Enabling Continuous Testing of HPC Systems using ReFrame

***Abstract*—Regression testing of HPC systems is of crucial importance when it comes to ensure the quality of service offered to the end users. At the same time, it poses a great challenge to the systems and application engineers to continuously maintain regression tests that cover as many aspects as possible of the user experience. In this paper, we briefly present ReFrame, a framework for writing regression tests for HPC systems and how this is used by CSCS, NERSC and OSC to continuously test their systems. ReFrame is designed to abstract away the complexity of the interactions with the system and to separate the logic of a regression test from the low-level details, which pertain to the system configuration and setup. Regression tests in ReFrame are simple Python classes that specify the basic parameters of the test plus any additional logic. The framework will load the test and send it down a well-defined pipeline which will take care of its execution. ReFrame can be easily set up on any cluster and its straightforward invocation allows it to be easily integrated with common continuous integration/deployment (CI/CD) tools, in order to perform continuous testing of an HPC system. Finally, its ability to feed the collected performance data to well known log channels, such as Syslog, Graylog or, simply, parsable log files, make it also a powerful tool for continuously monitoring the health of the system from user's perspective.**

I. Introduction

HPC systems are highly complex systems in all levels of integration; from the physical infrastructure up to the software stack provided to the end users. A small change in any of these levels could have an impact on the stability or the performance of the system. It is of crucial importance, therefore, not only to make sure that the system is in a sane condition after every maintenance, but also to monitor its performance during production, so that possible problems are detected early enough and the quality of service is not compromised.

Regression testing can provide a reliable way to ensure the stability and the performance requirements of the system, provided that sufficient tests exist that cover a wide aspect of the system's operations from both the operators' and users' point of view. However, given the complexity of HPC systems, writing and maintaining regression tests can be a very time consuming task. A small change in the system configuration or the deployment may require adapting hundreds of regression tests at the same time. Similarly, porting a test to a different system may require significant effort if the new system's configuration is substantially different from that of the system that it was originally written for.

Most HPC sites use one or another type of regression testing to check some aspects of their system behavior. These efforts are usually custom, in-house solutions that tend to couple strongly the regression tests with the system configuration, increasing the maintenance burden significantly.

In this paper, we present ReFrame, a regression testing framework that tries to address these challenges. In fact, when designing ReFrame, we have set three major goals:

1) *Productivity*. The writer of a regression test should focus only on the logical requirements of the test and should not need to deal with any of the low level system details, e.g., how the test environment is loaded, how jobs are created and submitted, how output is parsed etc.
2) *Portability*. Configuring the framework to support new systems and system configurations should be easy and should not affect the existing tests. Also, adding support of a new system in a regression test should require minimal adjustments.
3) *Robustness and ease of use*. The new framework must be stable enough and easy to use by non-advanced users. When the system needs to be returned to users outside normal working hours, the personnel in charge should be able to run the regression suite and verify the sanity of the system with a minimal involvement.

Since recently, there are ongoing discussions and several exploratory projects in progress in the HPC community toward embracing well-established cloud technologies, such as containers, Continuous Integration and Continuous Deployment (CI/CD) and DevOps. An essential ingredient of such a convergence between classical HPC and the Cloud is robust testing. ReFrame can be easily integrated with well known CI/CD frameworks, such as Jenkins or Gitlab, and can be therefore used for continuously testing a system and, even for continuously deploying changes to the system. Also since ReFrame tests are written in a fully fledged programming language, they can adjust their behavior for different systems in a self-contained manner and with minimal changes, making it easy to create and maintain a suite of tests.

The rest of the paper is organized as follows: Section II presents the key design principles of the framework, Section III presents the syntax of a ReFrame test file and discusses how common tasks can be achieved. Section IV presents some basic aspects of configuring and running ReFrame. Section V presents the use cases of ReFrame at CSCS, NERSC and OSC. Section VI discusses related work and how ReFrame is positioned among other tools. Finally, Section VII concludes the paper and presents the future development directions.

II. Framework design

ReFrame is written entirely in Python3 and follows a layered design that abstracts away the system related details (Fig. 1). An API for writing regression tests is provided to the user at

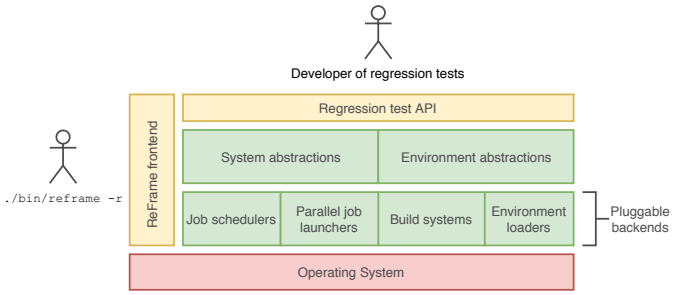


Fig. 1: ReFrame's layered architecture abstracts away the system details and allows to extend its functionality by implementing different backends.

the highest level, allowing the description of the requirements of the test. The framework defines and implements a concrete pipeline that a regression test goes through during its lifetime and the user is given the opportunity to intervene between the different stages and customize their behavior if needed. All the system interaction mechanisms are implemented as backends and are not exposed directly to the regression test developer. For example, the exact same test could be run on a system using either native Slurm or Slurm+ALPS or PBS+mpirun. Similarly, the same test can run "as-is" on system partitions configured differently. The writer of a regression test need not also care about generating a job script, querying the status of the associated job or managing the files of the test. All of these are taken care of transparently by the framework without affecting the regression test. This not only makes a regression test easier to write, but it also increases its readability, since the intent of the test is made clear right from its high-level description.

ReFrame also defines several internal APIs that interface different framework components, such as job schedulers, parallel job launchers, build systems etc. This allows the easy extension of its functionality by implementing different, independent, backends which do not affect the framework's core infrastructure.

A. The regression test pipeline

At the heart of ReFrame is the *regression test pipeline*. This is a set of well defined stages that each regression test goes through during its lifetime. Figure 2 depicts this pipeline in more detail. After initialization, each regression test will be tried for all the current system's partitions and all the programming environments supported by each partition. A test may choose to skip some systems, system partitions and/or programming environments. The tuple consisting of the regression test, the current partition and the current programming environment is called a *test case*. As soon as a test enters the pipeline, it will pass through all the stages sequentially, although some of them may be implemented as a "no-op" for certain types of tests. A detailed description of each of the pipeline stages can be found at the online documentation.

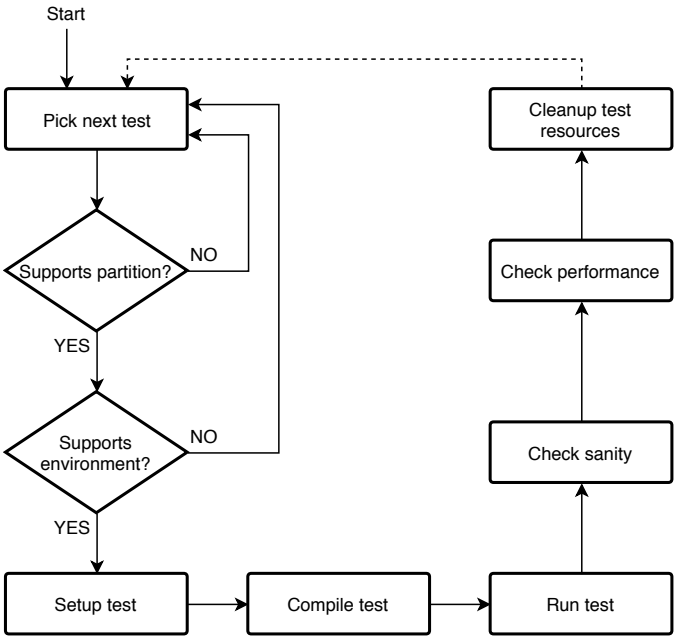


Fig. 2: The regression test pipeline. Every regression test run by ReFrame goes through these stages.

B. The frontend

The frontend of ReFrame is responsible for loading, filtering and running a set of regression tests. It communicates with the different backends and drives the regression testing pipeline. ReFrame can run a test suite in two ways: serially or asynchronously.

The serial execution policy will run all the selected tests sequentially for all their supported programming environments and system partitions, whereas the asynchronous execution policy executes the run stage of the test asynchronously. As soon as a test case is submitted for execution, ReFrame will not block and will continue by picking up the next test case to run. It will keep submitting new test cases until a user-defined concurrency limit is reached per partition. Internally, ReFrame keeps track of the currently running and ready tasks per partition and it regularly checks their status. As soon as a test case finishes, it resumes its execution from the sanity checking stage and runs it to completion. Meanwhile, it submits for execution to the corresponding partition any available ready tasks, so that it keeps the concurrency high.

C. Pluggable backends

ReFrame's core does not assume anything about the underlying system setup. Instead it communicates with the system utilities through well defined internal APIs, which are implemented by various backends. ReFrame works internally with abstract base classes, whose concrete implementations are provided as backends. As a result, the functionality of the framework may be extended easily without touching its internal structure. Currently, ReFrame defines low-level abstractions for job schedulers, parallel job launchers, modules

systems and build systems. Going into details about the internal APIs is beyond the scope of this paper, but we will briefly describe the different backends currently available in ReFrame:

1) Job scheduler backends:

- `local`: This pseudo-scheduler simply launches OS processes on the current host asynchronously.
- `slurm`: A wrapper to the SLURM job scheduler [1].
- `pbs`: A wrapper to PBS scheduler. This backend should be suitable for any PBS-derived job scheduler, e.g., Torque.

2) Parallel job launcher backends:

- `local`: An empty launcher; no command is emitted.
- `srun`: The native Slurm job launcher.
- `srunalloc`: The native Slurm job launcher but with job allocation options emitted. This is useful in combination with the `local` job scheduler, since it allows you to submit jobs directly without a job script.
- `alps`: The Cray ALPS job launcher
- `mpirun`, `mpiexec`

3) Environment modules backends:

- `nomod`: This is a pseudo modules system that simply implements the corresponding internal API as no-op. This is used for systems that do not normally have any modules system installed.
- `tmod`: The TCL implementation of the environment modules [2].
- `tmod4`: The TCL implementation version 4 of the environment modules [3].
- `lmod`: The Lua implementation of the environment modules [4].

4) Build systems backends: The build functionality inside ReFrame is abstracted away as well, so as to support different build processes seamlessly and expose an homogeneous and easy-to-use interface to the user. Build systems in ReFrame is a lightweight component that focuses more on the continuous integration aspect of compiling a code rather than the full installation process, which is the target of other more suitable tools, e.g., EasyBuild [5], Spack [6] etc. Currently, ReFrame supports the following build systems:

- `SingleSource`: This build system simply invokes the compiler on a single source file.
- `Make`: This build system invokes `make` on a directory.
- `Autotools`: This build system invokes the Autotools toolchain, i.e., `configure` and `make`.
- `CMake`: This build system invokes the CMake toolchain, i.e., `cmake` and `make`.

III. Writing a regression test in ReFrame

A regression test in ReFrame is simply a specially decorated Python class that ultimately derives from the `RegressionTest` base class. This class has several attributes that the user may set in order to control the behavior of his test. In this section, we will only present some basic aspects of the user API, in order to give the reader an idea of how a ReFrame test looks like. For a complete and detailed description of the API, the reader is referred to ReFrame's online documentation.

The following listing assumes a simple performance test that compiles and runs a CUDA-based dense matrix-vector multiplication code and checks its output and performance:

```
import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class Example7Test(rfm.RegressionTest):
    def __init__(self):
        self.descr = 'DMV CUDA performance test'
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-gnu',
                                    'PrgEnv-cray']
        self.sourcepath = 'example_dmv_cuda.cu'
        self.build_system = 'SingleSource'
        self.build_system.cxxflags = ['-O3']
        self.executable_opts = ['4096', '1000']
        self.modules = ['cudatoolkit']
        self.sanity_patterns = sn.assert_found(
            r'time for single dmv', self.stdout)
        self.perf_patterns = {
            'perf': sn.extractsingle(
                r'Performance:\s+(?P<Gflops>\S+) Gflop/s',
                self.stdout, 'Gflops', float)
        }
        self.reference = {
            'daint:gpu': {
                'perf': (50.0, -0.1, 0.1, 'Gflop/s'),
            }
        }
```

The base class of all regression tests in ReFrame is the `RegressionTest` class, so every user-defined test must eventually derive from this class. Every regression test class to be registered with ReFrame must be decorated with either the `@simple_test` or the `@parameterized_test` decorator. We will not cover the latter in this paper, but the reader is referred to the online documentation.

In a ReFrame test class, almost always, everything happens inside its constructor, which serves as the specification of the test. The important attributes in this code snippet are the following:

- `valid_systems`: This is the list of systems that this test can run on. A system is specified by its name and optionally a specific partition. In this example, this test is valid only for the `gpu` partition of the system named `daint`. Systems and their partitions are configured in ReFrame's configuration file. The test does not have to know about any irrelevant details. If the job scheduler in this system changes, the test will be still valid. Similarly, enabling this test for different systems should be as simple as extending this list and, perhaps, adapting some test-specific options. But no low-level system-specific details are exposed to the test.
- `valid_prog_environs`: This is a list of ReFrame programming environment names, that this test can be run with. These are just symbolic names that are resolved in ReFrame's configuration. The definition of a programming environment, e.g., modules that load it, can change without affecting the test. The framework will also take automatically care of any conflicts between the current

environment and the requested ones and will make sure to resolve them transparently.

- `build_system`: This is the build system that will be used for building the code of this test. Each build system defines a set of attributes that can be set to control its behavior. In this example, we simply set the compilation flags.
- `sourcepath`: This is the source file to be compiled. It can also refer to a directory, in which case `make` will be invoked automatically.
- `executable_opts`: This is a list of options to be passed to the generated executable.
- `modules`: This is a list of additional environment modules to load before compiling and running this test. ReFrame will handle automatically any conflicts and the test does not need to know the underlying modules system used.
- `variables`: This is a dictionary of environment variables to be set before compiling and running this test.
- `sanity_patterns`: This a lazily evaluated expression that will validate the output of the test. Its evaluation is lazy because it does not happen at the time it is called inside the test's constructor, but rather during the "sanity checking" stage of the regression test pipeline. ReFrame provides a library of useful *sanity functions* that a test can use directly to perform several tasks, e.g., extract and convert values from the output, calculate aggregate functions on the collected values, perform assertions etc. Users may also write their own sanity functions.
- `perf_patterns`: If this attribute is defined, then the test is considered as a performance test. It is essentially a dictionary of pairs of *performance variables* and lazily evaluated expressions, which will be used by ReFrame to extract the values of these performance variables from the output of the test. In the example shown, we extract the performance metric of Gflop/s that is printed in the standard output of the test.
- `reference`: This attribute stores the reference values for each performance variable/metric defined in `perf_patterns` for each of the supported systems or partitions. The reference information is a four-tuple consisting of the actual reference value and lower and upper thresholds expressed as fractional differences from the reference value and the measurement unit. In this example, the acceptable performance value for the `gpu` partition of the system `daint` for this test is 50 Gflop/s$\pm$10%. The reference dictionary is treated specially by ReFrame. In fact, the keys describing systems and system partitions define hierarchical scopes. When trying to resolve a performance variable, ReFrame will try several scopes in the `reference` dictionary and pick the most specific reference value. You may also define a global scope using the special `'*'` key, that would allow to run the test on unknown systems that you have not yet a reference.

There is a lot more to writing tests in ReFrame than the simple functional example that we have presented here, but it cannot be covered in this paper. The reader is referred to the online documentation for all the details and the reference guides. We will simply mention here that you can very easily differentiate the behaviour of the test (e.g., to change the compilation flags) based on the current system or the current programming environment. You may also create much more advanced sanity checking than the one presented here, purely in Python, without having to learn complex libraries or custom syntaxes. Using the `@parameterized_test` decorator you may ask ReFrame to generate families of tests by modifying individual parameters of your test. Finally, you need not take care about error handling when writing your test, since all errors are handled automatically by ReFrame's runtime.

IV. Configuring and running ReFrame

Configuring ReFrame for a new site is easy and straightforward. The site configuration comprises two basic sections: a section for configuring the systems and their partitions and another one for configuring the different programming environments. We will not cover in detail the configuration options of ReFrame; we will only outline some key configuration parameters. For configuring a new system in ReFrame, you just need to specify the modules system used (if any) and a set of hostname patterns of its login nodes that would allow ReFrame to recognize it automatically. Each system should have at least one system partition. The minimum information needed for a system partition to be configured is the job scheduler, the parallel job launcher and the programming environments to test. There are also additional options for (a) setting the required options to be passed to the backend scheduler in order to gain access to this partition, (b) specifying any necessary modules to be loaded every time ReFrame runs a test on this partition, (c) adjusting the concurrency limit for the asynchronous execution policy etc.

For programming environments, one can define the required modules that load them, set any additional environment variables and/or set the values for the basic compilers (C, C++ and Fortran) and corresponding default flags. Each programming environment is given a symbolic name, with which it is referenced also inside the tests. The same programming environment may also be redefined specially for different systems or system partitions. This allows users to reuse the same programming environment names for different systems and avoid changing their regression tests. The following listing shows a minimal configuration of a test system at CSCS.

```
site_configuration = {
    'systems': {
        'ault': {
            'descr': 'Ault␣TDS',
            'hostnames': ['ault'],
            'modules_system': 'lmod',
            'partitions': {
                'login': {
                     'scheduler': 'local',
                     'environs': ['PrgEnv-gnu'],
                     'descr': 'Login␣nodes',
                },
                'amdv100': {
                    'scheduler': 'nativeslurm',
```

```
                    'access':  ['-pamdv100'],
                    'environs': ['PrgEnv-gnu'],
                }
            }
        }
    }
    'environments': {
        'ault': {
            'PrgEnv-gnu': {
                'type': 'ProgEnvironment',
                'modules': ['gcc', 'cuda', 'openmpi'],
                'cc':  'mpicc',
                'cxx': 'mpicxx',
                'ftn': 'mpif90'
            }
        }
    }
}
```

For all possible configuration options and the exact syntax of the configuration file, the reader is referred to the online documentation.

A. Performance logging

ReFrame takes particular care in logging the performance data of performance tests. More specifically, for each performance test case and for each of the performance variables defined in the `perf_patterns` attribute of a test, ReFrame logs the achieved value and the reference. The log record format is fully customizable and may contain lots of test-specific details (e.g., job id, test directories, test tags etc.) The logging mechanism of ReFrame is built upon the Python logging framework, so it inherits lots of its functionality. Apart from the log record formatting, more important is that multiple handlers can be registered at the same time to log the performance data differently. Currently, three types of performance logging are supported:

- File: ReFrame, by default, will create a log file per test, system and system partition and will start appending data to it. The default log record prints a timestamp, the full test case description (test name, system partition and environment), the job id, and all the related performance information.
- Syslog: ReFrame can send log records to as syslog interface via the Python `logging` module.
- Graylog: ReFrame is capable of sending performance log data to a Graylog [7] server. These data can then be plotted with tools for analyzing Graylog records.

B. Running ReFrame

ReFrame is run from the command-line and offers several options that control its behavior. The simplest way to invoke it is by running "`reframe -r`". This will load all tests found in a predefined location and run them. When invoking ReFrame, after loading the configuration, the frontend goes through three phases:

1) Discovery of regression tests.
2) Filtering of regression tests.
3) Action on regression tests.

There are options controlling each of these phases. During the first phase, ReFrame searches for Python files in either a predefined location or a user-specific one, parses them and, if they contain ReFrame tests, it loads them and instantiates the tests. The following command will load and run all the tests defined in `mytest.py`: `reframe -c mytest.py -r`. The `-c` option can also be passed a directory and with the `-R` option, ReFrame will recursively search for tests inside it.

The second phase is the filtering of the loaded tests. Currently, tests in ReFrame can be filtered by name, by supported programming environments and by tags. There are options controlling each of these possibilities, and they can also be combined together to form more complex selection criteria. Each ReFrame test may be associated with a set of tags by setting its `tags` attribute. Tags allow the user to create arbitrary groups of tests, which can be invoked at any time. The following command will execute all the tests found under `mytests/` and are tagged with `production`:

```
reframe -c mytests/ -R -t production -r
```

You may also use Python regular expressions as arguments to any of the test selection options.

The final phase of the frontend is the actual action to be taken on the selected tests. Two actions are supported: (a) list tests by passing the `-l` or `-L` options and (b) run tests by passing the `-r` option. Listing of tests is quite useful for preparing a ReFrame run, since you can check which tests were actually found and selected before running them. Additional information per test is also printed, such as the file that defines it, its supported programming environments, its tags, its maintainers etc. The *run* action runs the tests. By default, all test cases are executed sequentially, but the execution policy can be changed using the `--exec-policy` option.

Apart from these basic options described briefly here, ReFrame offers several more for controlling the execution of the tests, passing through options directly to the backend scheduler, manipulating the environment modules etc. For more information the reader is referred to ReFrame's help and to the online documentation.

C. Dealing with test failures

When a regression test or particularly a test case fails, ReFrame will print a "FAIL" mark for it and continue to execute the rest of the test cases. It will not print any specific failure information inline, since this would make tracking this information difficult in large outputs. Instead, it will print a summary of all the failures with all the required details at the end of its output. The information printed includes the test name, the current system partition, the current programming environment, the current pipeline stage, the stage directory of the test case, the error message and, optionally, a stack trace depending on the type of failure. ReFrame takes particular care in properly managing errors in the context of a test execution, so as to never print error information inline. Even programming errors in a test will not break the frontend.

V. Use cases

ReFrame has been publicly released on May 2017 and is being actively developed. Since then it has gained visibility across

large computing centers, which are considering integrating it in their production testing workflows. To our knowledge, private companies in the HPC sector are using it, too. In this section, we will briefly discuss the use cases of ReFrame at the Swiss National Supercomputing Centre (CSCS) in Switzerland, at the National Energy Research Scientific Computing Center (NERSC) and at the Ohio Supercomputer Center (OSC) in the United States.

A. ReFrame at CSCS

CSCS uses ReFrame for both functionality and performance tests for all of its production systems, namely Piz Daint (Cray XC40/XC50 hybrid system), Piz Kesch (Cray CS-Storm used by MeteoSwiss for weather predition) and Monte Leone (HP DL 360 Gen 9). The same ReFrame tests are reused as much as possible across systems with minor adaptations. The test suite of CSCS (publicly available at the ReFrame repository) comprises tests for full scientific applications, scientific libraries, programming environments, compilation and linking, profiling and debugger tools, basic CUDA operations, performance microbenchmarks and I/O libraries. Using tags we have split the tests in three broad overlapping categories: production, maintenance and benchmarking tests. The first category comprises a large variety of tests and is run daily overnight using Jenkins. The maintenance suite is essentially a small subset of the production tests, comprising mostly application sanity and performance tests, as well as sanity tests for the programming environment and the scheduler. It is run before and after maintenance of the systems. The benchmarking tests are used to measure the performance of different computing and networking components and are run manually before major upgrades or when a performance problem needs to be investigated. We are currently working on a fourth category of tests that are intended to run frequently (e.g., every 10 minutes). The purpose of these tests is to measure the system behavior and performance as perceived by the users. Example tests are the time it takes to run basic Slurm commands and/or performance basic filesystem operations. Such glitches might affect the performance of running applications and cause users to open support tickets. Collecting periodically such performance data will help us correlate system events with user application performance. Finally, there is an ongoing effort to expand our ReFrame test suite to virtual clusters based on OpenStack. The new tests will measure the responsiveness of our OpenStack installation to deploy compute instances, volumes, and perform snapshots. We plan to make them publicly available in the near future.

1) Analyzing the performance data produced by ReFrame: The performance data produced by all performance tests that are run through our ReFrame installation is logged directly to an internal Graylog server. We then either use Grafana [8] queries for plotting the performance evolution over time or we use custom scripts for extracting the data and performing more advanced analysis. Our ultimate goal is to identify performance variations over time and try to correlate this with other events on the system and/or reports from users. In the following, we describe very briefly our ongoing effort on characterizing the performance of the system based on the ReFrame logs using some data analysis techniques.

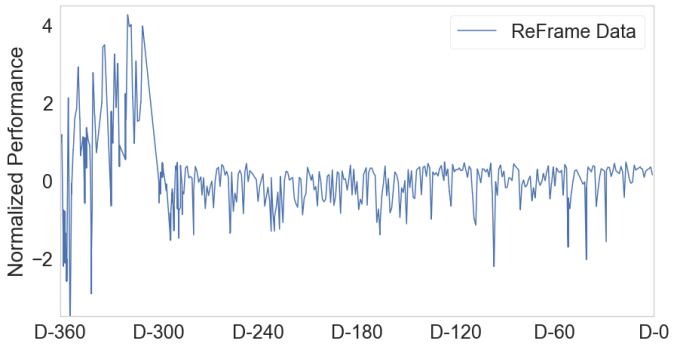
Figure 3a shows the performance data of a single GROMACS (GPU) test recorded over the past 12 months. This spiky performance "signal" is not something strange in a heavily shared HPC system. A first qualitative assessment allows to identify two to three main performance regimes in the time series based on the amplitude and frequency of the signal. However, due to the noise in the data, identifying quantitatively when the real performance degradation started is not as straightforward. We have been developing a set of tools recently with the intention to create a more automated way to determine performance changes in the system. We apply adaptive non-parametric clustering with regularization [9] to each individual ReFrame performance test, in order to identify performance *regimes* that are of particular interest.

A typical first step would be to reconstruct the signal of Fig. 3a using a rolling mean. Figure 3b shows two reconstructed signals created using different time windows. As the time window width increases, the noise will be smoothed out, but so will the information content with respect to the performance. Furthermore, the reconstructed signals still do not permit to clearly classify all performance regimes for every point in the series.

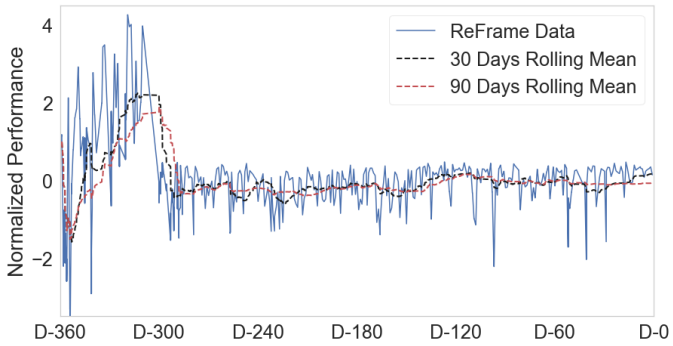
Using the aforementioned clustering approach on the raw performance signal, however, five clear performance clusters can be identified, which are shown in Fig. 3c. The sharp boundaries separating the performance clusters are a prerequisite for automation of the analysis process.

Adding variance to the analysis will give us three performance regimes, as shown in Fig. 3d, which is aligned to the initial perception by looking into the raw data. The important thing here is that clear regions where performance varies (either by its mean or its variance) can be identified, which allows an easier correlation with other tests or events. Another advantage of this method is that it does not impose any a priori probabilistic assumptions on the data. The performance and windowed variance signals were considered to be enough for this analysis since the majority of the problems should be reflected on the distribution's mean and the variance. The type of the tests' distributions are also important for characterising the overall computing system performance but it is not considered.

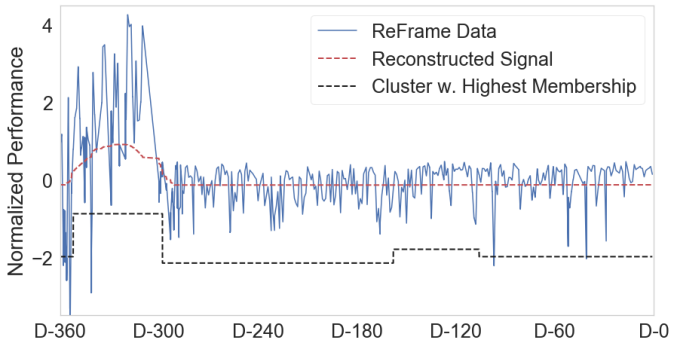
For future work, we plan to extend the analysis by adding outlier detection along with information about concurrent applications running on the same cabinet as the ReFrame test and also include the cabinet physical location in the analysis. The aim is to investigate the causality relation between performance drops and resource sharing in the cluster. The daily monitoring of the performance outliers, specially when multiple regression tests show unsatisfactory performance, can help identify faulty components of the system.



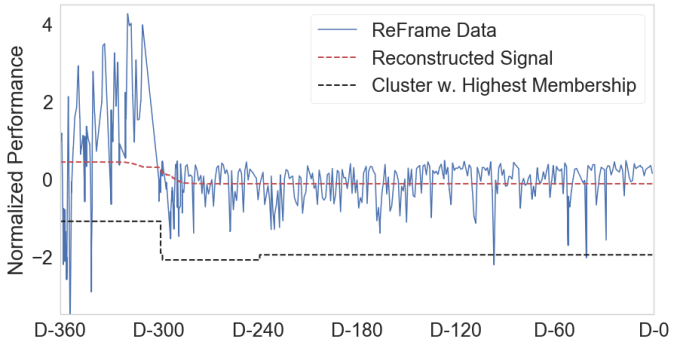
(a) ReFrame raw performance data.



(b) Reconstructed signal based on rolling mean for different window sizes.



(c) Clustering based on rolling mean (30 day window).



(d) Clustering based on rolling mean and variance.

Fig. 3: Progression from raw performance data obtained with ReFrame over the past 12 months to performance cluster indices with highest membership for a Gromacs GPU-only test.

B. *ReFrame at NERSC*

ReFrame at NERSC covers functionality and performance of its current HPC system "Cori", a Cray XC40 with Intel "Haswell" and "Knights Landing" compute nodes; as well as its smaller Cray CS-Storm cluster featuring Intel "Skylake" CPUs and NVIDIA "Volta" GPUs. The performance tests include several general-purpose benchmarks designed to stress different components of the system, including HPGMG [10]–[12] (both finite-element and finite-volume tests), HPCG [13], Graph500 [14], IOR [15], and others. Additionally, the tests include several benchmark codes used during NERSC system procurements, as well as several extracted benchmarks from full applications which participate in the NERSC Exascale Science Application Program (NESAP) [16]. Including NESAP applications ensures that representative components of the NERSC workload are included in the performance tests.

The functionality tests evaluate several different components of the system; for example, there are several tests for the Cray "DataWarp" software which enables users to interact with the Cori burst buffer. There are also several Slurm tests which verify that partitions and QoSs are correctly configured for jobs of varying sizes. The Cray programming environments, including compiler wrappers, MPI and OpenMP capability, and Shifter, are also included in these tests, and are especially impactful following changes in defaults to the programming environments.

The test battery at NERSC can be invoked both manually and automatically, depending on the need. Specifically, the full battery is typically executed manually following a significant change to the Cori system, e.g., after a major system software change, or a Cray Linux OS upgrade, before the system is released back to users. Under most other circumstances, however, only a subset of tests are typically run, and in most causes they are executed automatically. NERSC uses ReFrame's tagging capabilities to categorize the various subsets of tests, such that groups of tests which evaluate a particular component of the system can be invoked easily. For example, some performance tests are tagged as `daily`, others as `weekly`, `reboot`, `slurm`, `aries`, etc., such that it is clear from the test's Python code when and how frequently a particular test is run.

ReFrame has also been integrated into NERSC's centralized data collection service used for facility and system monitoring, called the "Data Collect" [17]. The Data Collect stores data in an Elasticsearch instance, uses Logstash to ingest log information about the Cori system, and provides a web-based GUI to display results via Kibana. Cray, in turn, provides the Cray Lightweight Log Manager [18] on XC systems such as Cori, which provides a syslog interface. ReFrame's support for syslog, and the Python standard `logging` library, enabled simple integration with NERSC's Data Collect The result of this integration with ReFrame to the Data Collect is that the results from each ReFrame test executed on Cori are visible via a Kibana query within a few seconds of the test completing. One can then configure Elasticsearch to alert a

system administrator if a particular system functionality stops working, or if the performance of certain benchmarks suddenly declines.

Finally, ReFrame has been automated at NERSC via the continuous integration (CI) capabilities provided by an internal GitLab instance. More specifically, GitLab was enhanced due to efforts from the US Department of Energy Exascale Computing Project (ECP) [19] in order to allow CI "runners" to submit jobs to queues on HPC systems such as Cori automatically via schedulable "pipelines". Automation via GitLab runners is a significant improvement over test executed automated by cron, because the runners exist outside of the Cori system, and therefore are unaffected by system shutdowns, reboots, and other disruptions. The pipelines are configured to run tests with particular tags at particular times, e.g., tests tagged with `daily` are invoked each day at the same time, tests tagged `weekly` are invoked once per week, etc.

C. ReFrame at OSC

At OSC, we use ReFrame to build the testing system for the software environment [20]. As a change is made to an application, e.g. upgrade, module change or new installation, ReFrame tests are performed by a user-privilege account and the OSC staff members who receive the test summary can easily check the result to decide if the change should be approved.

ReFrame is configured and installed on three production systems (Pitzer, Owens and Ruby). For each application we prepare the following classes of ReFrame tests:

1) default version – checks if a new installation overwrites the default module file,
2) broken executable or library – i.e. run a binary with the `--version` flag and compare the result with the module version,
3) functionality – i.e. numerical tests,
4) performance – extensive functionality checking and benchmarking,

where we currently have functionality and performance tests for a limited subset of our deployed software.

All checks are designed to be general and version independent. The correct module file is loaded at runtime, reducing the number of Python classes to be maintained. In addition, all application-based ReFrame tests are performed as regression testing of software environment when the system has critical update or rolling reboot.

ReFrame is also used for performance monitoring. We run weekly MPI tests and monthly HPCG tests. The performance data is logged directly to internal Splunk server via Syslog protocol. The job summary is sent to the responsible OSC staff member who can watch the performance dashboards.

VI. Related Work

Regression testing of HPC systems has long been an important matter among user support and operations teams of HPC centers. Several tools have been proposed in the past trying to accomplish this complex task. In this section, we present the tools and frameworks targeting HPC system testing and discuss briefly their advantages and shortcomings.

Another approach to system testing is BuildTest [21], also Python-based, which follows a slightly different path to handle regression testing. Tests in BuildTest are simple YAML files, but they are not completely self-contained as in ReFrame. Job scheduler related information is kept in separate YAML files and the tests do no specify environment modules. This can be viewed also as an advantage, since it decouples completely the test from the system, but on the other hand it makes harder to keep track of the requirements of a test. An inherent disadvantage of the YAML syntax for tests is that it imposes restrictions on the validation of the test results, since eventually you might have to move more complex test logic to an external script. Finally, BuildTest seems not to target explicitly performance tests, although it provides some basic microbenchmarks in its test suite.

Another tool that uses regression tests in YAML is the freshly released Pavilion2 from LANL [22]. One interesting feature is that you can extend the test syntax by writing plugins in Python that implement the new syntax element. This promises to make the framework easily extensible, but there is a possibility that the logic of a test is scattered between plugins and the YAML file. Since rather new, there is not yet a proof-of-concept on how this solution scales in terms of maintainability and portability to hundreds of tests.

A Python-based tool, similar to ReFrame, for tackling HPC regression testing is the Automated Testing System (ATS) [23] developed at LLNL. Tests in ATS are written in Python2 and common functionality is provided by the framework. It supports also homogeneous web reports, customizable tests with command line options, test dependencies and performance testing. However, there is no portable definition of programming environments and machine configurations require coding to set up. This makes writing even a simple test a not so straightforward process. This tool is no more actively developed.

The recently proposed Testpilot [24] is a user-centric regression testing framework which is used to verify the overall health of an HPC cluster. It is implemented as a combination of shell and Perl scripts and it consists of three "subtools" (Testpilotunit, Testpilotwing and Testpilotpatrol), which handle individual application testing, cluster regression testing and continuous assessment of a cluster health, respectively. Compared to ReFrame, it currently supports only PBS job schedulers, while the job scheduler-specific details are defined in each test and are not handled by the framework. Additionally, it does not fully support parallel execution of tests.

Another interesting approach is the JUBE framework developed at Jülich Supercomputing Centre [25]. This framework targets chiefly benchmarking rather than sanity checking and uses XML files for configuring new regression tests. Among others, it offers sandboxing of a regression test, statistical utilities for reporting the performance of the benchmarks and a parameter "template" mechanism that allows to run the same test multiple times with different parameters. However, since

it does not target specifically sanity checking, it is not as straightforward to run the exact same check with multiple programming environments. The portability of a regression test across different systems is not very easy either, since the only way to differentiate the behavior of a regression test is by masking in or out parts of the checks using special XML tags.

Toward the standardization of the deployment and regression testing of HPC systems, is also the OpenHPC initiative [26]. OpenHPC offers a regression suite, written in M4 and shell scripting, for checking deployments according to its standards. Despite the clean structure of the regression suite, a concrete knowledge of the actual deployment is still required in order to write a regression test. Also, the way of checking and reporting the test result is left upon each test. Maintaining therefore a common uniform structure across all regression tests could incur further unnecessary maintenance overhead. Another shortcoming of the OpenHPC regression suite is that it assumes a "standard" cluster setup and deployment, which makes it difficult to adopt for different cluster solutions, such as for Cray sites.

A couple of proprietary tools are also available. IBM India has developed JACE [27], which couples with other IBM tools to handle regression testing. It is menu-driven, supports parallel execution and is job-script agnostic. Additionally, a centralized report database as well as Excel spreadsheet reports allow an easy analysis of regression test outputs. Intel has also developed DART [28], a framework for distributed automated regression testing of large-scale network applications. DART supports automated execution of a suite of distributed tests, where each test consists of an XML script as well as the test code and data. Regarding HPC application testing, though, it is not clear whether DART is aware of job schedulers which are mainly used for job submission in HPC systems.

VII. Conclusions and future directions

In this paper, we presented ReFrame, a framework for writing regression tests for HPC systems and scientific applications. It was designed from ground up to allow users to easily write regression tests for their systems or software, focusing only on the logic of their tests and not mixing up the low-level system-specific details. At the same time, it does not restrict the tests' capabilities, by allowing its users to write their regression tests in a high-level modern programming language, such as Python. ReFrame's frontend allows the efficient management and execution of tests on any system (from a local desktop to a supercomputer) and enables its easy integration with well established continuous integration tools. We have also presented the concrete use cases from three supercomputing centers, namely CSCS, NERSC and OSC, that have deployed ReFrame for testing, benchmarking and monitoring their systems. Due its clean and simple interfaces, ReFrame was able to be integrated seamlessly with other tools for monitoring and continuous integration forming robust system testing workflows.

ReFrame is actively developed and publicly available on Github (https://github.com/eth-cscs/reframe). Some major features that we plan to implement in the future are seamless support for containerized tests, test dependencies and improvements to the high-level test API that would facilitate even more the writing of a test.

VIII. Acknowledgements

CSCS would like to thank the members of the Scientific Computing Support and the HPC-Operations groups for their valuable feedback regarding the framework and their contributions in writing regression tests for the system.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

[1] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. [Online]. Available: https://slurm.schedmd.com/

[2] J. L. Furlani and P. W. Osel, "Abstract yourself with modules," in *Proceedings of the 10th USENIX Conference on System Administration*, ser. LISA '96. Berkeley, CA, USA: USENIX Association, 1996, pp. 193–204. [Online]. Available: http://dl.acm.org/citation.cfm?id=1029824.1029858

[3] *Environment Modules*, Open Source. [Online]. Available: http://modules.sourceforge.net/

[4] R. McLay, *Lmod: A New Environment Module System*. [Online]. Available: https://lmod.readthedocs.io/

[5] K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt, "Easybuild: Building software with ease," in *2012 IEEE International Conference on Services Computing (SCC)*, vol. 00, Nov. 2013, pp. 572–582. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SC.Companion.2012.81

[6] 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," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 40:1–40:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807623

[7] *Enterprise Log Management for All*, Graylog Community. [Online]. Available: https://www.graylog.org/

[8] *Grafana: The open platform for beautiful analytics and monitoring*, GrafanaLabs. [Online]. Available: https://grafana.com/

[9] I. Horenko, "Finite element approach to clustering of multidimensional time series," *SIAM Journal on Scientific Computing*, vol. 32, no. 1, pp. 62–83, 2010. [Online]. Available: https://doi.org/10.1137/080715962

[10] M. Adams, J. Brown, J. Shalf, B. V. Straalen, E. Strohmaier, and S. Williams, "HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-6630E, May 2014. [Online]. Available: http://escholarship.org/uc/item/00r9w79m

[11] W. Ma, Y. Ao, C. Yang, and S. Williams, "Solving a trillion unknowns per second with HPGMG on Sunway TaihuLight," *Cluster Computing*, May 2019. [Online]. Available: https://doi.org/10.1007%2Fs10586-019-02938-w

[12] H. Shan, S. Williams, Y. Zheng, A. Kamil, and K. Yelick, "Implementing High-Performance Geometric Multigrid Solver with Naturally Grained Messages," in *2015 9th International Conference on Partitioned Global Address Space Programming Models*, Sept 2015, pp. 38–46.

[13] J. Dongarra, M. A. Heroux, and P. Luszczek, "HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems," Electrical Engineering and Compute Science Department, University of Tennessee, Knoxville, Tech. Rep. UT-EECS-15-736, Nov 2015. [Online]. Available: https://library.eecs.utk.edu/storage/594phpwDhjVNut-eecs-15-736.pdf

[14] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability Barriers for Graph exploration on distributed-memory machines," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2012, pp. 1–12.

[15] G. Lockwood. (2019) IOR and mdtest. [Online]. Available: https://github.com/hpc/ior

[16] T. Kurth, W. Arndt, T. Barnes, B. Cook, J. Deslippe, D. Doerfler, B. Friesen, Y. H. He, T. Koskela, M. Lobet, T. Malas, L. Oliker, A. Ovsyannikov, S. Williams, W.-S. Yang, and Z. Zhao, "Analyzing Performance of Selected NESAP Applications on the Cori HPC System," in *High Performance Computing*, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds. Cham: Springer International Publishing, 2017, pp. 334–347.

[17] C. Whitney, E. Bautista, and T. Davis, "The NERSC Data Collect Environment," in *Cray User Group 2016*, ser. CUG16, 2016. [Online]. Available: https://cug.org/proceedings/cug2016_proceedings/includes/files/pap101s2-file1.pdf

[18] (2019) Cray Lightweight Log Manager (LLM). [Online]. Available: https://pubs.cray.com/content/S-2393/CLE%207.0.UP00/xctm-series-system-administration-guide/cray-lightweight-log-manager-llm

[19] J. Sauers. (2018) Onyx Point works with Exascale Computing Project to bring CI to supercomputing centers. [Online]. Available: https://www.onyxpoint.com/onyxpoint-works-with-ecp-to-bring-ci-to-supercomputers/

[20] S. Khuvis, Z.-Q. You, H. Na, S. Brozell, E. Franz, T. Dockendorf, J. Gardiner, and K. Tomko, "A continuous integration-based framework for software management," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, ser. PEARC '19. New York, NY, USA: ACM, 2019, pp. 28:1–28:7. [Online]. Available: http://doi.acm.org/10.1145/3332186.3332219

[21] S. Siddiqui, *Buildtest: A HPC Application Testing Framework*. [Online]. Available: https://github.com/HPC-buildtest/buildtest

[22] *Pavilion2*. [Online]. Available: https://github.com/lanl-preteam/pavilion2

[23] P. F. Dubois, "Testing scientific programs," *Computing in Science Engineering*, vol. 14, no. 4, pp. 69–73, July 2012.

[24] K. Colby, A. K. Maji, J. Rahman, and J. Bottum, "Testpilot: A flexible framework for user-centric testing of hpc clusters," in *Proceedings of the Fourth International Workshop on HPC User Support Tools*, ser. HUST'17. New York, NY, USA: ACM, 2017, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/3152493.3152555

[25] *JUBE Benchmarking Environment*, Jülich Supercomputing Centre. [Online]. Available: https://apps.fz-juelich.de/jsc/jube/jube2/docu/index.html

[26] *OpenHPC: Community building blocks for HPC systems*. [Online]. Available: https://github.com/openhpc/ohpc

[27] S. Merchant and G. Prabhakar, "Tool for performance tuning and regression analyses of hpc systems and applications," in *2012 19th International Conference on High Performance Computing*, Dec 2012, pp. 1–6.

[28] B. N. Chun, "Dart: Distributed automated regression testing for large-scale network applications," in *Principles of Distributed Systems*, T. Higashino, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 20–36.