Towards Acceptance Testing at the Exascale Frontier

Verónica G. Vergara Larrea, Michael J. Brim, Arnold Tharrington,
Reuben Budiardja, and Wayne Joubert
National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN, USA
Email: {vergaravg,brimmj,arnoldt,reubendb,joubert}@ornl.gov

***Abstract*—At the 2007 Cray User Group meeting, the Oak Ridge Leadership Computing Facility (OLCF) introduced the OLCF Test Harness (OTH), a framework[1] used for acceptance testing of the Jaguar supercomputer[2]. Since then, the OTH framework has evolved to version 2.0 which adds new features and streamlines usability. The OTH is the key piece of software used to orchestrate acceptance testing for all OLCF computational resources before they are deployed for production use, including our leadership class high performance computing (HPC) systems. The OTH framework is written in Python and is publicly available[3].**

In this paper, we first describe the requirements, design, and structure of the OTH. Then, we present specific improvements developed to support acceptance testing of the OLCF's Summit system[4]. We will also showcase new OTH features that have been added to streamline the acceptance test process as well as the motivation behind those changes. As part of this work, we also evaluated different workflow tools in order to determine whether these tools could complement the OTH in two key areas: automation and reporting. The advantages and disadvantages identified with each tool will be discussed. Lastly, we summarize the challenges and lessons learned collected from using the OTH for the acceptance of the last three flagship systems at the OLCF. These may be useful for other HPC centers developing their own testing frameworks or those interested in using the OTH.

***Keywords*-automated testing framework, high performance computing, workflows**

I. Introduction

In 2007 at the Cray User Group meeting, the Oak Ridge Leadership Computing Facility (OLCF) introduced the framework used to complete acceptance testing of the Jaguar supercomputer [2]. The framework, now known as the OLCF Test Harness (OTH), was used for acceptance testing of each Jaguar upgrade which brought the system from 25 teraflops in 2005 to more than 1 petaflop in 2008.

Due to the scale and complexity of the system, the OLCF conducted a rigorous acceptance test with each upgrade. The Jaguar acceptance test included two main stages: hardware acceptance (HW) and final integration testing (FI). Table I summarizes the stages and their elements. The first stage consisted of hardware diagnostics performed by Cray personnel to ensure that each individual component (e.g., processors, memory, interconnect) met the required specifications. The second stage included three elements: functionality testing (FT), performance testing (PT), and stability testing (ST) [1]. The FT element ensures that individual system software components such as the scheduler, compilers, and libraries are working correctly. The PT element ensures that applications can be successfully executed at scale and are able to achieve the required performance targets. The ST element ensures that the system can support execution of a diverse workload continuously for an extended period of time.

To successfully execute all acceptance test elements, the OLCF needed a testing framework that enables staff to simulate a realistic diverse workload to evaluate whether a particular system was ready for production. Several key features were identified as requirements for the framework:

1) Workloads should be launched and executed in the same way a user would execute them.
2) Workloads should run continuously on the system without requiring manual actions by staff.
3) Individual executions of an application must be uniquely identified and tracked.
4) The level of effort to include a new application had to be low.
5) In-house expertise is required in order to quickly identify and fix potential issues during critical periods of acceptance testing.

The OTH was designed with these requirements in mind and was able to closely mimic the activities performed by real users of HPC systems. These activities include building a scientific application, generating a batch script, submitting the batch script to the job scheduler, and verifying application results after the job completes. The OTH was also able simulate execution of realistic production workloads

Table I
JAGUAR ACCEPTANCE TEST STAGES

Acceptance Test Stage		Description
Jaguar HW		Hardware diagnostics of individual components
Jaguar FI	FT	Functionality of system software (e.g. scheduler, compilers)
	PT	Performance and scalability of applications on the system
	ST	Stability of the system under a realistic workload

that fully occupy an HPC system for an extended period of time using a diverse set of applications.

Since Jaguar's acceptance testing, the OLCF has continued using this methodology to conduct acceptance testing of its flagship systems including Titan and Summit. In addition, the tests developed for acceptance of each system are used during production to verify correct functionality and performance after a software upgrade. As a result, the OTH was modified to adapt to the expanded use cases, to provide more flexibility, and to minimize effort required to add and monitor tests. The new version of the testing framework, which includes a significant restructuring of the original framework as well as additional features, is known as version 2.0 of the OTH. The OTH leverages standard configuration formats, provides logging capabilities, and simplifies the steps required to add a new application over its predecessor.

In this paper, we first describe the original design of the OTH followed by the modifications that resulted in the latest version of the framework. Then, we discuss specific features that were added to support acceptance testing of Summit and those that are being added for acceptance testing of Frontier. We then discuss our efforts towards automating the OTH and the workflow tools we have evaluated to achieve that goal. Finally, we present the lessons we have learned from building and upgrading the OTH and briefly discuss future development work that is planned for the framework.

II. OLCF TEST HARNESS DESIGN

In this section we describe the original OTH framework design along with a description of how the framework runs tests.

A. Overview of the OTH execution

The OTH has two independent components that are prerequisites to successfully launch a set of tests: (1) the OTH driver (i.e., `runtests.py`) and (2) a repository of tests.

The OTH workflow shown in Figure 1 is kicked off by the OTH driver.

The first step is to initialize the OTH runtime environment (RTE). This is accomplished by loading a single Lmod [5] modulefile. The initialization of the OTH RTE sets several environment variables used by the harness, including:

- `RGT_MACHINE_NAME`: Used as a path name component for many OTH environment variables.
- `RGT_PATH_TO_SSPACE`: Defines the working scratch space directory to build applications and run jobs.
- `RGT_PATH_TO_REPO`: Defines the URL of the upstream version control repository for software application tests.

Secondly, the user creates an OTH input file (e.g., `rgt.input`) which informs the OTH which application tests to run (i.e., `Test` keyword), the path to the application tests staging area (i.e., `Path_to_tests` keyword), and the harness tasks to perform (i.e., `Harness_task` keyword). Upon launch of the OTH, it reads the input file and performs the listed tasks for each application test in sequence (see Figure 2). The OTH can perform three tasks: `check_out_tests`, `start_tests`, and `stop_tests`.

The task `check_out_tests` does a single initial `svn checkout` of an application test to the staging area.

The task `start_tests` initiates a chained launch of test instances of the checked out application test. Each test instance across all application tests is assigned a unique identification number (UID). It is important to note that that for the `start_tests` task, the initial run process is controlled by the login terminal, but subsequent instances of the same application test are launched from other execution hosts. For that reason, the series of tests cannot be killed by a `SIGHUP` from the shell of the initial login terminal.

The task `stop_tests` stops the chained set of test instances of a given application test by creating a hidden file named `.kill_test` in the test's `Scripts` directory which informs the OTH to cease the repetition of this test.

B. Implementing Tests

As depicted in Figure 3, a software application test (`app1`) logically contains a `Source` directory and a set of test directories (`test1`, `test2`).

The `Source` directory contains the application's source and various helper scripts for performing the tests. For example, one needs scripts that will build the application, verify results, and set the tests' runtime environments. There are no restrictions on the internal layout of the `Source` directory. The only requirement is that the helper scripts and the application source be located within the `Source` directory.

A test directory's (e.g., `test1`) only requirement is that it contains a directory named `Scripts`. The `Scripts`

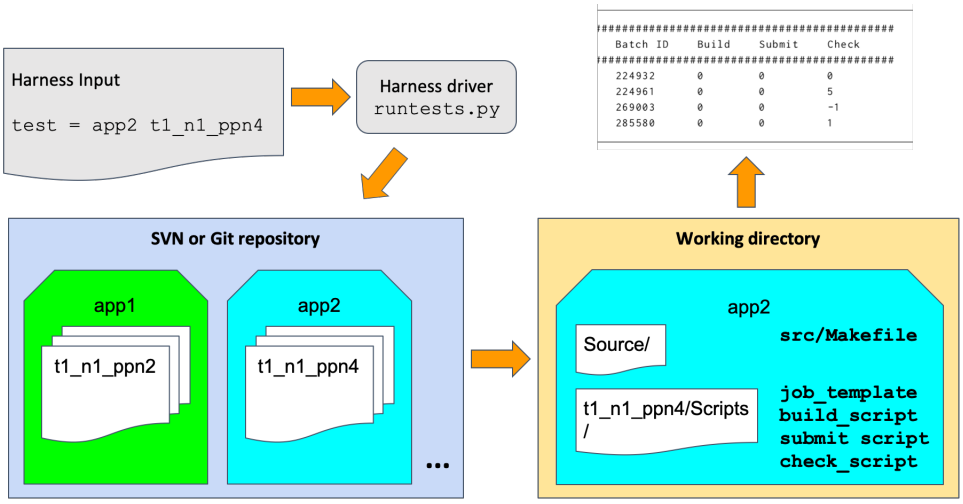

Figure 1. OTH v1.0 workflow

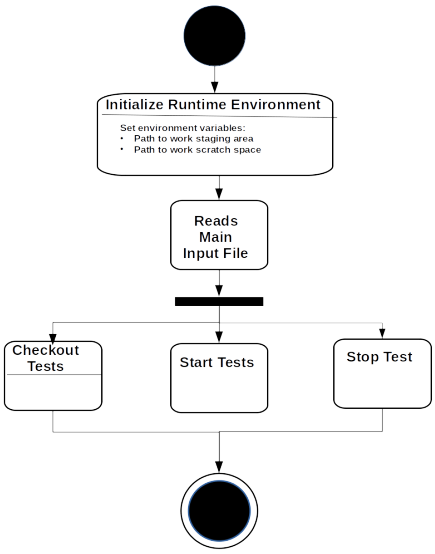

Figure 2. Overview of OTH v1.0 execution.

directory at a minimum must contain three scripts: `build_executable.x`, `submit_executable.x`, and `check_executable.x`.

For each test instance, `build_executable.x` is the first script called and is responsible for building the binaries for a given application. Its API is:

```
build_executable.x -i <UID> -p <scratch space>
```

where the `-p` argument is the path to the working scratch space defined by `RGT_PATH_TO_SSPACE`, and the `-i` argument is the unique identification number (UID) given to this test instance.

The next script called is `submit_executable.x`, and

```
# app1 application with 2 tests
<PT>/app1/Source/
         /test1/Scripts/
               /Run_Archive/<UID>/
               /Status/rgt_status.txt
         /test2/Scripts/
```

Figure 3. OTH directory structure example (`PT` represents the directory specified by `Path_to_tests`)

its intended purpose is to create a batch script and submit it to the scheduler. Its API is:

```
submit_executable.x -i <UID> -p <scratch space>
```

The only requirement for the submission script is that the last lines of the generated batch submission script be:

```
# Resubmission section.
case $RGT_RESUBMIT in
    0 )
       echo 'No resubmit';;
    1 )
       test_harness_driver.py -r;;
esac
```

in order to have the test restart itself.

The last script needed is `check_executable.x` and its API is:

```
check_executable.x -i <UID> -p <scratch space>
```

where the `-p` argument is the path to directory containing the data to verify and the `-i` argument is the UID. This call is typically placed in the generated batch submission script just before the resubmission section.

The OTH requires that all the scripts `build_executable.x`, `submit_executable.x`,

and check_executable.x be called from the tests Scripts directory.

C. Viewing Test Results

Once test instances have launched, the OTH automatically creates per-test directories named Run_Archive and Status (see Figure 3).

The OTH will create a file called rgt_status.txt within the Status directory that contains a row per each individual test instance launched (see Figure 4). The row contains information for when the given instance was launched (Start Time), the UID (Unique ID), the job ID for the submitted job (Batch ID), and three columns to display the status of the build, submit, and check scripts (Build, Submit, Check). As a convention, the OTH will use 0 to denote a step was successful. Any failures will result in a non-zero status.

III. Harness Evolution

In this section we describe OTH's evolution to support acceptance of larger, more complex systems. The features that have been introduced since v1.0 simplify test creation and provide a uniform standard that application developers can follow.

A. Upgrading the OTH for Summit's acceptance test

Starting with version 2.0, the OTH now relies on a *machine*-centric design by introducing the machine_types abstraction to centralize various pieces of the OTH that prior to this version existed only at the test level. The base_machine definition includes information about the scheduler it uses, the application launcher, and the physical characteristics of the system. Each instance of a derived machine_type must also specify both a scheduler and a jobLauncher. As shown in Figure 5, this new design has a base_machine at the top of the class hierarchy which ensures that derived machine_types implement the build, submit, and check steps that the OTH must complete per test instance. By centralizing these steps, we are able to reduce the level of effort an application developer needs to commit in order to provide a test. At the time of this writing, three types of *machines* have been implemented based on systems available at the OLCF: cray_xk7, rhel_x86, and ibm_power.

As shown in Table II, there were several additional key changes that were introduced in OTH v2.0.

We moved away from providing support for Subversion repositories in favor of Git repositories. Initially, a single repository was being used to host all applications for a given system. For that reason, in this version we also added support for sparse-checkout. This allowed OTH users to checkout only the applications listed in the OTH input rather than the full list of applications available in the repository. We also introduced the concept of a per test configuration input file (i.e., rgt_test_input.txt) that was used to determine whether a given application was using OTH v1.0 or OTH v2.0. This input file used key-value pairs to describe the configuration settings for a specific test (e.g., job name, number of nodes, wall time, etc.).

In order to support Summit's acceptance, we also updated the OTH to include:

- Support for IBM's Load Sharing Facility (LSF) job scheduler and Job Step Manager (JSM) application launcher.
- A mechanism for application developers to leverage built-in functionality for test builds, batch script generation and submission, and result checking. This version still provided support for custom user-provided scripts to maintain backwards compatibility with OTH v1.0 tests.
- Event logging to mark the start and end of each step (e.g., build start, build end).
- A reporting module to collect test-specific metrics that we would like to track.

B. Upgrading the OTH for Frontier's acceptance test

In preparation for Frontier, we have completed a significant restructuring of the OTH code to improve readability and maintainability. The code cleanup also involved removal of unused functions and deprecating rarely used features. We had two main objectives that motivated the code restructuring: (1) improve the ease of developing new tests, and (2) provide more flexible options for running selected test steps.

Earlier versions of the OTH required some harness-specific commands to be executed within the test's scripts for correct operation of the harness. To ease test development, our goal is that each test script should be self-contained such that it is able to execute successfully independent of the OTH. For example, a test's build script is fully responsible for setting up the build environment (e.g., by loading necessary compiler and library modules) and executing the necessary configuration and build commands. The OTH can then cleanly wrap the execution of the test scripts to capture output and log important information such as event timestamps. There are cases, however, when a test script could benefit from information flowing from the harness to the script. In these cases, we modify the environment used by the script to pass information such as harness working directories. This makes the information available to the scripts, but does not require its use.

Our second objective was to improve the flexibility of running tests. Originally, there were only two modes of operation a user could choose for running a test - run all three test steps once, or run continuously. Often during initial test development and debugging, we would encounter a problem with one particular step (e.g., build) and desired a method to execute just that step. Support for individual step execution was added to test_harness_driver.py,

```
################################################################################
#Start Time                  Unique ID           Batch ID  Build  Submit  Check
################################################################################
2020-06-25T09:08:19.050588  1593090499.0219295  86987     0      0       0
2020-08-18T22:32:09.658387  1597804329.6306179  88864     0      0       1
2020-08-24T15:02:13.520704  1598295733.4882843  88979     0      0       0
```

Figure 4. OTH status file example (`rgt_status.txt`).

Table II
OTH MAJOR CHANGES PER VERSION

Target System	OTH Version	Version Control System	Repository Type	Test Input	Test Input Format
Jaguar, Titan	1.0	SVN	Per-machine	Not supported	N/A
Summit	2.0	SVN, Git	Per-machine	Optional	key-value
Frontier	2.1 (beta)	Git	Per-application	Required	INI

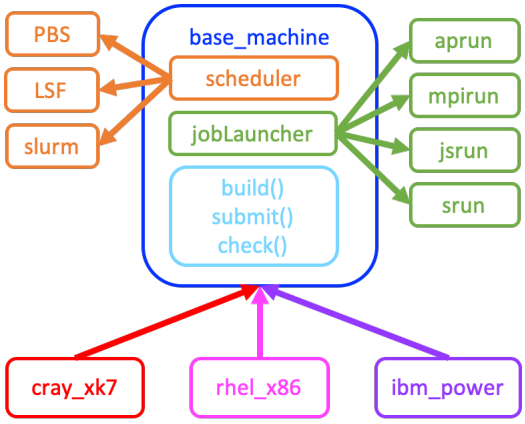

Figure 5. OTH v2.0 - Machine types abstraction

which is the driver program executed for each test in the input file passed to `runtests.py`. The ability to execute individual steps also paved the way for integration into existing workflow software, as described later in IV-A.

In addition, we have added the following features to the OTH that will be available in the next release:

- Support for SchedMD's Slurm job scheduler and application launcher.
- Error checking to prevent job submissions after a failed build.
- Unit testing for the full OTH workflow to ensure functionality of individual components of the framework.
- A new format for the test configuration input file that uses the INI standard (i.e., `rgt_test_input.ini`).
- Support for per-application repositories instead of a single monolithic per-machine repository (see Table II).
- A machine configuration file that uses the INI standard. This file defines the values used to instantiate the appropriate machine type as described above, further reducing the information OTH users need to provide.

IV. HARNESS AUTOMATION

In this section we describe two related efforts that have allowed us to automate execution of the OTH to meet two separate but equally important goals. The first goal is to automate harness execution in an easily trackable way. The second is to automate execution in order to verify correct functionality of the OTH itself.

A. *FireWorks Integration*

FireWorks [6] is a general purpose workflow framework that is designed for use within HPC environments. Its workflows are described using simple YAML specifications, or programmatically using Python APIs. Workflow state is maintained within a centralized MongoDB database called the LaunchPad. Tasks called "fireworks" are fetched from the database and executed by workflow agents called "Fireworkers". FireWorks has built-in support for interacting with compute system resource managers such as PBS, LSF, and SLURM to run workflow tasks within batch jobs. Additionally, FireWorks provides an interactive web-based GUI interface, "webgui", for monitoring and querying workflow activity.

At its core, the OTH is also a simple workflow engine. Each application test has a predefined workflow consisting of three tasks: (1) *build* the application, (2) *run* the application within a batch job, and (3) *check* the application results. The workflow task commands are specified in the test input configuration file. Workflow state is tracked using unique files for each instance of an application test. The OTH directly executes the build and check tasks, and interacts

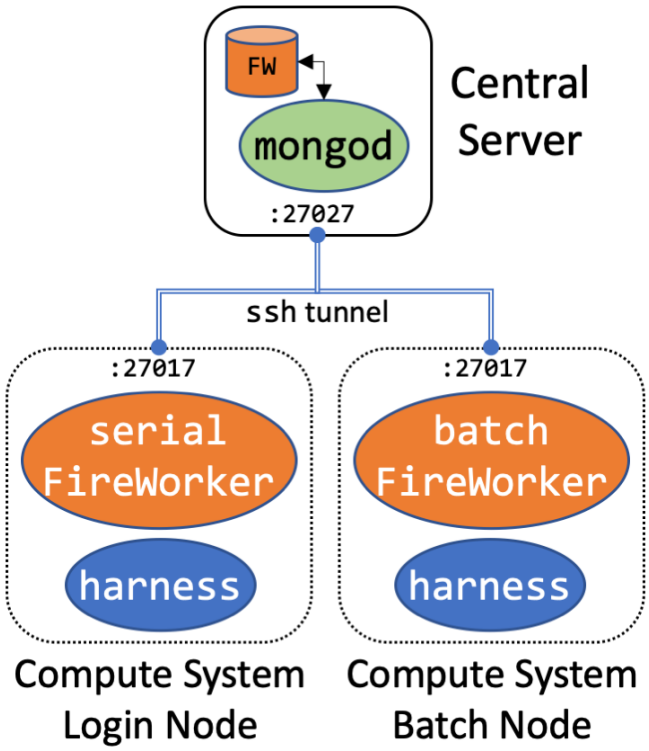

Figure 6. OLCF Test Harness FireWorks Deployment Architecture

with resource managers to submit batch jobs that execute the run tasks.

Given the similarity between the capabilities of FireWorks and the OTH, we decided to investigate the benefits of integrating the two systems. This integration proved to be fairly straightforward due to the recent improvements in the OTH that allow execution of individual task steps. In roughly 50 lines of Python code, we define the FireWorks workflow for each application test and submit it to the LaunchPad. The workflow consists of three fireworks, each containing a single `ScriptTask` that executes our OTH driver in the appropriate mode. Since FireWorks manages job submission, we added a new `run` mode to the OTH driver that simply executes the batch script generated by the OTH, rather than submitting it to the resource manager. We use the existing OTH `Run_Archive` directory created for each test instance as the launch directory for all the fireworks so that we capture the FireWorks temporary files. We use custom batch job templates to ensure that OTH test configuration settings are respected.

As shown in Figure 6, we place the centralized FireWorks LaunchPad database on a host accessible to all compute systems. For each compute system, we configure two Fireworkers, one for serial tasks (build and check), and one for batch job tasks (run). The workflow assigns categories to each firework (e.g., "`<system>-batch`" or "`<system>-build`") so that the appropriate Fireworker is used for execution. The Fireworkers run continuously and poll the LaunchPad for new tasks. Using this architecture, we can use the FireWorks webgui to monitor OTH workflows submitted from any compute system as shown in Figure 7. The webgui also permits filtering to workflows from a specific system by issuing a MongoDB query matching one of our assigned categories.

In addition to the benefits of having a central database and improved monitoring, the FireWorks command-line tools have proven useful for dynamic resolution of problematic workflows. LaunchPad queries simplify the task of identifying failed tasks; previously this involved examining the individual status files for each application test. Further, the ability to relaunch specific fireworks has been useful for situations where our batch systems have temporary glitches, or for debugging scripting errors during test development.

B. GitLab Integration

The OTH code is currently hosted in an internal GitLab [7] instance. Starting with v2.0, we have begun leveraging GitLab's continuous integration/continuous delivery (CI/CD) features to integrate automatic unit testing. A GitLab runner [8] has been deployed in each target system and each one is responsible to ensure that a simple OTH workflow is successful upon merging a request into the main development branch. Now that we support more systems, manual verification on each one can be time consuming. Adding automated unit testing to our development pipeline has significantly reduced the likelihood of introducing a bug during development that could prevent the OTH from running on one of our supported systems.

V. Lessons Learned

The OTH has evolved significantly since it was first introduced for Jaguar's acceptance testing. Executing acceptance testing of each new system revealed potential gaps that could be improved in the OTH.

For example, during Summit's acceptance testing, the OLCF executed a wide range of benchmarks, mini-applications, and real codes [4] to ensure that individual components as well as the system at scale could support the workloads that OLCF users would bring to the facility. As a result, the Git repository that held all Summit acceptance testing applications became excessively large. Having a single repository also introduced false dependencies between updates to different applications. To address this issue, the OTH can now point to individual application repositories rather than a per-machine repository.

From v1.0 of the OTH, we learned that while having the flexibility to create your own per-test submit, build, and check scripts to interface with the OTH driver can be advantageous, it also increases the maintenance overhead for the OTH development team. Although the OTH development

Figure 7. Monitoring OLCF Test Harness Workflows on Two Systems - Peak and Lyra

team is not responsible for maintaining those interfaces, during acceptance testing they are required to conduct "live" troubleshooting. Having a consistent mechanism to build applications, interact with the scheduler, and verify results simplifies the debugging process and completely decouples the application scripts from the OTH interfaces. Centralizing those interfaces also allows the application developer to focus only on providing scripts that they should already have as part of their regular workflow.

A. *Software Best Practices*

Custom Code vs. Standard Libraries. Starting with Summit acceptance, we had added support for using a simple key-value test input file to simplify keyword substitution in batch job template scripts. However, this custom file format required development and maintenance of a corresponding parser and substitutions dictionary. For v2.1, we decided to switch to a standardized format instead (i.e., INI). This allowed us to leverage standard Python3 libraries (e.g., configparser) rather than custom code.

Code Modularity. A significant portion of the software engineering for v2.1 targeted improved code modularity. For instance, two common operations within the harness were to determine the application and test names based on the full path to the test's `Scripts` directory and to construct paths to other test-specific directories used by the harness. Previously, these operations were implemented in each place they were required, in a variety of mostly equivalent ways. We now use a single Python class that maintains the canonical file system layout for an application test. This class defines constants for prescribed file and directory names and provides utility methods for retrieving the absolute paths to various test directories used by the harness. When we wish to introduce a new file or reorganize the file system layout, changes can be made in a single module rather than the previous tedious process of search and replace.

VI. Related Work

While several open source testing frameworks exist, few are designed specifically for HPC environments. At the time of this writing, ReFrame [9] and Pavilion [10] are two of the most popular open source HPC testing frameworks used at HPC centers.

ReFrame is an open source Python regression testing framework for HPC systems developed at the Swiss National Supercomputer Center (CSCS). ReFrame is regularly used at large-scale facilities such as CSCS, NERSC, and KAUST to conduct regression testing. Tests in ReFrame are represented by decorated Python classes. Each decorator tells ReFrame about a specific characteristic of the test. Alongside the framework, CSCS makes available their own list of tests [11] available to ReFrame users. The list includes 15 scientific applications and a large number of benchmarks and kernels used to conduct regression testing on Piz Daint. Besides being widely used, ReFrame provides detailed documentation and an archive of tutorials on how to use the framework. While ReFrame is indeed a very powerful, the application developer must be familiar with ReFrame in order to add a class for new application. We consider this as a drawback as it will increase the level of effort required from an application developer which could decrease the number of application and tests that we have available to us for acceptance testing.

Pavilion [12] is a Python HPC testing framework developed at Los Alamos National Laboratory. Pavilion is used by LLNL and LANL for acceptance and regression testing. Similarly to the OTH, Pavilion has been significantly rewritten and has been relaunched as Pavilion 2 [10]. The latest version shares several design goals with the OTH and also relies in a per-host configuration file. In addition, Pavilion 2 provides a mechanism to add mode configuration files and plug-in support. One main difference with OTH is that Pavilion 2 uses YAML files for the various configuration pieces. To write a new test, Pavilion 2 requires the source for the application and a YAML file that defines how the test should be built and executed. The test YAML file includes sections to load environment modules, specify compilers, and include run commands. Unlike its predecessor, Pavilion 2 uses the information in the test YAML to generate a build script and a run script. In addition, Pavilion 2 also supports automatic replacement of job launcher commands which can be beneficial to reduce the burden on application developers. Pavilion 2 is a powerful testing framework and has significant overlap with the functionality already provided by the OTH. We also have in-house expertise at the OLCF on the internals of the OTH which is a requirement of any framework used for acceptance testing. In addition, after several successful acceptance tests for OLCF resources, we have built a large collection of OTH tests which would require a significant porting effort.

VII. Future Work

There are several features that are in the roadmap for the OTH that we plan to add in the near future. Currently, the application developer still needs to provide a batch submission template with keywords that are replaced at launch time. This could potentially also be abstracted to further reduce the files that the application developer creates when adding a test.

Automating the OTH with FireWorks will allow us to have fine-grained control of which tests are launched. We plan to continue improving the integration to leverage the capabilities that the workflow tool provides.

We also would like to provide a robust and flexible monitoring system for OTH results. Building on work done in [13], we are working on deploying a new monitoring system that will provide OLCF staff with an easy way to store and view results instead of relying on the flat status file.

While the OTH has been open sourced, we are in the process of open sourcing the latest version that will provide more flexibility to users external to OLCF. As part of that effort, we are producing documentation that will be publicly available on how to use the OTH and how to configure it for another site.

VIII. Conclusion

The OLCF has a long history of deploying leadership-class HPC systems. Having a testing framework that can simulate realistic workloads on an HPC system has proven to be a key resource not only for acceptance testing, but also to identify potential at-scale issues during normal operations and before a major upgrade [14]. As these systems have become more complex and have grown in scale, so have our acceptance and regression testing efforts. In order to simplify and standardize internal testing procedures, in 2007, the OLCF developed the OLCF Test Harness (OTH). Since then, the OLCF has continued to refine and enhance the OTH based on in-house expertise gained during use of the framework for acceptance of Jaguar, Titan, and Summit. This continuous evolution has allowed us to optimize the framework for the OLCF ecosystem. Today, the OTH is a Python3 testing framework that supports the most commonly used schedulers (PBS, LSF, Slurm) and several job launchers in use at the OLCF.

A major restructuring of the OTH was completed to support acceptance testing of the Summit supercomputer [15]. During that effort, we identified several opportunities to further simplify the test creation process and new features to address functionality gaps we encountered. The updates described in this paper will become part of the OTH version 2.1, which will be used for the forthcoming acceptance testing of the OLCF's exaflop supercomputer Frontier. Our hope is that the features we have added for v2.1 will simplify

the steps needed to support new systems, and will allow other HPC centers to leverage the OTH in their own testing.

ACKNOWLEDGMENT

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

REFERENCES

[1] A. Tharrington, "An overview of nccs xt3/4 acceptance testing," in Proceedings of the Cray User Group 2007 conference, 2007.

[2] A. S. Bland, W. Joubert, R. A. Kendall, D. B. Kothe, J. H. Rogers, and G. M. Shipman, "Jaguar: The world's most powerful computer system–an update," Cray Users Group, 2010.

[3] A. N. Tharrington, "Nccs regression test harness, version 00," 9 2015. [Online]. Available: https://www.osti.gov//servlets/purl/1232564

[4] "Summit: Scale new heights. discover new solutions." [Online]. Available: https://www.olcf.ornl.gov/summit/

[5] R. McLay, K. W. Schulz, W. L. Barth, and T. Minyard, "Best practices for the deployment and management of production hpc clusters," in SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–11.

[6] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, D. Gunter, and K. A. Persson, "Fireworks: a dynamic workflow system designed for high-throughput applications," Concurrency and Computation: Practice and Experience, vol. 27, no. 17, pp. 5037–5059, 2015. [Online]. Available: http://dx.doi.org/10.1002/cpe.3505

[7] "GitLab," https://docs.gitlab.com/, 2020.

[8] "GitLab Runners," https://docs.gitlab.com/ee/ci/runners/README.html, 2020.

[9] "ReFrame," https://reframe-hpc.readthedocs.io/, 2020.

[10] "Pavilion 2.0," https://pavilion2.readthedocs.io, 2020.

[11] "ReFrame: CSCS Checks," https://github.com/eth-cscs/reframe/tree/master/cscs-checks, 2020.

[12] "Pavilion," https://github.com/lanl/Pavilion, 2017.

[13] C. Kuchta, R. Budiardja, and V. Melesse Vergara, "Harmony: A harness monitoring system for the oak ridge leadership computing facility," PEARC 2019 Conference Proceedings, 7 2019.

[14] V. G. V. Larrea, H. S. Oral, D. B. Leverman, H. A. Nam, F. Wang, and J. A. Simmons, "A more realistic way of stressing the end-to-end i/o system," in Proceedings of the Cray User Group 2015 conference, 2015.

[15] V. Melesse Vergara, W. Joubert, M. J. Brim, R. Budiardja, D. Maxwell, M. Ezell, C. Zimmer, S. Boehm, W. Elwasif, H. Oral, C. Fuson, D. S. Pelfrey, O. Hernandez, D. B. Leverman, J. A. Hanley, M. Berrill, and A. Tharrington, "Scaling the Summit: Deploying the World's Fastest Supercomputer," 6 2019.