A regression framework for checking the health of large HPC systems

Vasileios Karakasis* Victor Holanda Rusu† Andreas Jocksch‡ Jean-Guillaume Piccinali§
Guilherme Peretti-Pezzi¶
Swiss National Supercomputing Centre (CSCS),
Via Trevano 131, Lugano, Switzerland
* *vasileios.karakasis@cscs.ch* † *victor.holanda@cscs.ch* ‡ *andreas.jocksch@cscs.ch*
§ *jeanguillaume.piccinali@cscs.ch* ¶ *guilherme.peretti-pezzi@cscs.ch*

***Abstract*—In this paper, we present a new framework for writing regression tests for HPC systems, called ReFrame. The goal of this framework is to abstract away the complexity of the interactions with the system, separating the logic of a regression test from the low-level details, which pertain to the system configuration and setup. This allows users to write easily portable regression tests, focusing only on the functionality.**

Regression tests in ReFrame are simple Python classes that specify the basic parameters of the test. The framework will load the test and will send it down a well-defined pipeline that will take care of its execution. The stages of this pipeline take care of all the system interaction details, such as programming environment switching, compilation, job submission, job status query, sanity checking and performance assessment.

Writing system regression tests in a high-level modern programming language, like Python, poses a great advantage in organizing and maintaining the tests. Users can create their own test hierarchies, can create test factories for generating multiple tests at the same time and can also customize them in a simple and expressive way.

At CSCS we have re-implemented our regression tests in ReFrame and have put the framework in production since the last upgrade of the system on December 2016. Our regression suite comprises 437 test cases that are run daily checking the system's behavior. By using ReFrame we were able to reduce our regression test codebase by almost $5\times$ compared to our old shell script based solution, covering even more test cases.

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 users. A small change in any of these levels could have an impact on the stability or the performance of the system perceived by the end users. It is of crucial importance, therefore, not only to make sure that the system is in a sane condition after every maintenance before handing it off to users, 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 system configuration or 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 than 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 before returning it to users. To our knowledge, though, these efforts are usually custom, in-house solutions that tend to couple strongly the regression tests, even those at the user level, with the system configuration. As a result the maintenance burden increases significantly, which ultimately makes people reluctant to invest more in a proper regression testing of the system.

There have been recently some efforts to "standardize" the deployment and regression testing of HPC systems through the OpenHPC initiative [1]. This is indeed an ambitious goal given the diversity in system architectures of large sites and in the actual needs of their users. Actually, Cray sites could not benefit much from this initiative, since the Cray Linux Environment (CLE) differs significantly from a standard Linux cluster deployment. Nonetheless, OpenHPC offers a regression suite for checking deployments according to its standards. Written in M4 and shell scripting, although well structured, a concrete knowledge of the actual deployment is still required in order to write a regression test. The way also 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. Notwithstanding the advantages of shell scripting, e.g., abundance of text manipulation utilities and direct interaction with the system, our experience at CSCS has shown that maintaining in the long term a large codebase of shell scripts with interdependencies can be quite a tedious task.

Another interesting approach to the problem of regression testing of large HPC systems is the JUBE framework from Jülich Supercomputing Centre [2]. This framework targets chiefly benchmarking rather than sanity checking of the system 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 regression test is by masking in or out parts of the checks using special XML tags. In fact, XML makes the description of a regression test rather difficult to follow and maintain, especially for more complex checks.

In CSCS we used to have a shell script based regression suite that has been developed over the years for testing the sanity of our systems. The key disadvantage of this suite was similar to the other suites described above: the logic of interacting with the system (e.g., job monitoring, result checking etc.) was exposed to the regression tests. As a result when the operations team started the migration to native Slurm, we had to reimplement the new submission logic to all the regression tests we had. The same had to be done every time we discovered a bug. It was apparent that this regression suite could not be maintained properly in the long term. Besides, people of different teams were quite reluctant to write new checks due to this complexity whenever they discovered and fixed something in the system.

In this paper we describe REFRAME the new regression framework we have developed in CSCS and have put in production after the upgrade of Piz Daint at the end of last year. When designing the new framework we have set three major goals:

1) *Productivity*. The writer of a regression test should focus only on the logical structure and requirements of the test and should not need to deal with any of the low level details of interacting with the system, e.g., how the environment of the test is loaded, how the associated job is created and has its status checked, how the output parsing is performed 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.

We have written the new framework entirely in Python and followed a layered design that abstracts away the system related details. An API for writing regression tests is provided to the user at 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 writer of the check. 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 by the framework without affecting the regression test. This not only makes a regression test easier to write, but it increases its readability as well, since the intent of the test is made clear right from its high-level description.

To meet the requirement of robustness we have employed a test-driven development process along with continuous integration right from the beginning of the framework's development, so as to make sure that it is tested thoroughly as it grows larger. As a matter of fact, the amount of unit test code accompanying the framework almost matches the amount of the framework's code itself. Regarding the ease of use we have tried to make the common case of invoking the regression suite as simple as possible by selecting reasonable defaults or by allowing to set default settings values per system configuration.

The rest of the paper describes in detail the REFRAME regression framework and is structured as follows: Section II presents the pipeline of the regression and discuss in detail the different phases a regression test goes through. Section III presents the API for writing a regression test and shows several real-life examples from the actual regression tests running on CSCS' systems. Section IV describes the regression's front-end and command-line interface. Section V discusses the actual use case of CSCS for migrating to the new framework and, finally, section VI summarizes and concludes the paper.

II. THE REGRESSION PIPELINE

The backbone of the REFRAME regression framework is the pipeline of the regression check. This is a set of well defined phases that each regression test goes through during its lifetime. Figure 1 depicts this pipeline in detail.

A regression test starts its life after it has been instantiated by the framework. This is where all the basic information of the test is set. At this point, although it is initialized, the regression test is not yet "live", meaning that it does not run yet. The framework will then go over all the loaded and initialized checks (we will talk about the loading and selection phases later), it will pick the next partition of the current system and the next programming environment for testing and will try to run the test. If the test supports the current

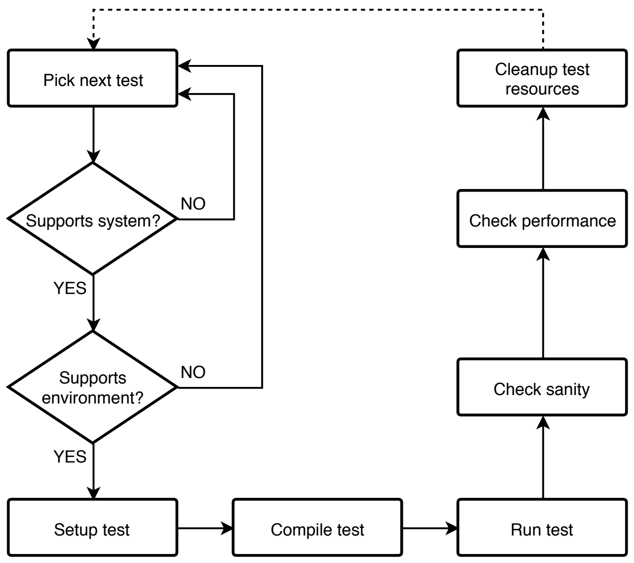

Figure 1: The regression check pipeline.

system partition and the current programming environment, it will be run and will go through all the following seven phases: (a) setup, (b) compilation, (c) running, (d) sanity checking, (e) performance checking and (f) cleanup. A test could implement some of them as no-ops. As soon as the test is finished, its resources are cleaned up and the regression's environment is restored. The regression will try to repeat the same procedure on the same regression test using the next programming environment until no further environments are left to be tested. In the following we elaborate on each of the individual phases of the lifetime of a regression test.

A. The initialization phase

Although this phase is not part of the regression check pipeline as shown in Fig. 1, it is quite important, since it sets up the definition of the regression test. It serves as the "specification" of the check, where all the needed information to run the test is set. A test could go through the whole pipeline performing all of its work without the need to override any of the pipeline stages. In fact, this is the case for the majority of tests we have implemented for CSCS production systems.

B. The setup phase

A regression test is instantiated once by the framework and it is then reused several times for each of the system's partitions and their corresponding programming environments. This first phase of the regression pipeline serves the purpose of preparing the test to run on the specified partition and programming environment by performing a number of operations described below:

(a) *Setup and load the test's environment*: At this point the environment of the current partition, the current programming environment and any test's specific environment will be loaded. For example, if the current partition requires `slurm`, the current programming environment is `PrgEnv-gnu` and the test requires also `cudatoolkit`, this phase will be equivalent to the following:

```
module load slurm
module unload PrgEnv-cray
module load PrgEnv-gnu
module load cudatoolkit
```

Note that the framework automatically detects conflicting modules and unloads them first. So the user need not to care about the existing environment at all. He only needs to specify what is needed by his test.

(b) *Setup the test's paths*: Each regression test is associated with a stage directory and an output directory. The stage directory will be the working directory of the test and all of its resources will be copied there before running. The output directory is the directory where some important output files of the test will be kept. By default these are the generated job script file, the standard output and standard error The user can also specify additional files to be kept in the test's specification. At this phase, all these directories are created.

(c) *Prepare a job for the test*: At this point a job descriptor will be created for the test, that wraps all the necessary information for generating a job script for it. However, no job script is generated yet. The job descriptor is an abstraction of the job scheduler's functionality relevant to the regression framework. It is responsible for submitting a job in a job queue and waiting for its completion. Currently, the REFRAME framework supports three job scheduler backends: (a) *local*, which is basically a "pseudo-scheduler" that just spawns local OS processes, (b) *nativeslurm*, which is the native Slurm [3] job scheduler and (c) *slurm+alps*, which uses Slurm for job submission, but Cray's ALPS [4] for launching MPI processes on the compute nodes.

C. The compilation phase

At this phase the source code associated with test is compiled with the current programming environment. Before compiling, all the resources of the test are copied to its stage directory and the framework changes into it. After finishing the compilation, the framework returns to its original working directory.

D. The run phase

This phase comprises two subphases:

(a) *Job launch*: At this subphase a job script file for the regression test is generated and submitted to the job scheduler queue. If the job scheduler for the current partition is the *local* one, a simple wrapper shell script will be generated and will be launched as a local OS process.

(b) *Job wait*: At this subphase the job (or local process) launched in the previous subphase is waited for. This phase is pretty basic: it just checks that the launched job (or local process) has finished. No check is made of whether the job or process has finished successfully or not. This is the responsibility of the next pipeline stage.

Currently, these two subphases are performed back-to-back making the REFRAME framework effectively serial, but in the future we plan to support asynchronous execution of regression tests.

E. The sanity checking phase

At this phase it is determined whether the check has finished successfully or not. Although this decision is test-specific, the REFRAME framework provides the tests with an easy way for specifying complex patterns to check in the output files. Multiple output files can be checked at the same time for determining the final sanity result. Stateful parsing (e.g., aggregate operations such as average, min, max, etc.) is also supported and implemented transparently to the user. We will present in detail the output parsing mechanism of the framework in Section III-E.

F. The performance checking phase

At this phase the performance of the regression test is checked. The framework uses the same mechanism for analyzing the output of the tests as in the sanity checking phase. The only difference is that the user can now specify reference values per system or system partition, as well as threshold values for the performance. The framework will take care of the output parsing and the matching of the correct reference values.

G. The cleanup phase

This is the final stage of the regression pipeline and cleans up the resources of the environment. Three steps are performed in this phase:

(a) The interesting files of the test (job script, standard output and standard error and any additional files specified by the user) are copied to its output directory for later inspection and bookkeeping,
(b) the stage directory is removed and
(c) the test's environment is revoked.

At this point the regression's environment is clean and in its original state and the regression can continue by either running the same test with a different programming environment or moving to another test.

H. Types of regression tests

As mentioned earlier, a regression test may skip any of the above pipeline stages. The REFRAME framework provides three basic types of regression tests:

(a) *Normal test*: This is the typical test that goes through all the phases of the pipeline.
(b) *Run-only test*: This test skips the compilation phase. This is quite a common type of test, especially when you want to test the functionality of installed software. Note that in this test the copying of the resources to the stage directory happens during the run phase and not during the compilation phase, as is the case of a normal test.
(c) *Compile-only test*: This test a special test that skips completely the run phase. However it makes available the standard output and standard error of the compilation, so that their output can be parsed during the check sanity phase in the same way that this happens for the other type of tests.

III. WRITING A REGRESSION TEST

The regression pipeline described in the previous section is implemented in a base class called `RegressionTest`, from which all user regression tests must eventually inherit. There exist also base classes (inheriting also from `RegressionTest`) that implement the special regression test types described in Section II-H. A user test may inherit from any of the base regression tests depending on the type of check (*normal*, *run-only* or *compile-only*).

The initialization phase of a regression test is implemented in the test's constructor, i.e., the `__init__()` method of the regression test class. The constructor of a user regression test is only required to allow keyword arguments to be passed to it. These are needed to initialize the base `RegressionTest`. Of course, a user regression test may accept any number of positional arguments that are specific to the test and are used to control its construction. The following listing shows the boiler plate code for implementing new regression tests classes:

```
class HelloTest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__(
            'test_name',
            os.path.dirname(__file__),
            **kwargs)
        # test's specification
```

The base class' constructor needs two positional arguments that must be supplied by the user tests: (a) the name of the test and (b) its prefix. The prefix of a regression test is normally the directory it resides in and it will be used in later phases for resolving relative paths for accessing the test's resources.

The rest of the regression pipeline stages are implemented as different methods by the base class `RegressionTest`. Normally, a user test does not need to override them, unless it needs to modify the default behavior. Even in this case though, a user test need not care about any of the phase implementation details, since it can delegate the actual implementation to the base class after or before its intervention. We will show several examples for modifying the default behavior of the pipeline phases in this section. A list of the actual `RegressionTest`'s methods that implement the different pipeline stages follows:

- `setup(self,system,environ,**job_opts)`: Implements the setup phase. The `system` and `environ` arguments refer to the current system partition and environment that the regression test will run. The `job_opts` arguments will be passed through the job scheduler backend. This is used by the front-end in special invocation of the regression suite, e.g., submit all regression test jobs in a specific reservation.
- `compile(self)`: Implements the compilation phase.
- `run(self)` and `wait(self)`: These two implement the run phase. The first call is asynchronous; it returns as soon as the associated job or process is submitted or created, respectively.
- `check_sanity(self)`: Implements the sanity checking phase.
- `check_performance(self)`: Implements the performance checking phase.
- `cleanup(self, remove_files=False)`: Cleans up the regression tests resources and unloads its environment. The `remove_files` flag controls whether the stage directory of the regression test should be removed or not.

As we shall see later in this section, user regression tests that override any of these methods usually do a minimal setup (e.g., setup compilation flags, adapt their internal state based on the current programming environment) and call the base class' corresponding method passing through all the arguments.

The following listing shows a complete user regression test that compiles a "Hello, World!" C program for different programming environments:

```
import os

from reframe.core.checks import \
     RegressionTest

class HelloWorldTest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__(
            'hello_world',
            os.path.dirname(__file__),
            **kwargs)
        self.descr = 'Hello␣World␣C␣Test'
        self.sourcepath = 'hello.c'
        self.valid_systems = [
            'daint:gpu', 'daint:mc' ]
        self.valid_prog_environs = [
            'PrgEnv-cray',
            'PrgEnv-gnu',
            'PrgEnv-intel',
            'PrgEnv-pgi' ]
        self.sanity_patterns = {
            '-': {'Hello,␣World\!': []} }

def _get_checks(**kwargs):
    return [ HelloWorldTest(**kwargs) ]
```

After the base class' constructor is called with the boiler plate code we showed before, the specification of the test needs to be set. The `RegressionTest` base class makes available a set of member variables that can be used to set up the test. All these variables are dynamically type checked; if they are assigned a value of different type, a runtime error will be raised and the regression test will be skipped from execution. Due to space limitations, we will not go into all of the `RegressionTest`'s member variables. We will only discuss the most important ones that come up in our examples.

The "Hello, World!" example shown above shows a minimal set of variables that needs to be set for describing a test. `descr` is an optional arbitrary textual description of the test that defaults to the test's name if not set. `sourcepath` is a path to either a source file or a source directory. By default, source paths are resolved against `'<testprefix>/src'`, where `testprefix` is stored in the `self.prefix` member variable. If `sourcepath` refers to a file, it will be compiled picking the correct compiler based on its extension (C/C++/Fortran/CUDA). If it refers to a directory, REFRAME will invoke `make` inside that directory. `valid_systems` and `valid_prog_environs` are the variables that basically enable a test to run on certain systems and programming environments. They are both simple list of names. The names need not necessarily correspond to a configured system/partition or programming environment, in which case the test will be ignored (see Section IV-E on how a new systems and programming environments are configured). The system name specification follows the syntax `<sysname>[:<partname>]`, i.e., you can either a specify a whole system or a specific partition in that system. In our example, this test will run on both the "gpu" and "mc" partitions of Daint. If we specified simply "daint" in our example, then the above test would be eligible to run on any configured partition for that system. Next, you need to define the `sanity_patterns` variable, which tells the framework what to look for in the standard output of the test to verify the sanity of the check. We will cover the output parsing capabilities of REFRAME in detail in Section III-E.

Finally, each regression test file must provide the special method `_get_checks()`, which instantiates the user tests of the file. This method is the entry point of the framework into the user tests and should return a list of `RegressionTest` instances. From the framework's point of view, a regression test file is simply a Python module file that is loaded by the framework and has its `_get_checks()` method called. This allows for maximum flexibility in writing regression tests, since a user can create his own hierarchies of tests and even test factories for generating sequences of related tests. In fact, we have used this capability extensively while developing the Piz Daint's regression tests and has allowed us to considerably reduce code duplication and maintenance costs of regression tests.

A. Setting up job submission

ReFrame aims to be job scheduler agnostic and to support different job options using a unified interface. To the regression check developer, the interface is manifested by a simple collection of member variables defined inside the regression check. These variables are converted internally by the framework to express the appropriate job options for a given scheduler. Table I shows a listing of these variables and their interpretation in SLURM. Normally, these variables are set during the initialization phase of a regression test. Since the system that the framework is running on is already known during the initialization phase of the regression test, it is possible to customize these variables based on the current system without the need of overwriting any method. An example is shown in the following listing:

```
def __init__(self, **kwargs):
    ...
    if self.current_system.name == 'A':
        self.num_tasks = 72
    else:
        self.num_tasks = 192
```

An advantage of writing regression tests in a high-level language, such as Python, is that one can take advantage of features not present in classical shell scripting. For example, one can create groups of related tests that share common characteristics and/or functionality by implementing them in a base class, from which all the related concrete tests inherit. This eliminates unnecessary code duplication and reduces significantly the maintenance cost. An example could be the implementation of system acceptance tests, where longer wall clock times may be required compared to the regular everyday production tests. For such tests, one could define a base class, like in the example below, that would implement the longer wall clock time feature instead of modifying each job individually:

```
class AcceptanceTest(RegressionTest):
    def __init__(self, **kwargs):
        ...
        self.time_limit = (24, 0, 0)

class HPLTest(AcceptanceTest):
        ...
```

Even though the set of variables described on Table I are enough to accommodate most of the common regression scenarios, some regression tests, especially those related to a scheduler, may require additional job options. Supporting all job options from all schedulers is a virtually impossible task. Therefore ReFrame allows the definition of custom job options. These options can be appended to the test's job descriptor during the test's setup phase. In the following example, a memory limit is passed explicitly to the backend scheduler, here SLURM:

```
class MyTest(RegressionTest):
    ...
    def setup(self, system,
              environ, **job_opts):
        super().setup(system,
                      environ,
                      **job_opts)
        self.job.options += [
            '--mem=120000' ]
```

Note that the job option[1] is appended after the call to the superclass' `setup()` method, since this is responsible for initializing the job descriptor. Keep in mind that adding custom job options tights the regression test to the scheduler making it less portable, unless proper action is taken. Of course, if there is no need to support multiple schedulers, adding any job option becomes trivial as shown in the example above.

B. Setting up the environment

ReFrame allows the customization of the environment of the regression tests. This can be achieved by loading and unloading environment modules and by defining environment variables. Every regression test may define its required modules using the `self.modules` variable.

```
self.modules = [ 'cudatoolkit',
                 'cray-libsci_acc',
                 'fftw/3.3.4.10' ]
```

These modules will be loaded during the test's setup phase after the programming environment and any other environment associated to the current system partition are loaded. Recall from section II-B that the test's environment setup is a three-step process. The modules associated to the current system partition are loaded first, followed by the modules associated to the programming environment and, finally, the regression test's modules as described in its `self.modules` variable. If there is any conflict between the listed modules and the currently loaded modules, ReFrame will automatically unload the conflicting ones. The same sequence of module loads and unloads performed during the setup phase is generated in the job script that is submitted to the job

[1]This is not to be confused with the `job_opts` argument, which is meant for controlling the job submission from the front-end.

Table I: Regression checks' member variables related to job options and their SLURM scheduler associated options

`RegressionTest`'s member variable	Interpreted SLURM option
`self.time_limit = (10, 20, 30)`	`#SBATCH --time=10:20:30`
`self.use_multithreading = True`	`#SBATCH --hint=multithread`
`self.use_multithreading = False`	`#SBATCH --hint=nomultithread`
`self.exclusive = False`	`#SBATCH --exclusive`
`self.num_tasks=72`	`#SBATCH --ntasks=72`
`self.num_tasks_per_node=36`	`#SBATCH --ntasks-per-node=36`
`self.num_cpus_per_task=2`	`#SBATCH --cpus-per-task=2`
`self.num_tasks_per_core=2`	`#SBATCH --ntasks-per-core=2`
`self.num_tasks_per_socket=36`	`#SBATCH --ntasks-per-socket=36`

scheduler. Note that programming environments modules need not be listed in the `self.modules` variable, since they are defined inside ReFrame's configuration file and the framework takes automatically care of their loading during the test's setup phase.

Since the actual loading of environment modules happens during the setup phase of the regression test, it is important to define the `self.modules` list before calling the `RegressionTest`'s `setup()` method. The common scenario is to define the list of modules in the initialization phase, but on certain occasions, the modules of a test might need to change depending on the programming environment. In these situations, it is better to create a mapping between the module name and the programming environment and override the `setup()` method to set the `self.modules` according to the current programming environment. Following is an actual example from CSCS' Score-P regression tests [5]:

```
def __init__(self, **kwargs):
    ...
    self.valid_prog_environs =
        [ 'PrgEnv-cray',
          'PrgEnv-gnu',
          'PrgEnv-intel',
          'PrgEnv-pgi' ]

    self.scorep_modules = {
      'PrgEnv-cray'  :
        'Score-P/3.0-CrayCCE-2016.11',
      'PrgEnv-gnu'   :
          'Score-P/3.0-CrayGNU-2016.11',
      'PrgEnv-intel' :
          'Score-P/3.0-CrayIntel-2016.11',
      'PrgEnv-pgi'   :
          'Score-P/3.0-CrayPGI-2016.11' }

def setup(self, system,
          environ, **job_opts):
    self.modules = [
          self.scorep_modules[environ.name]]
    super().setup(system, environ,
                  **job_opts)
```

Of course, one could just differentiate inside the `setup()` method, but the approach shown above is cleaner and moves more information inside the test's specification.

In addition to custom modules, users can also define environment variables for their regression tests. In this case, the variable `self.variables` is used, which is as a dictionary where the keys are the names of the environment variables and the values match the environment variables' values:

```
self.variables = { 'VAR' : 'value' }
```

This dictionary can be used, for example, to define the value of the `OMP_NUM_THREADS` environment variable. In order to set it to the number of cpus per tasks of the regression test, one can set it as follows:

```
self.variables = {
    'OMP_NUM_THREADS' :
     str(self.num_cpus_per_task)
}
```

C. Customising the compilation phase

ReFrame supports the compilation of the source code associated with the test. As discussed in the hello world example, if the provided source code is a single source file (defined by the member variable `self.sourcepath`), the language will be detected from its extension and the file will be compiled. By default, the source files should be placed inside a special folder named `src/` inside the regression test's folder. In the case of a single source file, the name of the generated executable is the name of the regression test. ReFrame passes no flags to the programming environment's compiler; it is left to the writer of a regression test to specify any if needed. This can be achieved by overriding the `compile()` method as shown in the example below:

```
def compile(self):
    self.current_environ.cflags = '-O3'
    self.current_environ.cxxflags = '-O3'
    self.current_environ.fflags = '-O3'
    super().compile()
```

Note that it is not possible to specify the compilation flags during the initialization phase of the test, since the current programming environment is not set yet.

If the compilation flags depend on the programming environment, like for example the OpenMP flags for different compilers, the same trick as with the **self**.modules described above can be used, by defining the flag mapping during the initialization phase and using it during the compilation phase:

```
def __init__(self, **kwargs):
    ...
    self.prgenv_flags = {
        'PrgEnv-cray'  : '-homp',
        'PrgEnv-gnu'   : '-fopenmp',
        'PrgEnv-intel' : '-openmp',
        'PrgEnv-pgi'   : '-mp' }

def compile(self):
    flag = self.prgenv_flags[
        self.current_environ.name]
    self.current_environ.cflags = flag
    self.current_environ.cxxflags = flag
    self.current_environ.fflags = flag
    super().compile()
```

If the test comprises multiple source files, a Makefile must be provided and self.sourcepath must refer to a directory (this is the default behavior if not specified at all). REFRAME will issue the make command inside the source directory. Note that in this case it is not possible for REFRAME to guess the executable's name, so this must be provided explicitly through the **self**.executable variable. Additional options can be passed to the make command and even non-standard makefiles may be used as it is demonstrated in the example below:

```
def __init__(self, **kwargs):
    ...
    self.executable = './executable'

def compile(self):
    self.current_environ.cflags = '-O3'
    super().compile(makefile='build.mk',
    options="PREP='scorep'")
```

The generated compilation command in this case will be

```
make -C <stagedir> -f build.mk \
    PREP='scorep' CC='cc' CXX='CC' \
    FC='ftn' CFLAGS='' \CXXFLAGS='' \
    FFLAGS='' LDFLAGS=''
```

Finally, pre- and post-compilation steps can be added through special variables (e.g., a configure step may be needed before compilation), however, REFRAME is not designed to be an automatic compilation and deployment tool.

D. Customising the run of a test

REFRAME offers several other options for customizing the behavior of regression tests. Due to space limitations, we only list some of them here. For a complete list, the reader is referred to the online documentation.

1) Executable options: REFRAME allows a list of options to be passed to regression check executable.

```
def __init__(self, **kwargs):
    ...
    self.executable = './a.out'
    self.executable_opts = [
        '-i␣inputfile',
        '-o␣outputfile' ]
```

These options are passed to the executable, which will be invoked but the scheduler launcher. In the above example, the executable will be launched as follows with the SLURM scheduler:

```
srun ./a.out -i inputfile -o outputfile
```

2) Pre- and post-run commands: The framework allows the execution of additional commands before and/or after the scheduler launcher invocation. This can be useful for invoking pre- or post-processing tools. We use this feature in our Score-P tests, where we need to print out and check the produced traces:

```
def setup(self, system, environ,
          **job_opts):
    super().setup(system, environ,
                  **job_opts)
    self.job.pre_run = [
        'ulimit␣-s␣unlimited' ]
    ...
    self.job.post_run = [
        'otf2-print␣traces.otf2' ]
```

3) Changing the regression test resources path: REFRAME allows individual regressions tests to define a custom folder for their resources, different than the default src/ described in Section IV-D. This is especially important for applications with a large number of input files or large input files, where these input files may need to be saved in a different filesystem due to several reasons, such as filesystem size, I/O performance, network configuration, backup policy etc. The location of this folder can be changed be redefining the self.sourcesdir variable:

```
def __init__(self, **kwargs):
    ...
    self.sourcesdir = '/apps/inputs/'
```

4) Launcher wrappers: In some cases, it is necessary to wrap the scheduler launcher call with another program. This is the typical case with debuggers of distributed programs, e.g., ddt [6]. This can be achieved in REFRAME by changing the job launcher using the special LauncherWrapper object. This object wraps a launcher with custom command:

```
def __init__(self, **kwargs):
    ...
    self.ddt_options = '--offline'

def setup(self, system,
          environ, **job_opts):
    super().setup(system,
                  environ, **job_opts)
    self.job.launcher =
    LauncherWrapper(
        self.job.launcher,
        'ddt', self.ddt_options)
```

Note that this test remains portable across different job launchers. If it runs on a system with native SLURM it will be translated to

```
ddt --offline srun ...
```

whereas if it run on a system with ALPS it will be translated to

```
ddt --offline aprun ...
```

E. Output parsing and performance assessment

REFRAME provides a powerful mechanism for describing the patterns to look for inside the output and/or performance file of a test without the need to override the check_sanity() or check_performance() methods. It allows you to search for multiple different patterns in different files and also associate callback functions for interpreting the matched values and for deciding on the validity of the match. Both the sanity_patterns and perf_patterns follow the exact same syntax and the framework's parsing algorithm is the same as well; the only difference is that a reference value is looked for in the case of performance matching. In the following, we will use sanity_patterns in our examples and we will elaborate only on the additional traits of the perf_patterns when necessary.

The syntax of the sanity_patterns is the following:

```
self.sanity_patterns = {
  <filepatt> : {
    <pattern_with_tags> : [
      (
        <tag>,
        <conv>,
        <action>
      ),
      # rest of tags in pattern
    ],
    # more patterns to look for in file
  },
  # more files
}
```

The filepatt is any valid shell file pattern but it can also take special values for denoting standard output and error: (a) '&1' or '-' for standard output and (b) '&2' for standard error. The pattern_with_tags is a Python regular expression with named groups. For example, the regular expression '(?P<lineno>\d+):␣result:␣(?P<result>\S+)' matches a line starting with an integer number followed by ':␣result:␣', followed by a non-whitespace string. The integer number is stored in match group named lineno and the second in a group named result. These group names will be used as tags by the framework to invoke different actions on each match. The regular expression pattern in the sanity_patterns' syntax is followed by a list of tuples that associate a tag with a conversion function (conv) and an action to be taken. For each of the tags of the matched pattern, the framework will call action(conv(tagvalue), reference). The reference is set to None for sanity patterns checking, but for performance patterns it is looked up in a special dictionary of reference values (we will discuss this later in this section). The action callable must return a boolean denoting whether the tag should be actually considered as matched or not.

If you are not interested in associating actions with tags, you can place an empty list instead. In that case, a named group in the regular expression pattern is not needed either, as is the case of the "Hello, World" example, where the parsing mechanism behaves like a simple grep command invocation:

```
self.sanity_patterns = {
    '-' : { 'Hello,␣World\!' : [] }
}
```

To better support stateful parsing (see below), REFRAME also define a special regex pattern ('\e') that matches the end-of-file. This pattern can only be associated with a callback function taking no arguments, which will be called after the processing of the file has finished.

Figure 2 shows general concept of the algorithm used by the framework for matching the sanity and performance patterns. It starts by expanding the file patterns inside sanity_patterns, and for each file it tries to match all the regex patterns associated with it. As soon as a pattern is matched, it is marked and every of its associated tag values is converted and passed to the action callback along with its corresponding reference value (or just None for sanity

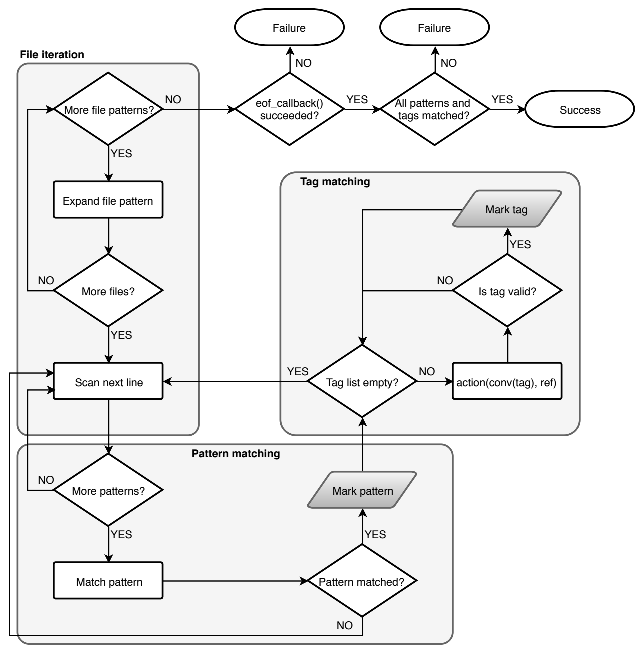

Figure 2: The regression framework's algorithm for scanning sanity and performance patterns in the output of regression tests. The algorithm here is a bit simplified, since it does not show the resolution of performance references.

checking). If the action callback returns `True`, then the tag is marked as matched. The process succeeds if all patterns and all tags of every file have been matched *at least once* and the end-of-file callback, if it has been provided, returns also `True`.

A more complex `sanity_patterns` example is shown below:

```
self.sanity_patterns = {
  '-' : {
    'final␣result:\s+(?P<res>\d+\.?\d*)':
    [
      ('res', float,
       lambda value, **kwargs: \
         standard_threshold(
           value, (1., -1e-5, 1e-5)))
     ],
  }
}
```

This is an excerpt from an actual OpenACC regression test from our suite. The test program computes an axpy product and prints `final result: <number>`. Since the output of different compilers may differ slightly, we need a float comparison. Achieving that with the `sanity_patterns` is quite easy. We tag the matched value with `res` and we then associate an action with this tag. The action is a simple lambda function that checks if the value of the tag is 1.0 ± 10^{-5}. `standard_threshold` is a function provided by the framework that checks if a value is between some tolerance limits. Note that inside the lambda function `value` is already converted to a float number by the conversion callable **`float`**.

1) Tag resolution and reference lookup: The only difference between `sanity_patterns` and `perf_patterns` is that for the latter the framework looks up the tags in a special dictionary that holds reference values and picks up the correct one, whereas for `sanity_patterns`, the reference value is always set to `None`. A representative reference value dictionary is shown below:

```
self.reference = {
    'dom:gpu' : {
        'perf' : (258, None, 0.15)
    },
    'dom:mc' : {
        'perf' : (340, None, 0.15)
    },
    'daint:gpu' : {
        'perf' : (130, None, 0.15)
    },
    'daint:mc' : {
        'perf' : (135, None, 0.15)
    }
}
```

This is a real example of the reference value dictionary for our CP2K [7] regression test. A reference value dictionary consists of sub-dictionaries per system, which in turn map tags to reference values. A reference value can be anything that the action callback function could understand. In this case the reference value required by the `standard_threshold` function is a 3-element tuple containing the actual reference value and lower and upper thresholds expressed in decimal fractions.

Although reference value dictionaries behave like normal Python dictionaries, they have an additional trait that makes them quite flexible: they allow you to define scopes. We call such dictionaries "scoped dictionaries." Currently, they are only used for holding the reference values, but we plan to expand their use in other API variables. A scoped dictionary is a two-level dictionary, where the higher level defines the scope (or namespace) and the lower level holds the actual key/value mappings for a scope. Scopes are defined by the outer level dictionary keys, which have the form `'s1:s2:...'` or `'*'` for denoting the global scope. When you request a key from a scoped dictionary, you can prefix it with a scope, e.g., `'daint:mc:perf'`. The last component of the fully qualified name is always the key to be looked up, here `'perf'`. This key will be first looked up in the deepest scope, and if not found it will be looked up recursively in the parent scopes, until the global scope is reached. If not found even there, a `KeyError` will be raised. The reference value dictionary uses two scopes: the system and the system partition. In the above example, we provide different reference values for different system partitions, but one could provide a single reference value per system. Although the global scope `'*'` seems not to offer anything in a reference value dictionary (what would be the need to have a global reference for any system?), it is quite useful when we need stateful parsing for performance patterns.

2) Stateful parsing of output: It is often the case with the output of a regression test that you cannot judge its outcome by just looking for single occurrences of patterns in the output file. Consider the case that you have to count the lines of the output before deciding about success of the test or not. You could also only care about the n-th occurrence of a pattern, in which case you would call the `standard_threshold` function for checking the performance outcome. In such cases, you need to keep a state of the parsing process and defer the final decision until all the required information is gathered. In a shell script world, you would achieve this by piping the `grep` output to an `awk` program that would take care of the state bookkeeping.

Thanks to the callback mechanism of the REFRAME's output parsing facility, you can define your own custom output parser that would hold the state of the parsing procedure. The framework provides two entry points to any custom parser: (a) the action callback function and (b) the end-of-file callback The action callback must be a function accepting at least two keyword arguments (`value` and `reference`), whereas the eof callback need not have any named keyword argument. Below is a concrete example of how you would count exactly 10 occurrences of the `'Hello,␣World!'` pattern:

```
class Counter:
    def __init__(self, max_patt):
        self.max_patt = max_patt
        self.count = 0

    def match_line(self, value,
                   reference, **kwargs):
        self.count += 1
        return True

    def match_eof(self, **kwargs):
        return self.count == max_patt

parser = Counter(10)
self.sanity_patterns = {
    '-' : {
        '(?P<line>Hello,␣World\!)' : [
            ('line',
             str, parser.match_line)
        ]
    },
    '\e' : parser.match_eof
}
```

Note that it is the eof callback that actually decides on the final outcome of the sanity checking in this case. If you wouldn't need to count the exact number of occurrences, but just a minimum number, you could then omit completely the `match_eof()` function and have the `match_line()` just return **`self`**`.count >= `**`self`**`.max_patt`.

The ability that the framework offers you to leverage the callback mechanism of sanity and performance output checking in order to perform stateful parsing is quite important, since it abstracts away the "boring" details of managing the output files, thus adding to the clarity of the regression test description. Additionally, you may not even need to implement your own parser, since the framework provides a set of predefined parsers for common aggregate and reduction operations. A parser is an object carrying a state and a callback function that will be called if a match is detected. The default callback function is always returning `True`. The parsers offer a simple API that can be used as an action callback in `sanity_patterns` or `perf_patterns`:

- `on(**kwargs)`: Switches on the parser. By default, all parsers are in the "off" state, meaning that their matching functions will always return `False`.
- `off(**kwargs)`: Switches off the parser.
- `match(value, reference, **kwargs)`: This is the function that performs the state update and is to be called when a match is found. The `value` is the value of the match and `reference` is the reference value for the current system (or `None` for sanity checking). It returns `True` if the match is valid. All parsers determine the validity of a match in two stages. First, they check against their state (e.g., *"is this the fifth match?"*) and if this check is successful, then they call their callback function to finally determine the validity of the match. This allows validity checks of the form *"the average performance of the first 100 steps must be within 10% of the reference value for this system."*
- `match_eof(**kwargs)`: This function is to be called as an eof handler. Again, here, the validity of the match is checked in two stages as in the `match()` method. This method clears also parser's state before returning.
- `clear(**kwargs)`: Clears the parser's state.

REFRAME offers the following parsers to the users:

- `StatefulParser`: This parser is very basic, storing only an on/off state. Its `match()` method simply delegates the decision to the parser's callback function, if the parser is on.

- `SingleOccurenceParser` This parser checks for the n^{th} occurence of a pattern and calls its callback function if it's found.
- `CounterParser`: This parser counts the number of occurrences of a pattern and calls its callback function if a certain count is met. The parser can be configured to either count a minimum number of occurrences or an exact number.
- `UniqueOccurrencesParser`: This parser counts the unique occurrences of a pattern and calls its callback function if a certain count is met.
- `MaxParser`: This parser applies its callback function to the maximum value of its matched patterns.
- `MinParser`: This parser applies its callback function to the minimum value of its matched patterns.
- `SumParser`: This parser applies its callback function to the sum of the values of its matched patterns.
- `AverageParser`: This parser applies its callback function to the average of the values of its matched patterns.

The following listing shows an example usage of the `AverageParser` from the actual NAMD check used in CSCS' regression test suite for Piz Daint:

```
self.parser = AverageParser(
    standard_threshold)
self.parser.on()
self.perf_patterns = {
    '-' : {
        'long_pattern␣(?P<days_ns>\S+)␣'
        'days/ns' : [
            ('days_ns', float,
             self.parser.match)
        ],
        '\e' : self.parser.match_eof,
    }
}
```

F. Check tagging

To facilitate the organization of regression tests inside a test suite, ReFrame allows to assign tags to regression tests. You can later select specific tests to run based on their tags.

```
self.tags = { 'production', 'gpu' }
```

It is also possible to associate a list of persons that are responsible for maintaining a regression test. This list will be printed in case of a test failure.

```
self.maintainers = [ 'bob@a.com',
                     'john@a.com' ]
```

IV. Running your tests

Before going into any details about the frameworks front-end and command-line interface, the simplest way to invoke ReFrame is the following:

```
./bin/reframe -c /path/to/checks -R --run
```

This will search recursively for test files in `/path/to/checks` and will start running them on the current system.

The ReFrame's front-end goes through three phases: (a) test discovery, (b) test selection and (c) action. In the following, we will elaborate on these phases and the key command-line options controlling them. For a complete reference, the user is referred to the online documentation.

A. Regression test discovery

When ReFrame is invoked, it tries to locate regression checks in a predefined path. The default path is `<install_prefix>/checks/` and it is searched recursively. As mentioned previously, in ReFrame regression tests are essentially Python source files that provide the special `_get_checks()` function, which returns the actual regression test instances. The front-end loads the Python source files and tries to call this special function for each of them; if this function cannot be found, the source file will be ignored. At the end of this phase the front-end will have instantiated all the checks found in the path. The default check search path can be overriden by specifying the `-c` or `--checkpath` options. Note that by default ReFrame does not descend recursively into directories specified with the `-c` option, so it is necessary to explicitly request that by using the `-R` or `--recurse` options, as we did in the example above. Multiple `-c` options can also be chained to construct a custom search path. Finally, `-c` option accepts also regular files. This is very useful when implementing new regression checks, since it allows to load and run only the check of interest:

```
./bin/reframe -c /path/to/my/check.py --run
```

B. Regression test selection

After the regression tests are discovered and loaded, a finer selection can be applied in this phase. Currently, regression tests can be selected in one of the following ways:

- By programming environment using the `-p` or `--prgenv` option,
- by name using the `-n` or `--name` options and
- by tag using the `-t` or `--tag` options

All these options can be combined and chained together. For example, the following will list all tests supporting `PrgEnv-gnu` and are tagged with `foo` and `bar`:

```
./bin/reframe -p PrgEnv-gnu -t foo -t bar -l
```

Currently, the selection mechanism is not 100% flexible, since the selection criteria cannot be negated or OR'ed together. However, it has already proved useful enough in our use case, since it allowed us to categorize our tests in more than one dimensions.

C. Regression test actions

At this last phase the front-end takes an action on the previously loaded and selected regression tests. There are two available actions supported by ReFrame: (a) listing of the selected tests and (b) execution of the selected tests. An action is always required to be specified by the user, otherwise ReFrame issues an error.

The list action is determined by the `-l` or `--list` options. It will simply produce a listing of the previously selected checks showing the test name, its description, its tags and its list of maintainers.

The execution action is determined by the `-r` or `--run` options. This will run all the selected regression tests with each test going through the regression pipeline described in Section II. There are several options controlling the execution of the regression test that are beyond the scope of this paper. The reader is referred to the online documentation of ReFrame for more information.

D. Organizing the regression tests

ReFrame allows the users to organize their regression tests in any way that is the most convenient for their needs. The only soft requirement imposed by the framework is that a `src/` folder should be present at the same level as the test's source file[2].

Users can group together related tests in a common directory sharing the same `src/` folder as in the `foobar` family of tests in the following example. This sharing can eliminate duplication at the level of regression test resources, which can prove beneficial in maintaining a large regression test suite. For run-only regression tests the `src/` directory can be empty or contain other resources relative to the test, e.g., input files. The following directory structure visualizes the organization concepts described:

```
mychecks/
  compile/
    helloworld/
      helloworld.py
      src/
        helloworld.c
    foobar/
      bar.py
      foo.py
      src/
        bar.c
        foo.c
  apps/
    prog1/
      src/
      prog1.py
    prog2/
      src/
        input.txt
      prog2.py
```

[2]This is not a hard requirement, it is just the default behavior. The users may override this by redefining the **`self`**`.sourcesdir` variable in their tests.

E. Configuring a new site

ReFrame provides an easy and flexible way to configure new systems and new programming environments. As soon as you have configured a new system with its programming environments, adapting an existing regression test could be as easy as just adding the system's name in the `valid_systems` list and its associated programming environments in the `valid_prog_environs` list. Due to space limitations, we will not provide here a detailed description of the ReFrame's configuration (the reader is referred to the online documentation), but we are going to highlight the capabilities of the configuration mechanism.

From the regression's point of view each system consists of a set of logical partitions. These partitions need not necessarily correspond to real scheduler partitions. For example, Daint comprises three logical partitions: the login nodes (named `login`), the hybrid nodes (named `gpu`) and the multicore nodes (named `mc`), but these do not correspond to actual Slurm partitions. Logical partitions may even use different job schedulers. An obvious example is the `login` partition that uses the `local` scheduler, since regression tests for login nodes are meant to run locally. A logical partition may also be associated with a job scheduler option that enables access to it. For example, on Piz Daint the hybrid and multicore nodes are obtained through Slurm constraints using the `--constraint` option. On other systems the logical partitions may be mapped 1–1 to real scheduler partitions, in which case the `--partition` option of Slurm would be used. You can associate also modules and environment variables with logical partitions. These modules will always be loaded and environment variables will be set before a regression test runs on that partition. For example, on Piz Daint, you have to load a specific module on each partition, which makes available an optimized software stack for the nodes of the partition. Finally, a partition is associated with a list of (programming) environments to test, e.g., `PrgEnv-cray`, `PrgEnv-gnu` etc. These are defined inside a scoped dictionary (see Section III-E1) keyed on the system or system partition. This allows you to define programming environments for a specific system only or override environment definitions. For example, on one of our systems we needed to override the default definition of `PrgEnv-gnu` to use `mpicc`, `mpicxx` and `mpif90` as the compiler wrappers. The nice trait with this is that the regression tests supporting `PrgEnv-gnu` do not need to change, even if the compiler wrappers change.

F. *Regression test failures and framework errors*

Each regression test is run in a sandbox that isolates it from the framework's environment. More specifically, a regression test runs within its own environment and all of its resources (sources, input/output files etc.) are copied to a temporary stage directory. If the regression test is successful, its stage directory is wiped out and only a set of interesting files are archived for a future reference. If the regression test fails, the stage directory remains untouched, so that the user can manually inspect and reproduce the error. Inside the stage directory the user can find the exact (job) shell script that was automatically generated for running the regression test. To reproduce the error the user can simply submit or run the generated script, since all the necessary information of the test is inside this script (job scheduler options, test environment setup etc.) This is quite useful for debugging regression tests, since you need not go through the whole regression pipeline every time.

A regression test may not only be due to a problem in the system or its configuration, but it could also fail due to a programming error in its own code. Although such errors are usually detected early enough during the initialization phase of the test, there are still cases that a test may fail due to a programming error later on, while the regression suite is running. In such cases, we don't want the whole regression suite to abort just because of a programming error in a regression test. For that reason, almost all errors raised during the execution of a regression test are treated as non-fatal: the test is marked as failed and the framework continues with the execution of the next test.

V. Use cases

The ReFrame framework has been put into production with the upgrade of the Piz Daint system in early December 2016. We have two large sets of regression tests: (a) production tests and (b) maintenance tests. We use tags (see Section III-F) to mark these categories and a regression test may belong to both of them. Production tests are run daily to monitor the sanity of the system and its performance. All performance tests log their performance values and we use Grafana [8] to graphically monitor the performance of certain applications and benchmarks over time.

The set of production regression tests comprises 104 individual tests. Some of them are eligible to run on both the multicore and hybrid partitions, whereas others are meant to run only on the login nodes. Depending on the test, multiple programming environments might be tried. In total, we run 437 test cases from 157 regression tests on all the system partitions. Table II summarizes the production regression tests.

The set of maintenance regression tests is much more limited, since we want to decrease the downtime of the system. The regression suite runs at the beginning of the maintenance session and just before returning the machine to the users, so that we can ensure that the user experience is at least at the level before the system was taken down. The maintenance set of tests comprises application performance tests, some GPU library performance checks, Slurm checks and some POSIX filesystem checks. The total runtime of the maintenance regression suite for all the partitions is approximately 20 minutes.

Table II: Regression tests running on Piz Daint.

Type	Partition	#Tests	#Test cases	Total
Production	Login	15	24	437
	Multicore	61	190	
	Hybrid	81	223	
Maintenance	Login	2	2	38
	Multicore	7	7	
	Hybrid	19	19	

We are now porting the regression suite to the MeteoSwiss production system Piz Kesch. Configuring this system for ReFrame was trivial: we have just added a new system entry in the framework's configuration file describing the different partitions and redefined the `PrgEnv-gnu` environment to use different compiler wrappers. Porting the regression tests of interest is also a straightforward process. In most of the cases, adding just the corresponding system partitions to the `valid_systems` variables and adjusting accordingly the `valid_prog_environs` is enough.

For demonstration purposes we have also ported the framework to Cray's TDS Swan cluster. Of course, since ReFrame does not offer a Torque backend yet, we could not run any submission tests, but the rest of the framework's unit tests have all passed successfully. Here is the configuration entry for the Swan's login nodes:

```
'swan' : {
    'descr' : 'Cray␣Swan␣TDS',
    'hostnames' : [ 'swan' ],
    'partitions' : {
        'login' : {
            'scheduler' : 'local',
            'environs'  : [
                'PrgEnv-cray',
                'PrgEnv-gnu',
                'PrgEnv-intel',
                'PrgEnv-pgi'
            ],
            'descr' : 'Login␣nodes'
        }
    }
}
```

A. *Overhead of maintaining the regression tests*

To better understand the difference in maintenance burden of the ReFrame framework compared to our previous re-

Table III: Source code line count comparison of REFRAME framework vs. CSCS' old solution.

Component	Old framework	REFRAME
Core	N/A	3660 loc
Front-end	1038 loc	958 loc
Regression tests	14635 loc	2985 loc
Avg. regression file size	179 loc	93 loc
Avg. regression test size	179 loc	25 loc

gression suite, Tab. III summarizes the total amount of code of different components. It is obvious that the difference in the amount of code needed to be written for the regression tests is tremendous. From a total of almost 15K loc of the former solution, the regression tests code has gone down to less than 3K loc, covering even more aspects of system behavior. Averaged over 32 total files implementing our new regression tests, it gives an average of 93 loc per regression test file. However, as discussed in previous sections, the framework offers you the possibility to create factories of tests, so that a single file could basically generate a multitude of actual regression tests. Given an absolute total of 122 tests generated, the effective number of loc per test is only 25!

Separating the logical description of a regression test from all the unnecessary implementation contributes significantly in the ease of writing and maintaining new regression tests. You can see from Tab. III how the implementation details were spread out across all the regression tests in the past. The REFRAME framework really focuses on abstracting away all the gory details from the regression test description, hence letting the user to concentrate solely on the logic of his test.

VI. CONCLUSIONS & FUTURE DIRECTIONS

In this paper we have presented REFRAME, a regression framework for easily writing portable regression tests for HPC systems. REFRAME hides the complexity of dealing with the system details and lets the user focus on the logic of his regression tests. The user writes his regression test logic in Python and the framework translates this to a concrete regression pipeline that his test goes through. The user is given the flexibility to intervene into the pipeline stages and modify his test's behavior. Additionally, writing regression tests in a modern programming language, like Python, offers significant flexibility and capabilities in structuring and organizing them correctly, eliminating code duplication and hence minimizing maintenance costs. REFRAME is in production at CSCS and is used daily to check the health of our HPC systems, as well as before and after every maintenance. It has been working reliably since the last upgrade of Piz Daint on December 2016 and we are expanding its use on more systems at our site and with more regression tests.

Our team at CSCS is actively developing REFRAME using modern software engineering techniques to ensure the highest quality and stability of the framework. We have several features in our backlog to implement in future releases, the most prominent ones being the following:

- Support for proper logging of the framework's activities, since currently everything is printed out, leading to a rather verbose output.
- Support for asynchronous execution of regression tests.
- More flexible way for differentiating the regression test behavior per system and per programming environment, minimizing the need for overriding methods.
- Support for other job submission backends apart from Slurm.
- Improvements of the internal interfaces to facilitate the expansion of the framework with different backends.

VII. ACKNOWLEDGEMENTS

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

REFERENCES

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

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

[3] A. B. Yoo, M. A. Jette, and M. Grondona, *SLURM: Simple Linux Utility for Resource Management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. [Online]. Available: http://dx.doi.org/10.1007/10968987_3

[4] *Cray Programming Environment User's Guide*, Cray Inc., Jun. 2014. [Online]. Available: http://docs.cray.com/books/S-2529-116//S-2529-116.pdf

[5] *Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes*, Virtual Institute – High Productivity Supercomputing. [Online]. Available: http://www.vi-hps.org/projects/score-p/

[6] *Allinea DDT: The debugger for C, C++ and Fortran threaded and parallel code*, Allinea Software. [Online]. Available: https://www.allinea.com/products/ddt

[7] J. Hutter, M. Iannuzzi, F. Schiffmann, and J. VandeVondele, "CP2K: atomistic simulations of condensed matter systems," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 4, no. 1, pp. 15–25, 2014.

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