Redo: Reproducibility at Scale

Ivo Jimenez, Carlos Maltzahn
(UC Santa Cruz)

Adam Moody, Kathryn Mohror
(Lawrence Livermore National Laboratory)

Introduction

A key component of the scientific method is the ability to revisit and reproduce previous results [1,2]. In high performance computing reproducibility is extremely challenging due to the large-scale nature of the environment. A result from a computational experiment has the following dependencies:

- *System.* Hardware, OS stack, system libraries, system configuration.
- *User.* Binaries/scripts and 3rd party libraries.
- *Input data and parameters.*
- *External services.* Databases, parallel file system, etc.

We use the term *Computational Job* (or just *Job*) to refer to the dependencies of an experiment. Having access to the exact same job should, ideally, enable the regeneration of a result[1]. A job's result is not solely defined by its output; instead, changes to experiment's dependencies can result in one or more of the following:

- *Accuracy.* Result is not identical to that of a previous run of the same job, or does not fall within valid range.
- *Performance.* Observe a performance difference from a previous run of the same job.
- *Portability.* Can't execute the same job on a distinct software stack, platform, or compute site.
- *Longevity.* Unable to reproduce a result due to lack of identical resources as previous run of the same job.

We argue that if we want computational reproducibility, we must define unambiguous methods to compare dependencies and results of a computational experiment. That is, we need tools that automate the process of taking a result and comparing it against a previously generated one. Similarly, tooling should allow the comparison of two jobs in order to identify changes between them. Here, we present the design of *Redo*, our reproducibility framework for capturing an experiment and its results, and automating job comparison.

[1]We say ideally since, in reality, re-executing a job might yield distinct results even if no dependencies are modified, e.g., due to non-determinism.

Redo: Our Reproducibility Framework

In order to understand *Redo*'s approach, we need to introduce first the concept of versioning and versioned repositories.

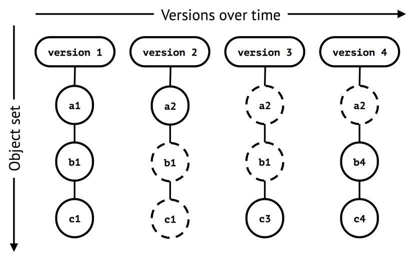

Fig. 1. Versioning of an object space.

Versioning: The ability to track multiple versions of a datum is a powerful primitive that has had a wide variety of uses in computer science. At a conceptual level, versioning captures the state of a set of objects (see Figure 1) and performs time-travel operations on them such as rollback, reset, bifurcate, or merge changes.

Repositories: We view the dependencies of an experiment and its associated results similarly to commits in the repository of a revision control system. In our case, we use repositories to store and manage snapshots of not just code but any type of data, including input data, user/system environments, as well as output data and runtime statistics.

Redo's Meta Repository

Redo maintains a meta repository (a repository of sub-repositories), in which each commit is a list of references to specific revisions in each sub-repository[2] (Figure 2). This enables the tracking of lineage for inter- and intra-dependencies of a job and its results, as well as the execution of time-travel operations, allowing a computational researcher to "rewind" or "fast-forward" to particular points in time.

Given a Redo commit ID, we envision reproducing a result by (1) checking out a commit from Redo; (2) preparing the environment and data that the commit refers to; (3)

[2]This is similar to how `git submodules` work, but Redo supports any type of repository (not just `git`) via plugins.

re-executing the job; (4) capturing and marking the generated result; (5) storing runtime statistics of the execution of the job; (6) and obtaining metrics that allow the user to compare the just-generated result and its dependencies with previous ones.

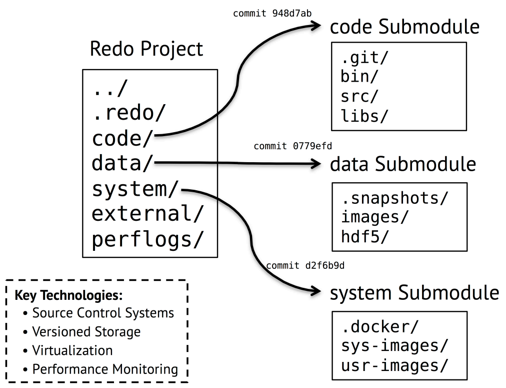

Fig. 2. A project in Redo is a super-project that captures the state of a scientist's environment by aggregating individual repositories and their associated versions.

Timelines, Bifurcations and Merges

Redo's repository can be seen as as a mechanism for capturing snapshots so that scientists can mark points in a timeline (Figure 3). Every change of a job's dependencies triggers the creation of a new point in the timeline. Multiple changes to a single version have the effect of creating multiple branches.

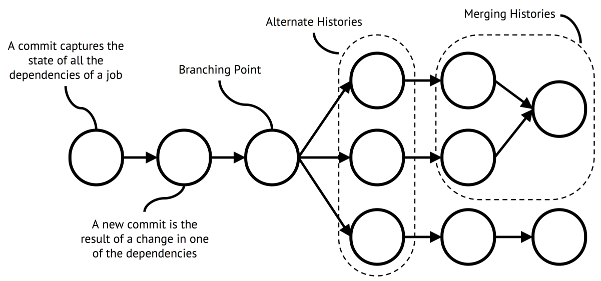
A commit captures the state of all the dependencies of a job
Branching Point
Alternate Histories
Merging Histories
A new commit is the result of a change in one of the dependencies

Fig. 3. The lineage of a project can take multiple forms, depending on the use case.

Some changes (e.g., an update to an external dependency) can result in having two branches that are never joined back. There are times when merging two timelines is needed, e.g., when two or more scientists collaborate on the same code base and they want to reconcile their changes into a single timeline.

Quantifying Reproducibility

We define a *Differentiable Type* as any type of data for which, given two values A and B, we can define a *Similarity Function* (SF) that produces values within the [0,1] range. For some types, similarity might be obtained with respect to a third (relative or absolute) baseline value C.

Examples of differentiable types are: **time** and **space** units; **variables** of an array (e.g., two temperature values); **source** files; **states** of an external service (e.g., two versions of the genome browser); virtual machine images (e.g., they can be introspected in order to determine their content).

Being able to define differentiable formats is of extreme importance in the context of reproducibility since it allows to precisely quantify how much dependencies and results of an experiment differ.

Preliminary Results

We show results of two experiments (Figure 4) that emphasise the scalability issues of reproducibility in large-scale scenarios. The first (Figure 4a) illustrates the problem of checking out a system's image from a central repository to all the nodes in a supercomputer. The dashed line corresponds to having all the nodes read from the parallel file system (PFS); the dotted line corresponds to a broadcast operation done on supercomputer's fast network.

The second (Figure 4b), typifies the trade-offs that manifest as the number of versions of a dataset increase. For a computational simulation, versions within a single timeline (e.g., checkpoints) represent complete overwrites, which results in small savings. This changes if the number of branches increases, as a result of slightly modifying a common parent version (e.g., small tweak to a checkpoint's data).

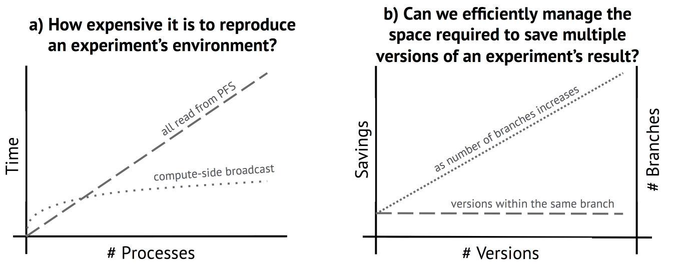

Fig. 4. Preliminary results. **Note**: These are graph mockups. We will include actual results on our final version of the poster.

References

[1] J. Freire, P. Bonnet, and D. Shasha, "Computational reproducibility: State-of-the-art, challenges, and database research opportunities," *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, New York, NY, USA: ACM, 2012, pp. 593–596.

[2] V. Stodden, F. Leisch, and R.D. Peng, *Implementing reproducible research*, CRC Press, 2014.