

# artdaq: An Event-Building, Filtering, and Processing Framework

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 commercial off-the-shelf (COTS) 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 are using 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 event-processing framework shared by new experiments at Fermilab. In an *artdaq* program, the digitized data are transferred into computing nodes using commodity Peripheral Component Interconnect Express (PCIe) cards, event fragments are transferred between distributed processes using MPI, and assembled into complete events. These events are then processed by a configurable selection of user-specified algorithms, commonly including filtering and compression algorithms, using the *art* event-processing framework.

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 both at Fermilab and elsewhere.

**Index Terms**—Data acquisition, concurrent programming, distributed programming.

## I. INTRODUCTION

THE *artdaq* project has been established to design and develop a generic toolkit for the construction of efficient and robust event<sup>1</sup> building, filtering and analysis programs within data acquisition systems for future medium-scale experiments, such as those planned at Fermilab for the next decade. These experiments have fewer collaborators than recent and current collider experiments, and so cannot easily afford to develop and maintain as much customized infrastructure 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 these experiments to concentrate their efforts 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 commercial off-the-shelf (COTS) computers, rather than (as is traditional in the field) 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 COTS computers have multiple cores, *artdaq* is designed to take advantage of the inherent parallelism of the event-building process, to perform event-filtering of independent events in parallel, and to make the development of modular event-processing algorithms that internally use parallel programming techniques convenient. 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 we have worked, the development of online and offline event-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 historically has been a time-consuming challenge. To alleviate this problem *artdaq* makes use of the *art* [1] event-processing framework, already in offline use by many of the upcoming 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 filter system can be verified 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 every 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. 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*.

Manuscript received June 11, 2012; revised January 18, 2013.

This work was supported in part by the U.S. Department of Energy, Office of Science, HEP, Scientific Computing.

All authors are with the Scientific Computing Division, Fermi National Accelerator Laboratory, Batavia, IL, USA. Email: biery@fnal.gov, greenc@fnal.gov, jbk@fnal.gov, paterno@fnal.gov, ron@fnal.gov.

<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 by the modular algorithms of the event-processing framework.

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 filtering algorithms (or other data processing algorithms, such as data compression algorithms needed by some experiments to reduce the bulk of the data to be stored) will be run. This includes experiments that have no data filtering in front-end hardware and event-processing times that vary widely from event to event. Depending on the computing resources available to an experiment, it may be beneficial to use the same computing nodes for both event-building and filtering.

In a software system that contains both shared-memory and distributed parallelism, the optimal distribution among computing node of the processes of the system depends on the amount and types of computing hardware available, the amount of data movement necessary, and the exact nature of the computing to be done. In order to make convenient the testing needed 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.

### B. Filter Algorithm Execution

Many experiments want to perform as little event filtering as possible in hardware, in order to obtain as much flexibility as possible for modification of algorithms and thresholds used in this filtering. Event filtering in software provides the opportunity to obtain this flexibility. One of the goals for our project is to find to what extent modern computing hardware provides the computing capacity necessary to do the work.

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 filter algorithms in the same program, on the same event stream.

Because of the degree of sophistication of filter algorithm software, experiments want to enable all physicists interested in working on development and testing of these algorithms to do so. Thus experiments want to be able to run these 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.

### C. Single-Node Processing Capacity

Modern experiments need to make use of modern computing hardware, which means taking advantage of multicore platforms. In an era of tight budgets, it is critical to take full advantage of the most affordable COTS computing resources

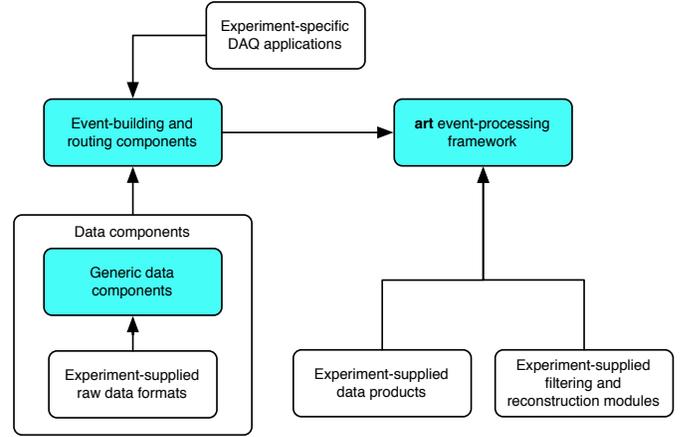


Fig. 1. Major elements of the **artdaq** architecture. The arrows indicate dependencies, e.g. experiment-supplied raw data formats depend upon the **artdaq** generic data components. The colored components are those delivered as part of **artdaq**. The remaining components are supplied by the experiments that use **artdaq**.

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 filtering tasks, and which utilizes multicore hardware and high-throughput networks to their greatest advantage.

## III. THE ARCHITECTURE OF ARTDAQ

**artdaq** is a set of C++ libraries and programs, i.e., a *toolkit*, for use in the construction of event-building, filtering, and processing programs as part of a DAQ system. **artdaq** contains three major subsystems:

- software components for routing data between threads within a process, and between different processes, possibly on different machines, and for assembling complete events from these data;
- software components that encapsulate the data being routed, which experiments specialize to provide type-safe access to the data being routed;
- the **art** event-processing framework, to allow loading and execution of experiment-specific software modules for processing the data being routed.

Since **artdaq** is a toolkit, it does not contain any complete DAQ applications; such applications are to be built by each experiment to that experiment’s specific requirements, constraints, and preferences. The major components of **artdaq** are shown in Fig. 1. All user-visible 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 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: a `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, trigger blocks, or end-of-run markers), while ensuring 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. This does not demand the form of the experiment's raw data matches; we describe the restriction on the raw data in section III-E. We make use of so-called *move semantics* introduced in C++ 2011 [2] to allow 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. We describe this in more detail in section III-B.

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 `std::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 III-E. 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 may be written to disk 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`, again making use of move semantics 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. Advantages of C++ 2011

An important goal of `artdaq` is to facilitate the writing of robust and efficient code. One key to efficiency is the minimization of copying of data. In C code, copying of data is avoided by passing pointers to values, rather than by passing the values themselves. However, C provides no automatic memory management, so great care must be taken to avoid the introduction of memory leaks. Robust code is thus typically obscured by the quantity of error-handling and memory-management code required; modification of such code is error-prone. C++ provides a mechanism (referred to as *resource acquisition is initialization*, or RAII [3]), which, through the use of the strictly-defined lifetimes of stack objects, removes the need for the programmer to explicitly manage memory. C++ container classes and class templates (e.g. `std::vector`) encapsulate this use. But naive use of these classes, while producing robust code, can introduce unacceptable performance overheads from the copying of data. The 2011 C++ standard introduced several language features, most importantly *rvalue references* and *move semantics*, that allow simple and maintainable code to rival the efficiency of C. The following example code shows a simplified portion of the `artdaq` code, in which ownership of the data managed by a `Fragment` is given over to a `RawEvent`, rather than being copied.

If written in C, this code might appear as the following. The event data would be organized into a `struct` that organized the fragment data; the data for each fragment is kept in an array, with each array being of arbitrary size:

```
typedef struct {
    unsigned long nfrag;
    unsigned long *sizes;
    unsigned long **data;
} raw_event;
```

The function `build_event` has the task of accepting the fragments (and the additional data of their sizes, and the number of fragments), building the `raw_event`, and releasing the calling code from ownership of the data it has passed into the function:

```
void build_event(unsigned long ***fragments,
                unsigned long **sizes,
                unsigned long nfrag,
                raw_event *e) {
    e->nfrag = nfrag;
    e->sizes = *sizes;
    e->data = *fragments;
    *sizes = 0;
    *fragments = 0;
}
```

In this code, the `build_event` function passes ownership, and thus responsibility for memory management, from the caller of `build_event` to the user of the `raw_event` structure. This code is efficient, but is not robust. Most importantly, there is no check that the structure `e`, on input to the function, does not already reference some memory; any such memory referenced would be leaked when the data of the struct are overwritten. Using the GNU C compiler (gcc version 4.7.1, with `-O3` optimization), this code produces the following assembly code:

```
00:  mov    (%rsi),%rax
03:  push  %rbp
04:  mov   %rdx,0x10(%rcx)
08:  mov   %rsp,%rbp
0b:  mov   %rax,0x8(%rcx)
0f:  mov   (%rdi),%rax
12:  movq  $0x0,(%rsi)
19:  mov   %rax,(%rcx)
1c:  movq  $0x0,(%rdi)
23:  leaveq
24:  retq
```

We note this is only 11 instructions.

The approximate C++ equivalent of this code, similar to the code in `artdaq`, is shown below. First, we introduce a few `typedefs` to reduce the amount of typing required.

```
typedef std::vector<unsigned long> Fragment;
typedef std::vector<Fragment>      FragVec;
typedef std::unique_ptr<FragVec>   FragVecPtr;
```

Next, we define the `RawEvent` structure, which contains (and thus controls the lifetime of) the `Fragments`.

```
struct RawEvent { FragVecPtr data; };
```

Finally, have the C++ version of `build_event`.

```
void build_event(FragVecPtr&& frags,
                RawEvent& e) {
    e.data = std::move(frags);
}
```

This code makes use of three features added to C++ in the 2011 standard: `std::unique_ptr`, `std::move`, and *rvalue references*. The argument named `frags` is passed by rvalue reference; this tells the compiler that this argument can only be bound to an expression that is an rvalue reference, and furthermore that the compiler is then free to “move” the resources owned by that expression to being owned by `frags`; the `std::move` then tells the compiler it is free to pass ownership of the memory controlled by `frags` to the `RawEvent`’s data member `data`. The data are not copied; only ownership is passed. Unlike the C code above, this code is robust; if the input `RawEvent` contained `Fragments`, they would be deleted before the new pointer was assigned. The assembly language produced from this code (by the GNU C++ compiler of the same version, with the same optimization setting) is shown below.

```
00:  push  %rbp
01:  mov   %rsp,%rbp
04:  push  %r13
06:  push  %r12
08:  push  %rbx
09:  sub   $0x8,%rsp
```

```
0d:  mov   (%rdi),%rax
10:  movq  $0x0,(%rdi)
17:  mov   (%rsi),%r13
1a:  mov   %rax,(%rsi)
1d:  test  %r13,%r13
20:  je    70
# 25 instructions elided; they will be
# called only if the incoming event is
# non-empty.
70:  add   $0x8,%rsp
74:  pop   %rbx
75:  pop   %r12
77:  pop   %r13
79:  pop   %rbp
7a:  retq
```

We have elided 25 instructions, which are called only if the incoming `RawEvent` is non-empty—which is not the usual case. The main path of the C++-generated assembly code is 7 instructions longer than the C code. This demonstrates that the C++ language does not introduce any great deal of complexity behind the scenes. However, the C++ code is arguably more succinct, and unarguably more robust. Certainly, the C code can be made robust—but at the cost of simplicity, clarity, and maintainability.

### C. Event-Building Components

`artdaq` makes use of the Message Passing Interface (MPI) [4] 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 gives a running experiment great flexibility in responding to failure or reassignment of computing 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-building* 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 whatever tasks the experiment needs to perform on the data stream. Common examples include event filtering, track finding, and 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 reads data from a specific detector component (or set of components), and writes those data to a `Fragment`; the `FragmentID` assigned to that `Fragment` identifies the portion of the detector data which the `Fragment` carries. The fragment receiver is also responsible for looking into the detector data to find the (experiment-specific) data that are used to identify the event to which these data belong; this is used to create a `SequenceID` for this event, and which is used in a round-robin to direct the `Fragment` to the event builder responsible for handling that event. We provide the class `SHandles` to encapsulate the coordination of multiple MPI buffers used in sending, and to automatically record some performance metrics.

Each event-building process receives all the `Fragments` from the subset of events bound for it, possibly out-of-order, and is responsible for building complete events from them. We have provided a class `RHandles` to manage multiple MPI buffers used for reading and to record additional performance metrics. Using techniques like those described in section III-B, we have taken care that once a `Fragment` has been read into 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-building processes is `EventStore`, which is responsible for managing the thread that runs the `art` event-processing framework (described in section III-D), 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-building layer aggregates the `Fragments` it receives into `RawEvents`. When it determines that the receipt of a `Fragment` has completed a specific event, the `EventStore` layer removes that `RawEvent` from its internal cache of incomplete events and sends it to 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 fragment-receiver process identifies an end-of-data condition. Each of these processes then sends an end-of-data `Fragment` to each event-building process. When an event-building process has seen as many end-of-data fragments as it expects, it sends an end-of-data “event” to the thread running `art`, and then awaits the termination of that thread. That thread terminates when the `art` has completed processing any events it has buffered, ending with the end-of-data “event”, which lets `art` know no more events are coming.

The event-building system keeps monitoring statistics at a number of critical points. These include statistics regarding the number, size, and time taken for MPI data transfers, the number

of incomplete events currently in the `EventStore`, and the number of completed events sent to `art`. We are currently implementing a system that can report all these statistics asynchronously to a DAQ system control program.

#### D. The `art` Framework

The `art` framework is used to execute experiment-supplied algorithms for both online and offline tasks. Such tasks vary by experiment, but typically include filtering, reconstruction, data compression, and writing of data files. It provides configuration ability through use of the Fermilab Hierarchical Configuration Language (FHiCL) [5]. The framework can run an arbitrary collection of algorithms, chosen 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 a 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 experiment-defined 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 or excessive 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. For the most severe errors, gracefully shutting down the entire program, and thus avoiding improperly truncated files, broken network connections, *etc.*, is an option.

#### E. 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 *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 contiguous sequence of 64-bit unsigned integers (contained in a `std::vector`). The sequence of integers is not interpreted in any way by **artdaq**; neither packing nor unpacking of data is done. It is 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**. 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.

#### IV. GUIDING USE CASES

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

The *NO $\nu$ A* [6] experiment at Fermilab will search for the oscillation of muon neutrinos to electron neutrinos. *NO $\nu$ A* will have two detector components, a Near Detector sited at Fermilab, and a Far Detector sited at Ash River, MN. The ND will comprise more than 18 thousand readout channels, and the FD 340–368 thousand channels. A prototype ND, comprising up to 12 thousand instrumented channels, is being built and is in partial operation. The *NO $\nu$ A* design features a free-running, dead-time free, continuous readout. This system will collect data at 2 GB/s, and buffer up to 20 seconds worth of data. The data must be searched to correlate observed detector energy deposits with beam spills from the NuMI [7] beam facility, identifying *events* that consist of those hits in a time window of 5 ms. The *NO $\nu$ A* event-building system predates the development of **artdaq**.

*NO $\nu$ A* is investigating the design and development of a Data Driven Trigger (DDT). This gave us the opportunity to verify that the filter portion of **artdaq** can readily be adopted into an existing online software system, and to demonstrate to the experimenters that algorithms developed in the context of the **art** framework, already used by the *NO $\nu$ A* offline processing, could easily be deployed online.

In the prototype DDT, the data are read from the event-building buffer at full rate, and sent to **art**, which is configured to run analysis modules that examine the data to identify event topologies of interest. The analysis result—an accept or reject decision—is then fed back into the experiment’s global triggering system to form a data-driven decision for a whole event.

The first physics algorithm to be completed was a track finding algorithm based on the Hough transform [8]. An **art** module with a preliminary implementation of the algorithm was developed and tested in the offline environment, and then integrated in the online DDT environment. This module was then used successfully in the live data stream from the portion of the *NO $\nu$ A* Near Detector that was complete at the time of the testing. The implementation is amenable to many optimizations, and *NO $\nu$ A* scientists will continue to improve it. The successful exercise of the preliminary implementation in the online DDT environment has demonstrated that study and optimization of the algorithm in the offline environment will yield software that can be directly deployed in the online environment.

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

*DarkSide-50* is a direct dark matter search experiment located at the Laboratori Nazionali del Gran Sasso, in Italy [9], [10]. For *DarkSide-50*, we used **artdaq** to create a prototype event-building and processing system that would require only one multicore COTS computing node to keep up with their front-end data rate. At the time we created the prototype, the plan for this experiment was to utilize five front-end digitizer boards, each containing eight 12-bit ADC channels for a total capacity of 40 channels, of which 38 were to be used. Each board would aggregate up to eight channels onto one fiber optic link that would supply data to the processing node through a Peripheral Component Interconnect Express (PCIe) bus. The digitizers would operate at 250 MHz. Events would consist of one 300  $\mu$ s sampling interval across all channels, yielding 1.2 MB of data per board, and would occur at a rate not to exceed 50 Hz. To accommodate this rate, the computing system is required to handle a continuous average data rate of 300 MB/s. Due to practical considerations, such as cost of permanent storage, the output stream is required to not exceed 30 MB/s. Fig. 2 shows the organization of this prototype. Because front-end hardware was not yet available to us, we provided components that emulate missing functions. To emulate the data link layer through the PCIe bus, we use an 8x 40 Gb/s QDR InfiniBand (IB) network interface controller connected to an 18-port switch. For the event-processing system, we used a single node containing 4xAMD6128 chips (32 total cores), with 64 GB of RAM. Using actual digitizer test stand data, we created a data generation software library capable of generating event

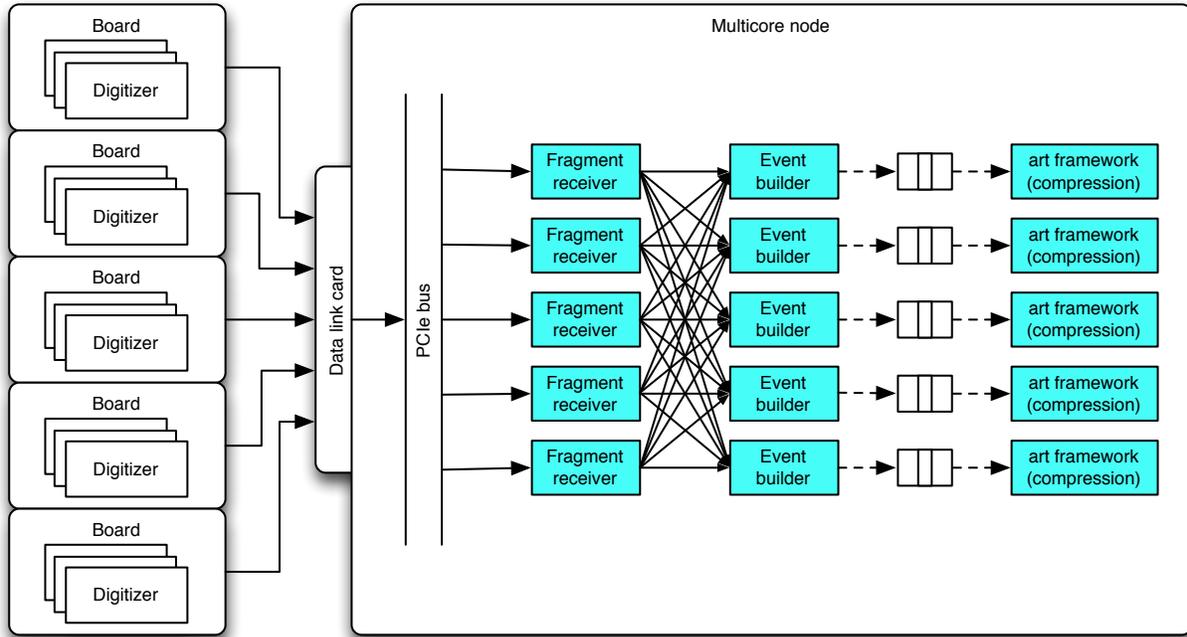


Fig. 2. The major components of the prototype DarkSide-50 event-building system. Solid lines indicate inter-process communication, done mostly through MPI. Dashed lines indicate communication between different threads in the same process.

fragments, each representing the data of a board with eight channels. We used three more nodes, each identical to the event-processing node, to emulate the data generation of the five front-ends (for a total of 40 channels of data). On the single processing node we ran 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 on the event-processing node, we configured **artdaq** with five event processors. This configuration yields five parallel full-event streams for algorithms to operate on.

We used this system to evaluate the rate at which a single node can ingest data from the digitizers and perform the event-building task, the rate at which we can run a compression algorithm on the data stream, and the compression ratio that can be achieved.

We chose to use Huffman coding [11] in our first compression algorithm, partly due to its simplicity, speed, and ability to achieve reasonable compression. We parallelized the algorithm using OpenMP [12], using one thread for the compression of the data from each board, yielding 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.

With this configuration, we are able to operate the system at an average of 246 events/s, while achieving an average compression ratio of 4.9:1. This is approximately 5 times faster than the required 50 Hz rate.

### 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 [13]. Providing

a software system that will perform event filtering at full rate will currently require an aggregate throughput of about 30 GB/s from approximately 275 front-end detector sources. The filtering software will need to reduce the input data stream to about 30 MB/s. Assuming that digitized waveform data can be made available on a PCIe bus within a COTS 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 for 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 a system of five nodes (of the same configuration described earlier) of the IB-connected system for these tests.

The configuration of the event builders 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 simulate each of the five nodes being connected to the experiment's front-end hardware by having each node run a data-generator process. Each data-generator process sends its data directly to a single fragment-receiver process on the same node. Each node also runs an event-builder process. Each fragment receiver sends fragments to all event builders. 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, and without tuning the MPI implementation, we measured an average aggregate throughput of 3.6 GB/s (or approximately

## V. 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 COTS computing hardware.

Using tools commonly used in the HPC community, but not typically used in DAQ systems (e.g. MPI, OpenMP, and InfiniBand networking), we have demonstrated the feasibility of building fully-configurable, distributed, multi-process programs, without having to write any low-level code, and requiring a very limited amount of experiment-specific 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.

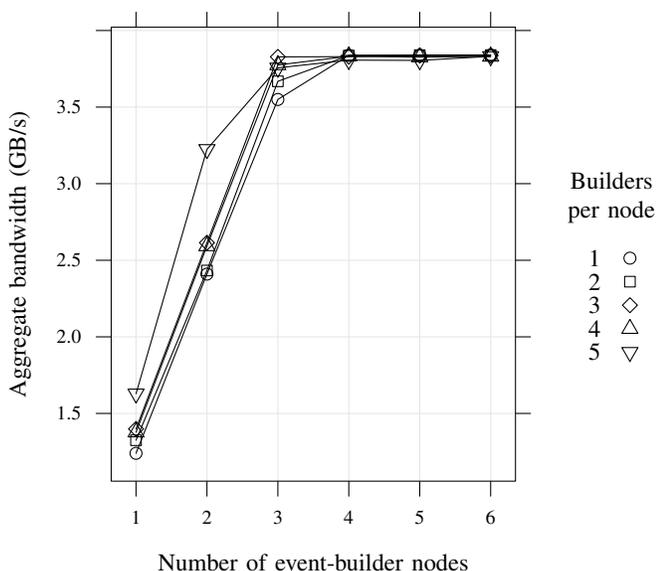


Fig. 3. Aggregate bandwidth of the prototype DarkSide-50 DAQ system, for varying numbers of event-building nodes and number of event-builder processes per node.

730 MB/s per node).

This early result is encouraging. If additional scale-up tests indicate that similar rates can be maintained, it shows that it is feasible to construct a 30 GB/s data processing system at a reasonable cost.

In order to measure the degree to which **artdaq** allows experiments to have useful access to the computational power of the multiple cores available on modern platforms, we performed tests using the computing nodes purchased for the DarkSide-50 DAQ. These consisted of four Intel-based machines, each with 2xIntel Xeon E5-2620 chips (12 total cores per node), and six AMD-based machines, each with 4xAMD6212 chips (32 total cores per node), all connected on a QDR InfiniBand network. We configured the system according to the design of the DarkSide-50 DAQ, using the four Intel-based nodes to send data (running fragment-receiver processes) and used the AMD nodes to run the event-builder processes. We varied both the number of nodes used for event building, and the number of event-builder processes run on each node, and measured the aggregate throughput of the system in each configuration. Our measurements are shown in Fig. 3. The plateau bandwidth is 3.8 GB/s; this is reached using four nodes for event building, or with just three nodes if four or five event-builder processes are run per node. Using additional event-builder processes is not seen to slow the data handling. Thus **artdaq** allows experiments to take advantage of the additional cores on event-building nodes either through process-level parallelism (which does not require experiments to implement algorithms in a thread-safe manner), or thread-level parallelism, for those experiments who have the resources and expertise to attend to thread safety in the implementation of their algorithms.

## REFERENCES

- [1] C. Green *et al.*, “The **art** framework.” *J. Phys. Conf. Ser.*, to be published.
- [2] ISO, “Information Technology—Programming Languages—C++,” 2011, ISO/IEC 14882:2011.
- [3] B. Stroustrup, *The design and evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994.
- [4] “The MPI-2 standard.” [Online]. Available: <http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm>
- [5] “The **FHiCL** home page:.” [Online]. Available: <https://cdcvns.fnal.gov/redmine/projects/fhicl>
- [6] R. E. Ray, “The NOvA experiment,” *J. Phys. Conf. Ser.*, vol. 136, p. 022019, 2008.
- [7] S. E. Kopp, “The NuMI beam at FNAL and its use for cross-section measurements,” 2007. [Online]. Available: <http://arxiv.org/pdf/0709.2737.pdf>
- [8] R. Duda and P. Hart, “Use of the Hough transformation to detect lines and curves in pictures,” *Comm. ACM*, vol. 15, pp. 11–15, January 1972.
- [9] “Darkside: A depleted argon dark matter search?” [Online]. Available: [http://www.fnal.gov/directorate/program\\_planning/Nov2009PACPublic/DarkSideProposal.pdf](http://www.fnal.gov/directorate/program_planning/Nov2009PACPublic/DarkSideProposal.pdf)
- [10] A. D. Rosso, *The DarkSide of Gran Sasso*, ser. CERN Courier. IOP Publishing, May 2012. [Online]. Available: <http://cerncourier.com/cws/article/cern/49692>
- [11] D. Salomon and G. Motta, *Handbook of Data Compression*. Springer London, 2010.
- [12] O. A. R. Board, “OpenMP application program interface version 3.1,” July 2011. [Online]. Available: <http://www.openmp.org/mp-documents/OpenMP3.1.pdf>
- [13] F. Cervelli, “The Mu2e experiment at Fermilab,” *J. Phys. Conf. Ser.*, vol. 335, p. 012073, 2011.