

# artdaq: An Event Filtering Framework for Fermilab Experiments

K. Biery, C. Green, J. Kowalkowski, M. Paterno, and R. Rechenmacher

**Abstract**—Several current and proposed experiments at the Fermi National Accelerator Laboratory have novel data acquisition needs. These include (1) continuous digitization, using commercial high-speed digitizers, of signals from the detectors, (2) the transfer of all of the digitized waveform data to commodity processors, (3) the filtering or compression of the waveform data, or both, and (4) the writing of the resultant data to disk for later, more complete, analysis.

To address these needs, members of the Accelerator and Detector Simulation and Support Department within the Scientific Computing Division at Fermilab have chosen to use parallel processing technologies in the development of a generic data acquisition toolkit, **artdaq**. The **artdaq** toolkit uses MPI (Message Passing Interface) and **art**, an established common event framework for Intensity Frontier experiments. In an **artdaq** program, the digitized data are transferred into processors nodes using commodity PCIe cards, event fragments are combined into complete events using MPI, and filtering and compression algorithms are run on the data using **art**. To test the toolkit, a cluster of five 32-core high-performance computing nodes has been assembled and connected with a QDR InfiniBand network. Initial testing of data throughput shows event building rates in excess of 1.5GB/s.

This paper describes the architecture and implementation of the first phase of the **artdaq** toolkit and shows early performance results with configurations that match upcoming experiments such as Mu2e, NO $\nu$ A, and DarkSide-50.

## I. INTRODUCTION

**T**HE **artdaq** project has been established to design and develop a generic toolkit for the construction of efficient and robust event filtering and analysis programs within data acquisition systems for future experiments at Fermilab. These experiments have fewer collaborators than Fermilab's collider experiments, and so cannot easily afford to develop as much customized software as could the larger experiments of the TeVatron era.

An important aim of the **artdaq** project is to allow the sharing of data acquisition (DAQ) infrastructure between experiments, helping them to work within the smaller budgets available to them. We are able to help smaller experiments with limited resources to concentrate their effort on the parts of the system that are experiment-specific, and to relieve them of the burden of supporting the parts of the code that can be dealt with in a generic (*i.e.* non-experiment-specific) manner.

A second aim of **artdaq** is to allow use of commodity computers, rather than special-purpose hardware such as Field-Programmable Gate Arrays (FPGAs), as close to the data

source as possible. This makes programming easier, because many more physicists know how to program general-purpose computers than know how to program special-purpose hardware. Since modern commodity computers have many cores—and in the near future, possible hardware accelerators such as General-Purpose Graphic Programming Units (GPGPUs)—**artdaq** is designed to take advantage of multiple cores, and to make the development of modular algorithm convenient, some of which might be implemented on GPGPUs. Additionally, we aim to take advantage of the high throughput of modern machines, using high-performance networks, hardware buses, and interconnects.

In many of the experiments with which the authors have worked, the development of online and offline event<sup>1</sup>-processing code has proceeded separately, by communities who interact and exchange code with insufficient frequency. The result is that the integration of the online and offline codes has been a time-consuming challenge. To alleviate this problem **artdaq** makes use of the **art** [1] event-processing framework, which is also used as the offline event-processing framework for many of the future Fermilab experiments. Experiments who use **artdaq** would thus gain the benefit of a larger community of developers for the online system (the offline system is typically understood by more collaborators). Additionally, this means much of the code used in online triggers system can be verified and profiled for performance improvements in the offline environment.

## II. PROBLEMS ADDRESSED

In this section, we describe three of the problems that **artdaq** addresses. They are general in nature; although not all apply to each experiment, most experiments encounter one or more of them. In the subsequent section, we describe some concrete use cases, for specific experiments, that have guided the development of **artdaq**.

### A. Trigger Algorithm Execution

Many experiments require the ability to run algorithms, some of which may be complex, to select the subset of events to be written. In order to fine-tune the event selection, experiments want the ability to modify selection thresholds and to replace algorithms, without rebuilding programs. Additionally, many experiments want the ability to run multiple trigger algorithms in the same program, on the same event stream.

Because of the degree of sophistication of trigger algorithms, experiments want to enable all physicists interested in working

<sup>1</sup>An event, in our terminology, is a collection of data associated with one time window, and is the smallest unit of data to be processed.

on trigger algorithm development and testing to do so. Thus experiments want to be able to run the trigger algorithms in the offline framework, as well as in the online system. This allows for easier development, as well as study of the algorithms within the simulation, without the concern inherent in the comparison of two different implementations of (what is intended to be) the same algorithm. Seamlessly supporting multiple environments also permits extensive algorithm debugging and performance studies using typically more readily-available offline computing resources.

### B. Event Building

The detectors built by most experiments are read out through multiple, often heterogeneous, DAQ front-ends. Each front-end is responsible for reading a fixed portion of the detector hardware. One of the important tasks to be undertaken by the DAQ system is the assembly of all the readouts corresponding to a single event. We call this assembly process *event building*. Event building often requires the coordinated work of several computing nodes.

The throughput rates of the hardware and software that make up the event building system directly limit the amount of data that an experiment can process in a given period of time. Thus it is imperative to communicate data efficiently and reliably from data collection nodes to wherever the triggering algorithms (or other data processing algorithms, *e.g.*, data compression) will be run. This includes systems that have no data filtering in front-end hardware and event processing times that vary widely. Depending on the computing resources available to an experiment, it may be beneficial to use the same computing nodes for both tasks. In order to be able to allow the necessary testing to determine the optimal event building system configuration, experiments want to be able to reconfigure the system (adding more processing capacity, or reacting to loss of hardware) without reprogramming.

### C. Single Node Processing Capacity

Modern experiments need to make use of modern computing hardware, which means taking advantage of multicore platforms, and, when useful, accelerators such as GPGPUs. In an era of tight budgets, it is critical to take full advantage of the most affordable commodity computing resources available. Increasingly, this means taking advantage of both distributed and shared-memory parallel computing technologies. However, it is not reasonable to expect that all contributors to online software development will become experts in parallel programming techniques. In **artdaq**, we are working to develop tools that simplify the development of software that is able to take advantage of the parallelism inherent in the event-building and triggering tasks, and which utilizes multicore hardware and high-throughput networks to their greatest advantage.

## III. GUIDING USE CASES

### A. The NO $\nu$ A Prototype Data Driven Trigger

The NO $\nu$ A [2] experiment at Fermilab currently features a free-running continuous readout system without dead time,

which collects and buffers time-continuous data from over 350 thousand readout channels. The raw data must be searched to correlate it with beam spill events from the NuMI [3] beam facility.

The NO $\nu$ A event-building system is designed to continuously process data at full sampling rate from the NO $\nu$ A detectors, using commodity networking and computing equipment. For the far detector, custom designed upstream hardware delivers fragments of data in 5ms time slices to more than 180 multicore commodity buffering nodes using standard gigabit Ethernet switches. The fragments are assembled into full time-synchronized windows, which are then written to shared-memory segments. The time windows are indexed to allow for efficient search and delivery to downstream applications upon receipt of positive trigger broadcasts. The system can sustain a raw data input rate of greater than 2GB/s and buffer in excess of 20 seconds' worth of data.

NO $\nu$ A is investigating the design and development of a Data Driven Trigger (DDT). In the prototype DDT, the shared-memory segments are ingested by a process that reads all raw time slices at full rate, and feeds them into **art**, the event-processing framework also used by the NO $\nu$ A offline software. This framework runs analysis modules that examine each slice in real-time to identify event topologies of interest. The analysis results then are fed back into the experiment's triggering systems to form data-driven decisions.

The first physics algorithm to be completed was a track finding algorithm based on a serial version of the Hough transform algorithm. Because the physics algorithms that are part of the prototype NO $\nu$ A DDT are implemented as **art** modules, the algorithms can be developed, tested, and subject to performance profiling and tuning in the offline environment, and then used in the online environment. New algorithms can be added to the DDT by changing a configuration file, with no need to recompile the DDT program.

Using a few of the components of **artdaq** in early studies for NO $\nu$ A, we have been able to read real data out of the NO $\nu$ A NDOS DAQ via the existing shared memory interface and queue those data for processing by an instance of **art**. Figure 1 shows the distribution of time taken by the trigger algorithm to process each event. This algorithm is amenable to parallelization and certain optimizations, which will be the subject of further developments. Even without these improvements however, the mean time of 98ms gives confidence that the problem is tractable using commodity hardware. Figure 2 shows that the framework and data movement processing time is small compared to the trigger algorithm run time. The current near detector has approximately half the channels of the finished near detector, meaning that with no improvement in the algorithm we would extrapolate to an average processing time of order 400ms per event compared to a time budget of 55ms (as constrained by the size of the round-robin buffer in shared memory and the DAQ rate). If necessary however, this budget could be made larger by a factor of approximately four by using inter-node communications to process each of the four shared memory buffers on a separate node. NO $\nu$ A's far detector is expected to have approximately 20–25 times the hit channels per event as the near detector but with a budget of up to 1s to process each

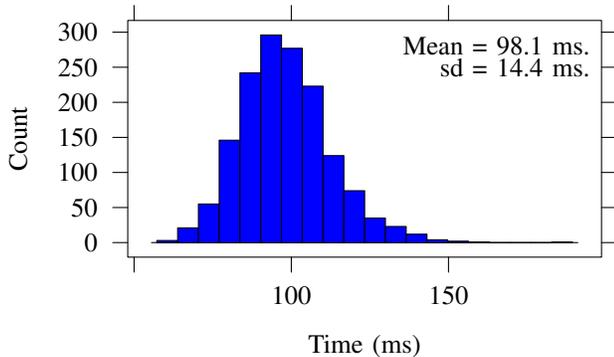


Fig. 1: The distribution of trigger algorithm processing times for a sample of events from the (incomplete) NO $\nu$ A NDOS.

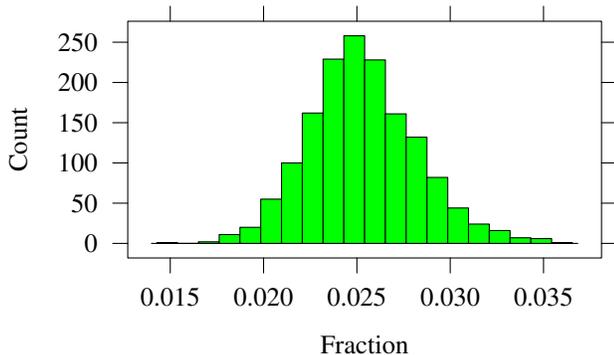


Fig. 2: Distribution of the **art** framework's data processing overhead, shown as a fraction of the total processing time for each event.

event. Similar multi-node and parallelization options are also available in this case though, and are being actively pursued.

### B. Fast Compression and High Data Rate at DarkSide-50

For the proposed direct dark matter search experiment DarkSide-50 [4], we used **artdaq** to create a prototype event builder and processing system that would require only one multicore commodity computing node to keep up with their front-end data rate. The current plan for this experiment is to provide five front-end digitizer boards, each containing eight channels for a total capacity of 40 channels, of which 38 are used. Each board will aggregate up to eight channels onto one fiber optic link that will supply data to the processing node through PCIe. The digitizers will operate at 250MHz. Events will consist of one 300 $\mu$ s sampling interval across all channels and will occur at a rate of 50Hz. To accommodate this rate, the computing system is required to handle a continuous average data rate of 300MB/s. Due to practical considerations, such as cost of permanent storage, the output stream is required to

not exceed 30MB/s. Figure 3 shows the organization of the prototype DarkSide-50 data acquisition system. We have started a number of studies to determine if a low-cost commodity computing could handle the requirements of DarkSide-50. The first two studies answer the following preliminary questions: (1) at what rate can a single node ingest data from the digitizers and perform the event-building task, and (2) at what rate can we run a compression algorithm on the data stream and what compression ratio can be achieved. Later studies will explore the question of what it will take in terms of computing resources and algorithms to achieve the 10-fold data reduction from input to output. We have established a system necessary to explore the first two questions and initial results that answer them. An interesting aspect of these tests are that what is delivered from the front-ends are all the samples in each 300 $\mu$ s window. All the algorithmic work is carried out within a standard commodity node.

Because front-end hardware does not yet exist for this experiment, we have provided components that emulate missing functions. To emulate the data link layer through the PCIe bus, we use an 8x 40Gb/s QDR InfiniBand (IB) NIC connected to an 18 port switch. For the processing system, we use a 1U 4xAMD6128 (32 total cores) system with 64GB of RAM. Using actual digitizer test stand data, we created a data generation software library capable of generating event fragments, each representing the data of a board with eight channels. The system is configured to use three nodes to emulate the data generation of each of the five front-ends (for a total of 40 channels of data). On the single processing node is configured five fragment receiving processes, each tied directly to one of the data generators through the IB network. In order to fully utilize the available 32 cores, **artdaq** is configured with five event processors, each of which constructs and processes full events from the fragments delivered by the receivers. This configuration yields five parallel full-event streams for algorithms to operate on.

We chose to use Huffman coding in our first compression algorithm, partly due to its simplicity, speed, and ability to achieve good results. Because the signal and noise can be measured and are consistent through a run, and also because the set of symbols is small, we chose to pre-calculate the tables to be used by the compression algorithm. This algorithm was trivially parallelized using OpenMP [5]. In the initial implementation, each board (or event fragment) is assigned to one thread, effectively allowing for five-way parallelism for the processing of a single event. With five available event streams, each performing five-way parallelism, we are able to utilize 25 of the 32 cores available on the machine. The algorithm achieves an average compression ratio of 4.87:1; figure 4 shows the spread of compression ratios achieved for the sample of events. With the configuration described above, we are able to operate the system at an average of 246 events/s. This rate includes both the event building overhead and the compression algorithm processing times. Figure 5 shows the distribution of times taken to process each event (*n.b.* five event streams are running in parallel on the same node). Coincidentally, this is approximately 5 times faster than the required 50Hz rate. This rate also corresponds to a data rate of 1.47GB/s.

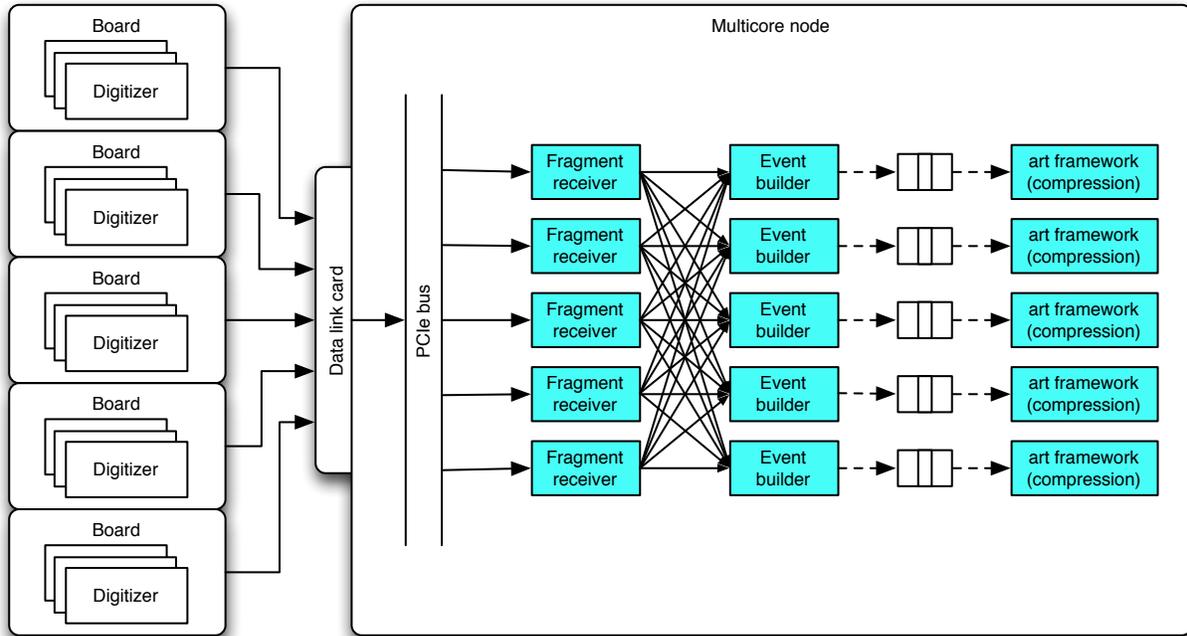


Fig. 3: The major components of the prototype DarkSide-50 event building system. Solid lines indicate inter-process communication, done mostly through MPI. Dotted lines indicate communication between different threads in the same process.

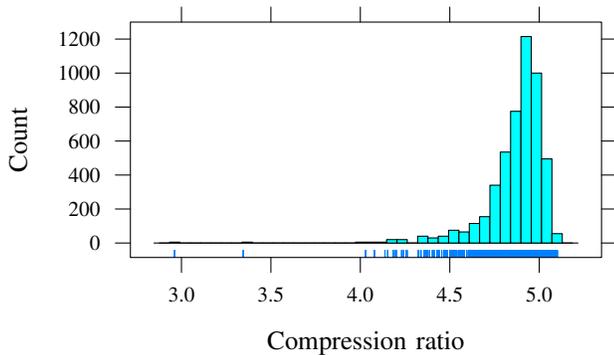


Fig. 4: Distribution of compression ratios for each DarkSide-50 event.

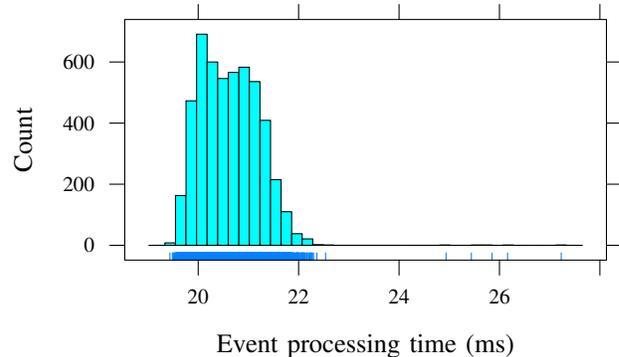


Fig. 5: Distribution of time taken to process and compress each DarkSide-50 event.

### C. Mu2e Multi-node Event Building

We have begun studying the feasibility of developing a full-rate DAQ (one which does little or no hardware filtering) event filtering system for the Mu2e experiment [6]. Providing a software system that will perform event filtering at full rate will currently require and aggregate throughput of about 30GB/s from approximately 275 front-end detector sources. The filtering software will need to reduce the input data stream to about 30MB/s. Assuming that digitized waveform data can be made available on a PCIe bus within a commodity computing node from the front-end hardware, the questions we are exploring are: how many nodes will it take to (1) handle this input data rate and (2) perform the event filtering functions. We have initial results from the first of these questions. Because

of the architectural similarity with DarkSide-50 and similar high data-rate requirement, we have been able to utilize the existing five node (each with 32 cores) IB-connected system for these tests.

The configuration of the event builder and data generators is somewhat different than the DarkSide-50 configuration. Here we use the IB network entirely for the event building and drive it using our MPI-based components. We also assume that each of the five nodes is connected to front-end hardware and reads out a portion of the detector. Each node contains (1) a data generator, (2) a data reader (detector fragment receiver or reader), and (3) an event builder that forms full events and processed them. This means that each node effectively sees

1/5 of the detector on readout and also 1/5 of the full events for processing and analysis. If the system scaled perfectly, we would expect a rate that is five times that of one machine. Partly because of the many-to-one function that being performed for event building, this is not possible. With this  $5 \times 5$  configuration, the measured average aggregate throughput is 3.4GB/s (or 700MB/s per node). Note that on this network, an individual node is able to send to another at a rate greater than 1.5GB/s. Studies have just begun to understand where the performance decrease occurs as nodes are added. One area we will explore is the MPI implementation; there are more efficient versions of MPI available than that which we are currently using. There are many other areas to optimize, including tuning MPI for better memory-to-memory copying and tuning the IB network to better handle our traffic pattern. This early result is already positive. If additional scale-up tests indicate that similar rates can be maintained, it shows that it is feasible to construct a 30GB/s data processing system at a reasonable cost.

#### IV. THE ARCHITECTURE OF ARTDAQ

**artdaq** contains three major subsystems:

- components for routing data between processes, possibly on different machines, and between different threads on the same machine, and for assembling complete events from these data;
- components that house the data being routed, which experiments use to describe their data;
- the **art** event processing framework.

Since **artdaq** is a toolkit for constructing event-building and filtering programs, it does not contain any complete DAQ applications; these would be built by each experiment to that experiment's exact requirements and preferences. The major components of **artdaq** are shown in figure 6. All user-visible

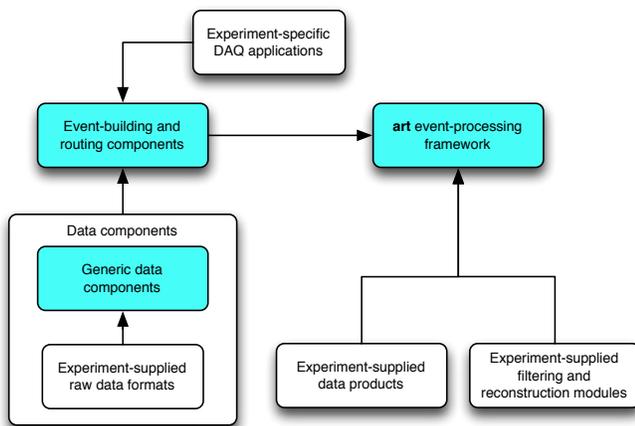


Fig. 6: Major elements of the **artdaq** architecture. The arrows indicate dependencies, *e.g.* experiment-supplied raw data formats depend upon the **artdaq** generic data components. The components in blue are those delivered as part of **artdaq**. The remaining components are supplied by the experiments that use **artdaq**.

classes in **artdaq** are defined in the `artdaq` namespace. For brevity, in this paper we provide class names without the namespace specification.

#### A. Data Model Components

The data model components of **artdaq** are written to require the least possible copying of data. To this end, we have made extensive use of some of the new features of the 2011 C++ Standard [7].

The primary data components are the classes `Fragment` and `RawEvent`. An instance of class `Fragment` represents a well-defined portion of the data from one event (likely that read by one front-end unit) as defined by the experiment. The interface of `Fragment` is sufficient to provide the information necessary for routing, and the implementation organizes the data for optimal throughput of the routing systems. Each `Fragment` is identified by a two-part identifier: an `SequenceID`, that denotes the event to which the `Fragment` belongs, and a `FragmentID`, which identifies which detector component (or components) are represented by the `Fragment`. Each experiment must choose what information from its own data is to be used to construct these two identifiers; for the experiments with which we have worked thus far, the identification has been trivial. `Fragments` also contain a type identifier, which is used to identify what type of data are being carried by the fragment. This allows experiments the flexibility of having different types of data (*e.g.*, detector data fragments, trigger blocks), while assuring that all can be handled with the same efficiency by the data-routing and event-building system.

The physical organization of the data in a `Fragment` consists of a `std::vector` of 64-bit unsigned integers. Because we are using the features of C++ 2011, this allows us to pass `Fragment` objects between software components *without* making a copy of the contained data. This allows us to keep the code simple to understand and to use correctly, risking neither memory leaks nor lack of exception safety. We deal with `Fragment` objects, not addresses in memory, but the resulting code is as efficient as if we worked with the pointers to the data directly.

The logical organization of the data in a `Fragment` consists of two parts: a header, which contains the routing information described above, and a payload, which contains the experiment-specific data carried by the `Fragment`. The first two elements of the physical `vector` contained in the `Fragment` contain the bit-packed header information; the interface of `Fragment` provides access to the data in a convenient and type-safe manner. The experiment-specific code that works with `Fragments` does so by overlaying a defined structure onto the payload part of the `Fragment`, as described in section IV-D. This system allows for payloads of arbitrarily large size; there are no compile-time limits set on the sizes of the experiment data.

`Fragment` objects are persistible through the **art** framework's persistency mechanism. This means that any experiment that uses `Fragment` in the definition of its raw data classes automatically obtains a means to write those data to the same type of file that is read by the experiment's offline system. In addition to providing the means of persistence for detector data, this also means that simulations can create data files in the same format as the experiment's raw data; thus the output of such a simulation can easily be fed through the data processing algorithms that will be applied to the detector data,

to help verify correct behavior of those algorithms, and for performance tuning.

The event-building process collects `Fragments` to build `RawEvents`, making use of the features of C++ 2011 to avoid copying the underlying data. The `RawEvent` can contain an arbitrary number of `Fragments`; again, there is no compile-time limit set. Due to the flexibility of the `Fragment`, the `RawEvent` can contain many different types of experiment-specified detector data. The event-building code that deals with `RawEvents` and `Fragments` does not need to be modified if new experiment-specific data types are added to an existing system.

### B. Event Building Components

**artdaq** makes use of the Message Passing Interface (MPI) [8] to create a multi-process, potentially distributed, event-building program. The use of MPI allows us to take advantage of high-performance network drivers written for the supercomputing community. We also obtain the flexibility of being able to move different computational tasks to different nodes with just a change in our configuration scripts, and with no need to recompile the application. This can allow a running experiment great flexibility in responding to failure of hardware. It also makes measuring the performance of different program configurations a relatively simple task; one needs only to change a configuration file and re-run the test program to observe the effectiveness of different process layouts.

An **artdaq** event-building and filtering program contains three processing layers.

- The *fragment receiver* layer receives data from the experiment's front-ends (using whatever communication mechanism the experiment chooses), and is responsible for sending the data to the correct *event builder*, through MPI.
- The *event builder* layer receives data from the fragment receivers, collating them into complete events. Complete events are then sent to another thread in the same process for *event processing*.
- The *event processing* layer runs the **art** event-processing framework, which performs the necessary tasks (*e.g.*, event filtering, data compression). The data are optionally written to persistent storage by **art**.

An **artdaq** event building program is configured at run-time to contain a number  $N$  of fragment receiver processes, and a number  $M$  of event builder processes; there is no requirement that  $N = M$ . Each fragment receiver receives a stream of `Fragments` from the same detector component(s), and thus with the same `FragmentID`, and routes each `Fragment` to the event builder process responsible for handling the event to which that `Fragment` belongs, based on a round-robin algorithm. We provide the class `SHandles` to encapsulate the coordination of multiple MPI buffers and to automatically record some performance metrics.

The event builder processes receive the `Fragments`; we have provided a class `RHandles` to manage multiple MPI buffers used for reading and to record additional performance metrics. We have taken care that once a `Fragment` has

been read from the MPI buffer, no additional copying of the underlying data is ever done, regardless of the number of times control of the `Fragment` is passed between different functions and even between different threads of the process.

The most important class in the event builder processes is `EventStore`, which is responsible for managing the thread that runs the **art** event processing framework, for accumulating complete events, and for sending complete events to the thread that runs **art**. The `EventStore` is configured at run-time to know the number of `Fragments` comprising a complete event. `Fragments` making up a particular event may come out of order, and some `Fragments` for a later event may show up in the event building layer before all the `Fragments` of an earlier event. The event builder layer aggregates the `Fragments` it receives into `RawEvents`. When it determines that the receipt of a `Fragment` has completed a specific event, the event builder layer moves that `RawEvent` from its internal cache of incomplete events and puts it onto a concurrent queue, shared with another thread in the same process, which is responsible for running the **art** event-processing framework. Separate threads of execution are used so that the thread that is building events can proceed at full pace even if the occasional event takes a longer-than-average time to process in the thread that is running **art**.

An orderly program shutdown is initiated when each detector component has sent an end-of-data `Fragment`. Upon receipt of an end-of-data `Fragment`, each fragment receiver process sends an end-of-data `Fragment` to each event builder process. When an event builder process has seen as many end-of-data fragments as it expects, it puts an end-of-data marker onto the concurrent queue, to communicate the end-of-data status to **art**, and then awaits the termination of the thread running **art**. That thread terminates when the **art** has completed processing any events buffered up in the queue, ending with the end-of-data marker that lets **art** know no more events are coming. The current code is in prototype status, and so we have not yet implemented the other control flow features necessary for a fully functional DAQ.

The event building system keeps monitoring statistics at a number of critical points. The user can obtain some important performance results without modifying code. We note this is prototype code, and we are still working on the optimal set of points to monitor.

### C. The **art** Framework

The **art** framework is used to execute experiment-supplied algorithms for triggering, data compression, reconstruction, and writing of data files. It provides configuration ability through use of the Fermilab Hierarchical Configuration Language (FHiCL) [9]. The framework can run an arbitrary collection of algorithms, decided at configuration time, not at program compilation or linking time. Experiment-supplied algorithms are implemented by writing **art** modules, which are classes that implement one of a handful of interfaces specified by **art**. Each module is built into separate dynamically loaded library. Based on the contents of the FHiCL configuration file, **art** loads the libraries necessary to run the named modules.

Algorithms can obtain read-only access to data products in the event, and add new data products of their own construction. **art** also supplies the scheduling features that allow different combinations of algorithms to be run on different events, based on pass-or-fail decisions made by experiment-supplied *filter* modules, all without rebuilding the application.

Provenance information is automatically stored for all data products. FHiCL allows experiments to provide “standard” configurations for all modules, and for a user to partly or entirely override a standard configuration on a case-by-case basis. The automatic provenance tracking records the parameters that were actually used to configure each module (regardless of whether they were the experiment defaults or the user-level overrides), and associates those parameters with the data product or products made by each module.

The framework has monitoring points around the invocation of each module, so event-by-event timing results can be obtained for every module. Additionally, simple memory usage analysis can also be performed, helping to identify any algorithms with uncontrolled memory usage.

**art** also provides a set of run-time-configurable policies for reacting to exceptions thrown by modules, and exception classes for experiments to use in their own code. The data in the exception communicates the *kind* of error that has been encountered (*e.g.*, observation of data corruption). The policy determines how the framework will respond to that kind of error. Among the choices are skipping the processing of the module that encountered the error, skipping the processing of that event entirely, and gracefully shutting down the entire program.

#### D. What The Experiment Provides

The **artdaq** toolkit, and the **art** framework that it relies upon, provide the generic, *i.e.*, experiment-neutral, parts from which an experiment can construct an event building and filtering system. Individual experiments make use of the provided infrastructure in several different ways.

At the highest level, individual experiments using **artdaq** must still write their own experiment-specific DAQ applications: **artdaq** is a prototype for a *toolkit*, not a collection of complete applications. The needs of experiments are sufficiently diverse that it is unfeasible for us to deliver complete applications to the experiments. Instead, our groups work with the experiments to help them produce software matching their specific needs.

Experiments must, of course, define the format of their own raw data objects. In order for the data products they define to be consistent with **artdaq**, it is required that the data of the individual product be contained in a contiguous series of bytes; this is because the data of the `Fragment` is a continuous sequence of 64-bit unsigned integers (contained in a `vector`). It is then straightforward (and strongly recommended) to write utility classes to handle the technicalities of reading and writing the data structure, and applying the data product overlay to the `Fragment`. This localizes the low-level bit manipulations to a limited number of classes, rather than having it be visible in many places in the code that uses these data. As a result, verification and modification of the code is simpler.

As part of their use of the **art** framework (both for offline and online purposes), experiments are responsible for defining their own data types to describe reconstruction results. These data products must conform to the restrictions imposed by the persistency system used by **art**. There is no required class inheritance involved, and experiments do not need to implement the functions that write the data to the **art** data storage format. For data products that are noted as non-persistable, these requirements are relaxed.

Also as part of their use of the **art** framework, experiments are responsible for defining their own reconstruction and filtering modules. In the terminology of **art**, *reconstruction* includes all data transformation: unpacking or decompression of data, translating from “electronics coordinates” to “physics coordinates”, applying calibrations, as well as what is typically thought of as reconstruction, *e.g.* track reconstruction. The framework provides a few base classes from which experiment-produced modules must inherit; this allows the modules to be dynamically loaded and invoked by framework without requiring recompilation of the framework.

## V. FUTURE PLANS

Short-term plans for the advancement of this work include:

- investigation of possible improvements in parallelization of NO $\nu$ A’s trigger algorithm, and use of inter-node communication;
- for DarkSide-50, further parallelization of and possibly other improvements to the compression algorithm, and if necessary, data reduction by removal of “uninteresting” regions to accommodate storage constraints;
- further study of the scaling performance for our Mu2e simulation, including investigation of several alternative MPI implementations.

In the medium to long term, we are continuing our work to introduce schedule-level parallelism (the ability to analyze multiple events at the same time in one process) to **art**. We are also planning to provide the facility to run GPGPU-enabled algorithms. Module-level parallelism is also of interest to the NO $\nu$ A DDT use case. For DarkSide-50, we plan to develop a FPGA/PCIe based data generator for experimental simulation at greater data rates. We are also starting to work with  $\mu$ BooNE and the muon g-2 experiments to apply the **artdaq** paradigm to their particular needs and situations. Given sufficient interest and available effort, the **artdaq** prototype package will transition into a complete, tested and polished toolkit for use by experiments.

## VI. SUMMARY

The initial prototype event-builders written using the prototype **artdaq** have been able to achieve adequate (in the case of DarkSide-50, much more than adequate) data throughput in a very short time, with limited development resources and with very modest demands placed upon the experiments’ developer resources, using a modest amount of commodity computing hardware.

Using tools (MPI) commonly used in the HPC community, we have been able to quickly move forward to building fully-configurable distributed multi-process programs, without having to write any low-level code.

We have demonstrated portability between offline and online software for testing and ease of debugging, and have established an environment in which we can carry on with our R&D tasks. We have already generated some enthusiasm in those in our local community who have been surprised by the speed of development and the resulting performance of the system.

#### REFERENCES

- [1] The home page for the **art** framework is <https://cdcv.sfnal.gov/redmine/projects/art>.
- [2] “NOvA Neutrino Experiment”, available at <http://www-nova.fnal.gov>.
- [3] “MINOS Experiment and NuMI Beam Home Page”, available at <http://www-nu.mi.fnal.gov>.
- [4] The DarkSide-50 Experiment Proposal is available at [http://www.fnal.gov/directorate/program\\_planning/Nov2009PACPublic/DarkSideProposal.pdf](http://www.fnal.gov/directorate/program_planning/Nov2009PACPublic/DarkSideProposal.pdf).
- [5] **OpenMP Application Program Interface**, <http://www.openmp.org/mp-documents/OpenMP3.1.pdf>.
- [6] “Mu2e Experiment”, available at <http://mu2e.fnal.gov>.
- [7] The C++ programming language is specified by the ISO standard *ISO/IEC 14882:2011 Information Technology—Programming Languages—C++*.
- [8] “Message Passing Interface Forum”, <http://www.mpi-forum.org>.
- [9] The home page for FHiCL is <https://cdcv.sfnal.gov/redmine/projects/fhicl>.