Regression Testing on Petaflop Computational Resources

Mike McCarty, **Troy Baer**, and **Lonnie Crosby** of the
National Institute for Computational Sciences (NICS)

ABSTRACT: *In order to deliver the best possible service to our users, it is critical that we understand the state of our systems at all times. Routine system checks performed after scheduled maintenance or emergency downtime give administrators an instantaneous glimpse of system performance but may not detect all performance issues. Rigorous testing, such as that performed for machine acceptance, provides more in-depth information on system performance. Both routine and rigorous testing is necessary to fully characterize system performance, and a mechanism to store and compare previous results is needed to determine the change in system performance over time. To this end a regression-testing framework has been developed at the National Institute for Computational Sciences (NICS), which provides a mechanism to measure the change in system performance over time. These performance results can also be correlated to system events such as downtimes, system upgrades, or any other documented system change. We will describe the design and implementation of the regression testing framework, including the development of test suites, interfaces to the batch system, and the extraction of performance data. The import of extracted data into a relational database for long-term storage, report generation, and real-time analysis will also be discussed.*

KEYWORDS: Testing, Regression Testing, Tools, Frameworks

1. Introduction

Regression testing is used during software development to find errors by partially retesting code after it has been modified with the goal of measuring the impact of the recent changes [1]. After the errors are fixed the tests are rerun periodically to ensure they do not re-emerge. We have taken this software development approach and applied it to maintaining supercomputers. Like software, a supercomputer is a system of extremely complex pieces. Each piece can be changed independently which sometimes dramatically affects unaltered components. The causes for these adverse effects can be difficult to track down since symptoms can emerge long after the initial modification was made. By capturing performance data and measuring how they change over time, we hope to gain a better understanding of how modifications to the system affect related components.

Regression tests are being used at NICS to perform system checks after a preventative maintenance (PM). Once the system is booted, a suite of tests automatically runs full machine jobs using three different applications. These jobs were previously executed manually and analyzed by hand before the regression test framework was developed. After each test completes, the results are analyzed and compared against previous results. For example, the walltime and core count for every test run within the framework is stored in a relational database. Using this database, the mean and standard deviation for a particular test can be calculated. One built-in analysis method verifies that the walltime for jobs with similar core counts is within three standard deviations of the mean (assuming a normal distribution). If the results are outside of a nominal range, an alert is issued which prompts for further investigation of the failure. Since the test results are all stored in a relational database, along with a time stamp and job characteristics, it is possible to analyze this data for trends to detect more insidious anomalies. These anomalies could be caused by an upgrade or modification to the operating system or perhaps a problem with the file system. System logs can be correlated with regression test data to uncover the underlying cause.

The framework views the tests that actually run on the system as a black box, and most of the tests written using the framework aim to stress the system in some

way. The regression test simply takes the executable and submits it to the scheduler and monitors the job until it is complete. It waits on the job to complete its execution. It is the test writer's responsibility to decide which application or benchmark to use, and to analyze the results to determine whether the test passed or failed. The framework does, however, contain a set of default analysis methods (assertions). If the default assertions are called at the end of a test, the test will check the walltime. Some tests written for system checkouts after a PM were compiled with the Fast Profiling library for MPI (FPMPI) option [7]. This option produces a profile file that contains performance metrics, which the framework imports into the database. A few reports have been produced using this data.

Regression testing can be a labor intensive task to perform manually. Rerunning tests periodically is necessary to build up regression data over time. One of the challenges faced in the beginning stages of deployment is that there is no regression data to compare to; therefore, it is difficult to determine what nominal values should be. We simply must run the tests often to build up statistics. Once data has been acquired, reports can be automatically generated, since the data is stored in a convenient format. We have created a web interface, using the Python web application development framework Django [2], to produce plots of scaling curves in the browser. The service implements a RESTful interface that allows the user to specify a date range (including a date to split the plot into two data sets). This allows the user to compare the performance before and after events such as unexpected downtime, an operating system upgrade, or any other type of systematic modification or anomaly.

2. Systems at NICS

Kraken is a Cray XT5 system with a peak performance of 1.03 PetaFLOPs containing 16,512 compute sockets and more than 129 terabytes of memory. Recently name the third fastest computer in the world, the XT5 system delivers in excess of 700 million CPU hours per year.

Athena is a 166 TF Cray XT4, which is dedicated to solving important problems, particularly in climate and physics. Athena has been operating in dedicated mode since October 1, 2009.

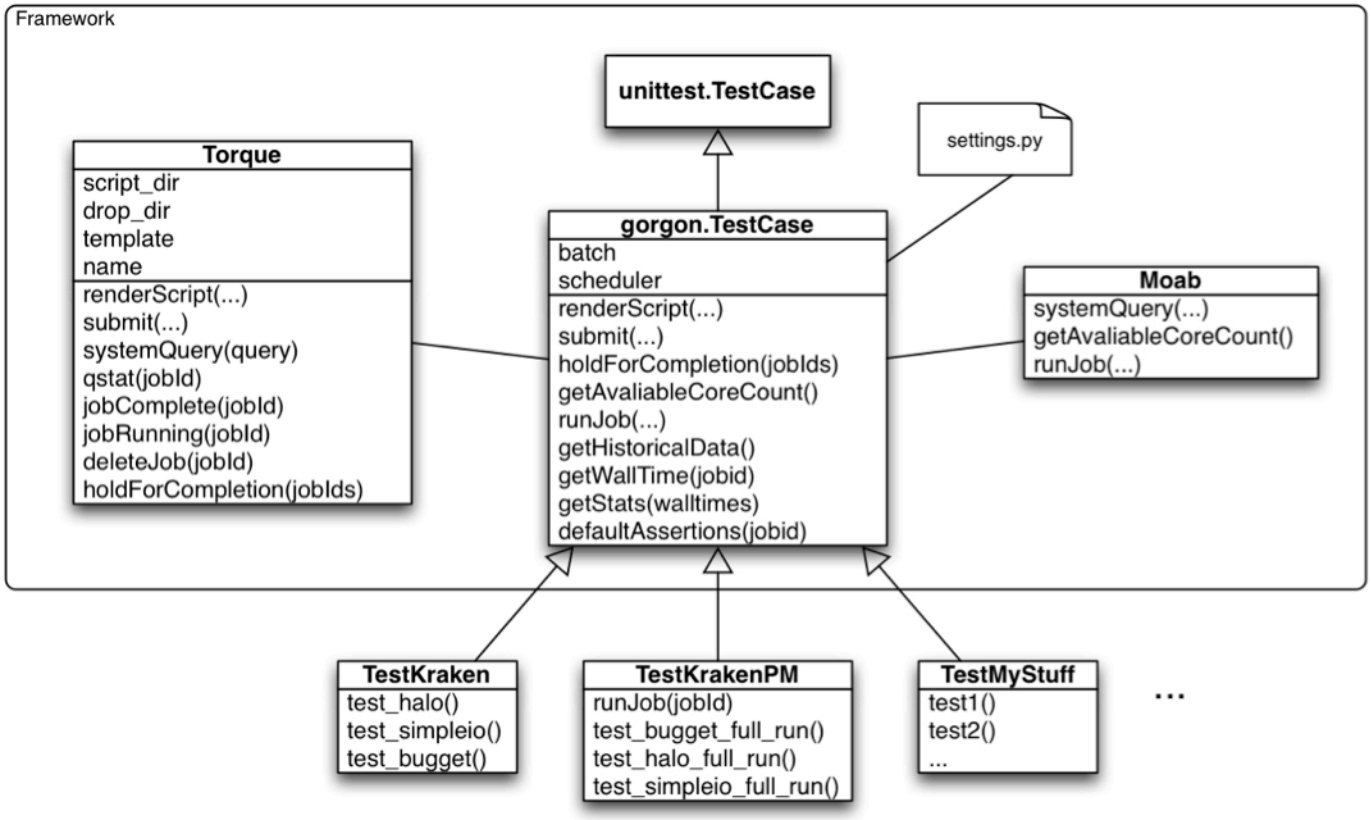

Figure 1 Regression Testing Framework Architecture

3. The Framework

The primary goal of the regression testing framework is to automate the role of the administrator as much as possible. We wanted test writers to only think about the specific test specifications. Parameters may include the number of cores, executable, whether or not to look for FPMPI profile data, the required walltime, and additional PBS job script options. The framework takes these parameters, which make up the specification and generates the PBS script. The PBS script is submitted programmatically behind the scenes by the framework. Options exist to force the job to run and hold the execution of the test until the job has completed. Holding for job completion is useful when the author wants to validate results in the job's output file. It is highly recommended that the test author make assertions at the end of a test to make sure the test was successful. Jobs can be submitted which result in an error at runtime; if there are no assertions at the end of a test then the framework will falsely report it as passing when it actually failed. A default assertion method is provided that attempts to check for the existence of an error-free output file. This method also compares the amount of walltime used with results from previous runs.

3.1 Architecture

The regression testing framework was developed in Python. A scripting language was chosen because we needed to coordinate with several existing systems, such as the batch system (TORQUE), the scheduler (Moab), the database (PostgresSQL), and the operating system itself [4][5]. A native object-oriented scripting language was chosen to leverage encapsulation and code reuse. Python's rich set of third party libraries for testing, web development, and plotting also made it the perfect choice.

Much of the testing portion of the framework is based on a testing library for Python, called PyUnit or unittest [3]. This library is generally used for unit-testing code, which is somewhat different from regression testing in that it aims to test individual, discrete units of code rather than larger portions of code. Here we are leveraging the test execution capabilities of the unit test library for our purposes. When a Python script containing tests is executed, the library looks for classes within the script that are derived from a base class, called TestCase. When it finds such a class it will run a method called setup, if it exists. The method is generally used to set up initial conditions for the test. After setUp runs, the library will find and call each method in the class that begins with "test". It is for this reason that all test methods in the regression test framework must begin with "test". We have extended the base class TestCase to meet our requirements. It contains interfaces to the system, the XT's batch system TORQUE, and the XT's workload manager Moab. All tests interface to these systems through the methods provided by our extended base class. See figure 1 for an architecture diagram.

Name	Description	Default
size	The number of processors for the job to run on.	12
machine	The name of the machine that the test will be ran on.	None
name	The name of the executable.	None
project	The name of the project to charge the job too.	None
walltime	The walltime limit for the job.	00:10:00
pbs_additions	A raw string for specifying custom PBS variables.	Empty String
env_vars	A raw string for specifying environment variables.	Empty String
preamble	A raw string for specifying a preamble of code that is inserted into the PBS script before the aprun command is issued.	Empty String
aprun_options	Options for the aprun command.	"-n $PBS_NNODES"
options	Options for the application's executable.	
profile	Boolean that controls whether or not to look for a profile from FPMPI.	True

Table 1 PBS Script Rendering Parameters

3.1.1 Batch System Interface

Kraken (XT5) and Athena (XT4) both use TORQUE as their batch environment. We implemented an interface to TORQUE using Popen, since there is not an API available for TORQUE in Python. The interface is found in the system Python module in a class called TORQUE [4]. The Torque class provides a method, renderScript, to create a PBS script from a set of test specifications. The PBS script is generated using a base template and is customized for each test through a set of keyword arguments. Table 1 describes these arguments.

Jobs are submitted to the queue by calling the submit method in the Torque class. This method returns the job id, which can be used to force the job to run and monitor its progress by calling other class methods.

3.1.2 Workload Manager Interface

The interface to the XT's workload manager, Moab, is fairly light. The Moab class is located in the systems module and also uses Popen to communicate with Moab. This class serves two main purposes, which are querying for the available number of cores at any given time and forcing a job to run.

3.2 Using the Framework

When authors write a test, they are simply writing a Python script, so they are free to leverage the language without restrictions. For example, they can use a loop to incrementally submit jobs to different numbers of cores. This can produce data for a scaling curve, as illustrated in Figure 2. Tests inherit the TORQUE and Moab interface methods from the base class, TestCase. The basic pattern to any test is to render the batch script, submit, and wait for the job(s) to complete. After that the writer may choose to inspect the job(s) output file(s) and make assertions.

```
from TestCase import TestCase

class TestKraken(TestCase):

    def test_halo(self):
        jobids = []
        for size in range(1056, 16000, 1056):
            self.renderScript(size     = size
                            , name     = "halo"
                            , walltime = "00:10:00"
                              )

            jobids.append(self.submit())
        self.holdForCompletion(jobids)

if __name__ == "__main__":
    print "Running tests..."
    import unittest
    unittest.main()
```

Figure 2 Example of a scaling curve test

The framework outputs the test results using character codes: a "." means the test completed successfully, an "F" means the test failed from an assertion, and an "E" indicates an error. In the case of an error, the Python trace back will be printed once all tests have completed [3]. When a test fails you get the output shown in figure 3. If there are no assertions at the end of a test and the code has no runtime errors the job will run, but the expected output may not be correct. For example, the application could fail without indication. It is recommend that the writer always, at least, call the default assertions. These check the output file for errors. Figure 3 below shows the test output.

3.3 Post Processing and Reporting

Post processing is performed on all job status files and FPMPI process statistics by a cron job that runs daily [7]. The data is placed in a drop directory specified in a settings file at the end of each test. Data from these files are parsed and inserted into the database for use in future regression tests and reporting. Post processing is performed on an external machine to avoid additional load on the XT service nodes. Figure 4 contains an

```
mmccarty@krakenpf7(XT5):/lustre/scratch/mmccarty/regression_tests> python ~/sandbox/gorgon/regression_tests/trunk/TestKrakenPM.py
Running tests...
.EF
======================================================================
ERROR: test_halo_full_run (__main__.TestKrakenPM)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/nics/a/home/mmccarty/sandbox/gorgon/regression_tests/trunk/TestKrakenPM.py", line 37, in test_halo_full_run
    raise
TypeError: exceptions must be classes, instances, or strings (deprecated), not NoneType

======================================================================
FAIL: test_simpleio_full_run (__main__.TestKrakenPM)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/nics/a/home/mmccarty/sandbox/gorgon/regression_tests/trunk/TestKrakenPM.py", line 52, in test_simpleio_full_run
    self.assertTrue(False)
AssertionError

----------------------------------------------------------------------
Ran 3 tests in 2.922s

FAILED (failures=1, errors=1)
mmccarty@krakenpf7(XT5):/lustre/scratch/mmccarty/regression_tests>
```

Figure 3 Test results output

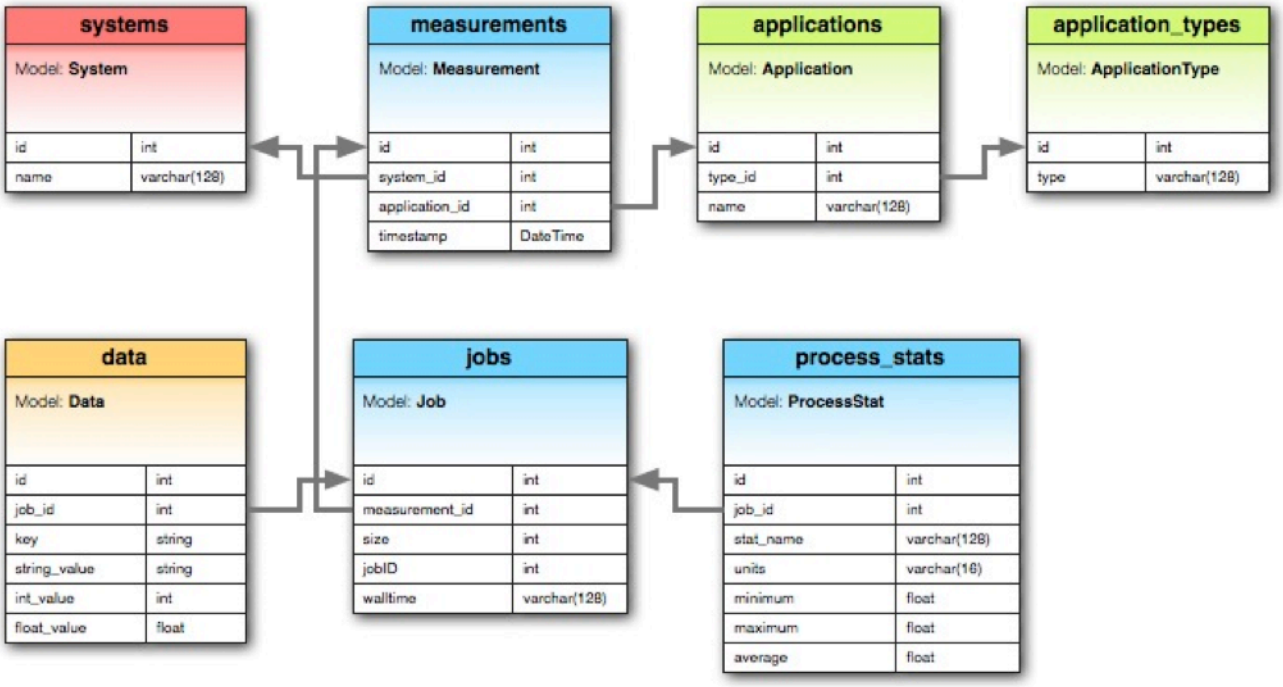

Figure 4 Database Entity-Relationship Diagram

entity-relationship diagram of the database.

Many analysis methods of regression test data are application specific, since they depend on the output of the applications themselves. Some standard post processing and analysis methods have been automated as described above in the introduction. For other non-standard analysis needs we plan to make application specific tests and generalize them when appropriate. The standardized reporting implemented to date consists of application-scaling curves. The idea is to produce scaling curves with various applications that stress the system in different ways. Normally scaling curves are used to determine the performance characteristics of applications as they run on increasing numbers of processors. Generally such information is used to modify the application to improve performance on a given system. However in this case, a static application will be run on a changing system. It is hoped that system changes that affect performance of applications will alter the characteristics of the affected applications scaling curve. Anytime we need to update an application the date must be tagged so the statistics will not be compromised.

As mentioned in the introduction, the system-scaling curves are generated by a web interface written in Django. Django is a Python based web development framework that is gaining a strong following in the web application development community. It employs the model-view-controller (MVC) design pattern, which separates an application into three layers. An object-relational mapper is used to represent database tables as classes and provides a persistent abstraction layer between the business logic and the database I/O [2]. Instances of classes represent rows in the database table. The controller layer contains logic for handling HTTP

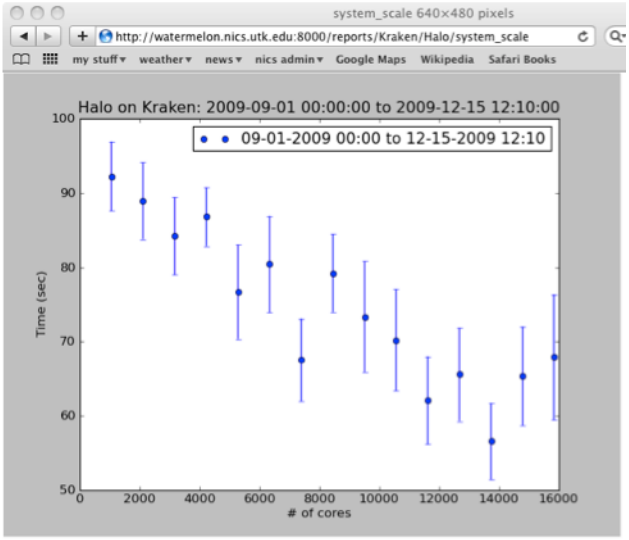

Figure 5 System Scale Plot in the browser

requests and generates responses. Views control the presentation of the application through XHTML template rendering engines.

Plots, produced using MatPlotLib, are generated on request via a RESTful URL [6]. For example, the URL /reports/Kraken/Simpleio/system_scale generates a system-scaling curve for all data collected for the application called SimpleIO which ran on Kraken. Keyword parameters for specifying the upper bounds on the number of cores and date ranges are also available. Figure 5 shows a system-scaling curve for another test application called Halo, which tests MPI communication times.

4. Experiences

We have been running a suite of tests developed using the regression test framework since January 1, 2010. These tests use three applications developed to stress the system in different ways. Halo, one test application, creates an MPI token ring through the XT system's torus network. Its primary purpose is to test the system's network connectivity. Another test application, called SimpleIO, aims to stress the file system but creating files for each process to make sure that they can write files. Bugget taxes the CPUs by perform matrix multiplications and produces performance statistics in its output.

The SimpleIO application tests have uncovered an anomaly in I/O performance that is currently under investigation. We are mining data from logs and other status snapshots to identify any correlations. We hope to present an analysis in the near future.

5. Future Work

One of the key issues moving forward with regression testing at NICS cannot be solved in software. A policy decision must be made on when and how often we can run tests. The tests developed to date are used after a preventative maintenance when we are trying to get the machine back to users as fast as possible. However, we plan to develop several suites of tests and applications that will focus on tracking memory, file system I/O, network, and MPI performance. During acceptance testing we tested the system for functionality, performance, and stability; which took nearly 24-hour jobs. Here we are only concerned with performance tests, but how complex should these tests be and how long should they run?

We also plan to make the framework available under an open source license. We are currently working the University of Tennessee on intellectual proper rights.

6. Conclusion

Test writing using the regression test framework is currently being used weekly at NICS to run and validate full machines jobs after maintenance on Kraken. The capability to automatically run machine checkouts while collecting additional data has had a positive impact on the machine maintenance process. As more tests are run, the statistics will become more useful in telling us how the system's performance is changing over time. Since Kraken has a 95% uptime requirement these tests can only be run once a week, therefore our sample size remains somewhat small at the time of writing this paper.

In the future we would plan to add more applications for benchmarking the system. We are currently working on an application for testing I/O patterns that we hope to integrate into the regression tests. The issue with running these tests is how often can we run them and collect useful information without affecting operations.

References

[1] Regression Testing:
http://en.wikipedia.org/wiki/Regression_testing
[2] Django:
http://www.djangoproject.com/
[3] PyUnit:
http://pyunit.sourceforge.net/
[4] "Cluster resources :: Products - TORQUE Resource Manager",
http://www.clusterresources.com/pages/products/torque-resource-manager.php.
[5] "Cluster resources :: Products - Moab Workload Manager",
http://www.clusterresources.com/pages/products/moab-cluster-suite/workloadmanager.php.
[6] MatPlotLib:
http://matplotlib.sourceforge.net/
[7] FPMIP:
http://archive.ncsa.illinois.edu/lists/perftools/apr02/msg00005.html

Acknowledgments

The authors would like to thank Patricia Kovatch, Nick Jones, and Matt Ezell for their comments and assistance on writing this paper.