ROOT – An Interactive Object Oriented Framework and its Application to NA49 Data Analysis

René Brun [a], Nenad Buncic [b], Valery Fine [f], Masaharu Goto [c], Fons Rademakers [d] Gunther Roland [b] Andres Sandoval [e]

[a] *CERN, Geneva, Switzerland*

[b] *IKF, Frankfurt, Germany*

[c] *Hewlett-Packard, Tokyo, Japan*

[d] *GSI, Darmstadt, Germany & Hewlett-Packard, Geneva, Switzerland*

[e] *GSI, Darmstadt, Germany*

[f] *GSI, Darmstadt, Germany & JINR, Dubna, Russia*

The ROOT system is a modular Object Oriented Framework for large scale data analysis that replaces the functionality of the ZEBRA, HBOOK, PAW and PIAF packages of CERNLIB. ROOT runs on Unix and Windows/NT systems. It contains a system independent hierarchical OO database, a C++ interpreter, advanced statistical analysis (multi dimensional histogramming, fitting and minimisation algorithms) and visualisation tools. Special care has been put into the design of the container classes and the I/O system for performance and system independence. A parallel file system has been developed with direct access to objects and object attributes which are a generalization of the PAW ntuples.
In the NA49 Experiment ROOT is being used for the interactive physics analysis of Pb+Pb collisions with more than 1,200 detected particles per event. For this, the parallel version of ROOT, PROOF has been installed on a PC cluster with large disks. The system gives an analysis throughput of more than 5 MBytes/s per processor allowing queries with millions of entries in seconds.
Besides NA49 the ROOT system is being used by various groups, in particular by the ALICE LHC Collaboration.

Key words: Object Oriented Framework; Data Mining; C++; PC farm; Linux-Cluster; parallel processing; LHC; Physics Analysis; Trees; ROOT; PROOF;

Preprint submitted to Elsevier Preprint *22 May 1997*

1 Introduction

The NA49 experiment[1] studies central collisions of Pb+Pb at 160 GeV/nucleon at the CERN SPS. Such collisions produce over 2000 hadrons in the final state. The detector setup of the experiment consist of four large volume tracking chambers (TPCs) of which 2 are inside superconducting magnets and of four large area time of flight hodoscopes. For a central Pb+Pb collision, approximately 1200 charged particles pass through the sensitive detector volume, producing some 1600 tracks in the detectors. These are reconstructed with better than 95% efficiency with good momentum resolution and particle identification for a large fraction of the particles. To study the physics such an extremely complicated many-body system, where possibly a plasma of quarks and gluons is formed at the early stage of the collision, correlations of all possible observables have to be made for samples of some millions of events. Existing physics analysis tools like PAW[2] and PIAF[3] using some 20 year old libraries can simply not scale to the required data mining task. In order to be able to solve this interactive analysis problem we started the ROOT project beginning of 1995.

It is natural, for the time being, to choose as a new technology for this kind of development the Object Oriented paradigm with C++ as programming language. Experience with large projects has shown that this approach indeed has many advantages in code performance and maintainability. It has also been realised that it is essential to base such developments on a framework that provides a user interface, I/O, object persistency, container classes, object versioning, evolution of the class definitions and support for many other basic functionalities that go beyond a simple Class Library.

Our aim has been to develop **ROOT** into such an **Object Oriented Framework** specially adapted to the needs and working mode of the HEP community and with it to replace the functionality of the ZEBRA, HBOOK, PAW and PIAF packages of the CERN library. Recently the first official release: ROOT 1.0 has been made and is accessible through the Web pages **http://root.cern.ch**, where the down-loadable binary, source files and test programs can be found together with the documentation, tutorials and hyperized reference manual.

The ROOT framework[4] contains among others a system independent hierarchical OO database, a C++ interpreter (CINT), advanced statistical analysis (multi dimensional histogramming, fitting and minimisation algorithms) and visualisation tools.

The user interacts with ROOT via a graphical user interface, the command line or batch scripts. The command and scripting language is C++ which is made possible by the use of the interpreter. Large scripts can be compiled and dynamically linked. Capability for dynamic loading of shared libraries is also provided. Special care has been put into the design of the container classes and the I/O system for performance, system independence and for the use of parallel direct access read and write file systems.

A parallel version of ROOT, the Parallel ROOT Facility (PROOF) has been developed in order to be able to perform data mining on very large data sets with multiple processors in real time.

The NA49 detectors have a total of 80 million sensitive elements with about 10% occupancy in a central Pb+Pb collision. During a 5 week running period more than 12 Terabytes of raw data are accumulated. With a zero suppressed event size of 10 Mbytes the data is written at 16 Mbytes/s on 100 GByte cassettes by a SONY DIR1000M high performance tape drive.

The raw data is reconstructed at SHIFT on a farm of 20 HP PA8000 processors with the DSPACK[5] analysis chain. Their processing takes 10 min/event and produces 1 Mbyte per event on the DST. One raw data tape produces 10 Gbytes of DSTs that are stored in cassettes on the CERN tape robot. A selection of 10 % of the information produces ROOT files of 1 Gbyte per raw data tape, for a total of 200 GByte of mini-DSTSs per run period. The ROOT-based mini-DSTs are interactively analysed on the Linux PC farm running the parallel ROOT.

Here we will concentrate mainly on the I/O, on benchmarks and on the parallel ROOT. The architectural overview of ROOT, as well as its capabilities have been presented in [4] and more extensively in the WWW pages.

2 A common solution to different Object I/O problems

An Object Oriented Framework has to be able to provide the exchange of objects in the following situations:

- Use of Shared Memory between different processes; for this a locking mechanism has to be provided.
- Use of Multi-Threads; locking is needed and globals and statics can not be used.
- Inter Process Communication. Messaging and Client/Server implementations are needed.
- I/O to local file; large containers and rapid access are needed.
- I/O to remote file; Web and remote servers are needed.

All these exchanges of objects have much in common like object serialization and have been implemented in ROOT by the method `Streamer`. The ROOT object I/O facility supports the transfer of arbitrarily complex polymorphic data structures from memory to a ROOT file and vice versa. The virtual method `Streamer`, is overridden in subclasses to stream an object's instance variables. Circular structures are linearized, and multiple references to the same object are restored properly. Storing pointers is implemented by an object table, which assigns an unique identifier to each transmitted object. This identifier can be transferred to other address spaces or to permanent storage.

Object I/O needs some information about the type of an object at runtime, because not only the state of an object but also its corresponding class type has to be transmitted. ROOT runtime support could provide enough information about an object's instance variables to implement the `Streamer` method generically in class `TObject`. However, we preferred the approach of a programmer selectively deciding which data members should be written to disk. Data members caching some state of an object that can easily be reconstructed in the `Streamer` method do not have to be transferred to disk. Another example is when variables can be compacted into short words or even single bits (booleans). To aid the programmer we provide a tool to generate automatically a default `Streamer` method.

The case of encountering an unknown class while reading back an object structure leads to the discussion of dynamic loading and linking. To handle this case gracefully, ROOT includes a mechanism to load a new class and link it to a running application. This dynamic linking support can be further used to extend a running system. In the case of the NA49 data analysis the library with the experiment specific classes is dynamically linked to the running interactive ROOT program.

The object I/O facility is also used as the standard format for transfering arbitrary data structures to other ROOT based applications running in other address spaces or on other machines. The transparent integration of dynamic linking into the object I/O mechanism allows the copying of instances of classes that are not known in the running application. This feature allows us to develop fairly easily advanced, web like, browsers that could operate on imported ROOT objects (for example, we could refit imported histograms without having to leave the browser).

An example of code for the `Streamer()` function is shown below for the ROOT class `TShape`. This code illustrates how the `Streamer()` function deals with base classes, including multiple inheritance. The two data members `fNumber` and `fVisibility` are integers. `fMaterial` is a pointer to the material definition for this shape. In the likely case that many different shapes will reference the same material, only one copy of the referenced material will be written. The code for this function is automatically generated by the ROOT Dictionary Generator.

```
void TShape::Streamer(TBuffer &b)
{
   // Stream a TShape object
   if (b.IsReading()) {
      Version_t v = b.ReadVersion();
      TNamed::Streamer(b);
      TAttLine::Streamer(b);
      TAttFill::Streamer(b);
      b >> fNumber >> fVisibility >> fMaterial;
```

```
   } else {
      b.WriteVersion(TShape::IsA());
      TNamed::Streamer(b);
      TAttLine::Streamer(b);
      TAttFill::Streamer(b);
      b << fNumber << fVisibility << fMaterial;
   }
}
```

3 The ROOT Trees

For many years, the data flow model in HEP has been:

Raw Data Tapes $\longrightarrow$ Data Summary Tapes $\longrightarrow$ Mini/Micro DST's

The introduction of Ntuples in the PAW framework has proven to be very successful. Many experiments are using Ntuples as a convenient replacement for mini-DSTs or even DSTs.

The PAW Ntuples, however, were restricted to very simple data objects, collection of single variables or arrays.

With ROOT, we are introducing a new concept that we call **Trees**. Trees provide the functionality of the Ntuples and much more. The Tree architecture extends the concept of the Ntuple to all complex objects or data structures found in Raw Data tapes and DST's. The idea is that the same data model, same language, same style of queries can be used for all data sets in one experiment. Trees are designed to support not only complex objects, but also a very large number of them in a large number of files.

In a conventional DST, all data structures of one event were written in a contiguous area on the file. This model has been very successful and robust for sequential files and when the analysis program requires access to a large number of attributes of one event. On the other hand, this model was particularly inefficient when one had to iterate on a subset of the events or when only a small subset of the event attributes was used.

A Tree (class `TTree`) is made of branches. Each branch (class `TBranch`) is described by its leaves (class `TLeaf`). The leaves can be simple variables, structures, arrays or objects. An array may be of variable length, the length itself being a variable in the same branch or another branch. Branches will in general be objects. However, we found it important to also support variables and structures for these applications not yet converted to C++ and objects. A structure, for example, could be a simple C structure or a list of variables in a Fortran common block.

When the fruits of one branch (detector data) are ready to be picked, they are collected into baskets (class `TBasket`). When the baskets are full they go to the store, i.e. the file.

Each branch will go to a different buffer (basket). Some buffers will be written maybe after every event, whereas other buffers may be written only after a few hundred events. The different buffers can be organized to be written to the same file or to different files. This mechanism is also well suited for parallel architectures. Note that this scheme allows also insertion of a new branch at any time in an existing file or set of files.

Due to this data clustering scheme queries can be executed very efficiently. Queries executed on one or more variables or objects, cause only the branch buffers containing these variables to be read into memory. Data are in general processed on different architectures with different memory sizes. In case the analysis is performed on a parallel architecture with a lot of memory, as many buffers as possible are kept in memory, maybe even all buffers.

The Tree data structure allows direct access to any event, to any branch and to any leaf even in the case of variable length structures.

3.1 Benchmark results

It is interesting to compare the performance of ROOT with the widely used ZEBRA RZ and ZEBRA FZ packages. For this, a standard test program `Event` has been used to generate the ROOT data file. This program generates 10,000 events. Each event has an average of 600 tracks. Each track has 17 data members. In addition each event includes an event-header object and an histogram object. The program Event was run with the option no compression and no split in order to compare with the only possibility offered by the ZEBRA Input/Output system. This program has been run in different configurations selected by a program argument to test the amount of time spent in creating and filling the event/track objects, the time to fill 20 histograms (about 120,000,000 fills). The time to write the file is computed as the total real time to run the program minus the time to create the objects (essentially the random number generator). The macro `eventa.C` is used to compute the total real time to read this file. Reading is via the interpreter. Both the ROOT file and ZEBRA files are written in a machine independent format. The ROOT library and the test program were compiled with the default level of optimisation.

A Fortran program `zEvent` has been written to perform exactly the same tasks as the ROOT C++ program. This program creates a top level ZEBRA bank `lEvent` with 3 down links: one bank to describe the event header, one bank to contain one histogram, one bank with a number of down links corresponding to the number of tracks per event. Each track bank has 17 data members. Histograms are created and filled using the HBOOK histogram package. The program includes options to compute the total time in creating and filling the ZEBRA data structures, the time to fill histograms and the total I/O time. Both the sequential I/O via FZ and direct access via RZ have been investigated.

The results of running the benchmark on a 200 MHz Dual Pentium Pro PC can be summarised as:

- ROOT is 1.6 times faster than HBOOK to fill histograms. The time to fill histograms with ROOT is independent of the number of histograms. With HBOOK, there is a log(n) dependency. By default the ROOT histogram filling functions always increment in double precision the quantities to compute correctly the mean value and standard deviations. If the similar option is activated in HBOOK, then the filling time increases from 206 seconds to 280 seconds. In reality, ROOT is at least 2 times faster than HBOOK to fill histograms.
- ROOT can write or read more than 100,000 objects per second.
- ROOT is 1.8 times faster than ZEBRA RZ to write.
- ROOT is 2.9 times faster than ZEBRA FZ to write.
- ROOT is 1.2 times faster than ZEBRA RZ to read.
- ROOT is 1.1 times faster than ZEBRA FZ to read.
- The ROOT file is 1.4 times smaller than the ZEBRA file.
- If the ROOT program is run with the file compression option for branches with integers and the split method, the size of the file is 1.8 times smaller than ZEBRA and the time spent in writing is 128 seconds compared to 74 seconds when no compression or splitting. Note that this mode is particularly interesting because it offers direct-access to any event in the file and direct-access to subsets of one event.
- If the ROOT program is run with the file compression option and the serialization method the size of the file is 271 Mbytes (2.50 times smaller than ZEBRA), but the time spent in writing is 605 seconds.

This test has been done with a very large file to avoid the system mapping the file into memory. A test with only 1000 events shows that ROOT can read 1000 events in 3.4 seconds when the file is mapped to memory. This corresponds to 13 Mbytes per real time second.

The preliminary releases of ROOT have already been used by several NA49 physicists for the analysis of the produced DSTs. For this an NA49 class library has been implemented. Event by Event fluctuation analysis with maximum likelihood fits and the construction of observables like two pion Bose Einstein interferometry have been successfully done. For this, queries on 30 GBytes of data with a total transversal volume of some Terabytes have been performed with no mayor problems. Results of these studies have been reported at several meetings[6].

The acceptance by the users and the performance of ROOT in these initial projects have been very encouraging.

4 PROOF – The Parallel ROOT Facility

Building on the experience gained from the implementation and operation of the PIAF[3] system we have developed the parallel ROOT facility, PROOF. The main problems with PIAF were due to the fact that its proper parallel operation depended on a cluster of homogeneous equally performant and equally loaded machines. Due to PIAF's simplistic partitioning of a job in N equal parts, where N is the number of processors, the overall performance was governed by the slowest node. The running of a PIAF cluster was an expensive operation since it required a cluster dedicated solely to PIAF. The cluster could not be used for other types of jobs without destroying the PIAF performance.

In the implementation of PROOF we made the slave servers the active components that ask the master server for new work whenever they are ready. In this scheme the parallel processing performance is a function of the duration of each small job, packet, and the networking bandwidth and latency. Since the bandwidth and latency of a networked cluster are fixed the main tunable parameter in this scheme is the packet size. If the packet size is choosen too small the parallelism will be destroyed by the communication overhead caused by the many packets sent over the network between the master and the slave servers. If the packet size is too large the effect of the difference in performance or each node is not evened out sufficiently.

Another very important factor is the data locality. In most cases we want to analyse a large number of data files which are distributed over the different nodes of the cluster. To group these files together we use a *chain*. A chain provides a single logical view of the many physical files. To optimize performance by preventing huge amounts of data being transfered over the network via NFS or any other means when analysing a chain, we make sure that each slave server is assigned a packet which is local to the node. Only when a slave has processed all its local data will it get packets assigned that cause remote data access. A packet is a simple data structure of two numbers: `begin event` and `number of events`. The master server generates a packet when asked by a slave server, taking into account the time it took to process the previous packet and which files in the chain are local to the slave server. The master keeps a list of all generated packets per slave, so in case a slave dies during processing, all its packets can be reprocessed by the left over slaves.

4.1 The NA49 PROOF Cluster

In NA49 we have built a PROOF cluster based on commodity PC hardware. The current configuration consists of 5 HP Vectra XU/6 PC's with dual PPro's (4 nodes have dual 200 MHz CPU's and 1 has dual 150 MHz CPU's), each node has a total of 20 GB Seagate 7200 rpm disks, 256 MB of RAM and a 100 BaseT Fast Ethernet interface. The machines are interconnected via a HP AdvanceStack Fast Ethernet switch running in full duplex mode. As operating system we use Linux (RedHat 4.0 and SMP kernel 2.0.27).

The measured memory to memory bandwidth is of the order of 10 MB/s with a latency of 200 μs and a maximum disk I/O rate of about 8 MB/s. The in total 100 GB disk pool is cross mounted between all nodes via NFS. We are working of a dedicated multithreaded ROOT file daemon that should improve remote file access considerable compared to NFS since it will have build in knowledge of the structure of the ROOT database files.

Based on a good networking infrastructure using a Fast Ethernet switch first tests show that packets of about 1 to 2 seconds give the best performance. We find that the throughput of data that can be analysed increases linearly with the number of processors. On an NA49 `.root` file of 600 MB queries with millions of entries can be made in a matter of seconds.

5 Conclusions

NA49 is the first of the new experiments (BaBar, COMPAS, STAR, Phenix, ALICE, ATLAS and the other LHC Experiments) with huge data volumes to process that can not be dealt with efficiently with the old Fortran based software. This has lead us to develop radically new infrastructures: DSPACK for DST production and ROOT as an Object Oriented Analysis Framework.

After two years of development the basic ROOT system version 1.0 has been released and its functionality replaces and extends that of ZEBRA, HBOOK, PAW and PIAF. Written in C++ and taking advantage of the OO paradigm, it gives the user a very fast and powerful basis in which to develop and run his/her applications by providing container classes, object versioning, schema evolution, persistent I/O, visualisation tools and the possibility to interact with it via a GUI and command line or batch scripts. Shared libraries can be dynamically linked. An object browser has been found to be of great use.

Tools for converting PAW/HBOOK histograms and ntuples to `.root` files and to convert the GEANT 3.21 geometry files to objects have been provided. Interfaces to several commonly used Event Generators could be implemented in a short time.

ROOT has already been used in NA49 to perform rather complicated Physics Analysis on very large data sets, and recently PROOF, the parallel version of ROOT has been successfully run on a PC farm which will allow us to do interactive Data Mining.

The prereleases of ROOT have attracted the interest of a large sector of the HEP community and beyond, with more than 2,000 copies down-loaded from our Web pages. We would like to encourage persons interested in participating in the development to get in contact with [7].

6 Further Information

This paper describes only a small part of the ROOT system. More detailed and up to date information can be found at:

`http://root.cern.ch/`

7 Acknowledgements

The authors of this article are grateful to the NA49 collaboration and the Hewlett-Packard Company for their support.

References

[1] S. V. Afanasiev et. al. NA49 Collaboration, Nucl. Phys A610 (1996) 188.

[2] PAW Users Guide, CERN Program Library Q121

[3] PIAF Users Guide, CERN Program Library

[4] R. Brun and F. Rademakers, NIM to be published, Proceedings AIHENP'96, Lausanne Sept. 2-6 1996

[5] R. Zybert and P. Buncic, DSPACK - Object Manager for High Energy Physics, Proc. CHEP95, p. 345

[6] G. Roland, NIM to be published, Proceedings AIHENP'96, Lausanne Sept. 2-6 1996

R. Stock, Proceedings ICHEP'96, Warsaw Jul. 1996 and IKF-HENPG/5-96

G. Roland, Proceedings Hirschegg'97, Hirschegg, Jan. 1997

[7] Rene.Brun@cern.ch or Fons.Rademakers@cern.ch