Popper: Making Reproducible Systems Performance Evaluation Practical

true

***Abstract*—Independent validation of experimental results in the field of parallel and distributed systems research is a challenging task, mainly due to changes and differences in software and hardware in computational environments. Recreating an environment that resembles the original systems research is difficult and time-consuming. In this paper we introduce the *Popper Convention*, a set of principles for producing scientific publications. Concretely, we make the case for treating an article as an open source software (OSS) project, applying software engineering best-practices to manage its associated artifacts and maintain the reproducibility of its findings. Leveraging existing cloud-computing infrastructure and modern OSS development tools to produce academic articles that are easy to validate. We present our prototype file system, GassyFS, as a use case for illustrating the usefulness of this approach. We show how, by following *Popper*, re-executing experiments on multiple platforms is more practical, allowing reviewers and students to quickly get to the point of getting results without relying on the author's intervention.**

I. Introduction

A key component of the scientific method is the ability to revisit and replicate previous experiments. Managing information about an experiment allows scientists to interpret and understand results, as well as verify that the experiment was performed according to acceptable procedures. Additionally, reproducibility plays a major role in education since the amount of information that a student has to digest increases as the pace of scientific discovery accelerates. By having the ability to repeat experiments, a student learns by looking at provenance information about the experiment, which allows them to re-evaluate the questions that the original experiment addressed. Instead of wasting time managing package conflicts and learning the paper author's ad-hoc experimental setups, the student can immediately run the original experiments and build on the results in the paper, thus allowing them to "stand on the shoulder of giants".

Independently validating experimental results in the field of computer systems research is a challenging task. Recreating an environment that resembles the one where an experiment was originally executed is a challenging endeavour. Version-control systems give authors, reviewers and readers access to the same code base [1] but the availability of source code does not guarantee reproducibility [2]; code may not compile, and even it does, the results may differ. In this case, validating the outcome is a subjective task that requires domain-specific expertise in order to determine the differences between original and recreated environments that might be the root cause of any discrepancies in the results [3–5]. Additionally, reproducing experimental results when the underlying hardware environment changes is challenging mainly due to the inability to predict the effects of such changes in the outcome of an experiment [6,7]. A Virtual Machine (VM) can be used to partially address this issue but the overheads in terms of performance (the hypervisor "tax") and management (creating, storing and transferring) can be high and, in some fields of computer science such as systems research, cannot be accounted for easily [8,9]. OS-level virtualization can help in mitigating the performance penalties associated with VMs [10].

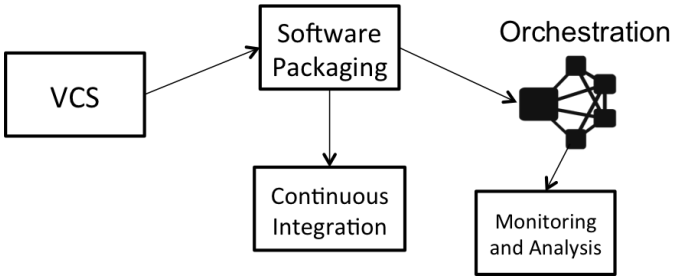

Fig. 1. The OSS development model. A version-control system is used to maintain the changes to code. The software is packaged and those packages are used in either testing or deployment. The testing environment ensures that the software behaves as expected. When the software is deployed in production, or when it needs to be checked for performance integrity, it is monitored and metrics are analyzed in order to determine any problems.

One central issue in reproducibility is how to organize an article's experiments so that readers or students can easily repeat them. The current practice is to make the code available in a public repository and leave readers with the daunting task of recompiling, reconfiguring, deploying and re-executing an experiment. In this work, we revisit the idea of an executable paper [11], which proposes the integration of executables and data with scholarly articles to help facilitate its reproducibility, but look at implementing it in today's cloud-computing world by treating an article as an open source software (OSS) project. We introduce *Popper*, a convention for organizing an article's artifacts following the OSS development model that allows researchers to make all the associated artifacts publicly available with the goal of easing the re-execution of experiments and validation of results. There are two main goals for this convention:

1. It should apply to as many research projects as possible, regardless of their domain. While the use case shown in *Section IV* pertains to the area of distributed storage systems, our goal is to embody any project with a computational component in it.
2. It should be applicable, regardless of the underlying

technologies. In general, Popper relies on software-engineering practices like continuous integration (CI) which are implemented in multiple existing tools. Applying this convention should work, for example, regardless of what CI tool is being used.

If, from an article's inception, researchers make use of version-control systems, lightweight OS-level virtualization, automated multi-node orchestration, continuous integration and web-based data visualization, then re-executing and validating an experiment becomes practical. This paper makes the following contributions:

- An analysis of how the OSS development process can be repurposed to an academic article;
- Popper: a convention for writing academic articles and associated experiments following the OSS model; and
- GasssyFS: a scalable in-memory file system that adheres to the Popper convention.

GassyFS, while simple in design, is complex in terms of compilation and configuration. Using it as a use case for Popper illustrates the benefits of following this convention: it becomes practical for others to re-execute experiments on multiple platforms with minimal effort, without having to speculate on what the original authors (ourselves) did to compile and configure the system; and shows how automated performance regression testing aids in maintaining the reproducibility integrity of experiments.

The rest of the paper is organized as follows. *Section II* analyzes the traditional OSS development model and how it applies to academic articles. *Section III* describes *Popper* in detail and gives an overview of the high-level workflow that a researcher goes through when writing an article following the convention. In *Section IV* we present a use case of a project following Popper. We discuss some of the limitations of Popper and lessons learned in *Section V*. Lastly, we review related work on *Section VI* and conclude.

II. The OSS Development Model for Academic Articles

In practice, the open-source software (OSS) development process is applied to software projects (Figure 1). In the following section, we list the key reasons why the process of writing scientific papers is so amenable to OSS methodologies. The goal of our work is to apply these in the academic setting in order to enjoy from the same benefits. We use the generic OSS workflow in Figure 1 to guide our discussion.

A. Version Control

Traditionally the content managed in a version-control system (VCS) is the project's source code; for an academic article the equivalent is the article's content: article text, experiments (code and data) and figures. The idea of keeping an article's source in a VCS is not new and in fact many people follow this practice [1,12]. However, this only considers automating the generation of the article in its final format (usually PDF). While this is useful, here we make the distinction between changing the prose of the paper and changing the parameters of the experiment (both its components and its configuration).

Ideally, one would like to version-control the entire end-to-end pipeline for all the experiments contained in an article. With the advent of cloud-computing, this is possible for most research articles[1]. One of the mantras of the DevOps movement [13] is to make "infrastructure as code". In a sense, having all the article's dependencies in the same repository is analogous to how large cloud companies maintain monolithic repositories to manage their internal infrastructure [14,15] but at a lower scale.

Tools and services: git, svn and mercurial are popular VCS tools. GitHub and BitBucket are web-based Git repository hosting services. They offer all of the distributed revision control and source code management (SCM) functionality of Git as well as adding their own features. They give new users the ability to look at the entire history of the project and its artifacts.

B. Package Management

Availability of code does not guarantee reproducibility of results [2]. The second main component on the OSS development model is the packaging of applications so that users don't have to. Software containers (e.g. Docker, OpenVZ or FreeBSD's jails) complement package managers by packaging all the dependencies of an application in an entire filesystem snapshot that can be deployed in systems "as is" without having to worry about problems such as package dependencies or specific OS versions. From the point of view of an academic article, these tools can be leveraged to package the dependencies of an experiment. Software containers like Docker have the great potential for being of great use in computational sciences [16].

Tools and services: Docker [17] automates the deployment of applications inside software containers by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux. Alternatives to docker are modern package managers such as Nix [18] or Spack [19], or even virtual machines.

C. Continuous Integration

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository frequently with the purpose of catching errors as early as possible. The experiments associated to an article is not absent of this type of issues. If an experiment's findings can be codified in the form of a unit test, this can be verified on every change to the article's repository.

Tools and services: Travis CI is an open-source, hosted, distributed continuous integration service used to build and test software projects hosted at GitHub. Alternatives to Travis CI are CircleCI, CodeShip. Other on-premises solutions exist such as Jenkins.

[1]For large-scale experiments or those that run on specialized platforms, re-executing an experiment might be difficult. However, this doesn't exclude such research projects from being able to version-control the article's associated assets.

D. Multi-node Orchestration

Experiments that require a cluster need a tool that automatically manages binaries and updates packages across machines. Serializing this by having an administrator manage all the nodes in a cluser is impossible in HPC settings. Traditionally, this is done with an ad-hoc bash script but for experiments that are continually tested there needs to be an automated solution.

Tools and services: Ansible is a configuration management utility for configuring and managing computers, as well as deploying and orchestrating multi-node applications. Similar tools include Puppet, Chef, Salt, among others.

E. Bare-metal-as-a-Service

For experiments that cannot run on consolidated infrastructures due to noisy-neighborhood phenomena, bare-metal as a service is an alternative.

Tools and services: Cloudlab [20], Chameleon and PRObE [21] are NSF-sponsored infrastructures for research on cloud computing that allows users to easily provision bare-metal machines to execute multi-node experiments. Some cloud service providers such as Amazon allow users to deploy applications on bare-metal instances.

F. Automated Performance Regression Testing

OSS projects such as the Linux kernel go through rigorous performance testing [22] to ensure that newer version don't introduce any problems. Performance regression testing is usually an ad-hoc activity but can be automated using high-level languages or [23] or statistical techniques [24]. Another important aspect of performance testing is making sure that baselines are reproducible, since if they are not, then there is no point in re-executing an experiment.

Tools and services: Aver is language and tool that allows authors to express and validate statements on top of metrics gathered at runtime. For obtaining baselines Baseliner is a tool that can be used for this purpose.

G. Data Visualization

Once an experiment runs, the next task is to analyze and visualize results. This is a task that is usually not done in OSS projects.

Tools and services: Jupyter notebooks run on a web-based application. It facilitates the sharing of documents containing live code (in Julia, Python or R), equations, visualizations and explanatory text. Other domain-specific visualization tools can also fit into this category. Binder is an online service that allows one to turn a GitHub repository into a collection of interactive Jupyter notebooks so that readers don't need to deploy web servers themselves.

III. The Popper Convention

Popper is a convention for articles that are developed as an OSS project. In the remaining of this paper we use GitHub, Docker, Binder, CloudLab, Travis CI and Aver as the tools/services for every component described in the previous section. As stated in goal 2, any of these should be swappable

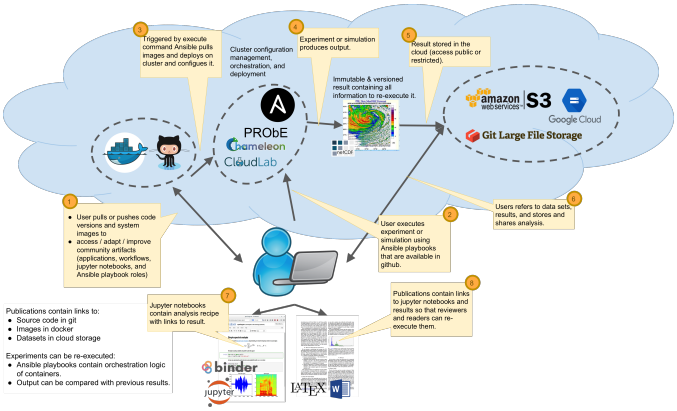

Fig. 2. End-to-end workflow for an article that follows the Popper convention.

for other tools, for example: VMs instead of Docker; Puppet instead of Ansible; Jenkins instead of Travis CI; and so on and so forth. Our approach can be summarized as follows:

- Github repository stores all details for the paper. It stores the metadata necessary to build the paper and re-run experiments.
- Docker images capture the experimental environment, packages and tunables.
- Ansible playbook deploy and execute the experiments.
- Travis tests the integrity of all experiments.
- Jupyter notebooks analyze and visualize experimental data produced by the authors.
- Every image in an article has a link in its caption that takes the reader to a Jupyter notebook that visualizes the experimental results.
- Every experiment involving performance metrics can be launched in CloudLab, Chameleon or PRObE.
- The reproducibility of every experiment can be checked by running assertions of the aver language on top of the newly obtained results.

Figure 2 shows the end-to-end workflow for reviewers and authors. Given all the elements listed above, readers of a paper can look at a figure and click the associated link that takes them to a notebook. Then, if desired, they instantiate a Binder and can analyze the data further. After this, they might be interested in re-executing an experiment, which they can do by cloning the github repository and, if they have resources available to them (i.e. one or more docker hosts), they just point Ansible to them and re-execute the experiment locally in their environment. If resources are not available, an alternative is to launch a Cloudlab, Chameleon or PRObE instance (one or more docker nodes hosted in Cloudlab) and point Ansible to the assigned IP addresses. An open question is how do we deal with datasets that are too big to fit in Git. An alternative is to use `git-lfs` to version and store large datasets.

A. Organizing Files

The structure of an example "paper repo" is shown in Figure 3. A paper is written in any desired format. Here we use mark-

down as an example (`main.md` file). There is a `build.sh` command that generates the output format (e.g. `PDF`). Every experiment in the paper has a corresponding folder in the repo. For example, for a `scalability` experiment referred in the paper, there is a `experiments/scalability/` folder in the repository.

Inside each experiment folder there is a Jupyter notebook that, at the very least, displays the figures for the experiment that appear in the paper. It can serve as an extended version of what figures in the paper display, including other figures that contain analysis that show similar results. If readers wants to do more analysis on the results data, they can instantiate a Binder by pointing to the github repository. Alternatively, If the repository is checked out locally into another person's machine, it's a nice way of having readers play with the result's data (although they need to know how to instantiate a local notebook server). Every figure in the paper has a `[source]` link in its caption that points to the URL of the corresponding notebook in GitHub[2].

For every experiment, there is an ansible playbook that can be used to re-execute the experiment. In order to do so, readers clone the repository, edit the `inventory` file by adding the IP addresses or hostnames of the machines they have available. The absolutely necessary files for an experiment are `run.sh` which bootstraps the experiment (by invoking a containerized ansible); `inventory`, `playbook.yml` and `vars.yml` which are given to ansible. The execution of the experiment will produce output that is either consumed by a postprocessing script, or directly by the notebook. The output can be in any format (CSVs, HDF, NetCDF, etc.). `output.csv` is the ultimate output of the experiment and what it gets displayed in the notebook. An important component of the experiment playbook is that it should assert the environment and corroborate as much as possible the assumptions made by the original (e.g. via the `assert` task, check that the Linux kernel is the required one). `vars.yml` contains the parametrization of the experiment.

At the root of the project, there is a `.travis.yml` file that Travis CI uses to run unit tests on every commit of the repository. For example, if an experiment playbook is changed, say, by adding a new variable, Travis will ensure that, at the very least, the experiments can be launched without any issues.

Aver can be used for checking that the original findings of an experiment are valid for new re-executions. An `assertions.aver` file contains assertions in the Aver language. This file is is given to Aver's assertion checking engine, which also takes as input the files corresponding to the output of the experiment. Aver then checks that the given assertions hold on the given performance metrics. This allows to automatically check that high-level statements about the outcome of an experiment are true.

When validating performance, an important component is to see the baseline performance of the experimental environment we are running on. Ansible has a way of obtaining "facts" about machines that is useful to have when validating results. Also, baseliner profiles that are associated to experimental results are a great way of asserting assumptions about the environment. baseliner is composed of multiple docker images that measure CPU, memory, I/O and network raw performance of a set of nodes. We execute baseliner on multi-node setups and make the profiles part of the results since this is the fingerprint of our execution. This also gives us an idea of the relationship among the multiple subsystems (e.g. 10:1 of network to IO).

[2]GitHub has the ability to render jupyter notebooks on its web interface. This is a static view of the notebook (as produced by the original author). In order to have a live version of the notebook, one has to instantiate a Binder or run a local notebook server.

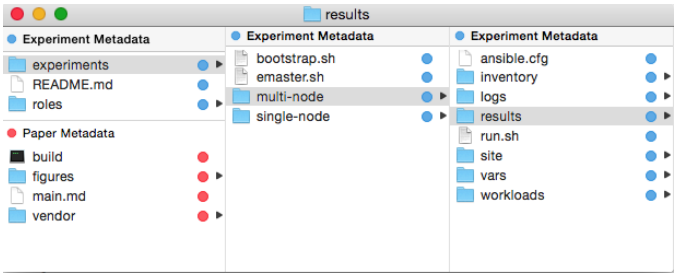

Fig. 3. Structure of a folder for a project following the Popper convention. The red markers correpond to dependencies for the generation of the PDF, while the blue ones mark files used for the experiment.

B. Organizing Dependencies

A paper repo is mainly composed of the article text and experiment orchestration logic. The actual code that gets executed by an experiment is not part of the paper repository. Similarly for any datasets that are used as input to an experiment. These dependencies should reside in their own repositories and be referenced in an experiment playbook.

1) Executables: For every execution element in the experiment playbook, there is a repository that has the source code of the executables, and an artifact repository (package manager or software image repository) that holds the executables for referenced versions. In our example, we use git and docker. Assume the execution of an scalability experiment refers to code of a `mysystem` codebase. Then:

- there's a git repo for `mysystem` that holds its source code and there's a tag/sha1 that we refer to in our experiment. This can optionally be also tracked via git submodules (e.g. placed in a `vendor/` folder).
- for the version that we are pointing to, there is a docker image in the docker hub. For example, if we reference version `v3.0.3` of `mysystem` in our experiment, then there's a docker image `mysystem#v3.0.3` in the docker hub repository. We can also optionally track the docker image's source (the `Dockerfile`) with git submodules in the paper repository (`vendor/` folder).

2) Datasets: Input/output files should be also versioned. For small datasets, we can can put them in the git repository of the paper. For large datasets we can use `git-lfs`.

IV. GassyFS: a model project for Popper

GassyFS [25] is a new prototype filesystem system that stores files in distributed remote memory and provides support for checkpointing file system state. The architecture of GassyFS is illustrated in Figure 4. The core of the file system is a user-space library that implements a POSIX file interface. File system metadata is managed locally in memory, and file data is distributed across a pool of network-attached RAM managed by worker nodes and accessible over RDMA or Ethernet. Applications access GassyFS through a standard FUSE mount, or may link directly to the library to avoid any overhead that FUSE may introduce.

By default all data in GassyFS is non-persistent. That is, all metadata and file data is kept in memory, and any node failure will result in data loss. In this mode GassyFS can be thought of as a high-volume tmpfs that can be instantiated and destroyed as needed, or kept mounted and used by applications with multiple stages of execution. The differences between GassyFS and tmpfs become apparent when we consider how users deal with durability concerns.

At the bottom of Figure 4 are shown a set of storage targets that can be used for managing persistent checkpoints of GassyFS. Given the volatility of memory, durability and consistency are handled explicitly by selectively copying data across file system boundaries. Finally, GassyFS supports a form of file system federation that allows checkpoint content to be accessed remotely to enable efficient data sharing between users over a wide-area network.

Although GassyFS is simple in design, it is relatively complex to setup. The combinatorial space of possible ways in which the system can be compiled, packaged and configured is large. For example, current version of GCC (4.9) has approximately 10^80 possible ways of compiling a binary. For the version of GASNet that we use (2.6), there are 64 flags for additional packages and 138 flags for additional features[3]. To mount GassyFS, we use FUSE, which can be given 30 different options, many of them taking multiple values.

Furthermore, it is unreasonable to ask the developer to list and test every enviroments packages, distros, and compilers. Merely listing these in a notebook or email is insufficient for sharing, colloborating, and distributing experimentl results and the methodology for sharing these environments is specific to each developer.

In subsequent sections we describe several experiments that evaluate the performance of GassyFS[4]. We note that while the performance numbers obtained are relevant, they are not our main focus. Instead, we put more emphasis on the goals of the experiments, how we can reproduce results on multiple environments with minimal effort and how we can ensure the validity of the results.

[3]This are the flags that are documented. There are many more that can be configured but not shown in the official documentation.

[4]Due to time constraints we had to limit the amount of experiments that we include. We will expand the number of experiments, as well as the number of platforms we test on.

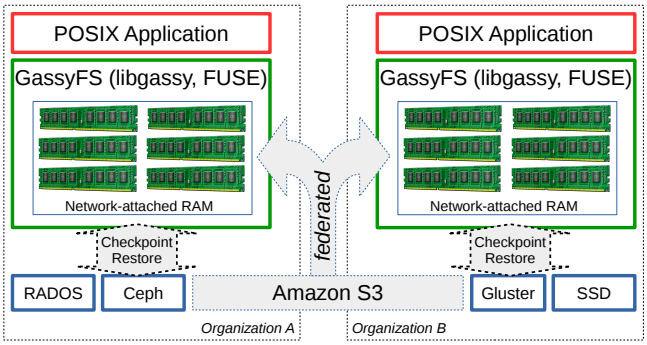

Fig. 4. GassyFS has facilities for explicitly managing persistence to different storage targets. A checkpointing infrastructure gives GassyFS flexible policies for persisting namespaces and federating data.

A. Experimental Setup

We have two sets of machines. The first two experiments use machines in Table 1. The third experiment uses machines in Table 2. While our experiments should run on any Linux kernel that is supported by Docker (3.2+), we ran on kernels 3.19 and 4.2. Version 3.13 on Ubuntu has a known bug that impedes docker containers to launch sshd daemons, thus our experiments don't run on this version. Besides this, the only requirement is to have the Docker 1.10 or newer.

TABLE I
MACHINES USED IN EXPERIMENTS 1 AND 2.

Machine ID	CPU Model	Memory BW	Release Date
M_1	Core i7-930 @2.8GHz	6x2GB DDR3	Q1-2010
M_2	Xeon E5-2630 @2.3GHz	8x8GB DDR3	Q1-2012
M_3	Opteron 6320 @2.8GHz	8x8GB DDR3	Q3-2012
M_4	Xeon E5-2660v2 @2.2GHz	16x16GB DDR4	Q3-2013

TABLE II
MACHINES USED IN EXPERIMENT 3.

Platform	CPU Model	Memory BW	Site
cloudlab	Xeon E5-2630 @2.4GHz	8x16GB DDR4	Wisconsin
cloudlab	Xeon E5-2660 @2.20GHz	16x16GB DDR4	Clemson
ec2	Xeon E5-2670 @2.6GHz	122GB DDR4	high network
ec2	Xeon E5-2670 @2.6GHz	122GB DDR4	10Gb network
mycluster	Core i5-2400 @3.1GHz	2x4GB DDR3	in-house

For every experiment, we first describe the goal of the experiment and show the result. Then we describe how we codify our observations in the Aver language. Before every experiment executes, Baseliner obtains baseline metrics for every machine in the experiment. At the end, Aver asserts that the given statements hold true on the metrics gathered at runtime. This helps to automatically check when experiments are not generating expected results

B. Experiment 1: GassyFS vs. TempFS

The goal of this experiment is to compare the performance of GassyFS with respect to that of TempFS on a single node. As mentioned before, the idea of GassyFS is to serve as a

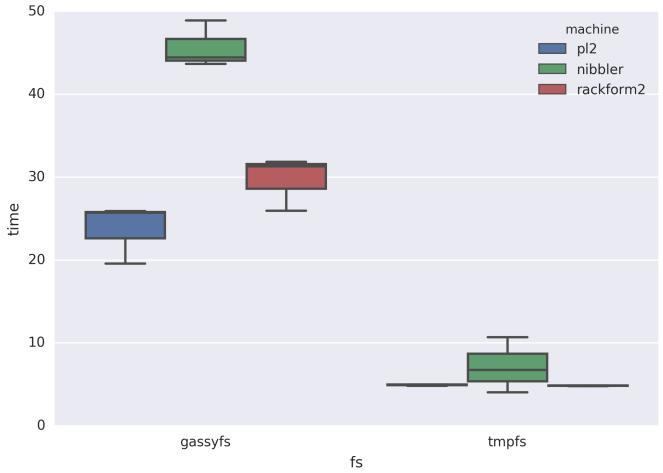

Fig. 5. [source] Boxplots comparing the variability of GassyFS vs TmpFS on a sequential `fio` workload. Every test was executed 3 times on machines listed on Table 1 (all except M_4).

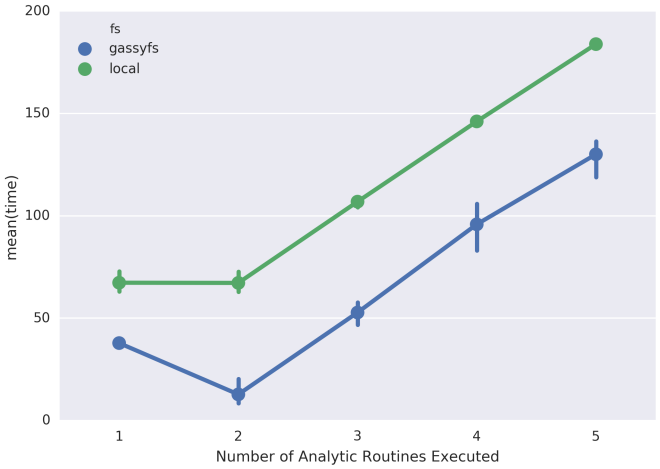

Fig. 6. [source] Performance of Dask on GassyFS vs. on the local disk. Dask is used to break the memory barrier for large datasets. Having GassyFS, users can scale-up DRAM by aggregating the memory of multiple nodes, which is an alternative to the conventional way in which that Dask is used. We show that even though Dask is efficient, having NetCDF datasets in GassyFS improves the performance significantly. The variability of the experiment comes from executing the same workload on all the machines listed in Table 1.

distributed version of TmpFS. Figure 5[5] shows the results of this test. The overhead of GassyFS over TmpFS is attributed to two main components: FUSE and GASNet. The validation statements for this experiments are the following:

```
when
  workload=*
expect
  time(fs='gassyfs') < 5 * time(fs='tmpfs')
```

The above assertion codifies the condition that, regardless of the workload, GassyFS should not be less than 5x worse than TmpFS. This number is taken from empirical evidence and from work published in [26].

C. Experiment 2: Analytics on GassyFS

One of the main use cases of GassyFS is in data analytics. By providing larger amounts of memory, an analysis application can crunch more numbers and thus generate more accurate results. The goal of this experiment is to compare the performance of Dask when it runs on GassyFS against that on the local filesystem. Dask is a python library for parallel computing analytics that extends NumPy to out-of-core datasets by blocking arrays into small chunks and executing on those chunks in parallel. This allows python to easily process large data and also simultaneously make use of all of our CPU resources. Dask assumes that an n-dimensional array won't fit in memory and thus chunks the datasets (on disk) and iteratively process them in-memory. While this works fine for single-node scenarios, an alternative is to load a large array into GassyFS, and then let Dask take advantage of the larger memory size.

Figure 6 shows the results of an experiment where Dask analyzes 5 GB worth of NetCDF files of an n-dimensional array. We see that as the number of routines that Dask executes increases, the performance of GassyFS gets closer to that of executing Dask, but up to a certain threshold. The following assertions are used to test the integrity of this result.

```
when
  fs='gassyfs'
expect
  time(num_routines = 1) < time(num_routines = 2)
;
when
  num_analytic_routines=*
expect
  time(fs='gassyfs') > time(fs='local')
```

The first condition asserts that the first time that Dask runs an analytic routine on GassyFS, the upfront cost of copying files into GassyFS has to be payed. The second statement expresses that, regardless of the number of analytic routines, it is always faster to execute Dask on GassyFS than on the local filesystem.

D. Experiment 3: Scalability

In this experiment we aim to show how GassyFS performs when we increase the number of nodes in the underlying GASNet-backed FUSE mount. Figure 7 shows the results of compiling `git` on GassyFS. We observe that once the cluster gets to 2 nodes, performance degrades sublinearly with the number of nodes. This is expected for workloads such as the one in question. The following assertion is used to test this result:

```
when
  workload=* and machine=*
expect
  sublinear(nodes,time)
```

The above expresses our expectation of GassyFS performing sublinearly with respect to the number of nodes.

[5]We don't link our figures to the corresponding notebooks (as we propose in this convention) due to double-blind review.

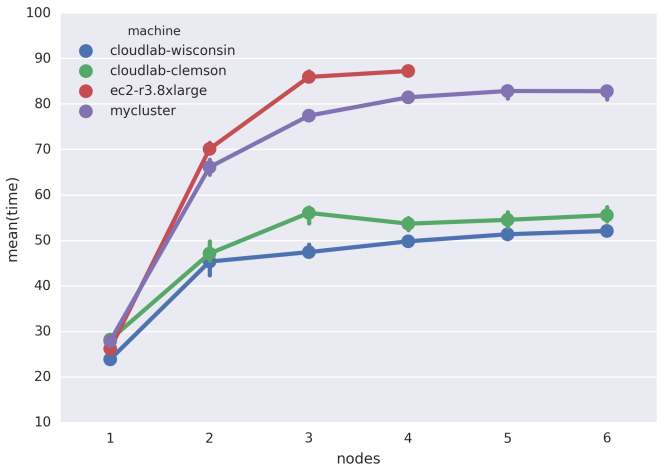

Fig. 7. [source] Scalability of GassyFS as the number of nodes in the GASNet network increases. The workload in question compiles `git`.

V. Discussion

A. We did well for 50 years. Why fix it?

Shared infrastructures "in the cloud" are becoming the norm and enable new kinds of sharing, such as experiments, that were not practical before. Thus, the opportunity of these services goes beyond just economies of scale: by using conventions and tools to enable reproducibility, we can dramatically increase the value of scientific experiments for education and for research. The Popper Convention makes not only the result of a systems experiment available but the entire experiment and allows researchers to study and reuse all aspects of it.

B. The power of "crystallization points."

Docker images, Ansible playbooks, CI unit tests, Git repositories, and Jupyter notebooks are all exemples of artifacts around which broad-based efforts can be organized. Crystallization points are pieces of technology, and are intended to be easily shareable, have the ability to grow and improvie over time, and ensure buy-in from researchers and students. Examples of very successful crystallization points are the Linux kernel, Wikipedia, and the Apache Project. Crystallization points encode community knowledge and are therefore useful for leveraging past research efforts for ongoing research as well as education and training. They help people to form abstractions and common understanding that enables them to more effectively commmunicate reproducible science. By having docker/ansible as a lingua franca for researchers, and Popper to guide them in how to structure their paper repos, we can expedite collaboration and at the same time benefit from all the new advances done in the cloud-computing/DevOps world.

C. Perfect is the enemy of good

No matter how hard we try, there will always be something that goes wrong. The context of systems experiments is often very complex and that complexity is likely to increase in the future. Perfect repeatability will be very difficult to achieve. We don't aim at perfect repeatability but to minimize the issues we face and have a common language that can be used while collaborating to fix all these reproducibility issues.

D. Drawing the line between deploy and packaging

Figuring out where something should be in the deploy framework (e.g., Ansible) or in the package framework (e.g., Docker) must be standardized by the users of the project. One could implement Popper entirely with Ansible but this introduces complicated playbooks and permantently installs packages on the host. Alternatively, one could use Docker to orchestrate services but this requires "chaining" images together. This process is hard to develop since containers must be recompiled and shared around the cluster. We expect that communities of practice will find the right balance between these technologies by improving on the co-design of Ansible playbooks and Docker images within their communities.

E. Usability is the key to make this work

The technologies underlying the OSS Development Model are not new. However, the open-source software community, in particular the DevOps community, have significantly increased the usability of the tools involved. Usability is the key to make reproducibility work: it is already hard enough to publish scientific papers, so in order to make reproducibility even practical, the tools have to be extremely easy to use. The Popper Convention enables systems researchers to leverage the usability of DevOps tools.

However, with all great advances in usability, scientists still have to get used to new concepts these tools introduce. In our experience, experimental setups that do not ensure any reproducibility are still a lot easier to create than the ones that do. Not everyone knows git and people are irritated by the number of files and submodules in the paper repo. They also usuaually misunderstand how OS-level virtualization works and do not realize that there is no performance hit, no network port remapping, and no layers of indirection. Lastly, first encounters with Docker require users to understand that Docker containers do not represent baremetal hardware but immutable infrastructure, i.e. one can't ssh into them to start services, one need to have a service per image, and one cannot install software inside of them and expect those installations to persist after relaunching a container.

F. Numerical vs. Performance Reproducibility

In many areas of computer systems research, the main subject of study is performance, a property of a system that is highly dependant on changes and differences in software and hardware in computational environments. Performance reproducibility can be contrasted with numerical reproducibility. Numerical reproducibility deals with obtaining the same numerical values from every run, with the same code and input, on distinct platforms. For example, the result of the same simulation on two distinct CPU architectures should yield the same numerical values. Performance reproducibility deals with the issue of

obtaining the same performance (run time, throughput, latency, etc.) across executions. We set up an experiment on a particular machine and compare two algorithms or systems.

We can compare two systems with either controlled or statistical methods. In controlled experiments, the computational environment is controlled in such a way that the executions are deterministic, and all the factors that influence performance can be quantified. The statistical approach starts by first executing both systems on a number of distinct environments (distinct computers, OS, networks, etc.). Then, after taking a significant number of samples, the claims of the behavior of each system are formed in statistical terms, e.g. with 95% confidence one system is 10x better than the other. The statistical reproducibility method is gaining popularity, e.g. [27].

Current practices in the Systems Research community don't include either controlled or statistical reproducibility experiments. Instead, people run several executions (usually 10) on the same machine and report averages. Our research focuses in looking at the challenges of providing controlled environments by leveraging OS-level virtualization. [28] reports some preliminary work.

Our convention can be used to either of these two approaches.

G. Controlled Experiments become Practical

Almost all publications about systems experiments underreport the context of an experiment, making it very difficult for someone trying to reproduce the experiment to control for differences between the context of the reported experiment and the reproduced one. Due to traditional intractability of controlling for all aspects of the setup of an experiment systems researchers typically strive for making results "understandable" by applying sound statistical analysis to the experimental design and analysis of results [27].

The Popper Convention makes controlled experiments practical by managing all aspects of the setup of an experiment and leveraging shared infrastructure.

H. Providing Performance Profiles Alongside Experimental Results

This allows to preserve the performance characteristics of the underlying hardware that an experiment executed on and facilitates the interpretation of results in the future.

VI. Related Work

The challenging task of evaluating experimental results in applied computer science has been long recognized [29–31]. This issue has recently received a significant amount of attention from the computational research community [4,32–34], where the focus is more on numerical reproducibility rather than performance evaluation. Similarly, efforts such as *The Recomputation Manifesto* [35] and the *Software Sustainability Institute* [36] have reproducibility as a central part of their endeavour but leave runtime performance as a secondary problem. In systems research, runtime performance *is* the subject of study, thus we need to look at it as a primary issue. By obtaining profiles of executions and making them part of the results, we allow researchers to validate experiments with performance in mind.

Recent efforts have looked at creating open science portals or repositories [37–40] that hold all (or a subset of) the artifacts associated to an article. In our case, by treating an article as an OSS project, we benefit from existing tools and web services such as git-lfs without having to implement domain-specific tools. In the same realm, some services provide researchers with the option of generating and absorving the cost of a digital object identifier (DOI). Github projects can have a DOI associated with it [41], which is one of the main reasons we use it as our VCS service.

A related issue is the publication model. In [42] the authors propose to incentivize the reproduction of published results by adding reviewers as co-authors of a subsequent publication. We see the Popper convention as a complementary effort that can be used to make the facilitate the work of the reviewers.

The issue of structuring an articles associated files has been discussed in [12], where the authors introduce a "paper model" of reproducible research which consists of an MPI application used to illustrate how to organize a project. In [1], the authors propose a similar approach based on the use of `make`, with the purpose of automating the generation of a PDF file. We extend these ideas by having our convention be centered around OSS development practices and include the notion of instant replicability by using docker and ansible.

In [43] the authors took 613 articles published in 13 top-tier systems research conferences and found that 25% of the articles are reproducible (under their reproducibility criteria). The authors did not analyze performance. In our case, we are interested not only in being able to rebuild binaries and run them but also in evaluating the performance characteristics of the results.

Containers, and specifically docker, have been the subject of recent efforts that try to alleviate some of the reproducibility problems in data science [16]. Existing tools such as Reprozip [44] package an experiment in a container without having to initially implement it in one (i.e. automates the creation of a container from an "non-containerized" environment). This tool can be useful for researchers that aren't familiar with tools

VII. Conclusion

In the words of Karl Popper: "*the criterion of the scientific status of a theory is its falsifiability, or refutability, or testability*". The OSS development model has proven to be an extraordinary way for people around the world to collaborate in software projects. In this work, we apply it in an academic setting. By writing articles following the *Popper* convention, authors can generate research that is easier to validate and replicate.

VIII. Bibliography

[1] C.T. Brown, "How we make our papers replicable."

[2] C. Collberg, T. Proebsting, and A.M. Warren, "Repeatability and Benefaction in Computer Systems Research," 2015.

[3] I. Jimenez, C. Maltzahn, J. Lofstead, A. Moody, K. Mohror, R. Arpaci-Dusseau, and A. Arpaci-Dusseau, "Tackling the Reproducibility Problem in Storage Systems Research with Declarative Experiment Specifications," *Proceedings of the 10th Parallel Data Storage Workshop*, 2015.

[4] 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*, 2012.

[5] D.L. Donoho, A. Maleki, I.U. Rahman, M. Shahram, and V. Stodden, "Reproducible Research in Computational Harmonic Analysis," *Computing in Science & Engineering*, vol. 11, Jan. 2009.

[6] R.H. Saavedra-Barrera, "CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking," PhD thesis, EECS Department, University of California, Berkeley, 1992.

[7] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, 1995.

[8] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J.N. Matthews, "Xen and the Art of Repeated Research," *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2004.

[9] G. Klimeck, M. McLennan, S.P. Brophy, G.B.A. Iii, and M.S. Lundstrom, "nanoHUB.org: Advancing Education and Research in Nanotechnology," *Computing in Science & Engineering*, vol. 10, Sep. 2008.

[10] I. Jimenez, C. Maltzahn, J. Lofstead, A. Moody, K. Mohror, R.H. Arpaci-Dusseau, and A. Arpaci-Dusseau, "The Role of Container Technology in Reproducible Computer Systems Research," *2015 IEEE International Conference on Cloud Engineering (IC2E)*, 2015.

[11] R. Strijkers, R. Cushing, D. Vasyunin, C. de Laat, A.S.Z. Belloum, and R. Meijer, "Toward Executable Scientific Publications," *Procedia Computer Science*, vol. 4, 2011.

[12] M. Dolfi, J. Gukelberger, A. Hehn, J. Imriska, K. Pakrouski, T.F. Rønnow, M. Troyer, I. Zintchenko, F.S. Chirigati, J. Freire, and D. Shasha, "A Model Project for Reproducible Papers: Critical Temperature for the Ising Model on a Square Lattice," vol. abs/1401.2000, Jan. 2014. Available at: http://arxiv.org/abs/1401.2000.

[13] A. Wiggins, "The Twelve-Factor App. The twelve-factor app."

[14] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic Configuration Management at Facebook," *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[15] C. Metz, "Google Is 2 Billion Lines of Code—And It's All in One Place. WIRED."

[16] C. Boettiger, "An introduction to Docker for reproducible research, with examples from the R environment," Oct. 2014. Available at: http://arxiv.org/abs/1410.0846.

[17] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, Mar. 2014.

[18] E. Dolstra and A. Löh, "NixOS: A Purely Functional Linux Distribution," *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 2008.

[19] T. Gamblin, M. LeGendre, M.R. Collette, G.L. Lee, A. Moody, B.R. de Supinski, and S. Futral, "The Spack Package Manager: Bringing Order to HPC Software Chaos," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[20] R. Ricci and E. Eide, "Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications,"*;login:* vol. 39, Dec. 2014–Dec. 2014.

[21] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd, "Probe: A thousand-node experimental cluster for computer systems research," *USENIX; login*, vol. 38, 2013.

[22] Intel, "Linux Kernel Performance."

[23] I. Jimenez, C. Maltzahn, A. Moody, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "I Aver: Providing Declarative Experiment Specifications Facilitates the Evaluation of Computer Systems Research," *TinyToCS*, vol. 4, 2016.

[24] T.H. Nguyen, B. Adams, Z.M. Jiang, A.E. Hassan, M. Nasser, and P. Flora, "Automated Detection of Performance Regressions Using Statistical Process Control Techniques," *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, 2012.

[25] N. Watkins, M. Sevilla, and C. Maltzahn, *GassyFS: An In-Memory File System That Embraces Volatility*, UC Santa Cruz, 2016.

[26] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok, "Terra Incognita: On the Practicality of User-Space File Systems," 2015.

[27] T. Hoefler and R. Belli, "Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[28] I. Jimenez, C. Maltzahn, J. Lofstead, A. Moody, K. Mohror, R. Arpaci-Dusseau, and A. Arpaci-Dusseau, "Characterizing and Reducing Cross-Platform Performance Variability Using OS-level Virtualization," *The First IEEE International Workshop on Variability in Parallel and Distributed Systems*, 2016.

[29] J.P. Ignizio, "On the Establishment of Standards for Comparing Algorithm Performance," *Interfaces*, vol. 2, Nov. 1971.

[30] J.P. Ignizio, "Validating Claims for Algorithms Proposed for Publication," *Operations Research*, vol. 21, May. 1973.

[31] H. Crowder, R.S. Dembo, and J.M. Mulvey, "On Reporting Computational Experiments with Mathematical Software," *ACM Trans. Math. Softw.*, vol. 5, Jun. 1979.

[32] C. Neylon, J. Aerts, C.T. Brown, S.J. Coles, L. Hatton, D. Lemire, K.J. Millman, P. Murray-Rust, F. Perez, N. Saunders, N. Shah, A. Smith, G. Varoquaux, and E. Willighagen, "Changing Computational Research: The Challenges Ahead," *Source Code for Biology and Medicine*, vol. 7, Dec. 2012.

[33] R. LeVeqije, I. Mitchell, and V. Stodden, "Reproducible Research for Scientific Computing: Tools and Strategies for Changing the Culture," *Computing in Science Engineering*, vol. 14, Jul. 2012.

[34] V. Stodden, F. Leisch, and R.D. Peng, *Implementing Reproducible Research*, 2014.

[35] I.P. Gent, "The Recomputation Manifesto," Apr. 2013. Available at: http://arxiv.org/abs/1304.3674.

[36] S. Crouch, N. Hong, S. Hettrick, M. Jackson, A. Pawlik, S. Sufi, L. Carr, D. De Roure, C. Goble, and M. Parsons, "The Software Sustainability Institute: Changing Research Software Attitudes and Practices," *Computing in Science Engineering*, vol. 15, Nov. 2013.

[37] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A.J. Elmore, S. Madden, and A.G. Parameswaran, "DataHub: Collaborative Data Science & Dataset Version Management at Scale," Sep. 2014. Available at: http://arxiv.org/abs/1409.0798.

[38] G. King, "An Introduction to the Dataverse Network as an Infrastructure for Data Sharing," *Sociological Methods & Research*, vol. 36, Jan. 2007.

[39] V. Stodden, S. Miguez, and J. Seiler, "ResearchCompendia.org: Cyberinfrastructure for Reproducibility and Collaboration in Computational Science," *Computing in Science & Engineering*, vol. 17, Jan. 2015.

[40] Center for Open Science, "The Open Science Framework. OSF."

[41] A. Smith, "Improving GitHub for science. GitHub."

[42] F. Chirigati, R. Capone, R. Rampin, J. Freire, and D. Shasha, "A collaborative approach to computational reproducibility," *Information Systems*, Mar. 2016.

[43] C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, and A. Warren, *Measuring Reproducibility in Computer Systems Research*, University of Arizona, 2014.

[44] F. Chirigati, D. Shasha, and J. Freire, "ReproZip: Using Provenance to Support Computational Reproducibility," *Proceedings of the 5th USENIX Conference on Theory and Practice of Provenance*, 2013.