An Analysis of Reproducibility and Non-Determinism in HEP Software and ROOT Data

Peter Ivie, Charles Zheng, and Douglas Thain

Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA

E-mail: {pivie|czheng2|dthain}@nd.edu

Abstract. Reproducibility is an essential component of the scientific method. In order to validate the correctness or facilitate the extension of a computational result, it should be possible to re-run a published result and verify that the same results are produced. However, reproducing a computational result is surprisingly difficult: non-determinism and other factors may make it impossible to get the same result, even when running the same code on the same machine on the same day. We explore this problem in the context of HEP codes and data, showing three high level methods for dealing with non-determinism in general: 1) Domain specific methods; 2) Domain specific comparisons; and 3) Virtualization adjustments. Using a CMS workflow with output data stored in ROOT files, we use these methods to prevent, detect, and eliminate some sources of non-determinism. We observe improved determinism using pre-determined random seeds, a predictable progression of system timestamps, and fixed process identifiers. Unfortunately, sources of non-determinism continue to exist despite the combination of all three methods. Hierarchical data comparisons also allow us to appropriately ignore some non-determinism when it is unavoidable. We conclude that there is still room for improvement, and identify directions that can be taken in each method to make an experiment more reproducible.

1. Introduction

Scientific discovery in high energy physics is a collaborative effort. Confidence in and acceptance of the work of colleagues is vital in making new discoveries. One of the ways we gain confidence in research is by seeing that the results are not an accident. *"We do not take even our own observations quite seriously, or accept them as scientific observations, until we have repeated and tested them. Only by such repetitions can we convince ourselves that we are not dealing with a mere isolated coincidence, but with events which, on account of their regularity and reproducibility, are in principle intersubjectively testable." [8]* A publication is normally geared more towards communicating ideas to colleagues rather than providing precise steps to compute the results again.

Reproducible research can be shared with colleagues in a way that the research can both be repeated and extended as a building block for other scientists. *An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions. [5]* The environment and full set of instructions used by the computer make it possible to check whether multiple runs of the same software produce the

same result. This may be done to validate whether a new machine produces correct results on old software, whether new software produces correct results on an old machine, or to otherwise compare repeated or extended iterations of the research.

Unfortunately, replicating an environment on disparate computing resources is very challenging due to the wide variety of hardware and software choices. But even assuming that the environment is unchanging, many additional technical issues in computing still make it surprisingly hard to get the same result twice. Non-determinism in both codes and data[6] can arise unexpectedly from the use of concurrency, random number sources, real-time clocks, I/O operations, and other sources[4]. Differences might also be due to fundamentally different algorithms, or from accidents of the runtime environment. As a result, one cannot simply compare objects at the binary level.

We attempt to address non-determinism with the following three approaches. 1) *Domain specific methods* are sometimes available to alleviate some of the non-determinism. If not, then new methods might be implemented for this purpose. 2) *Domain specific comparisons* could be applied to results to sort through results eliminating sources of non-determinism, such as timestamps and ordering issues. 3) *Virtualization adjustments* allow control of environmental sources of non-determinism without the need for domain specific considerations.

We evaluate a typical CMS workflow used by physicists at the University of Notre Dame considering the same three ideas. 1) We tap into domain specific methods in the CMS software which expose options such as a random seed setting in the configuration files. 2) We consider an existing tool for comparisons on CMS data and introduce a new tool called ROOT_diff for comparing ROOT files in CMS but also more generally. The ROOT_diff tool takes a hierarchical approach to equivalence which helps us to isolate differences. 3) We search the CMS workflow for possible sources of non-determinism using *strace*. While running what should be an identical task multiple times, a few red flags are identified, and their possible severity is discussed. Virtualization adjustments are also employed using the Parrot tool as a virtual environment with various parameters used to eliminate some non-determinism.

After applying domain specific methods, some sources of non-determinism still existed in each step of the workflow. The domain specific comparisons helped us see levels of equivalence that were not detectable with bitwise or hash comparisons, so the problem was less about the real results being unpredictable, and more about non-deterministic elements being embedded around the real results. Using virtualization adjustments, the Time Warp feature in Parrot seemed to produce the best results, and a fixed PID feature allowed for additional improvement. Using them in tandem with the other methods, we were able to get deterministic results for the first step in the workflow as long as we set a maximum event count of 121. We continue to investigate possible causes of non-determinism when more than 121 events are requested.

All three evaluated avenues can be further pursued to make the validation of CMS workflows more successful. 1) Additional domain specific methods may be needed to separate significant results from incidental or transitory meta-data or to ensure predictable entropy. 2) Continued work on domain specific comparison tools could enable the detection of more fine grained differences, such as statistical equivalence, or to provide a framework for automating conclusions based on various equivalence metrics. 3) More options for virtualization adjustment could force the environment to ensure more deterministic behavior, such as by adjusting the algorithm that 'warps' time, or detecting other system calls that result in non-deterministic behavior. Efforts in any and all of these avenues have the potential to improve the ability for researchers to validate results and gauge reproducibility in their workflows.

2. Domain specific methods

There are often domain specific parameters or configuration options built into tools that enable more deterministic behavior. Invoking these options can be helpful in avoiding non-determinism,

```
from IOMC.RandomEngine.RandomServiceHelper import RandomNumberServiceHelper
random_seed = sys.argv[2]
...
helper = RandomNumberServiceHelper(process.RandomNumberGeneratorService)
helper.resetSeeds(random_seed)
```

Figure 1. Python code for setting all random seeds up front.

but might not solve all problems. If identifiable problems are found for which configuration options are not yet available, it might be possible to add new options or otherwise modify the tools to behave more predictably.

Understanding common sources of non-determinism (which will be discussed later in the Virtualization adjustments section) can be helpful in general. But clearly, changes to domain specific tools are easier when a specific source of non-determinism can be clearly identified. So we will start by observing the specifics of a CMS workflow used by physicists at the University of Notre Dame before moving on to more generic observations.

After describing the workflow, we describe the behavior that we observed when attempting to run the same exact task twice. In addition, we used a domain specific random seed parameter to encourage the generation of deterministic results. The two runs of a given step are compared to see whether we get bitwise identical results or not.

2.1. CMS workflow description

The following 4 steps make up a chain of tasks used to simulate possible collision events using models based on the real events observed in the Large Hadron Collider. The only difference between the workflow we used for our evaluation and the one used for real research is one of scale. We simulate relatively few events for the purposes of our evaluation, but the full complexity of the code is employed. Each of the 4 steps is described below, and the output generated from earlier steps is used as the input for later steps.

Physics Simulation (step #1 - LHE): This is a simulation of the first part of the physics involved in the collision. There is no attempt to account for the detector at this stage. The acronym LHE stands for Les Houches Event [2].

Detector Simulation (step #2 - GEN-SIM): For very technical reasons, there is a second part of simulating the physics of the collision that happens in this step. After this, the effects of the detector are simulated, but the data format read out is not the same as what the detector readout produces.

Reconstruction (step #3 - DIGI-RECO): The next step, is actually broken into two separate sub-steps that are run sequentially: The DIGI step takes the simulation file output and changes it into a format that is identical to what the detector produces. After this step, no distinction needs to be made in the software between running on simulated and real data. The RECO step is the same reconstruction that's applied to real data that takes detector signals and figures out which particles would have made those signals in the detector.

Data Reduction (step #4 - MiniAOD): This last step takes the output of the RECO step (which is in a data format known as AOD = Analysis Object Data), and simplifies it into a reduced data format that contains the information that almost everyone needs to do analysis. Some small fraction of analyses actually need the level of detail in AOD and can't use MiniAOD, but most researchers use the MiniAOD data.

2.2. CMS workflow results

In an initial attempt to get deterministic results for each step in the workflow, the method shown in Figure 1 was used to force every execution of the step to use the exact same random seed.

Each step in the workflow was then executed twice on the same machine, in the same day, with the exact same command and parameters. The output of the first execution was even moved to a separate folder so the exact same command (including the folder name) could be used for the second execution immediately. For steps 2-4, where the results from the previous step are used as input, the result from the first run of the previous step was used for both runs of the following step.

2.2.1. Physics Simulation (step #1 - LHE): Unfortunately the checksums differ for the ROOT files generated by each run as shown at the top of figure 1.

	Run	checksums	Size
	1	b2ed825f...	6,545,067
	2	c35bc9c4...	6,545,072

2.2.2. Detector Simulation (step #2 - GEN-SIM): The same is true for step2. The checksums for the two output files differ from each other.

	Run	checksums	Size
	1	a2f8138c...	22,773,369
	2	40c5791a...	22,773,362

2.2.3. Reconstruction (step #3 - DIGI-RECO): Step 3 produces 3 ROOT files, but only one of them (File #1) is used as the input for step 4. File #2 is the output from the DIGI sub-step and is used as input for the RECO sub-step. File #3 is the data quality monitoring output. None of the files appears to be fully deterministic.

File	Run	checksums	Size
#1	1	b8e7810f...	21,320,833
#1	2	f3fb4d9c...	21,320,560
#2	1	81f04953...	38,799,122
#2	2	767237d7...	38,799,120
#3	1	4669be6b...	1,953,306
#3	2	534891a4...	1,953,460

2.2.4. Data Reduction (step #4 - MiniAOD): The final step also exhibits non-deterministic behavior. For all steps in the workflow, the exact same command produces results that are bitwise different each time it is run. Without some domain specific comparison tool or manual comparison by an expert, it is impossible to know whether the results are equivalent or if some underlying change in the environment caused the two results to diverge from each other in a significant manner.

	Run	checksums	Size
	1	ab202459...	3,384,806
	2	0fa6a17a...	3,384,594

Table 1. File checksums at each stage

3. Domain specific comparisons

The results of a CMS workflow are serialized into ROOT objects and stored in a ROOT file. We use the ROOT framework [3] to process experiment datasets since comparing checksums is insufficient. As shown in Figure 2, a ROOT file is a sequence of data records with a well defined format. A file header contains information such as a file identifier, file version, the compression level, etc. The file also contains blocks of object data, each with header data. An object header contains the object length, header length, pointer to the object, etc., for the binary object data which is found immediately after it in the file.

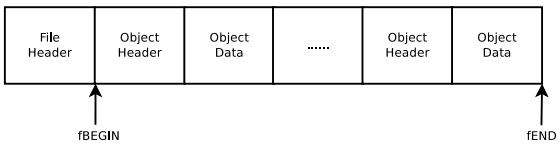

Figure 2. ROOT file Structure

We can validate successful reproduction of workflow by comparing the structure and contents of each object in the ROOT file. If a workflow is reproduced correctly, each ROOT object it produced should have a matching object from the result of the original workflow. Based on this observation, we developed *ROOT_diff*, a domain specific comparator for scientific workflows that produce ROOT files.

3.1. Comparison procedure

Simulation results of large physics workflows are often large and contain different intermediate ROOT files from multiple substages. There exists a tool developed by CMS for one-to-one EDM object comparison. This is a perfect tool for domain scientists who want to observe the difference between the events, particles and variables of two ROOT files. Since the comparison is conducted at the physics object level, a more fine-grained analysis of root files is required, which consumes time and resources. For comparing a typical RECO file, it launches 180 processes, takes 30 minutes and creates many files[1]. To simplify and speed up the validation we only compare the structure and contents of each data record.

We defined three levels of equivalence: (1) STRUCTURE-EQUAL: Two root objects have same object length, cycle number and class name. (2) CONTENT-EQUAL: Two root objects are structurally equal to each other and have the same object data content. (3) BITWISE-EQUAL: Two root objects are content equal to each other and have same timestamp. We claim that reproduction is successful, if two ROOT files are CONTENT-EQUAL to each other.

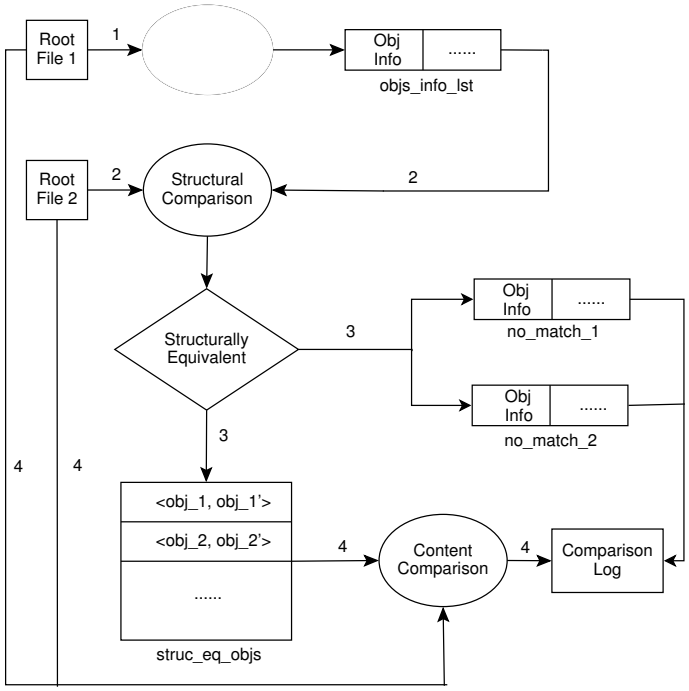

Figure 3. The ROOT_diff algorithm

The comparison procedure of *ROOT_diff* is shown in Figure 3, Step 1, Scan ROOT file 1, extract the information of each object and generate an object information list called *objs_info_lst*. Step 2, Compare the structure of each object in file 2 with every object cached in *objs_info_lst*. This step ensures that two root files have a different object order but the same object structure and contents will still be treated as equivalent. Step 3, If a matched object is found in file 1, then we generate an object information pair that has the two structurally equivalent objects from each file. All these pairs are stored in a list called *struc_eq_objs*. If every object in file 1 can find a matching object in file 2, then file 1 is STRUCTURE-EQUAL to file 2. Objects with no match are stored in lists *no_match_1* and *no_match_2* respectively for reporting purposes. Step 4, For each pair of structurally equivalent objects, the contents of the two objects are compared. If each object in file 1 has a matching object in file 2 and has the same data content, we say that file 1 is CONTENT-EQUAL to file 2. If all objects in file 1 have a CONTENT-EQUAL object in file 2 and share the same timestamp, then we say file 1 and file 2 BITWISE-EQUAL to each other.

Table 3. Ignored and Nonequal objects from LHE stage

Class Name	TTree	StreamerInfo	TTree	TTree	TTree	TTree	TTree	KeyList	FreeSegments
Object Name	Metadata	StreamerInfo	ParameterSets	Parentage	Events	LuminosityBlocks	Runs	HIG-RunIIWinter15	LuminosityBlocks
Ignored Times	1	1	1	1	1	1	1	1	1

3.2. Comparator performance

We benchmark the performace of *Root_diff* by conducting comparison on various sizes of ROOT files produced by Lobster[10]. The running time for comparing ROOT files from megabytes to gigabytes is shown in Figure 4. We also compare *ROOT_diff* with md5sum and sha1sum. The growth of the running time is not always linear, because ROOT files we chosen have different structures. As shown in the figure, *ROOT_diff* has better performance than md5sum and sha1sum, when the file size is smaller than 6GB. That is because *ROOT_diff* does not scan the entire file and ignores objects those are not related to the simulation results. It only reads the desired object buffer blocks and will stop comparing when the first different byte is encountered. When the file size hits the memory limit, *ROOT_diff* begins to suffer from the overhead caused by random reads, while *md5sum* and *sha1sum* read data sequentially without much of a decrease in performance.

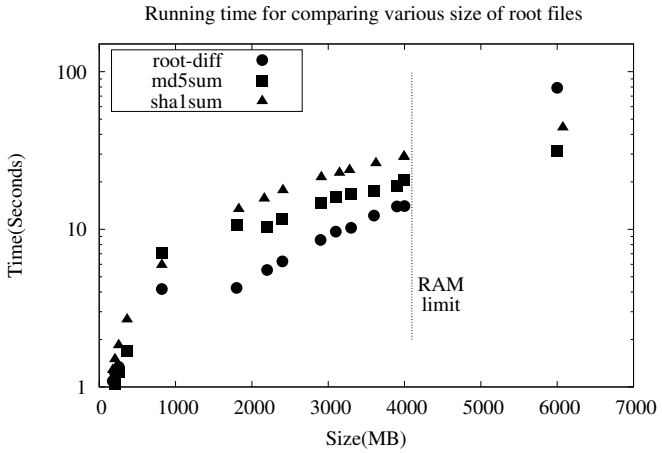

Figure 4. Performance of ROOT_diff

3.3. Comparison results for Lobster workflow

In Table 2, we show the results of comparing ROOT files generated by two simulation runs with same substages and seed. ROOT_diff ignores some of the objects that are only related to the structure of the file. Examples of ignored objects in the lhe stage are in Table 3. Four objects in file 1 named LHEEventProduct_exter have no matching objects in file 2. Four EventAuxiliary objects in file 2 cannot be matched to any object in file 1.

Stage Name		Same Seed					
		Number of Objects	Ignored	Not Equal	Structure Equal	Content Equal	Bitwise Equal
LHE		605	9	4	592	580	0
GENSIM		717	9	1	707	694	0
DIGI	1	1171	9	1	1161	1158	0
	2	6563	9	21	6533	6526	0
	3	204	21	7	176	175	0
MINIAOD		2723	9	4	2710	2671	0

Table 2. Comparison results for simulation stages

4. Virtualization Adjustments

The unix tool *strace* is an exploratory form of virtualization where system calls are captured and logged to a file. This log file can then be searched for calls to known sources of non-determinism such as random number generators provided by the kernel. While this tool is unlikely to eliminate non-determinism, we use it with the CMS workflow to identify some read flags that could be causing non-determinism.

The Parrot tool[9] is a translational form of virtualization where system calls are captured and can be modified before being forwarded to the operating system. It can be used to trick a program into behaving more deterministically[7].

4.1. Finding sources of non-determinism

Executing step 1 using strace two times in a row (keeping random seed, folders, and the machine fixed, as before) produced two log files that were very similar but had notable differences. Various categories of red flags appeared in the strace log files and in comparisons between the two files.

File and Folder names: File and folder names can be a source of non-determinism if they are later included in any output file that includes data that should be used as a basis for validation. The system call 'getcwd' was used which means that the execution could be affected by the current directory that it executes in. Exactly 16,356 total filenames were referenced with the same name in both runs. 23 additional filenames appeared in each run, but each used (what appeared to be random) file names of the format /tmp/tmpfjAMpSC.

Concurrency: Both runs included the 'execve' system call, which starts up a child process. This child process could have it's own set of issues preventing determinism.

Entropy: A total of 2 bytes were read from '/dev/urandom'. This was one of our primary concerns initially. The read occurred after the randomly named tmp files started being created, so it was not the seed for that potential source of non-determinism, at least.

Available Memory: One of the runs used 74 more (about 5% more) 'madvise' system calls than the other run. Even with the same available memory, available block sizes can cause non-deterministic behavior. Additional system calls allocating memory lead to additional time spent which can also affect the entropy of the operating system.

Input/Output: The 'socket' system call (and related calls) was used in the runs which indicates an implicit dependence on some external resources. Non-determinism in those resources and the connection to them can both cause unpredictable behavior in a task.

Time: Both the initial time and the passage of time due to possible congestion in the operating system can be an issue.

4.2. Eliminating non-determinism

Parrot was used to capture system calls made by step 1 of the CMS workflow. Parrot is often already used for high energy physics because it can translate file system requests for the CernVM File System (CVMFS) into a network request when using computing resources that can't easily mount the CVMFS file system directly (often due to Unix permission issues). A few additional flags in Parrot enabled us to translate additional system calls to encourage determinism.

Time-Stop in Parrot: A new feature in Parrot was enabled which always returns January 1st, 2000 at midnight when asked for the current system time. We hoped this would make the results for the LHE step more deterministic, however the task seemed to never complete. Upon further exploration it turns out that at one point the LHE step checks the current time and waits for a certain amount of time to pass before continuing, so it waited indefinitely.

Time-Warp in Parrot: In order to overcome the issue with the time-stop feature, a more intelligent feature called time-warp was used. The first time reported is January 1st, 2000 at midnight, but for each additional request, the time is incremented by 1/100th of a second. When set to a maximum of 1649 events, an LHE task completed on January 1st, 2001 at 4:18:57 (am). In other words, with the year 2000 being a leap year the task requested the time about 31,637,937 times. This feature in Parrot effected significant improvement in the determinism of the the task. After running the task twice, the contents of all structurally equivalent objects were also bitwise equivalent. In fact, for between 1 and 121 events the final ROOT file was bitwise identical. But for 122 and more events there were still a few objects that were different, requiring a more detailed comparison.

Fixed PID in Parrot: Looking more carefully at the 122 event cases, a colleague pointed out that aux.processGUID() causes differences in an identifier that gets included in the ROOT output file. So, a feature called pid-fixed was added to Parrot, so that the same PID would always be returned whenever the system was asked for the current PID. This could be a dangerous option if the provided PID is then used to terminate a process that happens to match an important running process, but no harm appeared to come from using this in the LHE stage. This feature made one additional object match between two LHE runs with 122 events, but additional differences still exist.

5. Conclusions

While the general causes of non-determinism in software are well known, managing them in a complex piece of software with many authors remains a challenge. Our first look at this issue highlights some of these challenges. CMS codes produce non-deterministic results, even when a random seed is fixed; and ROOT files contain provenance metadata intermixed with physics data. But, we have also shown that some of these effects can be mitigated through the use of system call interception, and various data comparison techniques. We expect that adding concurrency in the form of processes, threads, and accelerators will reveal new challenges if employed.

Going forward, our aim is to modify or augment CMS codes to achieve deterministic execution, and then to create techniques and tools to assist developers with managing additional non-determinism within the development process. We aim to improve both the productivity and reliability of computational science in high energy physics.

Acknowledgements

We gratefully acknowledge Kevin Lannon and Michael Hildreth for assistance with the CMS software stack.

This work was supported in part by the National Science Foundation under grants PHY-1247316 and OCI-1148330, and the Department of Education under grant P200A120206.

[1] A tool for one-to-one comparison of edm objects. `https://twiki.cern.ch/twiki/bin/view/CMSPublic/SWGuidePhysicsToolsEdmOneToOneComparison`. Accessed: September 28, 2016.

[2] J. Alwall, A. Ballestrero, P. Bartalini, S. Belov, E. Boos, A. Buckley, J. M. Butterworth, L. Dudko, S. Frixione, L. Garren, et al. A standard format for les houches event files. *Computer Physics Communications*, 176(4):300–304, 2007.

[3] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, P. Canal, D. Casadei, O. Couet, V. Fine, et al. Root - a c++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 182(6):1384–1385, 2011.

[4] A. Bánáti, P. Kacsuk, and M. Kozlovszky. Four level provenance support to achieve portable reproducibility of scientific workflows. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*, pages 241–244. IEEE, 2015.

[5] J. B. Buckheit and D. L. Donoho. *Wavelab and reproducible research*. Springer, 1995.

[6] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1345–1350. ACM, 2008.

[7] H. Meng, M. Wolf, P. Ivie, A. Woodard, M. Hildreth, and D. Thain. A Case Study in Preserving a High Energy Physics Application with Parrot. In *Journal of Physics: Conference Series (CHEP 2015)*, 2015.

[8] K. Popper. *The logic of scientific discovery*. Routledge, 2005.

[9] D. Thain and M. Livny. Parrot: An Application Environment for Data-Intensive Computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005.

[10] A. Woodard, M. Wolf, C. Mueller, N. Valls, B. Tovar, P. Donnelly, P. Ivie, K. H. Anampa, P. Brenner, D. Thain, et al. Scaling data intensive physics applications to 10k cores on non-dedicated clusters with lobster. In *2015 IEEE International Conference on Cluster Computing*, pages 322–331. IEEE, 2015.