Design and Commissioning of the First 32-Tbit/s Event-Builder

Flavio Pisani, Tommaso Colombo, Paolo Durante, Markus Frank, Clara Gaspar, Luis Granado Cardoso, Niko Neufeld, *Member, IEEE*, and Alberto Perro

***Abstract*—The large hadron collider beauty (LHCb) experiment is a forward spectrometer, designed to study beauty and charm quarks physics at the large hadron collider (LHC). To exploit of the higher luminosity that will be delivered during Run3, the full experiment needed a substantial upgrade, from the detector to the data acquisition (DAQ) and high level trigger (HLT). In this article, we will focus on the new DAQ system for the LHCb experiment that represents a substantial paradigm shift compared to the previous one, and to similar systems used by similar experiments in the past and present times. To overcome the inefficiencies introduced by a local selection implemented directly with the readout hardware, the Run3 system is designed to perform a full software reconstruction of all the produced events. To achieve this, both the DAQ and the HLT need to process the ∼30 MHz full event-rate. In particular, this article will introduce the final design of the system; it will provide a focus on the hardware and software design of the event building (EB) and how we integrated technologies designed for the high performance computing (HPC) world — like InfiniBand HDR (200 Gb/s) — into the DAQ system; we will present performance measurements of the full EB system under different operational conditions; and we will provide a feedback from EB operation during the beginning of the data-taking.**

***Index Terms*—Computer networks, data acquisition (DAQ), large hadron collider (LHC).**

I. Introduction

The large hadron collider beauty (LHCb) experiment [1] has been completely upgraded for the so-called Run3 of the large hadron collider (LHC), officially started the 5th of July 2022. The upgrade process involved all the aspects of the experiment, from the actual detectors to the full read-out and data acquisition (DAQ) chain, a comprehensive description of the process can be found in [2].

In this article, we will focus on the DAQ system, and in particular on the event building (EB) system. The new event-builder is designed to work at the unprecedented data rate of 32 Tb/s sustained. We will first introduce the architecture of the new DAQ system with a strong focus on the EB process; then we will provide a detailed description on how the EB functionality is implemented in software and how it is integrated with the experiment control system (ECS); we will describe the hardware architecture of the system and the network topology of the EB network; and finally we will present performance measurements of the EB system gathered in a controlled test environment, and the first validation of the full DAQ chain.

Manuscript received 16 November 2022; accepted 24 January 2023. Date of publication 30 January 2023; date of current version 16 June 2023.

Flavio Pisani, Tommaso Colombo, Paolo Durante, Markus Frank, Clara Gaspar, Luis Granado Cardoso, and Niko Neufeld are with CERN, Meyrin, 1211 Geneva, Switzerland (e-mail: flavio.pisani@cern.ch).

Alberto Perro is with CERN, Meyrin, 1211 Geneva, Switzerland, also with the Istituto Nazionale di Fisica Nucleare (INFN), Sezione di Torino, 10125 Turin, Italy, and also with the Department of Physics, Università degli Studi di Torino, 10124 Turin, Italy.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNS.2023.3240514.

Digital Object Identifier 10.1109/TNS.2023.3240514

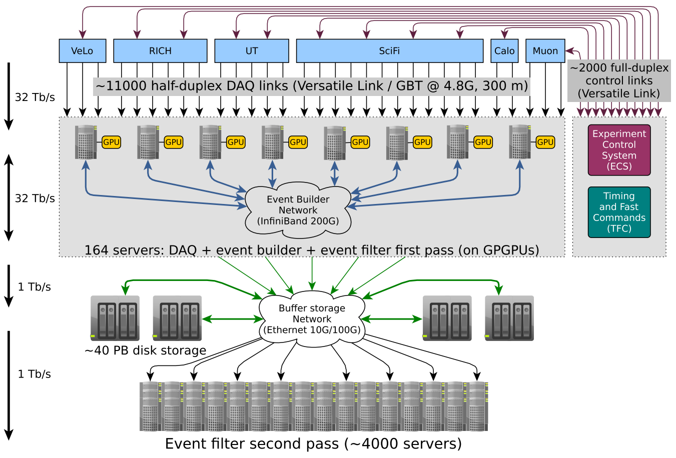

VeLo
RICH
UT
SciFi
Calo
Muon
~2000 full-duplex control links (Versatile Link)
~11000 half-duplex DAQ links (Versatile Link / GBT @ 4.8G, 300 m)
32 Tb/s
GPU
32 Tb/s
Event Builder Network (InfiniBand 200G)
Experiment Control System (ECS)
Timing and Fast Commands (TFC)
164 servers: DAQ + event builder + event filter first pass (on GPGPUs)
1 Tb/s
Buffer storage Network (Ethernet 10G/100G)
~40 PB disk storage
1 Tb/s
Event filter second pass (~4000 servers)

Fig. 1. Architecture of the LHCb readout system for Run3. The blue boxes in the upper part of the picture represent the various sub-detectors FE electronics. The central part contains the EB servers interconnected via a dedicated 200G InfiniBand network, and the GPGPU used for the first data reduction. The lower part depicts the temporary disk buffer and the server farm used to perform the final data reduction. The throughput numbers on the left represent the aggregated nominal values at each stage. On the right, a schematic representation of the ECS infrastructure is given.

II. LHCbDAQ System

The DAQ system of the LHCb experiment has been completely redesigned to perform: detector readout, EB, and event reconstruction at the full nominal colliding bunches rate of 30 MHz.[1] As depicted in Fig. 1, the full ∼32-Tb/s data-rate is extracted from the sub-detectors front-end (FE) electronic cards via ∼11 000 point-to-point multimode optical fiber links. To ensure the correct operation of the links in a radioactive environment, such as the LHCb experimental cavern, the links use a radiation hard implementation of the GBT

[1]Because of the geometry and the filling scheme of the LHC, the actual beam–beam collision rate at the experiment's interaction point is 26.7 MHz, the 30 MHz design value includes also 2 MHz of beam-empty and 1 MHz of empty-empty data, which are used for calibration and luminosity measurements.

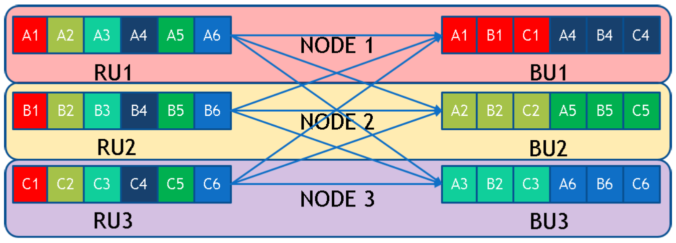


Fig. 2. Representation of the EB process. Event fragments generated by the same source are identified by the same letter, while the various events are identified by a unique event number. The event fragments flow from the RUs to the BUs to build complete events. Every readout unit (RU)/builder unit (BU) pair constitutes and EB node.

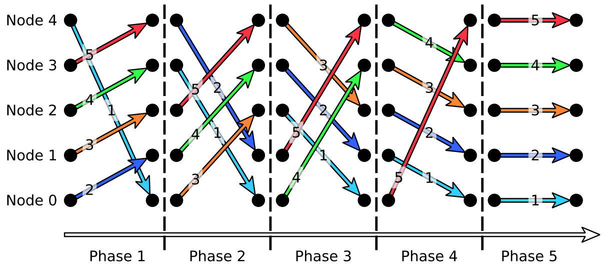


Fig. 3. Example of a linear shifting scheduling. The arrows represent the data exchange between the EB nodes during the various phases. Fragments of the same event can be identified either by arrow color or by the matching event number.

protocol [3]. Those links are terminated into 445 peripheral component interconnect express (PCIe)-based read-out cards, named Tell40; this custom FPGA-based card is responsible for receiving data over up to 48 GBT links and making it available for EB via the PCIe bus. Each card implements up to two independent event streams, and each stream transfers data to the host memory via a dedicated PCIe Gen3 $\times$ 8 interface. To perform a full ~32-Tb/s EB, a dedicated 200 Gb/s-based InfiniBand network is used.

After the EB process is performed the complete events are filtered by a two-step high level trigger (HLT) process; the first step runs on general-purpose graphics processing units (GPGPUs) hosted on the same servers used for DAQ and EB, and it consists of a full track reconstruction with nominal calibration and alignment [4]. The second step is a full offline-like reconstruction with the most recent calibration and alignment constants, and it is executed on a farm of ~4000 servers. To absorb any possible fluctuation in the distribution of the events, and to update the calibration and alignment of the various sub-detectors, a temporary storage buffer of ~40 PB is placed between the two HLT steps.

A more detailed description of the full system is beyond the scope of this article and it can be found in [2]. In the publication, we will focus on both the HW and SW details of the EB system.

A. Overview of the EB Process

The EB process consists of gathering into a single place all the fragments of a given event. In the LHCb experiment's DAQ system, this operation is performed by two logical units as follows.

1) *Readout Unit (RU):* Collects the fragments from the PCIe-based DAQ board and sends them to the BUs.
2) *Builder Unit (BU):* Receives and aggregates the fragments into complete events.

The system uses a push DAQ model in which the RUs are responsible for assigning a given event to a given BU; this is done to reduce the transmission latency and to remove the complexity and the potential bottleneck introduced by a centralized event scheduler.

Fig. 2 shows the flow of the different event fragments from the RUs to the BUs. Because the data flow only in one direction, to take advantage of the full bidirectional bandwidth of the EB network, we decided to implement a folded architecture, i.e., a BU and a RU share the same network port and establish an EB node. This architectural choice allows to reduce by a factor of two the number of network ports required to perform the EB process, at the cost of sharing other resources of the node, e.g., CPU and RAM.

Because every BU needs to gather data from all the RUs, the EB network traffic tends to create an instantaneous N-to-one over-subscription. This temporary over-subscription can lead to severe performance degradation, especially for high link usage applications [5]. On the other hand, the traffic generated by the EB process is highly predictable, and if the scheduling of the event across the BUs implements a fair policy, this link over-subscription can be averaged out by an adequate traffic shaping. In particular in the LHCb event-builder, we implement a *linear shifting* traffic shaping algorithm [6] which operates in the following way.

1) We build N events in parallel, and we split this process into N phases, where N is the total number of nodes.
2) In every phase, every RU sends data to exactly one BU and every BU receives data from one RU only.
3) During phase n RU i sends to BU $(n+i) \mod N$.
4) When a previously agreed condition is met all the nodes synchronously switch from phase n to phase $n+1$, usually the switching is triggered either by the number of sent events or by a fixed time window. If the fragment sizes differ, the larger fragments will dictate the actual event rate.

A schematic representation of a five node linear shifting scheduling is depicted in Fig. 3. The specific implementation details of the linear shifting algorithm will be described in Section III, including the extra complexity needed to maintain the synchronism required by the algorithm.

In a large distributed system, like the EB network of the LHCb experiment, the selection of nonconflicting source-destination pairs of nodes is a necessary but not sufficient condition to ensure the absence of link congestion in the full network topology; a detailed description of the HW and network layout of the system will be provided in Section IV, including the additional constraints needed to make the scheduling effective.

The typical fragment size in the LHCb experiment is O(100) B. No multi Gb/s network technology is designed to efficiently send messages of such a small size. To make the network transmission more efficient, fragments from multiple

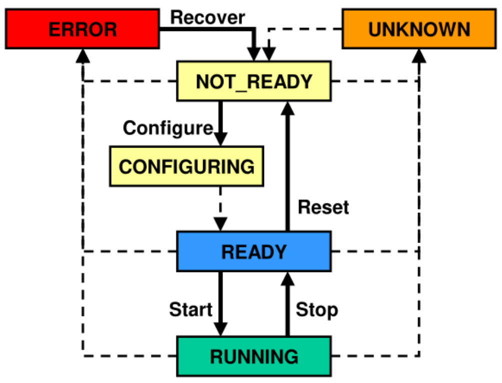


Fig. 4. State diagram of the DAQ components in the ECS. Each block indicates a possible state of the system, while the edges show the transition with the respective commands.

events are packed into a multiple-fragment packet (MFP). This packed data structure contains a fixed number of contiguous fragments from a given source, and it constitutes the minimum unit of data that is exchanged over the network by the EB nodes. In a similar way, the events on the BUs are assembled in multiple-event packets (MEPs) which contain a fixed number of contiguous complete events. All the considerations made for events fragments and full events are still valid for MFPs and MEPs, respectively; therefore, in the rest of this article, we will consider an EB system that sends MFPs and builds MEPs.

III. Software Architecture of the EB

The EB of the LHCb experiment is designed as a modular application written in C++. The code base is fully integrated into the experiment's Online framework (DataFlow), and it provides an object-oriented implementation of the base-building block of an EB, i.e., the RU and the BU.

A. DataFlow Framework Overview

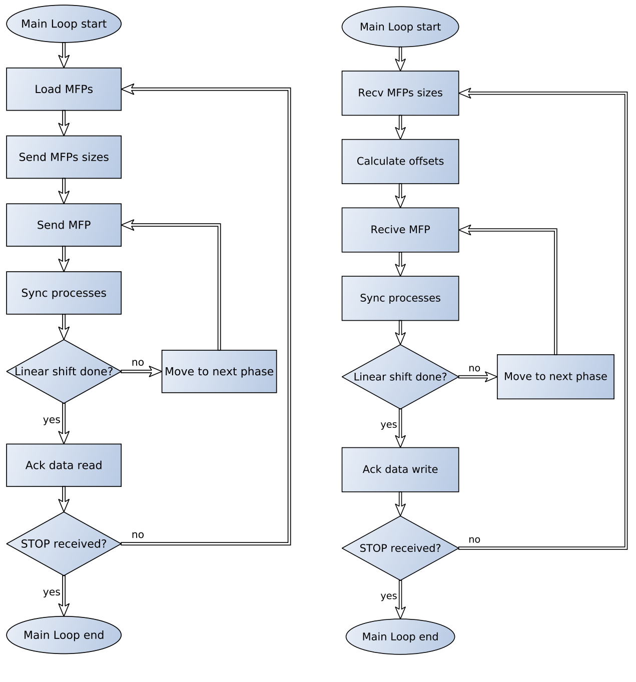
DataFlow is the framework used for most of the LHCb online applications. It is implemented in C++, and it provides easy to use interfaces to central online infrastructure: like the finite state machine (FSM) used by the ECS or the centralized monitoring infrastructure.

The base class of the framework is named `DataflowComponent`, and it implements an FSM which is steered by the ECS via a distribute information managment system (DIM) [7]. Fig. 4 shows the states and the transitions of the FSM used by the ECS to describe the DAQ processes. The behavior of the FSM can be configured thanks to a virtual interface which allows full customization by overriding the default behavior.

In addition to the FSM implementation, the DataFlow framework offers access to the configuration values set by the ECS and to the centralized monitoring system. The latter allows to gather and aggregate scalar values and histograms via DIM.

B. Building Blocks of the Software

The core functionality of the EB software is provided by the `BU` and `RU` classes, those two specializations of



(a) (b)

Fig. 5. Flowchart of the main loop executed by the `run` thread of the (a) RU and (b) BU.

the `DataflowComponent` implement the state machine required to configure and to run the EB with the linear shifting scheduling described in Section II-A. Because some of the functionality needed to perform a linear shift is common between the RUs and BUs, the two classes have a common ancestor called `Transport_unit` which provides all the commonly used functionality.

Fig. 5 shows the main loop flow for the RU and the BU. It is important to note that prior to the actual exchange of the MFPs more operations need to take place. In particular, the RUs needs to prefetch all the data that will be exchanged in the full round trip of the scheduling and to send the sizes of the MFP to the corresponding BU; the BUs need to receive all of the MFPs sizes and they have to calculate the offsets in memory at which every MFP should be written within the MEP. Those operations are necessary to write every fragment at the right place in memory, avoiding subsequent memory copies.

The interface to the low-level communication library is provided by the `Parallel_comm` class – an abstraction layer between the actual low-level network implementation and the high-level description needed by the EB units. This approach makes it possible to optimize the communication layer without further modification of the `RU` and `BU` classes, and it allows to change the low-level API without a major rewrite of the code.

The communication between the EB and the outside world, e.g., the DAQ cards or the HLT input buffer – is handled via a common object-oriented buffer interface. This buffer interface consists of two distinct interface classes: `Buffer_reader` and `Buffer_writer`. To read data from a buffer, the user can request the next element via the corresponding `Buffer_reader` object, and it will receive a pointer to the actual data. Multiple elements can be requested in subsequent calls and all the needed bookkeeping is handled internally by the library. After the data have been successfully processed, the user can acknowledge the read as completed, and therefore make the memory available to the buffer. In a similar way, to write into a buffer, the corresponding `Buffer_writer` object can be used to request a chunk of memory of a given size. Multiple chunks can be requested via subsequent calls and every time a pointer to the new chunk will be provided. After the data have been successfully written, the user can acknowledge the write as completed and therefore make the data in the buffer readable. The EB software stack currently provides multiple specialization of those two classes that allow to read/write data in different configurations like: reading data from the Tell40 cards, reading/writing from/to POSIX shared memory regions, injecting previously collected data or Monte Carlo simulated data and an interface to the central buffer manager used by the other components of the DataFlow framework. Thanks to a templated implementation, this buffer interface can be used both for MFPs and MEPs.

The last building block of the EB software that we will analyze is the `MFP_generator`. This `DataflowComponent`-based class provides a CPU data generator that can be used to replace the Tell40 cards. This data generator outputs MFPs with the same format as used by the Tell40, and can be configured to generate data of different sizes and at different event rates. This particular data generator has been used throughout all the development phase and it is still used to perform specific code optimization in a controlled environment.

C. Low-Level Communication Library: InfiniBuilder

We developed a custom interface library based on the low-level InfiniBand API (ibverbs) [8], giving us complete control over the InfiniBand network. This choice was driven by some limitations we found using OpenMPI [9] to exchange event data in the EB application.

OpenMPI's opaque memory management causes an initial performance degradation introduced by the memory set up for remote direct memory access (RDMA) transfers, leading to a significant increase in warm-up time. In InfiniBuilder, memory management is straightforward: by leaving the user flexibility on when and where buffers are allocated. Therefore, the warm-up time is greatly reduced.

An OpenMPI application runs as a monolithic entity composed of multiple processes, so if one process dies or encounters an irrecoverable error, the whole application needs to be terminated and reexecuted from scratch. This approach increases the dead-time caused by a local problem on one of the EB processes. The InfiniBuilder library solves this problem by only taking care of the InfiniBand communication, enabling the application to reset and establish the network at runtime.

InfiniBuilder provides network and memory management through an OpenMPI-like interface, simplifying its integration in the EB software. This approach gives us more flexibility compared to a standard OpenMPI implementation, and it allows for easier integration with the ECS.

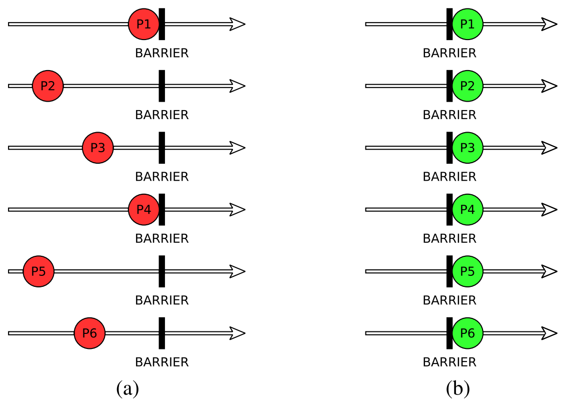


Fig. 6. Time diagram of the two phases of a synchronization barrier. (a) All the processes reach the barrier and they are not allowed to continue their execution. (b) After all the processes have reached the barrier, the execution can be resumed.

D. Synchronization Barrier

To synchronize the EB processes and therefore ensure a correct execution of the linear shifting scheduling, we decided to implement a synchronization barrier. This collective communication service is commonly used in the high performance computing (HPC) world to ensure the correct order of execution of distributed algorithms. This synchronization process is implemented as a two step procedure: first all the processes reach the execution of the barrier, this will send a notification over the network and it will block the process execution until the barrier is released; the second step takes place after all of the processes have successfully reached the barrier, and it consists of sending a notification to all the processes and consequently releasing them as depicted in Fig. 6.

Implementing a barrier operation on a large network in an efficient way is not a trivial task, and the way the interprocesses messages are sent can heavily influence both the execution time of the barrier and its scalability. The InfiniBuilder library offers two different barrier implementations: 1) a centralized barrier and 2) a tree-based barrier [10].

1) Centralized Barrier: This implementation is the simplest for a synchronization barrier, the previously described two-step process is implemented in the following way.

1) One process is selected as *master* process, this can be assigned statically during the configuration of the application.
2) All the processes send a message to the master process when they reach the barrier.
3) When the master process receives all the messages, it starts releasing all the others in a sequential order.

This purely sequential implementation has an execution time proportional to the number of processes in the distributed application.

2) Tournament Barrier: These algorithms exploit the parallelism of the release phase of the barrier by implementing a tree structure that can be represented as follows.

1) All the processes are inserted into a binary tree, equivalent to the bracket of a tournament. For every pair of nodes, a node is selected as *winner*. The selection is made upon a predefined criterion, for example, the parity of the node.

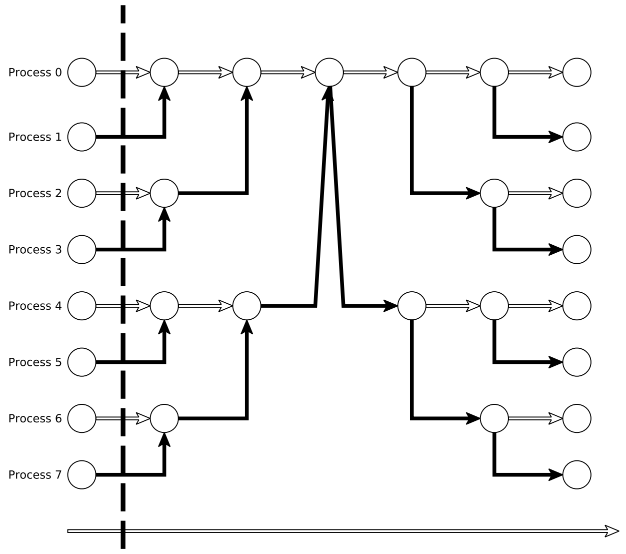


Fig. 7. Communication diagram of a tournament barrier. The black arrows represent a message exchange between two processes, and the white ones represent self-communication. The dashed line represents the transition between the first and the second phase of the barrier.

2) The *winners* will be connected to the next level of the tree.
3) This approach is repeated for all the layers of the tree, and the root of the tree becomes the *champion*.
4) Every node will signal to its local *winner* when the barrier is reached.
5) Each local *winner* will signal the next level *winner*, when all the processes in its level have reached the barrier.
6) This process is repeated at every layer until the *champion* is reached.
7) The *champion* will release the processes in its layer.
8) Every other *winner* process will propagate the release following the tree structure.

A schematic representation of the communication between the various processes is depicted in Fig. 7. This tree-based algorithm offers better scalability than the centralized implementation, thanks to its logarithmic execution time.

IV. Hardware Architecture of the EB

The EB infrastructure of the LHCb experiment is designed to use off-the-shelf components as much as possible; to reduce the significant costs of developing cutting-edge custom solutions.

In this section, we will discuss in detail the network configuration and the HW configuration of the EB servers.

A. Event-Builder Network Topology and Configuration

The EB network topology is constructed using 40-ports InfiniBand HDR top-of-rack (ToR) switches; the switches are arranged in a folded-clos/fat-tree [11] network topology, as depicted in Fig. 8. We decided to use this particular network topology because: it allows to fulfill all the requirements introduced by the linear scheduling algorithm, i.e., the network is nonblocking; it can be easily implemented with ToR switches; there are optimized routing algorithms available for this topology, and they are already implemented for InfiniBand networks [12].

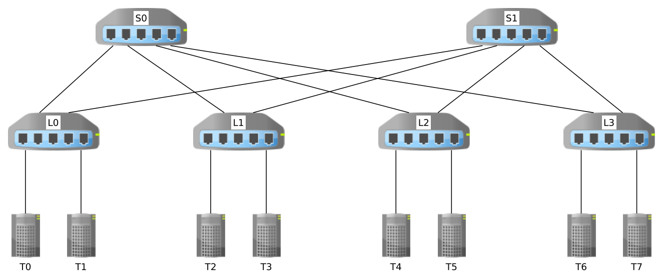


Fig. 8. Fat-tree network topology. The eight end nodes (terminals) are located at the bottom, the four switches in the first layer are called *leaf* switches, while the two switches at the top are called *spine switches*.

The actual network topology counts 18 *leaf* switches and ten *spine* switches. The ports on each leaf switch are split into two groups: 1) up to 20 ports are connected to the EB servers via copper cables and 2) the other 20 ports are connected via optical fibers to the spine switches. Each connection between *leaf* and *spine* switches uses two dedicated links to ensure the required bandwidth. Each spine switch is therefore connected to all the leaf switches using 36 ports; the other four ports are left unused for future expandability of the system. In the current configuration, the network has a capacity of 360 network ports, and it could be expanded up to 800 without loosing all the needed properties.

To fulfill the requirements imposed by a linear shifting traffic shaping, the routing algorithm should be carefully selected. If we limit ourselves to minimal distance paths, the only degree of freedom when selecting the path between a source and a destination port of the network is the selection of the spine switch used. To ensure a conflict-free operation with a linear shifting traffic, we need to verify, during every phase, the following conditions: the nodes on the same leaf are not using the same spine switch and on every spine switch there is no traffic for the same leaf switch.

A routing algorithm with those characteristics has been designed [12] and it has been implemented for the InfiniBand network technology. This algorithm uniformly distributes the traffic on the leaf-to-spine links according the port used on the leaf switch by the destination. On a simplified network topology like the one shown in Fig. 8, this routing function is equivalent to selecting the spine switch via

$$S(p_d) = p_d \mod N_{\text{spine}} \tag{1}$$

where p_d indicates the destination port and N_{spine} the total number of spine switches. A more general description of the InfiniBand architecture and routing can be found in [13].

To ensure the correct link utilization, we need to make sure that the mapping between the EB nodes and the network ports on the switches is consistent with the routing function. In the example network depicted in Fig. 8, we could assign node0 of the EB to run in the server T0, node1 on T1, and so on. This correspondence between EB nodes and network ports is not unique, and we can apply any permutation to the node-port mapping on a given switch, as long as the

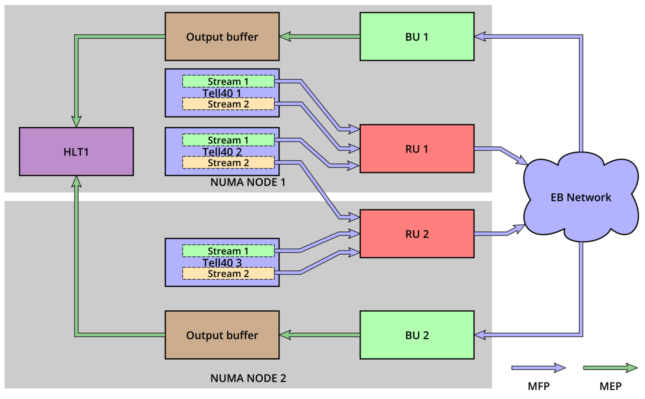


Fig. 9. Software data-flow of one EB server. The Tell40 cards are distributed across the two nonuniform memory access (NUMA) domains, and the six MFP streams are then equally shared across the two RU. Every RU/BU pair has exclusive access to one InfiniBand network card. After the MEPs are assembled into the BUs, the data are made available to the HLT1 via two dedicated output buffers.

same permutation is applied on all the switches. Moreover, we need to make special consideration in case a node is missing from one of the switches. In this case, to keep the right scheduling phase, a dummy node must be inserted into the scheduling. Dummy nodes are implemented in software and they have the only purpose of maintaining the scheduling and the routing in sync. Adding dummy nodes introduces a performance penalty proportional to the fraction of dummy nodes added. The ECS will therefore configure the mapping and the addition of dummy nodes accordingly.

B. Event-Builder Server Architecture and Configuration

The heart of the DAQ system is a dual-socket AMD “Rome” Epyc-based server solution, which can host up to 8 Gen4 × 16 PCIe devices. Each of these servers is equipped with: up to three Tell40 cards, two InfiniBand HDR network cards, and one general-purpose graphics processing unit (GPGPU).

Fig. 9 shows the data-flow of the events inside of an EB server. Each server will host two EB nodes, one on every non-uniform memory access (NUMA) node. The processes of every EB node have an exclusive access to one of the two InfiniBand network cards. This nonconflicting resource utilization keeps all the prior considerations about the network topology and the traffic scheduling valid. Each RU reads data from three MFP streams, and the stream-unit assignment maximizes NUMA locality. After the EB process is successfully completed, each BU writes MEPs to its own output buffer, and consequently the HLT1 processes the data on the GPGPU.

Thanks to the fact that the events are moved in MEPs and that the current baseline packing factor for MFPs and MEPs is 30 000 events, we can ignore statistical fluctuations in the event processing time of HLT1. Moreover, given that all the servers have the same amount of compute power, we can assume that the average processing time per MEP will be constant. Therefore, we do not need to implement any load balancing across the BUs. If a given server is not capable of coping with the incoming event rate, for example because of a hardware failure of the GPGPU, the affected BUs can discard events locally. This mechanism prevents a global backpressure and a severe reduction in the number of events processed.

If this condition cannot be recovered, it is possible to reconfigure the EB to run in a partly unfolded way, therefore moving the affected BUs onto a fully working machine. Similar considerations apply to the RUs which can be moved in case of a HW issue with one or multiple Tell40 cards. It should be noted that using a spare Tell40 requires to move the optical fibers connected to the front-ends (FEs). In our data-center, we have multiple warm spare servers ready to be used.

The architectural choice of buffering the events directly into the servers’ RAM poses interesting considerations concerning the latency susceptibility of the system. Given the usually large amount of memory available on modern server platform (O(100) GB), it possible to easily absorb latency spikes generated in the EB or in the HLT. On the other hand, the limited amount of memory on the Tell40 makes the PCIe transaction latency critical. To reduce the FPGA-CPU communication latency, the manufacturer provides a performance guide [14].

It is important to note that such a dense system configuration imposes a very high load on the I/O and memory of the platform. To reduce the memory throughput the full system is designed with a zero-copy paradigm in mind, the MFPs are copied into the server RAM by the Tell40, the InfiniBand network card sends the data into the output buffer via an RDMA transfer, and finally the HLT1 software sends the data from the server RAM directly into the GPGPU device memory. This design choice, in conjunction with the eight memory channels per socket provided by the server platform, allows such a dense configuration, and results in fewer servers needed.

V. Full Scale Performance Measurements

In this section, we present two different sets of performance measurements: 1) a scalability test performed using the `MFP_generator` and 2) a full DAQ chain test reading actual MFPs from the Tell40 cards.

A. Scalability Test

The aim of this test was to benchmark different configurations of the EB software in a controlled environment. We performed two sets of tests in different conditions: using a realistic event-size model, and by setting the highest data-rate to all the generators. The goal of the first test was to measure the overall performance of the system in a realistic scenario, and to evaluate if a strict synchronization is needed at every step of the scheduling. The second test was performed to check the current absolute limit of the system, and to evaluate different barrier implementation.

The plots in Fig. 10 show the event-rate and the aggregated throughput across all the EB nodes. The system is tested at three different scales.

1) *100%:* full system.
2) *60%:* only the tracking sub-detectors.
3) *10%:* only two racks (20 server and two leaf switches).

The test has been performed using a Poissonian event-size model. We used three different distributions to emulate the behavior of different sub-detectors, as depicted in Fig. 11.

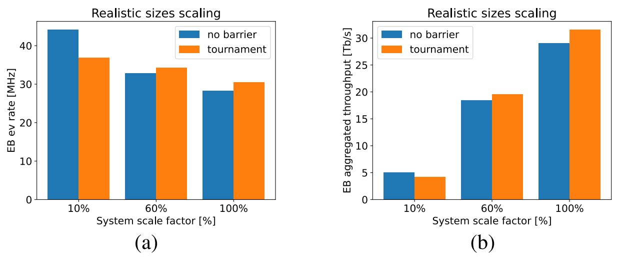


Fig. 10. EB scalability test using CPU-generated MFPs with a realistic event-size model. The plot on the left shows the event rate while the plot on the right the aggregated throughput. Both measurements are collected at different system scales and with different synchronization barriers. (a) Event rate. (b) Aggregated throughput.

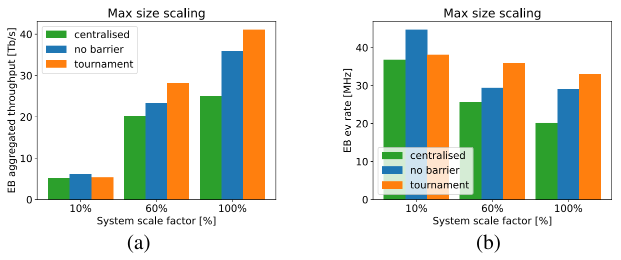


Fig. 12. EB scalability test using CPU-generated MFPs with the maximum event size. The plot on the left shows the event rate while the plot on the right the aggregated throughput. Both measurements are collected at different system scales and with different synchronization barriers. (a) Event rate. (b) Aggregated throughput.

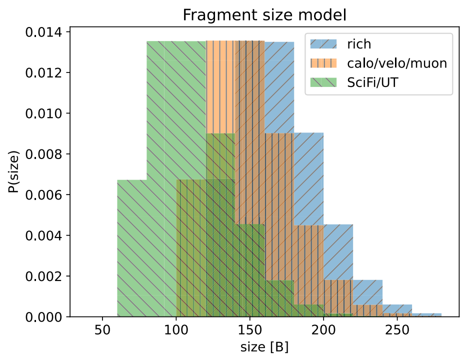


Fig. 11. Probability distribution of the event fragment sizes for different emulated sub-detectors. Sub-detectors with similar estimated fragment sizes share the same distribution. The total event fraction for the rich, calo/velo/muon, and SciFi/UT groups are: 0.28, 0.33, and 0.39, respectively.

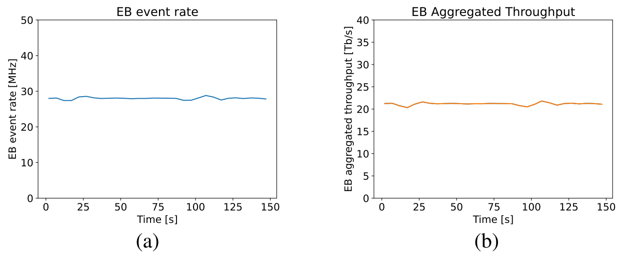


Fig. 13. Full DAQ chain test, the plot on the left shows the event rate while the plot on the right the aggregated throughput. (a) Event rate. (b) Aggregated throughput.

Each `MFP_generator` uses a distribution appropriate for the sub-detector that it is replacing. In particular, the three groups contribute to the total even size with the following fractions: 0.28, 0.33, and 0.39 for the rich, calo/velo/muon, and SciFi/UT groups, respectively. The average fragment size is 133 B, and the total event size is 128 kB for the 100% configuration.

The test shows that when the size of the system is very small the strong synchronization at every step of the linear shifting is not needed, and the overhead introduced by the synchronization barrier introduces a net performance penalty. On the other hand, when the system scales up, the scheduling violation generates a higher congestion level on the network, resulting in a more significant performance degradation, which can be up to $\sim$6% at full scale. Beside the importance of the synchronization barrier, we can also confirm that the system can sustain the 30 MHz event rate required by the LHCb experiment, with an aggregated throughput greater than 30 Tb/s.

The plots in Fig. 12 show the event-rate and the aggregated throughput across all the EB nodes. The test has been performed using the largest fragment size used by our sub-detectors on all the `MFP_generator` instances. This model uses a fragment size of 160 B and an event size of 154 kB for the 100% configuration.

This second test shows how different barrier implementations behave according to the size of the system. The comparison between running with the tournament barrier and without any barrier shows a similar behavior as the one seen in the previous test, the overall event rate is higher in both configurations; this is due to the fact that now all the MFPs have the same size, and therefore the time lost due to the synchronization overhead is lower. The performance penalty introduced by the lack of synchronization barrier is $\sim$10% at full scale. This higher performance penalty is expected because now the link utilization is higher, therefore the congestion introduced by the less strict scheduling policy is more severe. The maximum sustained throughput achieved in this test is greater than 40 Tb/s.

By comparing the results obtained with the two different barrier implementations, we found an event rate which is $\sim$40% lower for the centralized barrier compared to the tournament one. The simpler centralized implementation performs worse than not having any barrier at all, and it does not meet the requirements imposed by the experiment.

Time-based code profiling shows that the time spent in the synchronization barrier is 26% for the tournament barrier and 48% for the centralized one. The comparison between the two values is in line with observed performance degradation. The high value for the tournament implementation is due to the high invocation frequency. With a packing factor of 30 000 events per MEP, the barrier rate required to build 30 MHz of events is $\sim$1 kHz.

B. DAQ Integration Test

This integration test was performed to test the full DAQ chain from the FE electronics up to the EB. Fig. 13 shows the event rate and the aggregated throughput over time.[2] The maximum achieved event rate in this configuration was 28 MHz,

[2]The actual run-time was longer than the one represented in the plot, and the run has been stable for 1 h before we decided to stop the test.

and this performance degradation is probably generated by the combination of the Tell40 buffer not being large enough and some latency spikes on the acknowledge of the PCIe transaction. A new firmware and driver for the Tell40 are currently under development, and they will include changes to mitigate this issue. The aggregated throughput is 21 Tb/s with a total event size of 95 kB.

VI. Conclusion and Future Work

In this article, we presented the new DAQ system for the LHCb experiment, and we provided a detailed description of the hardware and software implementation of the new event builder. The scalability of the system has been tested in a controlled environment, and the EB can successfully process an event rate greater than 30 MHz. The full DAQ chain has been successfully tested and data from the entire experiment can be read out at 28 MHz.

The results of the tests performed show that the system is ready to readout the full collision rate produced at the LHCb interaction point of 26.7 MHz. Further improvements and optimizations to the firmware and the driver of the readout cards will be done to run the system at 30 MHz. Moreover further optimization of the EB software will be done. In particular, we are working on reducing the frequency of the synchronization barrier, and we are evaluating hardware accelerated barriers like SHArP [15].

References

[1] The LHCb Collaboration, "The LHCb detector at the LHC," *J. Instrum.*, vol. 3, no. 8, 2008, Art. no. S08005.

[2] The LHCb Collaboration, "LHCb trigger and online upgrade technical design report," Tech. Rep. CERN-LHCC-2014-016, LHCB-TDR-016, May 2014, Accessed: Nov. 16, 2022. [Online]. Available: https://cds.cern.ch/record/1701361

[3] P. Moreira et al., (2009). *The GBT Project*. Accessed: Nov. 16, 2022. [Online]. Available: https://cds.cern.ch/record/1235836

[4] R. Aaij et al., "Allen: A high-level trigger on GPUs for LHCb," *Comput. Softw. Big Sci.*, vol. 4, no. 1, pp. 1–11, 2020.

[5] T. Colombo et al., "Flit-level InfiniBand network simulations of the DAQ system of the LHCb experiment for Run-3," *IEEE Trans. Nucl. Sci.*, vol. 66, no. 7, pp. 1159–1164, Jul. 2019.

[6] D. P. Bertsekas, C. Özveren, G. D. Stamoulis, P. Tseng, and J. N. Tsitsiklis, "Optimal communication algorithms for hypercubes," *J. Parallel Distrib. Comput.*, vol. 11, no. 4, pp. 263–275, Apr. 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0743731591900336

[7] C. Gaspar, M. Dönszelmann, and P. Charpentier, "DIM, a portable, light weight package for information publishing, data transfer and inter-process communication," *Comput. Phys. Commun.*, vol. 140, nos. 1–2, pp. 102–109, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465501002600

[8] Nvidia Corporation. *RDMA Aware Networks Programming User Manual*. Accessed: Nov. 16, 2022. [Online]. Available: https://docs.nvidia.com/networking/display/RDMAAwareProgrammingv17

[9] *Open MPI*. Accessed: Nov. 16, 2022. [Online]. Available: https://www.open-mpi.org/

[10] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *Int. J. Parallel Program.*, vol. 17, no. 1, pp. 1–17, Feb. 1988.

[11] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vols. C–34, no. 10, pp. 892–901, Oct. 1985.

[12] E. Zahavi, G. Johnson, D. J. Kerbyson, and M. Lang, "Optimized InfiniBand^{TM} fat-tree routing for shift all-to-all communication patterns," *Concurrency Comput., Pract. Exper.*, vol. 22, no. 2, pp. 217–231, Nov. 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1527

[13] G. F. Pfister, "An introduction to the infiniband architecture," *High Perform. Mass Storage Parallel*, vol. 42, nos. 617–632, p. 102, 2001.

[14] A. Kashyap. (2020). *High Performance Computing: Tuning Guide for AMD Epyc^{TM} 7002 Series Processors*. Accessed: Nov. 16, 2022. [Online]. Available: https://developer.amd.com/wp-content/resources/56827-1-0.pdf

[15] R. L. Graham et al., "Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction," in *Proc. 1st Int. Workshop Commun. Optimizations HPC (COMHPC)*, Nov. 2016, pp. 1–10.