EXPERIENCE AND ANALYSIS OF A REAL TIME DATA ACQUISITION SYSTEM

by

Divya Swarnkar

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Summer 2005

EXPERIENCE AND ANALYSIS OF A REAL TIME

DATA ACQUISITION SYSTEM

by

Divya Swarnkar

Approved: _____
Martin Swany, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____
David Seckel, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____
Henry Glyde, Ph.D.
Chairperson, Department of Computer and Information Sciences

Approved: _____
Tom Apple, Ph.D.
Dean, College of Arts and Sciences

Approved: _____
Conrado M. Gempesaw II, Ph.D.
Vice Provost for Academic and International Program

ACKNOWLEDGMENTS

I would like to thank my Research advisor, Dr. Martin Swany , for all the guidance, patience, and support, throughout the thesis work.

I would like to thank my advisor Dr.David Seckel, for his tremendous support, guidance, and inspiration throughout the thesis work. I am grateful to him for keeping my spirits high all this time.

I would like to thank Dr.Simon Patton, for his help in the understanding of the Splicer.

I would like to thank my Project Manager, Mr.Abhay Agrawal and Group Leader, Mr.Paul Musto, at Cadence Design Systens for their support in completing my thesis.

I would like to thank my family and friends for all their help and support.

.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

IceCube is a one cubic kilometer international high-energy neutrino observatory at the South Pole. The IceCube experiment has two main components: InIce and IceTop. InIce located approximately 1500 meters below the surface of ice would deploy DOMs over 80 strings with 60 DOMs on each string. IceTop, located on the surface comprises an array of DOMs and would deploy 320 DOMs over 80 stations.

The data acquisition component of this experiment, known as the DAQ, is a collection of hardware and software components. There are several challenges involved in this experiment because of the real-time nature of the data being generated, large volumes of data to be handled without any losses, limited resources like power, network issues and extreme weather conditions at the site of deployment. The work done in this thesis is specifically in context of IceTop.

A part of this work focuses on improving a specific component of IceCube DAQ called the Splicer. The Splicer is a software module responsible for merging several input sources and producing a single ordered stream. It has been used extensively in the whole IceCube experiment. We have analyzed performance of the Splicer using several methodologies. Out of the several techniques experimented, the most effective was to reduce Garbage Collection and Object Cycling.

A second component that was designed and evaluated as part of this thesis was Monitoring System. The evaluation was done under different loads and various refresh rates to ensure the correctness and stability of the monitoring system.

Chapter 1

INTRODUCTION

IceCube is a one cubic kilometer international high-energy neutrino observatory built and installed in the clear deep ice at the South Pole. The IceCube experiment has two main components: InIce located around 1500 meters below the surface of the earth and IceTop, located on the surface of the earth. The main instrument for neutrino detection is the deep detector i.e. InIce. IceTop is mainly responsible for calibration of the deep detector, tagging of the high-energy particles to distinguish them from the background particles and perform cosmic ray science with InIce. The experiment comprises of several hardware and software components. A key component of this experiment is the Data Acquisition System, referred to as the DAQ. The DAQ is a collection of hardware and software modules capturing and forwarding events generated from the activities of nuclear particles. This thesis gives an overview of the system and discusses the design and implementation of some of the components.

The basic component of IceCube is an optical module which consists of a sensor that transforms light into electrical signals. A sensor is a photomultiplier tube housed in a glass pressure vessel. These sensors, referred to as Digital Optical Modules are strung on electrical cables and frozen more than 1500 meters below the surface of the ice.

As shown in Figure 1, the whole IceCube experiment has two main components – InIce and IceTop. This classification is based on the difference in the physical topology of the optical modules and the nuclear activity observed by them. As the name indicates, the InIce DOMs would be deep below the surface while the IceTop DOMs would be on the surface.

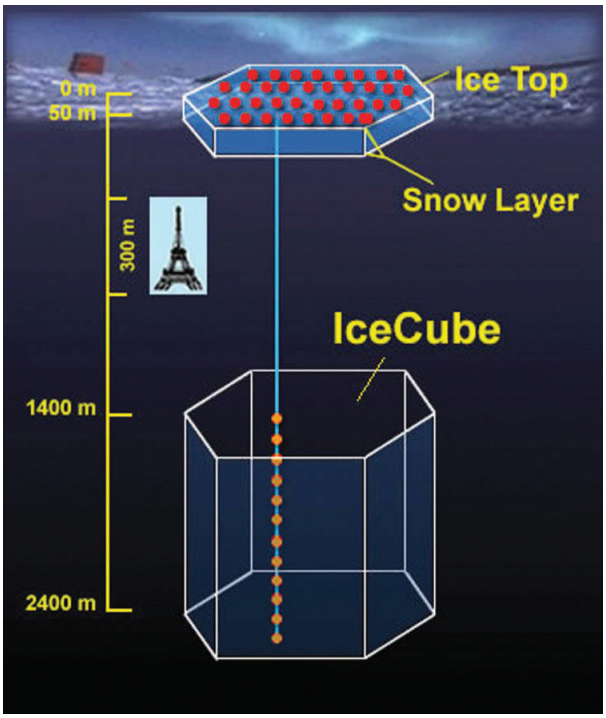


Figure 1. Physical topology of IceCube Experiment

InIce will occupy a volume of one cubic kilometer and would deploy DOMs over 80 strings with 60 DOMs on each string. IceTop, located on the surface, comprises an array of sensors to detect the activities of nuclear particles. The IceTop would deploy 320 DOMs over 80 stations. It will be used to calibrate IceCube and to conduct research on high-energy cosmic rays. The whole experiment on completion

will be operating around 5000 DOMs and would produce data in the order of Terabytes every day [2].

There is a significant overlap in the design requirements for the surface (IceTop) and deep (InIce) components of the IceCube experiment. Though similar, the surface array has specific goals including calibration, tagging and cosmic ray science, detector requirements, timing requirements and data volumes generated which are different from IceCube [2]. The main task for IceTop is air shower detection compatible with IceCube science goals. These air showers may be small air showers or large air showers or horizontal showers. The energy threshold of horizontal showers would be quite high because of low density of particles. This requires IceTop to have capabilities of handling large volumes of data [2]. The whole experiment consists of mainly the components shown in Figure 2.

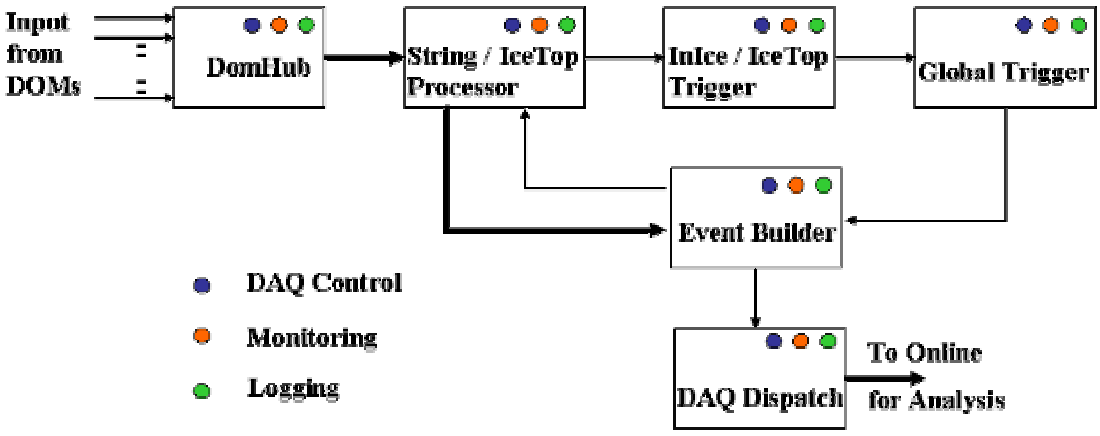


Figure 2. Software Components of IceCube Data Acquisition System

As shown, the major components of IceCube starting from the Digital Optical Module (DOM) and proceeding downstream are DomHub, String Processor, IceTop Processor, InIce Trigger, IceTop Trigger, Global Trigger, and Event Builder. The centralized control over all the components is done by DAQ Control with the help

of Logging and Monitoring modules. The DAQ components interact with each other in a well-defined manner as well as with other external utilities such as Control, Monitoring and Logging which essentially ensure that the whole system is functioning correctly in tandem and creates the ability to detect and react to unexpected scenarios. As shown in Figure 2, the whole IceCube system can be visualized as a pipeline where each component plays a very specific role and participates in a particular order in the chain. What flows in this pipeline is the event data generated by the DOMs upon capturing activities of high energy particles. These events are used to make some decisions, processed and forwarded by every module from DomHub to Event Builder [3], [4].

All the binary data generated from DOMs first go to a DomHub which provides a communication nexus for all DOMs attached to a given hub. This data contains the information required for triggers to make decision and the waveform data. The DomHub queries all the operational DOMs attached to it for any data contained in their buffers. The topology variations in the locations of DOMs demand a time correction to be performed on the data so that the timestamp information of data corresponding to an activity is synchronized throughout the data. The String Processor/IceTop Processor gets this data from the DomHub and performs time correction on it. This is the only system where all DOM data would be buffered for a period of approximately 30 secs. The String Processor extracts the necessary information including hit type, dom-id, timestamp, which are needed by triggers to make decisions. This information is then packaged into a payload and sent to the attached trigger. The triggers apply application-specific logic on this data to determine if there is an interesting event, either a physics event or a calibration event. A trigger is

identified, parameterized and forwarded to the Global Trigger .The Global Trigger is responsible for identifying detector-wide hit patterns on the trigger payloads it receives from the input detectors. Again, in case of an interesting event, the trigger is formed and sent to the Event Builder. Event Builder receives trigger requests from the Global Trigger. The request may be to read the entire experiment or a specific detector or a specific module in a detector. Event Builder decodes the request, and queries the appropriate String Processor or IceTop Processor for the waveform data. This is necessary because all the data including waveform data is buffered at String Processor or IceTop Processor and in the later stages of the pipeline it's only the necessary information that is being forwarded to the components. Hence, if there is some interesting activity detected, then the complete information corresponding to that activity is read out by Event Builder, dispatched and made available for analysis and reconstruction purposes.

Chapter 2

ANALYSIS AND EVALUTION OF SPLICER

Chapter 1 gives an overall architectural view of the IceCube Data Acquisition System. This section focuses on the deployment details of the whole experiment. Figure 3 shows the IceCube Data Acquisition system in detail.

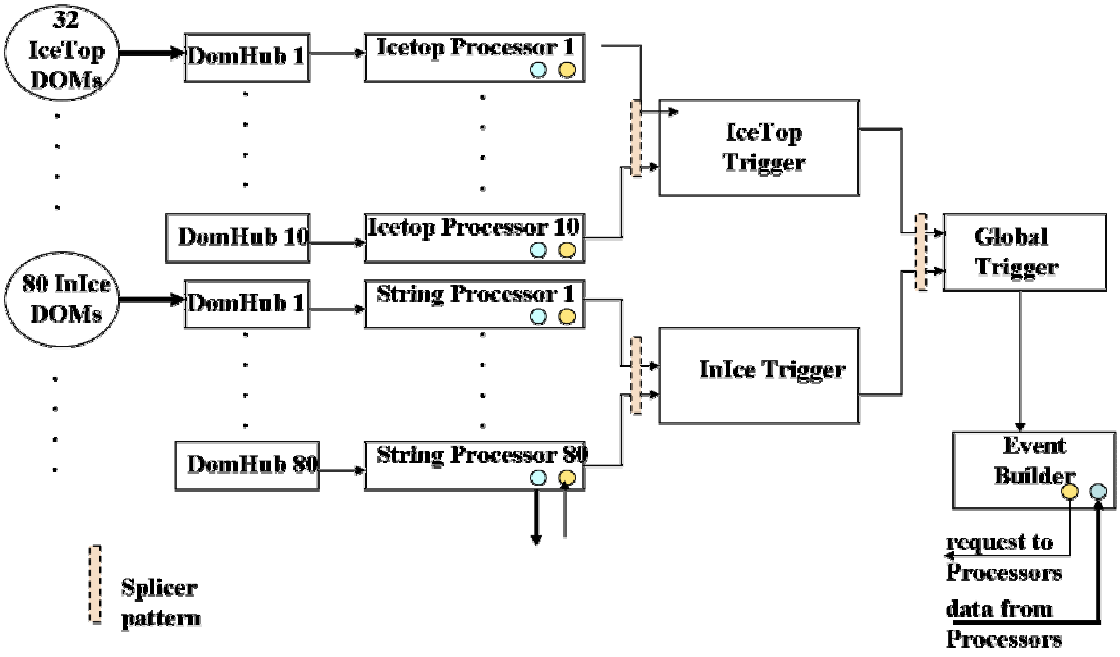


Figure 3. DAQ Software Components and their multiplicity

As discussed, DomHub is the sole electrical attachment point for all deployed DOMs and also the last hardware component in the pipeline. Further downstream, all components are software modules. Irrespective of functionality, every software component performs a common task of concentrating the data coming into it.

The output of a module upstream becomes input for the module downstream. Also, String Processor/IceTop Processor being software components have multiple instances running in parallel to expedite the data processing. These two modules need large buffers and processing power. Separating load over multiple instances creates the flexibility of running these modules over different processors and also leads to a more robust and better system in terms of handling failure. The InIce and IceTop Trigger would have multiple processors feeding data into them, and therefore they need a mechanism to multiplex the data from all input streams to a single ordered stream and then apply the processing logic to it. The global trigger needs to multiplex the data from two sub-detectors at its input i.e. InIce and IceTop Trigger.

InIce and IceTop Trigger receive hit payloads from String Processor/IceTop Processor and determine the source of origin of the event. Internally, at the microscopic level, there would at least be a Simple Majority Trigger and a Calibration Trigger running as part of InIce and IceTop Trigger. A Simple Majority Trigger is defined by sufficient number of hits within a coincidence time window. A Calibration Trigger is simply defined by the type and source of origin of the hit. The outputs of these internal triggers have to be merged internally to form a trigger from InIce/IceTop. Over the years, with new releases, there would be more internal triggers adding to the complexity of trigger and multiplexity of the merger .The experiment on completion would be taking data from approximately 5000 DOMs. As per the current design, there would be 10 instances of IceTop Processor and 80 instances of String Processor that would be in execution.

Thus, each of the modules of DAQ – String Processor, IceTop Processor, InIce Trigger, IceTop Trigger, Global Trigger all follow a similar pattern of

data – all of them receive time-ordered data from multiple sources, produce a single time ordered stream and analyze it. The above portrayed scenario led to the identification of an experiment wide pattern of data flow, referred to as the splicer pattern. Throughout DAQ, this data pattern is prevalent and hence this problem has been abstracted into a single problem. The recurring nature of this problem of multiplexing multiple asynchronous input streams and generating a single ordered output stream led to the design and implementation of the Splicer.

The Splicer has been designed and developed by Dr. Simon Patton who is one of the experienced Engineers at Lawrence Berkeley National Laboratory. The Splicer meets all the functional requirements and is a complete solution for the splicer pattern. It can be used by any module which exhibits the splicer pattern.

Motivation:

The main task for IceTop is air shower detection compatible with the IceCube science goals. The array is available for calibration and tagging along with study of cosmic ray cascades over a large and interesting range of energies. The science goal requires that IceTop operate simultaneously in two modes. In one mode IceTop needs to be self triggering, to be able to reconstruct air showers with good geometry and energy resolution independent of IceCube activity. In the other mode, IceTop acts as a veto to help eliminate a background of atmospheric muons which may contaminate the neutrino events which constitute the main science goal of IceCube. IceTop would detect small air showers (energy below threshold of 300 TeV), large air showers (energy > 300 TeV) and Horizontal showers (air showers with zenith angle greater than 60 degrees). The combined needs of a coincident trigger and large dynamic range for electromagnetic detection of air showers, a large cross-sectional area for muonic detection of horizontal showers lead to the design of stations. [2]. Figure 4 shows IceTop design with data from stations communicated to a DomHub which routes the data further to IceTop Processor.

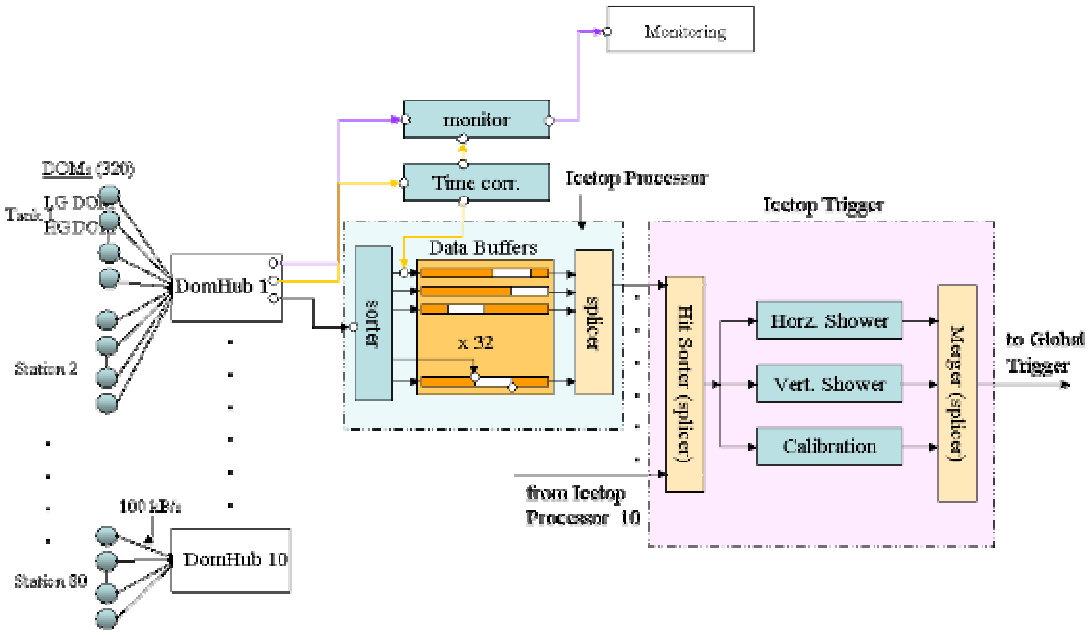


Figure 4. IceTop : Data Flow from DOM to Event Builder

Each DomHub receives data from DOMs at 3.2 Mbps. There are total 320 DOMs resulting in a data volume of approximately 2 GBps in a minute at an overall rate of 32 Mbps. DomHub, forwards data to the IceTop Processor which performs time correction on it. Assuming, it needs to hold data for 60 seconds (worst case), a total of 2 GB storage is required. Also, this brings a very strict constraint on the request coming from the Event Builder. The event builder should never request for data that is outside the time window of data in the buffer. There should never be a situation where there comes a request from Event Builder for some data which has been thrown away. Along with several other requirements, the above scenario directly demands the IceTop experiment to be highly performance oriented.

Figure 4 shows the data flow within IceTop subsystem. The data coming into the DomHub is a stream containing science data from the 32 DOMs. This stream is then demultiplexed into 32 different streams, one for each DOM. Thus, events are distributed in these streams based on the DOM from which they are

generated. At the output of DomHub, the Splicer is needed to merge the 32 streams into a single time-ordered stream and route it to the connected IceTop Processor. Each IceTop Processor after processing the input forwards the hit data to the IceTop Trigger. The IceTop Trigger at its front end has a Splicer which merges the input from 10 IceTop Processors into a single ordered stream. There is another Splicer at the end of the IceTop Trigger which acts as a merger and receives triggers from Calibration and Shower Triggers. The merger-Splicer processes the input and generates IceTop Triggers to be sent to the Global Trigger. As discussed, improving the performance of IceTop was regarded a top priority along with other goals of a correct, robust and stable system. After analyzing the whole system, the best target identified to improve the performance was the Splicer. This was for two reasons: Firstly, the Splicer had been developed keeping in mind the functional requirement of the system .Hence, there was a lot of scope of performance improvement in the Splicer. Secondly, the Splicer was one module that has been used extensively throughout the whole IceCube project. Success in improving the performance of one module would have great benefits because of the extensive use of this module throughout the software.

Reasons for Choosing Java:

IceCube is a complex distributed application, developed by programmers separated by large distances. The experiment will be completely developed over a span of 10 years. The system thus demands development in a language which is simple yet effective to communicate, robust, portable and has networking and distributed software development capabilities. This is a data-centric application and demands an application environment that best supports data handling over multiple asynchronous streams. Keeping in mind the long time frame of the project, the development environment should be such that it's always easy to find programmers working and learning it. Thus, using Java for this project was a collaborative decision and was motivated by the strong points of Java [7, 8] that are as follows

1.) Networking features :

The Java Platform has been designed to be network centric. Java makes it unbelievably easy to work with resources across a network and to create network based applications using client/server or multitier architectures. The Java API includes multilevel support for network communications. Low level sockets can be established between agents, and data communication protocols can be layered on top of the socket connection. The `java.io` and `java.nio` package contains several stream classes intended for filtering and preprocessing input and output data streams. APIs built on top of the basic networking support in Java provide higher level networking capabilities, such as distributed objects, remote connections to database servers, directory services, etc. The combination of the virtual machine (VM), portable secure

bytecodes, cross-platform capabilities and developer friendly language semantics make the Java environment powerful and productive for distributed programming.

2.) Portable

The principal advantage of Java is that it runs almost anywhere. The developer/user need only have a compatible Java virtual machine (VM), something that most operating systems and browsers now include as a standard feature. Java runs on most major hardware and software platforms, including Windows 95 and NT, the Macintosh, and several varieties of UNIX.

3.) Java is simple

There are many programmers who can understand and write code in Java, so that many people can participate in developing open source software. Java is an elegant language combined with a powerful and well designed set of APIs. The simplicity of this language enables in writing better code with fewer bugs, and thus reducing development time.

4.) Multithreading support

The ability to generate multithreaded agents is a fundamental feature of Java. Any class can extend the java.lang.Thread class by providing its own implementation of a run () method. Or by implementing the Runnable interface (which essentially means providing a run () method that represents the body of work to be done in the thread) can be wrapped with a thread by simply creating a new Thread with the Runnable object as the argument. Java also provides mechanisms for control and manipulation of its threads. Threads are assigned priorities that are publicly poll able and settable, giving the ability to suggest how processing time is allocated to threads by the Virtual Machine. Threads can also be made to yield to other threads, to sleep for

some period of time, to suspend indefinitely, or to go away altogether. These kinds of operations become important in asynchronous systems like IceCube, in which a thread is tasked with client polling and spawns new threads to service client requests.

5.) Java is not slow

Java is considered to be slow as compared to other Object Oriented Programming languages like C++. Though slower, Java is not too slow. When used correctly, Java code can be sometimes as fast as C++. There are changes proposed to the Java Virtual Machine specification that will allow several hardware features to be brought into play, thereby increasing performance further. There are many well understood optimization techniques and also code profilers which will help us to eliminate bottlenecks. Multithreading should also allow performance enhancements.

Design and Implementation of Splicer:

The Splicer is a Java module consisting of a set of Interfaces and Implementation, for producing an output stream for data exhibiting splicer pattern. It's a highly multithreaded and thread safe system fed by several asynchronous selectable channels, their data being stored into large Byte Buffers, and processed to produce a single stream of ordered objects.

The Splicer can take data from any number of input streams. This has been made possible by using the built-in support of threads provided by Java. The Splicer uses Java's Thread class that supports a rich collection of methods to start a thread, run a thread, stop a thread and check on a thread's status. In order to prevent shared data from being corrupted by multiple input threads, Java's synchronization feature is used to create mutually exclusive access to data objects. Synchronization is used to ensure that only one thread is in a critical region at once and is not interrupted in critical regions.

The Splicer is a generic solution for any data that follows the splicer pattern. This has been achieved by putting together several interfaces. Use of Java interfaces help in defining the protocol of behavior that can be implemented by any class anywhere in the class hierarchy. They are useful for capturing similarities among unrelated classes without artificially forging a class relationship. A class that implements the interface needs to implement all the methods defined in the interface, thereby agreeing to a certain behavior. Thus, domain specific solution for the splicer behavior can be obtained by implementing interfaces in the Splicer module.

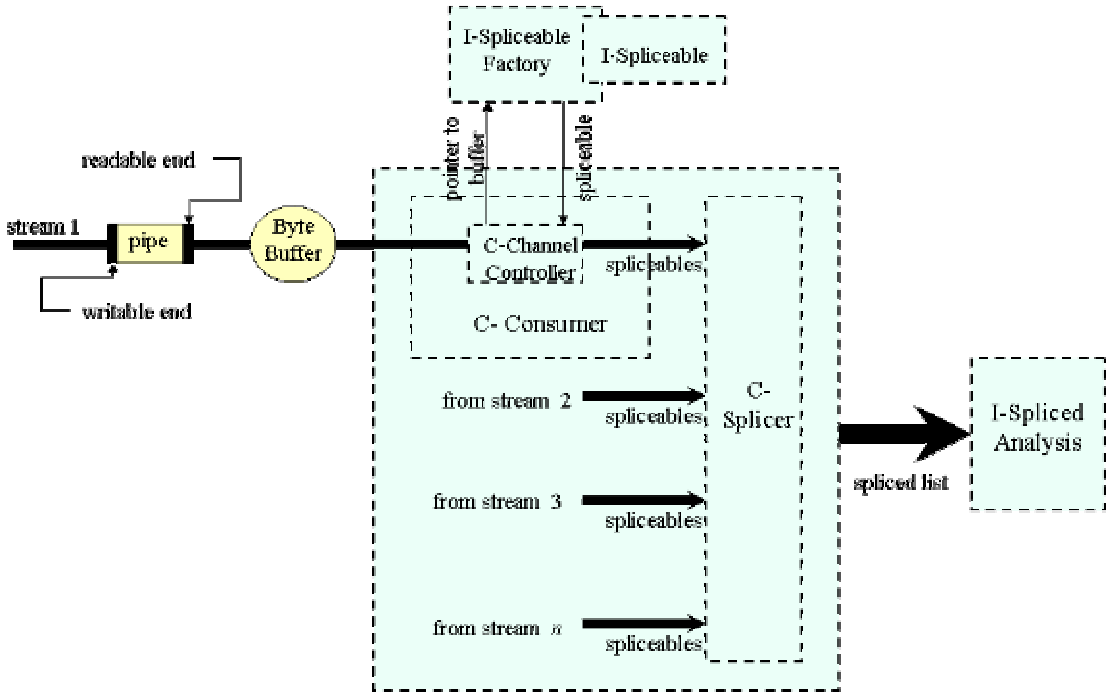


Figure 5. Components of Splicer

Figure 5 shows major interfaces and classes of the splicer module, implementation of which results in specific solutions. The I in front of class names indicate it's an Interface and C indicates it's a class.

Spliceable is an interface which extends Comparable interface. This interface marks an object as being able to be used by the Splicer .It has a single method, 'compareTo' that defines the order of elements in the output. This is a method of Comparable interface and imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.

SpliceableFactory is an interface that is used by the Splicer class to create the appropriate Spliceable instances and interrogate a Byte Buffer that contains the incoming data for a channel.

Splicer is a concrete class that uses Spliceable and SpliceableFactory instances to handle the mechanics of splicing together one or more input channels to create a single ordered output. This class accepts data from all added selectable SpliceableChannel objects and merges the resultant objects into an ordered list. This list is then passed to a SplicedAnalysis object for processing. A SpliceableChannel object is defined as an object which can be multiplexed via a selector and which implements both the SelectableChannel and ReadableByteChannel interfaces and whose byte contents can be converted into Spliceable objects by an appropriate SpliceableFactory object.

SplicedAnalysis is an interface whose implementation is called by the Splicer whenever there are new objects in the output. This is where any analysis of the ordered output takes place. This interface defines the methods that must be implemented by any analysis that wants to run on the results of a Splicer object.

Figure 5 also shows the working of the Splicer module. Apart from interfaces, the major concrete classes are Splicer, Consumer and Channel Controller. The Splicer module is started and stopped from Splicer class. The main activities that occur in this class are: registration of Readable Byte Channels from where data is to be read, creation of a ChannelController for each channel, creation of a consumer thread which would handle the processing of data from all channels, creation of the list of spliced objects from each channel.

Another important class is Channel Controller. A Channel Controller is identified by the channel it controls and the factory it uses to create Spliceables from that Channel. Each input stream has a Channel Controller associated with it. The main purpose of this class is data handling i.e. reading data from the pipe, putting it into

Byte Buffer, parsing the data from buffer and creating Spliceables using Spliceable Factory, and updating the ByteBuffer. It maintains the latest spliceables at each channel and provides it to the Splicer class.

The Consumer class is a thread created by Splicer class for managing all the selectable readable byte channels. This thread iterates over all the Channel Controllers to consume data from respective Byte Buffers, parse them, create list of Spliceables and gather the Spliceables from the list, to be added to the final list of Spliced objects

The data in Figure 5 can be explained as a unidirectional dataflow. All the data is fed to the Splicer through a pipe, the write end being registered to the data feeder and the read end is registered to the Splicer. All the binary data read from pipe is stored in a Byte Buffer. Hence there would be a Byte Buffer for each input channel. The data from Byte Buffer is read by Channel Controller and parsed to create Spliceables. To create Spliceables appropriate factory is used. The factory knows the definition of Spliceables. The Spliceables are then added to a list, which is again unique for each individual input channel. The Splicer class reads the list of Spliceables from each input channel to produce a single ordered list of Spliced objects. This list is finally sent to Analysis for analyzing and processing it.

In order to gather the Spliceables from each channel, three variables are used. Earliest Spliceable and Latest Spliceable are for each channel, while EarliestLatestSpliceable is for all the channels. Earliest Spliceable is defined as a Spliceable on a channel before which all the Spliceables have been processed. Latest Spliceable is defined as the last Spliceable on a channel. EarliestLatestSpliceable is defined as the earliest of the latest spliceables on all channels. The Splicer class uses

this value to construct the Spliced list. It takes Spliceables up to EarliestLatestSpliceable on each channel .Figure 6 diagrammatically shows it.

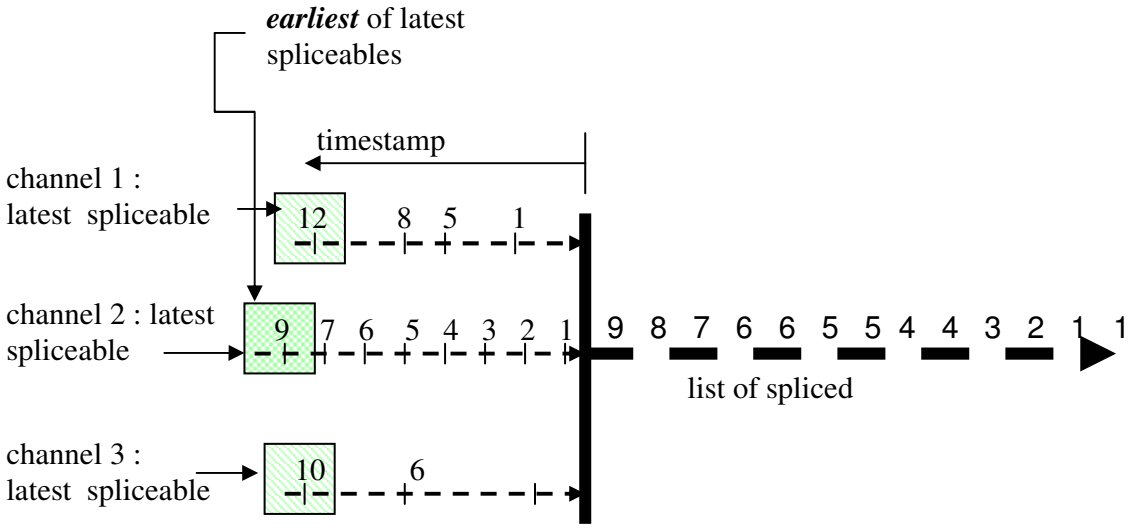


Figure 6. Working of Selectable input channels in Splicer

As shown in Figure 6, the LatestSpliceable on Channel 1 is 12, on Channel 2 is 9, and on Channel 3 is 10. The earliest of all these three is on Channel 2 i.e. 9. Hence, all the Spliceables up to this goes into the Spliced List Object. After forming the Spliced List, the Earliest Spliceable on each channel is updated to show the current status.

Experiments and Results:

As explained in earlier sections, IceCube is a complex, distributed Data Acquisition System and it is always desirable to keep such applications simple and efficient. For a real time data acquisition system, it's crucial that the data rate is well maintained to make sure that data reaches in time and in its entirety from source to destination. Therefore, at each of the components in the pipeline it's important to have enough buffer space so that there is no data loss, and each component should be fast enough so that the buffers never overflow. Hence, the rate at which data flows in the pipeline is important and every component has to be fast enough to maintain required speed of data flow. To improve the performance of this system, broadly we could follow two approaches – either improve the hardware or the software. Both approaches have different feasibility scopes and associated advantages/disadvantages.

1.) Improving the hardware

Improving the hardware which includes the processor, RAM etc. for overall system seems to be the simplest way of coming over the data rate requirements. The hardware can be improved by upgrading the configuration of all the machines. If each machine operates at a much higher speed than the current specification then definitely the processing time at each node would reduce thereby reducing the overall data processing time of the whole system. Another possibility is to go for load sharing by distributing the processing over multiple machines. This would help because there would be more number of processors doing the same task, and hence the rate at which data would forward in the pipeline would increase.

These solutions are not most preferred ones because of the resource limitations at South Pole. Adding to the budget is always a concern but more importantly it's the requirement of power, which is a scarce and invaluable resource at South Pole. The power generation for the whole experiment is done using fuel which is brought on planes and limited number of flights can be scheduled in a year. Hence, it's a project policy and requirement to optimize power usage. Increasing the multiplicity of processors directly increases the demand for electric power.

2.) Improving the software

The second and preferred approach followed in this project is to make the software code as efficient as possible to cope up with the available resources. The objective of software improvement is to identify the sources in the whole system where maximum time is consumed and try to improve them. As discussed in the previous section, one of the extensively used modules throughout the project is the Splicer and hence efforts have been made to improve its performance.

The Splicer, in terms of functionality is a very robust and generic module. It's written in such a way that although each module in DAQ is different, it uses the Splicer as an envelope and writes all the necessary code in it. Performance requirement of the Splicer were secondary at the time of development of Splicer, but has been becoming increasingly important and have been addressed in this section. Analysis of the Splicer had been an incremental step, the first step was to create an appropriate running environment for Splicer, the second step was to understand that environment and establish boundaries between Splicer and its environment, and then to improve the performance of the Splicer.

Methodology:

As already discussed, the Splicer is a module used throughout the IceCube project. As part of this thesis, it has been analyzed and evaluated in context of the IceTop sub detector. In order to analyze the performance of the Splicer, there has to be some data source simulating the behavior of input streams feeding data to the splicer. Also there has to be some data consumer consuming the data produced by the splicer i.e. the list of spliceables. Hence, in all the experiments there are three components involved in the whole setup. These are: a data producer, which provides data to the splicer over TCP/IP sockets; the Splicer, which reads data from producer, splices it, and forwards the spliced list to the data consumer; data consumer which is the user of output of the Splicer.

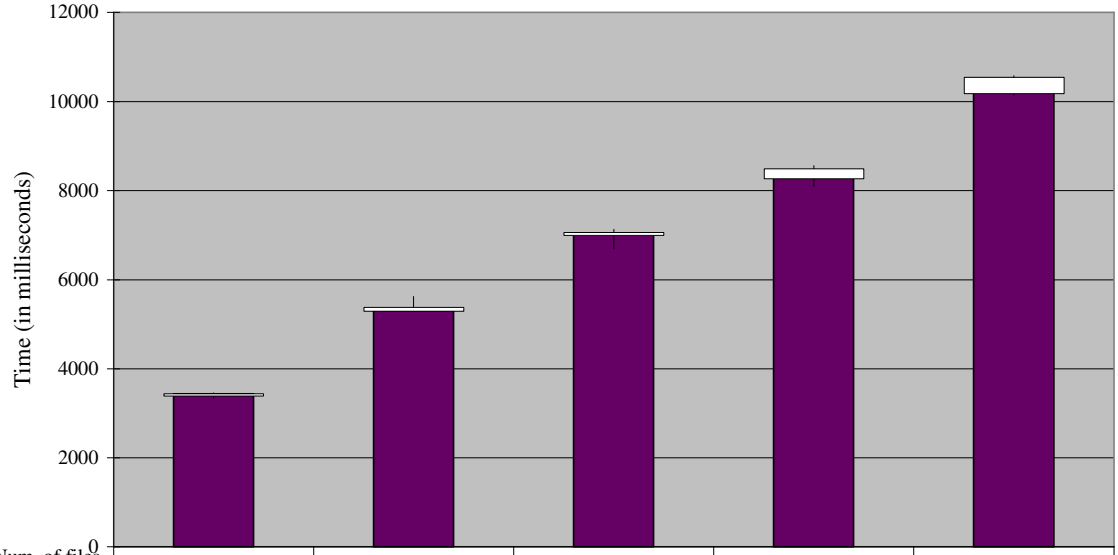
IceTop would be taking data from 320 DOMs. Hence, for simulation purposes, the data producer provides data over 320 channels. This is worth 1 second of data and is approximately equal to 8 MB. The 8 MB comes over 320 channels, with 2000 events on each channel. Each event is equal to 13 bytes, where the first 6 bytes is timestamp value of the event and the rest 7 byte correspond to the event data.

When the Splicer was developed, the environment to run Splicer didn't exist. Thus, in context of IceTop, a running environment for Splicer was created. The first two experiments, explained later in the chapter, helped to determine the test environment most appropriate for the Splicer. The next two experiments further helped in understanding the environment.

Experiments:

The results of all the experiments are expressed using box-and-whisker plots. As shown in graphs, Q1, the first quartile, is the median of the lower part of data. Q3, the third quartile, is the median of upper part of the data.

1.) The first experiment was done to establish a running environment for the Splicer. There were operating system imposed restrictions due to which approximately 250 files only could be opened at one time. The timing values obtained for different input streams are shown in Figure 7.



Num. of files	50	100	150	200	250
Median	3442	5341	7049	8409	10380
Q1	3391	5297	6994	8268	10182
Min	3342	5275	6688	8087	10138
Max	3463	5623	7130	8566	10581
Q3	3447	5388	7066	8495	10551

Figure 7. Disk Overheads at both producer and consumer

As the graph indicates, the time to process data increases almost linearly with the amount of data. The inability to open all input files in this experiment led to the

motivation for the second experiment i.e. to try reading the data from buffers rather than from files.

2.) *Disk Overhead Elimination*

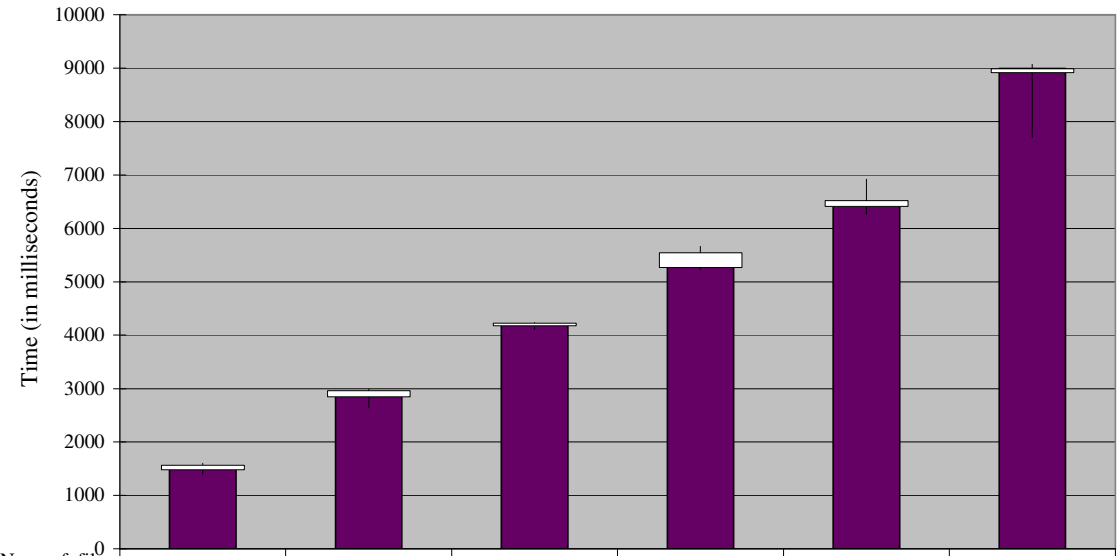
The next technique experimented was to eliminate the disk overheads involved in reading from and writing to the files.

Motivation:

I/O is a major factor in system performance. It places heavy demands on the CPU to execute device driver code and to schedule processes fairly and efficiently as they block and unblock. The resulting context switches stress the CPU and its hardware caches. I/O also exposes any inefficiency in the interrupt handling mechanisms in the kernel, I/O loads memory bus during data copy between controllers and physical memory, and again during copies between kernel buffers and application data space. Hence, an I/O driven application demands a lot of CPU time.

Experiment:

In real DAQ, the Splicer would be fed by data from a component of IceCube over a socket/pipe and would send the final ordered list to next component in the pipeline. Thus, there wouldn't be any reads or writes from files. Hence, file handling was removed to create a better environment for Splicer. To enable reading 320 input streams in the current setup, the concept of buffer was used to store the data from data-producer into a temporary buffer .The Splicer then reads data through sockets and stores them in its internal buffers. This helped in two ways. Firstly data could be read from all 320 streams. Secondly, as expected because of the disk read overhead elimination at the data-provider side, there had been reduction in the time needed by the whole setup to process the data and produce spliceables. Figure 8 shows the time consumption of current setup.



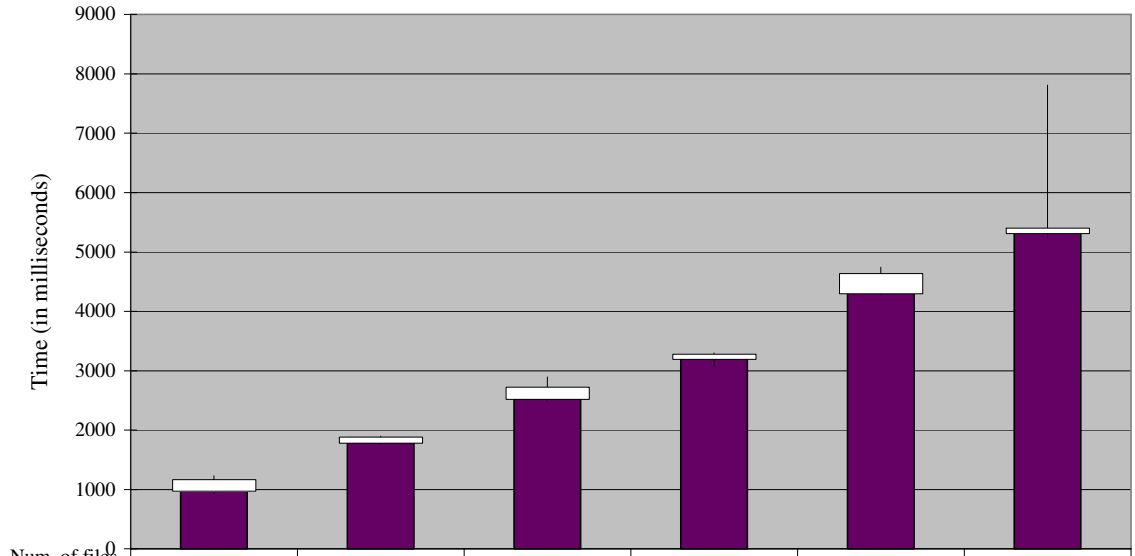
Num. of files	50	100	150	200	250	320
Median	1539	2922	4212	5329	6449	8992
Q1	1480	2850	4178	5268	6412	8913
Min	1387	2633	4101	5223	6259	7705
Max	1602	2992	4235	5663	6921	9070
Q3	1572	2967	4226	5544	6522	8995

Figure 8. No disk overheads at producer end

Conclusion:

Comparing Figure 7 and Figure 8, there was significant time improvement of approximately 50% in data handling when disk overheads were eliminated at data-producer side.

On the same principles disk write overheads were eliminated at the data-consumer end. The spliceables after being processed in the Analysis were not written to a Binary file, rather written to a buffer. Figure 9 shows the time consumption when disk overheads at the data consumer were also removed.



	50	100	150	200	250	320
Median	1143	1869	2669	3234	4404	5342
Q1	973	1780	2520	3194	4302	5311
Min	929	1772	2471	3081	4287	5229
Max	1229	1898	2896	3299	4745	7811
Q3	1174	1888	2729	3287	4643	5405

Figure 9. No disk overheads

Comparing the results of Figure 8 and Figure 9, it was observed that there was almost 30 to 40% time saved when disk overheads were removed at both data producer and consumer.

3.) *Performing Bulk Reads*

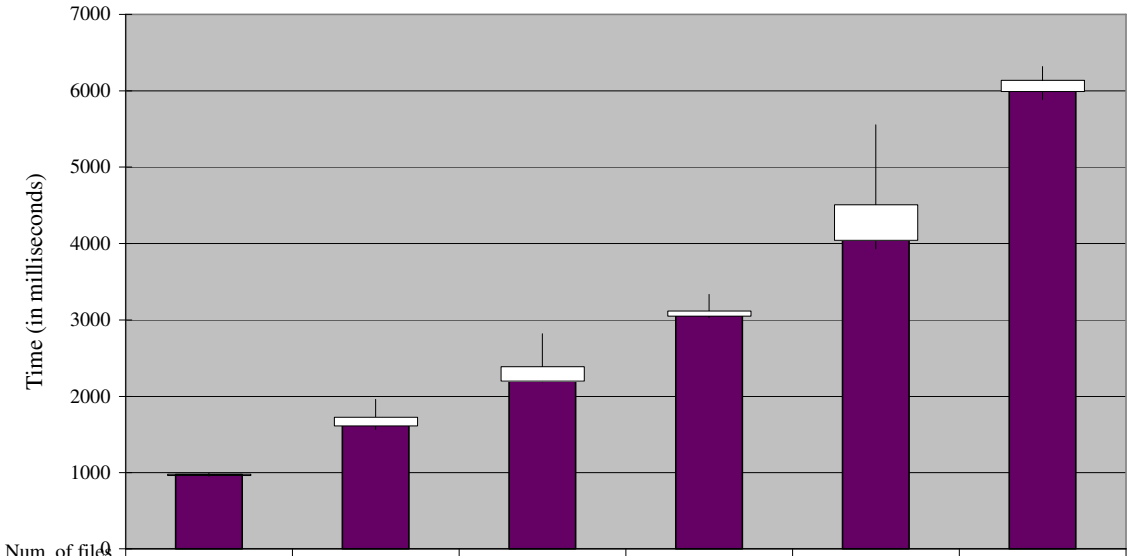
The third technique experimented to improve the performance was Bulk Read.

Motivation:

Buffering is a technique where large chunks of data is read from disk, and then accessed a byte or a character at a time. Buffering is a basic and important technique for speeding I/O, and several Java classes support buffering for the same reason.

Experiment:

The current setup of experiment has 13 bytes for each event. Out of 13 bytes, 6 bytes correspond to the time information that is being used to construct Spliceable objects. The other 7 bytes correspond to the payload information. The data producer puts all the data in the buffer and then the Splicer read the data through ports from their respective buffers .The splicer always reads 13 bytes at a time from the buffer , performs some processing to it and then sends it to the data consumer. Instead of reading each object from the socket, the entire data approximately equal to 26KB was read. Thus the number of accesses required to read the data from socket are reduced. The reason for expecting improvement was that reading data in bulk (or buffered data) is always considered to be efficient than reading small chunks of data. Figure10 shows the performance of the current setup when data is read in bulk from the sockets i.e. read accesses to socket is reduced.



	50	100	150	200	250	320
Median	973	1630	2275	3087	4208	6075
Q1	963	1611	2200	3051	4044	5995
Min.	949	1566	2187	3025	3930	5886
Max.	989	1958	2820	3335	5557	6317
Q3	983	1728	2389	3123	4509	6143

Figure 10. Reading in Bulk rather than in small packets

Conclusion:

Comparing the results of Figure 9 and Figure 10, it has been observed that there is hardly any improvement in time. It means that processing overhead involved in reading single chunk rather than bulk was not significant in the overall time consumption of the system.

4.) *Understanding Java Implementation and Processing Overheads*

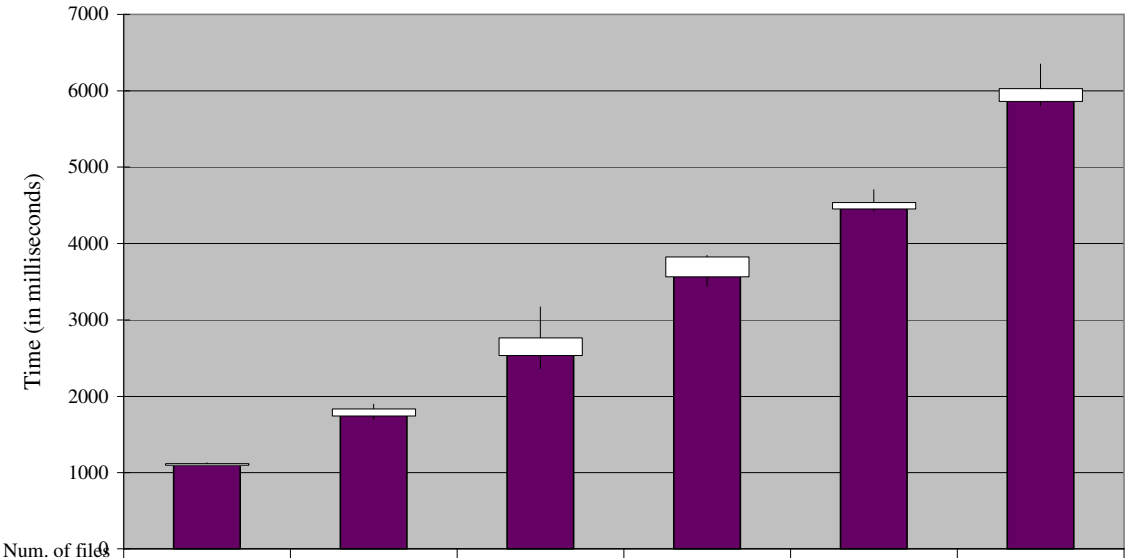
The next technique that was experimented was to understand Java implementation and processing overheads.

Motivation:

In a time critical application performing a small operation over each event may add to the overall time consumed by the whole application.

Experiment:

In the current setup, each event in total is of 13 bytes where 6 bytes is the time information. In order to create Spliceables, the Splicer needs the timestamp information associated with each event. This information has to be an 8 byte quantity (long in Java).To convert 6 bytes to 8-byte long in Java, a method is written which is called for every event. Figure11 shows the performance of the current set up.



	50	100	150	200	250	320
Median	1105	1806	2631	3701	4486	5949
Q1	1100	1745	2534	3565	4455	5862
Min.	1089	1699	2359	3440	4425	5811
Max.	1128	1897	3172	3846	4707	6351
Q3	1124	1840	2766	3828	4538	6034

Figure 11. Understanding Java implementation and processing overheads

Conclusion:

After comparing the results of Figure 9 and Figure 11, it has been observed that there is almost no improvement in the performance. This is again an indication that the

fraction of time spent in converting the 6 byte timestamp information to 8 byte java primitive data, is not dominating in the total processing. Also, to reduce the processing, instead of 6 bytes 8 bytes are supplied to the Splicer i.e. there is more data transfer now. A slight increase in time when number of files is more indicates that data transfer is more costly than small processing.

5.) *Load Balancing Technique*

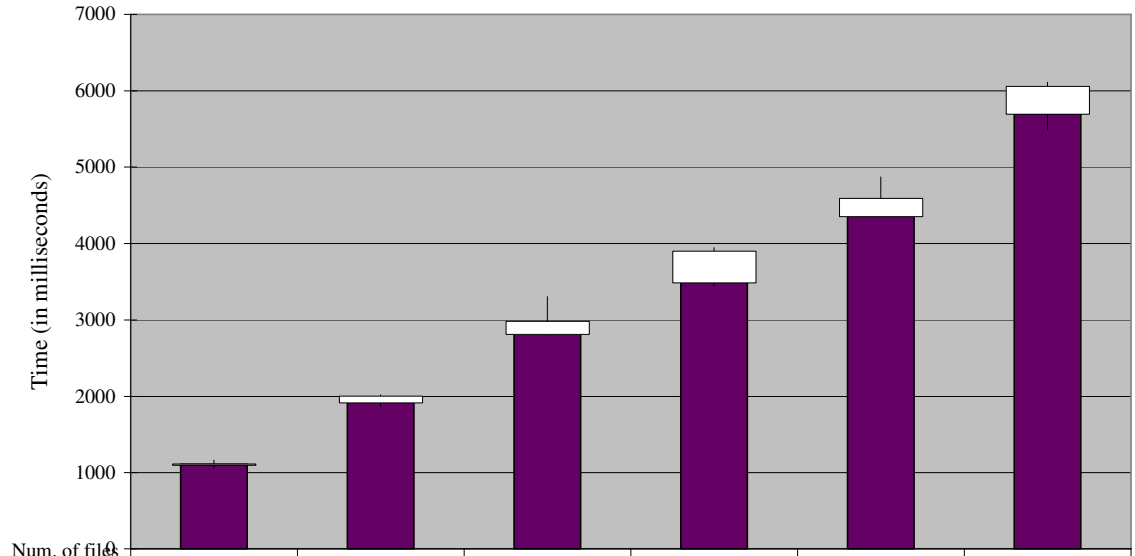
The next technique that was experimented was to separate the processing at data provider and data consumer.

Motivation:

Load balancing has been effective in solving big problems faster by distributing the workload on multiple processors working simultaneously. Complex problems can be modularized and solved faster using parallel processors.

Experiment:

In real DAQ, the Splicer would be on a different machine than the producer and consumer. By performing this experiment, we are modeling and testing the Splicer in an environment which is closer to the actual environment of the Splicer. The objective of this experiment was to separate the time consumption by data-producer from the Splicer, to see what proportion of time is used by the data producer. The two processors were connected by a cross cable. The same code was executed with data producer on one machine and the Splicer with data consumer on other machine and both of them talking to each other through sockets. Figure 12 shows the performance after removing the processing overhead.



	50	100	150	200	250	320
Median	1110	1956	2922	3582	4411	5830
Q1	1096	1914	2810	3487	4354	5694
Min.	1057	1864	2800	3439	4345	5493
Max.	1162	2021	3304	3945	4874	6114
Q3	1114	2004	2986	3906	4594	6064

Figure 12. Load Balancing by separating producer and consumer

Conclusion:

Comparing the results of Figure 9 (when all processes are on same machine) and Figure 12 (when producer is on a different machine) it has been observed that there is no significant difference in the timing when the whole processing is on one machine or on two machines. The plausible reason for this seems to be the fact that most of the processing is done at the Splicer. Also, in this case time consumed is little more than the case when all the components are on same machine which is possible because the data flows on a physical link to reach the Splicer. But, these results are helpful because it indicates that network latencies and limitations are not significantly going to affect the performance in real DAQ where data provider and the Splicer with consumer may be on different machines.

6.) *Reducing Garbage Collection and Object Cycling*

The next technique that was experimented was to reduce garbage collection and object cycling. So far the techniques were helpful to reduce the overall time of application by changing the environment in which splicer would be used. This technique would internally change the Splicer and hence, directly affects performance of the Splicer.

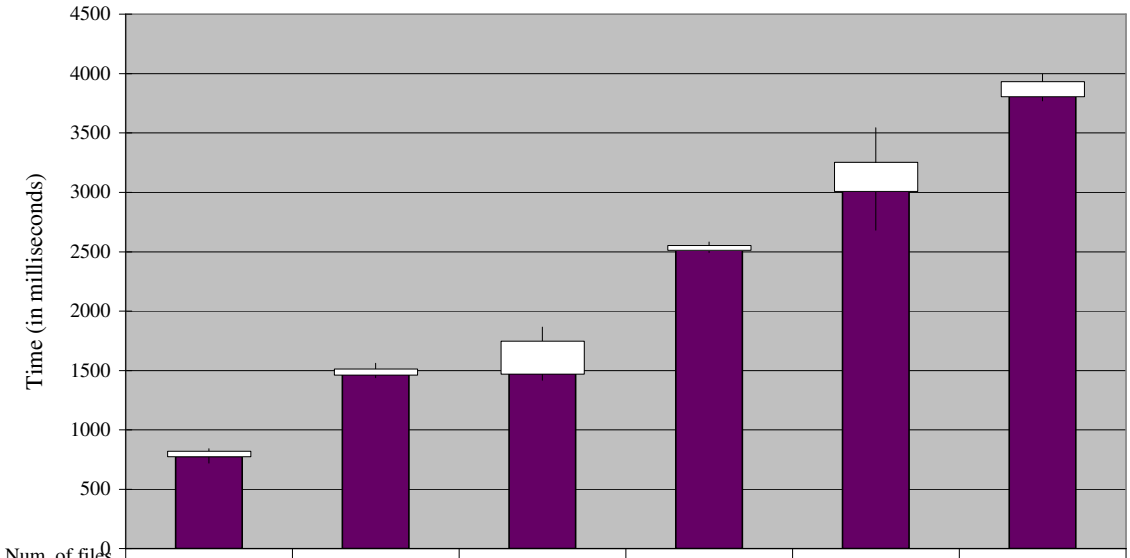
Motivation:

Object Creation costs time and uses CPU cycles. Garbage collection and memory recycling again takes time and CPU cycles. The less the object creation is, lesser is the burden of object cycling. The benefits of garbage collection are indisputable; increased reliability, decoupling of memory management from class interface design, and less developer time spent chasing memory management errors. The well known problems of dangling pointers and memory leaks simply do not occur in Java programs. However, garbage collection is not without its costs, along with it comes performance impact, pauses, configuration complexity, and nondeterministic finalization.

Experiment:

The list of spliceables generated by the Splicer and sent over to the Analysis was a collapsible vector of objects .The Spliceables were added to this list by Consumer and were removed from the list by Splicer after receiving a message from the Analysis. Hence, with Splicer to handle approximately 1000 events in a second, these elements were added and then removed from the list in 1 sec. This was time consuming. The optimization done here was to convert the list of Spliceables from a Vector to a simple array of Spliceable Objects. The size of this list was kept equal to the maximum number of elements it might process in a second. Thus, instead of creating objects every time, they need to be created once and reassigned every time with new values.

Hence, this saves object creation overheads and thus results in less work for garbage collector too. Figure 13 shows the performance with reduced garbage collection and object cycling.



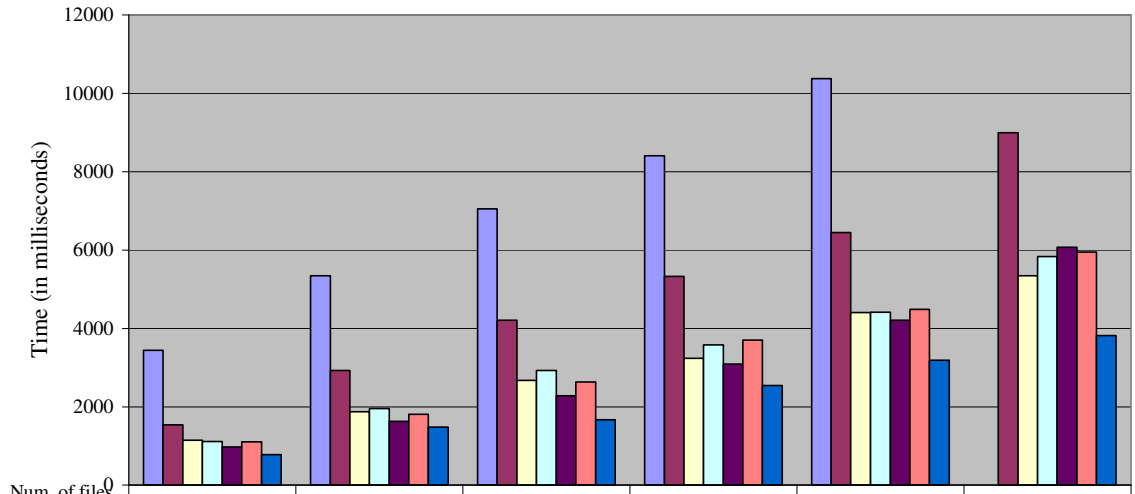
Num. of files	50	100	150	200	250	320
Median	777	1478	1666	2539	3185	3813
Q1	774	1462	1469	2514	3008	3808
Min	720	1440	1418	2494	2681	3772
Max	841	1562	1867	2584	3545	3998
Q3	822	1515	1749	2555	3256	3936

Figure 13. Reduced garbage collection and object cycling

Conclusion:

As is evident from the figures, this technique reduced the time consumption to almost 30% of the time consumed without the introduced garbage collection. Also, this technique directly improves performance of the Splicer. However, we have not been able to separate the statistics for object creation and garbage collection.

Conclusion:



Num. of files	50	100	150	200	250	320
A	3442	5341	7049	8409	10380	
B	1539	2922	4212	5329	6449	8992
C	1143	1869	2669	3234	4404	5342
D	1110	1956	2922	3582	4411	5830
E	973	1630	2275	3087	4208	6075
F	1105	1806	2631	3701	4486	5949
G	777	1478	1666	2539	3185	3813

Figure 14. Comparison of results obtained of all the techniques

A	Data fed to and read from Splicer with Disk Overheads
B	No Disk Overheads at Data Producer
C	No Disk Overheads
D	Different machine : No Disk Overheads
E	Performing Bulk Read
F	Reducing Processing Overheads
G	Reducing Garbage Collection

As shown in Figure 14, removing disk overheads and reducing garbage collection were the two most effective techniques to improve the performance of the application that would use Splicer. Improvement in time through elimination of disk overheads indicates to use performance efficient methodologies while reduction in garbage collection directly improves performance of the Splicer. The other techniques were also helpful in proceeding further because they helped in eliminating the probable targets of time consumption. Along with garbage collection, there are techniques like Object Pooling, use of ring buffers to eliminate compaction of data which seems to be promising. Object pooling is to maintain a pool of frequently used objects and grab one from the pool instead of creating a new one whenever needed. The theory is that pooling spreads out the allocation costs over many more uses. When the object creation cost is high, such as with database connections or threads, or the pooled object represents a limited and costly resource, such as with database connections, this makes sense. However, they couldn't be implemented and remain as a future work to be done.

Chapter 3

DESIGN, DEVELOPMENT AND EVALUTION OF ICETOP MONITORING SYSTEM

The IceCube experiment and its major software components have been discussed in Chapter 2. As already mentioned, the experiment will deploy 5120 optical modules and once operating, it will generate Terabytes of data daily. With such magnitude of data flowing all through the DAQ pipeline it becomes very necessary that there exists a central component to manage the whole experiment. DAQ Control is the system that performs the task of a central controlling unit in IceCube system. DAQ Control provides a single control view of all DAQ components. As such, it provides a single API through which higher software and operator levels can determine the overall state of the DAQ system and command that system to move to one of a small set of known operational states. Amongst several other functions one of the most important task of DAQ Control is to provide a single access point of control and monitoring of overall DAQ state and operation. DAQ control does this by using a centralized Monitoring System for IceCube.

The IceCube Monitoring System provides the mechanisms to centrally monitor and control all components of the IceCube system. The IceCube software shall provide a Monitoring System that allows for monitoring the performance of all facets of detector function, configuring the system, calibrating the system, identifying fault conditions, and controlling the operation of the Data Acquisition, Data Handling and

Satellite Data Transmission systems. The IceCube Monitoring System would request for monitoring information from the Components of the IceCube system. These can be String Processor, IceTop Trigger, and Event Builder etc. Thus each of the components will instrument "Monitor Points" and would report back their values to the Monitoring System.

Based on the same grounds, the IceTop Monitoring System provides monitoring information of a subcomponent of IceCube i.e. IceTop Trigger. The primary motivation behind development of IceTop Monitoring System is to respond to the central Monitoring and to make it possible to extract the details at all levels of the hierarchy. Having an ability to see into the details at each components / sub-component level not only helps in debugging and testing, but also helps to make good approximation of configurable parameters like buffer size for data at input/output streams. The graphical display of monitored data improves the Usability of the system by making it easier for the user to analyze the displayed information effectively. IceTop Monitoring System has been developed in isolation, but it has been designed such that interface of this system is adaptable.

Requirements:

The purpose of IceTop Monitoring system is to capture all the monitoring information of IceTop Trigger on a regular time basis as well as on the request of the user. The information captured and monitored by IceTop Monitoring system should be sufficient enough to give a complete idea of what's going on in the system and should be concise enough so that it does not significantly impact IceTop Trigger's performance. The main requirements of IceTop Monitoring system are governed by its uses, which are as follows:

1. The user should have control on the details of collection of data i.e. delta time of monitoring should be settable by the user. This is very important because it gives the user complete control on how frequently he wants to monitor the system. This feature is particularly useful in case of abnormal operations, where it's essential to get finer grained record of the system for some period of time to identify the point of origin/reason of problem.

2. It should give a complete picture of the IceTop Trigger. The information displayed by Monitoring about IceTop Triggers should be sufficient enough so that in case of some abnormal activities in the system, it's possible to identify the point of fault, and then scrutiny it in detail by making frequent snapshot requests.

3. Monitoring of network conditions is a crucial requirement for an application like IceCube. All through the DAQ pipeline, there is data continuously flowing through pipes/ sockets between the components. This data gets stored in buffers at the input side of all nodes. Thus, there is a need of information about rate of payloads on all

channels, buffer usage at each input channel and a histogram of latencies at each I/O stream to give a complete view of network traffic in the whole system.

4. Apart from diagnostic purposes, one of the most important uses of Monitoring system is interactive debugging. Because of the real time nature of data to be monitored, it's very difficult to ensure if the system is running correctly. The IceTop Monitoring provides detailed as well as summary of time stamps on I/O stream and their nodes. IceTop Monitoring System makes it possible to see events coming in at each input channel of a node, events coming into the node and events' going out of the node .This helps to debug problems like data loss, or programming errors like redirecting output to a wrong channel etc.

5. The Monitoring system should display information in a concise and easy to visualize manner. The better is the display, easier it is for the user to analyze the information. The Monitoring system therefore displays all information using graphical display. There may be large number of input/output channels associated with a node , showing the details of all of them can be unnecessary and confusing most of the time , hence a single input and output channel of interest can be viewed in detail at a time , while the others can be selected by just selecting a radio button. Also labeled panels are used for sub components so that they are easily available to user but not always in the front panel. Detail information about timestamps is always available but not displayed .When interested; user can view everything on a single click. Thus, all the information is made available to the user and only the most important information being displayed to him.

Design:

IceTop Monitoring System has been designed to monitor any application which can be represented as a network of nodes connected by edges i.e. a graph. The design should be flexible enough so that IceTop Monitoring doesn't restrict itself to monitoring IceTop only. This should be a generic and thus a customized solution for a problem which required graph monitoring. To accomplish this goal every component/sub-component that needs to be monitored is identified as a node in a graph. In IceCube DAQ, no node exists in isolation i.e. there are always one or more input coming to a component and one or more output going out from a component. In the current design, they are designated as Input channels to a node and output channels from a node.

Another design issue was how frequently the system should be monitored. This was of concern because of the large volume of data transfer that takes between the nodes/input channels/output channels to monitoring. The monitoring should be frequent enough so that all the interesting as well as abnormal activities are captured by the Monitoring. The monitoring cannot be very frequent since it interrupts the system from its normal function of forwarding data downstream the pipeline. Also, Monitoring should be customizable so that the user can define the time intervals when he wants to capture the snapshot of the system.

Based on the above design constraints, IceTop Monitoring system has been designed so that it provides the capabilities to the user to decide how frequently he wants to view the snapshots of the system. Thus under normal operations, this frequency can be less while if there are some problems then the user has control to

request the snapshots more frequently. Also regardless of the frequency of the user requests, the Monitoring system would request for all the Monitoring information on an hourly basis. The information received on this timed requests would be archived, and would be saved after each run.

Considering the architecture of the whole IceCube System every module can be visualized as a node that hosts at least one input channel and feeding at least one output channel. Figure 15 shows the nodes and channels architecture for IceTop Trigger.

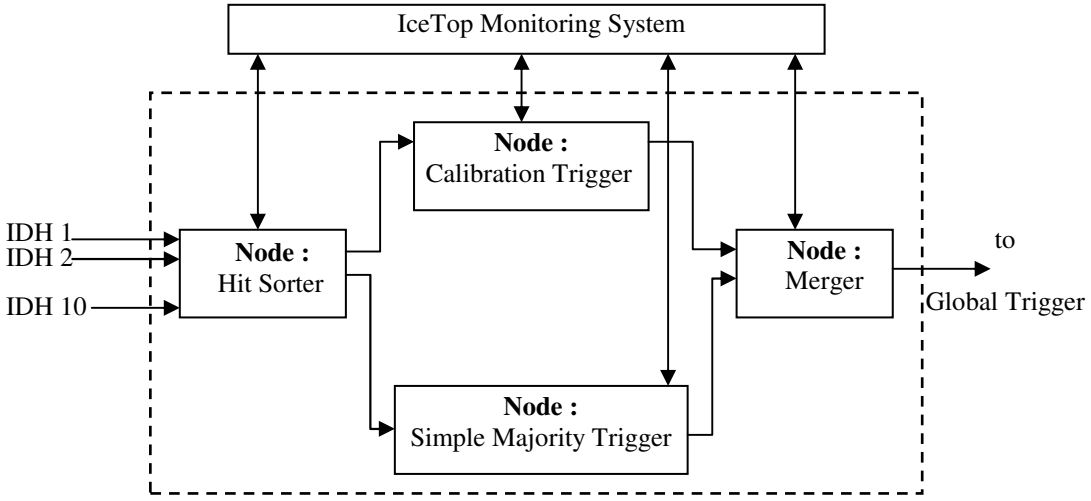


Figure 15. Nodes and their Input/Output channels in IceTop Trigger

As shown in Figure, internally there are 4 components which are part of IceTop Trigger. Each of these components can be visualized as a node, connected to at least one input and output channel. These components are also connected to Monitoring System and provide it with the real time values of monitoring points. To obtain the implementation of such a design, generic payloads for nodes and channels are defined. These payloads contain the monitoring information specific to a node/channel.

The node monitors the first timestamp and last timestamp of the all the events it has received in a given time frame. It also monitors a specified number of timestamps and stores them in a ring buffer so that the latest values are returned to the display system. The input channel monitors the first timestamp and last timestamp, the clock times corresponding to them, the data arrival rate and the buffer usage. The output channel monitors similar quantities, but at the output end of a node and for output channels.

Since each node can have varying number of input and output channels, they have been added as a vector, so that their size can be varied. Also there is another vector for low level sub-components. This is used to obtain hierarchical monitoring of the system i.e. if a sub system component has another subsystem component, then it adds the details of that sub component also.

Thus, the above schema helps to obtain a customizable information retrieval schema for a component where each component, can have varying number of input channels, output channels and sub-level components.

Implementation:

The IceTop Monitoring System has been implemented in Java and makes extensive use of Java's swing package for graphics. The application has three screens. The first screen displays the run-configuration of the IceTop trigger. It has all the static, run time settable configuration parameters of IceTop system. The second screen contains a tabbed panel of all the sub components. Each panel is identical but the information displayed is specific to a component. From second screen there is a mechanism to see the components one level down, if they exist. Also, the details of the timestamps can also be viewed from the second screen. The three main screens of the display system are shown in Figure 16

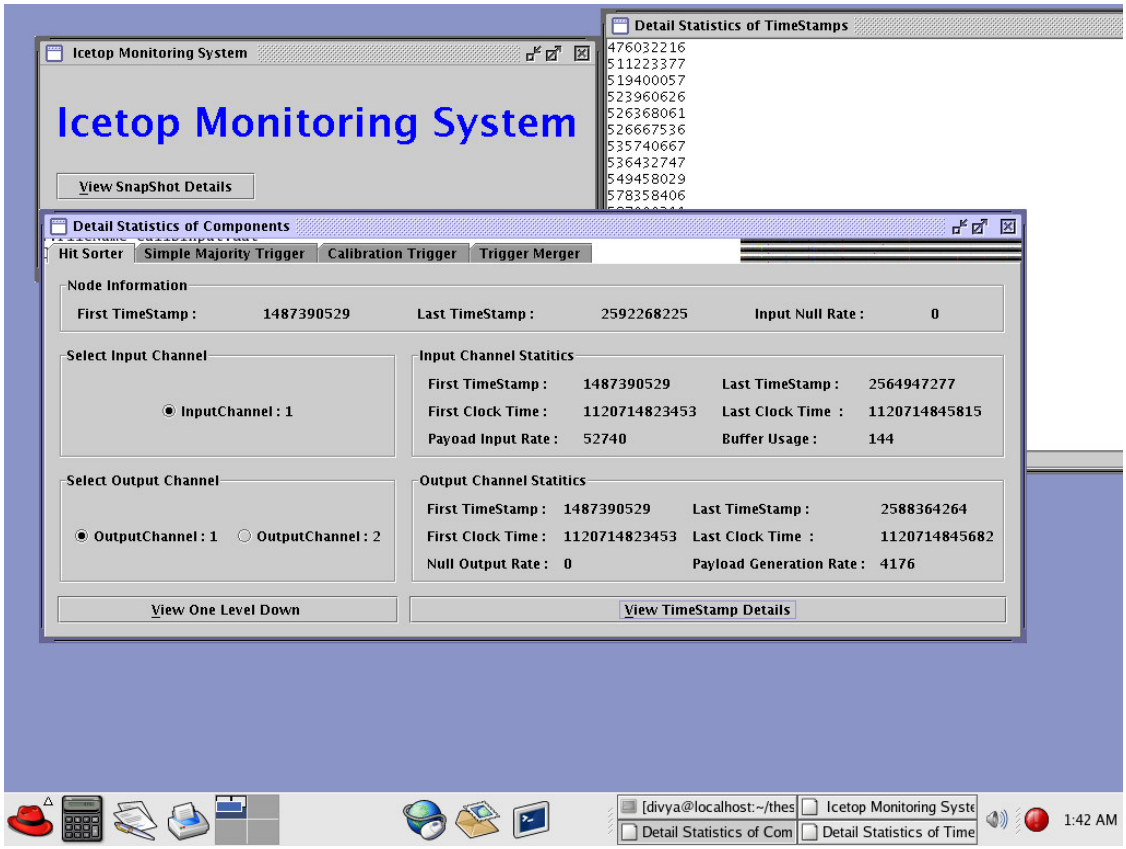


Figure 16. Output Screen snapshots of IceTop Monitoring System

As shown in the above figure, The IceTop Monitoring System displays monitoring information about the internal components of the IceTop Trigger .The screen showing detail statistics of the components in the figure shows details about the Hit Sorter (as selected in the Tabbed Pane).As per the scheme, there would be a similar screen with same data fields for IceTop Trigger, which would basically be an integrated report of all these internal triggers, but is not completely ready at this time. To get the monitoring points for IceTop trigger, it communicates to three packages – Splicer, TriggerUtil and IceTop Trigger. Figure 17 shows the communication between IceTop Monitoring System and these three packages for one node. This communication scenario would be true for any node communicating with the Monitoring System.

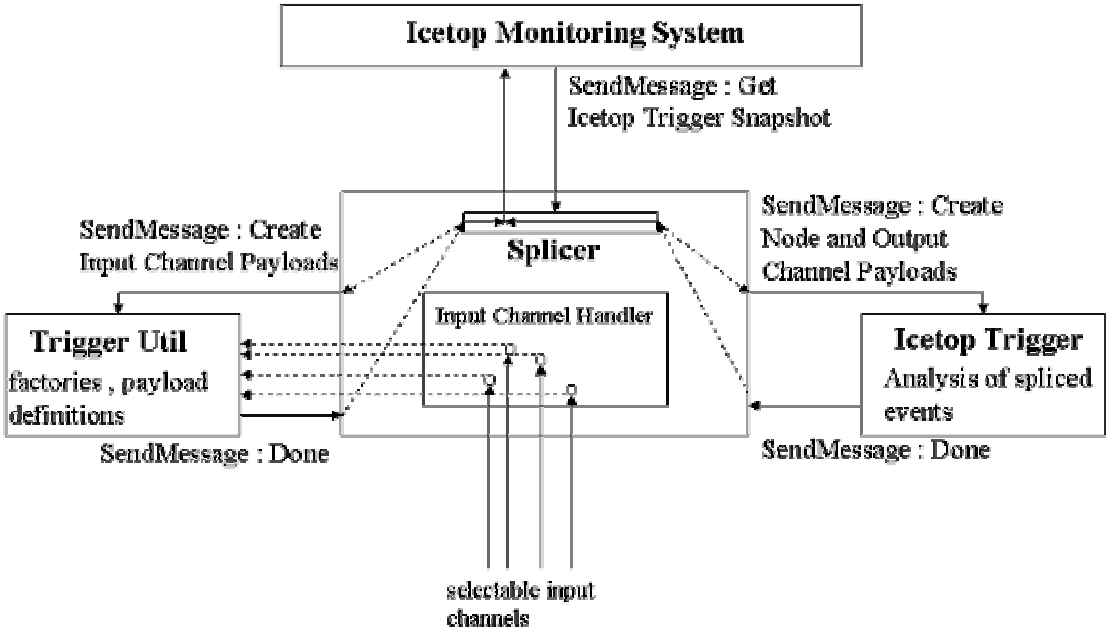


Figure 17. Process Communication in a Node

In brief the role of each module is: the Splicer acts as data controller – it reads data from buffers, gives them for processing and returns back the processed result to IceTop Trigger. Trigger Util package actually creates the spliceables from the Splicer, uses buffer-id from Splicer, reads buffers with appropriate factories and payload definitions, creates spliceable payload, and gives it back to splicer. IceTop trigger is the consumer of spliceables. It requests Splicer to create spliceable payloads. IceTop Monitoring System needs to communicate with all of them. This is because it monitors: input channels, for which the monitoring points have to be stationed at triggerUtil; output channels/nodes, for which monitoring points are at IceTop trigger. It also has to communicate with Splicer, which in this scheme is manager/controller, to signal when the monitoring needs a snapshot of the system.

Evaluation:

The memory usage and the CPU usage of IceTop Monitoring System have been evaluated while it's running and interacting with other modules like IceTop Trigger, Splicer and TriggerUtil.

The performance of IceTop Monitoring System has been evaluated under varying load conditions and refresh rates. By refresh rates, it means that the rate at which Monitoring would request for the most recent data from IceTop Trigger System i.e. most recent snapshot of the system. The events shown in the graph refer to the specified number of latest timestamp values returned by each node to the Monitoring System. The size of each of this event is 8 bytes. Apart from this, as discussed in previous section there is other information too going from nodes and input/output channels to the Monitoring System.

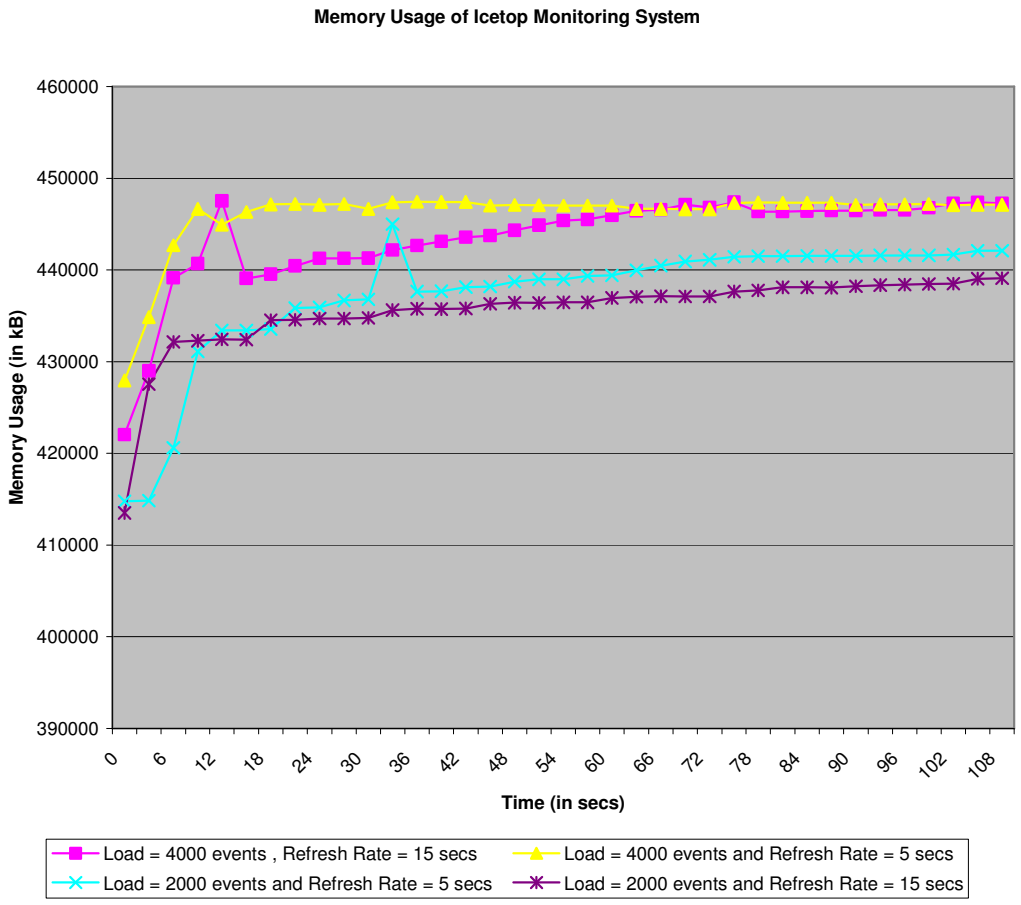


Figure 18 Memory Usage of IceTop Monitoring System

Figure 18 shows the memory usage of IceTop Monitoring System. The memory usage remains almost the same when the refresh rates are changed keeping the load same. This is clear because changing refresh rates is mainly impacting CPU usage; the memory requirements are the same because load remains the same. Supporting the above argument we do see differences in the memory requirement, when load is doubled.

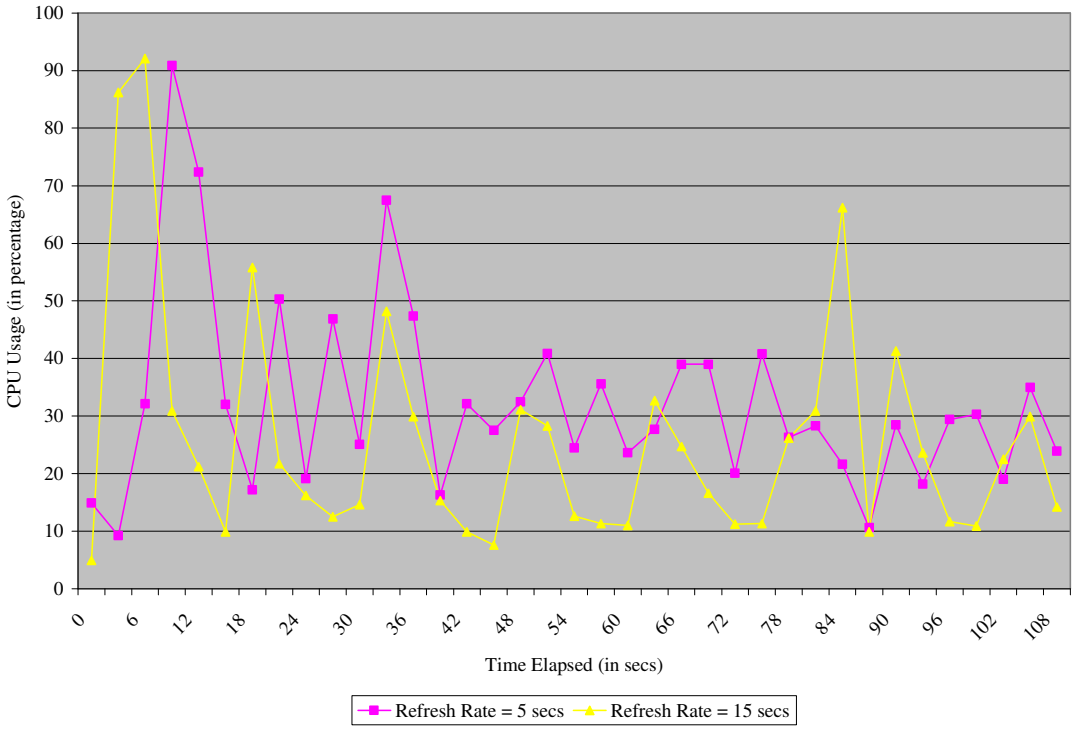


Figure 19. CPU Usage when load = 2000 events

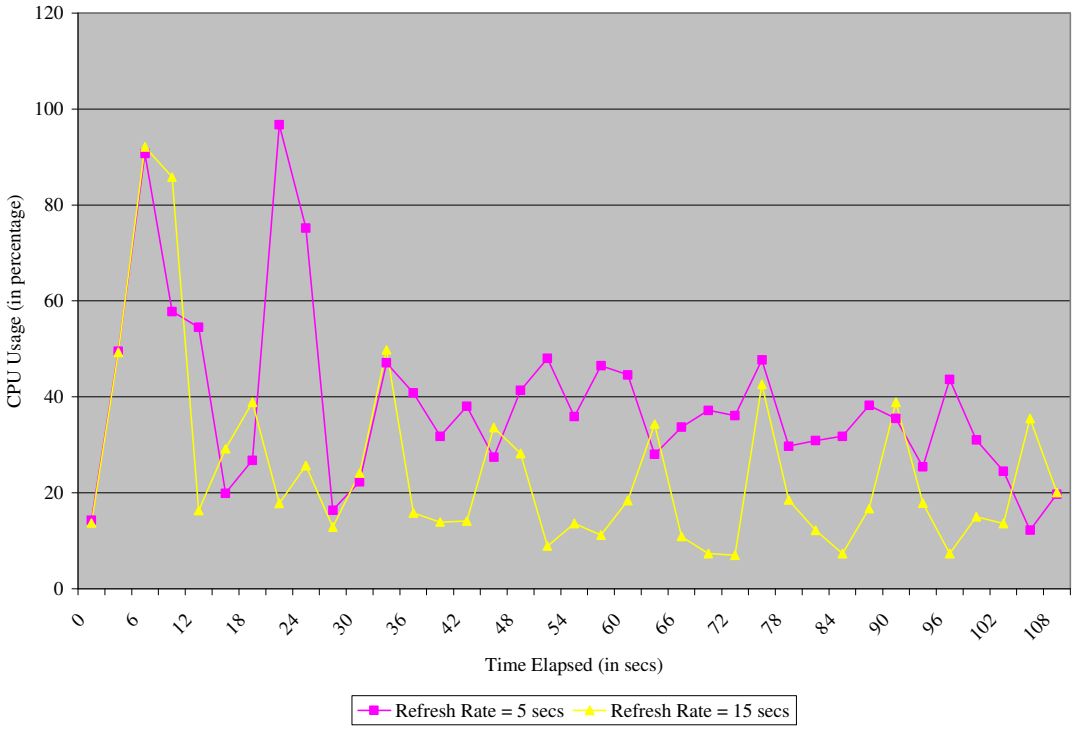


Figure 20. CPU Usage when load = 4000 events

Figure 19 and Figure 20, shows the performance of IceTop Monitoring System. In Figure 19 the load is 2000 events i.e. 16 KB while in Figure 20 the load is 32 KB. There is a similar pattern observed in both the figures i.e. as the refresh rates increases i.e. as the request to view most recent snap-shot becomes more frequent, the CPU load increases.

Conclusion and Future Work:

As part of evaluation of IceTop Monitoring System, it was executed under several conditions. The system was tested to work under high load as well as refresh rates to make sure that it survives under conditions of frequent requests. As observed, the display system runs successfully under all the tests.

There is always possibility of improvements in an application. The IceTop Monitoring system also has scope of improvement. One of the extensions would be incorporation of hierarchical scheme of components. The design has been done keeping it as a possibility, but the implementation doesn't support this feature as of yet and this remains a challenge to be accomplished further.

VOTE OF THANKS

I would like to express my acknowledgements to National Science Foundation, USA, the primary funding source of IceCube Project

I would like to thank Graduate Office, University of Delaware for giving me permission to go on Curriculum Practical Training and work on second part of my thesis work.

I would like to thank Cadence Design Systems, Inc., where I have been working as an intern since last 5 months. The experience gained at Cadence, working on a similar application as IceTop Monitoring System, was helpful in the design and implementation of this part of my thesis work.

REFERENCES

1. Version 1 , June 2002 , “IceCube : A New Window On The Universe “ <http://icecube.wisc.edu>
2. T.K.Gaisser, David Seckel, Serap Tilav, Paul Evenson, Xinhua Bai . “IceTop Preliminary Design Document “ June 30,2003
3. Chuck McParland “The Software Design Description of DAQ “ Oct. 24,2002
4. Simon Patton , LBNL “The Software Design Description of “Production” IceCube Software” Feb. 18,2003
5. Toni Coarasa , Doug Cowen , Seon Hee “Software Design Description of InIce Trigger” May 3,2004
6. C.McParland , J.Jacobson , D.Hays “Binary Representation of DOM Raw Engineering Event “ Sept. 12,2004
7. Silberschatz , Galvin , Gagne “Operating System Concepts” Sixth Edition
8. Herbert Schildt “Java The Complete Reference” J2SE 5Edition
9. Ignacio Taboada “Detector Wide Monitoring Plan for Year One WBS 1.5.3.1 “ Feb. 15, 2005
10. An Introduction to the IceCube Software Development Environment – Simon Patton , LBNL
11. John E Cavin “Software Requirement Specification for IceCube System Software “ Feb.25, 2002