Computing in High Energy and Nuclear Physics, 24-28 March 2003, La Jolla, California

Twelve Ways to Build CMS Crossings from ROOT Files

D. Chamont and C. Charlot
LLR, IN2P3, CNRS, Ecole Polytechnique, France

The simulation of CMS raw data requires the random selection of one hundred and fifty pileup events from a very large set of files, to be superimposed in memory to the signal event. The use of ROOT I/O for that purpose is quite unusual: the events are not read sequentially but pseudo-randomly, they are not processed one by one in memory but by bunches, and they do not contain orthodox ROOT objects but many foreign objects and templates. In this context, we have compared the performance of ROOT containers versus the STL vectors, and the use of trees versus a direct storage of containers. The strategy with best performances is by far the one using clones within trees, but it stays hard to tune and very dependant on the exact use-case. The use of STL vectors could bring more easily similar performances in a future ROOT release.

1. Introduction

The CMS experiment [1] is one of the two multipurpose experiments being under construction to operate at the future Large Hadron Collider (LHC) at CERN. A particularly important aspect of the CMS core software is the database system that will be used to handle the petabytes of data that the experiment will produce. The experiment has recently decided to move away from its Objectivity based system in favor of an hybrid solution based on ROOT I/O [2]. This paper describes the work done in this context so to evaluate the performances of using the very specific ROOT classes (especially `TTree` and `TClonesArray`) for the data storage. In the meantime, CMS also started a more direct replacement of the existing Objectivity implementation [3].

We did not choose to explore the many use-cases of CMS, but rather focused on a single representative one and studied the many ways to implement it with ROOT I/O.

The selected use-case is the last step of the simulation chain [4]: starting from the events produced by the detector simulation, we simulate the raw data produced by the detector. Due to the high luminosity of the LHC machine, this involves firstly the superimposition to the signal event of a number of pileup events. The resulting crossing is then digitized, that is the effect of the front end electronics is simulated so to produced a digitised crossing or raw data. From the applications currently developped by CMS, this step is the most critical from the I/O point of view: for the simulation of each raw data event of $\sim 2MBytes$ size, one must load in memory about one hundred and fifty minimum bias events of $\sim 300KBytes$ size. This requires huge memory and is an intensive data reading process. Moreover, this is an unusual use of ROOT I/O on several aspects:

- The CMS code contains C++ templates, STL containers and it uses external packages whose classes cannot be instrumented for ROOT. The ROOT support for templates, standard containers and external classes is quite new and still not fully mature.
- The pileup events, stored in ROOT files, are not read sequentially. Ideally, such events should be taken from a large enough statistic, so to produce uncorrelated sequences of pileup to be added on each signal event. Since we have only a finite statistic of pileup events, we chose to select them in a random fashion so to limit the possible effect of correlation between different sequences. Nevertheless, the selection method is not fully random (real randomness would imply a change of file for each new pileup event and would have a strong impact on performances).
- The events are not processed one by one in memory, such as in standard analysis jobs as described in the ROOT documentation. As explained before, we rather have at the same time in memory all the hits from the signal and from the many minimum bias events that we want to pile up.

In this context, we have compared the performance of four kinds of containers (section 2.2), combined with three different ways (section 2.3) to store them in ROOT files. This leads to the twelve strategies advertized in the title. Also worth to be mentionned, we really focused on the huge event data to be transfered: we did not store neither meta-data nor the links or pointers between the elements of the events. Surely, the latter should be integrated in a later version of the testbed.

The section 2 gives a more detailed description of the testbed, what we have tested and on which platform. Then, the section 3 presents the most interesting results.

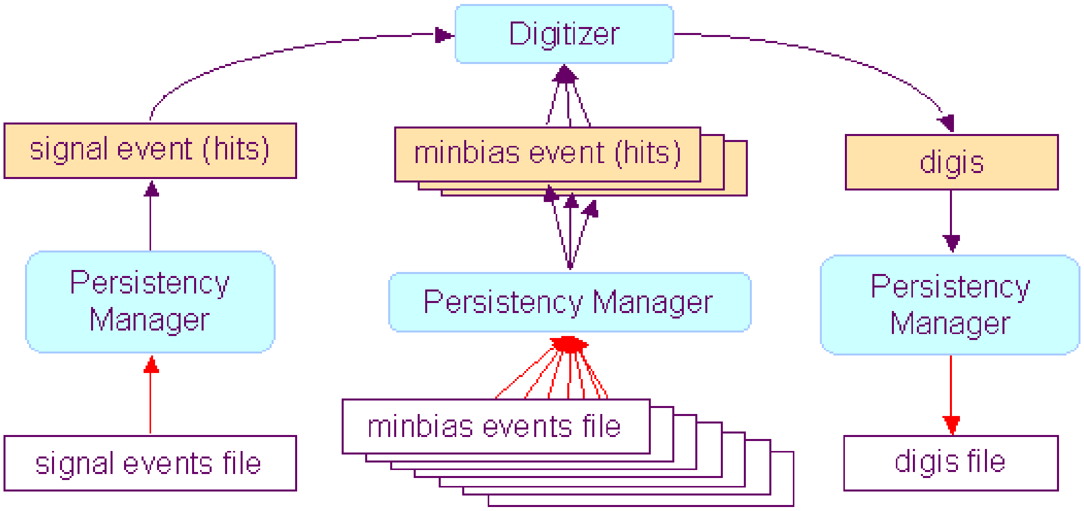

Figure 1: Main Use-Case

2. Testbed

2.1. Main Use-Case

The testbed is basically able to build crossings, as described in figure 1. As input, we have a file of five hundred signal events, and one hundred files of five hundred minimum bias events (actually, for the testbed, one hundred times the same original file). A typical job is the production of five hundred crossings. Building a crossing and digitizing it consists of the following steps:

1. A persistency manager loads in memory the next signal event from the signal file
2. Another persistent manager loads 153 minimum bias events from the minimum bias files (this number corresponds to the luminosity of $10^{34} cm^{-2} s^{-1}$). These events are selected pseudo-randomly: we take X consecutive events, then we do a random jump between 0 and Y, etc. X is called "burst" and Y is called "jump". Their typical values in CMS applications are 3 and 10. Also, when loading an event, we select randomly which rank it will have in the collection in memory.
3. The digitizer collects all the hits from the signal and minimum bias events in memory, then simulates the detector front end electronics and produces the correspondant digis.
4. A third persistent manager takes the digis and stores them as the next digitized crossing in the digis output file.

The expected bottleneck and area of interest is the loading of the minimum bias events. Below follows the mean number of objects and their size for such events:

- 351 instances of `RtbGenParticle` (whose raw size is 46 bytes).
- 584 instances of `RtbSimVertex` (whose raw size is 34 bytes).
- 169 instances of `RtbSimTrack` (whose raw size is 38 bytes).
- 3282 instances of `RtbCaloHit` (whose raw size is 20 bytes).
- 1871 instances of `RtbTrackHit` (whose raw size ist 56 bytes).

This leads to an event size of 208 Kbytes, if we just consider the pure data, without any adjunction for the support of persistency. As one can see, the size is mainly dominated by the detector simulated hits. We also computed that the mean event size would be 392 Kbytes if each numerical attribute would be of type `double` (this information will prove useful in section 2.3.2).

In the results section, we will especially study the size of the minimum bias files, and the time necessary to load 153 pseudo-random events from them.

2.2. Crossing Data Model

We chose to put all our persistent data in the shared hierarchy of folders proposed by ROOT (tree of instances of `TFolder`). This permits to decouple completely the persistency mechanism from the digitizing code.

The folder called `//root/crossing/digis` is the output "event" of the current crossing: it contains a container of instances of `RtbCaloDigis` and a container of instances of `RtbTrackDigis`.

The 153 folders which we have called `//root/crossing/minbias*`, plus the single one called `//root/crossing/signal` represent the input events composing the current crossing. Each of these folders contains a container for each kind of input event objects: `RtbTrackHit`, `RtbCaloHit`, `RtbSimTrack`, `RtbSimVertex` and `RtbGenParticle`.

Each time we run the testbed, we chose a given kind of container which is used for all the input and output data. All those containers inherit from a common abstract class:

```
template <class T>
class RtbVTArray<T>
 {
  public:
  // write interface
  virtual void clear() =0 ;
  virtual void add( const T & ) =0 ;
  // read interface
  virtual UInt_t size() const =0 ;
  virtual const T & operator[]
    ( UInt_t i ) const =0 ;
 } ;
```

This (simplified) class header shows that we rely on `size()` and `operator[]` to read the objects of a collection. On the other side, when creating a collection, we do not pre-allocate the size of the container to be filled: in the CMS code we have imported in the testbed, the size of the created collections is rarely known in advance. This write interface is not the most efficient, but it fits the user needs and what matters most for this testbed is the read efficiency. Four kinds of concrete containers have been implemented, which are described below.

2.2.1. Standard STL vector

The class `RtbStlArray<T>` wraps a standard `std::vector<T>`. The support of this kind of container is quite recent within ROOT. The main benefit of this container is that any kind of `T` can be collected (it is not needed that `T` inherits from `TObject`). The associated disadvantage is that ROOT does not apply any of its optimizations when storing/retrieving the objects, expecially its attribute-wise serialization.

2.2.2. Dynamic C Array

The class `RtbCArray<T>` contains a simple C array dynamicaly allocated. Each time the array is full, the size is multiplied by two and the array reallocated.

The objects which are collected in the C array must be instrumented with `ClassDef`, otherwise ROOT I/O will not be able to save them. Since we do not want to impose that `T` is instrumented for ROOT, we wrote a template class `RtbClassDef<T>`, which inherits from `T` and is instrumented with `ClassDef`. Each time we want to add an instance of `T` into the C array, we first change its type from `T` to `RtbClassDef<T>`. So to evaluate the eventual cost of this process, we also kept the possibility to compile the testbed with event data classes directly instrumented with `ClassDef` (this is done by unsetting the macro `RTB_FOREIGN`).

The class `RtbCArray<T>` should not be seen as a container to be used in a real application. It is rather a toy container, just written for comparison and to evaluate the performance of a good old simplistic C array.

2.2.3. `TObjArray`

The class `RtbObjArray<T>` wraps a `TObjArray`.

As for any ROOT collection, we cannot add an object to the `TObjArray` when its class does not inherit from `TObject`. Again, we do not want to impose that `T` is instrumented for ROOT, so we wrote a template class `RtbObj<T>` which inherits both from `TObject` and `T`. Each time we want to add an object into the collection, we first transform it into an instance of `RtbObj<T>`. So to evaluate the eventual cost of this process, we also kept the possibility to compile the testbed with top event data classes directly inheriting from `TObject` (this is done by setting the macro `RTB_TOBJECTS`).

As for `RtbCArray<T>`, `RtbObjArray<T>` should not be seen as a container usable in a real application. It is here for comparison, and should have the worst performances of all the containers, because it is handling the objects by pointers (and not by value).

2.2.4. TClonesArray

The class `RtbClonesArray<T>` wraps a `TClonesArray`. As for `RtbObjArray<T>`, the instances of `T` must be transformed into instances of `RtbObj<T>` before they are added to the collection.

The class `RtbClonesArray<T>` is the real alternative to `RtbStlArray<T>`. Since it is based on `TClonesArray`, which is the official optimized container of ROOT, it should show the best perfomance. The counterpart is that it is harder to use and the collected objects must be kind of `TObject`.

2.3. Persistency Managers

The task of a persistency manager is to transfer an event from memory (a `TFolder` and its contents) to disk (an entry in a `TFile`) and vice-versa. Three flavors have been implemented, which are described below.

2.3.1. RtbPomKeys

This persistency manager implements the simplest approach, which only uses the base ROOT I/O level and the class `TKeys`: one directly writes the `TFolder` to the `TFile`, each time with a different meaningful name: the name of the original folder plus an incrementing rank.

For example, in our main use-case, the output of the first digitized crossing will be saved as `digis0`, the second one as `digis1`, etc.

2.3.2. RtbPomTreeMatrix

The central idea of the second manager is to avoid the use of ROOT instrumentation and dictionaries: this is achieved by transfering the data of each container into an instance of `TMatrixD` and storing the matrix instead of the container. More precisely, each row of the matrix is the copy of one object from the original container, and each column corresponds to a given attribute of the class of the objects.

In this manager, we also chose to use a `TTree`. Each entry of the tree is an event. Each branch is dedicated to one kind of event objects, and attached to the corresponding matrix.

In this approach, we do not take profit of the ROOT persistency features, such as the generated streamers, and we must write by ourselves the code which transfers the data between the containers and the matrices (for each persistent class). Also, since any number is transformed into a `Double_t`, we expect files to be twice bigger than their normal size.

On the other hand, we do not suffer from the ROOT parser limitations and bugs. We are quite confident here in the data retrieved from the files, and this manager is primarily used in the testbed so to check that the others managers are also correctly retrieving the data.

2.3.3. RtbPomTreeDirect

This third kind of manager is the ROOT recommended approach. Each file contains a single `TTree` whose entries are the events. Similarly to the previous manager, there is one top level branch for each kind of event object, but this branch is directly attached to the corresponding container in memory.

The level of split is a parameter of the testbed, but we generally use the recommended default of 99. Concerning the size of buffers, we rather tend to reduce them to 8000 KBytes (empirical best value).

2.3.4. Main Use-Case

Once a persistent manager is built, and before writing or reading an event from a file, one must connect the manager to a given folder in memory and to a given file.

As one can see in the main use-case (see figure 1), there is a manager for the signal events, and another one for the digis, because they are always connected to the same files and folders.

With regards to the minimum bias events, we were not able to build a manager for each of the 153 events in memory (each manager has some internal buffers and this would require a large amount of memory space). Thus, we use a single manager, which must be reconnected to a new memory folder and eventually to a new file after reading each event. This connexion time is also something we have closely looked at during the analysis of the results.

2.4. Implementation issues

Provided one uses the option `-p` of `rootcint`, ROOT has very greatly improved its support of foreign classes, templates and std containers. It is now also possible to enforce the respect of ANSI C++ when compiling. However, there are still some issues with the use of ROOT that we discuss below.

2.4.1. Documentation

We really lack a central place where would be documented which subset of C++ is supported in the interpreter, which subset can be made persistent, which one can be used within a TTree, and which one can be used with a TClonesArray within a TTree. Since the ROOT team consider as a bug whatever is not supported, they try to fix any such case rather than report it. As a result, each user must rediscover by himself the unsupported cases, when they do not do invisible damage.

2.4.2. LinkDef

It has proved painful to write the configuration files for the generation of dictionaries. One must explicit all the classes which must be parsed, and in the right order. Even with only seven top classes to be made persistent, we felt the need to write a perl script for the generation of the `LinkDef` file. A key cause is that when one parses a given class, one must have parsed before all the classes of all the attributes, including each instanciated template. We wonder if one could not find a way to automatize this within rootcint.

2.4.3. Tuning of TChain branches

Since we handle a very large number of minimum bias files with the same internal tree, it was rather logical to use an instance of `TChain`. Actually, the fact that `TChain` inherits from `TTree` is misleading. In particular, if you get the branches and customize them, all your changes will disappear when the chain move internally to a new file. Thus, you must detect yourself any change of file and do again the branch customizations. In our use-case of random events and detailed branch tuning, `TChain` has finally not proved helpful.

2.4.4. TBranch attachment

When one attaches a branch to a given variable, it does not give the address of a variable, but the adress of a pointer to a variable. This will let any C++ programmer think that he can change later the pointer value, so to fill another variable. This is not true! We cannot imagine any technical reason for this, but if such an obstacle exists, the signature of the attachment method should be changed for the address of a variable.

2.4.5. TClonesArray

This class has really turned out to be hard to understand , with a least six size-like methods. We can understand that this comes from backward compatibility constraints, but still it is a problem as this class is the central piece of the persistency service. Our proposal to have a new ROOT collection class for the persistency (without gaps !) has not been supported by the ROOT team. Actually, much work is currently done for the efficient support of `std::vector<T>`, and we guess that this class could become what we would like to see.

2.5. Parameters of the Testbed

A few options can be set before compiling, thanks to macros in the central header file. They have been kept as compilation option because it was hard to make them runtime options, and not necessarily useful. Here they are:

- `RTB_FOREIGN`: if set (the default), the persistent classes are not instrumented with the macro `ClassDef`, and they will be considered as foreign classes by ROOT.

- `RTB_TOBJECTS`: if set (not the default), the top persistent classes inherit from `TObject` and are instrumented with `ClassDef` (whatever the value of `RTB_FOREIGN`). If not set, it implies the use of the `RtbObj<T>` when appending the objects to ROOT containers and the use of `RtbClassDef<T>` when putting them into a dynamic C array.

- `RTB_RESET`: if set (not the default), the empty constructors of the persistent classes set all their attributes to `0` (we were expecting an eventual impact on the compression performance).

At runtime, one must choose within four kinds of containers and three kinds of persistent managers. This leads to twelves base strategies. All the testbed results will be displayed as an array of twelve cells corresponding to these strategies (see Table I).

Table I

	Stl	**C**	**Obj**	**Clones**
Keys				
Matrix				
Tree				

In the table, the third column is just there to see how bad is `TObjArray`, the second column is just there to see the beavior of a good old dynamic C array, and the second line is mainly a way to counter-check the validity of the retrieved data. So, what matters most are the corners of the array, especially the top left corner (Keys/Stl) which is the strategy used by CMS for the direct replacement of its Objectivity implementation, and the bottom right corner (Tree/Clones) which is the solution advertized by ROOT team and evaluated in this work.

On top of the twelve strategies mentionned above, we have additional parameters, whose value can affect the performances of these strategies differently:

- Compression level: it should slow down the writing of objects, also slightly the reading, and reduce the size of files. A value of `1` is expected to be the best compromise.

- Split level: `TTree` and ROOT containers use some sort of attribute-wise storage mechanism. The split level is the depth of the decomposition.

- Size of tree buffers: amount of data which is read in one bunch from the file, for a given branch.

- Randomness: as described in the main use-case (see section 2.1), the burst and jump can be changed so to be close to a sequential use of minimum bias events, or on the contrary close to a really random access pattern.

- Size of containers: this parameter permits to reduce the size of events by a given factor, so to measure the effect of this size on the performance. The default is `1` (takes all the data). If the value is `10`, when the input files are prepared, only 1 from 10 elements is taken.

- Number of crossings: the use-case specifies that each job must build 500 crossings. This number can be lowered to see if the mean performance is the same or to shorten the execution.

2.6. Platform used for the tests

All the results which are given below have been obtained with a PC where the testbed was the only ap-

Table II Best Results

File size (Kb/event) Cpu time (s/crossing)	**Stl**	**C**	**Obj**	**Clones**
Keys	152	175	155	155
	3.16	4.82	9.65	4.43
Matrix	149	149	149	149
	2.44	2.85	3.15	2.72
Tree	153	176	156	54
	2.63	4.05	7.27	1.87

Table III Remove compression

File size (Kb/event) Cpu time (s/crossing)	**Stl**	**C**	**Obj**	**Clones**
Keys	341	568	427	384
	1.76	3.00	8.27	2.95
Matrix	400	400	400	400
	1.01	1.45	1.71	1.23
Tree	343	570	429	214
	1.53	2.70	6.17	1.16

plication running. Its characteristics and software environment are the following:

- Processor: Pentium 4, 1.8 GHz.
- Memory: about 512 Mbytes.
- Disk: IDE.
- System: RedHat Linux 7.3 .
- Compiler: gcc 3.2 .
- Root release: 3.05/03 .

3. Results

We can hardly give here all the results we have obtained when tuning the parameters of the testbed. We will rather start with the best performance we have obtained, then show and comment the effect of changing the value of some relevant parameters.

3.1. Best Results

In Table II, you will find the size of the minimum bias files (divided by the numbers of events, i.e. 500) and the mean time to read all the events of a crossing (153 minimum bias event plus a signal event). We made the 500 crossings and used all the input data. We also used the default compression level (1) which appeared to be always the best compromise.

As one can expect, `TObjArray` is always the worst choice for the read performance.

Also expected, all the strategies with matrices give files of the same size. The read time differs from one container to the other, because of the final read step where the data is taken from the matrices and transformed into new objects added to the containers. It can be seen as a measurement of the efficiency of the container `add()` method.

Let's now compare `std::vector` versus `TClonesArray`. Within a `TTree`, `TClonesArray` is by far the fastest, and the files are incredibly smaller. The key reason here is the split mechanism: each attribute of each persistent class is given its own branch and buffers (some sort of ”attribute-wise” storage). This, combined with the compression of data, proves very efficient. We do not have such performance with `std::vector<>` because ROOT does not support yet the splitting in such a case (yet one can notice a small improvement when the instances of `std::vector<>` are stored in a `TTree` rather than directly in the file). It seems that a maximum split is always worth, even when one reads back all the branches from the tree. With regards to the size of buffers, it appeared very complex to predict the best value, depending on the split level, the type of object attributes and the value of burst: it is useless to read much in advance, if there is a random event jump coming. We proceeded empirically and finished with a size of 8000 KBytes, largely smaller than the ROOT default.

The storage of `TClonesArray` directly in `TFile` (top right cell, Keys/Clones) exhibits a rather poor performance. The reason is surely because we had to switch off the `ByPassStreamer` option, apparently buggy in such a context. As a result, `std:vector<>` is the quickest alternative when not using a tree.

3.2. Remove compression

When switching off the default compression of the data (table III), one can measure how much it is efficient for the size of file! Without compression, the read time is globally smaller, without changing the classification of the corners.

Surprisingly enough, the strategy Matrix/Stl is really fast, and one can wonder what it would be if its implementation was improved. Also a surprise, but a bad one, only the Tree/Clones and the Matrix/* strategies have file sizes which match the predictions (see section 2.1). Other strategies have files largely too big, and we did not fully investigate why.

Table IV Increase randomness

File size (Kb/event) Cpu time (s/crossing)	**Stl**	**C**	**Obj**	**Clones**
Keys	341 2.20	568 3.42	427 8.63	384 3.40
Matrix	400 1.31	400 1.78	400 2.03	400 1.59
Tree	343 2.45	570 3.71	429 6.98	214 2.71

Table V Reduce data size by 10

File size (Kb/event) Cpu time (s/crossing)	**Stl**	**C**	**Obj**	**Clones**
Keys	34.9 0.82	54.2 0.96	43.6 1.51	39.5 1.05
Matrix	40.6 0.48	40.6 0.51	40.6 0.59	40.6 0.52
Tree	35.4 1.09	55.0 1.40	44.2 1.59	22.7 1.36

3.3. Then increase randomness and reduce data volume

Our next step has been to reduce the parameter `burst` to `1` and to increase `jump` to `1000`, i.e. to take the minimum bias events almost fully randomly.

In table IV, one can see that increased randomness reduces more the performance of the */Clones strategies than the */Stl, and reduces more the performance of Tree/* than Keys/*. This makes Keys/Stl clearly the best choice.

On top of that, if we reduce by a factor of ten the number of elements we have in the events (table V), the effect on the specialized ROOT classes is even worse: the strategy Tree/Clones becomes the worst strategy (apart from Tree/C and */ObjArray) !

The lesson is quite clear: in our specific CMS use-case, the use of `TTree` and `TClonesArray` is by far the most efficient strategy, but this cannot be generalized. It highly depends on the volume of data and the amount of randomness.

3.4. Other Results

Resetting the attributes to 0 in the empty constructors of the event data does not appear to help compression. So after trying it, we went back to an implementation where the empty contructors let undefined values in the attributes.

Unsetting `RTB_FOREIGN` or setting `RTB_TOBJECTS` has not greatly improved the performance of the ROOT collections, so we turned back to the use of our templated wrappers `RtbClassDef<T>` and `RtbObj<T>`.

For what concerns the write cpu time of the different strategies, the twelve strategies compares almost similarly: only the Tree/* strategies are found slightly slower.

At last, we must confess we did not systematically build the whole set of 500 crossings that is specified by the use-case. When we did, we always noticed that the overall performance was slightly better.

4. Conclusion

We have succeeded to read pseudo-random entries from a `TChain` and to dispatch them to a few hundred `TFolders` (despite the fact thar the tuning of the `TChain` branches has not been straightforward). Support for foreign classes, templates and C++ standard library has greatly improved in the recent releases of ROOT.

The magic couple TTree/TClonesArray has proved very efficient for our use-case, yet it requires top level `TObjects` and the benefits can become losses with smaller data volume or random access pattern. One can simply use STL vectors and store them directly into root files. Their integration in a `TTree` is not yet as good as a `TClonesArray`, but this could change in a future release of ROOT.

If this testbed were to be improved, one major step would be to add real pointers or `TRefs` between the objects (instead of the current indexes) and measure the impact on performance.

You can obtain the testbed source code (for linux with gmake and gcc) by contacting the authors.

Acknowledgments

We would like to thank the ROOT team who has always quickly answered to any of our question or bug report, and for the discussion we had concerning our results.

We are also grateful to Pascal Paganini for his contribution to the implementation of the CMS use-case in the testbed.

References

[1] ”CMS Technical Proposal”, CERN/LHCC 94-38, LHCC/P1, Dec 15, 1994.
[2] http://root.cern.ch/ .
[3] B. Tanenbaum, ”Implementation of Persistency in the CMS Framework”, CHEP, San Diego, 2003.
[4] ”The Spring 2002 DAQ TDR Production”, CMS NOTE-2002/034.