

Toward Composable Hardware Agnostic Communications Blocks - Lessons Learned

Bow-Nan Cheng
MIT Lincoln Laboratory
bcheng@ll.mit.edu

Shari Mann
L3 Communications
shari.l.mann@L-3com.com

Tim Arganbright
Rockwell Collins
timothy.arganbright@rockwellcollins.com

Abstract—In recent years, there has been a large push in the U.S. Department of Defense (DoD) to more rapidly respond and adapt to changing technology advancements and emerging threats. Nowhere is an accelerated pace of innovation more needed than in the airborne tactical domain where the predominant communication system, Link 16, has been in use for over 40 years. Airborne tactical systems are often stove-piped, highly integrated, and designed to be hardware-specific, making insertion of new technology at the component level extremely challenging. To resolve these limitations, DARPA has been exploring an architecture which separates the transceiver from the waveform processors from the application processors [1]. Each of these processors can be programmed with composable waveform blocks that can be chained together dynamically to leverage the available apertures to perform different missions from communications, SIGINT, electronic warfare, and others. As part of the effort, many of the current generation of airborne tactical waveforms were ported to the new architecture. In this paper, we describe some lessons learned in developing hardware-agnostic, composable, software communications blocks as a proof of concept of the modular new architecture.

I. INTRODUCTION

In recent years, the commercial world has developed highly capable general purpose processors that enable increasing functionality to be implemented very rapidly in software. Many capabilities that used to require hardware implementation can now be developed in software. By enabling more functionality in software, system designs are more easily modifiable and extensible. This paradigm change has been particularly seen in the cellular industry where smart phones have transformed devices initially intended to make and receive phone calls, into a general compute platform, capable of acting as global position system (GPS) devices, sensors, and video cameras. In the cellular world, the combination of the general purpose processor technology and rich development environments fueled an explosion of user-built applications tailored to consumer needs, enabling individuals to creatively leverage components on smart phones for use in applications.

In contrast, many communication systems in the U.S. Department of Defense (DoD), and in particular, airborne tactical communication systems like Link 16 [2], were developed as stove-piped, highly integrated systems that provide a single, clearly defined capability [3]. In fact, the design of many airborne communication systems is tightly integrated with the on-board aircraft applications, functions, and hardware, making technology insertion difficult. Modifications to the radio

system require whole-scale hardware-specific changes from the application layer down to the physical layer. This tight integration of the application, networking, medium access, and physical layers of the waveform has limited our ability to take advantage of technology advances in signal processing, rate adaptation, channel access, and networking to evolve our airborne communications capability.

Even with newer software-defined radio (SDR) technology, this problem persists. SDR moves traditionally hardware-centric components to software-based implementations that are reconfigurable. Although SDR technology has helped to ease the upgrade path of DoD radio systems, it still tightly couples systems and functions - the antennas, power amplifiers, transceivers, modem hardware, and networking software are all tied to a single waveform. Even though the platform is “software reconfigurable”, implementations are tied to the specific hardware and independent evolution of the waveform in a modular fashion is not yet practical. As shown in Figure 1, the SDR architecture slices the system horizontally, limiting cross-waveform, cross-transceiver, cross-antenna sharing.

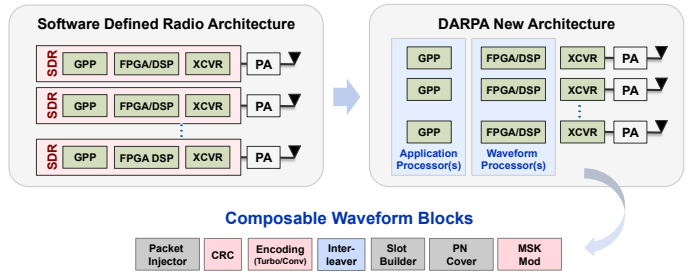


Fig. 1: Software Defined Radio (SDR) vs. New Architecture

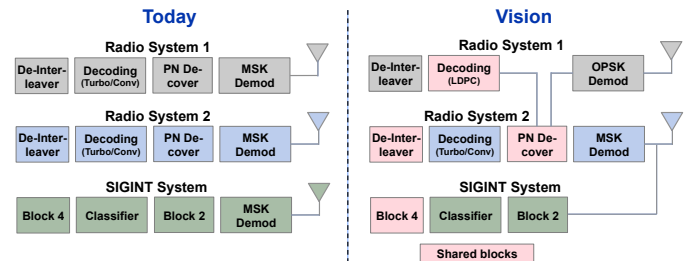


Fig. 2: Dynamically composable waveform blocks enable novel applications

To resolve these limitations, DARPA has been exploring

an architecture which separates radio functionality into three technology components: transceiver elements, waveform processing functions, and application processing functions as shown in Figure 1 [1]. Transceiver functions are functions associated with transmitting and receiving the signal (including A/D and D/A conversion) from the platform antennas. Waveform processing functions are generally real-time functions associated with generating and receiving the signal in space. The applications processing function includes non-real-time functionality such as networking and applications. As shown in Figure 2, the goal is that each of the processing functions can be programmed with composable waveform blocks that can be grouped together dynamically to leverage the available apertures to perform different missions from communications, SIGINT, electronic warfare, and others. To understand the feasibility of this architecture several current generation airborne tactical waveforms were ported to the new architecture.

In this paper, we present lessons learned in developing hardware-agnostic, composable, software communications blocks as proof of concept of the new modular architecture. Although the architecture supports composable processing blocks that can be FPGA, ASIC, or software components, the focus of this work is on the software components. The rest of the paper is organized as follows: Section II presents an overview of AMPLib, the hardware agnostic shim developed to enable the architecture while Section III overviews the use case and implementation. Section IV discusses several key lessons learned and implications of the approach. Finally, Section V concludes the paper with future work.

II. AMPLIB OVERVIEW

One of the key challenges of evolving airborne tactical communications systems is that many of the implementations are tightly coupled to the hardware and leverage hardware-specific elements of processors or resource. While optimizing for hardware enables developers to get the most performance out of the systems, processor advancement often outpaces the time it takes to fund and port software to new processors. There has been significant work in the recent decades in abstracting hardware specifics from applications developers using stack virtual machines [4], process virtual machines [5], and Software Communication Architecture (SCA) [6]. However, almost all the work has focused on abstracting hardware functions on a homogeneous set of processors (i.e. multi-core x86, etc.). There has been recent work in examining virtualizing a grouping of heterogeneous processors [7]. These techniques tend to focus on graphics processor unit (GPU) usage which has seen limited usage to date on communications systems. To enable heterogeneous processing on GPPs and DSPs, a hardware agnostic shim called Asymmetric Multiprocessing Library (AMPLib) was developed.

AMPLib is a shared library abstraction layer that hides underlying processor and operating system details and provides a common interface for scheduling, memory management, and inter-process communications (IPC). It has been ported to support Linux and Windows on x86, TI C66 DSPs, PowerPC,

Freescale/NXP StarCore SC3900FP DSPs, and ARM processors. Waveform processing blocks are “wrapped” with AMPLib, which handles the processor/operating system specifics. AMPLib provides several beneficial features including:

- An architecture-agnostic IPC mechanism that hides underlying IPC details from waveform implementers
- Hardware agnostic processing through a common threading, scheduling, IPC, and memory management approach
- Hardware-specific optimization abstraction
- Flow-based block composition - Each block may receive multiple inputs and generate multiple outputs to different blocks enabling flow-based usage

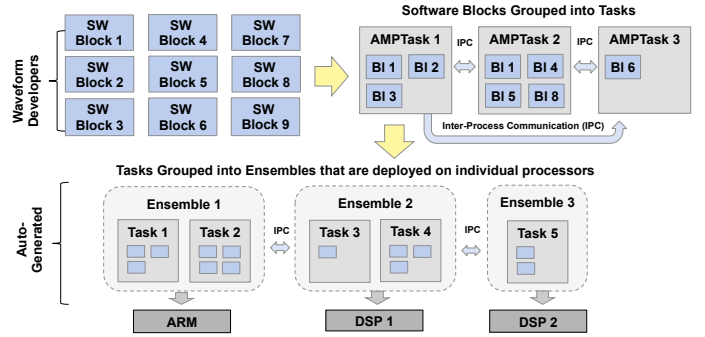


Fig. 3: Waveform processing blocks are grouped into AMP-Tasks and Ensembles for deployment to hardware

Figure 3 illustrates the general approach with wrapping waveform processing blocks with AMPLib. Waveform developers implement processing blocks with generic processing elements such as FIR Filters, modulators, etc. These generic processing blocks are grouped according to function into AMPTasks which constitute a single thread of operation. AMPTasks are then grouped into Ensembles either manually or through an automated process that understands the hardware and block constraints, which are deployed to a specific processor. Details of the AMPLib implementation and architecture are left to other work.

III. USE CASE

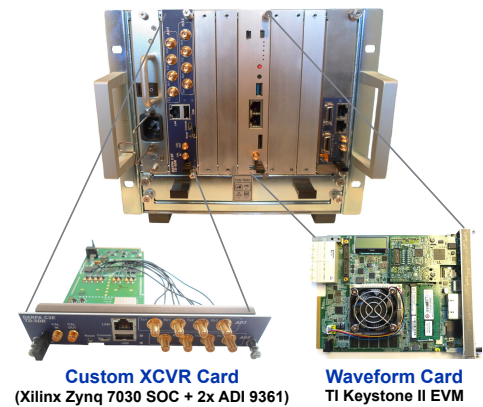


Fig. 4: Test hardware included a TI Keystone SOC, Xilinx Zynq SOC, and other processors

To evaluate the proposed architecture and the software-heavy approach, several DoD tactical waveforms were ported to the architecture and processing blocks wrapped in AMPLib. Processing blocks were written in a generic manner to enable sharing and each waveform was implemented such that atleast 50% of the blocks were shared by others. The implementation was evaluated on generic x86 systems, ARM processors, and TI C66 DSPs without any modifications to the waveform code. Figure 4 depicts the hardware leveraged for the effort. All the waveform PHY-layer processing was performed on the TI Keystone II system on a chip (SoC) [8]. The TI SoC comprises a quad core ARM A15 1.4 Ghz processor and 8 C66x DSPs at 1.2 Ghz. Combined, the SoC provides approximately 173.2 GOPS of processing power. After PHY-layer processing, I/Q streams were sent via a Serial Rapid IO (SRIO) [9] backplane to a custom transceiver card that comprised a Xilinx Zynq SoC [10] and two Analog Devices AD9361 RFIC [11].

Although the full PHY-layer portions of airborne tactical waveforms were implemented in software wrapped in AMPLib and run on the TI systems, the performance was significantly lower than an equal implementation in FPGAs. Performance of various systems varied from 10 - 80 kbps which fall significantly short of rates needed for each of the systems. Future work includes offloading some of the composable waveform blocks to accelerators either in FPGA or ASICs.

As waveforms were ported from VHDL to software, some waveform developers noticed a design time savings of 25% to 50%. When this occurred, designers attributed the savings to faster simulation times and better and faster debug tools. Developing the waveform algorithm on a fast x86 processor and using debug tools such as Microsoft Visual Studio and Eclipse allowed quick design iterations and debug before running the same algorithm on the hardware, supporting the new architecture goals of enabling rapid insertion of technology into airborne tactical waveforms.

IV. LESSONS LEARNED

While porting and implementing various airborne tactical waveforms to the new architecture, there were several lessons learned. In the following subsections, we discuss several lessons learned and provide recommendations for future instantiations of the architecture.

A. Software vs. Hardware Instantiations

In any waveform design, there is a tradeoff in implementing components in software vs. hardware. Hardware typically provides very tight timing guarantees at the cost of limited flexibility. Typical waveform mapping of which processing blocks are mapped to software vs. hardware begins with a high level block complexity analysis. Assumptions such as infinite memory/all access in L1 cache, hand assembly (no function call overhead/stack etc.), RISC architecture, etc. are all used to calculate the number of operations per second for each processing block. The ones that either require low latency or have very high ops/sec requirements, are typically mapped

to hardware and the ones that require significant configuration or customization are mapped to software.

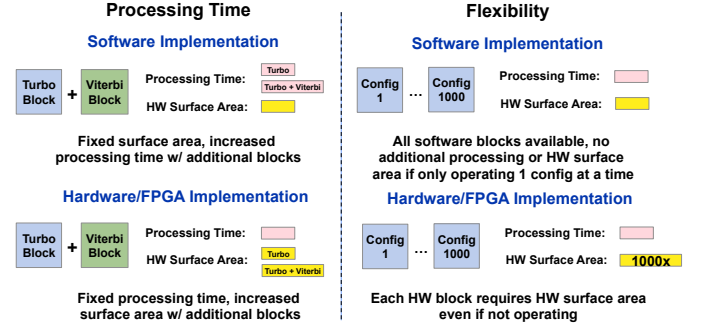


Fig. 5: Hardware vs. Software implementation tradeoffs

Figure 5 illustrates the tradeoff between implementing blocks in hardware vs. software. In terms of processing time, software implementations are often affected by shared load and are difficult to bound deterministically. In hardware, however, processing time is fixed. When comparing flexibility, software blocks are infinitely configurable and several pieces of code can be dynamically loaded to perform different functions. In contrast, there is limited configuration in hardware blocks and significantly different instantiations require additional surface area on the FPGA or ASIC to accommodate.

The goal of the new architecture is to reduce the time to insert new technology into the airborne domain. To achieve this, the flexibility and portability of software implementations push a software-heavy design with the expectation that general purpose processors will eventually advance to a state that will enable software-heavy designs to run at speed. One important recommendation is to limit the transition between software and hardware as data flows back and forth between the two may result in significant I/O requirements and increased latency.

B. Hardware Agnostic vs. Optimized Considerations

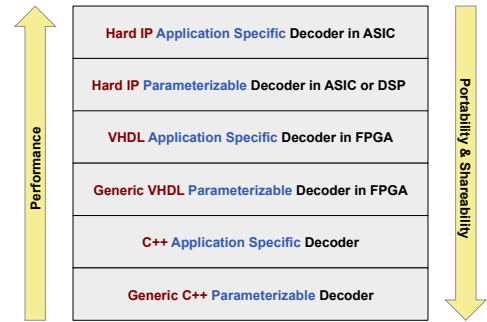


Fig. 6: Agnostic vs. Optimized Considerations

To ensure maximum shareability between each of the waveforms being implemented, each developer created highly parameterizeable waveform blocks using generic C++. These blocks were easily shared among the different waveforms and easily ported to different hardware platforms. However, the data rate performance fell significantly short of the desired

rates. This highlighted the portability, shareability, and performance tradeoff designers must make.

First, parameterizeable code is slower than application specific code. For example, an application specific decoder might be able to take advantage of symmetries in a trellis to improve performance. But, a generic decoder must cover all configurations and cannot take advantage of those symmetries.

Second, high level languages like C++ lend themselves to portability and shareability, but with a performance penalty. We were initially surprised that our implemented waveform code was running slower on the DSP processor than the general purpose processor. But, it became apparent that to gain performance, one must use the DSP vendor's libraries and coding styles. This would have resulted in an undesirable step away from a hardware agnostic system.

Going forward, to gain performance, the waveforms must leverage accelerators in many forms such as configurable hard IP in an ASIC, configurable IP in an FPGA, DSP hard IP accelerators, or custom application specific code. The performance versus portability and shareability of each option is shown in Figure 6. Again, the interfaces to these accelerators becomes the innovative and time-consuming part of the development. To remain hardware agnostic, the interfaces must be abstracted from the waveform designers and creating a common interface from the waveform to hardware accelerators and software implementations would allow ease of leveraging underlying components.

C. Hardware Abstraction Time and Interfaces

While the waveform design time was improved, creating a hardware agnostic eco-system took a lot of time and effort. AMPLib and the Board Support Package (BSP) must understand each new platform. While porting AMPLib to various general purpose processors has proven straightforward, much effort is required with each new platform in studying all new interfaces and creating a high performance method to use those interfaces while hiding that from the application and waveform developers. Examples of interfaces that were studied and abstracted are:

- Interfaces between software waveform components
- Interfaces between an SOC GPPs and DSP processor
- Interfaces between GPPs and FPGAs
- Interfaces between GPPs and semi-configurable ASICs
- Interfaces between processors on different cards
- Interfaces to different A/Ds and DACs

The lesson here is that there is still a large development component whenever new hardware is introduced. However, that development component resides with the hardware designers and the AMPLib providers. Porting the application software and waveform software is ideally unaffected.

D. Interprocess Communications and Memory Management

Within an asymmetric multiprocessing (AMP) system, the Inter-Process Communication (IPC) mechanism is a primary component. The system must factor in the overall computational costs of serializing and deserializing data across a wide

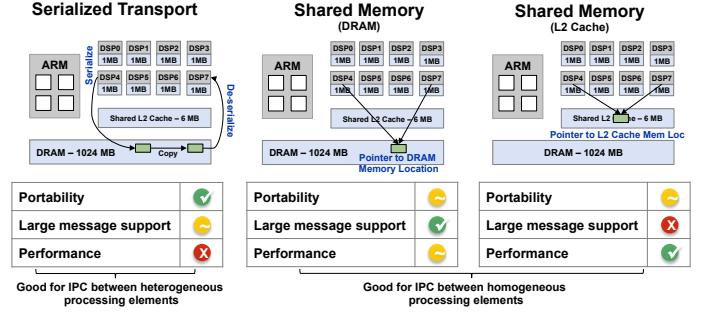


Fig. 7: Memory management

range of bus architectures. Alternatives to serialization of data must be considered for various deployments of components. It is imperative that the architecture be able to leverage shared memory IPC when available, and only 'fall-back' to serialized/deserialized IPC when necessary. Further, architectures that allow leveraging of shared memory in L2 cache, or even L1 cache, can tremendously impact system performance. Figure 7 shows the various transport mechanisms available on the TI system. The tables below the figure show the high level analysis of portability, large message size support, and performance for each method.

Asymmetric multiprocessing systems offer the ability to spread computational work across available cores. One example of this is the parallelization of an algorithm by distributing various pieces to idle cores. An inefficient IPC can defeat the benefits of performing parallel operations across asymmetric cores. If the total cost of communicating with the other cores, including serializing, deserializing and bus transfer times, exceeds the savings in performing parallel computations then the asymmetric multi-core deployment is not valuable.

Further enhancements to an IPC can be made by implementing reference counted shared memory. Waveforms that require the same data to be sent to multiple algorithms for processing in parallel can leverage the existence of a fan-out reference counted data transport. Rules need to be incorporated, such as copy-on-write, for cases when the receiver of the data needs to modify it. Applications need to evaluate the use of IPC between two homogeneous cores versus simply using the cores in symmetric multiprocessing (SMP) mode. Ideally, it would be built into the framework such that switching back and forth between the two modes of operation is a trivial task. It is extremely beneficial for this decision, and all the internals of the IPC, to be hidden from waveform processing blocks.

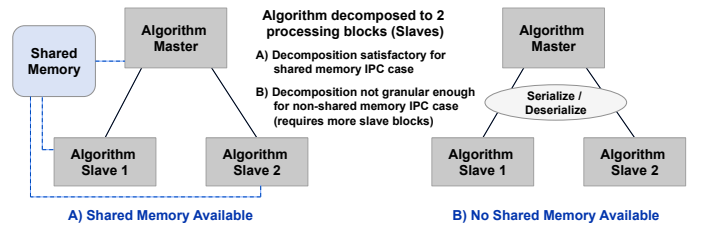


Fig. 8: IPC with and without shared memory

Finally, waveforms must be architected with consideration

given to the rates at which IPC will work on the target SoC (and future targets). Careful consideration has to be made before assuming there will be enough L2 cache, for example, to always have fast shared memory IPC available. The impact of this includes how granular the pieces of an algorithm need to be. If certain capabilities are assumed to be available, then an algorithm may not be decomposed to a small enough set of pieces to work on all AMP systems. Figure 8 illustrates the concept by showing an algorithm that was decomposed to two processing blocks that can be performed in parallel on separate cores. Part A shows a target with shared memory IPC between the cores. In scenario A, the algorithm runs fast enough that the two slave blocks are sufficient. Part B, however, shows a target that does not contain shared memory between all cores and thus must utilize a serialization method to transfer data to the slave cores. In scenario B, the algorithm does not run at the required speed, and since further algorithm decomposition is not possible, the approach cannot meet timing constraints.

E. Standardized Block Interfaces

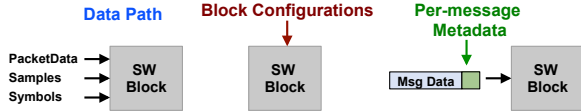


Fig. 9: Standard interfaces must support data and control on a per message level

Although an efficient and generic IPC and physical interfaces lay the groundwork for efficient splitting of processing blocks to different heterogeneous processors, block-to-block message formats and configurations must be well defined and standardized. In particular, developing generic blocks that can be used by multiple waveforms require clear separation of waveform specific metadata and waveform-agnostic configurations. Figure 9 highlights some of the major inputs needed for software processing blocks. For the data path, three types of messages were defined that handled almost all the types of data flowing between blocks:

- Packed Data which handles data from higher layers of the network stack and can be represented in bits/bytes
- Symbols which handles data traversing from encoder/decoders to modulator/demodulators
- Samples which describes I/Q values to be sent to the digital up/down converter and DAC/ADC

In addition to the data path, each block needs to be configured 1) externally through a control interface for setup, and 2) message-by-message. Static configuration and external block control and configuration is fairly straight forward and can potentially be a block-specific specification. It is recommended that blocks of the same type (i.e. encoders/decoders, modulator/demodulators, etc.) follow a similar specification. Waveform-specific configurations should be pushed to generic processing blocks through a waveform-specific block.

Block configuration on a per-message basis, also known as metadata, can be a bit more challenging. Metadata that travels

with each message can either be waveform specific or block specific. This metadata needs to travel with each message to ensure time alignment. The alternative is to have separate paths for metadata and message data which requires complex systems to ensure time alignment.

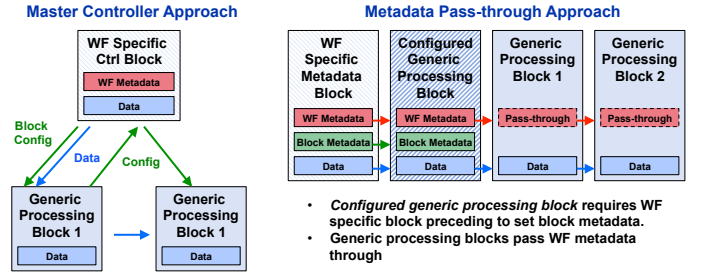


Fig. 10: Master controller vs. pass-through approaches

There are many ways to handle development of generic non-waveform specific processing blocks that can be leveraged for multiple waveforms. Figure 10 shows two potential approaches: a master controller approach and a pass-through approach. In the master controller approach, a waveform-specific block that handles the waveform-specific metadata and dispatches data and block-specific configurations to generic processing blocks. Once the processing blocks finish, the data is either pushed along to the next processing block while the output or metadata returns to the master controller block, or all of the data and metadata is pushed back to the master block to redirect to another generic processing block. The benefit to this approach is that there is clear separation of generic processing blocks and waveform specific information. However, should the master waveform block need to be split amongst different processors, additional complexity is added to synchronize and coordinate the master blocks.

In the pass-through approach, waveform-specific metadata is embedded in each message and travel along with the message. Waveform-specific blocks process the waveform-specific metadata and set block-specific metadata to be processed by proceeding configurable generic processing blocks. Processing blocks (generic or configurable generic blocks) ignore the waveform metadata and simply allows it to pass through the block. The benefit of this approach is that all metadata travels with the message and requires little to no synchronization. The tradeoff is that metadata is copied from block to block and although data can move from generic processing block to generic processing block if no metadata is needed, waveform-specific processing blocks are required prior to generic blocks to set appropriate block metadata for the next block. Additional considerations are needed for software blocks to interact in a generic manner with hardware blocks. Future work is addressing these issues.

As part of the effort, it was determined early on to leverage Google Protocol Buffers [12] to define messages to pass between blocks with the goal of ease of message definition and architecture independent operation. While Protocol Buffers provided flexibility for messages traversing from high layers

of the network stack, its utility became hampered and limited when dealing with samples. One example is that the smallest possible encoding for an integer is `int32`. Defining I/Q samples in 32 bit values significantly increases the message size unnecessarily. Future implementations should consider leveraging appropriate standards such as VITA-49 [13] and others as available for passing messages.

F. Flow Control

One of the key issues to address in any kind of waveform design is flow control in the system. Data packets may arrive asynchronously from higher layers of the network stack and at some point, will be required to map to synchronous sample rates expected by the DAC. This is especially important in software-based designs because estimating each block's processing time can be difficult to estimate deterministically and varies with processor load, parameter complexity, etc. Inter-block queues are needed to pass messages between blocks and due to the different block processing times, these queue lengths can vary greatly.

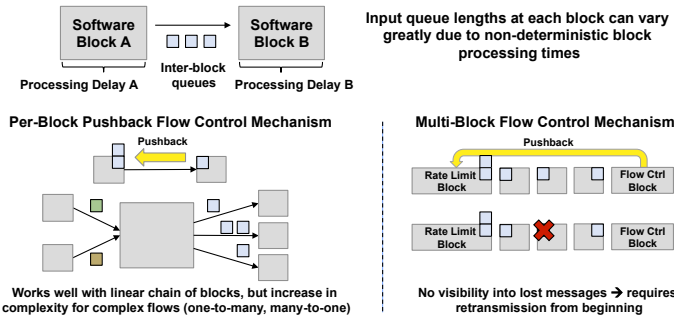


Fig. 11: Flow control to support multiple fan-in/fan-out implementations require a scalable approach

Figure 11 illustrates some options to implement flow control in software-based waveform designs. One approach is to characterize the performance of each block under several conditions to bound the processing delay. Queuing theory can then be leveraged to determine the inter-block queue size required between each block to guarantee no message drops due to queue overflow in the system. The benefit is that this simplifies processing at each block and leaves queue configuration to a system-level designer. The tradeoff is that processing delays can be difficult to bound in software-based implementations and if underlying IPC is not stable, the approach can fall apart.

Another method to resolve this issue is to have a push-back mechanism either on a block-by-block level or on a multi-block level. In the block-by-block pushback approach, queues between each block are small and blocks processing data stop a previous block from sending new data until it is done. While the approach is simple for a single chain of blocks, the goal of the new architecture is to allow any number of inputs and outputs to/from each block. This enables any other block to “tap-off” streams of data and perform additional processing. Significant complexity would be needed in the push-back mechanism to accommodate this approach. In multi-block

push-back, specific blocks are developed to rate limit processing blocks earlier in the system. While this approach enables flow control without the complexity of managing block-by-block pushback, it is difficult to pinpoint data loss anywhere in between the flow control and rate limit blocks. Additional considerations may be needed to accommodate multiple inputs and outputs to intermediate blocks.

G. Fixed vs. Floating Point

Typical waveform designs begin with algorithms implemented in floating point for simplicity, which are then converted to fixed point for performance. Floating point implementations of waveform processing blocks are typically easier to standardize, implement, and result in no loss of dynamic range. This enables rapid development and evaluation of new algorithms. The performance, however, is typically slower than fixed-point implementations. Additionally, since hardware is typically fixed point, conversion may be needed. Fixed-point implementations, however, yield higher performance, but are harder to implement and harder to standardize. Interfaces can be 16, 32, 64, etc bit fixed point, and converting from one to another can result in loss of dynamic range.

The problem is exacerbated by the fact that multiple conversions between fixed and floating point induces errors and can cause increased latency in the system. Developing generic processing blocks requires standardization of fixed/floating point expected input/outputs to ensure enough dynamic range is available. Furthermore, identifying boundaries for fixed and floating point conversion is essential to minimize errors and create bottlenecks for performance.

V. CONCLUSION

In recent years, there's been a large push in the DoD to adapt and respond to threats in shorter timescales, particularly in the airborne tactical domain. To mitigate this issue DARPA is examining architectures that enable insertion and update of waveform technology at the processing block-level. The vision is for these waveform processing blocks to be composed together dynamically to form various communications, SIGINT, EW, etc waveforms. The processing blocks can be either software blocks or FPGA/ASIC blocks and as part of the effort, several DoD airborne tactical waveforms were ported to the new architecture. In this work, we focus on the software-based processing blocks and describe lessons learned in implementing DoD airborne tactical waveforms in the architecture. Future work includes adding FPGA and ASIC accelerators into the processing chain and deployment on different hardware platforms.

VI. ACKNOWLEDGEMENT

This work is sponsored by DARPA under Air Force Contract # FA8721-05-C-0002. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. The authors would also like to acknowledge the following people: Richard Ertel, Ryan

Hinton, Andy Morrical, Alan Stone, Robert Klinkhammer, Russ Hamilton, Jason Hillger, and Aradhana Narula-Tam.

REFERENCES

- [1] DARPA Strategic Technology Office (STO), "Communications in Contested Environments (C2E)," *DARPA Broad Agency Announcement (BAA)*, no. DARPA-BAA-14-02, December 2013.
- [2] DoD MIDS Program Office, "System Specification (SS) for Link 16 Waveform for the Multifunctional Information Distribution System Joint Tactical Radio System (MIDS JTRS)," DoD MIDS International Program Office, Tech. Rep. SS-J-10002 Rev B, April 2007.
- [3] B.-N. Cheng, F. J. Block, B. R. Hamilton, D. Ripplinger, C. Timmerman, L. Veytser, and A. Narula-Tam, "Design Considerations for Next-Generation Airborne Tactical Networks," *IEEE Communications Magazine*, May 2014.
- [4] "Java Virtual Machine Wiki." [Online]. Available: https://en.wikipedia.org/wiki/Java_virtual_machine
- [5] "Android ART and Dalvik." [Online]. Available: <https://source.android.com/devices/tech/dalvik/>
- [6] JTNC. (2015) JTNC Standards. [Online]. Available: <http://www.public.navy.mil/jtnc/sca/Pages/default.aspx>
- [7] AMD. Heterogeneous Computing. [Online]. Available: <https://developer.amd.com/resources/heterogeneous-computing>
- [8] TI. C6000 Multicore DSP + ARM SoC. [Online]. Available: http://www.ti.com/lscds/ti/processors/dsp/c6000_dsp-arm/overview.page
- [9] "RapidIO Specifications." [Online]. Available: <http://www.rapidio.org/rapidio-specifications>
- [10] Xilinx, "Zynq-7000 All Programmable SoC." [Online]. Available: <http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/>
- [11] Analog Devices, "AD9361: 2 x 2 RF Agile Transceiver." [Online]. Available: <http://www.analog.com/en/rfif-components/rfif-transceivers/ad9361/products/product.html>
- [12] Google. Protocol Buffers. [Online]. Available: <https://developers.google.com/protocol-buffers/>
- [13] VITA, "ANSI/VITA 49.0 VITA Radio Transport (VRT) Standard," VITA, Tech. Rep., 2015.