

HARDWARE-CENTRIC IMPLEMENTATION CONSIDERATIONS FOR BBC-BASED CONCURRENT CODECS

William L. Bahn
Leemon C. Baird III
United States Air Force Academy

***Abstract**— The present trend in communications technology is toward greater reliance on software-defined radios and a lessening dependence on dedicated hardware components. However, in general, dedicated hardware components will likely maintain a significant edge in processing capability. Thus, for high-performance systems, it is worth considering the advantages and disadvantages associated with both full-custom and firmware-defined hardware architectures alongside software-defined ones. This paper briefly considers some of issues associated with both full-custom application specific integrated circuit (ASIC) approaches as well as more generic programmable logic options such as field programmable gate arrays (FPGAs) and digital signal processors (DSPs) in the implementation of BBC-based concurrent codecs.*

I. OVERVIEW

The new field of coding theory, concurrent coding theory, is steadily building a solid theoretical foundation [1], [2], [3] and issues of practical significance such as dealing with jitter and oscillator mismatch [4] and extending the critical density limits [5] have been explored, both from a theoretical perspective as well as in simulation and hardware demonstration using software defined radios. It is now appropriate to begin considering practical implementation issues including some of the options that are available to system designers and how the peculiarities associated with a BBC-concurrent codec might influence the resulting design decisions.

This paper is not an in-depth examination of any of the options discussed. Instead, it is intended to cover a broad spectrum of options and issues with, as a result, very broad brush strokes. The goal is to establish a somewhat holistic framework so that future work on specific issues is less likely to be performed in a vacuum with absolutely no awareness of the multitude of other issues. Without this framework, it is likely that much work will be performed that has little chance of surviving subsequent integration into a final system.

II. SOFTWARE-, HARDWARE-, AND FIRMWARE-DEFINED SYSTEMS

For a number of years a major trend in communications technology has been the drive toward greater reliance on software-defined radio architectures. A "software-defined radio", or SDR, is simply one in which much of the signal processing traditionally performed by dedicated hardware components are performed by software processing blocks instead. In the extreme case, a transmitter would consist of little more than a digital-to-analog converter (DAC) driving an antenna, perhaps through an wideband power amplifier. On the other end, a receiver would be little more than a wideband low-noise amplifier (LNA) taking the signal from an antenna and feeding it to an analog-to-digital converter (ADC). Everything else, including modulation and demodulation, would be performed digitally. Although some software-defined radios operate exactly this way, particularly in the HF radio bands, most SDRs are a compromise with software processing blocks performing baseband and perhaps intermediate-frequency (IF) processing and more traditional (or at least more rigidly-defined) hardware blocks performing final modulation and demodulation and/or up- and down-conversion to and from the carrier frequency.

Software defined radios bring to the table extreme flexibility and upgradability since their behavior can be largely, if not entirely, altered simply by changing the software running on them. This permits, for instance, radios to be upgraded to conform to waveforms defined after the radio was already in the field. It also permits a radio's performance to be enhanced without requiring hardware modifications. For instance, a bandpass filter that is found to give unacceptable performance in a given operating environment can be easily replaced by a better one, as long as the processor remains capable of running the upgraded software.

However, it would be a mistake to conclude that software-defined elements are always the route to take. Indeed, that they are an alternative at all has been made possible

primarily by the tremendous growth in digital processing power and speed over the last couple of decades. However, in some respects this is basically equivalent to saying that software processing capabilities are finally becoming "good enough" to take the place of the more costly and less flexible hardware components that have been "good enough" for decades and which, like the processors they are now competing with, have continued to improve over that same time period.

Not surprisingly, it will always be the case that whatever a software-defined component can do, a dedicated hardware component can be built to do it faster, better, and – in sufficiently large production – cheaper. As a result, as long as there are applications that are pushing the state of the art in communications technology, there will always be a place for dedicated communications hardware. But while a hardware implementation may take months or even years to successfully implement, a software-defined solution can frequently be implemented in days if not hours.

Sitting somewhere between hardware-defined and software-defined implementations are firmware-defined ones. A firmware-defined solution, in this context, is one that relies on the configuration of programmable logic devices to define the logical behaviors of the hardware. For our purposes, a program that is stored in an embedded processor is still a software-defined system. Having said that, it is inescapable fact that the distinctions between hardware, firmware, and software are becoming increasingly blurred.

For clarity and brevity, the remainder of this paper will use "hardware-centric" and "hardware-defined" to refer collectively and generally both to full custom hardware approaches and programmable logic approaches. Where the former are being referred to specifically, terms such as "full-custom" or ASIC will be used. Where the latter is being singled out, terms such as "programmable logic" or FPGA will be used.

III. PROCESSING BOTTLENECKS IN CONCURRENT CODECS

The advent of concurrent coding theory and its application to jam-resistant communications [1] is a case in point where a new technology may not be practical to implement using today's SDR capabilities (although that is far from certain at this point) but may, at least initially, need to leverage the capabilities offered by hardware-centric solutions.

The processing bottleneck of a concurrent codec will almost certainly be the packet decoder. This is both due

to the nature of the encoding and decoding algorithms and also because this, by design, is the element that an adversary is going to need to overwhelm. Hence the packet decoder must be highly optimized since it's ability to keep pace with the decoding workload will, in practice, define the realizable processing gain of the system.

The key feature of a concurrent codec that lends itself to hardware-centric architectures is the fundamentally parallelizable nature of the decoding algorithm. The most obvious such feature is the tree-nature of the decoding path. In a sequential processor, each time a tree branches one branch is pursued while the other branch is placed on hold. However, in a hardware-centric solution, multiple decoder cores could be implemented and placed in a resource pool such that each time a branch is encountered, the new branch is spawned off to one of the available decoders, which then proceeds concurrently and independently to decode the new branch. When a branch dies, that decoder is released and returned to the pool of available decoders. While this is conceptually similar to a normal sequential processor spawning threads, it must be kept in mind that those threads do not truly run concurrently but rather time share the processor (or processors in the case of multi-core chips). In a hardware-centric solution, it is possible to have as many decoder cores as the cost and other constraints dictate and that number could easily be in the dozens or even in the hundreds (with thousands not an unthinkable number).

One detail that must be considered is how the situation is handled when a branch is reached and no available decoders are in the resource pool. In order to prevent the decoder from locking up at this point, each decoder should have the capacity to continue performing a depth-first search on its assigned branch using the single-processor approach without spawning new branches to the decoder pool. This, in turn, begs the question of whether a decoder that has switched to this mode should be capable to switching back to using the decoder pool once decoders are again available. While the answer is ideally yes, doing so introduces additional bookkeeping overhead since now each decoder must keep track of which branches were spawned and which weren't. While this can be done with a single status bit per message bit per decoder, the total number of memory bits grows quickly.

In comparison, if a given decoder starts in a cooperative mode (meaning that it will spawn new branches to the decoder pool) and then (possibly) switches to and stays in an independent mode later (meaning that it will process branches internally), then it only needs to keep track of the

depth at which it was invoked and the depth at which it switched modes. Conceptually it would be possible to use a "mode stack" to keep track of subsequent switches in and out of these modes, but eventually the possibility exists that these registers will be exhausted and that the decoder must then branch internally from that point on, even if all of the other decoders have finished their work and are available.

Since the goal is unkeyed jam-resistance, it must be assumed that any adversary will have access to the decoder's architecture and firmware and will attempt to exploit that knowledge to develop effective attacks. In this case, it is important that the attacker not be able to easily force a situation where, due to the above behavior, a packet is reduced to using only a small number of available decoders. In order to create this situation, the attacker would need only to construct a packet that rapidly branches, thus exhausting the decoder pool, and then steers one branch to exhaust its mode stack thereby locking it into an independent mode. At this point the other branches can be allowed to expire. Constructing such a packet may or may not be trivial. One possible way to defeat this attack is to add a stochastic component to the mix whereby decoders have a finite probability of opting not to spawn a branch even if a decoder is available in the pool. An alternative would be to randomly pick which branch in a path is spawned and which branch is maintained by the present decoder, since a spawned branch always starts off with a fresh mode stack. The tactics would greatly increase the difficulty of constructing an effective attack packet.

IV. MEMORY BOTTLENECKS

In a typical software-defined radio, the memory needed for all aspects of operation are in a unified memory space. In a hardware-centric implementation that is exploiting decoder parallelization the potential for encountering memory access bottlenecks in such a unified space must be considered. For instance, each decoder needs to access the packet contents, possibly at the same location at the same time. In addition, each decoder needs to access memory in order to compute the next message-prefix hash as well as maintain and update its state variables. Fortunately, a hardware-centric implementation has the ability to segment memory resources with a very fine degree of resolution. For instance, each decoder can maintain local memory to manage its own state information. Since each memory segment can be accessed independently and simultaneously, the associated bottlenecks are eliminated.

In theory, each decoder could receive its own copy of the packet; however, since practical packet sizes are likely

to be in the hundred kilobit to perhaps gigabit range, this will likely be prohibitive. However, while this bottleneck may prove to be the limiting factor in determining decoder performance, the fact that each decoder must compute message prefix hashes at each stage of decoding means that a significant amount of time is available between necessary packet data queries for a given decoder. Thus it should be possible to use a standard memory access queue to service all of the needs for access to packet data for a fairly large number of decoders before this bottleneck begins affecting performance.

Should, however, packet data access prove to be a limiting bottleneck, it can be greatly alleviated by segmenting the packet into subpackets, each with its own access queue. The degree of segmentation possible depends on whether the packet data is stored in standard memory chips or internally within the processing device. If stored externally, then I/O pin count limitations can quickly come into play. However, modern FPGA's can have as much as 10 megabits of internal block RAM – and this number will only increase as time goes by – while fully custom ASICs can have as much as the designer is willing to pay for. It should be noted, however, that placing large amounts of memory on a CMOS processor is quite expensive compared to using a separate DRAM chip.

One "trick" that can be leveraged if DRAM is implemented on-chip is that there should be no need to perform memory refresh cycles. Since each memory location will be written two each time the packet advances, which will happen at the codeword baudrate (or perhaps some reasonably small multiple of that period), the memory will automatically be refreshed with each advance.

V. HASH FUNCTION GENERATOR

As mentioned previously, each decoder must make regular calls to a hash function to compute mark locations. Depending on the details of the algorithm, this might entail multiple calls per message bit. In a typical CPU-based SDR, a hash function call is a relatively expensive processing step. However, a hardware-centric design can implement a hash function block that can compute the output very quickly and efficiently and it is entirely reasonable to dedicate a block to each decoder and configure the logic to pre-compute the next potential hash outputs at the same time that the decoder is querying the results from the previous hash to determine which path, if any, to follow.

Also, the BBC-algorithms do not require cryptographically secure hash functions. Research to date indicates that linear feedback shift registers (LFSR) having sufficiently

long internal state are adequate. One promising approach is to use a novel LFSR architecture that permits branching in response to the state of a message bit as well as being able to run backwards to “unwind” up the decoding tree. If this approach proves workable then hardware-based implementations can leverage the rich selection of fast and efficient LFSR optimizations available.

VI. JITTER AND OSCILLATOR MISMATCH COMPENSATION

Concurrent codecs are highly tolerant of signal jitter and mismatch between transmitting and receiving oscillators. However, this tolerance comes at the expense of a non-trivial amount of computational overhead. However, like the core decoding functions, this overhead lends itself to a number of parallelization strategies when using a hardware-centric implementation.

Compensating for signal jitter by way of receiver-side mark extension can be offloaded to the logic performing the packet data accessing. In essence, a global parameter indicating the degree of mark extension is used to automatically query not only the nominal mark location placed in the access queue by the decoder, but also the appropriate number of adjacent mark locations. There are number of ways to accomplish this and which method is best depends on a number of factors such as how much margin the data access block has to work with and whether or not the packet data memory space has been segmented.

Compensating for oscillator mismatch using a traditional CPU-based SDR involves many multiplications of packet location indices by the same scaling factor. In a hardware-centric implementation, the scaling factor can be applied to one input of a dedicated hardware multiplier and then the indices applied to the other with the scaled output being available more-or-less immediately at the output and ready for use. Furthermore, the process of generating the set of scaling factors to be used for one packet can be performed concurrently with other processing, possibly even while the prior packet is being decoded. An other alternative is to pre-compute the scaling factors for every possible position within the compensation window (a number likely to be between a few tens and a few hundreds) and store them in a look-up table.

An method offering even greater potential efficiency is to place a hardware multiplier next to each scale factor register. This then allows the simultaneous computation of all of the compensated mark locations. A status bit can be maintained for each scale factor register. The bit would initially be set according to the packet contents within the

window and then cleared as marks are not found. Once all of the status bits are cleared, the decoder knows that no messages exist within that packet and can move on.

Whether the packets are decoded one scale factor at a time or simultaneously, the hardware will need to be able to scan a section of memory (either the packet bits within the window or the status bits) and move quickly from one HI bit in the string to the next HI bit. In a traditional CPU the ability to do this efficiently is very limited. However, in a hardware-based implementation a token passing and capture scheme can be used to permit the system to skip all of the intervening LO bits (within reason) in a single clock cycle.

VII. CONCLUSION

While concurrent codecs in general, and the BBC algorithms in particular, are easily implemented in a software-defined radio architecture, the decoding algorithm is resource intensive and it’s performance is the limiting factor in overall receiver performance, particularly in a hostile environment. To mitigate this vulnerability, the decoder can be implemented in a hardware-centric architecture that leverages the inherently parallelizable nature of the decoding algorithm. This can be achieved either in a semi-custom FPGA-based implementation or, to an even greater extent, in a full-custom ASIC implementation.

VIII. ACKNOWLEDGEMENTS

This work was sponsored in part by the Air Force Information Operations Center (AFOIC), Lackland AFB, TX, and was performed principally at the Academy Center for Cyberspace Research (ACCR) at the United States Air Force Academy.

REFERENCES

- [1] L. Baird, W. Bahn, and M. Collins, “Jam-resistant communication without shared secrets through the use of concurrent codes,” United States Air Force Academy, Tech. Rep. USAFA-TR-2007-01, 2007.
- [2] L. Baird, W. Bahn, M. Collins, M. Carlisle, and S. Butler, “Key-less jam resistance,” in *Proc. 8th Annual IEEE SMC Information Assurance Workshop (IAW)*, jun 2007, pp. 143–150.
- [3] W. Bahn, L. Baird, and M. Collins, “Jam-resistant communications without shared secrets,” in *Proc. 3rd International Conference on Information Warfare and Security (ICIW08)*, apr 2008, p. (CD).
- [4] —, “Oscillator mismatch and jitter compensation in concurrent codecs,” in *Unclassified Proc. IEEE Military Communications Conference (MILCOM:08)*, nov 2008, p. (CD).
- [5] W. Bahn and L. Baird, “Extending critical mark densities in concurrent codecs through the use of interstitial checksum bits,” United States Air Force Academy, Tech. Rep. USAFA-TR-2008-XX, 2008.