In Proc. of the IEEE Asilomar Conference on Signals, Systems, and Computers, pages 243-247, Pacific Grove, California, November 2009. Invited paper.

# Integration of Dataflow Optimization Techniques into a Software Radio Design Framework

George F. Zaki*, William Plishker*, Tim Oshea†, Nick McCarthy†,
Charles Clancy†, Eric Blossom‡, Shuvra S. Bhattacharyya*

*Electrical and Computer Engineering Department
University of Maryland
College Park, Maryland, USA
{gzaki, plishker, ssb}@umd.edu

†Laboratory for Telecommunications Sciences
University of Maryland
College Park, Maryland, USA
{namccar,clancy}@ltsnet.net, tim.oshea@ieee.org

‡Blossom Research, LLC
Reno, NV, USA
eb@comsec.com

***Abstract*—Application specific design frameworks, such as GNU Radio for software defined radio, facilitate fast design flows by leveraging the common application structures of particular domains and rich libraries of elements tailored to them. However, due to their focus on the application, implementations derived from these frameworks are often unable to take advantage of target platform features to achieve high performance. To apply more extensive optimizations, applications described in such frameworks can be refined to formal models, making them more amenable to analysis and optimization. In this work, we present a method for integrating GNU Radio with a dataflow design framework, the Dataflow Interchange Format (DIF). We describe a translation from GNU Radio applications into a formal dataflow model and the software infrastructure we have used to integrate the two software packages. This integration allows GNU Radio to employ a variety of dataflow schedulers to improve performance on existing applications. Furthermore, by applying formal models to the application and the target architecture, this integration should allow for target specific optimizations for additional performance and target flexibility.**

## I. Introduction

With the increasing proliferation of wireless standards and the rise of the complexity of many of them, software-defined radio (SDR) has proven to be useful in providing application developers with a fast, flexible development environment as well as the ability to target general purpose processors (GPPs). While hardware implementations are able to achieve low power at low per part costs with sufficient volumes, SDR-based implementations have the ability to change to accommodate new standards while implicitly taking advantage of Moore's law through increases in processor performance.

Recently, performance increases in processor design have not come from frequency increases or improvements in uniprocessor architectures, but instead from instantiating multiple cores on a single die. This effectively increases the computational horsepower on-chip while not adversely affecting power consumption. To continue to track Moore's law in this multicore environment, SDR applications and design flows must be able to express the style of parallelism necessary to take advantage of these multicore devices. For many of these multicore architectures, programming models and environments are customized to each, to take advantage of their particular cores, memory hierarchy, and topology.

As SDR applications have the parallelism and performance demands to be suitable for many of these nascent multicore architectures, this creates the potential of many unique targets to be considered when going to implementation. Porting code to new architectures can be an error-prone time-consuming process that involves careful tweaking of signal flow parameters to achieve satisfactory performance.

One promising candidate for bridging the gap between the high-level application description and the high performance multicore architectures is dataflow. Dataflow models have been used for decades to represent processing structures as directed graphs where each node consists of code that executes on data flowing across the vertices. This model is well suited to digital signal processing where the graph typically consists of filtering operations and other types of coarse grain functions that operate on streams of data. SDR can leverage dataflow design concepts and allow for modular construction of signal processing systems where the type of signal being processed can be dynamically changed based on the required application.

In this work we discuss the impact of applying dataflow scheduling techniques to a popular open source SDR package:

GNU Radio [1]. This has immediate benefits by allowing for derivation of static, single-threaded schedules that execute blocks in a static graph in streamlined manner that minimizes that required inter block buffers. Longer-term, we plan to accept descriptions of hardware architectures, and systematically interface with architecture-specific system calls to efficiently schedule GNU Radio flowgraphs on diverse and heterogeneous platforms. We demonstrate some of the performance potential of these devices by implementing a common SDR building block on modern multicore device: a graphics processing unit (GPU).

## II. Background

### A. Dataflow Modeling

Dataflow models are widely used in design, analysis and implementation of DSP systems. Different models exists to match various types of applications as synchronous dataflow (SDF) [2], cyclo-static dataflow (CSDF) [3] and Boolean dataflow (BDF) [4]. A dataflow model of an application captures important data dependency information between system modules. The model consists of a directed graph $G$ where nodes or *actors* represent computation modules and edges represent communication links between actors. Depending on the level of abstraction at which the application is modeled, actors can represent simple operations as multipliers or more complicated computations, such as fast Fourier transforms. In typical computational platforms, edges are implemented as FIFO buffers. Data in the FIFO buffers are encapsulated in *tokens*. When an actor $a$ executes or *fires*, it consumes zero or more tokens from each input edge and produces zero or more tokens on each output edge.

Using the dataflow model of an application, different design space exploration techniques can be applied. Such techniques can be useful to detect and exploit possibilities for parallel execution across actors, schedule actors for efficient execution, and derive buffer bounds. Some of these analysis techniques can be particularly useful when the DSP system is targeted to a multicore platform. We elaborate further on this in Section IV-D.

### B. Dataflow Interchange Format

The Dataflow Interchange Format (DIF) [5] is a standard language and associated software package that provide for mixed-grain specification, analysis and synthesis for dataflow-based design and implementation of signal processing systems. DIF provides a unified framework to facilitate technology transfer of applications among different DSP design tools. In order to achieve this goal, the DIF front end includes various tools to automate the importing and exporting of application between the DIF environment and other dataflow-based design tools. An initial step towards this task between DIF and GNU Radio is described in the section V-A.

To facilitate rapid prototyping, designers can focus on describing the dataflow behavior of their application using the DIF package, which comes with a set of algorithms and engines to analyze and optimize different application properties. These features of DIF, together with a set of library module implementations that are targeted to a specific platform, can be used to derive high quality embedded software implementations with a high degree of automation [5].

### C. Software Defined Radio

In recent years, we have witnessed rapid growth in the computational capacity for fixed and floating point arithmetic in processors. This has allowed radio tasks that could once only be implemented in dedicated analog circuits, analog/digital ASICs, or FPGA logic to now be achievable using software. Additionally numerous modern wireless communications standards have chosen to exploit the low cost of computational resources and have begun to significantly drive up the complexity of waveforms in order to achieve improved spectral efficiency and coverage.

This increase in complexity has led to two choices for those building radios for these standards: continue to fabricate vastly larger ASICs or explore software-based solutions. With the increases in application complexity, ASIC-based solutions must consume many more gates to implement all of the functions that the radio may need to perform, and requires complex hardware state machines to orchestrate the interaction of the various specialized subsystems. Software defined radio avoids much of these problems by reusing computing resources on a more fine grained level. This reuse is achieved by simply implementing all of the necessary routines in software, and implementing flow graphs for signal processing routines in software.

### D. Related Work

Previous research to reduce the gap between application and multicore processor modeling is reported in [6]. In this work, the authors develop a new programming language that is able to describe SDR systems and their implementation on Single Instruction Multiple Data (SIMD) platforms. In [7], a hierarchical dataflow model that captures different communication patterns between actors as well as data flow behavior within an actor is proposed. The authors validate their work on a multiprocessor system-on-chip architecture. In [8], the authors investigate trade-offs associated with using different data flow models for selected SDR actors.

## III. Formal Modeling of Software Defined Radio Applications

### A. Current GNU Radio model

GNU Radio is an open-source SDR engine and a collection of many common radio primitives. GNU Radio allows users to specify a directed acyclic graph (DAG) by using a Python script to instantiate previously-compiled blocks and interconnect them at run-time. These blocks represent common signal processing operations, ranging from digital filters to modulators to forward error correction.

GNU Radio provides a hierarchal flowgraph mechanism that allows primitive signal processing blocks to be rapidly connected together to form a full flowgraph. A typical flowgraph

begins with a data source block, proceeds sequentially down a number of signal processing blocks, and then terminates in a data sink block.

Between each block is a buffer that is managed transparently. Buffers are generally refined into implementations that are appropriate for the targeted architecture and operating system. Most commonly the buffer implementations utilize memory allocated on the heap and are carefully matched to the system page size. Blocks contain buffer readers and writers that maintain their appropriate pointers into each of their input and output buffers, all of which is typically transparent to the user.

All primitive blocks take one of the following two forms, which becomes an important consideration for scheduling.

- Synchronous Blocks — Primitive blocks inheriting from a *gr-sync-block* always maintain a fixed ratio of input items consumed to output items produced. This may be 1-to-1 or $N$-to-$M$, but it is a fixed value and this is enforced by the methods available to each block implementation.
- Non-Synchronous Blocks — Primitive blocks inheriting from a *gr-block* do not need to maintain a fixed input to output item ratio. Instead, the work function (core block functionality) determines during each execution how many items will be consumed from the input buffer and how many will be produced in the output buffer.

GNU Radio currently has two automated run-time schedulers. The original one is single-threaded. A topological sort of the blocks is executed in order, where each actor is executed until its input buffer is exhausted. When the last block is completed, execution resumes at the first block. The multithreaded scheduler instead instantiates each block in its own thread, where mutexed buffered FIFO queues are used to pass data between them. The second scheduler involves more system overhead, but allows GNU Radio to run efficiently on multicore processor architectures.

### B. Dataflow Modeling of GNU Radio Actors

In this work, the goal is to develop new capabilities in DIF for synthesizing efficient SDR implementations. As shown in the previous subsection, the scheduling of tasks is currently handled without offline analysis. Incorporating dataflow scheduling by generating a DIF file through a new module in GNU Radio will allow us to perform more design space exploration.

SDR blocks that have fixed production and consumption rates can be easily mapped to SDF models. In this case, multiple optimization techniques can be applied at compile-time to meet specific platform constraints (e.g., see [5]). On the other hand, variable rate actors can be mapped to the more flexible *core functional dataflow* model of computation [9]. In this model, every actor has a set of *modes* such that the production an consumption rates are fixed in each mode, but can vary across different modes. Core functional dataflow representations are amenable to *quasi-static scheduling*. This form of scheduling permits dynamic changes in dataflow behavior while fixing significant portions of schedule structure at compile time, which generally increases efficiency and predictability compared to conventional dynamic scheduling approaches. Quasi-static scheduling for core functional dataflow graphs is addressed in [10].

## IV. Multicore Scheduling for SDR Applications

A variety of multiprocessor scheduling techniques can be applied to SDR applications. In this section, we discuss the different tasks involved in multiprocessor implementation, and then discuss issues related to the exploitation of parallelism for different types of applications. We also discuss implementation aspects for a specific multi-core platform — the Cell Broadband Engine (CBE).

### A. Dataflow-based Scheduling and implementation for Multicore Processors

Multiprocessor implementation involves a number of different tasks, which are enumerated below. A related decomposition of multiprocessor implementation, but one that is focused specifically on the scheduling aspect, is presented in [11].

1) Clustering and assignment — this involves grouping subsets of actors into *clusters*, and assigning each cluster to a processing unit.
2) Ordering — Ordering can take place at two levels. The first is ordering the execution of clusters that form the graph, and the second is ordering the execution of actors within a single cluster.
3) Buffering — this includes calculating the sizes (capacities) for the FIFO buffers that are required between different clusters and different actors.
4) Synchronization — the specific form of synchronization that we examine in this paper is *barrier synchronization*, where all of the processing units wait until they reach a certain point in the flow of execution. Once all processors arrive at this point, they exchange results and then resume concurrent execution.

Depending on the application graph type, the first three tasks can be performed at compile-time or at run-time. Compile-time clustering, assignment, and buffering provide low-overhead implementation, and can be applied by extracting statically schedulable regions from the application [10]. Analysis of such regions can allow designers to explore important trade-offs among context switching, parallel execution, and memory management efficiency.

### B. SDR Applications

As the number of processing elements per die increases, different levels of parallelism can be exploited in wireless communication systems. Application level parallelism occurs when two independent branches of the application graph can execute simultaneously. In this case, clustering dataflow dependent actors and running them on a single processing unit may facilitate synchronization and reduce communication overhead.

A more fine grained level of parallelism can be exploited across multiple invocations of the same actor for certain kinds

of actors — e.g., actors that lie outside of strongly connected components in the dataflow graph. Such actors can often be distributed across many processing units while requiring relatively low synchronization overhead. Practical examples of such actors include finite impulse response (FIR) filters and quadrature amplitude modulators.

### C. Implementation on Multithreaded Processors and the CBE

GNU Radio currently provides several features that make it relatively easy to exploit thread-level parallelism while building waveforms on top of multi- and many-core processors. The current model typically runs each primitive signal processing block within a separate thread and then protects interconnecting buffers using traditional locking mechanisms. Block execution is then essentially driven by interprocess communication signaling mechanisms as data is added to the incident input buffers.

This execution model has been successfully leveraged on non-symmetric multi-core platforms such as the CBE as well simply by allowing the work functions of certain blocks to batch work out to a co-processor and then block, yielding their time on the primary processor until a result returns. While this imposes some new constraints based on available cache size and memory latency, it has successfully achieved near-linear speed up on 16-core CBE systems for work functions that run greater than 50 microseconds when the work functions utilize double buffering on the SPE.

This model still presents several difficulties. If each primitive block represents one thread of execution, the granularity of the work function must be carefully considered to ensure that there will be enough concurrency to adequately load all of the available cores, and at the same time to not burden the waveform with excessive threading overhead by implementing primitives of excessively small granularity. These two constraints present fundamental difficulties with providing portable signal processing blocks appropriate for both fine grained and coarse grained parallel architectures.

### D. Multicore Model

When mapping systematically from an application model to an architecture model, the following parameters are important to consider in relation to the targeted architecture model.

- The number and type of processing units. Processing units can vary from complete processors with elaborate instruction sets to customized hardware acceleration units. The number of processors and functional units gives us bounds on the amount of parallelism that we can exploit from the application graph. Also, common DSP functions can make use of of dedicated hardware that guides the process of actor-to-processing-unit assignment.
- Memory hierarchy and communication schemes. The access speed, size and latency per transaction of different memory levels affect the performance of DSP functions. Values of these parameters give us guidelines on how much computation to perform per memory transaction in order to mask communication latency. Second, the memory hierarchy affects where to place the FIFO buffers required between actors (i.e., multiple data dependent actors can take advantage of communication through fast but small memories).

## V. Preliminary Results

Models for applications as well as for target platforms are useful in order to perform effective multicore scheduling. A well-designed, dataflow-based application model allows the designer to study and analyze coarse-grain parallelisms and other useful forms of high level application structure. *The DIF Language* (TDL) provides a language for representing DSP-oriented dataflow application graphs [5], [9]. A distinguishing characteristic of TDL is support for a variety of different dataflow models of computation, and for integrating subsystems in different representations (dataflow models) so that individual subsystems can be represented and analyzed in terms of dataflow techniques that are tailored towards the behaviors and constraints associated with those subsystems.

### A. GNU Radio Exporter to DIF

An TDL exporter was written to transform system descriptions from python scripts in GNU Radio to equivalent TDL representations. Figure 1 shows a snapshot of a system generated by the basic GNU radio graphical user interface and its corresponding graph as generated by the DIF package.

The exporter starts by flattening the application graph, and computing a topological sort. Then for every block in the GNU Radio model, a DIF actor and its incident dataflow edge connections are created and stored temporarily in linked lists. After traversing all of the application graph blocks, the lists of actors and edges are exported to a TDL file using the appropriate TDL syntax.

One difference between the GNU Radio representation format and that of DIF is in the expression of dataflow production and consumption rates. In DIF, these rates are represented as integer numbers of tokens produced and consumed for every actor invocation. On the other hand, in GNU Radio, the rates are managed in terms of individual floating point numbers that represent ratios between correpsonding production and consumption rates. Thus, part of the process of converting a GNU Radio representation to TDL involves approproriate conversion of data associated with production and consumption rates.

In this section, the CUDA GPU architecture is taken as an example of a target multicore platform. As described in section III-B, the main factors that affect actor performance include the number of the processing units, memory hierarchy, and available communication schemes. To explore effects related to these factors in software radio implementation, we examine an FIR filter implementation on a GPU.

### B. FIR Filter Implementation

Figure 2 shows the memory hierarchy of the CUDA GPU. In this implementation, we take advantage of fine grain parallelism within an actor. To launch the filter execution, first the

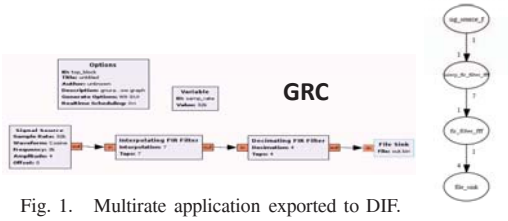


Fig. 1. Multirate application exported to DIF.

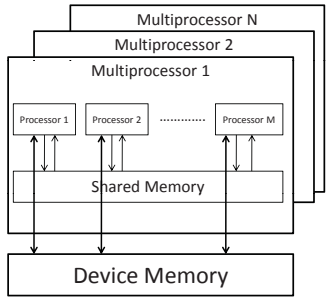


Fig. 2. GPU Memory Hierarchy.

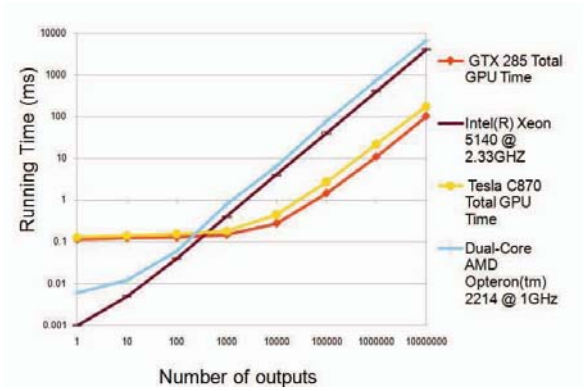


Fig. 3. Total running time versus number of outputs.

input data must be copied from the CPU to the GPU device memory. Then parallelism can be exploited by configuring each multiprocessor to run a number of threads equal to the number of filter taps. Initially, all of the threads will perform a load of a chunk of input elements to the shared memory within a multiprocessor. Then every thread will be responsible for calculating the product of a filter tap with an input, and adding this product to the partial sum of the output. Partial sums are stored in the shared memory in order to take advantage of fast communication between multiprocessor threads. After processing a chunk of inputs, the threads perform a block store of the calculated results to the GPU device memory.

Figure 3 compares FIR filter performance in terms of the execution times required for GPU and CPU implementations. Even though the GPU can compute more floating point operations per second than the CPU, using it becomes beneficial only if the amount of output required is large enough to compensate for the communication overhead between the CPU DRAM and the GPU device memory. This number (i.e., the number of processed outputs) can be added as an application attribute to help in deciding on actor-to-processing-unit assignment. The execution time per input token is another important attribute that not only takes into consideration the number of processing units but also the behavior of relevant implementation details.

## VI. Conclusion

In recent years, the growing capabilities of multicore processors have accelerated the trend towards software defined radio systems. Such systems are increasingly used to facilitate implementation and integration of new radio protocols without paying the cost of fabricating custom integrated circuits. Applying mutliprocessor scheduling techniques on dataflow models for such systems can reduce the gap between system development and optimized implementation. In this paper, preliminary steps towards building an application model and architecture model are taken for the development of a dataflow-based design methodology for GNU Radio. We have shown how to transform GNU Radio specifications to the dataflow interchange format (DIF) in order to apply DIF-based techniques for static, dynamic, and quasi-static scheduling. Also, we presented results on the amount of speedup and associated limitations associated with implementing a common SDR module on a CUDA GPU.

## References

[1] E. Blossom, "GNU radio: tools for exploring the radio frequency spectrum," *Linux Journal*, June 2004.

[2] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Transactions on Computers*, February 1987.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

[4] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs using the token flow model," in *In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.

[5] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.

[6] Y. Lin, R. Mullenix, M. Woh, S. Mahlke, and T. Mudge, "Spex: A programming language for software defined radio," in *Proc. of the SRD 06 Technical Conference and Product Exposition*, 2006.

[7] Y. Lin, M. Kudlur, S. Mahlke, and T. Mudge, "Hierarchical coarse-grained stream compilation for software defined radio," in *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM, 2007, pp. 115–124.

[8] H. Berg, C. Brunelli, and U. Lucking, "Analyzing models of computation for software defined radio applications," in *Proc. IEEE International Symposium on System-on-Chip*, Nov. 2008, pp. 1–4.

[9] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

[10] W. Plishker, N. Sane, and S. S. Bhattacharyya, "A generalized scheduling approach for dynamic dataflow applications," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, France, April 2009, pp. 111–116.

[11] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real time DSP," in *Proceedings of the Global Telecommunications Conference*, November 1989.