Design of a request/response buffering application for I/O intensive workloads

F Grötschla[1,2], G Lehmann Miotto[1] and R Sipos[1]

[1] CERN, Geneva, Switzerland
[2] ETH Zurich, Zurich, Switzerland

E-mail: fgroetschla@ethz.ch, giovanna.lehmann@cern.ch, roland.sipos@cern.ch

Abstract. The performance of I/O intensive applications is largely determined by the organization of data and the associated insertion/extraction techniques. In this paper we present the design and implementation of an application that is targeted at managing data received (up to ~150 Gb/s payload throughput) into host DRAM, buffering data for several seconds, matched with the DRAM size, before being dropped. All data are validated, processed and indexed. The features extracted from the processing are streamed out to subscribers over the network; in addition, while data resides in the buffer, about 0.1 ‰ of them are served to remote clients upon request. Last but not least, the application must be able to locally persist data at full input speed when instructed to do so. The characteristics of the incoming data stream (fixed or variable rate, fixed or variable payload size) heavily influences the choice of implementation of the buffer management system. The application design promotes the separation of interfaces (concepts) and application oriented specializations (models) that makes it possible to generalize most of the workflows and only requires minimal effort to integrate new data sources. After the description of the application design, we will present the hardware platform used for validation and benchmarking of the software, and the performance results obtained.

1. Introduction

The idea for a generic request/response buffering application arose when developing a detector readout system for the Deep Underground Neutrino Experiment (DUNE) [1]. The readout is the subsystem of the data acquisition system (DAQ) that processes incoming data from various detector front-end electronics and buffers the data for a certain amount of time. During this time, other subsystems may request data from the readout via a request/response mechanism. The readout runs on commercial off-the-shelf (COTS) servers and receives data via various means, such as PCIe FPGA carrier boards (like the FELIX [2]), high bandwidth network interface cards (NIC) or others. Creating an application that supports a wide range of front-end electronics with various characteristics and offers the necessary quasi real-time performance at the same time has lead to a design that is modular and generic enough to not only be used in a detector readout system, but in much wider use cases. We want to discuss the strict requirements we were met with and derive a specification for the application and its subcomponents. We will then go into more detail on each of the subcomponents and conclude with a demonstration of the application that was tested on the ProtoDUNE [3] setup at CERN.

2. Specification

The specification of the overall application was driven by the requirements for a detector readout system. In contrast to other readout systems we base our work on a DAQ that is self-triggered, where triggers are software defined and processing has to be done in quasi real-time. Although this had an impact on our design present a specification and implementation of the system that is generic enough to support a wide variety of use cases (also outside of a detector readout system) by focusing on reusability.

Characteristics of front-end electronics The application has to support a wide range of detector front-end types that can vary in their characteristics. The most notable characteristics that we are working with are as follows:

- Size of payloads can be constant or variable
- Arrival rate can be fix (same time delta between payloads) or variable
- Payloads can arrive in or out of order (regarding an index)

These differences may impact performance, therefore a careful implementation of data processing, aggregation and memory allocation is required. To accommodate these characteristics our software is agnostic towards them and relies on interchangeable modules that implement performance critical tasks while offering a unified interface. At the same time, the software has to be performant enough to keep up with quasi real-time, high throughput I/O.

Buffering and data requests At the core of the application, payloads have to be buffered, meaning that they have to be stored in memory for a certain time until they are allowed to be dropped and thereby deleted. While the data is buffered, window queries (data request that ask for payloads for a given interval) will request payloads that are buffered. In order to have a meaningful way to define window queries, payloads have to be indexed. This index can then be used to lookup and search for certain data. We can usually think of this index as a timestamp for the payloads and the window queries asking for payloads in a certain time interval.

Data processing The application has to provide hooks to register functions that

(i) Pre-process payloads in a sequential pipeline before they are buffered and
(ii) Post-process data with a set of functions that are executed in parallel on immutable payloads.

Pre-processing provides a way to run functions that inspect the incoming data. In the context of the detector readout system this can be used for error and consistency checks and may also filter payloads that should not be buffered. Pre-processing functions are allowed to change the payload, once it is stored in the buffer it becomes immutable. This also guarantees that the payload index does not change and that we always return the same data for the same request. Post-processing functions can be more heavy on resources and one use case is feature extraction.

Persistency While data requests use a request-response mechanism to extract data from the buffer, we also specify a second type of request that we hereby call recording request. Once initiated, all payloads currently residing in the buffer and arriving for a time specified in the recording request have to be transferred to a separate data stream (e.g. stored locally on the server). Recordings are designed to persist data for an extended period of time. For DUNE, this feature is needed to store all payload in the event of a supernova burst. Here, recordings are initiated for 100 seconds and data with a rate of up to 100 Gbit/s (for 10 links) has to be stored locally.

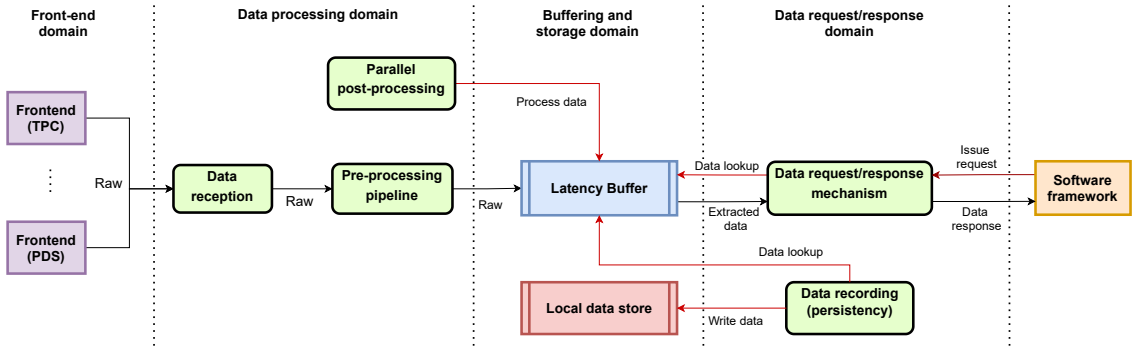

Figure 1. Dataflow diagram of the buffering application. The diagram highlights the system's subcomponents and their interactions. Black arrows symbolize the flow of data while red arrows symbolize control flow.

3. Components and design

To facilitate different input types we use a modular software architecture that separates different features into interchangeable components. The implementations for components can then be selected for different use cases to provide front-end specific features or to optimize certain core functionalities. The five components of the application are: the *Readout Type*, *Latency Buffer*, *Frame Processor*, *Request Handler* and *Readout Model*. A dataflow diagram depicting the interactions between (sub-)components of the application is shown in Figure 1. Payloads are received by the application from different front-ends through the data reception block. This is done by the *Readout Model* which forwards the data and invokes the pre-processing pipeline that is implemented by the *Frame Processor*. After being pre-processed completely, data then reaches the buffering and storage domain where it is stored in the *Latency Buffer*. Once payloads are added, the post-processing tasks can be executed in parallel and buffered payloads are available to be extracted by data requests. While the post-processing is also part of the Frame Processor, both recording and data requests are handled by the *Request Handler*. It is also responsible for removing old data from the Latency Buffer. Recording requests do not respond with data but write it to a local store. All described subcomponents are contained within the *Readout Model* that is also responsible for interfacing with the surrounding software framework.

Readout Type Payloads are represented by their *Readout Type* which is an aggregation of their corresponding front-end data frame. It provides access to the underlying data by functions that expose the memory location of each frame and its size in memory. Static fields expose information about the payloads and can be evaluated on compile time. These fields define the size of the Readout Type, a comparison function (implicitly defining the index on the data),whether the payload can have variable size and the memory location of the wrapped data. The Readout Type is designed as an aggregation of data frames to reduce the number of total payloads that are passed through the application. It can be more efficient to work on bigger but less payloads as they are pushed through the processing steps in bigger chunks and the number of add and remove operations on the buffer becomes smaller.

Latency Buffer The interface of the Latency Buffer specifies functions to add new data, access and search for data by the means of the underlying index and remove data. It provides the core functionalities for our application and is performance critical. The Latency Buffer needs to be searchable to enable queries on it. Different implementations of the Latency Buffer exist

that optimize performance for different payload characteristics. Deleting data from the Latency Buffer is an explicit operation that needs to be done regularly to avoid it reaching its maximum capacity. The Latency Buffer does not remove data automatically. The reason for this decision is that other processing tasks can depend on data in the buffer and need to be carefully synchronized with any removal. This could be a request extracting some data or a parallel post-processing task. Data is deleted by the component working on it, namely the Request Handler.

Frame Processor The Frame Processor is responsible for two processing tasks: pre- and post-processing functions. Pre-processing functions use as a sequential pipeline model and functions are called in the order they were registered. A wide range of processing and checks can be done on the payloads here, e.g. error checks or data-driven stream duplication where payloads with a certain characteristic are copied to a specified output stream. The functions also have to be fast enough to process all incoming payloads in time and only afterwards the payloads are added to the Latency Buffer. Post-processing functions are called in parallel once a payload is added to the buffer and every registered post-processing function processes payloads in the order that they are added to the buffer. Multiple registered functions can work independently from each other and therefore process their streams in parallel. These tasks may be more resource intensive, but still have to finish processing a payload before it is deleted from the buffer. If a post-processing function is not keeping up with the data rate, the remaining components of the application must be unaffected.

Request Handler The Request Handler is responsible for both data and recording requests. Data requests are read-only on the Latency Buffer and can therefore be implemented to run in parallel. The Request Handler uses data residing in the Latency Buffer and is aware of any access to it and the occupancy of the buffer. It therefore also removes ("cleans up") elements from the buffer on a regular basis (i.e. when the occupancy of the Latency Buffer reaches a certain threshold). The cleanup has to be executed at a frequency that ensures that the Latency Buffer has space to store more payloads at any time. Lastly, the Request Handler is also able to queue requests that it cannot answer yet (i.e. requested data is not available in the buffer yet). These requests are delayed until they can be responded to. To make sure that requests are not delayed forever, a timeout can be set after which a waiting request is terminated and the requester is notified.

4. Implementation details

We differentiate between software interfaces of the components and their implementations. Interfaces are called concepts and their respective implementations are called models. Different models for the same concept can exist and work in an interchangeable manner. For example, different Latency Buffer models can be used and swapped out without changing other components. Templating is used in our C++ 17 implementation to achieve this goal.

Latency Buffer The available implementations for the Latency Buffer are based on well tested and established datastructures from the Folly open source library [4]. The first two are based on a modified version of the producer consumer queue, while the third one uses Folly's concurrent skiplist implementation. These datastructures support concurrent operations, which is one main requirement in our use case as adding, accessing and removing items is done by different entities. The first buffer implementation supports payloads that are added at a fixed rate, i.e. the timestamp difference between arriving payloads is always the same. Payloads that do not arrive at a fixed rate but in order can be stored in the second implementation based on the same producer consumer queue but extends the lookup with a binary search. For elements that can arrive out of order, we provide an implementation based on the skiplist.

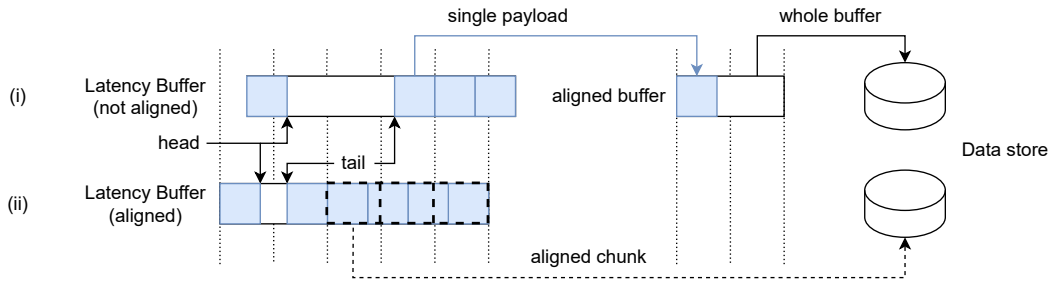

Figure 2. Recording implementations that support writing with alignment restrictions. Dotted vertical lines represent aligned memory addresses for the buffers, blue boxes represent payloads in the buffer. Data is copied from the cyclic buffers starting with the oldest element (at the tail position) to the most recent one (head). Implementation (i) copies payloads from a non aligned Latency Buffer to an aligned intermediate buffer. Implementation (ii) uses an aligned Latency Buffer and copies are done for aligned chunks (dashed). Note that the first chunk copied from the buffer starts with the first aligned payload location. The previous payload was skipped.

Frame Processor The Frame Processor implements the pre- and post-processing functions on the payloads. As such, it depends on the details of the front-end it receives data from. In our readout system, it has to run different checks (errors, calibration, etc.) and support functionalities for different front-ends. To facilitate this, one Frame Processor is implemented for every front-end. A generic Frame Processor class is provided that can be extended for front-end specific features and exposes an interface to register the desired processing functions.

Request Handler To parallelize the request handling, we make use of a thread pool that contains a fixed number of threads. When a request is received, it is posted to the thread pool and executed in parallel with other data request handlers. The occupancy of the Latency Buffer is monitored and once it reaches a certain threshold, a cleanup is requested. The cleanup waits for all currently running data requests to finish, new data requests that arrive after the cleanup are not executed directly. They wait until the cleanup is finished. This construction ensures that a cleanup is executed on its own while requests can run concurrently.

In contrast to data requests, recordings run for an extended amount of time and we can not hinder the cleanup from running while we record the data. To write data to disk efficiently and fast enough we use the Linux flag `O_DIRECT` [5] on file descriptors. The flag indicates that no kernel buffers should be used for copies, avoiding costly copy operations. It comes with restrictions regarding the alignment of the user space buffer that data is copied from, the number of bytes that can be copied with one call and the alignment of the file offset. To address them, data is copied from a memory aligned user space buffer. Figure 2 shows two of our implementations. Implementation (i) copies payloads to an aligned intermediate buffer (implemented using boost streams [6]) first. Copies of data to the aligned buffer is done in the cleanup function just before a payload is deleted, ensuring that all data in the buffer is persisted once a recording is requested. Implementation (ii) copies the data from a separate thread and uses an aligned Latency Buffer. Due to the alignment restrictions, aligned chunks are copied and the first non-aligned payloads are skipped. The number of skipped payloads is bounded and the buffer can be oversized to compensate for this effect. A downside of this implementation is the fact that it only supports a shallow copy of data in the buffer. Payloads that store data on the heap (e.g. variable sized payloads) can not be copied with this approach.

5. Demonstration

The open-source implementation is available online on github [7]. The implementation was successfully tested as part of the dunedaq software framework [8]. It was both tested and fully operational with software defined payload emulators and real front-end electronics and aggregator I/O devices. The software emulators are able to generate data-streams with combinations of variable/fixed rate and size. Besides functional and integration testing, we also investigated scalability and resource management for high-throughput workloads. The integration with real hardware was tested with FPGA carrier boards (in our case the FELIX) for the TPC readout of the DUNE Vertical-Drift prototype and with network interface cards for the photodetector of ProtoDUNE-SP with a legacy front-end. For the FELIX workload we receive data from 10 input links, each of them operating at ≈1GB/s. In total this makes for a rate of incoming data of about 89GBit/s. We tested this workload on a single reference readout server with dual socket Intel Cascade Lake (Xeon® Gold 6242) (64 cores and 192GB RAM). The server was able to handle data requests and software hit finding as a post-processing step with a total CPU load of less than 50% per socket, and system-wide memory bandwidth utilization of less than 60%. These results look promising as there is still headroom of resources, and potentially we can even use more I/O devices and storage units in PCIe Gen4 machines.

In addition, we tested the recording feature with our different implementations thereof. We were able to keep up with sustained writes of about 89 GBit/s (the data rate of one APA in DUNE) for over 100 seconds to a software RAID0 of 4 NVMe SSDs (Samsung 970 Pro 1TB) equipped on PCIe adapter card. By using the implementation that copies data directly from the Latency Buffer to the device, this added only the maximum necessary memory utilization overhead of copying all data to the storage devices.

When scaling up the application, i.e. by hosting more devices in one server, we observed that device and resource locality becomes a key issue that has to be handled. The system configuration regarding allocated cores and memory per task and pinning of threads (among others) has to match the server specification to minimize CPU and bandwidth utilization.

6. Outlook

The main focus for future work lies on scalability and resource locality. On one hand, CPU and memory bandwidth utilization could be improved further by the use of dynamic thread affinity balancers. Extending this to multi-socket systems, we also want to exploit NUMA awareness and socket interconnect capabilities by reducing the amount of data that is copied between sockets to a minimum. Lastly, the buffering application is currently tied to the dunedaq software framework and uses some capabilities that it provides. For the future, we want to make the library a standalone package and integrate it into other DAQ software frameworks to make it usable for even more use cases. One such candidate is DAQling [9] which is based on similar principles as dunedaq.

References

[1] Abi B *et al.* 2020 *Journal of Instrumentation* **15** T08008–T08008
[2] Borga A *et al.* 2019 *IEEE Transactions on Nuclear Science* **66** 993–997
[3] Sipos R 2019 *IEEE Transactions on Nuclear Science* **66** 1210–1216
[4] Folly: Facebook open source library URL `https://github.com/facebook/folly`
[5] O_direct kernel flag URL `https://man7.org/linux/man-pages/man2/open.2.html`
[6] Boost c++ libraries URL `https://www.boost.org/`
[7] Generic buffering application implementation URL `https://github.com/DUNE-DAQ/readoutlibs`
[8] Dune daq software framework URL `https://dune-daq-sw.readthedocs.io/en/latest/`
[9] Boretto M, Brylinski W, Miotto G L, Gamberini E, Sipos R and Sonesten V V 2020 *EPJ Web of Conferences* **245** 01026