6/26/13

Level 3 of the K0TO DAQ System

Nikola Whallon

About This Document

This document is aimed at giving an over of Level 3 (L3) of the K0TO data acquisition (DAQ) system. It specifically covers the online software, l3_daq (called “meow” in May 2013...). This document and all other L3 documentation and software mentioned here or otherwise are version controlled by git, and accessible via codeblue (http://codeblue.umich.edu). To use git and the L3 git repository, see the presentation “K0TO Using Git.pdf.” The current (6/26/13) L3 documentation, including this document and “K0TO Using Git.pdf,” are also contained on both the Indico page (http://kds.kek.jp/) and K0TO page (http://hep0.physics.lsa.umich.edu/K0TO/).

Purpose

The purpose of L3 of the K0TO DAQ system is ultimately to store K0TO event data on computer harddrives. This is achieved using 29 nodes of Mandolin, a cluster of 40 nodes. Each node runs the program l3_daq which collects packets of event data sent by the Level 2 (L2) trigger boards, builds those individual packets into whole events, and stores the event data on its harddrive. This event data can then be sent to KEK CC or other computers for further analysis.

Features

l3_daq not only has the capabilities of building and storing events with minimal data loss, but also the capacity to cut out events based on basic cluster information and compress events to ~25% of their raw size. All of these features can be tuned, tweaked, and toggled in the easy-to-edit configuration file, k0to_config.h.

Design

The basic design of l3_daq is based around 4 event pools - event_pool_collecting, event_pool_cutting, event_pool_compressing, and event_pool_storing - and 4 processing threads, created using pthread routines. The 4 threads run simultaneously, collecting, cutting, compressing, and storing the events in the respective event pools. After a spill of events is over, indicated by a packet from the next spill being captured, the pointers to the event pools swap, such that the recently collected events become the next events to cut, and the recently cut (or rather, not-cut) events become the next events to compress, etc. Semaphores are used to control these pools and threads to make sure that no 2 threads alter data from the same event pool.

The Collecting Thread

The collecting thread captures packets using pcap routines and builds them into events by grouping the packets based on their timestamp and ordering the packets based on their L2 trigger board number and fiber number. These events are stored in event_pool_collecting.

Relevant Functions

I will explain the functionality of the most relevant functions executed on this thread. Since there are several, and data is passed linearly from one to the next, I have organized them below based on both what files the functions declaration and definition lie in, and their chronology of execution.

In pkt_receiver.h/pkt_receiver.cpp, the following functions are relevant:

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)

This function is called everytime pcap receives a packet. It checks to see if the packet belongs to an event. If the length of the packet corresponds to that of an event packet, the packet is sent to pkt_process(const char * payload) to be processed, if not, the packet is ignored.

void pkt_process(const char * payload)

This function processes an event packet. It calculates the timing and position of the packet, which are used later to build an event. This function also looks to see whether or not the packet is part of a new spill. If the packet is part of a new spill, all the event pools (collecting, cutting, compressing, and storing) are swapped. After this check, the packet is sent to the EventManager function ProcessData(uint16_t * data, uint32_t timing, uint32_t pos) to be copied into an event buffer.

In EventManager.h/EventManager.cpp, the following functions are relevant:

ProcessData(uint16_t * data, uint32_t timing, uint32_t pos)

This function finds which event the packet of data, “data,” belongs to based on the packets timing, “timing,” using a map data structure. If the packet indicates a timing not shared by previously captured packets, a new event in event_pool_collecting is used to store the packet and future packets with the same timing. The function GetAvailableEventPoolIndex() is called to retrieve the next free event in event_pool_collecting. If there are no free events, this indicates a serious error, and an error message is printed out. Once the packet is associated with an event, it is sent to the event object’s method AddData(uint16_t * data, uint32_t position) to be added to the event’s buffer.

In Event.h/Event.cpp, the following functions are relevant:

AddData(uint16_t * data, uint32_t position)

Finally, the packet of data, “data,” is copied into the appropriate event object’s event buffer using memcpy(). If the data cannot be copied because it would overflow the event buffer, this indicates a serious error, and an error message is printed out.

The Cutting Thread

The cutting thread performs a clustering algorithm on all events in event_pool_cutting. The cluster information calculated for each event includes a total number of clusters, the number of crystals in each cluster, the total energy of each cluster, and the center of energy of each

cluster. These pieces of information can be used to determine whether an event should be cut, and thus either not written out by the storing thread, or written out to a separate file so that cut events can be analyzed to determine the effectiveness of the cut.

The algorithm is performed using Crystal and Cluster structs, defined in Crystal.h and Cluster.h. Each Crystal has the member variables cluster_in, which is the number of the cluster the crystal is in, energy, x, y, and is_above_threshold. Each Cluster has the member variables crystals[], which is an array of all the Crystals that are in the Cluster, energy, crystals_num, crystals_index, coe_x, and coe_y. EventManager.h/EventManager.cpp has arrays of Clusters and Crystals.

The clustering algorithm is a standard O(n^2) clustering algorithm, with two main parameters which effect how it works. These are the energy threshold that determines whether a crystal is part of any cluster, and the maximum distance between 2 crystals in the same cluster. Both of these are configurable in k0to_config.h. The algorithm works as follows.

Step 1: Sum all 64 samples from each of the 2716 CSI channels, after subtracting an approximate pedestal value of the channel (calculated by averaging the first 3 samples of the channel).

Step 2: Cut crystals whose sum from Step 1 do not surpass the energy threshold defined in k0to_config.h.

Step 3: Start with one of the surviving crystals, assign its cluster to cluster 0, and add the crystal to cluster 0. Then calculate the distance between the crystal and all other surviving crystals and compare it to the maximum crystal distance defined in k0to_config.h - if the distance is smaller than the maximum crystal distance, assign the new crystal to cluster 0.

Step 4: Repeat step 3 with all crystals in cluster 0 until all crystals from cluster 0 have been processed.

Step 5: Pick a new surviving crystal and assign it to cluster 1. Repeat Steps 3 and 4 with crystals from cluster 1.

Step 6: Repeat Step 5 until all surviving crystals have been processed.

Relevant Functions

The cutting thread is executed by the funcion “static void * CutEventPool(void * param)” in EventManager.h/EventManager.cpp.

The Compressing Thread

Data is compressed based on a variable bit compression scheme. The basic idea is that if the data of a channel of 16-bit words has a range within 2^n, then that data can be stored as n-bit words plus a 16-bit word to store the minimum value of the channel plus a single byte to indicate the value of n. Thus, for example, a channel of 64 16-bit words whose range is within 256 can be stored in 67 bytes instead of 128 bytes.

The actual implementation of this algorithm for event data is slightly more complicated, as words consecutive in a channel are not consecutive in the event buffer, variable bit words requires sub-byte data manipulation which works best with big endian-data (while all raw event data is little-endian), and raw energy words have a 2 bit “header”.

In order to combat these complications, the algorithm for compressing data uses pointers, swaps bytes, masks energy word header bits, and packs bits in a big-endian style appropriately.

Relevant Functions

The compressing thread is executed by the function “static void * CompressEventPool (void * param)” in EventManager.h/EventManager.cpp, which in turn calls the function “void

Compress()” in Event.h/Event.cpp. This function compresses the uncompressed 16-bit event buffer into a compressed 8-bit event buffer of the same or smaller byte size.

The Storing Thread

The storing thread uses fwrite to write out event headers, event buffers from the events in event_pool_storing, and event trailers to the current run file. If compression is being used, then the compressed event buffers are written out, otherwise, the raw event buffers are written out. The thread is also capable of writing out a small subset of events useful for monitoring events each spill.

The thread checks if each event’s buffer is full before writing the event out. If the buffer is not full, the event is not written out and an error message is displayed. The thread has the capacity to write out these “broken events” to separate files.

Relevant Functions

The storing thread is executed by the function “static void * StoreEventPool(void * param)” in EventManager.h/EventManager.cpp.

Data Format

The data format of compressed run files is described in detail in the document “K0TO Run File Format 6.pdf.” The data format of uncompressed run files is described in detail in the document “K0TO Run File Format 5.pdf.”

Performance and Limitations

With the configuration used in May 2013, each of l3_daq’s 4 threads were able to process in parallel a spill’s worth of data in approximately 1 second. One spill occurred approximately every 6 seconds, and future goals include having one spill occur approximately every 3 seconds. While the 4 threads are processing data, the mandolin nodes use 2-4 cores at 50-100% each. While the threads are not processing data, negligible amount of processing time is used. l3_daq required approximately 2 GB of memory to run, and each mandolin node has 16 GB of memory.

There are 2 main limitations currently in l3_daq. Both involve clustering and the cutting thread.

One limitation of the cutting thread is the amount of crystals that can be processed by the clustering algorithm. The clustering algorithm is O(n^2), so as the number of crystals to be processed, n, increases, the processing time of the thread increases as n^2. This is not quite a practical limitation, however, as the number of crystals to be processed is not the full 2716 crystals in the CSI array, but rather only a small subset of those which pass an energy cut. Though there has not been a detailed study on the typical number of crystals above a useful energy threshold, empirical observations have indicated that the number is well below 100. The cutting thread is capable of processing up to 500 crystals in approximately 1 second. If the number of crystals in an event which are above an energy threshold is more than 500, the clustering algorithm is not performed on the event. The energy threshold and maximum number of crystals to be processed are both configurable in k0to_config.h.

Even though the O(n^2) clustering algorithm does not seem to be a practical limitation, I have figured out and prepared a document outlining the steps necessary to perform an O(n) (max O(n^(3/2))) clustering algorithm. This document, “K0TO O(n) Clustering.txt” can be found in the L3 git repository.

The second limitation is that while clustering information can be used for an online cut, it is not kept in memory for each event to be eventually output. However, if the specific clustering

information the cutting thread calculates is necessary for further analysis, the offline program find_clusters can be used.

The reason the clustering information is not kept in memory for each event is because if the maximum number of crystals to be processed is the full 2716, approximately 70 GB of memory would be needed to store the information. However, if the maximum number of crystals to be processed is only 500 (the configuration in May 2013), only approximately 2 GB of memory would be needed to store the information, so it is possible to modify l3_daq to do this.

Conclusion

All in all, l3_daq and the Level 3 system is running incredibly smoothly, though there is room for enhancements such as added features and more concise coding. I am hesitant to, and advise anyone else not to, add or modify the l3_daq code without a proper testing environment. However, since the code is version controlled in the codeblue git repository, should anything happen to make l3_daq unusable, an earlier used and proven version of the code can always be accessed.

To-Do

The cutting thread currently only performs the clustering algorithm, but does not perform a cut. The run in May 2013 unfortunately ended a month early, giving no time to study various ways of cutting events based on the cluster data calculated. However, the May 2013 run gave plenty of time to test the performance and accuracy of the clustering algorithm, which are quite satisfactory, and explained in the section “Performance and Limitations” above.

The faster clustering algorithm is documented “K0TO O(n) Clustering.txt” could be implemented.

The configuration file “k0to_config.h” should be reconciled with the configuration file “config.h” found in the git repository at daq/k0to-l3/config.h. The reason for this is mentioned in the document “K0TO Level 3 Software.pdf.” Simply put, using one configuration file (config.h) for both online and offline use would be more simple and efficient and less prone to error and confusion.

The clustering and compression functions used in l3_daq could be extracted to be shared between online and offline programs, similar to how config.h should be used. This is also mentioned in “K0TO Level 3 Software.pdf.”