system build tools, though it will use the functions and classes found in the C++ standard library and Apple’s Accelerate framework if they are available. naCL is a split implementation, with the lower layer depending on, and providing an abstraction of, OpenCL, and the upper layers providing the APIs for data-flow processing. Chapter 4 describes the Salt namespace implementation, and in-depth description of naCL is covered in Chapter 7.

3.1.3 Runtime

The runtime layer is responsible for the actual data-flow processing: the creation of the data-flow graph, data buffering and transport between buffers, mapping of block execution onto physical processors, and scheduling for block processing. In this layer, data is generally treated as unformatted bytes because there is no need to have knowledge of what the actual data represents – that knowledge is only needed at higher layers.

The runtime system does require basic knowledge of the block API, and hence blocks are split into two layers. The lower layer is purely the API used by the runtime system, while the the upper layer is for signal processing implementations as defined by the user – e.g., filling out the required methods for a forward FFT.
Surfer provides the runtime layer via its Surfer runtime kernel, Shark, as a single namespace including all base buffer and block classes, with the exception of the naCL block runner which is included in the naCL namespace. This layer’s design and implementation is described in Chapter 5 and its naCL counterpart is found in Chapter 7.

3.1.4 Block Processing

The block processing layer is responsible for defining the top-level block APIs, from which all user-defined blocks will inherit; the glue to utilize and interact with the runtime layer is provided by the lower level block APIs. In this layer, data is generally represented as frames or packets. Each specific block has knowledge of the actual data type or packet format, because the block needs this information to do processing. All blocks must provide at least the following functionality.

- A method to verify the current topology – the number and type of input and output streams;
- A method to return the sample rate change from any input stream to any output stream, if known;
- A method to estimate the number of input items consumed per stream or the number of output items generated per stream, given the other number of items;
- A method to handle data processing; and
- Methods to start and stop the block, e.g., for an audio source to interface with the OS and tell it to start or stop streaming data.

Surfer defines the type-specific buffer, and base state, flavor, and block implementation classes in its corresponding layer, called the Surfer signal-processing framework, Spf, and residing in the Spf namespace. The naCL interface inherits
from the flavor and block interfaces provided in Spf, and augments them with methods specific to performing signal processing using OpenCL. Spf is discussed in Chapter 6, and the naCL expansion is found in Chapter 7.

3.1.5 Script

The script layer is responsible for defining the user-accessible scripting language interfaces to create and manipulate waveforms, and possibly the runtime system. Possible languages include C++, Python, and markup or similar descriptive text-based files. Scripts can be compiled for a target processing device, or interpreted either directly or indirectly.

Surfer provides two interfaces, both in C++, one for block-centric programming and the other for buffer-centric. The former uses the API from the runtime block layer, allowing blocks to be instantiated and connected in any order. This style of waveform definition is common to all other SDR frameworks. The latter uses a new interface, the Surfer algebraic language interface, Saline, which is covered in Chapter 8.

3.1.6 Graphical

GUIs provide a means for model based design of waveforms, using the “drag-and-drop”, “point-and-click” techniques available in almost any modern OS. Examples of current commercially-available products providing this functionality for SDR and/or related data-flow style processing include MathWorks Simulink, National Instruments LabVIEW, and Agilent VEE; open-source or academic projects include GNU Radio Companion, No-Flo, OpenWire, and Ptolemy. Surfer does not provide a GUI yet, but we do plan on implementing one; this is left as future
3.1.7 Application

The waveform developer’s application is defined in this layer, which can utilize all layers below it in some combination to define the desired waveforms and execute them. Some SDR frameworks provide a special class from which to inherit for defining applications, while others leave application development entirely up to the user. *Surfer* does the former, providing a *Surfer::app* class with methods that must be defined for creating block runners, the supervisor, as well as building all waveforms. The *Surfer* application API is discussed in Chapter 5.

3.2 Design Goals

An overarching design goal behind *Surfer*, beyond experimentation with the runtime system, is to move complexity from the waveform developer to *Surfer* developers, while maintaining a high level of functionality – yet still giving the user control over system behavior if desired. There are many ways, both obvious and subtle, in which this goal is being accomplished. Yet there is also plenty of future work discussed in Section 10.1 to make *Surfer* even more user-friendly and functional. This section provides a high-level overview of some of the goals we had going in to developing this project, where we succeeded, and where more work is required.

3.2.1 Background Dependencies

It is difficult to create a significantly complex project without using some other framework. POSIX is a useful API, providing a mostly-consistent OS abstraction.
available on many modern OSs. When using C++, the standard library is indispen-
sable and almost ubiquitous, being available with most C++ compilers. Boost con-
tains many useful classes and functions, but can be challenging to compile. Qt pro-
vides a robust graphical interface, if it is available for the host OS, and can be chal-
 lenging to compile if not available as a pre-compiled binary. Though these framewor-
k frameworks provide desirable functionality, they make a project less portable as well as increase the challenges for developers and end-user in compiling and using the given project.

Beyond the required build system – e.g., GNU Autotools, CMake, or QMake – and compiler, all other dependencies should be made optional so-as to keep depen-
dencies at a minimum. This goal was accomplished in Surfer in several ways, from providing multiple flavors each of which can fail initialization if their dependent framework is not available, to allowing the user to disable any dependency from within each application. Dependencies are discussed in Section 4.2.5 and flavors in Section 6.2.

3.2.2 Pool of Runners

Multi-threaded execution was a must, to utilize modern multi-core CPUs and many-core GPUs in a heterogeneous fashion. As threads are controlled by the OS kernel, they represent a potential bottleneck in processing even when they are assigned a high execution priority. The more threads in use, the more the OS kernel has to work to keep the threads running. Too few threads reduces the SDR system’s performance. The SDR system should provide a default number of threads used to handle data processing – we call these threads block runners, or just runners – but this number should also be user-controllable for specific appli-
cation. We have found through experimentation that the number of cores should be around half the total number of blocks found in overall application, which includes all waveforms – but, that the mapping and scheduling has a significant impact on this number. We will discuss some actual mappings in Chapter 9.

Because blocks and threads should be separate, the SDR framework should allow each block to have a runner affinity – a specific runner that will always handle this block’s processing. Blocks without affinity will be added to a runner of the correct flavor – e.g., local CPU or GPU via OpenCL. This affinity, combined with thresholds, would allow a pre-processor to review the waveform and determine an implementation that is optimal in some sense – and then specify these configuration parameters to the SDR framework.

*Surfer* accomplishes two of these goals through the use of block runners and their affinity settings as found in Section 5.3. Externally-created configurations are not yet natively supported, and are thus potential future work.

### 3.2.3 “Dumb” Runtime, “Smart” Supervisor

The SDR runtime kernel should have high computational intensity – the ratio of the time spent performing signal processing to that spent performing overhead. To maximize this value, the kernel should not use complex, variable-time algorithms if at all possible. Those algorithms should be located in a separate, non-performance critical thread such as the system supervisor. In effect, the runtime kernel should be “dumb” – moving bytes around and directing runners to handle processing when it is time to do so. A method to check for processing can be called whenever data changes in a buffer, and thus this part of the kernel’s processing can be distributed across multiple threads, further reducing the kernel’s
complexity. If all blocks have their runner affinity set, then the kernel actually has no work to do – all of its functionality has been spread out to the runners and the supervisor. The kernel will be used only for blocks without runner affinity, to have the supervisor set the affinity; after that, the kernel should be waiting for commands, so-as to reduce CPU usage. In Surfer, most of these goals have been met, as will be discussed in Chapter 5.

3.2.4 Connection Information and Use

Connections between blocks contain information that can be used to define certain system parameters or APIs. Each connection can be defined for a type – e.g., int, float, or complex < double > – and often the data sample rate can be determined via a source, sink, or other rate-setting block.

Most blocks’ input stream types are defined by the prior connected-block’s output. Instead of having to specify any connected block’s input types, a connection implementation could take this information from prior blocks. Such data type propagation requires that the waveform be defined from source(s) to sink(s), and that all prior blocks be provided to the new block when it is created. In this way, the new block can use the data types and user preferences embedded in the prior blocks to configure itself. Using this waveform definition methodology reduces the number of redundant user-specified parameters, and thus also the chances of adjacent blocks being configured to handle their streams differently. For example, if a block outputs packets of a given length, then instantiating any block reading those packets with the incorrect packet length will likely cripple the data-flow.

Not all blocks will benefit from this creation style, and thus the SDR framework should offer both as means for waveform definition. Surfer provides both styles,
the block-centric “instantiate and connect” approach in Section 5.2.1 and the buffer-centric approach in Chapter 8.

Another useful connection quality that is often available is the item sample rate. We partially implemented sample rate determination and use in Surfer, hence it is left as future work.

3.2.5 Frames and Packets

Data chunks that are processed come in the form of frames of unstructured data, and packets of structured data. Frames are generally used for synchronous data processing, and packets for asynchronous. But, packets are also sometimes used in over the air transmissions to allow the receiver to verify correct reception.

Some blocks use just one kind, while others can use either or both kinds. Frames will be buffered such that items are stored in sequential memory location, using a circular buffer. Packets can be placed inline in the circular buffer, or reside in their own buffer and a pointer or handle used as a placeholder in the circular buffer. Both types should allow for meta-data to be attached to a given sample or packet, and then propagated through processing as appropriate.

Surfer meets these goals, by providing synchronous data processing and movement because that is the core of any real-time SDR system, along with utilities and through-blocks to encode and decode Surfer-style packets. This topic is discussed in Section 5.4.

3.2.6 Reconfigurability

To support cognitive and dynamic functionality, the runtime system must provide the traditional reconfiguration of a single block, subgraph, or whole waveform.
Prior reconfiguration methodologies involve changing block parameters, or swapping the block out during processing with minimal to no data-flow interruption. Reconfiguration could also mean changing the throughput / latency processing point, or the mapping from blocks to processing devices, both during runtime. As these latter types had not yet been accomplished, we decided to take *Surfer* in these alternative reconfiguration directions, augmenting the prior reconfiguration methodologies.

Controlling the throughput / latency processing point can be accomplished by setting thresholds on each input and output buffer used in connections between blocks, and then under normal operating conditions processing exactly a threshold’s data. Each threshold can be changed during runtime, which changes the amount of data processed and thus the throughput and latency of the data-flow. This change is discussed in Section 5.2.3.

Changing the mapping from blocks to computing devices during runtime requires either swapping blocks targeting different processing devices, or just the signal processing implementation or flavor, inside the block. We believed that the latter would be workable, and in most cases provide seamless transitions between flavors. This goal was accomplished through the use of signal processing flavors and related changes, as discussed in Chapter 6.

### 3.2.7 Statistics Collection and Use

To control the runtime system internally, information is required as to how the system is running – e.g., throughput, latency, and processor load. Ideally, APIs would be available for handling the collection and use of these statistics. The statistics should be first and second order, and might weigh recent samples more
than old samples – possibly even just dropping older samples after some time.

Block throughput can be compared with the known sample rates in the waveform graph to determine how much margin each block has compared with real-time processing. The waveform latency can be measured against that required by a given standard, to make sure it is being met. The processor load should be available for the overall host OS, the Surfer process, as well as individual threads.

Using this information, a supervisor can determine if the waveform as currently configured is executing within the required specifications – and if not then warn the user and try reconfiguring an individual block or clusters of blocks in order to bring the waveform into compliance. This goal was accomplished through a statistics API as described in Section 5.5.1 and supervisor in Section 5.5.

3.3 Next Steps

Chapters 4 through 8 describe the implementations of Salt, Shark, Spf, and Saline, respectively. Readers more interested in examples of the runtime capabilities of this SDR system can find them in Chapter 9. The next chapter provides an overview of the Salt abstraction layer, with details where classes are significantly different than that of other similar frameworks.
CHAPTER 4

SALT : THE SURFER OS ABSTRACTION LAYER

With the overarching goal of providing a robust object-oriented interface to the underlying OS that could optionally use already installed frameworks, we created the *Surfer* OS Abstraction Layer, Salt. Salt is written entirely in C++, and currently comes in at around 20k lines of code (not including comments) and more than 100 separate classes. Salt programming uses inheritance and templates as much as possible to reduce the code footprint; all classes, functions, types, and variables reside somewhere in the `Salt` namespace. Many of the classes and functions provided in Salt are common to other OS abstractions, and will be listed only for completion. A full list of the Salt classes is provided in Section 4.1. Those classes and functions that are either novel or significantly augmented from that provided in other current frameworks or APIs will be described in Section 4.2. We finish this chapter off by summarizing the Salt abstraction layer.

4.1 Classes

In this section, we list all of the classes created for and used by the *Surfer* framework. Classes are grouped into common categories.

- Compiler: debug, empty class, global initialization, inline, item type, namespace, non-copyable class, standard types, stringify, varargs.
- Concurrency: condition, mutex, scoped lock, thread.
• C++ Standard Library Oriented: complex, double linked list, double linked node, indexed table, lookup table, math, pool, printing, queue (FIFO and priority), reference counter, string, vector.

• General: App, buffer (base, via kernel shared memory, circular, double) callback, 32-bit cyclic-redundancy check (CRC32), DynamicStructure, executable, packet.

• OS-related: context, file, CPU affinity (process and thread), endianness, error, kernel memory (shared, map, unmap), memory manipulation, number of CPUs / cores, page size, process, resource time usage (OS, process, thread), signals, sleep, thread key, time.

4.2 New and Augmented

This section provides descriptions of those that are new or significantly augmented when compared with what is available in similar abstraction layers such as Qt, apr, or boost. Some of the augmented functionality is generic but directed at data-flow style processing.

4.2.1 Application and Executable

Salt provides its own “main” routine, which calls App::run in its place. This method must be defined by the user’s program when using just the Salt framework; Surfer provides its own App::run, and thus also its own required method for “main” functionality. The App class provides a method called during program initialization to handle parsing of the user’s arguments, and an “atexit” method for handling cleaning up after “main” has returned.

The “executable” class allows the current thread to spawn a new thread, fork the current process, or create a whole new process. The new child executable is synchronized with the parent such that the child will have reached a certain point of execution before the method creating it returns in the parent thread’s execution.
Each executable has its own signal handling routines, message queues for inter-executable communications, and state; the inheriting class need only provide an `execute` method to use the new thread or process.

4.2.2 Debugging and Inline

All projects provide some style of debugging. `Surfer` allows for project-wide global debugging with different levels of verbosity, as well as file- and class-local debugging. Global debugging overrides local debugging, and the `inline` compiler keyword is disabled for any files in which debugging is enabled.

4.2.3 DynamicStructure

Given multiple flavors that provide execution on different processors and / or using different compilers, the block state must be made transportable between processor memories and cross-processor interpretable. A standard C++ class instantiation / C structure can be copied between threads of the same application, and even shared between different processes executing on the same processor / OS. But, in general, neither can safely be used by different processors / OSs, whether copied or shared in some common memory, due to differences in alignment requirements, type sizes, and endianness. Hence, a new structure class was developed to address these deficiencies; we designed this new construct to meet the following requirements:

- To allow for simple copying, all variables and their padding and alignment must be stored within a contiguous memory space;
- Each variable must be able to be aligned independent of all other variables as well as the memory space;
• The memory space must be resizable to accommodate changing array-style user parameters, e.g., the number of filter coefficients and string names;

• Both the C++ and C interfaces to variables must be consistent, independent of where the actual memory space is allocated or how it is sized;

• All variables must be available for accessing before and after resizing (not necessarily during), and all variable values must remain the same before and after resizing;

• Any variable can be dynamically added to, and removed from, the structure, without affecting the other variables;

• The C++ API should match that for scalars and C++ standard library (std::) vectors and strings, such that these variables are as close as possible to drop-in replacements for the standard C++ ones; and

• The resulting C structure must provide all of the information needed within its contiguous memory space, such that all variables can be found and interpreted on the host processor independent of any differences in endianness or type sizes.

Given the nature of this construct, we call variables using it *dynamic structure* variables. DynamicStructure variables can currently be of any scalar type, a string, or a vector. Future implementations will include multi-dimensional matrices and support for *Surfer* packets. An example of a block state using this construct is provided in Figure 4.1, including five variables of different types and how each relates via a handle (pointer to pointer) to the actual memory allocated for it.

The structure header information and glue necessary for variable interpretation are shown in their correct locations. The header is similar to that of packets as discussed in Section 4.2.9, providing the total structure length and offset to the first variable’s sub-header “glue”. Each sub-header contains the total memory allocation for the variable including this sub-header, the required alignment, and the offset to the first valid byte. Individual variable alignment inside the structure is provided knowing that many SIMD commands require their arguments to be aligned, but for many block states it can be ignored.
Each variable is accessed in C++ through its dynamic structure counterpart, internally via doubly-dereferencing the handle, does not in general via the middle-layer pointers because they are subject to change as variables are added, removed, or resized. No matter where the actual memory space is allocated, the variable’s value (scalar or array) remains the same through first copying the current value to the new location and then updating the pointer value; the handle value always remains valid once it is set.

When accessed in C or after copying to another memory address, *Surfer* provides routines to verify and fix the DynamicStructure to meet its internal alignment and size requirements. For example, it is easy for alignments to get invalidated through a simple memory copy if the host aligns to 8 bytes while the target aligns to 4 bytes. These routines validate the transferred structure, and if found invalid will reallocate it some the new memory location. Once validated, a new set of handles is returned to access internal variables.

Figure 4.1. Example DynamicStructure implementation.
4.2.4 Errors

Calling a system function that sets a global error number (typically, `errno`) is not generally thread-safe. Salt’s `error` class replicates the functionality provided by the host OS, and augments it to include `Surfer`-specific errors as well as thread-safe handling of the global error when needed. Given the inherently multi-threaded nature of `Surfer`, all use of system functions is required to use this error class. The actual system call is wrapped in a macro that uses a shared mutex – an OS-provided object that controls execution flow between multiple threads. The mutex allows only a single thread to execute the system call – and hence to set “errno” – at a time. As system calls in `Surfer` are generally made during initialization and by non-performance critical threads, we are not concerned with the overhead associated with the use of, and time spent waiting for, this mutex.

4.2.5 External Dependencies

Salt (and `Surfer`) minimally requires a C++03-compliant compiler along with GNU Autotools as the build system. The frameworks use for specific flavors – e.g., OpenCL, FFTW, Accelerate, and Portaudio – are optional and, even if found, can be disabled inside the user’s application. Common types used by the compiler (e.g., float, int) and OS (e.g., thread, mutex, condition) are abstracted to reduce the difficult in porting `Surfer` to different OSs; the availability and other properties of these types is determined during configuration. All OS functions are wrapped in C++, and C++ namespaces are used extensively to set `Surfer`’s APIs aside from the OS and other frameworks. In this way, background dependencies are kept to a minimum, allowing the end-user to concentrate on using, not compiling, `Surfer`.

67
Some Salt classes will use their eponymous namesake found in the C++ standard library if that library is available, but enough functionality is provided such that this library is optional. Specific functions are vector-optimized when an appropriate framework is both available and enabled – e.g., see the section on math below. Through the use of templates these functions have generic versions and thus the use of this framework is optional. External dependencies are checked for during configure, but are not enabled by default; each dependency must be explicitly enabled by the user’s application.

4.2.6 Global Initialization

Some OSs and external dependencies provide duplicate functionality, e.g., bcopy and memcpy to copy data from one memory location to another. Numerous functions are searched for during the configure stage; sometimes multiple potential versions are found for a given functionality, while sometimes none are found. Salt provides a global initialization class, which is executed just before the call to App::run, and provides a means for runtime checking of such functions.

All C++ compilers provide a mechanism for initializing global variables before “main” is called. Salt duplicates this ability while allowing for specific ordering of the initializations, using a directed acyclic graph to represent dependencies. A global “init” and “atexit” class can be defined for any need, such as setting global variables and verifying that system routines actually function as expected and if not then checking other system functions and/or reverting to a generic version.

Salt current provides built-in classes to check for, and when possible verify the functionality of, the following classes.

- context manipulation
• CPU affinity
• CPU load: overall; for user, system, idle, and “nice”.
• error: thread-safe global error number
• memory: functions copy, zero, and set
• message: initialization of the standard inter-executable ones
• mutex: solely as a dependency for other classes
• number of CPUs available: OS fixed value
• page size: OS memory fixed value
• print: setting the function, depending on how far into the boot process the application is;
• reference counter: initializing the global table that holds all references
• time: determining “get time of day” equivalent functionality.

Any of these classes can also provide an “atexit” method which is executed in the reverse graph order from the “init” dependencies. This method allows the class to perform garbage collection, e.g., the reference counter will delete the global table, which deletes any left-over variables created and when discrepancies are found prints an error message.

This class completes the process of “finding during configure time, building into executables during compile time, and verification during runtime” used by Salt for most externally-provided functions. That said, for some external frameworks only specific functionality is tested for, and if found the other functionality is presumed. This partial testing is a reasonable compromise between no testing at all and exhaustive testing which would be a significant burden on the programmer.
4.2.7 Item Type

The C++ standard library provides a mechanism for determining and comparing the type of a currently-valid variable via the typeid facility; more information and examples of this C++ capability are provided in Appendix B. Salt contains a item_type class that uses this facility to create an object representing a type. This class provides easily modified methods for parsing the mangled strings generated by typeid, and currently determines type information for most combinations of the Salt classes and types for complex, packets, pointers, scalars, and vectors. Determination of a type’s characteristics is heavily relied upon by Saline, as will be discussed in Chapter 8.

4.2.8 Math

Many of the math functions provided by the C++ standard library cmath are augmented to provide specialize-case and vector versions. For example, the basic min function is provided for arbitrary types, via

\[
\text{template } < \text{typename T } > \\
\text{inline static T min} \\
(T \text{ left}, T \text{ right});
\]

and new functions were added to handle other useful cases, such as setting a variable to the minimum of itself and another variable (min_equals) as well as for finding the minimum of a vector.

\[
\text{template } < \text{typename T } > \\
\text{inline static T& min_equals} \\
(T& \text{ left}, T \text{ right});
\]
template < typename T >
inline static T min
(const vector < T >& vec,
 size_t* index_of_first_min = 0);

template < typename T >
inline static T& min_equals
(T& left, vector < T >& right);

If Apple’s Accelerate framework is available and enabled, then it will be used via
template specialization for float and double types for finding a vector minimum.

#ifdef USE_APPLE_ACCELERATE
template <> inline static float min
(const vector < float >& vec,
 size_t* index_of_first_min = 0);

template <> inline static float& min_equals
(float& left, vector < float >& right);

template <> inline static double min
(const vector < double >& vec,
 size_t* index_of_first_min = 0);

template <> inline static double& min_equals
(double& left, vector < double >& right);
Similar functionality can be provided for other vector-oriented frameworks, through the use of a priority-based function registry. We leave that as future work.

4.2.9 Packet

Salt provides a simple and generic class to handle packet building and retrieving. A packet structure will be built inline in a provided buffer, and can then be manipulated by the application. Likewise, provided with a buffer the packet structure can be retrieved and verified. The packet header consists of 16 bytes of data: the total packet length in bytes (4), the type of packet (2), flags being passed along (2), alignment to the payload start address (2), the actual offset in bytes to the payload start address (2), and the payload length in bytes (4). A CRC32 will be appended after the payload when a built packet is finished. The header information allows a packet to be copied while maintaining alignment if using the class’ copy constructor; if some other means is used to copy a buffer, alignment cannot be guaranteed. That said, the actual packet size can generally be set such that alignment is maintained.

For the purpose of data-flow style processing, inline packets will be required to be built in contiguous memory locations. Hence this class provides methods for finishing the current packet and building the next one in contiguous memory, as well as disposing of the current packet and retrieving the next one from contiguous memory. Other methods return a pointer to the header, any of the header’s individual values, a pointer to the payload, and for setting the packet type and
flags. A special method is provided for checking that a retrieved packet is valid – that the header is valid, its values are self-consistent in length and alignment, and, optionally, that the CRC32 is correct.

4.2.10 Printing

Because printouts can occur from any thread, Salt provides a mechanism for thread-safe printing such that no two threads can print at the same time. Printouts are generally used for informational purposes, and are thus are not performance-critical; during non-debug system operation, very few printouts are made. Hence, a special thread is created during application initialization that contains a queue; as print commands are issued, they are placed into the queue and then the thread signaled. The enqueued item contains the process and thread that own this print command, a timestamp for when the command was issued, and flags for which parts to print out. During normal operation, this thread spends most of its time waiting for the signal that a print item has been enqueued. Upon receiving this signal, the thread prints out all available queued items and then goes back to waiting for a new one to come in. In this manner, printouts using this class will not overlap, allowing debug output to be more easily parsed. Salt uses \texttt{std::cerr} when it is available, and \texttt{printf} otherwise, for performing the actual printouts. A global initialization class is provided for printing that is dependent on “time” being initialized. Before initialization, printouts will not include a timestamp but are thread-protected.
4.2.11 Statistics

As statistics are an integral part of the Surfer runtime system, it is important that they are robust and provide a variety of functionality. An eponymous class was created to handle the collection and computation of first and second order statistics for all provided samples; options include several combinations of timing, weighting, and windowing. The timing class collects the first and second order statistics for both the samples as well as the time of arrival – allowing the easy computation of throughput. The windowing class collects only a given number of samples before discarding the oldest ones, and is useful when only recent samples are trusted. The weighting class using a simple constant weight value in (0, 1) with the exponent being the distance in samples from the current one; more complicated weighting classes can be created by inheriting from this base class and overloading the method used to update the weight for each sample. Statistics collection objects are used by a supervisor to provide the information necessary to make decisions to control Surfer runtime behavior, as will be discussed further in various sections in the next chapter.

4.3 Summary

This chapter described the Salt OS abstraction layer, which both replicates functionality typically provided by a framework such as boost and the C++ standard library or uses some of those frameworks when they are available and enabled. The Salt API is near the minimum required to implement Surfer and its various capabilities, as found in the next four chapters. We start this implementation discussion in the next chapter with the Surfer runtime kernel, Shark.
The core of Surfer is found in its runtime kernel, Shark, which will be discussed in this chapter. Shark currently weighs in at around 6k lines of code, not including comments, implementing around 30 classes. As the name implies, Shark is fast but not particularly smart – the “smarts” are left to a supervisor that the waveform developer creates. Shark is about handling bytes, not items, quickly and efficiently yet with flexibility. Although the waveform developer never has direct access to Shark internals, the application’s supervisor does and can modify system runtime behavior, e.g., by using statistics to determine a new mapping from signal processing block to processing device, or changing the throughput / latency processing point of a set of blocks.

All of the runtime capabilities needed for SDR are provided by the Shark layer, in a manner that gives good default performance yet is configurable. As shown in Figure 5.1, Shark defines five groups of classes: application, blocks, block runners, buffers and related interfaces, and the supervisor. The Shark kernel maintains lists of all created objects, providing a means for printing out the state of the system or creating a GUI to display the state; this latter is the goal of future work. These groups, and specific classes in them, are discussed in the next sections, followed by a summary of the Shark layer.
Figure 5.1. Class relationships for the Shark namespace implementation.
5.1 Application

The Shark app class provides methods that the waveform developer must define in order to create block runners and a supervisor, and define all waveforms. Shark defines the Salt App methods required by that library for linking, passing on the “main” functionality to the Shark App’s run method. This App is an executable, residing in its own thread. The same function that creates this App also creates the print thread, a system information thread, and the actual Shark kernel. This kernel is actually executed in the program’s “main” thread. Surfer Apps are a simple object-oriented abstraction away from “main”, providing over-loadable methods for parsing, execution, and handling cleanup during exit, and the minimal C++ glue to hold it all together.

5.2 Blocks

We split blocks into runtime and signal processing methods, to remove unnecessary complexity from the block development side. The signal processing Spf blocks are discussed in Section 6.5 and primarily provide an API for developing actual signal processing blocks. Shark blocks provide the core APIs necessary for developing a waveform: the means to connect blocks together via buffer readers and writers, and for determining when any block is ready for processing. There is a 1:1 correspondence between Shark blocks and Spf blocks, and both know of each others’ methods. Hence the runtime block’s methods that handle checking for processing will call certain Spf block’s methods, and if an Spf block is reconfigured it can access and change its corresponding Shark block.

The Shark block API provides methods for controlling the state of block execution: starting, stalling, resuming, and pausing. The state of a block – started,
running, paused, or stopped – is kept and available via similarly named methods. A block is started just once, when all flavors of the paired Spf block have been initialized and the block passes its topology check via a method provided by the paired Spf block. The block is stopped when it is deleted or when Surfer stops. The start and stop commands are handled locally and then passed to the Spf block, which in turn passes the command on to all flavors. A block can be paused and resumed any number of times, and these commands impact the paired Spf block and the currently active flavor only.

As connections are made, the Spf block will be queried as to whether the topology is valid; if it is, then the block will be resumed if it not already running. If the topology is not valid, then the block will be paused if it not already so. In this manner, Surfer is a “live” SDR system, processing blocks as soon as data is available even while other parts of the waveform graph are being created. Obviously, nothing will happen until a source is hooked up to generate a data-flow. The application can disable this auto-run feature on a per-block basis.

5.2.1 Connections

The Shark block API has methods following the traditional SDR block-centric programming style, allowing a block to be connected to or from another one. The caller must supply the end-point reader and writer, and can optionally specify the thresholds and port numbers. If thresholds are not provided, they are set to default values and can always be changed later. If port number are not provided then they are set automatically to the next available.

Block connections in this layer can be created and removed in any order. When they are not created sequentially – either via explicit port values or using defaults...
– any unconnected ports between used ones are attached to a stalled buffer, which
is discussed in Section 5.4.8. Those unconnected ports can be connected, again
in any order, in which case the stalled reader or writer is replaced by a valid one.
Currently valid connections can be reconnected at any time, though the system
will print a warning about overwriting a valid connection.

When a connection has been requested, a common method is called to deter-
mine if the new connection would make a cycle, and if so then it is disallowed.
Surfer does not currently allow for cycles, hence this is a topic of future work.

Connections can also be removed at any time; like many other Surfer classes,
connections between blocks are implemented using a multi-client / server model.
When removing a connection that is attached to the highest port number, the con-
nection is simply removed, the maximum port count decremented, and all related
structures resized to the new port count. When removing any other connection,
the stalled buffer is put in its place.

5.2.2 Connection Parameters

Connections between blocks are made using a reader and writer that can be
of the same or different types. Internally, a Salt item_type is used to provide
easy access to the type’s properties: basic name, size and endianness, and whether
it is complex, a string, packet, a vector, or some combination. A wildcard type
is provided and used for readers only, meaning that this connection will accept
any data type. An OFDM mapper block can be configured to take any type of
data, and hence can use the wildcard type for its single input buffer; it can also
be configured to take Spf packets, in which case the packet type is used.

Although each connection can have a nominal sample rate, this value is cur-
rently not set or used. We propose to do so as future work, integrating the sample rate detection mechanism into the Saline interface.

5.2.3 Thresholds

Each input and output to a block is provided with a threshold that can be changed during runtime, and represents, at this level, a number of bytes. When the threshold is initially set, the desired value is stored for later use, the Spf block method for verifying threshold values is called, and the returned threshold is then used. The Spf block’s method must be used because some blocks require a block of data at a time – e.g., a ‘packet converter’ requires a packet’s worth, and an ‘FFT’ requires and FFT’s worth. A ‘sum’ block can generally take any number of items, though specific flavors might require a specific number at a time for optimal performance. The stored threshold is used when the Spf block needs to change the actual threshold value due to a change in setting, e.g., upon determination of the actual packet size when the first packet is successfully received. As will be discussed in Section 9.1.1, threshold values are set independently of the buffer’s data sample rate yet are directly related to the input to output latency.

By using a threshold instead of an algorithm, determining when a block is ready for processing is reduced to a simple value comparison across all buffers. Using the variables shown in Figure 5.2, $T_i$ is the input threshold, $T_o$ is the output threshold, and $N_b$ is the amount of data in the buffer connecting the blocks; assume that each value is in bytes. An input threshold is met when the amount of buffered data is at or above the threshold ($N_b \geq T_i$) while for an output the threshold is met when the amount of buffered data is at or below the threshold ($N_b \leq T_o$). Once a threshold has been met, it will remain met until the block
is processed; hence, a simple counter is used, holding the number of buffers left to threshold. When a buffer meets its threshold, the counter is decremented and compared with 0. If the counter becomes zero, then the “ready for processing” sequence described in Section 5.2.5 is entered. The number of stalled buffers is pre-decremented from the possible total number of buffers left to pass threshold.

By moving the determination for when a block is ready for processing into a separate thread, the time required for this determination becomes a constant, on average. Further, a separate thread – in this case the system supervisor – can set threshold values on a block-by-block, cluster of blocks, or whole waveform basis; it is not limited to just a single block. The supervisor can use a-priori determined values for thresholds, or perform some local, cluster, or global optimization to determine them.

Surfer initially allowed arbitrary settings for both input and output thresholds [34]. For a reasonably complex graph, we regularly ran into situations where the thresholds were set such that processing would deadlock without carefully checking threshold settings. An example of a potential connection deadlock is found in Figure 5.2 showing the situation where the output threshold – the red line – in block 1, $T_o$ in bytes, is set lower than that of the input threshold for block 2, $T_i$ in bytes. There is just a single connection, using a buffer holding $N_b$ bytes, and it cannot be simultaneously true that $T_o < T_i$, $N_b \leq T_o$, and $N_b \geq T_i$ – with the latter two conditions being those for meeting the output and input threshold, respectively. For this situation, assuming that neither threshold was already met, then neither block’s buffer will meet its threshold and neither block can perform processing; both blocks will deadlock. In the situation where $N_b \leq T_o$, deadlock will not occur so long as the number of items added to the buffer always results
in $N_b \geq T_i$, or if either block is self-processing (as discussed below). Surfer checks for these conditions as best it can, and warns of potential deadlock cases – but it still allows all threshold settings.

After some consideration, we determined that setting either the input or output threshold would usually accomplish the same task as originally desired. For blocks where the relative sample rate change from input to output is greater than 1, the output threshold defines when the block is ready for processing; when this change is less than 1, the input threshold defines when the block is ready for processing. An example of the former is an OFDM mapper, where incoming bits are mapped to constellation points of some, generally complex, type; under most circumstances the relative sample rate change will be much greater than 1, and hence it is possible to generate output without consuming input. Source blocks, of course, require only output thresholds while sinks require only input thresholds. Most blocks can use either threshold, but it is important to provide the ability to set both threshold values and then determine which the Shark block will use once the Spf block passes its topology check.
5.2.4 Self-Processing

Self-processing blocks are ones where data is created elsewhere and pushed into the graph via a thread separate from Surfer, or one that require the processing method to sleep while waiting for an event or time duration. These blocks are generally sources, but all sources are not necessarily self-processing. Most blocks with both inputs and outputs are not self-processing, though there are exceptions. Self-processing blocks are allowed to estimate that no data will be consumed or generated without being marked as finished. Their processing method is still called such that the block can do pre-processing and post-processing, but whatever the processing method returns is ignored. Although this self-processing technique can violate thresholds, we see it as a worthwhile trade-off for the added functionality and reduced complexity.

5.2.5 Processing Sequence

The block’s thresholding sequence is show in Figure 5.3. The “Wait for Notification” step is not a literal step; blocks do not execute in their own thread, so this step is just an entry and exit point. When a block finishes processing, it updates each input and output buffer, in turn, with the number of items consumed and generated, respectively. During this update, the buffer sends a notification to the attached writer, and all attached readers, that the buffer has been updated. This notification is the starting point of the flow chart. This processing sequence is valid for both self-processing and normal blocks.
Figure 5.3. Threshold processing sequence.
5.3 Block Runners

Also already mentioned in Section 3.2.2, Surfer uses a pool of block runners to handle block processing. When combined with flavors, block runners allow for arbitrary mapping from the virtual waveform graph topology to physical processing device topology. With enough runners, waveform execution can be just as fast as a thread per block model, but with less resource utilization.

All block runners inherit from the Shark base runner class that contains a priority-based queue for blocks ready for processing by this runner. The base runner class’ only public method is for appending a block to this runner’s list for processing. Because each block runner executes in its own thread, it must be created using a static friend method defined inside the class, so-as to provide initial parent / child synchronization. The actual runner implementation defines its own creation function, which must minimally start the new runner to do the aforementioned synchronization. This creation function can also perform other initializations or tasks, as needed by the runner’s interface to its target processing device.

The waveform developer’s application controls the number and type of runners created, any initial affinity between block and runners, and whether to allow setting runtime affinity for those blocks not assigned affinity a-priori. Once running, each block runner executes a simple sequence, as shown in Figure 5.4. The initial step, “Wait for Notification”, unlike that for the block processing sequence mentioned above, is actually waiting on a condition to be signaled. Once notification occurs, the runner will continue processing until there is nothing left in its queue. All of the steps in this figure occur in the specific runner’s thread. All runner threads by default are waiting for input and hence generating no load except for
when performing processing.

The two block runner types provided by Surfer handle processing on the local CPU and via naCL. For the local CPU, the base class already defines the primary sequence for handling blocks on the local CPU, but must provide the initial creation and initialization. The naCL runner is discussed in Section 7.3.1. Both block runners are very simple classes.

5.4 Buffering

Surfer’s buffering classes are divided in specific function units, in order to enable reconfiguration through dynamic addition and removal of blocks. These classes are the buffer itself, a buffer master and writer, and one or more readers.
An example setup is shown in Figure 5.5, where block 1 has been connected to block 2 via the writer, master, and reader 1, and block 1 to block 3 via reader 2. This figure shows the unidirectional data path, and the bidirectional control paths. The actual classes are discussed further in subsequent sections.

5.4.1 Types

Buffers are used to store data of any type – scalars, vectors, matrices, packets, whatever. At the Shark layer, data is manipulated as bytes. At the Spf block layer, macros are provided to place types on the buffers, based on the block’s processing requirements; this topic is discussed further in Section 6.1.1. At the Saline script layer, all input and output streams are strongly typed to make sure that block inputs and outputs are the anticipated type, as will be described in Section 8.2.3.
5.4.2 Base

The base buffer class provides the API necessary for defining, mapping, resizing, unmapping, and otherwise manipulating buffers – but, it leaves the actual implementation up to the inheriting class. Typical actual implementation will be single or double, mapped or unmapped; we choose a double mapped variety that is defined in Spf, as discussed in Section 6.1.2.

Internally, each buffer must keep track of a number of parameters, including the desired and actual buffer length, amount of saved space, and the wrap offset – the offset beyond which any advance due to reading or writing will be wrapped back to the buffer start. The desired buffer length and amount of saved space are changeable during runtime; the other parameters are determined by the inheriting class as they are implementation dependent. Figure 5.6 depicts an example buffer with these parameters laid out. Each parameter’s value can be obtained, in bytes, via a similarly named method. Any inheriting class must define methods returning these values in items.

This class provides public methods for attaching and detaching the buffer with a master, as well as for advancing an offset by a given amount. This last method is dependent on the actual implementation, and thus is left to the inheriting class; it is used to advance a read or write offset, to handle wrapping and saved space. The goal of Surfer buffers is to always provide the desired buffer length’s worth of valid memory.

5.4.3 Affinity

Buffer device affinity is important when determining the mapping of a block to a processing device. Surfer uses a coarse means for determining affinity, by keeping
track of the processor on which the data resides – e.g., the local CPU, the GPU via naCL. When a block is ready to be processed, data not already on the current flavor’s processing device must be copied or mapped to it. This coarse affinity is used to determine which buffers must be copied or mapped. Data already on the target device is accessed directly, without having to copy or map it, because the flavor is executing on the target device. The actual copying / mapping is left up to the specific block runner and flavor base for the target device.

5.4.4 Masters

As shown in Figure 5.5, the buffer master is at the heart of Surfer’s buffer system. A master is attached to a buffer, writer, and one or more readers, and provides the synchronization point for control messages. The master is programmed such that readers can be added and removed dynamically without impacting data-flow to other readers, using a multi-client/server connection model. The buffer can be swapped out, though this situation happens primarily only when it is necessary to reallocate the buffer due to a command to resize it. The writer can also be
swapped out, via a single-client/server connection method. This reconfiguration is used by subgraph flavors, which will be discussed in the next chapter.

The master retains knowledge of the attached buffer and writer, all readers, the writer offset, and the base pointer where the buffer is mapped to in local memory. Methods are provided to retrieve parameters, access the buffer and its parameters, and determine the maximum amount of available space – which loops over all attached readers querying each in turn for its available space. Because multiple readers can be attached to a master, and readers are designed to be dynamically attached and detached, it makes more sense for each reader to keep track of its own offset.

5.4.5 Readers

Each buffer reader keeps track of its dequeue pointer, which is the starting memory address at which data can be read. This dequeue pointer is provided to the reader when it attaches to its master, and is updated when the reader’s advance (+=) operator is called by the Shark block which is using this reader. The advance operator updates the reader’s copy of the its dequeue pointer, then notifies the master. Multiple readers can be attached to a single master, and thus the master’s knowledge of the reading offset is not changed until all non-stalled readers have signaled consumption of data. The actual change in offset is the minimum of all readers’ advances, taking into account type of buffer, the amount of saved space, and the wrap offset – all qualities handled through the master.
5.4.6 Writers

Each buffer writer keeps track of its enqueue pointer, which is the starting memory address at which data can be written. This enqueue pointer is provided to the writer when it attaches to its master, and is updated when the writer’s advance \((+=\)\) operator is called by the Shark block which is using this writer. The advance operator updates the writer’s copy of the its dequeue pointer, then notifies the master. A single writer will be attached to a single master, and thus the master’s knowledge of the writing offset is changed when it is notified of the advance. The actual change in offset must take into account type of buffer, the amount of saved space, and the wrap offset – all qualities handled through the master.

5.4.7 Control

Figure 5.5 shows control paths as well as data paths. Control is bidirectional but, as always, the master is in the middle of it all. When a reader is updated, this notification is propagated through the master to the attached writer and thence to the writer’s block. This block will check the connection’s port to see if its threshold has been met. When a writer is updated, this notification is propagated through the master to all attached reader and thence to each reader’s block. Each block will then check the connection’s port to see if its threshold has been met. The use of advances and notifications replaces the singular “push” and “pull” of older data-flow implementations. When using multiple threads for block processing, each block pushed data into output buffers and pulls data from input buffers. No requests are necessary, because they are replaced by notifications from the attached master.
5.4.8 Stalled

During *Surfer*’s boot sequence, a stalled buffer is created, and attached to it are a stalled reader and writer. The reader is the equivalent of a constant source generating all 0’s, sometimes called a null source. The writer always accepts items, without overflow, but ignores them, a style of which is sometimes called a null sink. A block cannot in general function with stalled input buffers, though special exceptions can be made. A block can function correctly with a stalled writer – any generated data is just ignored on that connection. The number of stalled buffers is automatically decremented from the total required to pass threshold before processing can occur.

5.4.9 Size

The waveform developer can set the desired buffer size, and also has control over a multiplier used to augment this size. By default, the buffer size is set to hold 3 times the maximum threshold value, in bytes. We have found that this “3 times” provides enough buffer margin to accommodate multiple processing outputs. This multiplier allows the average sample rate to be maintained on the connection; too small a multiplier results in greater chances of overflows or underruns, while too large a multiplier is wasteful of resources.

5.5 Supervisor

The Shark supervisor has control over the system runtime behavior. It can change the flavors of an individual block or group of blocks, e.g., to use the host computer’s GPU instead of it’s CPU. It can set or remove runner affinity for any block, as well as change the thresholds for any block in order to modify
the overall throughput / latency processing point. The supervisor provides a means to access the “smarts” behind Surfer, to provide for multiple forms of dynamic runtime reconfiguration; the forms provided in Surfer are described later in this section. The supervisor, in general, uses statistics of various sorts to make decisions. Examples of statistics include the overall CPU load and the throughput of a specific block.

During the Surfer boot sequence, the waveform developer’s application must provide a method that is used to create and initialize the supervisor. If a supervisor is not created, then Surfer will assign a simple one that just provides opportunistic mapping from block to runner – taking each block in turn and assigning it to the first runner of the correct type. The waveform developer can optionally set a supervisor parameter to assign block affinity to the first runner used, or can configure the block with runner affinity after the block is created.

By default, each block and block runner collect runtime statistics on execution timing. These processing statistics for each block are the number of bytes generated or consumed, generally depend on whether this block is a source or not, respectively, and timestamps for

- when the block was queued for processing – no matter if using affinity or the supervisor’s mapper;
- when the block was dequeued by the block runner;
- when processing started;
- when processing finished; and
- when the block runner released the block.

The above data samples can provide a number of interesting statistics, including throughput, overhead time, and latency. They can also be used in conjunction
with other block’s similar statistics to determine and possibly tweak runtime system concurrency – e.g., disable a block runner or create a new one.

These samples are stored in the block if a callback to the application is provided to handle them; otherwise they are collected but not stored. This callback can specify the number of samples to provide per callback call, as well as the decimation of these statistics to keep processing requirements in check. The waveform developer can request raw statistics and then perform computations on the during the callback, or pre-computed statistics such as the average instantaneous throughput, overhead time, and input-to-output latency. Statistics collection can be reset, e.g., when a threshold is changed on a specific block; both the threshold change and statistics reset are performed by the supervisor.

These processing statistics for each block runner are a timestamp each for when the runner had nothing in its queue and thus entered its waiting state, and another for when it was woken from this waiting state due to a signal from another thread that a block was added to its processing queue. The combination of timestamps for all blocks and runners can be used to create graphs showing the pipelining of data, in both time and concurrency, as data is processed. Some graphs of this style are shown in Chapter 9.

5.5.1 Statistics

*Surfer* provides monitors for different types of statistics, including:

- Raw Processing Timing: per signal processing block, when queued, dequeued, processing started, processing ended, and released, and the number of items and bytes processed;
- Throughput: per signal processing block, in items / second and bytes / second;
- Latency: per signal processing block, in seconds;
• Overhead: amount of time Surfer spends performing overhead tasks, for each thread or total across all threads except the supervisor;

• CPU load: for the whole OS;

• Surfer process CPU time; and

• any Surfer thread CPU time.

A base statistics monitor class is provided that can be used to create other statistics for use within the supervisor - e.g., energy use of the host computer, via some hardware-based monitor. Further, the supervisor can determine its control methodology in any other way – Shark and the rest of Surfer have built-in error checking such that most improper configurations will be detected and either not used or fixed to be usable. This base class allows the waveform developer to select both the frequency of callbacks and the amount of data decimation or averaging. The frequency defaults to 20 times a second; decimation / averaging has no default, and must be specified by the waveform developer when initializing the callback.

5.5.2 Reconfiguration

In Section 3.2.6, reconfiguration was mentioned as a means for SDR to support cognitive radio and provide dynamic spectrum access. Through the use statistics and asynchronous callbacks from specific blocks, the waveform developer can create a supervisor that provides multiple types of reconfiguration – from block parameters to flavor selection, adding and removing blocks to using a switch for multiple potential paths. Some example supervisors are described in Chapter 9.
5.6 Summary

In this chapter, information about the Surfer runtime layer was provided. This layer, called Shark, provides the “dumb” means for connecting a graph together, as well as the “smarts” to create a supervisor to handle system runtime behavior. At this layer, reconfiguration is provided through the use of a buffer master, which can handle multiple readers and a single writer and buffer. The buffer master provides the means for attaching to and detaching from readers, as well as swapping the buffer or writer – all during runtime. Shark handles byte-oriented data, and does not deal in types in any direct sense. Types are left to the next higher layer, Spf, details of which are provided in the next chapter.
CHAPTER 6
SPF : THE SURFER SIGNAL PROCESSING FRAMEWORK

The Surfer runtime, Shark, as discussed in the previous chapter is primarily about byte-based data-flow through buffers, thresholds for determining when a block is ready for processing, and a supervisor to handle runtime determination of the block / processing device mapping and threshold values. This chapter presents the signal processing “framework” used by Surfer, called Spf, whose purpose is split between adding item type to the buffer-oriented runtime classes and defining the APIs needed for developing actual signal processing blocks.

Spf currently weighs in at around 2k lines of code, not including comments, implementing around 20 classes. Figure 6.1 shows the class relationships for those defined in the Spf namespace and discussed in this chapter. A general-purpose diagram of a block is shown in Figure 6.2, this chapter describes the internal parts of the block, and our changes to this general model to allow for dynamic runtime behavior modification. The item-specific changes are discussed in Section 6.1, and the new APIs in subsequent sections. This chapter is finished with a summary of the Spf interface in Section 6.6.

6.1 Typed Classes

Unlike Shark, which almost exclusively deals in bytes and not items, Spf provides buffer-oriented classes with an item-based API. The four buffer-oriented
Figure 6.1. Class relationships for the Spf namespace implementation.
Figure 6.2. General diagram of a SDR signal processing block.

classes are for readers, packets, shared memory double-mapped buffers, and writers.

6.1.1 Buffer Readers and Writers

As already discussed in Section 5.4, each buffer is owned by a master, which can dynamically add or remove any reader, swap the single writer with another writer, or swap the buffer with another buffer. The readers and writers inherit from their Shark counterparts, and in Spf are template classes referring to a specific item type. The buffer is not required to be used for this type only; for example, the internal method for building an inline packet takes a buffer of type `char*`, no matter what the external type is. These classes set the buffer `item_type` and related information, and provide access to the internal buffers via `operator[]`, `at`, and the current enqueue or dequeue item pointer.
6.1.2 Shared Memory Double-Mapped Buffers

The Shark base buffer class provides a generic, byte-oriented interface for handling buffers. In Spf, we provide a specific buffer type that uses kernel shared memory as described in Section 2.3.4, allocation type 3 – with the end goal being to provide a virtual circular buffer by mapping the same kernel memory allocation into contiguous local memory. GNU Radio uses this virtual circular buffer concept as well, though the actual implementation is different. We have found the following method to be extremely reliable, so long as the kernel shared memory allocation is large enough to contain all of the buffers needed for the blocks in use.

Figure 6.3 shows a visualization of the process for creating this buffer type, for an actual buffer length of size $S$. When mapping kernel memory to user-space, a map-to address can be provided. In the figure side (a), this address, $A$, is obtained by creating a mapping of twice the actual needed size and mapping it into user-space, then deleting both the map and initial kernel memory allocation. The resulting address, $A$, now points to free user-space memory of the required size ($2S$). In (b), a kernel memory space is allocated of the actual buffer length ($S$), and then mapped twice, into user-space addresses $A$ and $A+S$, such that $A[0+n] == A[S+n]$ is true for all $n$ in $[0, S-1]$. In this manner, a virtual circular buffer of size $S$ can always be created for all $n$ in $[0, S-1]$; the buffer wrap is always set to no more than $S-1$ to guarantee this condition.

The class performing this task, called a *shm_dm_buffer*, inherits from the *Surfer* base buffer class, and provides the necessary methods for buffer construction, mapping, unmapping, resizing, and destruction. This buffer extensively uses the Salt class providing keyed kernel shared memory manipulation, which handles the OS-interface side of the required tasks. As kernel shared memory cannot be
Figure 6.3. Visualization of a kernel shared memory double-mapped buffer allocation: (a) An initial allocation of twice the necessary size is made, to retrieve a valid address, $A$, for containing the double mappings. (b) The mapping from (a) is removed, and a new one of the actual desired length created and then mapped twice in the same contiguous memory space.

resized, the resize feature is the equivalent of allocating a new double-mapped buffer and copying from the current buffer to the new one. That said, because the allocation size of each kernel memory buffer must generally be a multiple of some set size, e.g., the OS page size, the actual mapped buffer size will be no smaller than the requested user-space buffer size. Hence resizing to a smaller size is handled internally without need for reallocating; reallocating is performed only when necessary.

6.1.3 Packets

An Spf packet is a template class that inherits from the byte-oriented Salt packet. The item type for a packet refers to its payload only, and the Spf packet class provides methods for accessing the payload as a typed buffer via
operator[] and at. Interfaces for both build and retrieve methods use templates to allow for any external typed buffer as well as any type of reader or writer; internally, as already mentioned, the buffer is typed as just char*. The payload length in items is made available, as an alternative to the payload length in bytes.

6.2 Flavor Base

For runtime dynamic reconfiguration, some sort of switch as found in Figure 6.4 is required to select the waveform, block, or computation being performed. Instead of switching between waveforms or blocks – tasks Surfer was already capable of performing – we added another style of reconfiguration by moving the switch inside the block itself and separating the signal processing algorithm implementation, what we call a flavor, into its own class. In place of the signal processing, we added a flavor table, which will be described in Section 6.3 and depicted in Figure 6.7 with six possible flavors.

The idea behind flavors is that each provides identical functionality, such that given the same state and data input, each will produce the same output data to within machine precision. All flavors interface with Surfer on the local CPU on which Surfer is executing, and then also with the remote processing device to do the actual processing. In this manner, Surfer allows for heterogeneous processing via flavors targeting processing devices other than the local CPU. Current flavors target the local CPU and GPU, but new ones could be written to target an attached DSP, FPGA, or other type of processing device, given the appropriate API / framework.

Each flavor is allowed to store variables specific to its use locally, but also has use of the overall block state in order to allow dynamic reconfiguration. In
In order to function correctly, many flavors must use both state and local variables. As an example of local variables that do not impact the block state, different FFT flavors use “twiddle factors” or “plans” to speed up their computations. These variables and their storage are flavor-specific, while the FFT length and any other user-specified block parameters – e.g., for a multiple-dimensional FFT which dimension is first – are part of the overall state.

From the flavors in the block’s table, any one can be selected to do processing – even switching between them for each time the block performs data processing – because state is stored separately from processing. Flavors make efficient use of resources because they contain only the code that has to be switched; any method or variable common to all flavors is found in the block’s class.

Surfer provides at least one flavor for each block – the “generic” implementation for the host CPU. Many blocks contain flavors for “apple+accelerate” and “naCL+generic”; FFT flavors are provided for FFTW3 and different naCL implementations, while audio flavors are currently available for “apple+audio” and “portaudio”. A special subgraph flavor type is also provided, allowing the exten-
sion of the waveform graph via dynamic connection of a subgraph. The base class for some specialized flavors is described in sections immediately following this one.

A “flavor hierarchy” is shown in Figure 6.5 with the various single and multiple inheritance flavors for the \texttt{Sum} block. We use virtual inheritance to minimize the number of methods that must be specified in the end-flavor class implementation. Virtual inheritance is used when multiple class inheritance paths join to form some new class, and, effectively, allows any method prototyped in the top-level base class and defined in some inheriting class to be used in the final inheriting class as if it were defined there. In this way, for example, the naCL flavor inherits from both the naCL base flavor code and the \texttt{Sum} flavor base code, and both of those inherit from the top level flavor base class via virtual inheritance. The naCL base flavor code overloads some top-level methods in order to interface with the naCL framework, as well as adds some new methods that are naCL specific. The use of virtual inheritance means that the naCL base flavor methods will be selected for use instead of those in the top level flavor base, and thus that they do not have to be redefined in or executed from any other flavor class. Note that it is often the case that multiple inheritance paths converge to the resulting flavor, and hence virtual inheritance is an important C++ keyword used to reduce programming complexity by clarifying ambiguity over which method is actually inherited.

The waveform developer can add flavors, either replacing or adding to those in any block’s table. Each block using flavors contains a default flavor – generally the first one added to the table – as well as an optional user-supplied priority list ordering the available flavors. Each flavor can, but does not have to, be assigned to a specific block runner of its type – e.g., naCL flavors can be executed only within an naCL block runner, since they require different handling than a flavor.
Figure 6.5. Example hierarchy of flavors for the Sum block.
executing on the local CPU. In this way, Surfer allows for either runtime or a-priori block mapping via flavor affinity with a given block runner.

Both Surfer and flavor compilation and execution are highly dependent on system-provided libraries, headers, and frameworks implementing and providing access to the classes, functions, and variables specific to the flavor's programming. As such, usability is determined at three points: (1) at configure time: whether or not the required system-provided libraries, headers, and frameworks are available; (2) at compile time: whether the items found in (1) work with this implementation; and (3) at run time: whether the flavor initializes correctly. An example of run time checking is flavors that provide a specific variant on the overall flavor, e.g., an FFT that works solely for power-of-2 lengths. In this case, even when all other checks pass, if the user specifies a non-power-of-2 length FFT then this flavor will fail initialization and hence will not be available in this block's flavor table. As flavor usability is determined on a block-by-block basis, it is possible for different instantiations of the same signal processing block type to have different flavors available to use during runtime.

The flavor abstraction for signal processing comes with very little overhead in terms of additional programming complexity or latency. The actual processing method / function call is handled through the lookup table, and hence incurs an additional pointer dereference, but otherwise the additional complexity is borne by the programmer / developer of the block and / or flavor. The point where potential overhead could occur is if a new flavor is added during runtime to the flavor table. The flavor must be initialized, and this task can be performed in the same thread as that handling processing, or pushed off to a separate thread and executed asynchronously to data processing threads. This event occurs only once
while that flavor is in active use, and hence the overhead latency associated with using this flavor – assuming it is used for a significant number of times – will be much less than the actual time spent processing. Hence there can be an additional up-front cost to using flavors, but this cost will be negligible in long-term use.

When a flavor’s owning signal processing block is started or stopped, that command is propagated through to all flavors; these methods are executed exactly once during the flavor’s lifecycle, and thus are meant for initializing and clean up any local storage specific to this flavor’s needs. When a flavor is swapped in for processing, it is activated and will remain so until it is swapped out, or deactivated. The activate method is called with any block runner affinity, the block state, and the readers and writers associated with the owning block. Activation and deactivation is generally handled by the supervisor thread. In this manner, any temporary objects can be created and initialized with minimal interruption to signal processing.

Once a flavor is active, it can be stalled and resumed any number of times; in general, the flavor will be stalled only when its owning block is stalled. A stalled block will not allow itself to be queued for processing, thus stalling needs to be handled carefully.

6.2.1 Apple Accelerate Base

Apple has provided its “vector DSP” library, called Accelerate [7] for many generations of Mac OS X. This framework is an optional external dependency to Surfer that is checked for during configuration. The Surfer user’s application can enable Accelerate by including the macro

```
define SURFER_USE_APPLE_ACCELERATE
```
before any headers are included. If this macro is not used, and Accelerate was
available for use, then the application will print a “NOTE” during its boot se-
quence to this effect, but should print nothing further about Accelerate.

Accelerate provides a generic API for accessing the host CPU’s specialized
vector instructions – thus far Altivec on PowerPC and SSE1/2/3 on Intel pro-
cessors. Functions are provided in Accelerate for performing arbitrary length dot
products, changing the ordering of items in a buffer, complex and real FFTs, and
basic arithmetic such as addition and multiplication of two vectors or one vector
and a scalar – among numerous other functions. Accelerate provides functions for
a limited number of types: single and double precision floating point, and some-
times 32-bit integer. Because this type limitation, the Accelerate flavor base class
uses a Salt item_type to verify that any item type being handled by Accelerate
is of an acceptable type, and if not then that flavor will fail initialization.

As an example of local-flavor state, blocks using the Accelerate flavor for per-
forming FFT, multiplexing, ftshift, and the prepending of a cyclic prefix allocate
temporary buffers when activated, and delete them when deactivated. For the
FFT, these buffers serve to store intermediate results or to convert between “in-
terneled complex” and “split complex” data formatting. For the other flavors,
the buffer is a precomputed table containing the index mapping from input to
output items. Some flavors, such as the FIR filter, allocate temporary buffers
only when the item type is complex.

6.2.2 FFTW3 Base

FFTW version 3 is an optional external dependency to Surfer that is checked
for during configuration; each of its variants – single, double, and long double –
are checked for. The *Surfer* user’s application can enable FFTW3 by including the macro

```c
#define SURFER_USE_FFTW3
```

before any headers are included. If this macro is not used, and FFTW3 was available for use, then the application will print a “NOTE” during its boot sequence to this effect, but should print nothing further about FFTW3.

The FFTW3 flavor base actually inherits from an intermediate class providing support for those flavors using the local CPU, e.g., the generic version, FFTW3, and Accelerate – which in turn inherits from a common flavor class. Each intermediate class performs error checking on initialization, as well as provides some specific common functionality. For example, the local CPU common flavor code provides the entry point for processing, determining the FFT length (from state) and number of FFTs to perform, and then repeatedly calls a local `do_fft` method. This method splits execution depending on the FFT type: real to real, real to complex, complex to real, or complex to complex. Dummy methods are provided for each of these methods, that throw an error when called. The FFTW3 base flavor overloads these methods, and iterally selects the correct implementation type – single, double, or long double precision – and then performs the appropriate call to the FFTW3 function executing the FFT. Although this class hierarchy sounds tedious – and, truthfully, it was to program – it provides a robust model for verifying that the flavor can be used, allocating temporary buffers, and performing the actual FFT. Similar intermediate classes are provided for the naCL and Accelerate FFT flavors.
6.2.3 naCL Base

naCL is an abstraction layer over OpenCL, and is discussed in the next chapter; this specific class is found in Section 7.3.2.

6.2.4 Subgraph Base

As mentioned in Section 2.3.1, the definition of what is “small” in terms of creating blocks is an area of active work. This is the problem we call computational coarseness. With the raw processing power found in consumer-oriented computers’ CPUs and GPUs, signal processing algorithms must be developed that targets each processor device’s attributes. Depending on the target processing device, a flavor might be a self-contained algorithm implementation or a subgraph of blocks. The former flavor types have been discussed above.

As an example of the need for different levels of computational coarseness, consider a FIR decimate-by-L filter. When implemented for a single thread, the FIR filter is applied to every $L^{th}$ input sample. On a many-core GPU, we have found that it is generally more efficient to implement the single threaded-version per core, and then have each core process a subset of all inputs. In this way, the GPU implementation avoids the synchronous adder and the implementation is massively parallelized. If using a CPU and the number of filter taps is much greater than the number of available processor cores, then it is generally more efficient to implement this flavor in its polyphase representation – splitting the filter taps into smaller chunks, and then spreading the filtering across all available cores. This multi-core version is actually a subgraph, made up of a demultiplexer, FIR filters, and a synchronous adder, as shown in Figure 6.6.

The core of subgraph flavors are the buffers readers, writers, and masters.
already described; they allow the dynamic addition and removal of blocks, during runtime. Shark exposes each block’s buffer readers and writers to a flavor when it is activated. For subgraph flavors, the adjacent readers and writers are used to connect the subgraph into the waveform graph with no intermediate buffering. Subgraphs add no extra latency to the overall graph when compared with a graph using the same blocks but without those designated as a subgraph. Shark treats the subgraph as just another part of the normal data-flow graph. Each block in the subgraph is just a normal block, with its own selectable flavors; hence, a waveform can contain a subgraph within another subgraph. All functionality is still controlled by the system supervisor in a global manner.

Figure 6.6 shows a potential subgraph flavor of the “downsample by L” block, utilizing “smaller” blocks to create a polyphase downsampling filter. This figure also provides the readers and writers for the interface blocks, showing how the flavor’s input reader just takes from the same writer as block’s reader. Because only one writer is allowed per master, when the subgraph flavor is activated the block’s writer’s master is retrieved and the master swaps out its current writer with the flavor’s writer. The flavor’s reader is attached to the block’s reader’s master, and the block’s reader is stalled – it ignores data coming into it. When the flavor is deactivated, the block’s writer is swapped back, the flavor’s reader detached, the block’s reader resumed, reverting functionality to what it was without the subgraph.

6.3 Flavor Table

A flavor table is a type of lookup table, with the key being a string and the item being the pointer to a flavor. Given the generally small number of flavors
for any given block, and our belief that flavors will be accessed only infrequently compared with the number of times the owning block is queued for processing, the use of a table provides the necessary functionality without overdoing it. For example, we could have used an ORB or some database using a structured query language (SQL) variant, but these would add a background dependency as well as provide much more functionality than required – and hence be wasteful of resources.

Each signal processing block contains its own unique flavor table object, an example of which is shown in Figure 6.7 and hence different instantiations of this class can have different flavors selected. Each flavor table is initialized by the owning block during its construction, and provided with a reference to the block such that any flavor can use the table to get access to the owning block. The other variables stored in each flavor table include the vector of string names of all added flavors, a pointer to the current in-use flavor, a pointer to the next flavor to use, and a mutex to protect against multiple threads manipulating the table simultaneously.

If a given flavor initializes correctly, then it will be registered in its block’s
flavor table. The user can register new flavors by using different names than those provided in Surfer, or overwrite any provided flavor by reregistering one using the identical name. The first registered block is also selected for use, since any instantiated block must have an active flavor. Activating a flavor is, broadly, a two-step process, first selecting the new flavor and then swapping it in and activating it. If the flavor being swapped out is a subgraph type, then it is first flushed before being swapped out. If the flush process takes too long, then data can be lost; if no flush is issued, then the flavor swap will not be seamless. Hence, the flavor table first tries to estimate the amount of time the flush will take, and then decides on the best path depending on that estimate.

When a subgraph flavor is instantiated, its internal blocks are created and connected to form the subgraph; these blocks are deleted only when their owning flavor is deleted. When a subgraph flavor is swapped in, any readers are connected into the currently-active graph to the prior block’s output buffer masters; likewise,
any writers are connected to the next block’s input buffer masters. The subgraph literally describes itself – it is a subgraph inserted into the overall waveform graph when activated, and processes data in the same manner as any graph in Surfer.

When the subgraph flavor is deactivated, any readers and writers are detached from their masters. All connections and detachments occur during runtime, without stopping data-flow processing; they generally occur in the supervisor’s thread, and hence this particular overhead happens concurrently while block runners are handling other blocks’ processing.

When the owning block is started or stopped – each command happens once during the block’s lifecycle – the command is propagated through to all flavors, allowing them to initialize and clean up after themselves, e.g., via hooks into the OS. When the owning block is stalled or resumed – events that can happen any number of times during normal processing – the command is passed through the flavor table to the active flavor only to allow that flavor to internally handle stalling or resuming data processing. The method `operator->` is used to provide quick access to the currently selected flavor. Methods are also provided to return the name of the current flavor, the list of flavor names, and whether a given named flavor is known to this table.

6.4 State Base

Given multiple flavors that provide execution on different processors and / or using different compilers, most blocks’ state must be made transportable between processing device memories as well as cross-device interpretable. A standard C++ class instantiation / C structure can be copied between threads of the same application, and even shared between different processes executing on the same
processor / OS. But, in general, neither can safely be used by different processing devices / OSs, whether copied or shared in some common memory, due to potential differences in alignment requirements, type sizes, and endianness. Some blocks provide a very specific functionality – e.g., one implementing an Ettus Research UHD receiver – and thus the state can be integrated with the actual signal processing block class because there cannot be more than one flavor.

Most signal processing blocks perform an algorithm that can be implemented using different techniques; for these a new portable state construct was put in place. For this case, the state base is simply a DynamicStructure as discussed in Section 4.2.3. Each block’s state should contain only the variables required for describing the current state of the algorithm, no matter the implementation; it should not contain flavor-specific variables such as temporary buffers required by the flavor to internally convert between data formats. For example, a multiplexer block – which takes items spread across $N$ input streams and sequentially and synchronously merges them in a single output stream – will require the number of streams, the merging direction, and the next input stream from which to take an item. When using the naCL flavor, all of its allocated objects are stored within the instantiated flavor only because they are flavor-specific. No other flavor needs to know about naCL’s allocations.

Each block will contain its own state which inherits from the base state class. The base block class provides a method returning this state base, which can then be upcast to the actual state type if a local method wants access to it, via the macro BLOCKS_UPCAST_STATE. The inheriting block must provide a reference to the block’s actual state at instantiation, to guarantee that the state is available during the whole lifetime of the block.
6.5 Block Base

The block defined in *Surfer* is used as an interface to Shark; it does not define the APIs required for performing actual signal processing. These APIs are provided by the Spf base block class, from which all signal processing blocks must inherit. Each Spf block must be attached to a *Surfer* block such that they can communicate with each other, but they are separate classes. *Surfer* blocks primarily use Spf blocks to determine threshold values, check topology, verify scheduling values, and perform actual processing; these are the core methods of the Spf API.

Each block contains a pointer to the *Surfer* block to which it is attached, a pointer to its state, a flavor table, the string name of this block, and a mutex for internal use to keep multiple threads from manipulating this block simultaneously. Methods are provided for controlling block execution: start, stall, resuming, reset, flush, and stop. Start and stop are called exactly once, when the paired *Surfer* block is started or stopped. Other methods return the execution state, the block name provided at instantiation, if this block is a source or sink, and whether it can do self-processing of data. Each Spf block has an interface to its flavor table, allowing the creation of flavors as well as other flavor-oriented tasks. Methods are provided for selecting, swapping, finding, retrieving the current flavor’s name, and the list of flavor names available.

Processing is handled through a protected method, \texttt{process} that is called solely by the paired friend *Surfer* block; this method, in turn calls the flavor’s \texttt{process} method, and then block’s \texttt{post\_process} method. Post-processing is provided with the after-processing state, buffers, and number of items actually consumed and generated; this method might update the next stream for demultiplexing, or check a file source for end-of-file and issue a callback.
The Spf block base class provides the API for runtime processing, and not much more. In this way, the separation of block-oriented classes between runtime, processing, and flavors works well for waveform developers creating new block types or flavors for an already existing block. One goal in this separation is that a single block class can be created to cover all flavors, no matter the target processing device or framework. That said, our particular implementation of specific blocks and flavors is meant more to prove the usefulness of this approach much more than to be runtime optimized. Hence, there is future work that will be useful in moving Surfer from a fully functional yet proof of concept system to a fully runtime optimized one.

6.6 Summary

This chapter described the Spf classes’ APIs. These APIs provide a robust means for creating signal processing blocks and flavors. Combining the ability to schedule the amount of data to process from the prior chapter, with the availabilities of changing the mapping from blocks to processing devices and selecting subgraph flavors that insert and remove blocks from the waveform graph during runtime, multiple forms of runtime system control and reconfiguration are available to be controlled by a supervisor. The primary missing link to meeting this thesis’ goals is true heterogeneity. The next chapter provides details this link: our OpenCL abstraction layer, called naCL, and its connection back to Shark and Spf.
In order to use any computing device beyond the host computer’s CPU for signal processing, an API is required that provides functionality for program manipulation – creation, execution, transferring data to and from, and so forth. As Surfer is designed to provide signal processing capabilities on any processing device, via flavors, it made sense to go with an API providing similar qualities. The Khronos Group’s Open Computing Language (OpenCL) [72] is an open, cross-platform standard allowing for parallel programming on heterogeneous computers [45]. It is available for most modern OSs, including Mac OS X, Windows, and Linux, and has even been ported to some embedded systems. OpenCL was chosen instead of AMD ATI Stream [3] or NVIDIA CUDA [108] because it works with most major-manufacturers’ GPUs – including both NVIDIA and AMD – as well as other processing devices including FPGAs [5] and DSPs such as the Cell Broadband Engine [18] and TI C64x+ Series [80]. Using OpenCL, a programmer can create a single application that spans multiple processors of different types, taking full advantage of each processor’s attributes.

OpenCL assumes a fairly generic hardware architecture, as shown in Figure 7.1. As this architecture matches the physical topology of many computers reasonably well, OpenCL should introduce only a small extra overhead compared with a device-specific, non-generic framework. The figure shows a single host computer
Figure 7.1. Generic hardware architecture assumed by OpenCL.
connected via some sort of bus to multiple compute devices; in OpenCL terms, a *compute device* is a single, possibly multi-core, CPU, GPU, DSP, FPGA, SOC, or any other processor that can be mapped into OpenCL’s framework. Each compute device is modeled as having four types of memory – global (dynamic), global (constant), local, and private – and one or more *compute units*, each of which has one or more *processing elements*. Processing occurs on each processing element; hence, data access is fastest for private memory and slowest when using global memory. Mapping OpenCL terminology to a multi-core CPU, the CPU is the compute device, and each core is a compute unit containing one processing element. For an NVIDIA GeForce 8600M GT (as found on some Apple MacBook Pro models), OpenCL reports that there is 1 compute device containing 4 computing units, each with 512 processing elements – the equivalent, roughly, of a 2048-core processor at CPU speeds from 5 or so years prior. The number of computing units and processing elements is important when determining the allocation of work items, which is further discussed in Section 7.2.3.

The OpenCL 1.1 specification provides an API for both C and C++; as *Surfer* is written in C++, the latter API would have been used except that it does not provide a full object-oriented interface. The OpenCL standard C++ header is mostly a thin wrapper around the C functions, and provides a similar API just upgraded for C++; error handling / exceptions and information querying are quite object-oriented, but the primary functions are not. Hence, we created an API that is object-oriented; we call this project *naCL*, a mixing of “another OpenCL” taking into account the general theme of this project – *Surfer*, Salt, and Saline. *naCL* is designed to be independent of Salt and *Surfer* through the use of typedefs for internal types; it does rely on the C++ standard library directly,
without going through an abstraction layer. naCL currently weighs in at around 7k lines of code, not including comments, implementing around 40 classes.

naCL provides a more natural C++ / object-oriented interpretation of how OpenCL works – based on the OpenCL 1.1 C specification – such that dependencies are automatically met when issuing commands. For example, an OpenCL Context is valid only within the Platform in which it was created; hence, the Platform must be used to create it, will keep track of it, and will handle deleting it. This ownership methodology is used throughout naCL as much as possible. Information querying is handled internally when an object is created, with the object providing methods to retrieve pertinent information without having to do an explicit query on the underlying object.

The following section provides an overview of the naCL class’s interface overlay with OpenCL. Section 7.2 describes additions to naCL specifically for performing data-flow processing as done in Surfer or other SDR frameworks. The interface between naCL and Surfer is provided in Section 7.3, followed by some future work and then a summary of this chapter.

7.1 Classes

The Khronos Group provides an OpenCL C++ bindings header file, named cl.hpp, implementing the C++ bindings specification [71]. Their class model is shown in Figure 7.2[72, Figure 2.1 on p. 20], and their implementation is split somewhere between a thin layer over the C implementation and true object-oriented programming; it does not significantly take advantage of explicit object dependencies built in to the OpenCL specification. This figure is in Universal Modeling Language (UML) format, which is beyond the scope of this thesis; there
Figure 7.2. Universal Modeling Language diagram of OpenCL 1.1 classes.
are plenty of good tutorials and books available on the subject \[124\][143]. As an example (one of many), a CommandQueue is created in C via the function

```c
extern cl_command_queue clCreateCommandQueue
(cl_context context,
cl_device_id device,
cl_command_queue_properties properties,
cl_int* errcode_ret)
```

and in C++ the CommandQueue constructor is declared as

```cpp
CommandQueue
(const Context& context,
const Device& device,
cl_command_queue_properties properties = 0,
cl_int* err = NULL);
```

Note the similarity between the function and method. The creation of a CommandQueue requires a valid Context and Device; further, the Context must have been instantiated to contain (at least) the provided Device, hence some error checking must be put in place to verify this condition. As is the case for most of the OpenCL C++ interfaces, this class definition is a thin wrapper around the C implementation.

One object-oriented way of implementing this constructor is to have the Context provide a method for creating a CommandQueue taking as one argument the index of a Device already in the Context. A pointer to the creating Context can be stored in the new CommandQueue, making deleting it a simple matter of
telling the Context to do so, while making the actual constructor and destruc-
tor protected. With this object-oriented class hierarchy in mind, not just for the
CommandQueue class, but overall, we created “another OpenCL” C++ interface,
residing in the naCL namespace.

The primary class hierarchy of the naCL OpenCL abstraction framework is
shown in Figure 7.3. Within each class, explicitly declared and implicitly derived
dependencies are shown. Thus, for example, the CommandQueue will be created
by a Context, using one of the devices within that Context. Each of these classes,
and the rest of naCL, are discussed in the next sections, roughly following the
figure’s hierarchy.

7.1.1 Platform

An OpenCL Platform is the set of host computer’s properties, and computing
devices and their properties, that can be utilized by OpenCL. A single host com-
puter can have multiple platforms, depending on the OpenCL implementation and
available hardware. A naCL Platform consists of a set of one or more devices, and
one or more contexts each using one or more of those devices. During application
initialization, before the C function “main” is called, the naCL top-level names-
pace is populated with all of the known OpenCL platforms, all devices within each
platform, and the various meta-data for each platform and device (e.g., version,
Vendor, type, extensions). When the application finishes, the memory for these
platforms and devices is freed. The naCL Platform class’ constructor and destruc-
tor are both protected, such that a Platform can be created or deleted via special
“init” and “atexit” functions only; these function were described in Section 4.2.6.
Figure 7.3. Class relationships for the naCL framework implementation.
7.1.2 Device

An OpenCL Device is any single computational resource on the host computer with an OpenCL interface. A multi-core CPU is counted as one Device, with multiple compute units. OpenCL Devices span the range from CPUs to GPUs, FPGAs to DSPs. In naCL, all Devices are allocated during “init”, and deleted during “atexit”. The naCL Device class encapsulates the actual OpenCL Device object as well as all of the meta-data associated with a Device (e.g., vendor, max compute units, global memory size). The naCL Device class’ constructor and destructor are both protected, such that a Device can be created or deleted by a naCL Platform only.

7.1.3 Context

An OpenCL Context is a set of one or more Devices, the memory accessible to those Device(s), Program(s) and their Kernel(s), and CommandQueue(s) used to schedule execution of the Kernel(s) or operations on memory objects. A Context is the environment within which an OpenCL Program’s synchronization and memory management is defined. An nACL Context encapsulates the actual OpenCL object, along with of one or more naCL Devices selected from a single Platform, one or more CommandQueues assigned to each Device, and one or more Programs.

A Context is created via one of the naCL Platforms generated during naCL’s “init”, using a specific Device reference or index, a vector of Device references or indices, or a processor type from whence one or more Devices of that type will be selected. Each Device reference or index is verified to be within the given Platform. On success, a non-NULL Context pointer is returned; on failure, a
NULL Context pointer is returned. The naCL Context class’ constructor and
destructor are both protected, such that a context can be created or deleted by a
naCL Platform only. The Platform keeps track of all created Contexts, and will
delete them during “atexit” if they have not already been destroyed.

7.1.4 Program

An OpenCL Program is a set of one or more ProgramSources and Kernels re-
sulting from building those Source(s) for one or more specific Devices. The naCL
Program class encapsulates the actual OpenCL object, as well as all Program-
Sources and their Kernels. When building each Kernel, naCL verifies that the
provide Device belongs to the naCL Context in which this Program exists. The
naCL Program class’ constructor and destructor are both protected, such that a
program can be created or deleted by a naCL Context only. The Program keeps
track of all created ProgramSources and Kernels, and will delete them during
“atexit” if they have not already been destroyed.

7.1.5 Kernel

An OpenCL Kernel is a Program compiled for a specific Device, with the
capability of being queued for execution using a CommandQueue for the given
Device. The naCL Kernel class encapsulates the actual OpenCL Kernel object, as
well as pointers to the arguments used to invoke the Kernel when it is executed.
The naCL Kernel class’ constructor and destructor are both protected, such that
a Kernel can be created or deleted by a naCL Program only.
7.1.6 KernelArgument

In anticipation of using naCL for data-flow style processing, where repetition will be required, we created a class to hold Kernel arguments. OpenCL does not provide a distinct KernelArgument class. naCL contains a distinct KernelArgument class, as it provides a generic object-oriented approach to handling different Kernel argument types. There is a base class from which all KernelArguments inherit from, with generic functionality provided when not using a MemoryObject. MemoryObjects are an OpenCL specific type discussed below in Section 7.1.9 and, per the OpenCL 1.1 specification, require special handling when used as a KernelArgument. A KernelArgument is simply an enclosure for an actual Kernel argument – typically a constant value of some type, or a pointer to a buffer of a given type – allowing for a single method in the Kernel class to set each arguments, or all arguments, via a reference to the base KernelArgument class. Because a KernelArguments can potentially be used by multiple Kernels, the naCL KernelArguments class’ constructor and destructor are both public. KernelArguments must be maintained by the host application, along with the variables and memory used to create them.

7.1.7 ProgramSource

An OpenCL ProgramSource is code to be compiled, or one or more precompiled binaries. If the latter, each binary is defined for execution on a specific Device, with that information embedded within the binary. Because a ProgramSource can potentially be used by multiple Programs, the naCL ProgramSource class’ constructor and destructor are both public. ProgramSources must be maintained by the host application, along with the variables and memory used to create them.
them.

7.1.8 CommandQueue

An OpenCL CommandQueue is a queue that holds commands to be executed on a single specific Device within a single specific Context. Depending on the OpenCL implementation and user preferences, commands entered into a CommandQueue may be executed out of order, and timing information may be collected associated with each command – a time stamp each for when the command was enqueued, processed, placed for execution, and finished executing.

The naCL CommandQueue class encapsulates the actual OpenCL CommandQueue object, pointers to the Device and Context in which it is defined, as well as all Events and MemoryObjects related to it. The latter are included because they are defined within the OpenCL standard solely within the scope of their owning CommandQueue. The naCL CommandQueue class’ constructor and destructor are both protected, such that a CommandQueue can be created or deleted by an naCL Context only. The CommandQueue keeps track of all created Events and MemoryObjects, and will delete them during “atexit” if they have not already been destroyed.

Each CommandQueue can be flushed – blocking or non-blocking – which guarantees that all queued commands are issued. Flushing is not an enqueued command, but rather affects the actual CommandQueue. When non-blocking the flush does not guarantee that all of the commands will have been issued when the method returns, only that they will have been issued to their respective Devices. The blocking version, in OpenCL terms, “finishes” the queue; this type of queue flushing is a synchronization point, while the non-blocking version is not.
A variant on non-blocking flushing is to enqueue a barrier, which is also a synchronization point. This command is added to the CommandQueue, and ensures that all prior queued commands are issued before it – no matter whether the prior commands are being processed in-order or out-of-order. A barrier has no Event associated with it, and thus cannot be used as a dependency for future commands.

A marker event is a synchronization point, similar to a barrier command in that all prior enqueued commands must be finished before this command can be executed. The difference is that a marker takes an Event and hence can be used as a dependency for future commands.

7.1.9 MemoryObject

An OpenCL MemoryObject is an object representing a region of global memory on the CommandQueue’s Device. Some types of OpenCL MemoryObjects include Buffers and Images. MemoryObjects are created for reading, writing, or both, of a given size, and can optionally be copied from or to a host-side buffer. The naCL MemoryObject class encapsulates the actual OpenCL MemoryObject, as well as any MemoryMaps used in handling this specific MemoryObject. There are two methods of moving data between memory objects (whether on the device or host): copying and memory mapping. The naCL MemoryObject class’ constructor and destructor are both protected, such that a MemoryObject can be created or deleted by a naCL CommandQueue only. The MemoryObject keeps track of all created MemoryMaps, and will delete them during “atexit” if they have not already been destroyed.
7.1.9.1 DeviceBuffer

OpenCL does not provide a distinct Buffer class; a Buffer is just another type of MemoryObject. naCL contains a distinct DeviceBuffer class, as it provides a useful object-oriented approach to handling this specific MemoryObject type. A naCL DeviceBuffer inherits from the base MemoryObject class, and can be viewed, roughly, as a frame of data: a block of unformatted memory on the Device on which it was created.

7.1.9.2 DeviceImage

OpenCL does not provide a distinct Image class; a Image is just another type of MemoryObject. naCL contains a distinct DeviceImage class, as it provides a useful object-oriented approach to handling this specific MemoryObject type. A naCL DeviceImage inherits from the base MemoryObject class, and can be viewed, roughly, as a packet containing a data payload: a block of formatted memory on the Device on which it was created. An Image can be 2D or 3D, of a given size in pixels, and a variety of other properties.

7.1.9.3 DeviceRect

OpenCL does not provide a distinct Rect class; a Rect is just another type of MemoryObject. naCL contains a distinct DeviceRect class, as it provides a useful object-oriented approach to handling this specific MemoryObject type. A naCL DeviceRect inherits from the base MemoryObject class, and can be viewed, roughly, as a packet containing a data payload: a block of formatted memory on the Device on which it was created. A Rect is a rectangular region within a 2D or 3D buffer, allowing, for example, a contiguous local buffer’s contents to be
sparsely spread onto a Device’s buffer.

7.1.10 MemoryMap

OpenCL does not provide a distinct MemoryMap class; instead, methods are available via a CommandQueue to map and unmap a MemoryObject. On some OpenCL implementations a MemoryMap works just like one provided within the host OS when mapping from kernel-space to user-space memory: once the map is valid, data set in one space is available within a few clock cycles in the other space without first unmapping the memory. That said, as of OpenCL 1.1 data validity when using a MemoryMap is not guaranteed until the memory has been unmapped. Hence, correct use of memory mapping in OpenCL for reading/writing data to a Device is to create the map using a local buffer, read/write data from/to the local buffer, and destroy the map (“unmap” the local buffer). As of OpenCL version 1.1, a MemoryMap must be created using a valid MemoryObject, and refers to, effectively, a sub-buffer on the MemoryObject’s Device. In order to create a MemoryMap using a sub-buffer on the host side, one has to create a whole new MemoryObject and map it – or map the whole buffer along with an offset value specific to that buffer.

Different MemoryObjects use different OpenCL commands for the mapping and unmapping; hence the need for independent naCL classes for DeviceBuffer, DeviceImage, and DeviceRect. naCL contains a distinct MemoryMap class, as it provides a useful object-oriented approach to handling mapping and unmapping of Buffers, Images, and Rects: once the MemoryMap has been created, unmapping it is the equivalent of destroying the MemoryMap. An “unmap” method is provided such that actual naCL MemoryMap object need not be destroyed in the process,
just the OpenCL MemoryMap. The naCL MemoryMap class’ constructor and
destructor are both protected, such that a MemoryMap can be created or deleted
by the actual MemoryObject only – the naCL DeviceBuffer, DeviceImage, or
DeviceRect.

7.1.11 Event

An OpenCL Event holds the status of a command in a CommandQueue. All
OpenCL commands are placed into a CommandQueue in some order, but are
not necessarily executed in the enqueued order. Events can be attached to most
enqueue commands to set prior commands upon which the current one depends.
The naCL Event class encapsulates the actual OpenCL Event object as well as
the command status and timing information, including when the command was
entered into the queue, submitted for execution, started execution, and finished
execution. When using OpenCL 1.1, a callback method is provided to allow the
application to asynchronously determine when a command has finished execution.
The naCL Event class’ constructor and destructor are both protected, such that
an Event can be created or deleted by a naCL CommandQueue only.

7.2 Additions for Data-Flow Processing

In this section, we describe the additions to naCL specifically for performing
data-flow processing as done in Surfer and other SDR frameworks. These additions
utilize the buffer device affinity, runtime compiling, device properties, and queue
profiling information required by all OpenCL implementations. The most useful
addition is a class allowing for repetitive tasks to be performed in a trivial manner
once initialized, yet also allowing for specific tasks – naCL commands – to be
updated between task executions.

7.2.1 RepetitiveTask

Signal processing in any single SDR block when using OpenCL can be modeled as a repetitive task, with the primary changes between repetitions being where data is located (e.g., offset within a buffer) and how much data to process. Unless reconfiguration takes place, the executed binary will remain the same. When reconfiguration is required, all related naCL objects must be destroyed and re-created – but this process can be executed from the Surfer supervisor thread, asynchronously from data-flow processing. The actual repetition involves the three broad steps:

1. Pre-processing: Copy data that does not currently reside on the target Device to it.
2. Processing: Execute the Kernel.
3. Post-processing: Copy data from the target Device as needed.

Using this broad repetitive view of block-level signal processing, a RepetitiveTask class was created. The goal behind this class is to allow the user to create a sequence of tasks to be executed, in a given order and with set dependencies, and then execute that sequence using a single command with no arguments: execute(). naCL currently implements the following repetitive tasks, all involving the same CommandQueue:

- ExecuteKernel: Given a Kernel binary, enqueue it and its arguments for execution.
- MapDeviceBufferToHost: Given an already-created DeviceBuffer, enqueue a command to map it to a host buffer.
- Sync: Enqueue a barrier or marker synchronization point.
• UnmapDeviceMemoryObject : Given a valid MemoryMap, enqueue a command to unmap it.

Each RepetitiveTask recreates the actual task completely, down to Event dependencies. RepetitiveTask are used extensively in the interface with Surfer, as discussed later in this section.

7.2.2 Runtime Compiling

As already stated, Surfer makes extensive use of C++ templates. For example, all current block implementations are provided via C++ templates, which allows the definition of many block types very concisely. Using Salt’s item_type class as found in Section 4.2.7, naCL flavors can also be declared as templates and then internally modified to match the template requirements as well as the value of other variables.

For most Devices, OpenCL provides a runtime compiler that converts a ProgramSource into a Kernel – the binary executable specific to each Device. The ProgramSource is itself just a text string representing C-code compliant with C99 and the specific OpenCL naming conventions. Hence, runtime replacement points can be added to the ProgramSource string – e.g., using the string “%0” to represent the first replaced item, “%1” the second, and so forth. Each replaced item can represent anything that can be converted into a string, e.g., the name of a type, or the value of a variable. In this manner, a unique ProgramSource can be created to handle a specific instantiation of a given algorithm.

As a relatively simple example, consider an $N$-way synchronous adder, taking $N$ input streams and adding same-index items together to form one output stream. One implementation of a hypothetical block’s Process method – iterating across
items first then streams – could be implemented in C++ with the following code snippet, assuming that the number of streams to add together is at least two.

```cpp
template < typename item_t >
void Process
(vector < item_t* >& in_bufs,
 vector < item_t* >& out_bufs,
 vector < size_t >& n_gen_items)
{
    item_t* out_buf = out_bufs[0];
    item_t* in_0_buf = in_bufs[0];
    item_t* in_1_buf = in_bufs[1];
    for (size_t n = 0; n < n_gen_items[0]; n++) {
        out_buf[n] = in_0_buf[n] + in_1_buf[n];
    }
    for (size_t m = 2; m < in_bufs.size (); m++) {
        in_1_buf = in_bufs[m];
        for (size_t n = 0; n < n_gen_items[0]; n++) {
            out_buf[n] += in_1_buf[n];
        }
    }
}
```

The above code has the advantage of being easily vectorizable, and, indeed, some of the flavors for this block take advantage of this property. GPUs sometimes include vector types and related instructions, but these are typically much smaller than that provided by processor-native vector instructions. GPUs excel
at massively parallel execution, and vector instructions are generally a small part of the processing gain provided by GPUs.

The above code has the disadvantage of being difficult to parallelize because each value of the output buffer must be protected during each addition to make sure other threads are not accessing it at the same time. This protection adds overhead for the protection code as well as the delays while waiting for access. The other primary way of implementing this algorithm – iterating across streams first then items – is inherently parallelizable. This version is shown in the following code snippet, which is not restricted to two or more streams.

```cpp
template < typename item_t >
void Process
(vector < item_t* >& in_bufs,
   vector < item_t* >& out_bufs,
   vector < size_t >& n_gen_items)
{
    item_t* out_buf = out_bufs[0];
    for (size_t n = 0; n < n_gen_items[0]; n++) {
        item_t t_out = in_buf[0][n];
        for (size_t m = 1; m < in_bufs.size (); m++) {
            t_out += in_buf[m][n];
        }
        out_buf[n] = t_out;
    }
}
```

The above code is easily parallelizable and somewhat simpler to understand
than the easily-vectorizable version, and serves as a starting point for an OpenCL version. In OpenCL, a set of the outer loop iterations – the “for” loop over \( n \) – will be executed on each core of the selected Device. For the purpose of this example, let the set be just a single item per core such that each executing Kernel will perform the sum of exactly one output item, across all input streams. Assuming that there are \( n_{\text{gen\_items}}[0] \) executing Kernels and the function \( \text{get\_global\_id(0)} \) returns a unique number for each executing Kernel, then the following code snippet will perform the same algorithm as that found in the above easily parallelizable C++ code for two input streams of type “float”.

\[
\text{__kernel void kernel_sum\_generic} \\
\text{\{ \}} \\
\text{\quad size\_t i = get\_global\_id (0);} \\
\text{\quad float sum = (float) 0;} \\
\text{\quad sum += in\_0\_buf[i];} \\
\text{\quad sum += in\_1\_buf[i];} \\
\text{\quad out\_buf[i] = sum; \}}
\]

Abstracting the type and repetitive strings in the above code results in the following code snippet

\[
\text{\quad __kernel \text{ void} \}}
\]

138
kernel_sum_generic
(__global const int* generated_items,
%0
__global %2* out_buf)
{
    size_t i = get_global_id (0);
    %2 sum = (%2) 0;
%1
    out_buf[i] = sum;
}

where the % replacements are made in the following order

- %0 is a repeated string over %3, one per input stream number, created for all streams first and then replaced as a meta-string in the above code:
  " __global const %2* in_%3_buf,"

- %1 is a repeated string over %3, one per input stream number, created for all streams first and then replaced as a meta-string in the above code:
  " sum += in_%3_buf[i],"

- %2 is the name of the item_t type (e.g., “int”, “float”)

Surfer currently uses just simple replacements such as those described above. We call the current naCL flavors “generic” because they are not optimized in any significant way, and none use more than just the global id to determine their unique input and/or output item. Although the above OpenCL program performs as desired, the set of input / output items it processes over is probably not optimal when the total number of items is much greater than the total number of cores available for handling Kernel execution. For this and other cases, we will need set-handling versions of this OpenCL code, by executing the Kernel using properly set global and local work items.
7.2.3 Work Items

When executing a Kernel, OpenCL issues a number of “workers” – one per core for a user-specified number of cores. Each worker is assigned a unique global identifier that can be accessed in the Kernel code. This global id can also be broken down into group and local identifiers, which are also available in the Kernel code. Using these ids, the Kernel code can target specific data sets from within the overall input and output data provided to the Kernel. Figure 7.4 shows a work items scenario for the first 15 workers when the number of work dimensions is 1 – as will generally be the case for SDR-style data-flow processing – and the local work size for the first dimension is 5. In OpenCL 1.1 and prior, the global id is always the 0-based worker number index. For this collection of workers, there will be 3 groups – 15 workers / 5 workers per group – and each worker in each group will have a local id in [0,4]. Groups are highlighted with a dashed box.

One potential aspect of ProgramSource and Kernel runtime optimization involves setting the number of both global and local work items to closely match the Device’s physical architecture – primarily the maximum number of compute units and maximum work group size – while taking into account the number of
items to consume and/or generate. During normal stream processing, the number of items to be processed per input stream is known a-priori – the threshold value. Thus, the number of workers and groups can be pre-set for a specific Program-Source that is optimized for those specific values. Some future work, found in Section 7.4 includes determining optimized Kernels for the various naCL block flavors.

7.2.4 CommandQueue Profiling

The OpenCL 1.1 standard mandates that any CommandQueue must provide profiling – timing information that can optionally be retrieved for any Event. This information consists of a timestamp each for when the command was enqueued by the host, submitted to the Device, started on the Device, and finished executing. The RepetitiveTask for executing Kernels contains an Event that is used to determine the actual time spent performing data processing, such that the statistics for naCL flavors are more precise than simply measuring the time used by the call to the flavor’s Process method.

7.3 Connection to Surfer

All of the above discussion would be useless without a connection back to the Surfer framework. Given the amount of work put into making naCL highly functional for data-flow style processing, there are only two classes that need to be defined, both of which are discussed in this section.
7.3.1 Block Runners

The naCL block runner class inherits from `Surfer::block_runner_base`, which was described in Section 5.3. naCL block runners are specific to a given CommandQueue, which, reviewing Section 7.1.8, requires a specific Context and one or more Devices. We limit each naCL block runner to one Device, which is provided to the static create function used for creation and initialization of, and synchronization with, the runner thread. On initialization, this block runner creates a Context using the Platform on which the provided Device resides, and then a CommandQueue using the Context. The triplet of the Device, Context, and CommandQueue cannot be changed during the life of the runner. Other runners using the same Device can be created, each with its own Context and CommandQueue; OpenCL will internally figure out how to queue commands for the same Device. As OpenCL can be used with any type of processing device, one could create a naCL block runner for the host CPU – though using OpenCL on this Device will generally result in lower computational intensity due to the added overhead of OpenCL as well as not being able to use processor-specific vector instructions. Each naCL block must have pre-defined affinity for an naCL-base block runner such that when it the block is activated, the runner’s triplet of the Device, Context, and CommandQueue are available for setting up the naCL objects required for processing.

7.3.2 Flavor Base

The naCL base flavor class inherits from `Spf::flavor_base`, which was described in Section 6.2. naCL blocks are specific to a given naCL block runner, because each’s Kernel is compiled for a specific Device within a given Context,
and the Context must also contain the CommandQueue issuing commands to the Device. Since the `Process` method’s arguments are unchanging during normal runtime data-streaming, the number, types, and ordering of KernelArguments are created a-priori by the thread activating the naCL flavor. Argument ordering is as follows, assuming $M$ input buffers and $N$ output buffers.

- **State**: as a single contiguous buffer, no matter how big; read / write.
- **Number of Consumed Items**: as a vector of $M$ unsigned 32-bit integer values; read / write.
- **Number of Generated Items**: as a vector of $N$ unsigned 32-bit integer values; read / write.
- **Input Buffer 1**: as a vector of items of an a-priori known type and length in bytes; read only.
- **Input Buffer 2**: as a vector of items of an a-priori known type and length in bytes; read only.
- **...**
- **Input Buffer $M$**: as a vector of items of an a-priori known type and length in bytes; read only.
- **Input Buffer Offsets**: as a vector of $M$ unsigned 32-bit integer values; read only.
- **Output Buffer 1**: as a vector of items of an a-priori known type and length in bytes; write only.
- **Output Buffer 2**: as a vector of items of an a-priori known type and length in bytes; write only.
- **...**
- **Output Buffer $N$**: as a vector of items of an a-priori known type and length in bytes; write only.
- **Output Buffer Offsets**: as a vector of $N$ unsigned 32-bit integer values; read only.
All of the required naCL objects for the above KernelArguments, and their related RepetitiveTasks, are stored in the naCL flavor base, such that they can be destroyed when the flavor is deleted. They are created when this flavor is activated—not when it is initialized—because each naCL flavor has affinity for a specific naCL block runner. This flavor has non-overloadable methods for creating the Program from a ProgramSource and creating the Kernel from the Program, and an overloadable method for building the Program for one or more specific Devices. The last is overloadable because it involves compiling the source, which for some ProgramSources might involve special compiler flags.

The three broad repetitive steps for data-flow processing from Section 7.2.1 are expanded into 7 stages for naCL. When processing a block, the following RepetitiveTasks are issued in the listed order.

1. Enqueue un mappings of the write MemoryMaps created in stage 3—which is also the resulting stage when activating any naCL flavor.

2. Enqueue read MemoryMap creations for block state, certain output buffers, the vector of number of items actually consumed per input stream, and the vector of number of items actually generated per output stream: It is possible that a naCL Kernel will manipulate the state, the number of items consumed for any input stream, or the number of items generated per output stream. Hence, these data are retrieved after each processing, and then used by Surfer to update local copies of block state and buffer offsets. For those output buffers being used on different Devices or the host, a read MemoryMap is created such that as items are written to this Device’s buffer they are automatically copied to the destination buffer. These mappings are made dependent on each’s corresponding unmapping event from the prior stage, if it exists; otherwise, the MemoryMap will have no dependencies.

3. Enqueue synchronization marker: In order to guarantee that all of the mappings from the prior stages have taken place, a non-blocking marker is inserted into the queue; markers were discussed in Section 7.1.8.

4. Enqueue Kernel for execution, dependent on the Event generated in the prior stage. The various KernelArguments were already assigned during activation of the naCL flavor.
5. Enqueue unmappings of the read MemoryMaps created in stage 2. Each unmapping is dependent on the Kernel finishing execution from the prior stage.

6. Enqueue write MemoryMap creations for the block state, certain input buffers, the vector of number of items to consume per input stream, the vector of number of items to generate per output stream, and the vectors of input and output offsets per stream: The end-state – after initial activation or processing – for all naCL flavors is that all of the arguments to the Process are mapped for writing from host to Device, such that as items are written to them, they are automatically copied to the mapped Device’s buffer. As already mentioned in Section 7.1.10, OpenCL as of version 1.1 does not provide the capability of executing a MemoryMap using a sub-buffer on the host side; hence, offsets must be available to the Kernel for computing the first valid input and output data sample for each stream. For those input buffers coming from different Devices, a write MemoryMap is created such that as items are written to the input buffer they are automatically copied to the this Device’s buffer. These mappings are made dependent on each’s corresponding unmapping event from the prior stage, if it exists; otherwise, the MemoryMap be dependent on the Event from stage 4.

7. Enqueue finish: In order to guarantee that all MemoryMaps from the prior stage have been executed, a blocking flush command is inserted into the queue; this command is described in Section 7.1.8. This blocking behavior is desirable given the way MemoryMaps are used to handle data.

The above 7 stages provide a robust model for block-based signal processing using naCL with virtually no added overhead compared with not using the RepetitiveTask class to handle the stages. Further, once set up, the only changes needed between calls to Process are to update the input and output buffer offsets – a simple matter of copying typically a few elements per input and output stream.

7.4 Future Work: Optimized naCL Flavors

All current naCL flavors, excepting the externally provided ones for FFTs, are written in a generic fashion, consuming or producing one item per core. naCL programs can be optimized in three possible ways, which are interrelated. Because
each naCL program is constructed once and then used many times without having
to recompile it, and the construction is performed in its own thread – away from
data-processing functionality, programs can be optimized as much as desired. We
believe that these optimizations will bring naCL flavors performance much nearer
to that of Accelerate or other vector processing implementations for 1000 item-
threshold values or even lower.

One way is to dynamically create the program to be executed, which is already
done in Surfer in small ways. An example of major dynamic program creation is
an FFT that is optimized through loop unrolling and butterflies for the specific
length and data types being requested. Apple already provides such an FFT,
though it is more limited in scope that what would be required for Surfer. Many
blocks will benefit from parameter-specific programs.

Another way is to move data into local memory and use vector instructions
to increase the amount of data processed per instruction. Vector instructions are
already used for complex data – retrieving two non-complex items at a time – but
could be expanded into larger vectors depending on the actual number of items
to process and/or generate. Because in Surfer these numbers are fixed during
normal runtime processing for many blocks, programs can be optimized for the
actual threshold values, again through unwrapping of loops.

The final primary type of optimization is to maximize the utility of each core,
such that OpenCL does not have to iterate through multiple rounds of core usage.
For example if the GPU has 1024 cores, then in the current generic naCL when the
number of items requested is more than 1024, OpenCL internally determines that
two rounds of processing are required and allocates resources accordingly. Multiple
rounds add latency to the overall computation, and hence should be avoided when
possible by grouping items together and providing that set to each instantiation of the program. OpenCL provides information on each processing device, including the maximum number of cores and maximum group size. Although the total number of items for optimal processing is capped by the OpenCL API to the device, for modern GPUs this number is generally of much larger than that being processed by any SDR system. Hence, it should be possible to create a runtime algorithm for determining both the number of cores to use as well as the group size per core, while keeping these values within each device’s constraints.

7.5 Summary

We created a new C++ interface to OpenCL, called naCL, that is a more object-oriented interpretation of how OpenCL works. For data-flow style processing, a RepetitiveTask class was created to handle the 7 stages of MemoryMaps, synchronization points, and Kernel execution. In Chapter 9, we will provide throughput / latency plots for various flavors including some naCL ones, to show the runtime efficiency of this implementation. But first, we describe an alternative text-based interface for waveform definition.
CHAPTER 8

SALINE: THE SURFER ALGEBRAIC LANGUAGE INTERFACE

As already mentioned in Section 2.3.12, all SDR text-based interfaces use a block-centric approach to waveform definition. This block-centric approach to stream-based signal processing differs from that used by industry standard applications such as MathWorks MATLAB and GNU Octave [46]. Going back to the early 1980’s, MATLAB (among related projects) has provided digital signal processing capabilities with a relatively simple learning curve. MATLAB script is written in a buffer-centric algebraic-like programming language, which is now being used by millions of end-users – many in academia but significant numbers in industry as well [88]. By “algebraic-like”, we mean a mathematical expression in which only numbers, variables, and arithmetic operations are used. MATLAB scripts can currently work with scalars, arrays, and matrices of varying types, but not with streams beyond splitting stream data into arrays and doing array processing. One goal in this work is to show that MATLAB-style C++ code can be made to work with streams using data-flow techniques.

Block-centric (traditional SDR) and buffer-centric (industry standard) digital signal processing can be used to accomplish the same task – albeit using different language abstractions – so long as the resulting graph is acyclic; they are dual representations of the same problem under the acyclic graph restriction. For example, in typical Monte Carlo experiments for testing a channel-coding model,
vectors of random data are generated and then processed using operations in a set order that represent the encoder, channel, and decoder. With each new vector of random data, the simulation converges towards a result; each operation may keep state between random vector iterations. Such simulations are often written first in a MATLAB-style script because it generally provides the fastest development time. If the interpreted script is too slow, then it can be converted into a compiled language.

To perform such a Monte Carlo simulation using MATLAB-style script, an extended loop is used, within which the random data is created and processed; statistics are kept using variables declared outside of the loop. Such a simulation can also be performed using SDR-style processing techniques, where the random data vectors become frames generated by a random source, and the ordered operations are SDR blocks joined to make a “waveform” of sorts, performing signal processing on those frames. Statistics can be handled either directly inside a decoder sink, or via a callback from the sink. The primary difference between these MATLAB-style and SDR-style scripts is the programming abstraction. One of our goals in this work is to start bridging the gap between these styles, with the hope of making SDR-style programming more accessible, and with a more-familiar learning curve, to programmers who learned signal processing using MATLAB-style programming.

Our Surfer SDR framework aims to enhance the user’s experience by pushing complexity into the framework’s programming. Continuing this trend from the perspective of reducing the user’s learning curve for creating script-based waveforms, we augmented Surfer to provide an alternative, algebraic-like language interface – using a buffer-centric approach similar to that provided by MATLAB
and Octave.

The Surfer algebraic language interface (Saline) is written in C++, as an independent layer that resides in its own C++ namespace, and provides an algebraic-like language interface for waveform definition. Saline currently weighs in at around 1k lines of code, not including comments, implementing 3 primary classes. Saline is split into the core user accessible classes, functions, and macros, and an interface to the underlying SDR framework. Although Saline is designed with Surfer in mind, an interface into other SDR frameworks, e.g. GNU Radio, could be created relatively easily. Saline is an abstraction layer that merely acts as an alternative means for coding waveforms in comparison to current techniques; it has no more overhead that any other technique. The underlying SDR framework supplies the blocks / components and actual connections methods, and thus the framework is responsible for any heterogeneous processing – e.g., via OpenCL – or specialized instructions such as SSE, Neon, or Altivec.

Both Surfer and Saline make extensive use of C++ templates; for example, all currently implemented Surfer blocks are written as templates and thus must be explicitly instantiated by the user’s waveform application. By using template classes for all blocks, Surfer avoids code redundancy and related bug duplication issues, and also allows for easier debugging of code issues because the source is directly available as a header file. That said, explicit-typed Saline functions and Surfer blocks can be created and utilized through the use of Surfer’s signal-processing flavors.

Appendix B describes various specific C++ features used in our Saline implementation, and is provided for completeness. Templates in and of themselves cannot be used to create a robust algebraic abstraction in C++. We still desire
standard math operators (e.g., +, *, &, <, and %) to be available; in C++, we can use operator overloading to fulfill this need. Given the availability of templates and operator overloading, one can almost construct a C++ extension providing an algebraic-like abstraction. The missing key is for operator overloading in cases when the argument types are not identical. In this case, in order for the C++ code to compile and function correctly, the arguments’ types must be able to be compared. The C++ standard library type_info class provides this utility. With the above three C++ concepts in mind, we now describe the C++ extension allowing for algebraic-like language waveform definition.

Section 8.1 contains a comparison of block-centric and buffer-centric programming styles. In Section 8.2, we describe the Saline programming implementation, with emphasis on the types of operators required to create its algebraic-like interface as well as how C++ templates can be used to promote type propagation through the waveform graph. Example scripts and C++ code snippets are provided throughout, displayed in the Courier font in order to help set them apart from the rest of the text. An example using Saline is shown in Section 8.3. A summary of this chapter, finally, is provided in Section 8.4.

8.1 Abstraction Comparison

In the following subsections, we provide scripts showing different programming abstractions for waveform description.

8.1.1 Block-centric Abstraction

Current SDR frameworks, in defining the waveform graph via scripts or text-based programs, use a block-centric abstraction. In this abstraction, each block
is created and then connections are formed between adjacent blocks’ output and
input ports. Data is manipulated by each block’s signal-processing algorithm,
taking data from input ports, performing processing, and then writing data into
output ports. In a block-centric language, individual buffers are generally hidden
from the user’s script; the primary user-interface is each block: its instantiated
object, input / output ports, and state.

Listing 8.1: Polyphase downsample-by-N written using block-centric program-
ming.

```c
output = pp_down_N_block (input, N, options)
{
  // declare blocks first
  s2p = serial_to_parallel (N, options)
  for n = 1:N {
    filter[n] = fir_filter (options.ppf[n])
  }
  acc = sum (options)
  // connect blocks second
  connect ((input, 1), (s2p, 1))
  for n = 1:N {
    connect ((s2p, n), (filter[n], 1))
    connect ((filter[n], 1), (acc, n))
  }
  return (acc)
}
```

As a simple example of block-centric programming, consider the polyphase
downsampling-by-N implementation found in Listing 8.1, written in a script
combining features of MATLAB and C++ in order to reduce code complexity
while providing the features necessary for comparison and contrasting with other programming abstractions. This listing shows a function named `pp_down_N_block` that takes three arguments (`input`, `N`, and `options`) and returns one (`output`). The function `serial_to_parallel` takes a stream of items and parses them to `N` output streams, in order and without duplication. The function `fir_filter` creates a finite-impulse response filter using the provided vector as the filter taps.

For the function `connect`, the first argument pair always refers to a block and its output port, and the second argument always refers to a block and its input port; this function creates a graph connection between the provided pairs. The `connect` function is meant for demonstration purposes only, and can be assumed to be type-agnostic. For the sake of simplicity, we assume for this and related listings that array and port indices are 1-based (i.e., start numbering with 1, not 0), and that `options` contains the polyphase filter coefficients in the variable `ppf[n]` as well as anything else required for instantiation the blocks. For block-centric programming, the variables `input`, `s2p`, `filter[n]`, and `acc` refer to instantiated block objects.

In this code listing, blocks are created first and then connected together to form the waveform graph; one could combine the block creation and connection stages, but it does not change the underlying abstraction. Note that with this abstraction, the waveform graph can be connected in any order: from source to sink, sink to source, or from internal blocks towards both the source and sink. No matter the graph creation ordering, `connect` calls are required to form the graph.
8.1.2 Buffer-centric Abstraction

MATLAB and Octave scripts are written in an algebraic-like language using a buffer-centric abstraction to define signal-processing algorithms. In this abstraction, data – in the form of scalars, arrays, or matrices – is manipulated by functions in the user’s script via data buffers. In a buffer-centric language, objects are exposed in the user’s script to the degree that the user and language allow.

Listing 8.2: Polyphase downsample-by-N written using buffer-centric programming.

```plaintext
output = pp_down_N_buffer (input, N, options)
{
    s2p = serial_to_parallel (input, N, options)
    acc = 0
    for n = 1:N {
        // 'acc' reused
        acc += fir_filter (s2p[n], options.ppf[n])
    }
    return (acc)
}
```

Listing 8.2 provides a buffer-centric script for the polyphase downsample-by-N function. This listing shows a function named pp_down_N_buffer with the same function arguments and return as pp_down_N_block, and where the internally used functions have the same purpose. For buffer-centric programming, the variables input, s2p, and acc refer to output buffers from previous operators. The line acc = 0 is shorthand for zeroing out the buffer before it is used; this code is written for clarity, not efficiency. Because connections are defined implicitly through the manipulation of buffers, the underlying waveform graph is
abstracted away from the user’s script and hence no `connect` calls are required. The use of the `+=` operator provides a more intuitive language interface than that found when using block-centric programming, and also allows for certain runtime graph optimizations – both of which will be discussed further in later sections.

8.2 Algebraic Abstraction

Saline is written as an independent user-interface layer that resides in its own C++ namespace, and provides an algebraic-like language interface for waveform definition. This section describes the basic classes and concepts required to implement Saline, including the types of variable classes and operators, how templates are used to perform type propagation through the waveform graph as it is being defined, and its runtime operation. We then briefly describe the interface to the underlying SDR framework. As SDR is another form of scientific programming, we follow the general models already set up in that domain [37].

8.2.1 Types of Variables

Algebraic expressions are combinations of variables and operators on those variables. A variable might represent static data in the form of a scalar, vector, matrix, or constant stream, dynamic data being generated by a source, or processed data generated by an operator. In order to represent these algebraic expressions using Saline, three basic variable-oriented classes are required. Examples of the latter two classes are provided after their definition.

1. A `base stream` class from which all other stream-oriented variable classes are derived.

2. An `operator` class that can be either directly or indirectly instantiated, which represents the output buffer(s) resulting from some specific operator. When
multiple output buffers are available, they are obtained using array indexing via overloading the C++ \texttt{operator[]} method.

3. An \textit{enclosure} variable class that contains a reference to an operator variable. Enclosure variables are either explicitly declared in the waveform script or program, or created as temporary placeholders when multiple operators are executed before the \texttt{operator=} method is issued. When declared as the latter, a new object is created and knowledge of this memory allocation is retained by Saline for later deletion.

The first two classes' prototypes can be defined as

\begin{verbatim}
namespace Saline {
    template < typename item_t >
    class stream_base;

    template < typename item_t >
    class enclosure :
        public stream_base < item_t >;
}
\end{verbatim}

such that the enclosure class inherits from the stream_base class, and both are defined within the Saline C++ namespace. Prototypes for operators such as \texttt{fft} and \texttt{serial_to_parallel} can be defined similarly to that of the enclosure class. Note that the stream_base template type defines the output buffer data type, as is required by buffer-centric implementations. For Saline, types are defined explicitly at compile type; when using runtime-compiled kernels such as those available in OpenCL, types can be defined at compile-time or runtime as needed. An example of runtime compiling was provided in Section 7.2.2.

For the case in which a single template type is used in the class definition, then at least one input or output stream must be of that type. It is possible to use explicitly typed streams when defining operators, but this definition technique is discouraged. In order to use input and output streams of different types, multiple
template parameters can be provided and used. Thus, for example, to create an
fft operation taking in, processing, and returning streams of different types, one
could use the class prototype

namespace Saline {

    template < typename in_t,
                typename proc_t,
                typename out_t >
    class fft_i_p_o :
        public stream_base < out_t >;

}

Note that only the output buffer type of the new fft class is provided to the base
stream class.

8.2.2 Types of Operators

There are six primary C++ compatible types of operators needed to form
Saline. Each is described briefly, with example functions. For all operator types
below, op refers to an operator function, options to user-supplied initialization
parameters, stream to a single class inheriting from Saline::stream_base,
and streams to two or more stream arguments separated by commas.

1. op (options) : Type 1 operators take no data streams as arguments,
only options. Examples include stream sources, such as reading from a file,
generating random data, or providing a constant value.

2. op (stream1, ..., streamN, options) : Type 2 operators take
an a-priori fixed number of input streams – one at minimum – followed
by options. Examples include many common math and signal-processing
functions, such as sin, and fft, and if the number of output streams is
known, serial_to_parallel.
3. `op(stream1, ..., options)`: Type 3 operators take an a-priori unknown, an thus variable, number of input streams – one at minimum – followed by options. This operator type is currently implemented as `(options, stream1, ...) because C++03 does not robustly handle a variable number of arguments to a function or method [132]. C++11 should provide the necessary functionality via variadic templates, which will allow for the desired argument ordering. Examples, under appropriate conditions, include `sum`, `parallel_to_serial` and `analysis_filterbank`, all of which in this case take multiple input streams and return a single stream. Certain native-C++ mathematical operators, such as `+`, are implemented as both this type as well as the next.

4. `stream op stream`: Type 4 operators are those that overload built-in C++ math functions. Examples include command math functions such as `+`, `*`, `&`, `<`, and `%`, but, instead of operating on scalars (or vectors, as provided by some libraries) these operators are overloaded to handle streams. C++ handles just a single instance of this operator at a time: the command `A + B + C` is interpreted by C++ to be `(A + B) + C`, where the parentheses denote operator ordering. Saline internally splits this equation into two separate equations: `tmp = A + B` and `tmp + C`, where `tmp` is a temporary enclosure variable allocated as a placeholder for the first sum. Under some circumstances, multiple type 4 operators can be combined together, as a sort of runtime optimization; this technique is described in the type 6 operator.

5. `stream = stream`: Type 5 operators overload the built-in C++ function `operator=`. The left-hand side (LHS) argument must be an enclosure variable; this property is checked for during runtime only.

6. `stream op= stream`: Type 6 operations overload a built-in C++ math function as well as set an enclosure variable. Examples of this operator include `+=` and `%=`, though some operators are specific to certain types (e.g., integers only). The LHS argument must again be an enclosure variable, and this property is checked for during runtime only. Defining the LHS stream as `A`, and the right-hand side (RHS) stream as `B`, this operation is internally expanded into either `A = tmp_A op B` when the operator referenced by `A` is not identical in name and all types to than that being requested; `tmp_A` is an temporary enclosure variable that holds the value of `A` when this expression is issued. When the operator referenced by `A` is identical in name and type to that being requested, the variable `A` is augmented with `B` as another input. This operator type allows for internal graph optimization beyond what any C++ compiler can provide. For example, the accumulator used in Listing 8.2 can be reduced from N-1 + operators into a single N-way sum,
as shown in Figure 8.1. Such runtime optimizations are currently limited to identical operators using identical stream types.

The runtime optimization shown in Figure 8.1 works best when there are multiple flavors of the block type available for use. For example when using the `sum` as shown in Figure 8.1, the stream and item ordering can be modeled as in Figure 8.2. A “small” number of streams and items would likely be handled most efficiently by a block utilizing vector-based addition – e.g., as provided by GNU Radio’s new VOLK [93], or Apple’s Accelerate framework [7] – where each CPU core can do the vector summation of a set of streams, across all items indices. A “large” number of streams and/or items would likely be handled most efficiently on a GPU or equivalent many-core processor, where each of typically 100’s of cores can do the summation of a set of items indices, across all streams. Using Saline and Surfer, the actual block flavor selection can be handled during runtime, once the number of input streams and items to process per stream are known.

Using the above operator types, we can now implement the functionality to use templates for type propagation through the waveform graph as it is created.

8.2.3 Type Propagation via Templates

As described in Appendix B, the C++ typeid facility is used to provide internal type-conversion, and templates are used for both operator classes and their methods to allow these operator variables to take and return the same or different argument types. This robust type handling means that the user’s script is not required to explicitly declare all function types. As an example using Listing 8.2 – which uses just type 2 operators – the `serial_to_parallel` function prototype could be defined in C++ as

```cpp
namespace Saline {
```
Figure 8.1. Runtime optimization of N-1 2-way adders into a single N-way summation.

Figure 8.2. Visualization of data ordering for a block with N input streams, each of which contains M items.
where the C++ template type, \( \text{arg}_t \), is used to define both the input and output stream types. Given the input argument’s type, the C++ compiler will choose the correct template expansion of the \text{serial_to_parallel} function. Similarly, the type of the chosen \text{fir_filter} function can be defined implicitly via the type of its input – in this case the variable \( s2p[1] \) or \( s2p[n] \). In this manner, variable types are implicitly used to determine function template expansions, and these types are propagated through the waveform program. This form of type propagation from input to output requires the sole constraint that any prior streams must already be defined and available for use before the current operator is instantiated. Thus, unlike block-centric programming – which allows the waveform graph to be defined in any order – Saline, and buffer-centric programming in general, implicitly restrict the user’s waveform definition to be from source(s) to sink(s). This definition order also prohibits direct graph cycles, and is the reason for our up-front acyclic restriction for duality. The acyclic restriction is already found in current SDR frameworks; hence Saline’s source(s) to sink(s) definition ordering is aligned with current SDR waveform definition practices.

8.2.4 Runtime operation checks

Certain features of Saline can be determined during runtime only; hence the user’s program will be checked for correctness / validity during runtime as well.
as compile time. The three primary cases in which runtime checking is performed are discussed below.

1. Variable overwriting : Consider the code

   ```
   Saline::enclosure < int > A;
   A = 5;
   A = 10;
   ```

   The last line overwrites the expression from the prior line, effectively hiding the prior setting of the variable A. C++ will compile the above code, and it will execute as directed no matter the variable overwriting. During runtime, Saline will print out a warning that the LHS variable is being overwritten; internally, the last line of code is reinterpreted as

   ```
   tmp_A = A;
   A = 10;
   ```

   where tmp_A is a temporary enclosure variable. This reinterpretation allows for a variable to be set for some more-legitimate purpose than the above example, and then overwritten once that purpose is no longer in scope. Variable overwriting can be performed any number of times, using any of the operators from Section [8.2.2](#) with each overwrite reinterpreted as a uniquely named temporary enclosure variable as above but for the given operation.

2. Implicit type changes : Consider the code

   ```
   Saline::enclosure < int > A;
   Saline::enclosure < float > B;
   A = 5;
   B = A;
   ```

   where the variable A is set to a constant_source with the integer value 5, and then the variable B is set to be the same value as A – except as the type float instead of int. Internally, the last line of the code is reinterpreted as

   ```
   tmp_A = Saline::type_converter < int, float > (A);
   B = tmp_A;
   ```

   where tmp_A is a temporary enclosure variable holding the type-converted version of the variable A. When the user’s code is augmented in this fashion, Saline will print a warning about the implicit type conversion, allowing the code to execute but letting the user know about the potential issue.

3. Variable declaration order : Consider the code
where the variable \( A \) is set to \( B \) before \( B \) is set to anything. This code will compile in C++ without warnings, but does not make algebraic sense because the variable \( B \) has not been set before it is used. Saline will throw a runtime error when executing this code, stating that the RHS variable is being used before being set.

8.2.5 Interface to the Underlying SDR Framework

As each operator is created in Saline, its corresponding signal processing block is instantiated by the Saline SDR interface layer using the underlying SDR framework. This layer is responsible for instantiating, connecting, and deleting the SDR framework's blocks; it keeps track of buffers and related objects, but not the SDR blocks. It also provides the glue to manipulate the runtime status of the SDR framework – for example starting, pausing, locking, unlocking, and stopping both individual blocks and the runtime system. Surfer can generally handle the insertion and removal of blocks, during runtime without having to stop and restart processing. Further, this layer when using Surfer automatically starts system-level processing before the user's waveform program is executed, and stops it when the program is finished.

8.2.6 Example Block Implementation

As an example of a whole Saline block implementation, the class implementation for the N-way sum from Figure 8.1 is provided in Listing 8.3. Saline blocks use their own namespace: \( \text{::blocks} \) and then the block operation name. Each block must inherit from \( \text{Saline::stream_base} \), and is responsible for keeping track of any enclosed SDR block(s). Each SDR block can either be created in the Saline block's constructor or in the init method depending, respectively, if the
block has a default constructor or not. If the init method is used for constructing the SDR block, then it must be called immediately after the Saline block has been constructed, from inside the user-space operator function (e.g., \texttt{operator+} or \texttt{operator+=} for a sum block).

The \texttt{create\_block} function takes the type of the block to create, and returns a pointer to the new block; any arguments passed to this function are passed on to the block’s constructor. This function is used to allow Saline to keep track of the SDR blocks that have been created – along with various other variables associated with the SDR framework. Because the Saline block encloses the SDR block, the method \texttt{operator->} – which returns a reference to the SDR block – is the easiest way to access its exposed methods; one could replicate the SDR block’s public interface in the Saline block, but the use of this method is preferred due to its simplicity.

The SDR block can be initialized via the Saline block’s \texttt{init} method, which is unique to each block’s implementation and hence its prototype can be whatever is needed for creating and/or initializing the specific SDR block. For the sum block, there is nothing to do for \texttt{init}, which is why the \texttt{create\_block} function for this listing is found in the constructor; this method is shown in the listing for completeness. The SDR block is deleted in the Saline block’s destructor via the Saline function \texttt{delete\_block}; this function will determine any remaining connections associated with the SDR block, and delete them as well. Each block should provide the method \texttt{op\_name}, which returns a short description of the operation – in this case, we use an \texttt{item\_type} to determine the name of the \texttt{type\_t} type, and create a string with the type name and a “+”.

Listings \texttt{8.4} and \texttt{8.5} show the programming providing for the “+” and “+=”
operators, respectfully; both follow the same basic programming path, with the primary difference being the Saline function called to handle error checking, buffer creation, and block connections. Each operator takes two streams as the input arguments. The RHS argument is checked to make sure it is of the correct type, and if not then the Saline function check_io_types prints a warning about the required implicit type conversion, and inserts a type converter block between the LHS and RHS blocks. If no conversion is necessary, then this function returns a pointer to the original argument; otherwise, it returns a pointer to the new type converter. The LHS argument and the type-corrected RHS are then provided to a function used to do the actual operation creation – in this case a Saline sum block. The Saline function create_x_op_y verifies that both the LHS and RHS are valid – the output of some other operator – and then checks for the possibility of the runtime optimization shown in Figure 8.1 before creating a new operator and setting its arguments to those provided. A function version of the “+” operator can easily be defined via a template function or macro such as

```c
#define s_sum(X,Y) ((X) + (Y))
```

The Saline function create_x_op_equals_y verifies that the RHS is valid, as before, and that the LHS is an enclosure variable, then checks for the possibility of the runtime optimization before creating a new operator and setting its arguments to those provided.

Creating Saline blocks is relatively straightforward once the underlying logic for stream classes is worked out. Now that we know how to create an actual block, we put the pieces together using the downsampler script.
template<typename item_t>

class Saline::blocks::sum::ai_block :

    public Saline::stream_base < item_t >
{
    public:

        typedef Surfer::Blocks::sum::sp_block < item_t > sdr_impl_t;
        typedef sdr_impl_t* sdr_impl_p;
        typedef sdr_impl_t& sdr_impl_r;

        sdr_impl_p d_sdr_impl;

        ai_block (const std::string& name) :
            Saline::stream_base < item_t > (name),
            d_sdr_impl (Saline::create_block < sdr_impl_t > ()) {}}

    virtual ~ai_block () { Saline::delete_block (d_sdr_impl); } 

    void init () {};

    virtual std::string op_name () const {
        item_type it = item_type::create < item_t > ();
        std::string name ("< ");
        name.append (it.name ());
        name.append (" > +");
        return (name);
    }

    sdr_impl_r operator-> () { return (*d_sdr_impl); }
}
Listing 8.4: Saline `operator+` function implementation.

template < typename LHS_T,
    typename RHS_T >
Saline::blocks::sum::al_block < LHS_T >&
operator+ (Saline::stream_base < LHS_T >& lhs,
    Saline::stream_base < RHS_T >& rhs)
{
    Saline::stream_base < LHS_T >* rhs_to_use =
    Saline::check_io_types < LHS_T > (rhs, "x + y");
    return (Saline::create_x_op_y < Saline::blocks::sum::al_block
        < LHS_T > > lhs, *rhs_to_use, "operator+"));
}

Listing 8.5: Saline `operator+=` function implementation.

template < typename LHS_T,
    typename RHS_T >
Saline::blocks::sum::al_block < LHS_T >&
operator+= (Saline::stream_base < LHS_T >& lhs,
    Saline::stream_base < RHS_T >& rhs)
{
    Saline::stream_base < LHS_T >* rhs_to_use =
    Saline::check_io_types < LHS_T > (rhs, "x += y");
    return (Saline::create_x_op_equals_y
        < Saline::blocks::sum::al_block < LHS_T > >
        (lhs, rhs_to_use, "operator+="));
}
8.3 Example Saline Use

Listing 8.6 provides a Saline-based version of the polyphase downsample-by-N function. This listing shows a function named \texttt{pp\_down\_N\_Saline} with the same function arguments and return as \texttt{pp\_down\_N\_buffer}, and where the internally used functions have the same purpose. For buffer-centric programming, the variables \texttt{input}, \texttt{s2p}, and \texttt{acc} refer to output buffers from previous operators. When compared with Listing 8.2, the line \texttt{acc = 0} is not used because it is interpreted to mean a \texttt{constant\_source} of value 0, which at least for the used operator (+) is unnecessary. This code is written for efficiency, and does not include error checking on the inputs as would be typical of such a function. Also, this code includes all of the required C++ glue for compiling, while Listing 8.2 is meant as an example of an interpreted script.

\begin{Verbatim}
Listing 8.6: Polyphase downsample-by-N written in C++ using Saline.

namespace Saline {
    template < typename arg_t >
    stream_base < arg_t > pp_down_N_Saline
    (stream_base < arg_t >& input, size_t N, options_t& options)
    {
        enclosure < arg_t > s2p, acc;
        s2p = serial_to_parallel (input, N, options);
        acc = fir_filter (s2p[1], options.ppf[1]);
        for (size_t n = 2; n < N; n++) {
            acc += fir_filter (s2p[n], options.ppf[n]);
        }
        return (acc);
    }
}
\end{Verbatim}

168
This listing shows a number of the properties mentioned above: implicit operator type selection, type propagation from input to output, operator overloading for greater code clarity, and the use of the += operator for potential runtime optimization. Compared with Listing [8.1] the core programming (not the C++ glue) reads similarly to MATLAB or Octave script – but for data streams instead of scalars, vectors, or matrices. In more complicated examples – e.g., OFDM modulation or demodulation – Saline programming will reduce the chances of incorrect connections as well as the overall program length. Providing a scripting experience similar to that of MATLAB should make the transition from general-purpose signal processing to SDR easier for many users.

8.4 Summary

We have developed an SDR extension in C++ that provides an algebraic-like programming language interface. This extension, called Saline, currently works with our Surfer SDR framework but has been designed independent of Surfer and hence could be ported to other SDR projects. We accomplished this task by leveraging standard C++ properties and classes to define the variable types and operators, and creating a form of variable type propagation through the use of appropriate template functions and classes. Saline reduces complexity compared with current SDR text-based programming interfaces by using a MATLAB-style buffer-based implementation that provides implicit graph-style connections. Saline is an alternative API for using Surfer, and, as a recent addition, examples have not yet been ported to using it; we leave that as future work. The next chapter presents examples of Surfer performing actual signal processing, swapping flavors, and be-
ing interoperable with GNU Radio via an OFDM-based transmission / reception demonstration.
CHAPTER 9

EXAMPLES

In the last few chapters, we have described numerous design and implementation aspects of the Surfer SDR system. This chapter presents examples of Surfer performing signal processing via a relatively simple narrowband FM (NBFM) demodulator and downsamplers, and a somewhat more complicated GNU Radio-compatible over the air OFDM transmitter. The former example demonstrates, as best can be in writing, that the supervisor interface provides a robust means for controlling system runtime behavior through data-specific callbacks. In this example, flavors are swapped at a trigger, while seamlessly continuing to process data. The latter example shows interoperability with GNU Radio via using a receiver programmed in GNU Radio to decode the OFDM encoded audio and play it out on the host computer’s speakers.

The test computer for all examples is an Apple MacBook Pro containing an Intel Core 2 Duo (T9500) CPU running at 2.6 GHz, and an NVIDIA GeForce 8600M GT GPU. This computer is running Mac OS X 10.6.8 and only typical applications such as email and a music player; during testing, all non-Surfer applications were executed in the background. This OS provides native OpenCL 1.0, with a runtime compiler and queue profiling. This computer was used because it was the primary laptop of its time containing OpenCL capable hardware; the laptop form factor was chosen for portability, being able to take it to conferences.
or for company demos. Given that this computer already represents 4 year old technology, not to mention uses a low-powered GPU as required for extended laptop battery life, we feel that the performance of all examples will only be improved with updates to modern hardware. Further, using a dedicated GPGPU such as the NVIDIA Tesla should dramatically improve naCL flavor performance. These are left as future work.

9.1 Narrowband FM Decoder and Downsampler

Figure 9.1 shows a simple application graph that demonstrates that even a basic supervisor monitoring the CPU load can provide useful monitoring, and Surfer can provide seamless processing using flavors. Approximately 9 seconds worth of NBFM data was taken using an Ettus USRP1 and GNU Radio, and stored into a file. The NBFM data is decimated within the USRP1 from 64 Mega-samples / second (MS/s) down to 250 kS/s. Each stored sample is a complex-valued (in-phase and quadrature) integer with 2 bytes per value or 4 bytes / sample, requiring at least 1 M-byte/s sustained throughput for real-time processing. This file is then used as the data source for the example, where the FM signal is optionally rate limited to 250 kS/s and then decoded via a quadrature demodulator. The resulting audio signal is downsampled by a factor of 10, to 25 kS/s, via a low-pass FIR filter using 1651 taps and with a cutoff frequency of 2.7 kHz; we chose the long filter to test out the functionality of different flavors. The resulting audio signal is then stored back to another file.

As shown in the figure, the file source and sink have just a single flavor utilizing the host OS FILE API. Both the quadrature demodulator and downsampler have flavors that can execute on the local CPU as well as GPU (via naCL). On
the local CPU we provide a generic flavor using no external APIs, and one that is hand-optimized to use Apple’s Accelerate framework. The naCL flavor uses the host computer’s GPU and is not optimized; it assigns each GPU core a single output item to create instead of a set of output items. We believe that further optimizations will result in somewhat better performance, as discussed in Section 7.4.

For this example, the rate limiter was used to simulate real-time processing as if the incoming data were being generated by a USRP1 in the matter in which is was collected. A supervisor was initialized to monitor the CPU load of the host computer at the rate of 20 times a second; the actual CPU load callback sequence is depicted in Figure 9.2. When the CPU load is at 60% or higher for 3 sequential samples, then the supervisor swaps flavors to naCL in order to reduce Surfer’s CPU load. While swapped, once the CPU load decreases below 60% for 3 sequential samples, then the supervisor swaps flavors back to the original
Figure 9.2. Narrowband FM supervisor sequence, demonstrating swapping flavors at a trigger point.

settings.

With a graphical CPU load display running and recording data, we started Surfer executing the NBFM waveform demonstration; the resulting recoded data is plotted in Figure 9.3 for both the overall CPU and just Surfer. After a few seconds, we then separately started an external process that fully loaded the host CPUs for a short duration. As shown in this figure, shortly after the external process reached our user-set threshold of 60% CPU utilization, the supervisor started moving blocks from executing on the local CPU to using OpenCL which generates a smaller CPU load for Surfer. During the switch in flavors, as well as the entire time the external process is running, Surfer continued processing data both seamlessly and in real time.

Once the external process finished execution and the CPU load dropped below 60%, the Surfer supervisor started switching flavors back from naCL to the local CPU, generic flavor. This switch resulted in the local CPU executing all of the
application’s flavors again, hence the resumed moderate load. Throughout this example, the host OS is running other user and system tasks, and hence there is a difference between the total CPU load and that incurred by Surfer alone. There is also a short lag before the supervisor switches flavors, due to the load detection algorithm. Note that Surfer maintains real-time throughput during the entire waveform execution time.

9.1.1 Latency and Throughput

Latency in SDR-style processing is a combination of the time required to accumulate data for processing – either while waiting or performing other tasks – and the time required to process that data – including any overhead associated with pre-processing and post-processing. Instantaneous throughput is the speed at which data can be processed by a given block or waveform, without restricting
data flow. For example with the NBFM receiver, if the rate limit is not used, then any processing results in instantaneous throughput. When data rates are restricted, e.g., via an audio source or UHD sink, then instantaneous throughput refers to the amount of data being processed by a given block divided by the time required for that processing.

Using the same demo code, but without the external CPU load application, we added a supervisor callback, as described in Section 5.5, to return runtime timestamps from all blocks and block runners. The frequency was set to just once, when manually triggered from an end-of-file callback from the source block, and to collect at most the last 1000 samples. Once end-of-file is reached, data processing trails off quickly and thus the collected statistics reflect waveform execution as would be normally performed by Surfer. Waveform execution was performed on the same computer setup as already mentioned, and under the same circumstances, but without the rate limit block so-as to be able to compare absolute performance.

Figure 9.4 shows the overall execution time for the four primary parameter sets, while Figure 9.5 provides the end-to-end throughput. As expected, the processing time at low threshold values is somewhat higher than at high threshold values primarily due to the amount of overhead time required to perform any signal processing. As the number of items being processed decreases, the amount of overhead time per item processed increases. The overhead time when using any naCL flavors is significant, and thus this flavor performs best at higher threshold values. For most threshold values, the all-generic parameter set performs worse than all other flavors. Quite interestingly, the accelerate and single-runner naCL flavors perform comparably even at moderate threshold values. The two-runner naCL result shows the using a single device with two queues can lead to extra
Figure 9.4. Narrowband FM waveform execution time versus source threshold, different flavors, 1 to 4 block runners (#CPU, #naCL), no rate limit, with affinity only.

overhead due to those blocks competing for the same resource. The single-runner naCL version does not have this issue, because only one block can be in processing in a given runner at any time.

9.1.2 Execution Timing

Throughput and latency computation is useful for comparing absolute system performance under different thresholds and flavors, at a high level. Execution timing diagrams show performance at a low level, e.g., at the time scale for a single set of source to sink processing iterations. The source to sink latency is the time from when the source is enqueued for processing until the sink finishes processing. These diagrams can show system inefficiencies due to poor mappings, excessive Waiting between threads, or a poor implementation to some external library. Our goals in these execution timing visualizations are to show that mapping of block
to running is important, and to explain timing diagram resulting from runtime data collection.

Figure 9.6 shows typical timing execution for the NBFM demo when viewed by block. In order to show actual preprocessing and postprocessing timing, we use the naCL flavor for the quad demod and downsampler blocks. Because only one block can be in processing at a given time on any block runner, the naCL blocks must take turns being processed; the quad demod spends as much time waiting after being queued for processing as it does being processed. As such, the source to sink latency is delayed by around 0.1 ms, with a total latency of around 331 $\mu$s. The source to sink path is highlighted, showing the actual path that data takes, when blocks are waiting for processing, and the notifications between blocks that data has been consumed or generated. If the quad demod and downsampler blocks could be processed concurrently then the resulting latency would be reduced by

Figure 9.5. Narrowband FM waveform throughput versus source threshold, different flavors, 1 to 4 block runners (#CPU, #naCL), no rate limit, with affinity only.
Figure 9.6. Typical execution timing for the poorly mapped NBFM demo, viewed by block, with runner thread 0 targeting the local CPU and runner thread 1 targeting the GPU via naCL.

almost 1/3 to around 200 µs.

Another way to visualize Figure 9.6 is to look at execution timing by runner thread, not block. This version is shown in Figure 9.7 for exactly the same data set as the prior figure. As expected, the GPU thread is running full out all of the time, while the CPU thread hosting the source and sink is sleeping most of the time. This mapping from block to runner results in non-equal thread loading, which in turn increases latency.

Changing the mapping of the quad demod from the naCL runner thread to the local CPU runner thread results in the execution timing diagram shown in Figure 9.8. Although the local CPU runner thread still spends time sleeping, this mapping provides better load balancing than the original mapping, and hence also
Figure 9.7. Typical execution timing for the poorly mapped NBFM demo, viewed by thread, with runner thread 0 targeting the local CPU and runner thread 1 targeting the GPU via naCL.
lower source to sink latency. The gaps between the source postprocessing and the quad demod waiting is when that thread is sleeping, waiting for the downsampler to notify the quad demod that it has consumed data and thus there is now enough output buffer space for the quad demod to perform processing. Using different threshold values to allow the quad demod block to be processed earlier would result in the sink also being processed earlier and hence lower overall latency.

From the above discussions we draw two conclusions. First, the mapping of blocks to runners is important when determining latency. Poor mappings results in delays while waiting for other blocks to finish processing and thus increase latency. Poor mappings result in non-equal loading, which also increases latency. Using one runner per block can, with the correct mapping, be made to provide
the best latency and throughput for a given threshold selection. But for many waveforms fewer runners can generally be mapped in such a way as to provide the same result.

Second, that the block threshold selection directly changes the input to output latency, and hence there is a throughput / latency curve describing the processing point for a given mapping and set of thresholds. Smaller thresholds generally result in smaller latency, while higher ones generally result in higher latency. Because the actual latency is the sum of waiting and processing times, each threshold value will have some impact on its value. Matching thresholds across relative sample rate changes from block to block will result in the minimum source to sink latency for that threshold value, but not necessarily the absolute minimum latency. Increasing a single threshold will result in the average latency being increased by time required for this data to be accumulated.

Through this relatively simple NBFM example, we are able to demonstrate a supervisor providing runtime control by choosing the flavor of execution. This supervisor uses runtime statistics, provided via a callback; although only the overall system CPU load was used, other statistics are provided via similar callbacks as configured by the user when the supervisor is initialized. This example further demonstrates that the techniques we developed for allowing this style of runtime dynamic reconfiguration can successfully process data seamlessly.

9.2 OFDM Over-the-Air Transmitter

The next example uses a more complicated graph, for the purpose of showing interoperability with another SDR system as well as seamless processing of audio on a live system. In Surfer we created an application for the waveform found in
Figure 9.9. OFDM transmitter waveform, including available flavors in which multiple are available.

Figure 9.9. This demo shows interoperability with another SDR system, as well as the seamless processing and runtime system behavior control that *Surfer* is capable of performing.

This waveform uses inline packet encoders, embedded in the data-flow streams, and three blocks created to mimic the signal processing performed in GNU Radio’s OFDM modulator. The mapper takes a *Surfer* style packet, and performs the mapping from bits to constellation using the packet’s payload – which is a GNU Radio-style embedded packet. The mapping data is passed to the GNU Radio style preamble inserter, which uses the flag stream to signify the start of each packet and thus when to insert the preamble. All of the blocks to this point contain a single, generic, flavor. The next four blocks are *Surfer* specific and provide multiple flavors as shown in the figure. The final block uses the Ettus Research UHD API to transmit data via a USRP1, at the pre-selected frequency of 5 GHz and data rate of 1 MS/s. Each output sample is of type `complex` <
float>, and thus for real-time processing the system must be able to handle generation of data at 8 M-bytes/s.

On the receiver end, we used the GNU Radio Companion to construct the waveform in Figure 9.10. Internal to the “OFDM Demod” block are blocks handling synchronization as well as the cyclic prefix removal, forward fft, and preamble detection. Internally, this block uses asynchronous packets to transfer data between itself and its next connections.

To demonstrate Surfer functionality, this system is executed with a supervisor taking input from the keyboard of the host computer to change flavors and thresholds. Flavors can be rotated through for each individual block, and the default is the “generic” flavor for all blocks except the inverse FFT which defaults to the Apple Accelerate vector optimized implementation. Thresholds can be increased or decreased to within limits, and must a multiple of some number of items—e.g., the inverse FFT must be in multiples of 512 items (one OFDM symbol). To demonstrate latency control, the supervisor was programmed with three pri-
mary settings, inside of which thresholds and flavors could be changed. These three settings are discussed below, and are: just defaults, high up-front threshold, and minimum FFT shift threshold. Our goal in this demonstration is primarily to show how changing threshold values will effect instantaneous throughput and latency.

The default threshold values follow the sample rate change along the graph, and result in processing exactly one input packet’s worth of data for each block. A typical timing execution chart as described for the NBFM receiver, under default settings, is shown in Figure 9.11. For these defaults, Surfer is handling processing at around 8 times what is required for real time, and results in a source-to-sink latency of between 1 and 2 ms. Performance could easily be improved by changing flavors such that all of them use vector processing, and all but the inverse FFT could also be handled on the GPU via the naCL flavor. The “generic” inverse FFT in naCL is not optimized, and cannot perform fast enough to handle real time processing; as already discussed in Section 7.4, optimizing naCL flavors is important future work for Surfer to be truly heterogenous of processing capabilities.

Switching to the high up-front threshold value of 10 packets on the GNU Radio packet encoder resulted in the timing execution found in Figure 9.12. Because this encoder is operating at the slowest sample rate in this waveform, increasing its input threshold dramatically changes the perceived latency in the audio loopback. The 1 to 2 ms latency incurred by the default settings is slightly noticeable, but the 100 ms latency found in this setting is very obvious. The figure shows that Surfer’s data processing is being performed as expected, e.g., with all of the notifications happening much faster than required for real-time processing. But, Surfer’s interface to UHD caused the instantaneous throughput to decrease to just
Figure 9.11. Typical execution timing for the OFDM demo under default settings, consuming exactly one input packet’s worth of audio data.
Figure 9.12. Typical execution timing for the OFDM demo when the GNU Radio packet encoder input threshold is set to 10 packets worth of data.

slightly more than required for real-time processing.

The final setting reverts the GNU Radio packet encoder input threshold back to one packet’s worth of data, and decrease the input threshold for blocks starting with the FFT shift down to one OFDM symbol’s worth of data. This setting represents the minimum latency obtainable by Surfer with the selected flavors. As shown in Figure 9.13, typical latency is reduced slightly but at the expense of more overhead spent for Surfer to process the same amount of data as found in the default settings. This figure also shows that the interface to UHD is not
functioning as well as it could, reducing the instantaneous throughput to very close to that required for real-time processing. As there are other ways to provide data to UHD, we will investigate alternatives to the method used for this demonstration as future work.

9.3 Summary

In this chapter, we presented two examples of Surfer waveform processing. The first example used a NBFM demodulator to demonstrate use of a simple supervisor
performing flavor swapping at a given CPU load threshold. The second example provided a live, over the air OFDM-based audio transmission by *Surfer*, with GNU Radio receiving, decoding, and pushing the audio to the host computer’s speaker. Both examples demonstrate the flexibility provided by *Surfer* to the waveform developer for runtime behavior modification, as well as that *Surfer* is capable of performing real-time signal processing and interoperating with other SDR systems. Although most of this thesis was dedicated to describing design and implementation aspects and issues, such as system is incomplete without real-world applications. Hence, these examples complete our thesis, which we summarize and provide fodder for future work in the next chapter.
CHAPTER 10

CONCLUSIONS

This thesis described numerous design and implementation aspects of a new framework for handling general-purpose software defined radio data-flow style processing, called *Surfer*. *Surfer* was developed for the purpose of allowing runtime system behavior control, with the overarching goal of moving complexity away from the waveform developer – placing it in *Surfer* – while still providing a highly functional yet configurable SDR. We chose examples that were relatively simple to implement, yet provide the necessary functionality to show that this implementation performs well and achieves our goals.

Various forms of dynamic reconfiguration are available in *Surfer*, including a novel form providing heterogeneous processing capabilities via multiple signal processing implementations, or flavors, in a single block. Each flavor, when provided with the same state and inputs, must compute the same outputs to within device precision. Each block contains a lookup table containing one or more flavors, and each flavor can target a different processing device and level of optimization. Flavors can be switched during runtime, generally seamlessly with respect to data processing. *Surfer* currently provides flavors that utilize the host computer’s CPU and GPU, and can easily be expanded to use more specialized processors.

Internally, to allow for both pre-determined mappings and schedules, a simple threshold is used on each input and output buffer. These values set the nominal
amount of data to consume or generate during normal data streaming; the actual amount consumed or generated depends on the block state. Each threshold is set as the number of items, and can represent a packet or frame of data. The use of thresholds reduces the complexity of determining when a block is ready to be processed to a simple value comparison per connection.

Surfer allows the user to define a supervisor that makes use of runtime statistics to control system behavior. Potential statistics include the overall CPU load and a specific block’s instantaneous throughput averaged over a number of samples. The supervisor can control almost every aspect of system runtime, from the mapping of blocks to processing devices via flavors, to the nominal amount of data to consume and/or generate per processing iteration. Feedback from currently in-place blocks can be used to dynamically change a waveform – e.g., for OFDM to use a different cyclic prefix length or to enable and disable specific carriers.

Surfer relies on a pool of block runners to handle processing, allowing for fully deterministic or opportunistic processing, or some mix. The supervisor can also add and remove blocks and block runners, as well as the affinity of any block to a given runner so long as their types match. In this manner, the various layers of Surfer’s implementation provide significant data-flow runtime capabilities.

For a script-based interface, Surfer provides two programming styles that can be intermixed as desired for waveform definition. Traditional software defined radio systems use block-centric programming – instantiate then connect – while Surfer provides a new one that uses a more MATLAB-like algebraic-like buffer-centric programming. Because of constraints inherent in an algebraic language, runtime checks and waveform optimizations can easily be performed that are not simple using the traditional approach. A buffer-centric interface reduces pro-
gramming complexity, with the primary tradeoff being that the waveform must be defined from source(s) to sink(s).

The work described in this thesis demonstrates that dynamic reconfiguration, mapping, and scheduling during runtime on can be implemented robustly commodity hardware, entirely in software. *Surfer* fills the role of being more accessible than heavily markup-based SDR implementations, but more reconfigurable than the easier to use implementations. Because real-time signal processing as provided by SDR systems is a relatively young field, we hope that this work provides useful insight into the development and implementation of such systems. Further, this initial foray into runtime experimentation is hopefully just the beginning.

10.1 Future Work

In the development of *Surfer*, we have come upon a number of interesting issues. Some were resolved on the spot because they were relatively simple to implement and augmented *Surfer’s* capabilities in desirable ways. Others are left as future work, and some of those are discussed in the following sections.

10.1.1 Cognitive Supervisor

*Surfer* provides the toolset to handle various forms of dynamic reconfiguration, from parameter based to adding and removing individual and clusters of blocks to changing the signal processing device via a flavor change. All of these forms of reconfiguration require a supervisor, and the ones shown as examples are meant purely to demonstrate that the supervisor concept works. Given the statistics being made available to a supervisor, we believe a much more sophisticated supervisor can be developed. Such a cognitive supervisor would merge the func-
tionality of cognitive radio with runtime system control over mapping, scheduling, and throughput / latency.

For cognitive radio functionality, all possible dynamic reconfiguration should be utilized. One particular area where Surfer can stand out is in adapting the runtime waveform to meet quality-of-service (QoS) requirements. In some communications systems, the QoS is set via some control mechanism, in the receiving waveform – much like parts of today’s cognitive radio. For example, the IEEE 802.22 standard provides three primary levels of service, each with four sub-levels, providing data rates from about 4.5 M-bits/s to 22.7 M-bits/s. 802.22 requires the ability to change the coding rate, cyclic prefix length, carrier modulation, channel bandwidth, and channel center frequency in the transmission waveform. Surfer is capable of handling runtime modification to each of these parameters, and thus could handle the required cognitive radio functionality.

We are proposing to augment Surfer to allow the waveform developer to set a range of block types for a specific purpose, and then allow the cognitive radio to handle determination of which actual type to use. For example, we envision having a single “coding” block that can be dynamically switched between different rates, which are selected by the cognitive supervisor based on statistics from monitoring the spectrum as well as how Surfer is internally performing. So long as the coding block meets the topology requirements, then it could be “hot-swapped” to change the waveform functionality. Once the desired QoS is determined the various forms of Surfer’s dynamic reconfiguration can be used to select the appropriate scheduling and mapping parameters.
10.1.2 Machine Learning for Channel Estimation

Channel estimation is important for real communications systems, in order to adapt transmissions to current channel conditions. Typical estimation methods include least-squared and minimum mean square error (MMSE) by using a-priori known training data as part of the transmission preamble. Machine learning, e.g., via support vector machines or genetic programming, can also be used to perform channel estimation. In machine learning, a major focus is to create a system that automatically learns to recognize complex patterns and makes intelligent decisions based on input data. When using machine learning for channel estimation, the goal would be to compare traditional algorithms with machine learning-based ones, both for speed and accuracy. *Surfer* provides blocks that can retrieve training data and then issue a callback to the supervisor to handle this data. Once an estimate is available, the supervisor can perform system reconfiguration — likely just parameter based — to change waveform behavior. In this way, different estimation techniques including machine learning based ones can be evaluated offline with test data or live.
APPENDIX A

ACRONYMS, ABBREVIATIONS, CONCEPTS, AND TERMINOLOGY

There are lots of acronyms, abbreviations, concepts, and terminology required to understand SDR and related technologies. This appendix provides a list of acronyms and abbreviations used throughout this dissertation, as well as simplified definitions of the most important concepts needed to understand the contents of this dissertation.

A.1 Acronyms and Abbreviations

This section provides a list the acronyms and abbreviations used in this dissertation, listed alphabetically. Important ones are defined in later sections.

- ADC  Analog to Digital Converter
- AL   Abstraction Layer
- ALU  Arithmetic Logic Unit
- API  Application Programming Interface
- CORBA  Common Object Request Broker Architecture
- CPU  Central Processing Unit
CR  Cognitive Radio
DAC  Digital to Analog Converter
DAG  Directed Acyclic Graph
DSA  Dynamic Spectrum Access
DSP  Digital Signal Processor / Processing
FBP  Flow-Based Programming
FPGA  Field Programmable Gate Array
GPU  Graphical Processing Unit
GPGPU  General Purpose use of Graphical Processing Unit
GPP  General Purpose Processor / Processing Device
HAL  Hardware Abstraction Layer
IF   Intermediate Frequency
JTRS Joint Tactical Radio System
LHS  Left Hand Side
NBFM Narrowband FM
OFDM Orthogonal Frequency Division Multiplexing
 POSIX Portable Operating System Interface
RHS  Right Hand Side
SCA  Software Communications Architecture
SDF  Synchronous Data-Flow
SDR  Software Defined Radio
SIMD Single-Instruction Multiple-Data
SOC  System on a Chip
A.2 Concepts and Terminology

This section provides definitions of the most important concepts and terminology needed to understand this dissertation, listed alphabetically by term, acronym, or abbreviation.

- **Actor** : In this context, the same as a *block*.
- **Acyclic [Graph Theory]** : Not having or allowing cycles or data feedback in a graph. A cycle is when a node’s input somehow relies on its output via the graph (thus, feedback of a sort). Cycles are not used in SDR due to the inefficiencies introduced by the small amount of data in the feedback loop.
- **Adaptive** : During runtime, live, and changing over time often due to comparing and contrasting past information with new information.
- **Affinity [CPU]** : The ability to assign a given application, thread, or fiber to a specific device or core.
- **Affinity [Memory]** : The ability to assign given data to a specific memory (e.g., local cache, shared cache, global memory).
- **Ancestor [Graph Theory]** : Any node coming before the reference node (from which the ancestor quality will be determined). This term is mostly meaningful when working with acyclic graphs, because with a cyclic graph it is possible that a given node is its own ancestor.
- **Another OpenCL C++ Implementation** : A C++ framework, using the C++ namespace `naCL`, that uses Salt as the underlying API for interfacing with the OS, compiler, and OpenCL in a more object-oriented way than the standard OpenCL C++ framework.
- **Application [SDR]** : A collection of one or more waveforms, executing within the same SDR framework, possibly concurrently.
- **Application Programming Interface** : A set of header files and libraries providing classes, structures, variables, and functions that are used to perform some specific task(s).
- **Asynchronous** : Not synchronous; arriving or departing with irregular or unpredictable timing.
• Atomic [OS] : Executed in an uninterruptible and instantaneous fashion, such that the code being executed is in isolation with regard to other threads. For example, an OS lock must be, at some level, atomic in nature in order to guarantee that two threads do not acquire the lock simultaneously. Modern computing devices generally provide atomic instructions for the purpose of implementing a guaranteed mutual exclusion function – for example, via an atomic “read-write”, “test and set”, or “compare-and-swap” instruction.

• Block : The basic building element of a waveform / graph. Each block represents some signal processing algorithm; connecting multiple blocks together creates a graph that, generally, does something useful.

• Block Centric : Focusing on blocks for language definition, as opposed to buffer centric. In a block-centric language, blocks can be instantiated in any order, and then the blocks’ ports must be connected.

• Block Runner : See Runner.

• Buffer Centric : Focusing on buffers for language definition, as opposed to block centric. In a buffer-centric interface, blocks must be instantiated from source(s) to sink(s), and hence there is no way to create direct cycles in the resulting graph.

• Cluster : A sub-graph within the current graph; a contiguous grouping of connected blocks.

• CommandQueue [OpenCL] : A queue that holds commands to be executed on a single specific device within a single specific context. Depending on the OpenCL implementation and user preferences, commands are entered into a CommandQueue may be executed out of order, and timing information may be collected associate with each command: when the command was enqueued, processed, placed for execution, and finished executing.

• Compile Time : During the process of taking text code and generating object code that can be linked together to form an application or archived into a library. The text code should make use of macros determined at configure time that describe that features available in the OS and via installed frameworks. For example, checking to see if the function “bcopy” is available, and if not seeing if “memcpy” is, and so forth until a none are found and a generic version must be used.

• Component : Same as a block. This term originated with flow-based programming, and is also used by some SDR frameworks including JTRS SCA.
• Computational Intensity: The number of computations performed per unit of data transferred both to and/or from a given computational resource; the ratio of the time spent doing signal processing to that doing overhead tasks.

• Computational Resource: A physical processor, generally composed of one or more cores, capable of executing one application, thread, or fiber per core at any given time. Computational Resources can be a CPU, GPU, DSP, FPGA, ALU, SOC, or any other type of processor.

• Compute Unit [OpenCL]: One core of a processing device, no matter the device type. A CPU will generally have just a few compute units, while a GPU will typically have hundreds.

• Condition [OS]: A type of OS lock, augmented such that it can give up exclusive access while waiting for a signal from some other thread using the condition. Once the signal is received, exclusive access is regained before execution continues. In OS terms, a condition is also called a monitor.

• Configure Time: Before compile time, a time during which features available in the in the hardware and OS, and via installed frameworks, per the user-provided options (if any) are determined. For example, the user might specify to not use “reopy”, or to just use “memcpy”. Configuration generally determines the hardware basics such as processor type (and subtype, if applicable), compiler type, and the availability of various language types (and, often, their sizes in bits). One output of configuration is a set of macros that define what features are available to be used, and which are not – whether not available or disabled by the user.

• Connection [Graph Theory]: Same as an edge, the means by which nodes are linked. In a directed graph, a connection defines the relationship between the connected nodes.

• Connection [SDR]: A connection represents an SDR buffer, the means by which data is transferred between blocks. It also defines a relationship between blocks.

• Context [OpenCL]: A set of one or more OpenCL devices, the memory accessible to those devices, OpenCL programs and their kernels, and OpenCL command queues used to schedule execution of the kernel(s) or operations on memory objects. An OpenCL context is the environment within which an OpenCL program’s synchronization and memory management is defined.

• Context [OS]: The execution state of a thread, including CPU registers, stack, and any other relevant information such that if a context switch occurs – either voluntarily, because of a signal, or through normal OS initiated
preemption – thread execution can be re-started at the same point at which it was stopped as if it had never happened except that time has passed.

- Context Switch [OS] : The process of switching execution from one thread to another, at the OS level. The threads being executed may be within the same process or in different processes.

- Cooperative Multitasking : A means of sharing a given set of computational resources between one or more applications, where each application must voluntarily give up processing in order for other applications to execute; applications must cooperate in order for all to have time executing.

- Data Driven Processing : Same as push processing.

- Deadlock [OS] : A situation where two (or more) threads are waiting for the others to finish, and thus none ever does. A deadlock generally occurs through the use of an OS lock or related object.

- Dedicated Device : A computational resource that can be dedicated to execution of a specific application, thread, or fiber, typically with minimal OS overhead or intervention.

- Descendant [Graph Theory] : Any node coming after the reference node (from which the descendant quality will be determined). This term is mostly meaningful when working with acyclic graphs, because with a cyclic graph it is possible that a given node is its own descendant.

- Device [SDR] : Same as a computational resource, no matter if for SDR or OpenCL. An OpenCL device is any single computational resource on the host computer with an OpenCL interface. A multi-core CPU is counted as one device, with multiple “compute units”. OpenCL devices span the range from CPUs to GPUs, FPGAs to DSPs.

- Device [OpenCL] : Any single computational resource on the host computer with an OpenCL interface. A multi-core CPU is counted as one device, with multiple “compute units”. OpenCL devices span the range from CPUs to GPUs, FPGAs to DSPs.

- Digital Signal Processor : A processing device providing generic functionality as well as instructions tailored specifically for signal processing algorithm implementations.

- Digital Signal Processing : Programming that performs signal processing using digital samples; as opposed to analog signal processing, which is done using hardware only. The sample might come from data generated locally –
for example a sinusoid of a given frequency, offset, and sample rate – or via an analog to digital converter.

- Directed Acyclic Graph [Graph Theory] : A graph where connections are directed (from one node to other nodes), and where cycles are not allowed.

- Dynamic : Changing (and, generally, changeable) over time. Not pre-computed or otherwise predetermined.

- Edge [Graph Theory] : Same as a connection.

- Fiber : In modern OS terms, a fiber is the internal OS representation of a part of an executing application that can not be preempted by other fibers (but, it can be preempted by the host OS); fibers use cooperative multitasking, where each fiber must relinquish execution control in order for other fibers to be executed. Most modern applications do not use fibers, instead relying on threads. In older data-flow terms, a fiber is another name for a block.

- Flavor : The term given to each of a block’s signal processing algorithm implementations; each flavor might target a specific processing device or API, e.g., be generic, use Apple’s Accelerate framework, GNU Radio VOLK, OpenCL, AltiVec, or NEON.

- Flow-based programming : An programming style originating around 30 years before SDR, where an “information packet” is carried through a network of “components” that operate on the packets.

- Frame : A chunk of unformatted data. Generally transmitted / processed in a synchronous or isochronous manner.

- Framework [OS] : The collection of executables, libraries, and header, resource, and data files for a given project. In some OSs (e.g., Mac OS X), each framework is installed into its own directory structure set aside from other frameworks. In most UNIX system (including Linux), related files from each framework are installed into the same directory – e.g., all libraries are installed into a directory named “lib”.

- Graph [Graph Theory] : A collection of nodes and connections.

- Graph [SDR] : One or more waveforms; an application.

- Handle [C/C++] : A pointer to a pointer.

- Hardware Abstraction Layer : A software-based framework that provides a common API to the underlying hardware, no matter what type it is.
• Isochronous: Guaranteeing the availability of certainly data rate, though not necessarily that data is available at that rate. For example: in asynchronous packet transmissions using an isochronous data link, outgoing data is continually transmitted no matter if packets are available; the incoming packet data rate cannot be larger than the data link’s rate (including any encapsulation or unpacking overhead).

• Kernel [OpenCL]: An OpenCL program compiled for a specific device, with the capability of being queued for execution using a command-queue for the device.

• Kernel [OS]: The core of the operating system.

• Live [SDR]: Dynamically, during runtime.

• Live [User]: Ideally, instantaneously, or within human reaction time.

• Livelock [OS]: Similar to a deadlock, except that the threads are still executing and changing state, but not able to get beyond a certain point due to not having access to the shared resource.

• Lock [OS]: The broad category of mutual exclusions objects, including a mutex, semaphore, and scoped lock, and others. These objects are used as a synchronization mechanism for restricting access to a resource (e.g., a variable or section of code) when multiple threads can access the resource simultaneously. When writing code, it is generally best to avoid simultaneously access when possible so as to reduce the overhead associated with using a lock; that said, a lock is simple to use and the overhead is generally small on modern implementations. The primary downside of using a lock is the possibilities for deadlock (or livelock).

• Mapping: The process of determining which block (nodes) should be execute on which processing device; and, implicitly, connections (edges) onto device buffers.

• Memory Map: Using hardware-based memory management to mirror one section of memory into another section. When any item is changed in either section of memory, the change appears in the other section within typically a few clock cycles.

• Monitor [OS]: Same as an OS condition.

• Mutex [OS]: A type of OS lock, providing multiple threads access to shared resources such that only one thread at a time has access – any others trying to access the resource will be excluded until it is their turn. A mutex must
be, at some level, atomic in locking, such that only one thread is granted access at a time. When the current thread is finished, it unlocks the mutex – which then wakes (at most) one other thread waiting on the lock. A mutex can sometimes be set up as reentrant from within the same thread, such that once the thread has a lock, it can try to re-lock the mutex without negative effects.

- naCL: See Another OpenCL C++ Implementation.
- Node [Graph Theory] : The fundamental unit out of which graphs are formed. A graph consists of one or more nodes, but does not require connections (e.g., a bunch of unconnected nodes can be a graph).
- Node [SDR] : The same as a block.
- Non-Real Time: When the totality of processing in hardware, firmware, and software cannot keep up with the rate of incoming and/or outgoing data; data processing occurring in more time than required to keep up the average sample rate of incoming and/or outgoing data.
- Object [C] : A variable, structure, or function.
- Object [C++] : A ‘C’ object or C++ class, or member variable or function.
- Offline : Not live; not required to be in real time (but, might be).
- On the Fly : Generally the same as live.
- Online : Generally the same as live, but sometimes means in real time.
- OS Abstraction Layer : a software-based framework that provides a common API to the underlying OS, no matter what type it is. This layer needs to provide multiple implementations of a given function, in order to provide the greatest robustness for as many OSs as possible. In general, this layer should check for specific functionality at configuration time (to see if the potential functions are available), build time (to verify that the remaining functions are still available), and runtime (to verify that the remaining functions execute as desired).
- Packet : A chunk of formatted data, generally split into 3 ordered parts – the head (or header), payload, and tail; sometimes the head and tail are combined. The header generally provides information as to the packet
length and offset to the payload, and describing the formatting or data contents. The tail (whether explicitly as such or combined with the head) generally provides a checksum to verify the integrity of the packet. Packets are generally transmitted / processed in an asynchronous or isochronous manner.

- **Platform**: A host computer containing one or more computing devices.
- **Platform [OpenCL]**: The set of host computer’s properties, and computing devices and their properties, that can be utilized by OpenCL.
- **Pointer [C/C++]**: The starting memory address of an object.
- **Port [Graph Theory]**: The interface between a node and connection, sometimes numbered or named in order to allow for counting and/or unique identification.
- **Port [SDR]**: Ports can be named or numbered, depending on the framework. The high-level language style is to name ports and then refer to those names in any graph assignments. It is possible to assigned ports implicitly under some conditions. Block centric programming uses ports to define connections between blocks. Buffer centric programming assigned input ports implicitly, and output ports in some specified numerical order (that is understood by the system).
- **Preemptive Multitasking**: A means of sharing a given set of computational resources between one or more applications, where each application is provided some amount of time to execute before it is forced to give up processing for a while; the application is said to be preempted by the OS.
- **Process [OS]**: The internal representation of an executing application. A process consists of one or more threads, each potentially capable of executing independent of the others.
- **Process [Data-Flow]**: Another name for a block.
- **Program [OpenCL]**: The set of one or more ProgramSources, and Kernels resulting from building those ProgramSources for one or more specific devices.
- **Program [OS]**: The output from compiling a program source; a binary file capable of being executed on the target computing device.
- **Program Source [OS]**: Code to be compiled, generally in text form. The program source generally includes header files, and, for *Surfer*, C++ files.
• ProgramSource [OpenCL] : Code to be compiled as taken from a text string, or one or more pre-compiled binaries. If the latter, each binary is defined for execution on a specific device.

• Pull [Processing] : A data-flow processing technique where data is requested by the sinks(s), and those requests are propagated through the waveform until they reach the source(s). The source(s) then generate data to fulfill the request(s).

• Push [Processing] : A data-flow processing technique where data originates at the source(s), and is propagated (pushed) through the waveform until it reaches the sink(s). Also called data driven processing.

• Real Time : When the totality of processing in hardware, firmware, and software can at least minimally keep up with the rate of incoming and/or outgoing data (and, possibly, be much faster); data processing occurring in less time than required to keep up the average sample rate of incoming and/or outgoing data.

• Reference [C++] : See Section B.5

• Runner :

• Runtime : During execution of a compiled application. As opposed to Configure Time or Compile Time.

• Saline : See Surfer Algebraic Language Interface.

• Salt : See Surfer Abstraction Layer.

• Scheduler : A runtime algorithm, typically implemented as a specific class or independent thread, that performs scheduling.

• Scheduling [SDR]: The process of determining the amount of data to process, for a given block, at a given instant. For systems using a-priori determined scheduling, the amount will in general be static, while for those performing runtime scheduling the amount will generally be different from block to block and change between processing executions.

• Shared Device : A computational resource that is shared between multiple applications, threads, and fibers, typically with an OS to provide for scheduling.

• Shared Memory [Hardware] : Memory that is accessible to multiple processing devices and is thus shared between them.
• Shared Memory [OS] : Memory that resides in the OS kernel, and thus can be shared among multiple threads and processes if set up to do so; sometimes called kernel memory.

• Shared Resource [OS] : An area of memory, variable, or code that is made available to multiple threads and/or processes.

• Static : Not changing over time; generally, pre-determined by some means (e.g., assigned by the user or specified via a configuration file).

• Stream : Generic terms for a data connection between blocks. A stream of data might be parsed as frames or packets, depending on the connected blocks.

• Sub-Graph [Graph Theory] : A subset of the overall graph, where, generally, the blocks are contiguously connected.

• Surfer Abstraction Layer : Surfer’s implementation of combined hardware, OS, and compiler abstraction layers, called Salt. Salt provides a standard API for accessing many system-provided functions, in a object-oriented set of classes where possible and appropriate.

• Surfer Algebraic Language Interface : An algebraic-like C++ script-based interface allowing the user to instantiate blocks and create waveforms using buffer centric techniques.

• Synchronous : Arriving or departing with predictable timing (at least on average).

• Template [C++] : See Section B.1

• Thread [OS] : The internal representation of a part of an executing application that can be preempted such that other threads may execute. Most modern applications have multiple threads, even if not explicitly programmed as such.

• Thread [Data-Flow] : Another name for a block.

• Waveform : The SDR protocol of interest, defined from source(s) to sink(s). Sometimes expanded as waveform graph. An application in SDR terms is one or more waveforms.

• Waveform Graph : Generally the same as a waveform; the graph that describes a particular waveform.
Surfer relies heavily on C++ classes, abstractions, and features for its functionality. SALINE in particular relies on three standard features of C++ that are not available in C or other similar languages: templates, operator overloading, and runtime variable type comparison. These features, and the C++ concepts of namespaces and passing by reference, are covered briefly in the following sections. All of this information is available in any good C++ book (e.g., [132] [69]), and is provided here for completion and easy reference.

B.1 Templates

Templates are part of all recent C++ implementations, and allow the definition of any instance of a class, method, variable, or function concisely, with minimal redundancy. A template function or class is designed to work on many different data types without being rewritten specifically for each one. For example, let us compare the C and C++ implementations for a max function, which takes two same-typed arguments and returns the maximum of them. In C the function max of two same-typed arguments for the types int and float could be written

```c
int max_i (int a, int b)
{
    return (a > b ? a : b);
}
```

```c++```
float max_f (float a, float b)
{ return (a > b ? a : b); }

Note that in C the function names must be unique, as must the input arguments and return type for each function. Clearly, there is significant code redundancy, and to expand these functions to include other types beyond those provided would require more similarly named functions.

C++ provides templates to reduce the code complexity for such functions. The max function taking two same-typed arguments, as above, can be written generically via a C++ template function as

template < typename T >
  T max (T a, T b) { return (a > b ? a : b); }

This template function will be expanded by the compiler into an explicit-typed (non-template) function at compile-time. For example, if the code

float fm = max < float > (1, 2);

is issued, then the compiler will implicitly create the max function for type float.

Templates are a powerful abstraction that can be used to greatly reduce written-code size and increase code-reuse.

That said, the C++ standard in use by most compilers, called C++03 [21], does not allow for a variable number of template arguments (e.g., for a function taking those types as arguments); nor does C++ robustly handle a variable number of arguments to functions or methods. The recently ratified standard, called C++11 [22] does allow for such variadic templates, which in turn will allow for robust handling of a variable number of arguments to functions or methods. Once compilers are updated with this new standard, Surfer will move to using its features for enhanced functionality.
B.2 Operator overloading

In the C language, the math operators +, *, &, <, and % (and a handful of others similar to these) are defined solely for the built-in variable types, e.g., int and long; some are defined for float, but none can be made to work with user-defined classes. In C++, these operators can be overloaded to work with any class type; the function names for those listed above are operator+, operator*, operator&, operator<, and operator%. This type of operator overloading provides a robust abstraction mechanism, allowing end-user programs to hide complexity. For example, suppose we define a template class that stores a type value, as

```cpp
template < typename T > class foo {
public:
    T d_value;
    foo (T value = 0) : d_value (value) {}  
    ~foo () {}  
    T value () { return (d_value); }  
};  
```

Given this class, we want to be able to manipulate the stored value through operator overloading. One could concisely write the code for the + operator for same-typed arguments as

```cpp
template < typename T >
foo < T > operator+ (foo < T >& lhs, foo < T >& rhs) {
    return (foo < T > (lhs.value () + rhs.value ()))
}
```
such that the + operator for identical foo template types returns another foo with the internal value of the sum of the provided arguments internal values. The above code can be used to produce the algebraic-like code segment

```cpp
foo < int > a, b, c;
a = 1;
b = 2;
c = a + b;
```

B.3 type_info and typeid

The typeid facility provided by the type_info class allows for type comparison of almost any two active variables, as well as a means of retrieving the actual variable type as a string. The typeid facility is not limited to built-in C++ types, but is also available for user-created types. Continuing from the previous examples, suppose we wanted to add two potentially different foo class types. Then, using typeid, one way to implement this functionality is

```cpp
template < typename lhs_t, typename rhs_t >
foo < lhs_t > operator+ 
(foo < lhs_t > lhs, foo < rhs_t > rhs)
{
    lhs_t rhs_to_use = 0;
    if (typeid (lhs) == typeid (rhs)) {
        rhs_to_use = rhs.value ();
    } else {
        rhs_to_use = lhs_t (rhs.value ());
    }
}```
return (foo < lhs_t > (lhs.value () + rhs_to_use));
}

Admittedly, for basic types such as int and float, the above function could be written with less complexity because the C++ compiler will do any type conversion implicitly. That said, the above code could also be used with any lhs_t and rhs_t types, so long as the type cast from rhs_t to lhs_t is valid and operator+ is defined for the lhs_t type. The above code can now be used to produce the algebraic-like code segment

foo < int > a, c;
foo < float > b;
a = 1;
b = 2;
c = a + b;

where the addition is between the types int and float, and the result stored as an int. Similarly, we can also overload the operator= method (an in-class function), to allow code such as

foo < int > a;
foo < short > b;
foo < long > c;
a = 1;
b = 2;
c = a + b;
B.4 Namespaces

A namespace [104] is an abstract container that holds a set of identifiers, each of which has a unique name. In C++, an identifier can refer to a variable, function, or method. A namespace can be thought of as defining a scope in which the set of identifiers is valid. Different namespaces may hold same-named identifiers, and it is up to the user (and compiler) to distinguish between, and correctly utilize, them. For example, to define the integer variable foo within the bar namespace, one could use the code segment

```c++
namespace bar {
    int foo;
};
```

This variable is then referred to as bar::foo. Many frameworks define their own namespace(s); a common one is the standard template library provided by most C++ compilers, which resides in the std namespace [19]. Surfer makes extensive use of C++ namespaces: one each for Salt, naCL, Surfer, Spf, Saline, and SQuid.

B.5 Passing by Reference

A reference to a variable, in C++ declared using the character & between the variable type and name, is the equivalent of the actual variable, with the full capabilities provided as per how the actual reference is defined (e.g., a const reference cannot be modified) [121]. C++ introduced the idea of passing by reference as an alternative to passing by pointer, though with the actual generated code being roughly the same. The use of references provides the compiler to
greater knowledge of the intended use of any referenced variables, and provides the programmer with a guarantee that those variables are valid. When calling a function using a pointer, the programmer should always first verify that the pointer was “valid” (generally not null) before dereferencing it. Passing by reference, when using classes or structures and comparing with passing by variable (not pointer), has the advantage that only the reference (roughly a guaranteed pointer to the variable) itself is passed, not a copy of the entire class or structure.

Consider the following code snippet, which shows both passing by pointer and reference in order to compare and contrast the two methods.

```cpp
class foo {
public:
    foo () {};
}
void do_it (foo* f_p, foo& f_r, foo f_v);
```

The first variable to the function `do_it`, `f_p`, is a pointer to a `foo` class. This variable, as a pointer, can be any value within the size of pointers for the system on which this code is compiled – typically 32 or 64 bits. The function should generally be written to try to verify that this pointer is valid before using it, though in certain specific usages it will be known to be valid. The second and third variables must refer to valid `foo` class instantiations, otherwise the compiler will produce an error and no executable code will be generated. The use of the reference (`&`) differs from the pointer in that the variable `f_r` is guaranteed to be a valid instantiation of the `foo` class, and hence does not need to be verified. Passing by reference, when compared with passing by variable (the third argument), has the advantage that a copy of the instantiated class is not required.
APPENDIX C

BLOCKS AND FLAVORS

We have created Surfer blocks for both testing and real-time processing, centered around those required for an FM demodulator and an OFDM modulator. This appendix contains a list of blocks currently created, along with their flavors.

- **adder.2**: Adds two streams together. The two inputs and single output can each be a different type. Flavors are apple+accelerate (float and double only), generic, and naCL+generic.

- **audio_source**: Provides audio from one of the host computer’s sources at the requested rate, if available. Flavors are apple+audio, jack, and portaudio.

- **constant_source**: Generates a constant value of an arbitrary type. Flavors are apple+accelerate (float and double only), generic, and naCL+generic.

- **cyclic_prefix**: Takes a vector of inputs of a set length, and generates a longer vector made up of a set number of the final vector elements and the vector itself. The cycle and prefix length can be specified at block creation, as well as via block methods. This block supports runtime parameter-based reconfiguration. Flavors are apple+accelerate (float and double only), generic, and naCL+generic.

- **demux**: Parse a single stream’s incoming items into $N$ output streams, where $N$ is determined by the physical topology of the block when a new output connection is made or removed. The parsing can be forward – in increasing output stream order – or reverse – in decreasing output stream order. Flavors are generic and naCL+generic.

- **downsample**: Reduce the sample rate of the input stream by an integer number. Flavors are apple+accelerate (float and double only), generic, naCL+generic, and subgraph+polyphase.
• fft\_forward: Perform a forward FFT on vectors of inputs of a given length, set at construction time as well as changeable during runtime. Currently only real to complex or complex to complex. This block supports runtime parameter-based reconfiguration. Flavors are apple+accelerate (float and double only), fftw3 (float, double, and long double), generic, naCL+generic, and naCL+oclia.

• fft\_inverse: Perform an inverse FFT on vectors of inputs of a given length, set at construction time as well as changeable during runtime. Currently only complex to complex. This block supports runtime parameter-based reconfiguration. Flavors are apple+accelerate (float and double only), fftw3 (float, double, and long double), generic, naCL+generic, and naCL+oclia.

• fftshift: Perform a “frequency shift” on vectors of inputs of a given length, set at construction time as well as changeable during runtime, on items of any type. The shift swaps the lower and upper halves of the vector. This block supports runtime parameter-based reconfiguration. Flavors are apple+accelerate (float and double only), generic, and naCL+generic.

• file\_sink: A sink where items of an arbitrary type are written to one or more files. The base file name and length are supplied on construction, but can be modified during runtime. The flavor is unix+FILE.

• file\_source: A source where items of an arbitrary type are read from a file. The file name and length are supplied on construction, but can be modified during runtime. The flavor is unix+FILE.

• fir\_filter: Perform FIR filtering on the input stream, given the filter taps, with arbitrary input, output, and tap types. Taps can be set at construction as well as during runtime. This block supports runtime parameter-based reconfiguration. Flavors are apple+accelerate (float and double only), generic, naCL+generic, and subgraph+polyphase.

• gain: Scale the input stream by a constant, but not static value, using arbitrary input, gain, and output types. The value can be set at construction as well as during runtime. This block supports runtime parameter-based reconfiguration. Flavors are apple+accelerate (float and double only), generic, and naCL+generic.

• gr::ofdm\_mapper: Performs mapping from bits to constellation from the payload of inline Spf packets. The constellation item type is arbitrary. The mapper conforms to the functionality of the GNU Radio OFDM Mapper block, and thus has just a single flavor.
• gr::ofdm_modulator : A subgraph creating a full GNU Radio style OFDM Modulator, and thus has just a single flavor. The subgraph performs mapping -> insert preamble -> fftshift -> inverse fft -> gain -> add cyclic prefix.

• gr::ofdm_preamble_inserter : Inserts a preamble of arbitrary type when triggered to do so; the preamble is specified at construction and is static. This block conforms to the functionality of the GNU Radio OFDM Preamble Inserter block, and thus has just a single flavor.

• gr::packet_decoder : Tries to decode inline packets, into items of arbitrary type. This block conforms to the GNU Radio unmake packet utility, and thus has just a single flavor.

• gr::packet_encoder : Encode input items of arbitrary type into inline packets. This block conforms to the GNU Radio make packet utility, and thus has just a single flavor.

• mux : Interleave $N$ incoming streams’ items into one output stream, where $N$ is determined by the physical topology of the block when a new input connection is made or removed. The parsing can be forward – in increasing input stream order – or reverse – in decreasing input stream order. Flavors are generic and naCL+generic.

• packet_decoder : Tries to decode inline packets, into items of arbitrary type. This block conforms to Surfer packet format, and has just a single flavor.

• packet_encoder : Encode input items of arbitrary type into inline packets. This block conforms to Surfer packet format, and has just a single flavor.

• packet_source : Takes packets into a queue, and uses the queued packets as outputs. This block conforms to Surfer packet format, and has just a single flavor.

• quad_demod : Performs quadrature demodulation on input items, using arbitrary input, computation, and output types. Flavors are apple+accelerate (float and double only), generic, and naCL+generic.

• rate_limit : Limits the rate in items per second at which a single stream of arbitrary type passes data passes through this block.

• sinusoid_source : Generates a sinusoid of an arbitrary type, at a given sample rate, frequency, initial offset, and rotation direction. All parameters can be set during runtime. Flavors are apple+accelerate (float and double only), generic, and naCL+generic.
• sum : Synchronously adds $N$ input streams' items together, using the same arbitrary type for all inputs and outputs. $N$ is determined by the physical topology of the block when a new input connection is made or removed. Flavors are apple+accelerate (float and double only), generic, and naCL+generic.

• type_converter : Converts the items from a single arbitrary typed input stream into an another arbitrary type. There is just one flavor.

• uhd_rx : Provides an Ettus Research UHD stream receiver of any type allowed by UHD, with public methods for setting and retrieving all of the properties of a UHD receiver. There is just one flavor.

• uhd_tx : Provides an Ettus Research UHD stream transmitter of any type allowed by UHD, with public methods for setting and retrieving all of the properties of a UHD transmitter. There is just one flavor.

• vector_sink : Retrieves a vector's worth of arbitrary typed items from the input stream, issues a callback if provided, and then either blocks waiting for reset or throws away further incoming data. There is just one flavor. This block is primarily used for quality assurance testing.

• vector_source : Injects a vector's worth of arbitrary typed items to the output stream, repeating any number of times or forever. The vector of items can be replaced at any time, which will reset this block. There is just one flavor.
BIBLIOGRAPHY


Style for Numerical Software and its Optimization,” Scientific Programming,

38. J. R. Einstein, C. W. Glover, and M. L. Dickens, “A Tool to Aid in Mapping
Computational Tasks to a Hypercube,” in The Proceedings of the Fourth
581.

39. J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer,
S. Sachs, and Y. Xiong, “Taming Heterogeneity – The Ptolemy Approach,”

40. Electronic Privacy Information Center, “Comments in the Matter of Digital
Broadcast Copy Protection,” Federal Communications Commission, Wash-
ington, DC, USA; MB Docket No. 02-230, December 6 2002.


42. R. M. Firestone, “Parallel Programming: Operational Model and Detection
of Parallelism,” Ph.D. dissertation, New York University, New York City,
NY, USA, May 1971.


45. B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, Heterogeneous
Computing with OpenCL. Morgan Kaufmann, August 2011.


48. GNU Radio Discussion Email List, 15 May 2009, “Native Interface
between MATLAB / Simulink and USRP Available,” accessed April 2012.
2009-05/msg00306.html


