SEAMLESS DYNAMIC RUNTIME RECONFIGURATION IN A SOFTWARE-DEFINED RADIO

Michael Dickens (Consultant to RFware, LLC, South Bend, IN : mlk@rfware.com;
Graduate Student, University of Notre Dame, IN : mdickens@nd.edu)
J. Nicholas Laneman (Professor, University of Notre Dame, IN : jnl@nd.edu)
Brian P Dunn (RFware, LLC, South Bend, IN : brian@rfware.com)

ABSTRACT

We discuss implementation aspects of a software-defined radio system that allows for dynamic waveform reconfiguration during runtime without interrupting data-flow processing. Traditional software-defined radio systems execute a waveform statically, exactly as it is programmed. Reconfiguration is provided by executing a different waveform, which requires the system to stop processing data while reconfiguration occurs, and also may incur an unacceptable delay for some applications. Recent research has demonstrated basic reconfiguration by programming multiple branches into a waveform and dynamically switching between branches. This technique requires redundant resources and in general cannot be expanded to encompass all possible waveforms of interest, but, if implemented carefully, could be made to seamlessly process data. We propose a system that allows for dynamic insertion and removal of entire waveforms, individual constituent blocks, and block algorithm implementations tailored to specific processors. Our system performs this reconfiguration while maintaining processing state, seamlessly without interrupting data-processing, and with only the resources necessary for the given waveform and processors. In order to leverage this new level of reconfigurability, we created a new system component: a supervisor. This system supervisor monitors the state of each processor and waveform execution, and moves computations among available processors as their loads, capabilities, and block algorithm implementations allow. An example using a simple supervisor is provided to demonstrate the effectiveness of our system.

1. INTRODUCTION AND MOTIVATION

As software-defined radio (SDR) becomes more main-stream, devices using SDR will become more sophisticated. Already, such devices are moving from bulky handhelds with specialized processors and programming, to ones small enough to fit into a pocket – while using reprogrammable software executing on multi-core general-purpose processors. In the not-so-distant future, devices will likely be using many-core processors and advanced graphics processing units (GPUs), with the ability to do real-time SDR for complex waveforms.

Device functionality is moving from a few static waveforms, to smartphone capabilities including web-browsing, augmented reality, and communications including voice, video, and data – possibly all at the same time. These devices will be monitoring the whitespace and other devices' communications, and cognitively altering their own communications to both use available bandwidth as well as to avoid bandwidth in use by others [1-3]. Such systems cannot rely on a few static waveforms; they must instead provide dynamic reconfiguration of waveforms during runtime in order to maximize both device utility and battery life. Further, some high priority functions will require high quality of service communications capabilities. Given these requirements, there will be a need to move computations between processors on such devices, without impacting data reception or transmission – i.e., providing seamless runtime data processing and waveform reconfiguration.

To demonstrate the practicality of such processing in a SDR, we have taken *Surfer* [4], our SDR framework – the collection of executables and libraries, header, resource, and data files for a given project – and augmented it in such a way that it can support both "all-in-one" processing blocks and a new block abstraction allowing for seamless processing. As part of the changes, we developed a new component – a supervisor – that keeps track of the load on the device's processors as well as various SDR waveform execution parameters, and can modify waveform execution to meet user-specified requirements. We discuss these changes in Section 3, after providing relevant background information in Section 2 on how SDR processing works in a general sense. In Section 4, we describe a simple application and show the effectiveness of our technique through a simple load threshold detection supervisor. Conclusions and acknowledgements are then provided in Sections 5 and 6, respectively.

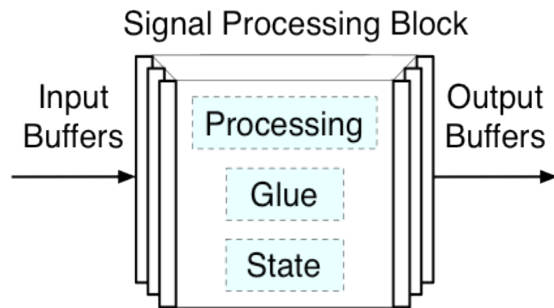

Figure 1 – General diagram of a SDR signal-processing block

2. BACKGROUND

In this section, we discuss how current SDR frameworks perform data processing, in as broad terms as possible. Our goal is to provide enough relevant information such the changes described in Section 3 can be compared with the current methods; we are not trying to fully describe how GNU Radio [5] or SCA [6] does its processing, but rather to look at the way processing takes place in a general sense.

2.1. Waveform as Graph, with Block Details

Each SDR waveform can be described by an acyclic graph, whether performing packet or frame processing of data. Such a processing abstraction allows for a graphical interface (GUI) to describe a given waveform – e.g., the GNU Radio Companion [7], MathWorks' Simulink [8], or National Instruments' LabVIEW [9]. Such high-level GUIs are excellent for visualization purposes, and for users who are not interested in the underlying implementation details. GUI representations of SDR waveforms hide implementation details from the user, including how data is buffered between signal-processing blocks, the state of each block, and where computations are actually performed. Sometimes it is useful to delve into the inner workings of an SDR framework to better understand its functionality and to experiment with modifications that might offer more robust performance.

Figure 1 shows a generic signal-processing block, including input and output buffers, the actual processing algorithm implementation, the block state, and the programming glue that holds the parts together. Some blocks will be input only (e.g., sinks, consumers), while others are output only (e.g., sources, producers); some do not need state (e.g., synchronous 2-stream adder) but most do (e.g., a FIR filter requires the N filter coefficients and the last N-1 input samples, and possibly other variables depending on the actual implementation). The size and number of each block's buffers can be related to those blocks preceding and following it; each buffer holds items of some specific type, entering and exiting at related, possibly identical, sample rates.

Each block is generally coded as a group of related variables and functions using those variables (e.g., a C++ class enclosing variables for state, and methods for handling processing and determining other relevant properties of the specific block). In some SDR frameworks, the block itself determines when it is ready to be scheduled to do processing – when there is enough input data and output buffer space, among the basic requirements – while in others it is handled by some external process. Some SDR frameworks evaluate the waveform as a whole a-priori to determine block scheduling timing and buffer sizes.

In the block configuration from Figure 1, all forms of dynamic runtime configuration require the equivalent of a switch, such as that in Figure 2, to handle selection of the block or waveform. Seamless data processing can be provided by switching between anywhere from individual blocks to whole waveforms. Note that the individual or group block state must be kept in sync between all blocks using the switch, or must be copied between blocks at switch time. Neither method makes efficient use of memory resources and both add extra complexity to the waveform graph – whether in GUI or script form. Although the former method could be practically implemented for any specific block, there is a more efficient abstraction that we use to provide seamless data processing that will be discussed in Section 3.

2.2. Data Processing and Reconfiguration

In SDR frameworks, data processing occurs when a pre-specified C++ method or C-style function is called (e.g., in GNU Radio the "general_work" method). This method resides within the signal processing block class or is assigned statically, such that using some other instantiation of the same algorithm requires creating a new block and re-connecting the graph (e.g., via the switch from Figure 2). In current SDR frameworks, dynamic reconfiguration takes place via the switch, or by stopping graph execution (whether telling the blocks to stop or by stopping the external controller), replacing the block of interest, and then restarting the graph. Although for some applications the latter reconfiguration style can be made to work robustly, it is not, in general, seamless with respect to data processing continuity and cannot be applied to real-time signal processing in a completely general sense. Making use of a switch can allow real-time signal processing, but does not use resources efficiently, adds complexity to the waveform script or GUI, and can work only with those blocks within the switch – adding in new versions of the same block requires modifying the GUI or script. Again, in Section 3 we provide a more robust abstraction that not only preserves the GUI or script, but also allows for seamless reconfiguration during real-time signal processing while providing more efficient use of resources.

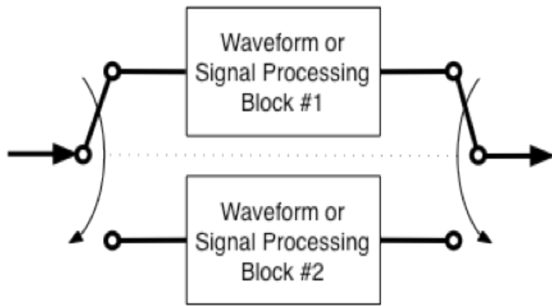

Figure 2 – Reconfiguration via synchronized switches, with each block providing alternative implementations of a given algorithm

2.3. *Surfer* Basics

We developed *Surfer* with a number of goals in mind; one in particular is to remove unnecessary complexity from the end-user's experience, while maintaining high functionality and user-selectable flexibility in processing. Instead of using a single thread for all processing, or a thread per block, *Surfer* takes a middle-ground approach by queuing blocks for processing in *block runners* – with one block runner per thread. A key feature of *Surfer* is the use of thresholds on both input and output buffers that determine when the block should be processed; using thresholds results in constant overhead time per block, with that overhead processing spread across all active runners. Each *Surfer* block can have specific affinity for a given runner or ordered list of runners, choose the runner with the lowest load, or just use the first available. Block runners can provide functionality on a variety of processors, from the local CPU to an attached DSP and GPU (e.g., via OpenCL [10], NVIDIA CUDA [11], or AMD ATI Stream [12]).

3. CHANGES FOR SEAMLESS PROCESSING

We augmented *Surfer* to allow it to handle data processing seamlessly during runtime, keeping in mind an overarching goal of *Surfer* development: abstracting complexity away from the user. This section describes the concepts we implemented in this augmentation, including the splitting off of the processing from the signal processing block, the need for a new state construct that allows state memory to be shared across networks and between physical processors, and a new system monitor that allows for automated control of the augmented system. First we provide a brief synopsis of our use of OpenCL for accessing a GPU for signal-processing purposes.

3.1. OpenCL

OpenCL, the "Open Computing Language", is an open standard for implementing general-purpose computations on heterogeneous computing devices. We chose to use it because it provides better cross-platform compatibility than NVIDIA CUDA or ATI Stream alone, while still providing high functionality. Optimized signal-processing capabilities using CUDA or Stream could be created as alternatives to those provided by *Surfer* in OpenCL.

OpenCL performs computations via commands placed into a queue that is owned by a context containing one or more processing devices. Each queue supports commands for transferring data to and from any device within its context, as well as commands to execute a kernel – a program compiled specifically for one or more devices. Data transfer can be accomplished directly (e.g., similar to the UNIX C functions 'bcopy' or 'memcpy'), or via a memory map. Data buffers can be allocated on the host or OpenCL device, and data easily transferred between them.

Most queue commands return an event that can be used as a dependency for other commands – for example, that a data transfer must occur before the kernel using that data is executed. Most commands can also take a list of events (e.g., as returned from other queued commands) that must finish before the command is executed. Command queuing can be done asynchronously, and in this way OpenCL allows for multiple commands to be queued in rapid succession so long as their event dependencies are correctly specified.

OpenCL signal-processing in a SDR for a given block follows the following chain of events:

- init OpenCL constructs;
- compile kernel from program;
- execute the kernel:
  1. data transfer(s) from host to OpenCL device;
  2. kernel execution
  3. data transfer(s) from OpenCL device to host.

The above chain of events can be separated into 3 distinct parts: initialization, kernel creation, and task execution. OpenCL constructs in this case are the context and queue, both of which might be used for multiple blocks. Hence we created an OpenCL-based block runner that contains these constructs. A OpenCL runner must be paired with any block executing OpenCL signal-processing.

Because the kernel can depend on runtime parameters, kernel compilation cannot take place until the waveform graph is fully defined. Once the kernel is compiled, assuming that the graph remains unchanged then this kernel does not need to be compiled again.

Once the OpenCL constructs and kernel are created, the actual steps to execute the kernel form a repeatable task. Hence we created a class for the specific purpose of issuing such repetitive tasks. This class allows the user to easily define event dependencies and all other relevant parameters. Once set up, task execution is entirely encompassed within the class, and the kernel execution is reduced to calling the task's "execute" method (with no arguments).

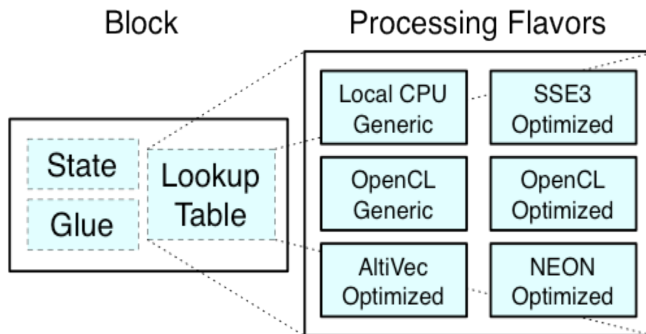

Figure 3 -- Split block with separate selectable flavors

OpenCL uses a runtime compiler that takes a string argument containing the program to be compiled; at least in theory a single program source can be used for any device that adheres to the OpenCL standard. Because OpenCL programs are strings, they can be manipulated as needed for specific needs during runtime (e.g., for an N:1 multiplexer block, setting the value of N without having to pass it in as an argument). We make use of this runtime compilation feature by setting program items such as data types and the number of input and output streams.

3.2. Computation Flavors

As implied when referring to Figure 2, some sort of switch is required to select the waveform, block, or computation being performed. Instead of switching between waveforms or blocks, we moved the switch inside the block itself, and separated the signal processing algorithm implementation into its own class. In place of the signal processing, we added a lookup-table containing instantiated signal processing algorithm implementations that all provide the same application-programming interface (API) – we call these *flavors*. This new block construct is shown in Figure 3 with six possible flavors. Each flavor is given a string name (e.g., "generic") and the list of names is made available external to the block so that any name in the list can be selected as the flavor to handle processing. The lookup table stores this selection as a pointer to the instantiated flavor, such that accessing it does not require searching through the table.

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. Flavors are classes specifically designed for processing, and not allowed to store state or any other local data – those must remain in the state as found in the block that owns the flavors. Note that the actual state and buffer data need not reside on the local CPU's memory, but rather can reside entirely on remote devices. 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 and nothing more; 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 – and an OpenCL implementation for blocks that can be efficiently programming in that language. The user 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., OpenCL flavors can only be executed within an OpenCL block runner, since they require different handling than a flavor executing on the local CPU. In this way, *Surfer* allows for either runtime or a-priori block scheduling.

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 powers-of-2 lengths. In this case, even when all other checks pass, if the user specified a non-power-of-2 length FFT then this flavor will fail instantiation and hence will not be available for this block's flavor table. As flavor usability is determined on a block-by-block basis, it is possible for different block instantiations of the same block type to have different flavors available for them to use during runtime.

As a more extensive example relevant to this work, during its configure stage of building *Surfer* tests for the OpenCL library and primary header. If the library is found, the configure script tries to link against it to make sure that the library is readable and usable by the user. If the header is found, the configure script tries to use it to determine the version of OpenCL. Assuming all tests pass, then macros will be created that define OpenCL as being available for use when compiling, as well as the version. OpenCL currently comes in version 1.0 or 1.1, with the latter being a superset of the former. Thus, during compilation, the OpenCL version macro is used to determine which OpenCL features to utilize. Assuming compilation succeeds, then when a block that includes an OpenCL flavor is instantiated

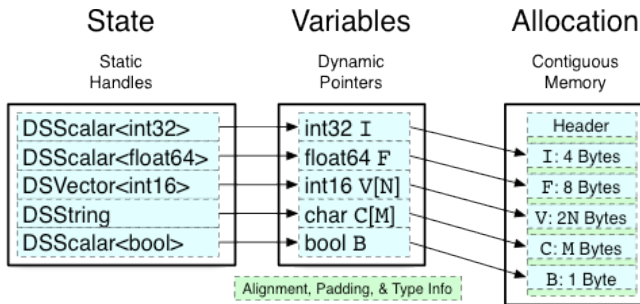

Figure 4 – Conceptual example of a block state using dynamic structure variables

by the user's application, before the flavor is added to its block's lookup table it will be initialized to make sure the necessary OpenCL function calls succeed – and if all three steps are successful then the OpenCL flavor becomes available for processing data.

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 does occur is when a new flavor must be initialized; this task can be done in the same thread as that handling processing, or pushed off to a separate thread and done asynchronously to processing. 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.

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 order to function correctly, many blocks 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" [13] 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.

3.2. Dynamic Structure Variables

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 state construct was put in place 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 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*. An example of a block state using this construct is provided in Figure 4, 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, but are not further described here. 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. The user accesses each variable in C++ through its dynamic structure counterpart – internally via doubly-dereferencing the handle – and does not in general have access to 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.

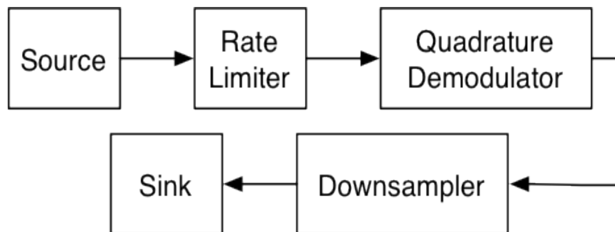

Figure 5 – Simple example application to demonstrate seamless runtime processing and reconfiguration

3.3. Supervisor

In order to leverage the dynamic functionality provided by flavors in an automated manner, information must be collected on both the flavor functionality (e.g., throughput, latency, energy use, overhead time) and system state (e.g., CPU load, network utilization). *Surfer* already provides the basic capabilities for collecting this data; we added modules for collecting the CPU use for a specific process down to individual threads in the process. We also introduced a system *supervisor* as the focal point for collecting and utilizing this data. *Surfer* creates a default supervisor at boot time that collects no data but also does not modify waveform execution. The user can overload this default with a different supervisor – e.g., one that monitors CPU load and then can changes the flavor of certain blocks based on user preferences. In creating a new supervisor, the user can select from *Surfer*-provided functionality-monitor modules, or use ones created outside *Surfer*.

Although a supervisor does not directly schedule blocks for processing, it can indirectly influence this processing through the changing of input and output buffer threshold values. More importantly, the supervisor can switch flavors for any block where there are multiple flavors available, and where the user has specified affinity to multiple flavors (e.g., choose a flavor whose processor has a lower load than the current flavor's, or to move back to a flavor with higher affinity once its processor load is below a given threshold).

4. APPLICATION EXAMPLE

Figure 5 shows a simple application graph that demonstrates that even a basic supervisor monitoring the CPU load can provide good performance and seamless processing using the new *Surfer* flavors. Approximately 9 seconds worth of narrowband FM (NBFM) data was taken using an Ettus USRP1 [14] 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 (I&Q) integer with 2 bytes per value or 4 bytes / sample, requiring 1 M-byte/s throughput for real-time processing. This file is then used as the data source for the example, where the FM signal is 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. The resulting audio signal was then stored back to another file. For downsampling when using the host CPU, we intentionally used a non-optimized FIR filter that requires more CPU utilization than an optimal one (i.e., rather than using SIMD specialized instruction sets such as SSE, AltiVec, or NEON). Both the quadrature demodulator and downsampler have flavors allowing execution on the local CPU as well as the host-computer's GPU (via OpenCL), and their *Surfer* blocks were configured to prefer using the local CPU to OpenCL. A supervisor was monitoring the CPU load of the host computer, and was programmed with a threshold of 60% max CPU load before switching flavors.

With a graphical CPU load display running, we started *Surfer* executing the graph in Figure 5, and then separately started an external process that fully loaded the host CPUs for a short duration. As shown in Figure 6, 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* continues 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 OpenCL to the local CPU. 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. This example demonstrates that the technique we developed for allowing runtime dynamic reconfiguration can successfully process data seamlessly.

5. CONCLUSIONS

We have developed an SDR framework with the capability of performing seamless dynamic runtime reconfiguration. We accomplished this task by taking the standard signal-processing block programming, and separating the actual processing into its own class. In the place of the processing functionality, a lookup table is used to store the possible processing flavors. Each flavor for a given block meets the same API requirements, such that given a specific state and inputs, each will generate the same outputs (within machine tolerance). The flavor abstraction allows for the ability of

SDR-based devices to seamlessly switch processors where the actual signal processing takes place.

6. ACKNOWLEDGEMENTS

This work has been supported by RFware, LLC, NIJ Grant 2006-IJ-CX-K034, and an NVIDIA Professor Partnership.

7. REFERENCES

[1] J. M. Chapin and W. H. Lehr, "Cognitive Radios for Dynamic Spectrum Access – The Path to Market Success for Dynamic Spectrum Access Technology", IEEE Communications Magazine, vol. 45, no. 5, pp. 96–103, May 2007.

[2] S. Haykin, "Cognitive Radio: Brain-Empowered Wireless Communications", IEEE Journal on Selected Areas in Communications, vol. 23, no. 2, pp. 201–220, February 2005.

[3] Z. Sun, G.J. Bradford, and J.N. Laneman, "Sequence Detection Algorithms for Dynamic Spectrum Access Networks", in *Proc. IEEE Int. Dynamic Spectrum Access Networks (DySPAN) Symp.*, Singapore, April 2010.

[4] M.L. Dickens, B.P. Dunn, and J.N. Laneman, "Thresholding for Optimal Data Processing in a Software Defined Radio Kernel", in *Proc. of the Karlsruhe Workshop on Software Radios (WSR)*, Karlsruhe, Germany, March 2010.

[5] GNU Radio Website, accessed September 2011: http://gnuradio.org/

[6] Software Communications Architecture Website, accessed September 2011: http://sca.jpeojtrs.mil/

[7] GNU Radio Companion Website, accessed September 2011: http://www.joshknows.com/grc

[8] The MathWorks, Simulink Website, accessed September 2011: http://www.mathworks.com/products/simulink

[9] National Instruments Corporation, LabVIEW Website, accessed September 2011: http://www.ni.com/labview

[10] The Khronos Group, OpenCL Website, accessed September 2011: http://www.khronos.org/opencl

[11] NVIDIA, CUDA Website, accessed September 2011: http://www.nvidia.com/object/cuda_home.html

[12] Advanced Micro Devices, ATI Stream Website, accessed September 2011: http://www.amd.com/stream

[13] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3", *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216-231, Feb. 2005.

[14] Ettus Research Products Website, accessed September 2011: http://www.ettus.com/products