Improving Transport Design for WARP SDR Deployments

Krishna C. Garikipati
Dept. of EECS
University of Michigan–Ann Arbor
gkchai@eecs.umich.edu

Kang G. Shin
Dept. of EECS
University of Michigan–Ann Arbor
kgshin@eecs.umich.edu

ABSTRACT

Software-Defined radios (SDRs) are a popular platform for developing and implementing wireless protocols. Their basic architecture consists of radio front-ends hosted on an FGPA board, and a back-end processing host for running bulk of the signal processing in software. The two components are bridged, usually by an Ethernet or PCIe interface that transports the radio samples. In addition to the processing delay in software, SDRs may experience a non-negligible transport latency, for example, due to the limited Ethernet bandwidth.

Wireless-Access Research Platform (WARP) is one such SDR platform that has recently gained a lot of attention. Research prototypes deploying tens of WARP radios over the Ethernet have become a familiar sight. WARP's transport design, however, is inefficient due to its linear increase in transport latency with the number of radios. We propose modifications to improve the current design. First, we utilize functional parallelism to run the read/write operations of multiple WARP radios concurrently. Second, we propose a high-bandwidth link at the host in order to support the combined transfer rates resulting from the parallel transport to/from the radios. As a result, we achieve a significant reduction in the transport latency by scaling back the linear increase to a constant overhead.

Categories and Subject Descriptors

C.2.1 [**Computer-communications networks**]: Network Architecture and Design — *wireless communication*; C.3 [**Special-purpose and application-based systems**]: Signal processing systems

General Terms

Design, Experimentation, Performance

Keywords

Software-Defined Radios, WARP

SRIF'14, August 18, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2627788.2627789 .

1. INTRODUCTION

From their modest beginning as experimental prototypes, Software-Defined Radios (SDRs) have gained significant traction across industry, military and academia. Today, a wide range of open-source and commercial SDR platforms are available: USRP [1], WARP [2], SORA [3], OpenAirInterface [4], NI FlexRIO [5], Nutaq PicoSDR [6] etc. SDRs are used in the industry to model, test, simulate and develop next-generation wireless protocols. While in academia, they provide a research and educational tool for wireless experimentation, validation and demonstration of new applications.

SDR is a highly-flexible radio that realizes most of the processing, such as filtering, (de)modulation and (de)coding as well as the link layer operations in software, instead of a specialized hardware. Though the split between the hardware and software components, and the type of interfaces used may vary, the architecture of a typical SDR is more or less well-defined [3, 7]: a radio front-end with RF transceivers, ADC/DACs and down/up-converters embedded on an FPGA or DSP board; a host machine, usually a general-purpose processor (GPP), which implements signal-processing in software; and a bridge interface, such as PCIe, Ethernet or USB, to transport the radio samples between the host's memory and the radio. Examples of this architecture include the USRP hardware that functions with the popular GNU Radio [8] open-source software, and the WARP board that is controlled through the WARPLab framework [2], both using the Ethernet interface. On the other hand, in a non-GPP based SDR architecture, the PHY and MAC-layer processing is implemented in the FPGA board itself, either as an FPGA core or on separate CPU. Its examples are the WARP 802.11 reference design and the USRP Embedded hardware series, respectively. Note that while the PCIe interconnect is much faster than a Gigabit Ethernet link (PCIe 2.0 x8 runs at 32Gbps in a single direction), the restrictions on the bus length and the additional hardware have limited its adoption. The SORA and the Nutaq SDR platform, however, utilize the PCIe bus interface.

The WARP SDR along with its WARPLab framework has recently received a lot of attention. Particularly, as a suitable platform for large-scale radio setups such as the Argos [9], which was built using 16 WARP nodes and a total of 64 transmit antennas. However, due to the limited Ethernet bandwidth, the significant delay in reading the radio samples from the hardware buffers into the host's memory and vice-versa, has become a processing bottleneck. For example, in the current WARPLab release, the average read/write delay

in transferring 32K samples from 4 WARP nodes is around 5ms — which is much greater than the 16us turnaround time of WLAN protocols, and even more than the 3ms processing deadline of LTE.

In this paper, we concern ourselves with only the transport latency of large WARP deployments (processing latency being the other aspect). The requirement of a large number of WARP nodes arises mainly from wireless technologies like Massive MIMO [10]. Already, prototypes with tens if not hundreds of transmit antennas have been realized [9, 11], while many others are still under development. While the processing latency is another important component that requires attention, we claim that, ultimately, it is dictated by the processing framework and its speed. For instance, replacing the MATLAB base in WARP framework with optimized C/C++ routines can provide a significant speed-up.

Our goal is to mitigate the high transport latency, a linear increase, seen when exchanging radio samples with multiple WARP nodes. Towards this, we propose our software and hardware design for WARP transport; the software code is available in [12] under the terms of the WARP open source license. As we demonstrate in this work, our design is readily applicable to the existing WARP software release and the current version of WARP hardware.

The rest of the paper proceeds as follows: §2 gives the motivation for improving the transport design while §3 specifies our SDR platform. §4 describes the transport functions and the performance of the existing design. §5 describes our proposed software and network design and its evaluation results are shown in §6. Finally, we conclude the paper with §7.

2. MOTIVATION

Wireless standards have stringent demands on latency — end-to-end and turnaround timing constraints are required to ensure the correctness of MAC protocols. Conventional radios that are built on DSPs or ASICs easily satisfy these requirements as they run at *hardware* speeds. Most SDRs on the other hand (with the exception of SORA), don't meet the timing constraints because of the variable delay of software-processing [7], and due to the non-negligible transport delay between the host and the radio front-ends [13].

Reducing the transport delay, and thus the overall processing delay is beneficial for a number of reasons. Argos [9] like systems can be studied in more diverse wireless environments, such as mobile and vehicular channels, where channel measurements need to be made within the coherence period ($<$ 10ms). Moreover, channel variations and channel feedback in large MIMO systems can be evaluated with greater detail [14]. More importantly, many research results, for e.g. [15], that are based on trace-based offline evaluation of the wireless channel can be emulated online. Not only communication networks, but wireless-based indoor-localization systems [16] built using WARP antenna arrays also stand to benefit from the processing speed-up.

3. PLATFORM DESCRIPTION

This section describes the WARP SDR platform, its signal-processing framework and the details of our testbed setup.

3.1 WARP

A product of the research work by Rice University, WARP is a scalable and programmable SDR platform. The latest WARP v3 hardware integrates a Virtex-6 FPGA and two RF interfaces on each board. It uses two MAX2829 dual-band RF transceivers covering both 2.4GHz and 5GHz ISM bands, with a maximum 40MHz RF bandwidth and a support for shared clocking. Additionally, it contains 12-bit ADCs/DACs, an FMC expansion slot and two gigabit Ethernet ports. An attractive feature of WARP is its extensibility through readily available add-on modules: FMC module for two additional RF interfaces, and a clock module to interface with external clock sources. Unlike most other SDRs, the large number of logic slices on WARP FPGA make it possible to implement the wireless protocol stack, such as 802.11 g/n PHY and MAC, on the FPGA itself. Though time-consuming to develop, the FPGA implementation lends real-time capability to WARP allowing it to function in standalone mode with real wireless deployments.

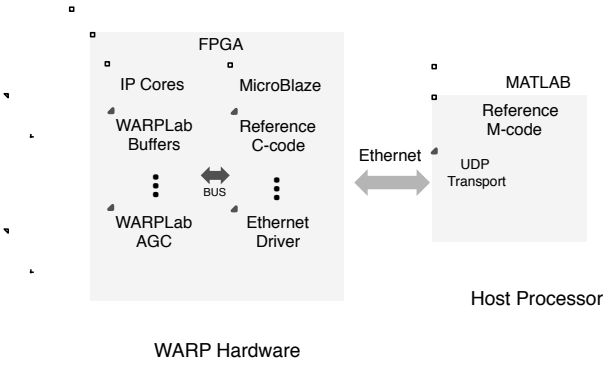

Figure 1: An illustration of the WARPLab architecture

WARP also serves as an ideal platform for MIMO implementation as it supports multiple RF interfaces, shared clocking and synchronization. Since each WARP node supports a maximum of only four antennas, building a massive-MIMO like system from WARP needs a large deployment of nodes. Therefore, the task of managing these nodes, which includes controlling the radio settings such as channel selection, Gain parameters, Tx/Rx duration, offsets, trigger modes etc., as well as the Tx /Rx configuration and coordination, requires a well-defined framework. This is discussed in the following.

3.2 WARPLab Framework

WARPLab is a flexible framework for developing wireless applications with a large array of WARP nodes. It utilizes the WARP hardware as RF transceiver entities (whose components include Amplifiers, Antennas, DAC/ADCs and Filters,) while the baseband (signal) processing is carried out at the host. An Ethernet interface connects the nodes and the host, and is used for carrying the baseband samples. Its instantiation, the WARPLab Reference Design, supplies a software library in MATLAB that provides simple user-level commands for coordination, configuration and control of WARP nodes. The library contains several modules: Node, Baseband, Interface, Transport and Trigger. Each of the modules acts as an interface to manage the corresponding hardware component. Among these, the transport module is a base module which is responsible for handling messages to and from the WARP hardware. WARPLab also provides

a FPGA reference design with custom IP cores (WARPLab Buffers, WARPLab AGC etc.) for the WARP hardware. In addition, it contains a C software design that runs on the Microblaze soft processor. The latter is required for Ethernet based control of the different hardware modules. Fig. 1 illustrates the described WARPLab framework. Given the abstraction of the hardware complexityand the easy-to-use programming interface, WARPLab is suited for rapid development of physical-layer algorithms.

The MATLAB's ease of use, however, comes at the cost of high delay in executing code, and reading/writing samples from the WARP nodes. As mentioned earlier, one solution to speed-up processing is to use the FPGA design as it is and migrate the software processing on the host to a compiled language like C/C++. This approach requires minimal modification since the current transport library, which is written in C can be reused, while the WARPLab user commands just have to be rewritten. But as we report in this paper, the transport design remains highly inefficient which results in excessive transfer delays as we increase the number of WARP radios.

In summary, WARP and WARPLab provide a self-contained SDR framework in that it allows a user to develop, deploy, transmit, receive and analyze a full-scale wireless communication system. Though limited to offline processing of baseband (I and Q) samples, it finds extensive use in prototyping a number of wireless applications, from MIMO [9] to indoor localization [16].

3.3 Testbed setup

Our experimental setup consists of 16 WARP v3 boards, each with 2 RF interfaces and connected through gigabit Ethernet. We use an HP ProCurve 6600 series Switch that provides 48 1GbE and 4 10GbE ports. The host processor is a hyper-threading enabled 32-core Intel(R) Xeon(R) E5-2660 CPU Linux machine with 128GB RAM. It has a dual-port 10GbE card, in addition to standard Gigabit Ethernet cards. We use the WARPLab 7.4 release, MATLAB 2012b and an Ubuntu 12.04 LTS operating system.

4. WARP TRANSPORT

A major design challenge for the WARPLab framework is to provide an efficient transport mechanism for moving I and Q samples between WARP buffers and the host machine's userspace over the Ethernet. Note that the transport of the rest of the control messages, for e.g. baseband and interface commands, is not considered as they require only few Ethernet frames. To keep matters simple, WARPLab implements packet buffer transfers (2^{15} samples is the maximum Tx/Rx buffer capacity per radio interface) instead of streams. This implies that the fixed-size I and Q buffer can be encoded into fixed number of Ethernet frames and sent from the source port to the destination port, which is the central idea of WARP transport. While in essence the basic transport methods are only *read* and *write*, which take the number of samples as an input parameter and return once the transfer is complete, this abstraction however hides the underlying complexity. In what follows we describe the WARP transport and its latency evaluation in detail.

4.1 UDP Protocol

To maximize throughput, WARPLAB uses the UDP transport protocol between the Host PC and WARP nodes. The source code for the host side is available as a self-contained C-based MEX file `wl_mex_udp_transport.c`, which needs to be compiled and built into the MATLAB environment. At the WARP node, the transport methods are provided in `wl_transport.c`, which further relies on the Xilnet library for socket operations. The UDP connection is established for each of the WARP nodes during initialization. A control layer on top of the UDP transport is further provided to guarantee reliable delivery of data. UDP packets are stamped with sequence numbers and checksums are appended. Correspondingly, routines for timeout, acknowledgement, checksum calculation and retransmission are provided.

4.2 Transport latency

The transport latency for a given node and a given number of samples is defined as the total delay in executing the transport function call in the userspace. In case of WARPLab, the userspace is the MATLAB environment. For instance, the read latency is the sum of the processing delay at the host in issuing the read command, the transfer delay of the read message, the turnaround time of the WARP hardware, the transfer delay of the Ethernet frames carrying the baseband samples, and finally the processing delay at the host in passing on the contents to the userspace. The turnaround time of the WARP hardware is further determined by the delay of Ethernet functions running on the MicroBlaze processor and the DMA (Direct Memory Access) transfer of WARPLab buffers. Clearly, the major component of transport latency is the transfer of baseband buffer samples over the Ethernet.

As one measure to reduce transport latency, WARPLab suggests using jumbo frames (MTU $\geq$ 9000B) wherever possible. While the default configuration specifies 1464 bytes (1508-byte Ethernet packet) as the maximum WARP transport payload, jumbo frames carry a 8960-byte (9004-byte Ethernet packet) payload in each packet. It is well known that for large Ethernet transfers, jumbo frames achieve higher throughput, and can be attributed to a smaller sender/receiver overhead (parsing of header, memory transfer, etc.) per byte transferred.

A quick back-of-the-envelope calculation reveals that without any overheads the maximum transfer rate of a 16-bit I and 16-bit Q sample stream over a 1Gbps Ethernet link is 31.2 Msps (samples per second). In contrast, the ADC/-DACs of WARP hardware can operate at more than hundred Msps. Even with the DMA transfer overhead, each WARP board still generates/consumes at a much faster rate than the capacity of the 1Gbps Ethernet link. This suggests that the Ethernet link is the bottleneck of WARP transport.

To investigate if this is indeed the case, we measure the network performance of a single WARP node connected to our server. Table 1 shows the raw packet throughput measured at the host in terms of packets captured per second (pps) and the bytes transferred. The resulting transfer rate that is measured from the number of WARP payload frames is also shown. The first point to note is that jumbo frames achieve almost 3$\times$ higher throughput. Second, with jumbo frames enabled, both read and write functions run close to the maximum 1Gbps line rate, and are hence limited by the Ethernet speed.

Next, we focus on the transport latency as the number of WARP nodes is increased. Fig. 2 shows the total latency of

Function	Packet Size (bytes)	#Samples	Throughput (Kpps)	Throughput (Mbps)	#calls (per sec)	Transfer rate (Msps)
Read	1508	32K	30.83	373.2	193.3	6.3
Read	9004	32K	13.57	972.8	314.2	10.3
Write	1508	32K	9.8	118.4	71.1	2.3
Write	9004	32K	13.67	979.9	336.7	11.0

Table 1: Raw packet throughput and the resulting transfer rate of the transport functions for a single WARP radio. Both read and write methods saturate the 1Gbps Ethernet link.

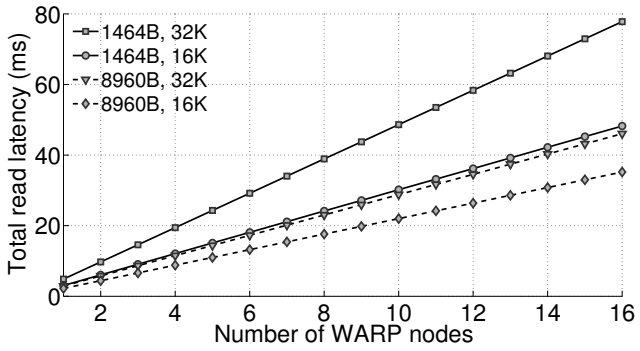

Figure 2: Measured total delay in reading with WARPLab 7.4 (Link speed = 1Gbps)

reading (values for write are similar) a single IQ buffer from the WARP nodes at different payload and sample sizes. For each configuration, 1000 measurements are made and the average value is reported. The variance was observed to be negligible. Since WARPLab transport is based on serial execution, the total delay is linear in the number of nodes. On an average each node takes around 4.8ms for reading 32K samples with jumbo frames disabled, and as a result the overall delay exceeds 75ms in case of 16 nodes.

As explained previously, jumbo frames reduces the transport latency, as seen in Fig. 2. However, the gains are insufficient and the latency is still linear in the number of nodes. This motivates us to consider an alternative transport design: reading/writing samples from multiple nodes concurrently rather than sequentially.

It is worth pointing out that the current release of WARPLab 7.4, on which our results are based is already an improved version of the earlier releases that were based on JAVA or PNET implementation, and did not utilize DMA for memory transfers.

5. PROPOSED DESIGN

We present an improved transport design for WARP SDR that is achieved with two modifications. First, a modified WARP transport library is implemented for parallel execution of read and write operations. Second, to achieve full benefits of parallelism, a high-bandwidth link, for example 10Gbps, is installed between the host and the switch.

5.1 Transport Parallelism

We base our design on the WARPLab UDP transport source code `wl_mex_udp_transport.c`. Though it is available as MEX file, we rewrote some of its methods so that our library compiles and runs in a standard C/C++ environment, independent of MATLAB. Thus, we have a standalone driver to interface with the WARP radios. On the other hand, the FPGA hardware design is unchanged and we use the WARPLab reference hardware design as it is.

We focus on implementing only the core transport commands. Additional control commands for AGC control, channel settings, buffer enable, etc. are obvious extension targets of our future implementation. However, this poses no problem in evaluation since the control commands are required only during initialization.

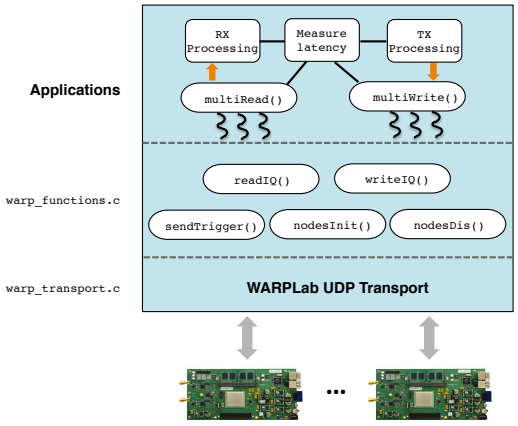

Figure 3: High-level organization of our code

Code description: Based on the WARPLab UDP transport methods, we design a standalone base transport script `warp_transport.c` which provides an interface for all message exchanges with the WARP nodes. It handles the socket creation, header creation, UDP functions etc. This is extended by the script `warp_functions.c` that abstracts the two transport methods `readIQ()` and `writeIQ()` with the following declaration:

```
void readIQ(double complex* samples, int
    start_sample, int num_samples, int
    node_sock, int node_id, int buffer_id,
    int host_id);
```

As seen from the declaration, the transport methods are defined w.r.t. single WARP node_id and buffer_id, and baseband samples are specified as an array of complex numbers. In addition, it provides methods for initializing nodes (sockets) and for disabling them. Further, the function `sendTrigger()` is used in broadcasting the *sync* packet to trigger the nodes to start their transmit/receive chain.

Transport Parallelism: In the current WARPLab implementation, reading or writing to multiple WARP nodes is done serially. But since the data — sample array, socket handle, headers, etc.— used in the transport call of each

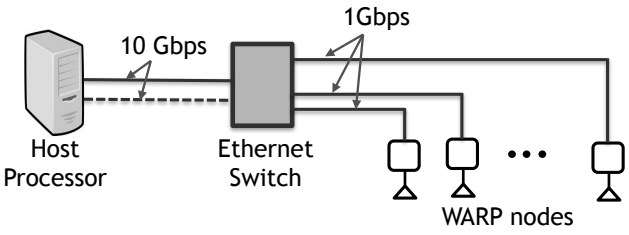

Figure 4: Proposed network design for deployment of 16 WARP nodes. An additional 10Gbps connection may be utilized to reduce the queuing delay.

WARP node is independent, we may readily apply functional parallelism by utilizing the multiple cores available on the host processor. By assigning a separate thread (each usually bound to a CPU core) for each WARP node (or buffer), the read/write operations can be run concurrently as long as the network interface on the host supports the resulting transfer rate. Implicitly this assumes that we use large enough send and receive socket buffers to prevent buffer overflow at the host.

To this end, we use the OpenMP API [17] that provides multiprocessor programming extensions for C/C++. OpenMP uses the `#pragma` directives to mark section of code that is to be parallelized. It also provides the loop construct, *omp for*, to split loop iterations among the threads where each iteration can be run independently. We use this loop construct while executing the `readIQ()` and `writeIQ()` functions of multiple WARP nodes so that they run in parallel.

5.2 Network Design

From the throughput values in Table 1 we can conclude that a single buffer transfer nearly saturates the 1Gbps link, which implies that a 1Gbps connection at the host can support only one WARP node. Therefore, to avoid the congestion delay when simultaneously reading or writing from multiple nodes, we require the host link to have a much larger bandwidth (>> 1Gbps). We achieve this by using the 10GbE ports on the server and the switch to create a 10Gbps connection, while the WARP nodes are connected to the switch with 1Gbps links. Fig. 4 illustrates the proposed network design.

Beyond 10Gbps: Ideally, assuming each WARP node saturates its own link, the 10Gbps bandwidth at the host should support the combined transfer rates of up to 10 nodes. However, beyond 10 nodes, there will be a queuing (congestion) delay. To overcome this, we use the second port on the 10GbE card to install an additional 10Gbps link between the host and the switch. The second interface is given a separate IP within the WARP subnet.

Since there are 16 nodes in our setup, traffic is routed so that each 10Gbps link is dedicated to a group of 8 WARP nodes. This is achieved by adding static route entries in the host's kernel routing table.

WARPLab currently supports a single IP for the host. However, we find that the addition of a new network interface at the host requires minimal changes to the transport code. This is because every response packet of the WARP node reuses the Ethernet header of the sent packet. In addition to that, transport methods function at the transport layer and are independent of the network interface used for routing.

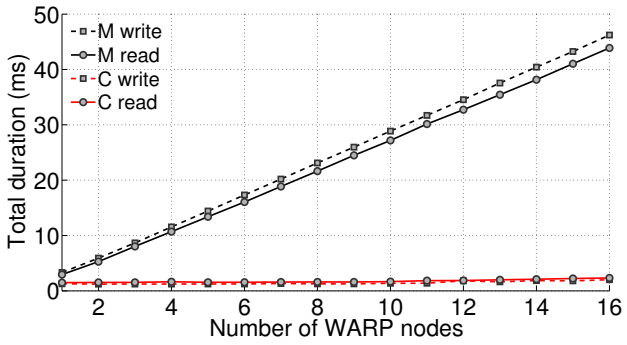

Figure 5: Comparison of total read and write delay of WARP nodes in MATLAB and the proposed C implementation. Host link speed is 10Gbps.

6. EVALUATION

This section evaluates the performance of the proposed transport design in our testbed setup.

In Fig. 5, we show the total latency of read and write methods in the MATLAB (WARPLab) implementation compared with our proposed C-based implementation.The reported values are the average value of 1000 runs. The total duration here refers to the total time it takes for the read or write function call of all nodes to deliver I and Q samples to the userspace. As shown in Fig. 3, we have a written a separate module to measure the latency of our multi-threaded implementation, and is based on the `clock_gettime` timer function. The buffer size is set to 32K samples while the jumbo frames are enabled to maximize throughput. We use the default OpenMP settings where the number of threads is set equal to the number of cores (32) on our server. The experiments are run when all the cores are lightly loaded.

Two observations can be made from our results. First, implementing transport methods directly in C is more efficient. Even for a single node, the average write delay is 1.5ms, which is less than half of the 3.3ms write delay in MATLAB. This is attributed to the fact that for data structure operations like creating headers, sample arrays etc., MATLAB/MEX-based implementation is slower than a pure C program.

Second, the total transport delay using parallelism shows negligible increase with the number of nodes—total write delay increases from 1.5ms to 1.61ms; total read delay increases from 1.74ms to 2.55ms. In contrast, in the MATLAB implementation the delay for both read and write increases linearly from around 3ms to almost 45ms.

We also evaluate the delay performance with an additional 10Gbps link at the host. As we vary the number of WARP nodes, we equally split the traffic among the 2×10Gbps links. Fig. 6 shows the read and write delays in transferring 32K samples with single and double 10Gbps links. As explained earlier, a single 10Gbps link may result in a queuing (congestion) delay at the switch. This is confirmed by the almost linear increase in read/write delay between 10 and 16 nodes which is in contrast to the almost constant behavior between 1 and 10 nodes. In Fig. 7, we again compare the total transport delay, but for a reduced buffer size

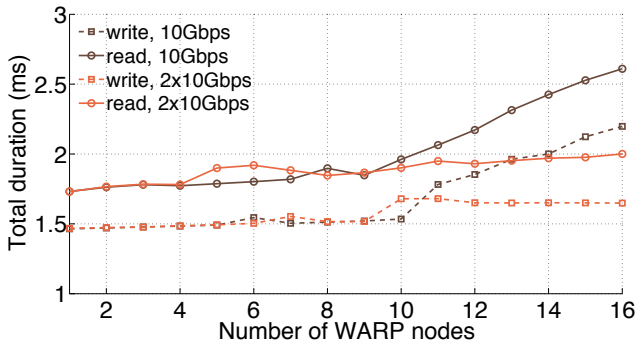

Figure 6: Total transport delay for 32K samples with single and double 10Gbps links. The improvement in the delay performance is noticeable beyond 10 nodes.

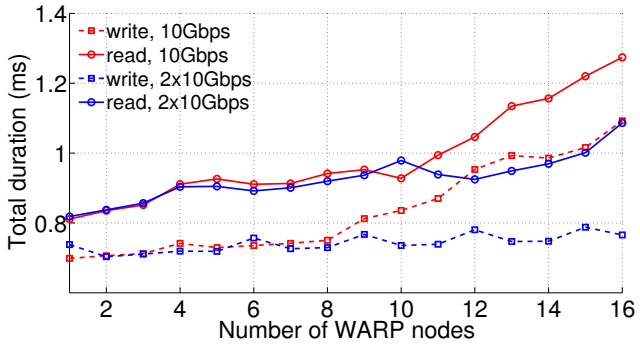

Figure 7: Total transport delay for 16K samples with single and double 10Gbps links.

of 16K samples. The observations are again consistent with our explanation.

The gains from the additional 10Gbps link are more prominent as the number of nodes is increased. In case of 16 nodes and 32K samples per node, the total read delay drops 24% from 2.6ms to 2ms while the total write delay goes down by almost 28%, from 2.2ms to 1.6ms. Adding more bandwidth between the host and the switch, therefore, is beneficial to the delay performance.

7. CONCLUSION

WARP deployments are constrained by the limited Ethernet bandwidth that results in a non-negligible delay in transporting radio samples. More importantly, this delay increases linearly with the number of the WARP nodes. In this paper, we showed that executing the read/write methods in parallel decreases the average delay of transferring 32K samples from 16 WARP nodes, from 45ms to under 2.5ms. Our implementation uses a 10Gbps Ethernet connection to the host processor to support the combined transfer rate of the nodes.

The improved delay performance enables new scenarios of experimentation for large-scale WARP deployments, for example, measuring the wireless capacity of a massive-MIMO system in mobile wireless channels.

Acknowledgments

The work reported in this paper was supported in part by the NSF under Grants 1160775 and 1317411.

8. REFERENCES

[1] "Universal Software Radio Peripheral." `http://ettus.com/`.

[2] "WARP Project." `http://warpproject.org`.

[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 *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[4] "OpenAirInterface." `http://openairinterface.org/`.

[5] "National Instruments FlexRIO SDR." `http://www.ni.com/sdr/`.

[6] "Nutaq PicoSDR." `http://nutaq.com/en/products/picosdr`.

[7] G. Nychis, T. Hottelier, Z. Yang, S. Seshan, and P. Steenkiste, "Enabling MAC Protocol Implementations on Software-defined Radios," in *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[8] "GNU Radio." `http://gnuradio.org`.

[9] C. Shepard, H. Yu, N. Anand, E. Li, T. Marzetta, R. Yang, and L. Zhong, "Argos: Practical many-antenna base stations," in *Proc. of ACM MOBICOM*, 2012.

[10] F. Rusek, D. Persson, B. K. Lau, E. Larsson, T. Marzetta, O. Edfors, and F. Tufvesson, "Scaling Up MIMO: Opportunities and Challenges with Very Large Arrays," *IEEE Signal Processing Magazine*, vol. 30, pp. 40–60, Jan 2013.

[11] Q. Yang, X. Li, H. Yao, J. Fang, K. Tan, W. Hu, J. Zhang, and Y. Zhang, "Bigstation: Enabling scalable real-time signal processing in large scale MU-MIMO system," in *Proc. of ACM SIGCOMM*, 2013.

[12] "CWARP." `https://github.com/gkchai/cwarp`.

[13] T. Schmid, L. Sekkat, and M. Srivastava, "An Experimental Study of Network Performance Impact of Increased Latency in Software Defined Radios," in *Proc. of the ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization (WINTECH)*, 2007.

[14] K. C. Garikipati and K. G. Shin, "Measurement-Based Transmission Schemes for Network MIMO," in *Proc. of ACM MOBIHOC*, 2014.

[15] X. Zhang, K. Sundaresan, M. Khojastepour, S. Rangarajan, and K. Shin, "NEMOx: Clustered Network MIMO for Wireless Networks," in *Proc. of ACM on Mobile Computing and Networking (MOBICOM)*, 2013.

[16] J. Xiong and K. Jamieson, "Arraytrack: A fine-grained indoor location system," in *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[17] "OpenMP API ." `http://openmp.org`.