# Exploring RapidIO Technology Within a DAQ System Event Building Network

Simaolhoda Baymani, Konstantinos Alexopoulos, and Sébastien Valat

***Abstract*—RapidIO (http://rapidio.org/) technology is a packet-switched high-performance fabric, which has been under active development since 1997. The technology is used in all 4G/LTE base stations worldwide. RapidIO is also used in embedded systems that require high reliability, low latency, and deterministic operations in a heterogeneous environment. RapidIO has several offloading features in hardware, therefore relieving the CPUs from time- and power-consuming work. Most importantly, it allows for remote direct memory access and thus zero-copy data transfer. In addition, it lends itself readily to integration with field-programmable gate arrays. In this paper, we investigate RapidIO as a technology for high-speed data acquisition (DAQ) networks, in particular the DAQ system of an LHC experiment. We present measurements using a generic multiprotocol event-building emulation tool that was developed for the LHCb experiment. Event building using a local area network, such as the one foreseen for the future LHCb DAQ, puts heavy requirements on the underlying network as all data sources from the collider will want to send to the same destinations at the same time. This may lead to an instantaneous overcommitment of the output buffers of the switches. We will present results from implementing an event building cluster based on RapidIO interconnect, focusing on the bandwidth capabilities of the technology as well as its scalability.**

***Index Terms*—Communication systems, data acquisition-protocol independent performance evaluator (DAQPIPE), DAQ networks, data analysis, interconnected systems, RapidIO, ROOT, scalability.**

## I. Introduction

RAPIDIO [1] is a high-performance, low pin count, packet-switched system-level interconnect standard. Initially intended to be a front side bus, RapidIO developed into a system-level interconnect. Since its first specification in 1999, over 200 million RapidIO fabric ports have been deployed worldwide. The latest specification was released in June 2016, supporting 100 Gbps networks. RapidIO has foremost been used within telecommunications, but also as part of space and industry applications. It is mainly utilized within embedded systems in chip-to-chip and board-to-board communications, but on account of a number of appealing features, such as low latency combined with scalability, it also lends itself well in other contexts, such as a larger scale fabric in high-performance computing (Fig. 1).

Manuscript received June 24, 2016; revised April 11, 2017 and June 1, 2017; accepted July 1, 2017. Date of publication August 1, 2017; date of current version September 14, 2017.

S. Baymani is with the IT Department, European Organization for Nuclear Research, CERN, CH-1211 Geneva, Switzerland (e-mail: sima.baymani@cern.ch).

K. Alexopoulos is with the IT Department, European Organization for Nuclear Research, CERN, CH-1211 Geneva, Switzerland, and also with the Electrical and Computer Engineering Department, National Technical University of Athens, 15780 Zografou, Greece.

S. Valat is with the Experimental Physics Department, European Organization for Nuclear Research, CERN, CH-1211 Geneva, Switzerland.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNS.2017.2734564

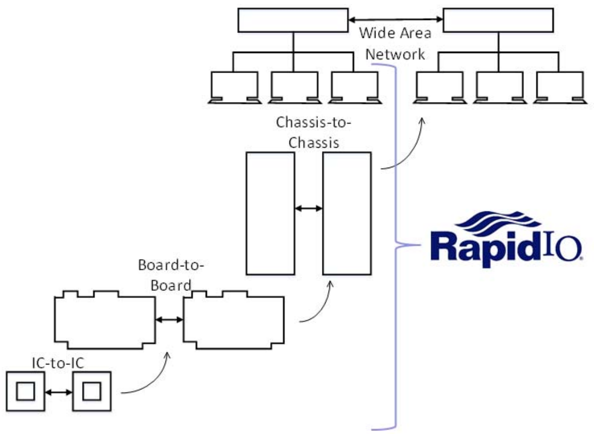

Fig. 1. Interconnect application domains where RapidIO is applicable [8].

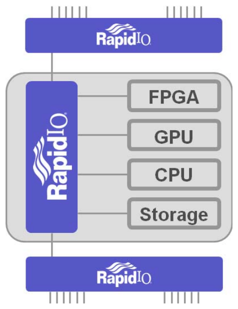

Fig. 2. Different types of devices may connect on the same RapidIO fabric. *Image source: IDT, Inc.*

The protocol specification allows the design of products that seamlessly integrate with multiple technologies. RapidIO exists natively on-chip, as well as on PCIe bridge cards, supporting a heterogeneous fabric capable of interconnecting CPUs, GPUs, storage units, and field-programmable gate arrays (Fig. 2).

RapidIO is primarily hardware implemented, offering error handling at the physical level as well as hardware termination. This enables low latencies and also offloads the CPU, permitting the scale-up of a realtime system. In addition, the RapidIO protocol supports destination-based routing, as well as the deployment of any network topology. Being an open standard [2], the RapidIO technology is supported on a number of diverse chips and cards for various applications.

Recently the commercial community around RapidIO has become interested in branching out of the known fields of application, and assesses RapidIO technology in a context outside of its current settings. Our particular project aims to explore RapidIO in different domains of networking applications. For this purpose, ROOT [3] and data acquisition protocol-independent performance evaluator (DAQPIPE) [4] were selected as suitable candidates. In this paper, we will present the work done in their respective fields of data analytics and DAQ.

The structure of the rest of this paper is organized as follows. Section II describes previous work utilizing the RapidIO interconnect. Section III gives an introduction to the RapidIO protocol and our software and hardware stack. Sections IV and V explain implementation details and provide the respective results and conclusions for the ROOT proof of concept and DAQPIPE ports to RapidIO. Section VI provides final conclusions drawn from the project as well as directions for future work.

## II. Related Work

The main reference to RapidIO is the set of specifications released by RapidIO.org [2]. In contrast to other comparable interconnects with previous work [5], such as Ethernet, Infiniband [6], and PCIe, the volume of academic research conducted on RapidIO is relatively small. There most probably is a vast amount of proprietary resources that are neither publicly accessible nor referable.

Relevant commercial white papers are [7], providing an overview and commentary on RapidIO and [8] that compares RapidIO with Ethernet. Although with focus on embedded systems, the latter has the closest resemblance to what we are interested in. The conclusion is that Ethernet dominates scaled-up systems, but depending on the requirements, RapidIO may be a viable choice.

Previously published work conducted on RapidIO has primarily focused on the field of embedded systems, foremost from a native (on-chip) perspective. In [9], RapidIO is evaluated as the interconnect for a space-based radar system with high requirements on real time and throughput capabilities. Published later, but also in the field of radar systems, Changrui *et al.* [10] presents a RapidIO- based heterogeneous signal processing system. RapidIO endpoint IP is investigated in [11], focusing on power consumption and [12], for use in RISC-V cores. Notwithstanding the fact that RapidIO is widely used commercially, the VIRGO collaboration [13] serves as an example of RapidIO applied to research projects, used for board-to-board communication for real-time control systems.

In summary, most work on RapidIO focuses on strictly embedded or hardware systems and designs. Investigations on RapidIO in computer network (LAN) contexts are scarce; to the authors' knowledge, there has been no work made in the field of data analysis nor DAQ related to RapidIO. Related work in DAQ or data analysis with alternative interconnects has not been referenced here, as in the context of this paper we are not comparing RapidIO to other established technologies such as Ethernet and Infiniband. The purpose of this paper is to evaluate this interconnect in a new context and establish its suitability in new domains.

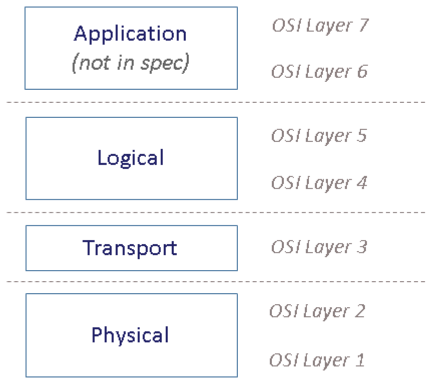

Fig. 3. RapidIO protocol specification layers and their corresponding OSI layers.

## III. Background

In this section, we present background on the RapidIO protocol as well as functionality offered by the library implementation made available to our project. Moreover, a description of the hardware setup used is given.

### A. Protocol

The RapidIO protocol is based on request and response transactions. The communication elements between the various endpoint devices in the system are packets. Control symbols are used for packet acknowledgment, flow control information, and maintenance functions. All error handling is done at the hardware level. Notable logical operations introduced in the protocol specification include read/write operations (direct memory access) and messaging [channelized messaging (CM)]. Speeds of up to 25Gbaud per lane are achievable according to the latest protocol specification.

The RapidIO specification defines three protocol layers: the physical layer, the transport layer, and the logical layer (Fig. 3).

The physical layer defines the electrical interface and the link layer operations. RapidIO is defined in a serial and parallel flavor, depending on what is needed in terms of clock signals, speeds, and pin counts. The physical layer packets are small, with low protocol overhead for faster processing and CPU offloading. Error recovery is done immediately in hardware, which takes care of acknowledging and resending packages. This provides for a reliable and fast delivery of individual packets [8].

The transport layer defines how endpoints are identified and how the network devices are enumerated. It also defines how packets are routed from source to destination. The protocol uses destination-based routing.

The logical layer defines the operations for moving data across the interconnect. In general, operations are implemented in hardware or are hardware accelerated.

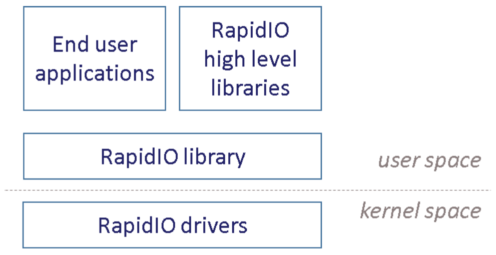

Fig. 4. IDT RapidIO Linux stack, ranging from device drivers up to high-level libraries.

### B. Library and Interface

User-space programs access the interconnect via library calls. The individual library calls interface the underlying driver, which handles the creation and termination of connections, as well as the orchestration of *CM* and *remote direct memory access (DMA)* operations (from now on referred to as rDMA) (see Sections IV-B and IV-C). A maintenance interface is also available for switch configuration and register access.

It should also be noted that the RapidIO specification limits the buffer size of messages to 4096 bytes, while rDMA operations are unrestricted. However, the following library constraints are imposed.

1) Due to a combination of restrictions in the Linux kernel and the current RapidIO library implementation, there is a maximum size of 2 MB for each rDMA allocation.
2) Due to hardware constraints, there can be maximum eight rDMA allocations at one time.

Libraries and device drivers have been provided by Integrated Device Technology, Inc. (referred to as IDT in this paper) and are under active development. This software (Fig. 4) provides several layers of interaction with the interconnect. At the lowest level are drivers that handle the protocol specific parts. They operate at a hardware level, providing the implementation of logical operations. On top of that, there is a library layer, providing an interface that takes care of driver calls and maintenance structures. Apart from this, an even higher abstraction layer is available, which aims to provide a simpler more standardized API.

In this paper, the library layer has been used in favor of the high-level libraries, as it was evident that the library layer interface was a good match for the application to be ported.

### C. Setup

Our hardware setup consists of a 2U Quad unit with four Intel Xeon L5640@2.27 GHz nodes, each with 48 GiB of RAM. Each server is equipped with an IDT Tsi721 PCIe to RapidIO Bridge, offering theoretical speeds of up to 14.5 Gbps. However, initial measurements using simple tests provided by the library show that the library introduces an overhead of 2.5 Gbps, lowering the total achievable bandwidth to 12 Gbps. The nodes are connected to a 38-port Top o Rack (ToR) RapidIO Generation 2 switch box using QSFP+ cables. The switch unit supports speeds up to 20 Gbps. The speed difference between the switch port and the PCIe bridge is due to PCIe bus limitations.

The servers run on CERN CentOS release 7.2.1511. The RapidIO software stack used is Linux kernel drivers and libraries provided by IDT.

## IV. ROOT

ROOT is a data processing framework developed at CERN. Originally targeted at data analysis and simulations related to high-energy physics, ROOT is now used by numerous teams around the world in applications ranging from data mining to astronomy.

It is a standard general-purpose CERN application, which lends itself well to porting, continuously sparking interest in numerous research activities [14], [15]. Therefore, ROOT is an obvious candidate for the first part of RapidIO's assessment: evaluating its suitability in the context of a real-world data analysis application.

Within the scope of our project, ROOT has been extended to use the RapidIO protocol instead of TCP/IP for data transactions. For our purposes, two implementations have been developed: one is utilizing CM and the other is rDMA.

### A. General Implementation Details

At its core, ROOT is highly modularized. Several base classes abstract operating system functionality enabling ROOT to run on different operating systems. The base classes provide functionality such as file system operations and connectivity. Moreover, on top of ROOT an interpreter is supported, facilitating rapid development of application-specific macros. Due to the above, ROOT lends itself well for fast prototyping and proofs of concept.

Through inheritance of one of the existing base classes, a new base class has been provided, replacing the default TCP/IP connectivity with RapidIO. All functionality around announcing, setting up or terminating a connection, and sending or receiving operations on a socket are in this way interfacing the RapidIO library.

However, as ROOT expects a stream-oriented socket interface and RapidIO is messaging based, certain measures need to be taken in order to bridge the paradigm gap, especially concerning data send and receive operations. The principle around such operations for both the CM and rDMA implementations is essentially the same. The general approach will be outlined here, with implementation specific details in their respective sections.

For sending data, the incoming buffer from ROOT needs to be split into chunks, which are individually sent. On the receive side, the original buffer is reassembled and returned to ROOT. During implementation, it became evident that flow control was needed in order to address overflowing of the send and receive queues. However, this introduced a significant overhead, which is excused for a proof of concept, but is not fitting for a production context.

### B. Channelized Messaging Implementation

The purpose of this implementation is to evaluate the use of CM, investigating both speed and robustness.

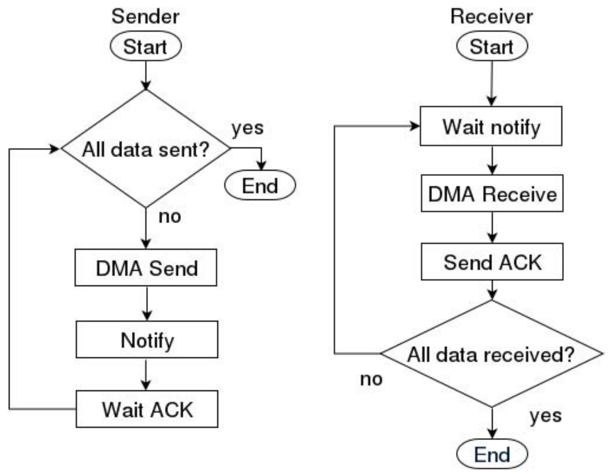

Fig. 5. Sequence flowcharts for send and receive operations for the rDMA extension of ROOT.

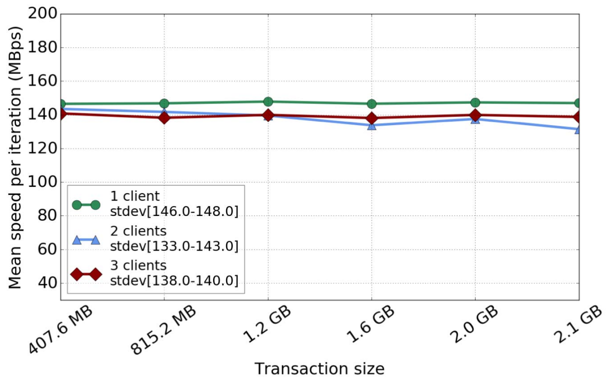

Fig. 6. ROOT—transfer rates with CM extension, client–server scenario.

Even though channelized messages are supposed to be used for orchestration, observing the protocol's performance under load is useful in establishing the boundaries set by the hardware and lower-level software.

As described in the library section, a certain asymmetry characterizes the CM send and receive operations. This contributes to messages being sent faster than they can be serviced. In combination with the absence of flow control, overflowing of send and receive buffers is subsequently caused. To compensate for that and to guarantee that no messages are dropped on the logical layer, flow control was implemented by acknowledging every individual message exchange, introducing a tangible overhead.

## C. Remote Direct Memory Access Implementation

In this implementation, all data transactions use rDMA. rDMA supports zero-copy, a technique that allows the direct copy of data between memory areas present on different hosts. This can be achieved without involving the CPU, eliminating significant overhead due to copying between intermediate buffers as well as context changes.

In the current library implementation, rDMA read and write operations will not issue a notification to the other end upon finishing. Consequently, a race hazard is introduced, where the sender may overwrite data in the memory, as it is being accessed by the reader. A high-level, application-level control is thus necessary in order to guarantee data integrity. This is addressed using channelized messages as a way to mark an end-of-operation. A bottleneck is hence presented, as CM operations introduce significant latency, even more so when compared to rDMA calls, between every block transfer.

The sequence of operations is depicted in Fig. 5.

## D. Benchmarking

In order to evaluate RapidIO's performance for this particular extension of ROOT, the following scenarios were outlined.

1) *A client–server model topology:* The server is running an instance of ROOT, which is accepting connections and, consecutively, data. The clients are sending data to the server concurrently.

   The parameters in the above scenario are the following.

   a) *The size of the transaction data:* The sizes selected span from 4076 bytes, which is exactly one buffer of payload, to 2.15 GB, which is the largest size ROOT's internal structures allow.
   b) *The number of clients:* Our current hardware setup limits the maximum number of physical clients to three.
   c) *The type of implementation to be used:* CM or rDMA.
2) A duplex connection, where each node is both a server and a client. Two transactions are started simultaneously so that each node is sending and receiving data concurrently. The parameters for this scenario are the same as above, accommodating for the fact that we have two nodes.
3) A sustained bandwidth scenario, where the consistency of the throughput, was investigated. Successive iterations of the largest possible transaction were run. This scenario only warrants a single execution and, thus, no parameters can be defined.

The above scenarios were run for all combinations of their respective parameters. This approach allows the investigation of the protocol's behavior in different areas. For every combination, a total of 32 iterations were run. The first and last samples were pruned as invalid data and the mean of the remaining samples was computed.

## E. Results

In Figs. 6–8, the results for a relevant range of transaction sizes for both implementations on the simplex and duplex scenarios are shown.

In order to correctly evaluate benchmarking results several factors need to be taken into account.

1) ROOT utilizes its internal bookkeeping. The various internal operations that are performed by ROOT introduce a relevant overhead, which may skew the results.
2) ROOT is a user-space application, managed by the Linux kernel. Scheduling operations can affect the performance

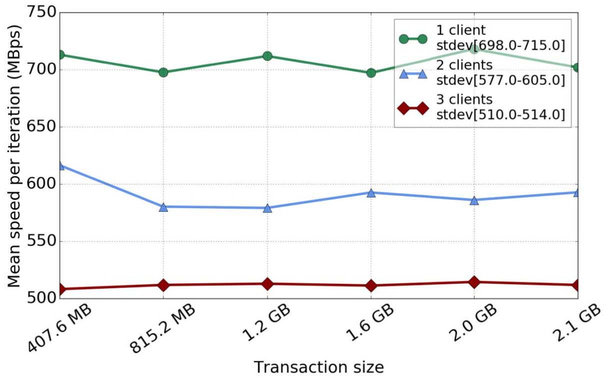

Fig. 7. ROOT—transfer rates with rDMA extension, client–server scenario.

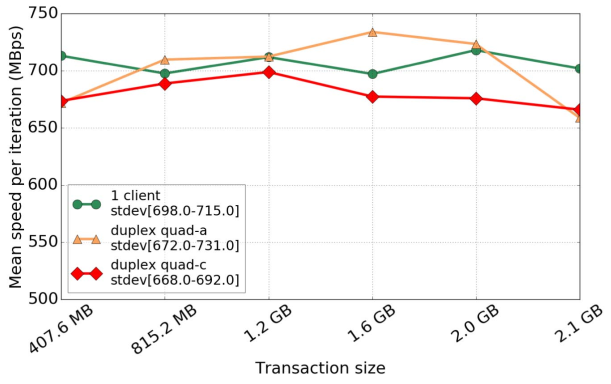

Fig. 8. ROOT—transfer rates with rDMA extension, duplex scenario.

of different transactions, introducing minor differences between each run.

3) ROOT is not optimized to exploit the multithreading capabilities of modern multi-core processors. It is CPU-bound and thus an application-layer bottleneck is introduced that hinders network load.
4) For the CM implementation, every data buffer sent needs to be acknowledged, as described in its respective section. These two operations require approximately the same time, effectively doubling the transfer time.
5) For the rDMA implementation, acknowledgments are also needed. Since the data operations are faster, an increase in performance is observed. However, the use of CM for orchestration limits the potential bandwidth.

Due to the factors stated above, utilization close to the nominal was not achieved. However, a significant difference in speed was observed between the CM and the rDMA implementation, with the former reaching a maximum around 120 MBps and the latter around 700 MBps, a speed that was sustained during the execution of our third scenario. This confirms the fact that CM operations are targeted at orchestration, whereas rDMA operations are better suitable for data transfers. Consequently, an immediate optimization would require the elimination of messaging between frequent rDMA operations, and their substitution with control sequences in the start or the end of a block of data, or the use of pure control buffers. Interleaving buffers should improve performance in both implementations.

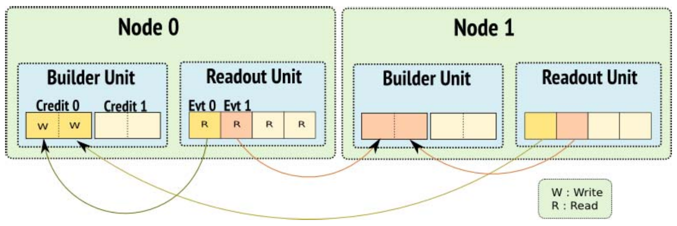

Fig. 9. Data aggregation in DAQPIPE [16]. Spreadout data are written by RUs to the BUs responsible for a specific event.

The above port has confirmed that RapidIO can also be a robust technology outside of the embedded world. With a few optimizations, over a simulated interface, the RapidIO protocol can deliver promising results, which warrant its further investigation in the field. For evaluating its bandwidth capabilities, a more suitable application, DAQPIPE, has been recruited.

## V. DAQPIPE

LHCb-DAQPIPE is a benchmark application to test network fabrics for the future upgrade of the LHCb experiment at CERN. DAQPIPE emulates an event-builder based on a local area network, such as the one envisioned in the LHCb upgrade. The application is protocol, topology, and transport agnostic, allowing for multidimensional testing of an interconnect. Several technologies have already been ported to DAQPIPE for evaluation [5].

The event-building network itself is a fully connected network where nodes receive data from the readout system. These data are subsequently sorted into events. Data first have to be aggregated for one event as it is may originally be spread out over several nodes.

DAQPIPE models the DAQ event building network using an *EventManager* (EM), *ReadoutUnits* (RUs), and *BuilderUnits* (BUs). The EM coordinates the exchange of data between RUs and BUs, following the configured aggregation scheme. RUs write requested data to the corresponding BU memory area, as depicted in Fig. 9.

The aggregation can be configured in many ways, with different aggregation protocols, memory layouts, and network topologies. DAQPIPE has a large set of configuration parameters, which allow for thorough testing on the parameter combination that gives optimal results for a specific interconnect.

Within the specific circumstances of our projects, the parameters that have been of most interest to evaluate are as follows.

1) *Credits:* The number of concurrent events handled per builder unit.
2) *Parallel:* The number of concurrently outgoing requests per credit.
3) *Buffer size:* The maximum size of the build unit write buffer.

### A. Implementation

DAQPIPE was ported to RapidIO using both channelized messages and rDMA. Channelized messages were used for commands, which are utilized by all unit types to organize the event building. rDMA buffers were used as data aggregation buffers.

Some changes to the DAQPIPE interface were needed in order to accommodate RapidIO. First, in the library implementation available to us, memory management is done entirely through the library. This is a consequence of providing zero-copy memory operations, where the library first has to allocate physical memory and then provide the application with a user space handle to it. DAQPIPE had to be extended to interface such a library memory allocation call. Second, an rDMA write notification command had to be added, to handle the fact that our RapidIO software stack does not notify the receiver about changes to the memory buffer. Otherwise, the paradigms used in DAQPIPE and RapidIO matched quite well.

As a first approach, threads were used to introduce asynchronicity in the operations for CM and DMA. It is correct that, for our library, CM operations are blocking (with a timeout) and rDMA operations may be asynchronous, using a cookie to separate the blocking wait of the operation. In order to keep the implementation simple, synchronous calls were chosen for all operation types, with each node having separate threads for CM and rDMA send and receive operations, respectively.

After the main porting work, the rDMA buffer limitations mentioned in Section III-B created the need for two versions of the port.

In the normal implementation of DAQPIPE, each node has a read and write buffer. The read buffer contains the readout data, i.e., parts of an event. The write buffer is for aggregating this event data into a full event. RUs use offsets to write the event data into the corresponding address of the recipient node. With limited buffer size, each node can only collect a certain number of events. This limits the maximum configuration buffer size for the BU.

In the alternative, multisegment implementation, each node instead has *multiple* write buffers, which allows for a larger accumulated buffer size. However, the number of allocations still needs to obey the maximum limit. Thus, this solution limits the number of readout cards that can be connected to each unit as well as the number of credits that can be handled per BU, to one.

## B. Benchmarking

DAQPIPE was run with both implementation versions across several sets of configurations, alternating between two and three units plus an EM. Benchmarking parametrization was influenced by the library limitations mentioned above.

In the multisegment implementation, there can be no more than one credit at one time, whereas the write buffer size can vary up to the RapidIO library limit of 2 MB. A second parametrization is possible for this implementation, with the parallel value.

For the normal implementation, parametrization is possible across a larger number of configuration variables; number of credits, parallel value, and buffer size. The buffer size is, however, limited based on the number of units. In order to

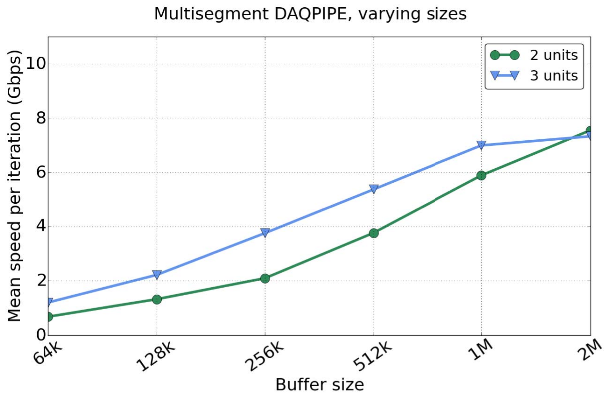

Fig. 10. Multisegment DAQPIPE—transfer rates for varying buffer sizes.

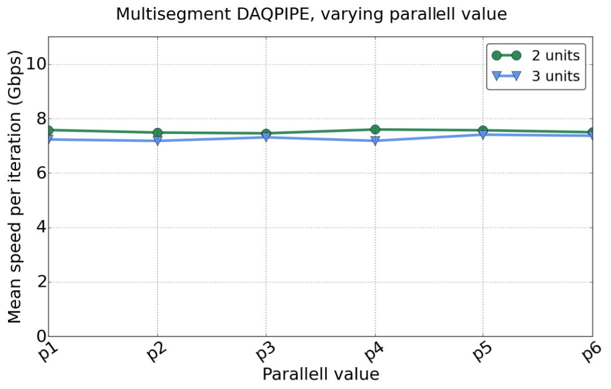

Fig. 11. Multisegment DAQPIPE—transfer rates for varying parallel values, i.e., the number of concurrently outgoing requests per event.

maximize the buffer size, two benchmarks were run with two units instead of three (see Figs. 12 and 13).

In all benchmarks, we used the pull aggregation scheme, namely, each node requests a piece of data from every other node. Except for when a parameter has been varied, the default parameter values are as follows. The default number of credits for the normal version is 4 and the default number of parallel outgoing requests for both versions is 2. The default maximum buffer size for the normal version is 1 MB, and 2 MB for the multisegment version.

Each configuration ran for 1 min, after which results for all runs were collected and a mean value per run was calculated.

## C. Results

In the following figures, results are shown for both the normal and the multisegment implementations of DAQPIPE. Speeds reported refer to an average of transfer speed per node when sending all pieces of requested event data. In Figs. 10 and 11, we can see the multisegment versions. In Figs. 12–14 we see the normal version benchmarks.

The DAQPIPE architecture uses commands to orchestrate the event data aggregation. Event data is subsequently stored in memory segments. In this sense, RapidIO CM and rDMA features fit well with the paradigms used by DAQPIPE. The need for an alternative write segment layout brings another interesting aspect into view, namely, how the interconnect handles multiple concurrent memory allocations. In both Figs. 10 and 14, we see that a higher bandwidth may be

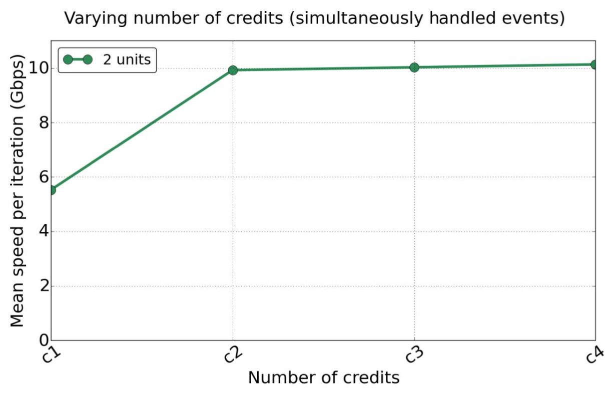

Fig. 12. Normal DAQPIPE—transfer rates for the varying number of credits, i.e., the number of events handled per BU.

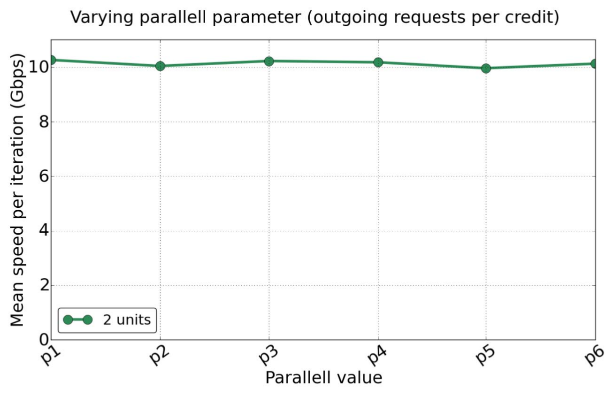

Fig. 13. Normal DAQPIPE—transfer rates for varying parallel value, i.e., the number of concurrently outgoing requests per event.

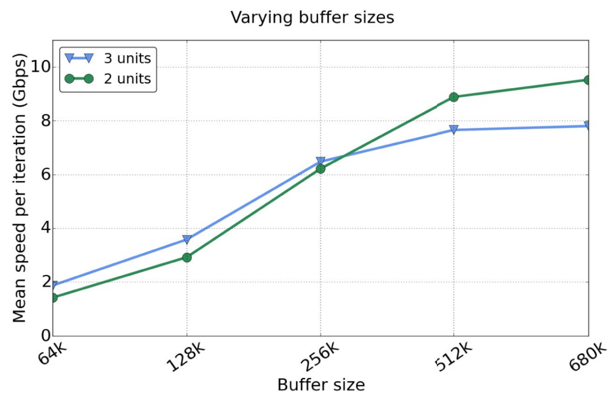

Fig. 14. Normal DAQPIPE—transfer rates for varying buffer sizes.

possible with more memory. A slight scaling effect can be seen, where the runs with 2 units are clearly reaching for higher bandwidths, whereas the runs with 3 units are reaching a plateau. Having access to larger memory buffers would allow further investigation.

In Figs. 11 and 13, it is clear that the number of outgoing requests per event does not significantly affect the overall bandwidth. It is possible that with a larger number of nodes, the parameter would come into effect, as there would be more nodes to request from. At the same time, bottlenecks in concurrent rDMA write operations could be investigated.

Finally, it is clear from Fig. 12 that letting each BU handle more than one event at a time increases the bandwidth significantly. It enables more concurrence at this scale compared to varying the number of outgoing requests. However, increasing the number of credits over two does not change the speed drastically, possibly because there are only two units available.

Our small sized setup proves it is possible to port and run DAQPIPE using RapidIO. A larger scale setup would be needed in order to further investigate how the technology scales, as well as how the switch handles various traffic patterns.

## VI. Conclusion

In this paper, we presented our work, evaluating the use of RapidIO in the context of two real-world applications outside of the previously known setting.

Using a common data analysis framework, ROOT, we have investigated the technology's suitability on a higher level application context. We note that the message-based paradigm offered by RapidIO had to be bridged to the stream-based sockets used in mainstream applications. A maximum speed of around 5.6 Gbps was achieved. However, further optimization is possible by improving the data transfer algorithm to work exclusively on rDMA and by introducing interleaving buffers. We conclude that RapidIO can be successfully utilized to offer robust performance in such a setting.

With the help of an event-building emulator, DAQPIPE, the current bandwidth capabilities of RapidIO were investigated. DAQPIPE matches well with RapidIO in terms of paradigm, a fact that made the porting procedure seamless. Speeds of up to around 10 Gbps were reached, a figure that approaches the nominal line rate. With a library implementation that simultaneously allows for a larger and a higher number of memory allocations, the performance can be improved. Even with the aforementioned limitations, the technology is promising to warrant further investigation.

A thorough comparison alongside 10 Gbps Ethernet would be relevant in order to draw further conclusions concerning the suitability of the RapidIO technology in this context.

In order for this comparison to be decisive, it is essential to scale up and investigate the behavior of the switch and network cards at our disposal.

## Acknowledgment

The authors would like to thank M. Akhter, D. Paul, B. Wood, and A. Bounine from Integrated Device Technology (IDT) and O. Bärring, N. Neufeld, and A. Grigore from the European Organization for Nuclear Research, for their support and collaboration. The authors would also like to thank IDT for its membership and technical collaboration with CERN Openlab.

## References

[1] RapidIO.org. *RapidIO Interconnect Specification Version 3.1*. Accessed: Jun. 22, 2016. [Online]. Available: http://www.rapidio.org/wp-content/uploads/2014/10/RapidIO-3.1-Specification.pdf

[2] RapidIO.org. *RapidIO.org*. Accessed: Jun. 24, 2016. [Online]. Available: http://www.rapidio.org/

[3] R. Brun and F. Rademakers, "ROOT—An object oriented data analysis framework," *Nucl. Inst. Meth. Phys. Res. A*, vol. 389, nos.1–2, pp. 81–86, 1997. [Online]. Available: http://root.cern.ch/

[4] D. H. C. Pérez, R. Schwemmer, and N. Neufeld, "Protocol-independent event building evaluator for the LHCb DAQ system," in *Proc. Realtime Conf.*, 2014, p. 7097440.

[5] B. Vőneki, S. Valat, R. Schwemmer, N. Neufeld, J. Machen, and D. H. C. Pérez, "Evaluation of 100 Gb/s LAN networks for the LHCb DAQ upgrade," in *Proc. IEEE-NPSS Real Time Conf. (RT)*, Padua, Italy, 2016, pp. 1–3.

[6] G. F. Pfister, "An introduction to the InfiniBand architecture," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. New York, NY, USA: Wiley, 2002, pp. 616–632.

[7] Architecture and Systems Platform. *RapidIO: An Embedded System Component Network Architecture. Motorola Semiconductor Product Section*. Accessed: Jun. 22, 2016. [Online]. Available: http://www.cs.ucr.edu/~mart/CS260/rapidIO.pdf

[8] G. Shippen. *System Interconnect Fabrics: Ethernet Versus RapidIO Technology*, Freescale Semiconductor Inc. Accessed: Jun. 22, 2016. [Online]. Available: http://cache.freescale.com/files/32bit/doc/app_note/AN3088.pdf

[9] D. Bueno, A. Leko, C. Conger, I. Troxel, and A. D. George, "Simulative analysis of the RapidIO embedded interconnect architecture for real-time, network-intensive applications," in *Proc. 29th Annu. IEEE Int. Conf. Local Comput. Netw.*, Nov. 2004, pp. 710–717.

[10] W. Changrui, C. Fan, and C. Huizhi, "A high-performance heterogeneous embedded signal processing system based on serial RapidIO interconnection," in *Proc. IEEE Int. Conf. Comput. Sci. Inf. Technol.*, vol. 2. Jul. 2010, pp. 611–614.

[11] M. Schmid, F. Hannig, and J. Teich, "Power management strategies for serial RapidIO endpoints in FPGAs," in *Proc. IEEE Int. Field-Programm. Custom Comput. Mach. Symp.*, May 2012, pp. 101–108.

[12] G. S. Madhusudan, C. Rebeiro, B. Ravindran, S. Raman, R. Bodduna, and N. Gala. (2015). *SHAKTI Processor Project*. [Online]. Available: https://bitbucket.org/casl/

[13] The Virgo Collaboration, "Advanced Virgo Technical Design Report," Virgo, Cascina, Italy, Tech. Rep. VIR-0128A-12, 2012, pp. 452–476. [Online]. Available: https://tds.virgo-gw.eu/ql/?c=8940

[14] M. Arsuaga-Ríos, S. S. Heikkilä, D. Duellmann, R. Meusel, J. Blomer, and B. Couturier, "Using S3 cloud storage with ROOT and CvmFS," *J. Phys. Conf. Ser.*, vol. 664, no. 2, p. 022001, 2015.

[15] A. Heisss, U. Schwickerath, and G. Ganis, "InfiniBand for high energy physics," in *Proc. Comput. High Energy Phys. Nucl. Phys. Conf.*, Interlaken, Switzerland, Sep. 2004, pp. 1217–1220.

[16] D. Cámpora, S. Valat, B. Vőneki, and S. Baymani. *LHCB-DAQPIPE Wiki*. Accessed: Jun. 22, 2016. [Online]. Available: https://gitlab.cern.ch/svalat/lhcb-daqpipe-v2/wikis/home