GRT: a Reconfigurable SDR Platform with High Performance and Usability *

Tao Wang *, Guangyu Sun, Jiahua Chen, Jian Gong, Haoyang Wu, Xiaoguang Li
Center for Energy-Efficient Computing and Applications, School of EECS, Peking University
{wangtao, chenjiahua, jian.gong, wuhaoyang, xiaoguangli2010}@pku.edu.cn

Songwu Lu *, Jason Cong *
UCLA Computer Science Department
{slu, cong}@cs.ucla.edu

* PKU-UCLA Joint Research Institute in Science and Engineering

ABSTRACT

The importance of software-defined radio (SDR) continues to increase. However, existing SDR platforms become less efficient as the wireless industry moves towards Gigabit WiFi. In this work, we propose a novel reconfigurable SDR platform named GRT. With the help of reconfigurable architecture and corresponding software support, SDR designs on GRT can leverage high performance of the underlying hardware and provide sufficient usability, including the support for efficient modular design, commodity interface, good programmability, code reusability, etc. We implement an 802.11a/g WiFi system on GRT to evaluate its performance. The results demonstrate that GRT can achieve a substantial improvement in usability while still satisfying the performance requirement.

1. INTRODUCTION

With the rapid development of wireless systems, there is an increasing requirement for fast validation and prototyping of various innovations in underlying wireless layers, such as the media access control layer (MAC) and physical layer (PHY), which are normally hidden and integrated firmly in conventional wireless adapters. Such a requirement, therefore, results in the rising importance and research interest in software-defined radio (SDR) systems. This is because SDR provides an opportunity for efficient modification and extension of algorithms and/or protocols in underlying wireless layers by programming. At the same time, SDR facilitates customized wireless systems that are tailored for extra low power, low latency, or long communication range. In addition, it enables new techniques such as the deep cross-layer optimization [14, 15, 18, 19, 20, 21, 22, 23].

Recently, various SDR platforms have been proposed [1, 3, 5]. However, as the wireless industry moves towards Gigabit WiFi, current SDR platforms cannot fully satisfy both performance and usability requirements at the same time. For example, GNU Radio [1], the most widely used SDR platform, provides high programmability and rich legacy codes, but its performance (around several Mbps throughput [2]), however, is lower than the industry standard by several orders of magnitude. Other SDR platforms may achieve higher throughput by compromising their usability [3, 5]. These limitations have impeded the adoption of SDRs in the research of state-of-art wireless systems.

*This paper is supported by National Natural Science Foundation of China (61370056, 61103028)

In order to overcome these limitations, we propose a reconfigurable SDR platform, named GRT. GRT can satisfy the performance requirement of wireless data processing by leveraging high performance of its underlying hardware. At the same time, its reconfigurable architecture design provides good programmability with corresponding software support. For any module design in SDR, GRT provides flexible switching between its hardware version and software version implementations. Thus, it enables module-level incremental refinement from legacy or newly designed software implementations (for fast development time) to their hardware counterparts (for high performance). In addition, a wireless system implementation on GRT can be directly used as a conventional wireless network adaptor without any changes of APIs at the application level. And the rich legacy codes of GNU Radio can be smoothly reused on GRT. Contributions of this paper are summarized as follows:

- We provide a comparison of existing SDR platforms and argue that the requirements of both performance and usability should be meet, which motivates our reconfigurable SDR platform named GRT.
- We present the architecture and corresponding software support of GRT and introduce how the design challenges are solved.
- We implement an 802.11a/g WiFi system in GRT to evaluate its performance. The evaluation demonstrates that the performance requirement is satisfied with a substantial improvement in usability.

The rest of this paper is organized as follows. In Section 2, we present a brief introduction to SDR and existing SDR platforms. Their advantages and limitations are discussed to motivate our proposal of GRT in Section 3. The architecture of GRT and software support of the framework are introduced in Section 4. In Section 5, we present an 802.11a/g WiFi implementation on GRT, followed by a conclusion in Section 6.

2. PRELIMINARIES

2.1 Background of SDR

SDR was first proposed as a programmable radio communication system, in which some components conventionally implemented in hardware are replaced by means of software. Recently, various alternative hardware, which include SIMD processors and FPGA, are proposed for high-performance SDR designs [8, 5, 7]. Generally speaking, an ideal SDR

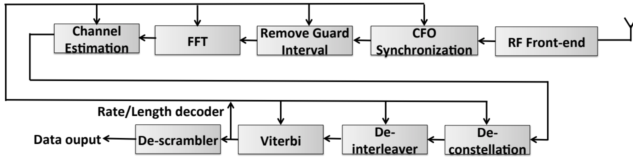

Figure 1: Illustration of the 802.11a/g PHY receiver

platform should satisfy the following goals.

- **High Performance**. In order to follow the state-of-art standard of wireless systems, both throughput and latency should be promised in an SDR design.
- **Efficient Modular Design**. It should support flexible creation/deletion/extension of any individual component with moderate design overhead.
- **Commodity Interface**. It should provide a convenient interface for upper network layers. For example, a wireless system designed on the platform could be directly used as a conventional wireless network adapter.
- **Good Programmability**. To reduce the learning curve, an SDR platform should be easy-to-program. Especially for SDR platforms that are developed for researchers in the wireless community, extraordinary programming skills are ungrateful.
- **Code Reusability**. To leverage the rich legacy codes from existing platforms (e.g., GNU Radio), it should provide sufficient compatibility.

Unfortunately, it is nontrivial to achieve these design goals due to some intrinsic obstacles of a wireless system. We take the PHY-layer receiver end of 802.11a/g as an example to explain the reasons. The structure of the receiver is illustrate in Figure 1. Among these modules, some (e.g., Viterbi) are dominated by bit-level sequential computation and can achieve a high clock frequency in hardware; while others (e.g., Channel-Estimation) contain massive parallel processing kernels to get high throughput with low achievable clock frequency. They should not work at the same clock frequency. Otherwise, the throughputs of the bit-level modules will be severely limited by the low clock frequencies of the massive parallel modules. Thus, multi-clock domains are needed in the design.

At the same time, for hardware module implementation, without a specially designed modular interconnect framework, a redesign of the whole system may be needed even for a minor revision to a single module, which is quite time-consuming and error-prone. Moreover, in a mixed software-hardware SDR design, the communication among these modules becomes a critical issue, especially when the SDR system is expected to work as a conventional wireless adapter. In addition, the legacy codes from GNU Radio are normally incompatible to those platforms based on the specific underlying hardware.

In summary, numerous design issues should be considered in the development of an SDR platform.

2.2 Existing SDR Platforms

GNU Radio [1] is the most widely used SDR platform. It is a pure software-based open-source platform. A computer running GNU Radio can be connected to an external radio frequency (RF) front-end device (e.g., USRP series [6]). There are rich legacy codes in GNU Radio. However, GNU Radio provides low throughput, mainly due to the slow digital processing speed of the general-purpose processors. It only supports data rates at the level of several Mbps, which is far behind the data rate requirement of real-world wireless standards nowadays.

Sora [3] is a software-dominant SDR platform on high-end multi-core machines running the Windows OS, with a special hardware aid in timing-intensive components. It maps modules of a PHY layer to various processor cores to achieve considerable performance. Sora satisfies the 54 Mbps data rate requirement of 802.11a/g. A recent two-pipeline 2x2 MIMO version of Sora [4] claimed to support a 117 Mbps data rate. However, due to the intrinsic process scheduling properties of Windows, Sora cannot fully guarantee the tight timing/real-time requirements of the MAC layer in WiFi standards. In addition, SDR development on Sora requires partitioning, mapping, and balancing of sub-tasks. Thus, special programming skills are needed.

SIMD processors have also been employed for SDR design to leverage the advantage of high-throughput processing [8]. Though SIMD processor-based platforms provide high programmability, the massive data parallel computing model is not suitable for all the wireless PHY modules. GPU-based platforms are too power-hungry for SDR systems.

WARP [5] is an FPGA-based SDR platform. A WARP platform can communicate with a host computer through Ethernet connections. The latest version of WARP equips a relatively powerful FPGA for signal processing. It can provide a higher processing capacity than software-based SDR platforms. The MIT AirBlue [7] is also an FPGA-based SDR platform. AirBlue observed the importance of modularity in programming wireless protocols. It provides two features, latency-insensitivity and data-driven control. Programming languages of WARP and AirBlue are HDL and Bluespec [9], respectively. They do not provide an efficient modularity design framework or specific support for programmability. And although the modules programmed in AirBlue can be interconnected with latency-insensitive FIFOs, all of them should be in the same clock domain, which may degrade the performance of the SDR system.

3. MOTIVATION

Software-based SDR platforms (including software on general-purpose, SIMD processors or GPU), though flexible, have intrinsic limitations to achieving the required throughput of high-speed wireless standards for two reasons. First, in wireless communication, both bit-level operations and massive parallel operations are critical in the signal processing stages, which are not suitable for these platforms. Thus it is difficult to achieve the state-of-art data rate requirement. Second, due to the scheduling algorithms/properties of the general-purpose OS, such as the units time intervals (1-10ms), interrupts, preemptive scheduling, etc., normal software-based SDR platforms cannot satisfy the strong timing requirement (1us precision). A specifically designed OS may help to satisfy the timing requirement, but the efforts of both building such an OS and implementing an SDR platform on top of it are less desirable.

An FPGA-aided SDR platform is able to meet the performance/timing requirement. FPGA is very suitable for both massive parallel computing and bit-level operations. Modern FPGAs can be clocked to several hundreds of MHz, so

a precise timing control is possible. However, the issue of programmability in FPGA should be considered. Though programming individual hardware modules is not very difficult for an experienced developer, the complexity increases significantly for integrating them into a high performance pipeline. Besides, in terms of usability, it is not easy to use the rich legacy software codes that the SDR community already has.

In order to meet the requirements of both performance and usability, we propose a reconfigurable SDR platform, GRT. It can satisfy the performance requirement of wireless data processing by leveraging high performance of its underlying hardware. At the same time, its reconfigurable architecture design provides good usability with corresponding software support.

4. DESIGN OF GRT

4.1 Design Challenges

4.1.1 Multiple Clock Domain

As mentioned in Subsection 2.1, multi-clock domains are needed for different hardware-based SDR modules to satisfy both throughput and latency requirements. Although it is possible to design a dedicated clock domain for each module, the overhead of asynchronous logic for communication among these clock domains is nontrivial. In addition, the implementation process is time-consuming and error-prone. Thus, the first challenge is how to provide an efficient method for designs with multiple clock domains in the SDR platform.

4.1.2 Modularity Related Issues

In an SDR platform, a convenient method should be provided to insert a new module, modify an existing module, and remove an obsolete module in the pipeline. In addition, switching between a hardware version and software version implementation of these modules is required to enable incremental refinement from convenient software implementations to their hardware counterparts.

However, it is not straightforward because various modules may have different input and output port requirements with different data widths, especially when communication between software and hardware modules is considered. It means that inserting/removing/replacing modules may result in the redesign of interconnects between modules, which is time-consuming and error-prone. These issues pose a design challenge in our proposed SDR platform, i.e., that of providing a support for the full modularity and interconnectivity of the software and hardware modules.

4.1.3 OS User/Kernel Cross-mode Communication

An SDR platform with high usability should provide a convenient network interface for the upper layers of the OS network stack so that a wireless system designed on the platform could be directly used as a conventional wireless network adaptor. According to the network stack structure in modern operating systems (e.g., Linux), and considering the efficient method of invoking hardware modules, the network interface provided by an SDR platform should be in the OS kernel mode. Some software modules in the SDR platform may also be in the OS kernel mode, such as the rate adaptation module at the high-MAC layer. Another example is the software end of the PCIe communication library, which is responsible for the communication between the host computer and the external SDR platform.

In addition, the requirement for code reusability/compatibility to the GNU Radio makes this problem even worse. Although a code-wrapper can be used to make the codes in hardware design compatible to those in GNU Radio at an certain level, a large gap exists because they reside in the OS kernel and user modes, separately. The communication between the modules in these two modes, as well as that between the modules and the OS network layer, is challenging.

4.2 GRT Architecture

In order to overcome the challenges discussed in Subsection 4.1, we propose an efficient reconfigurable SDR platform called GRT. Figure 2 illustrates its architecture.

4.2.1 Overview of GRT

As shown in Figure 2, the GRT system is composed of two main components: the software extension on the host and the reconfigurable logic (RL) hardware. The RL part consists of a global partition and several local partitions. The software part includes both OS user-level and kernel-level extensions.

An SDR researcher can develop his own software modules in either OS user level or kernel level and hardware modules on RL. In addition, GRT provides a tool called $ModuleGen$ to automatically generate the corresponding interconnect and bypass logic for the hardware modules on RL according to a configuration file. It facilitates the process of inserting/modifying/removing hardware modules in the pipeline. It means that SDR researchers only need to focus on the development of their own modules instead of worrying about the interconnect among the modules. Note that in GRT the modules can be programmed with HDL manually or generated from high-level source code (e.g., C codes) by high-level synthesis tools [12, 13].

4.2.2 Partition-based Architecture

Although different SDR modules may need different maximum clock frequencies, the total number of clock domains in the SDR platform should be controlled under a moderate level to reduce the complexity of synchronization across clock domains. Since the throughput of the whole pipeline is limited by the module with the lowest throughput, some modules do not have to work with their maximum working frequencies. Thus, one goal of partitioning is to select proper working frequencies for the modules.

For example, in our case study in Section 5, the *scrambler* module can be run under the frequency of 185 MHz with a throughput of 11840 Mbps; the BCC module can achieve 10800 Mbps under a frequency of 450Mhz. However, for *chn_esti* module, its maximum frequency is 35MHz with a maximum throughput of 291Mbps. Thus, it is feasible to group these modules together into the same clock domain with a 35MHz working frequency. All these modules in this clock domain can still achieve a throughput more than 125Mbps, which is the throughput of the whole pipeline limited by the module of $Viterbi$.

This grouping method can greatly reduce the number of required clock frequencies in an SDR system. At the same time, the cross-clock-domain components, such as asynchronous logic, which are several times more expensive than syn-

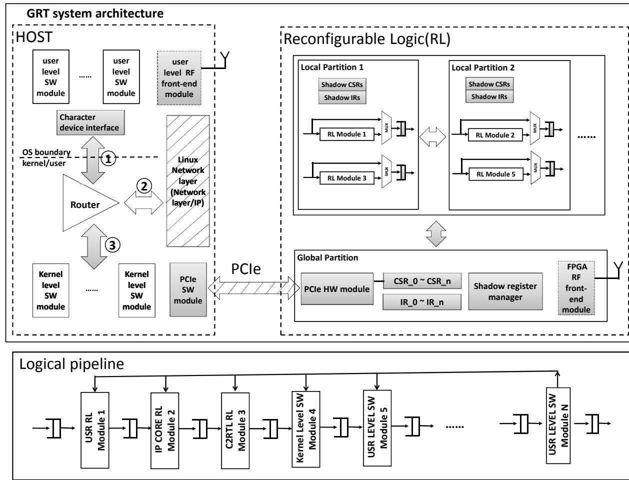

Figure 2: Illustration of the GRT architecture

chronous logic both in terms of latency and area, are no longer necessary for interconnecting the modules in the same partition (clock domain).

In GRT, the whole architecture is divided into a configurable number of partitions, each of which may have its unique clock frequency, as shown in Figure 2. The global partition is used to store the system-wide registers and contain the system modules/interfaces to the modules outside the RL part. Various numbers of local partitions with different clock domains can be created automatically or manually by the developers, with the help of ModuleGen.

4.2.3 Shadow Register

In addition to streaming data interfaces such as asynchronous FIFOs, communication among modules from different clock domains can also be realized by accessing an intermediate register. For example, a function module in a local partition would access a control and state register (CSR) in the global partition, which is set by the software side, to get some control instructions. Due to random access patterns of these registers, asynchronous FIFOs for streaming data transmission should not be employed. A proper method for cross-partition register access is needed.

We propose shadow registers and shadow register managers to facilitate the cross-partition register access. In GRT, the global partition contains all registers that are used for communication among different partitions. These registers are named global registers. After the communication among partitions is specified by the SDR developer, dedicated shadow registers are created in each partition. The shadow registers in a partition are logically mapped to a subset of global registers. Then, a shadow register manager, which is also created by the GRT system for that partition, is responsible for mapping the global registers into the shadow registers and guaranteeing the correctness during the cross-clock-domain data transmission process.

To this end, all modules in a partition can conveniently access the global registers by managing the corresponding local shadow registers. Since the modules and their local shadow registers are in the same clock domain, there is no extra effort needed for cross-clock-domain signal transmissions in a module design. And two modules from different partitions can communicate through the global registers by accessing their own shadow registers.

4.2.4 Interconnect and Bypass Logic

In GRT, we provide the automatic generation of interconnect and bypass logic, with the help of a configuration file (details in Subsection 4.3). A revision to the logic of a module does not require modification to the I/O interface.

When a new module is inserted, the GRT system automatically generates the interconnect logic for the modules according to the configuration file. When an obsolete module is to be removed, the developer can change the descriptions of the FIFO ports of its neighbor modules in the configuration file. Then, the GRT system will generate a new set of interconnect logic for those modules, which will be connected without the obsolete module.

A module may also need to be temporarily disabled without modifying and recompiling the pipeline. This scenario happens in the debugging process or when some selected modules are interconnected with some software modules to form a pipeline mixed with software and hardware modules. The GRT system can automatically generate the bypass logic for a module that needs to be temporarily disabled. The developer can simply indicate in the configuration file that a module may be temporarily disabled. Then, the GRT system will generate the bypass logic for that module during compiling. The bypass logic is controlled by a CSR and is activated when the software part of the GRT system sets the CSR (via the PCIe library).

4.2.5 Host-RL PCIe Library

In order to support an efficient mix software-hardware design, we provide a user-friendly streaming host-RL PCIe communication library in GRT [16]. This library supports low-latency CSR accesses and multiple streaming data channels between software modules and hardware modules. With such a library, the data can be directly transmitted between a software module and a hardware module in the streaming mode. The host-RL PCIe library was highly optimized and verified on various generations of Xilinx FPGAs, and archived a throughput of several tens of Gbps. We believe this library can fully support the aggressive data exchange requirement on SDR applications.

4.2.6 OS Kernel Module for Cross-mode Communication

To achieve efficient OS user/kernel cross-mode communication, we provide a new Linux OS kernel module, illustrated as the "router" in Figure 2, which supports a three-way communication. One direction is to the modules in the OS kernel mode. Those modules could be software modules in the OS kernel mode or the software part of the PCIe library for communicating to the hardware modules. The second direction is to the network layer (e.g., the IP layer) of the OS network stack, which is in the OS kernel mode, to exchange data between the GRT modules and the network layer .

The third direction is to an interface, which is provided by the router, accessible at the OS user mode. This interface is used to provide a data access channel for the software modules in the user mode. The router creates a pseudo character device as the interface. A software module in the user

mode can use read/write system calls to access the character device. Then, the data are actually transmitted across the OS mode boundary to the router. With the help of this OS kernel module, software modules in the user mode (① in Figure 2), software modules and the PCIe library in the kernel mode(③ in Figure 2), and the OS network layer in the kernel mode (② in Figure 2) can exchange data efficiently.

4.3 Software Support

In order to provide a good usability, GRT provides a tool named ModuleGen to facilitate the reconfiguration of the architecture. The input of this tool is a configuration file. In the configuration file, a developer can specify the number of partitions, the names of the partitions, the set of required CSRs in each partition, and some other information for generating the Verilog codes of the partitions. The developer can also specify which partition a module should be allocated to, the CSRs that the module should access, the input and output ports of the module. To generate the interconnect logic for a module, the configuration file also includes the neighbor modules that a module should be connected to. ModuleGen also generates the bypass logic for each module, which is activated when the module should be temporarily disabled.

To support an efficient modular design, we offer a C++/HDL co-design framework in GRT, with the help of the GRT host-RL PCIe library and a C++ wrapper for the RL modules. When a user designs his own pipeline, the modules in the pipeline can be a mixture of C++ codes for software modules and the HDL codes for hardware modules. A user can start her design by programming a module in C++, and then replace the software module with a hardware version. A hardware module can be manually programmed with HDL, or generated from C-to-HDL synthesis tools such as xPilot [12] or Xilinx AutoESL/Vivado HLS [13], following the same interface specification in GRT. In addition, we make the C++ wrapper have the same interface as that in GNU Radio for GNU Radio C++ building blocks; thus the rich legacy codes in GNU Radio can be reused.

5. 802.11A/G IMPLEMENTATION ON GRT

We implemented an 802.11a/g WiFi system to evaluate the efficiency of GRT. The reconfigurable architecture is implemented on an FPGA evaluation board Xilinx ML605 with a Xilinx Virtex-6 LX240T FPGA chip [10]. The board is connected to a host computer using the GRT PCIe library.

5.1 System Implementation

In order to evaluate the usability of the GRT system in a practical scenario, we implemented an 802.11a/g PHY and low-MAC system following the specification of maximum data rate (54Mbps). Note that this is not the peak performance that can be achieved by GRT. The maximum throughput of each module is discussed in Subsection 5.2.

To make the case generic and convenient for SDR developers, we chose to attach a radio frequency (RF) front-end device USRP N210 [6] to the Gigabit Ethernet port of the host computer using its official UHD driver.

In order to make the GRT system behave like a conventional wireless network adapter, we generate a network interface (grt0) for the GRT system in the OS and set an IP to the interface. Then, two computers with GRT systems can communicate through the air via their IPs of the GRT interfaces using a normal Linux wireless networking method. We successfully verified the correctness of the GRT system by using ping command, ssh, and TCP file transmission (socket programming).

Table 1: Comparison of the programmability

Case	GRT_raw	GRT_arch
+ Module	4 days	0.5 day
Pipeline	1 month	5 days

Table 2: Performance comparison on PHY modules

Major Modules	GRT_raw			GRT_arch		
	Fmax MHz	M.Th. Mbps	Slices %	Fpar MHz	P.Th. Mbps	Slices %
Viterbi	125	125	6.56	125	125	6.59
CFO_sync	85	230	1.38		230	1.38
FFT [17]	300	1016	9.58	85	287	9.60
iFFT [17]	300	1016	10.87		287	10.88
scrambler	185	11840	0.81		933	0.81
descrambler	210	13440	1.81		933	1.81
CRC_add	333	2664	0.47		280	0.47
CRC_check	303	2424	0.50		280	0.50
BCC	450	10800	0.07		840	0.07
interleav	400	7200	0.74	35	630	0.75
deinterleav	348	6264	0.75		630	0.77
constella	333	2997	0.30		315	0.30
deconstella	333	2997	0.88		315	0.90
chn_esti	35	291	3.42		291	3.42
whole_impl	N/A	125	60.16	N/A	125	60.64

We evaluate the programmability with the development period in two scenarios: 1) inserting a pre-written hardware module into the pipeline; 2) connecting all existing modules into a functional PHY pipeline. The comparison is based on implementations on two platforms. The first one is based on the GRT architecture (labeled as "GRT_arch") and the second one is a direct implementation on the same GRT hardware platform (labeled as "GRT_raw"). Both implementations are done by the same developer to ensure the same developing experience. Table 1 shows the results. We can see that the programmability is greatly improved with the GRT architecture.

5.2 Performance Evaluation of GRT

In this subsection, we compare the performances of the two implementations in GRT_raw and GRT_arch. Table 2 shows the clock frequencies, throughputs and FPGA slice occupations of the major PHY and low-MAC modules in these two implementations. We also show the data for the whole implementation (including the PCIe module and some other related modules).

From Table 2, we can see that the automatically generated interconnect logic and bypass logic have negligible impact on the resource occupations of the modules. And the total resource occupations by the interconnect logic, bypass logic, shadow register managers etc., which are introduced by the GRT architecture, are also negligible in the whole implementations (60.64% in GRT_arch vs. 60.16% in GRT_raw). The reason is twofold: 1) Even in GRT_raw, interconnect logic is also needed among modules; 2) In GRT_arch, within each partition, the relating modules are connected with syn-

Table 3: Latencies of PHY TX and RX sub-pipelines

Sub-pipeline	GRT_raw	GRT_arch
Transmitter (TX)	1.78 us	1.79 us
Receiver (RX)	30.98 us	31.84 us

chronous logic. The resource occupation of synchronous logic is much lower than that of corresponding asynchronous logic used by the modules with different clock frequencies.

The throughput of SDR on GRT_arch is also the same as that on GRT_raw. It is because the throughput of a pipeline is limited by the bottleneck module in the pipeline. The bottleneck modules on both GRT_arch and GRT_raw are the same. The automatically generated logic in GRT induces extra latency. Table 3 shows the latency comparison between GRT_arch and GRT_raw. We can find that the latency overhead is also negligible because of the replacement of asynchronous FIFOs with synchronous logic for the modules in the same partition. The benefits of the synchronous logic offset the latency overhead of the bypass logic.

We also find that the module *Viterbi* becomes the bottleneck in both implementations. This is limited by our current simple implementation of Viterbi. Note that a higher performance can be achieved by other implementations: for example, [11] shows a Viterbi implementation with a much higher performance. The reason that we did not include that implementation into our evaluation set is due to IP issues.

Pure software-based SDR platforms can not support the full PHY throughput of 802.11a/g (54 Mbps). Other kinds of SDR platforms, such as Warp and Sora, are able to support such throughput, as is GRT. However, as discussed, GRT provides a much higher performance potential (in terms of both throughput and latency) than Sora. And the current GRT implementation employs the same FPGA model as that of WARP, so GRT can provide the same level of performance of WARP, while GRT provides more usability/programmability at the same time.

6. CONCLUSION

To design an efficient SDR system, we propose a novel reconfigurable platform named GRT. The architecture and software support of GRT are carefully explored to handle the design challenges, which include multi-clock domain, modularity design, interconnect logic, cross-layer communication, etc. The experimental results show that GRT can provide both good usability and high performance.

7. REFERENCES

[1] GNU Radio, http://www.gnuradio.org/

[2] T. Schmid, O. Sekkat, M.B. Srivastava, "An Experimental Study of Network Performance Impact with Increased Latency in Software Defined Radios", in WiNETCH 2007

[3] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. Voelker, "Sora: High Performance Software Radio using General Purpose Multi-core Processors", in NSDI 2009

[4] J. FANG, Z. TAN, and K. TAN, Soft MIMO: A Software Radio Implementation of 802.11n Based on Sora Platform, in ICWMMN 2011

[5] WARP v3 Kit, http://mangocomm.com/products/kits/warp-v3-kit

[6] USRP Family of Products, https://www.ettus.com/product

[7] M.C. Ng, K.E. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan, Airblue: A System for Cross-Layer Wireless Protocol Development, in ANCS 2010

[8] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, SODA: A Low-power Architecture For Software Radio, in ISCA 2006

[9] R. Nikhil, Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications, in MEMOCODE 2004

[10] Xilinx Virtex-6 FPGA ML605 Evaluation Kit, http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm

[11] Xilinx LogiCORE IP Viterbi Decoder v8.0 Product Guide, http://www.xilinx.com/support/documentation/ip_documentation/viterbi/v8_0/pg027_viterbi_decoder.pdf

[12] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xPilot: A Platform-Based Behavioral Synthesis System" in SRC TechCon 2005.

[13] Xilinx High-Level Synthesis (HLS), http://www.xilinx.com/tools/autoesl_instructions.htm

[14] S.T. Aditya, and S. Katti, "FlexCast: Graceful Wireless Video Streaming," in MobiCom 2011

[15] S. Katti, S. Gollakota, and D. Katabi, "Embracing Wireless Interference: Analog Network Coding," in SIGCOMM 2007

[16] J. Gong, J. Chen, H. Wu, F. Ye, S. Lu, J. Cong, and T. Wang, "EPEE: An Efficient PCIe Communication Library with Easy-host-integration Property for FPGA Accelerators," in FPGA 2014

[17] Pipelined FFT/IFFT 64 points processor, http://opencores.org/project,pipelined_fft_64

[18] X. Wang, W. Huang, S. Wang, J. Zhang, C. Hu, "Delay and Capacity Tradeoff Analysis for MotionCast," IEEE/ACM Transactions on Networking, Vol. 19, no. 5, Oct 2011.

[19] W. Huang, Xinbing Wang, "Capacity Scaling of General Cognitive Networks," IEEE/ACM Transactions on Networking, vol 20, no. 5, 2012.

[20] C. Xu, L. Song, Z. Han, Q. Zhao, X. Wang, and B. Jiao, Efficient Resource Allocation for Device-to-Device Underlaying Networks using Combinatorial Auction, IEEE Journal on Selected Areas in Communications, special issue on Peer-to-Peer Networks, vol. 31, no. 9, 2013.

[21] T. Wang, L. Song, Z. Han, and B. Jiao, Dynamic Popular Content Distribution in Vehicular Networks using Coalition Formation Games, IEEE Journal on Selected Areas in Communications, special issue on Emerging Technologies, vol. 30, no. 9, 2013.

[22] C. Xu, L. Song, and Z. Han, Resource Management for Device-to-Device Underlay Communication, Springer Briefs in Computer Science, 2014.

[23] K.Chi, X. Jiang and S. Horiguchi, Joint Design of Network Coding and Transmission Rate Selection for Multihop Wireless Networks, IEEE Transactions on Vehicular Technology,Vol.59, No.5, 2010.