Abstract

The National Center of Computational Sciences (NCCS) and Cray Inc. have embarked on an aggressive program to bring a petascale supercomputer to Oak Ridge National Laboratory (ORNL). The process involves progressively upgrading its premier Cray XT3 system, code-name Jaguar, in 2006 from 25 teraflops to 50 teraflops and again in 2007 to 119 teraflops. Jaguar will be upgraded once again to 250 teraflops by late 2007. Finally, in late 2008, the center will install a Cray XT3 system, code-named Baker, with a peak performance of 1 petaflops.

At each stage along the way, the new systems must undergo acceptance testing to ensure they are capable of performing the computational and operational tasks for which they were designed. The acceptance testing involves both hardware and software tests and is used to demonstrate usability after each upgrade.

This paper provides a brief description of the applications for the software test and detailed discussion of the process for automating the tests themselves. The tests are orchestrated by a 'test harness' which has framework that initializes the submission of the jobs to the system, provides a means for stopping and starting running tests, coordinates the collection of application results, and provides a means of checking the pass rate of tests.

An Overview of NCCS XT3/4 Acceptance Testing

Arnold Tharrington

May 18, 2007

1 Introduction

The National Center of Computational Sciences (NCCS) and Cray Inc. have embarked on an aggressive program to bring a petascale supercomputer to Oak Ridge National Laboratory (ORNL). In 2005 a 25 teraflop (TF) Cray XT3 system, code-named Jaguar, was installed. In 2006, Jaguar was upgraded to 50 TF. In early 2007 Jaguar was upgraded to 119 TF. At the end of 2007 Jaguar will be upgraded to 250 TF. In 2008 a 1 petaflop (PF) system will be installed.

Each system undergoes acceptance testing, designated as Jaguar Acceptance Testing (JAT), to demonstrate its usability. The document ORNL NLCF Acceptance Test Plan explicitly states the requirements for accepting each system. The ORNL Cray Technical Representative (CTR) is the judge of whether or not the requirements of JAT have been met.

The acceptance testing for each system has two parts. The first part is the initial hardware acceptance and is designated as Jaguar Acceptance Test Hardware (JAT-HW). The hardware testing checks the processors, memory, and interconnects. This test is performed on a cabinet-by-cabinet basis at ORNL by Cray Personnel and is witnessed by ORNL Staff.

The second part is designated as Jaguar Acceptance Test Final Integration (JAT-FI). JAT-FI consists of 3 parts performed in the following order: functionality, performance, and stability. The functionality part, designated as Jaguar Acceptance Test Functionality (JAT-FT), verifies the readiness of key systems (file systems, batch scheduler, etc.), libraries (fftw, hdf5, petsc, etc.), and code development services (compilers, debuggers, etc.). The performance test verifies the performance and scalability of the system. The stability part verifies the stability of the system under a simulated workload of code development and production simulations for a sustained length of time.

The JAT-FI has several operational requirements that create demanding logistical execution barriers. One such requirement is that the CTR mandates there be 1 point of contact for running the JAT-FI parts. Another requirement is that all test results within the JAT-FI must be collected, archived, and communicated to the ORNL CTR for evaluation. The JAT-FI also requires the software applications be submitted in such a manner that the machine stays busy during each part of the JAT-FI. Moreover the stability test requires the pass rate for all combined software application be at least 95%, and all individual software applications must have at least 1 pass. This necessitates frequent periodic monitoring of the results of all the stability test applications. These logistical execution barriers were overcome by the use a test harness that automated the JAT-FI parts.

This paper is organized as follows. Section 2 presents the hardware configuration, key acceptance requirements, and software tests for each system. Section 3 presents the NCCS Test Harness, section 4 presents the future direction of the ORNL NLCF test applications and harness, and section 5 presents the conclusions.

2 System Tests and Requirements

Each system has a hardware environment that undergoes the JAT-HW and JAT-FI tests. The JAT-HW environment for the 50 TF, 100 TF, 250 TF, and 1 PF systems is respectively shown in Tables 1, 3, 5, and 7. The JAT-FI applications and libraries for the 50 TF, 100 TF, 250 TF and 1 PF systems are respectively shown in Tables 2, 4, 6, and 8. We limit our discussion to the JAT-FI part of which the

functionality, performance, and stability parts each have entry and exit requirements. Figure 1 gives an overview of the JAT for each system.

2.1 Functionality Test

The purpose of the functionality test is to ensure the system can perform the performance and stability test. The Functionality test entry requirement consists of the following:

1. Removal of any non-standard software and a complete listing of all software installed on the system
2. A successful build of JAT test components

A few important functionality test exit criteria are the following:

1. Continuous running of the functionality test for 12 hours without generating severity 1 or 2 defects
2. No obvious performance problems are detected, including job launch and exit
3. All deviations or limitations observed during JAT-FT deemed essential to ORNL to conducting the JAT-PT and JAT-ST are fixed by Cray

2.2 Performance

The purpose of the performance test is to ensure and fix any problems that would interfere with the performance of the stability test. The performance test entry requirements are the following:

1. Successful completion of the JAT-FT exit criteria
2. Remove any non-standard software installations (sandbox builds, personal fixes, patches, etc.) and install only released software from Cray
3. Provide a complete listing of all installed software, including patch level and any released patches to ORNL

A few important performance test exit criteria are the following:

1. Continuous running of the JAT-PT for 12 hours without generating any severity 1 or 2 defects
2. Successfully pass each of the performance measures
3. Successfully run the full range of problem sizes for each application code
4. Document the performance and scalability results obtained during the JAT-PT

2.3 Stability

The purpose of the stability test is to ensure the system is ready to assume a production workload environment. The stability test entry requirement consists of the following:

1. Successfully complete the JAT-PT exit criteria
2. Remove any non-standard software installations (sandbox builds, personal fixes, patches, etc.) and install only released software from Cray
3. Provide a complete listing of all installed software, including patch level and any released patches to ORNL
4. Negotiated performance metrics of the application tests are met and documented and integrated into the test harness for the stability test

A few of the stability test exit requirement consists of the following:

1. All applications run in the JAT-ST have exhibited the required number of days of hardware and software stability without generating severity 1 or 2 defects
2. Successfully demonstrate the reliability requirements of 95% job completion and 100% correct answers of all completed jobs
3. All application tests consistently run at near the negotiated performance levels during the entire stability test
4. Successfully run the full range of problem sizes for each application code
5. There are no unexplained job terminations

3 NCCS Test Harness

The NCCS Test Harness is a Python software package that was created to overcome the logistical barriers of managing the numerous jobs from running the applications and their associated tests during the JAT-FI parts. The test harness models a build/compile and run through batch work environment. Typically, a JAT-FI part has between 50 and 100 tests to be concurrently run for long sustained times. Therefore, it is critical to frequently monitor the results and intervene as soon as possible if any problems are detected.

In addition, the JAT-FI runs generate approximately tens of terabytes of data which must be archived for future audits. Because of the tight acceptance schedule it impractical to archive at the end of the JAT-FI. The only recourse is to archive during the JAT-FI parts.

Finally, the logistical execution barriers are further exacerbated by the requirement that there be 1 person managing the numerous tests.

3.1 Overall Harness Design and Operation

Figure 2 shows a conceptual depiction of the test harness. The test harness has 2 parts:

1. The harness driver

2. The SVN repository containing the applications and their associated tests

The test harness, through the harness driver, can check out an application with its tests from the SVN repository, start the tests, stop the tests, and calculate the percentage of passed tests. The test harness is operated by making the appropriate input files, setting the appropriate environmental variables, and executing the harness command 'runtests.py'.

Figure 3 is a depiction of the general overview of the harness in operation. The harness input file is created and some environmental variables are set by the user. The input file is located at 'path_a/⟨harness input files⟩/'. The purpose of the harness input file is to select which tests are to be run, and set the checkout location of the tests. The tests are checked out to 'path_b/⟨checked out apps⟩/'. For each test run instance, the harness will make a unique workspace located within the directory 'path_c/⟨checked out apps workspace⟩/' of which it is optional, but highly advisable for the test in question to use. Note that all of the tests for the XT3 acceptance used this unique workspace to prevent concurrent runs of the same tests from clobbering each other results.

3.1.1 Harness Driver Input File

The harness driver has 1 input file named 'rgt.input' of which an example is shown in Figure 4. Blank records and records starting with '#' are ignored. The input file can be logically organized into 3 sections. The first section specifies the path to the location where the applications and tests are checked out from the SVN repository. This is specified by the record starting with 'Path_to_tests'. This path must be any valid path that the user has r, w,and x permission.

The second section specifies the applications and tests to be run. The records starting with 'Test' specifies an application and test. An single test consist of specifying the application and one of the subtests of the application. Any number of tests can be run.

The third section specifies the action to be performed on the tests. The records starting with 'Harness_tasks' specify a task. Only one tasks at a time can be performed for a given number of tests. For our input file, simply uncomment a harness task.

3.1.2 Harness Environmental Variables

The environmental variables are defined by sourcing the configuration file shown in Figure 5, and loading the nccs_test_harness/0.2 module. An important environmental variable definition is RGT_PATH_TO_SSPACE. This variable defines the root location of where the tests are build and run. Note the build and run location are distinct from the path where the applications are checked out to.

3.1.3 Applications/Tests and SVN Repository

Figure 6 shows the layout of an application and test within the SVN repository. An single test is is created by creating the build_executable.x, submit_executable.x, and check_executable.x scripts.

3.1.3.1 build script The build script command line interface is

```
build_executable.x -i uniqueid -p location
```

where 'uniqueid' is the time from epoch. This is calculated the harness driver for each run instance of a test. The 'location' is the unique path and generally has a form similar to 'path_c/⟨checked out apps workspace⟩/⟨application name⟩/⟨test⟩/⟨uniqueid⟩' The function of the build script is to build the binary for running the test. Note that some test may not require a binary to be built. The build script only mandatory requirement is that it return one the following values:

1. 0 for a successful build
2. 1 for an unsuccessful build
3. ≥ 2 for an undetermined build

3.1.3.2 submit script The submit script command line interface is

```
submit_executable.x -i uniqueid -p location
```

The purpose of the submit script is to launch the test through the batch queue. The requirements of the submit script is that it returns one following values:

1. 0 for a successful launch
2. 1 for an unsuccessful launch
3. ≥ 2 for an undetermined launch

and that the batch script it launches has the following commands executed last:

```
check_executable_driver.py -i uniqueid
(contd)                        -p location

test_harness_driver.py -r
```

The command check_executable_driver.py will call the check_executable.x. The command test_harness_driver.py will start a new instance of the test.

3.1.3.3 check script The check script command line interface is

```
check_executable.x -i uniqueid -p location
```

The purpose of the check script is to check the results of the test. The only requirement of the check script is that it returns one following values:

1. 0 for a successful test.
2. 1 for an unsuccessful test
3. ≥ 2 for an undetermined test

3.1.4 Requirements of Interactions between Applications/Tests Scripts

As previously stated the harness driver will check out the application and tests to 'path_b/⟨checked out apps⟩/'. Figure 7 shows a detail depiction of the layout of an checked out application and test. For each instance of a test, the directories Run_Archive/⟨ID⟩ and Status/⟨ID⟩ will be created. The Run_Archive/⟨ID⟩ directory respectively store the critical files for archiving. The directory Status/⟨ID⟩ contains a file named 'status.txt' that stores the result of the check_executable.x script. The scripts build_executable.x, submit_executable.x, and check_executable.x scripts will be run from the Scripts directory. One should note that these scripts must be written in a manner that does not depend upon their absolute location but only upon the relative paths of the other directories and files within a test. During a run, the cumulative results of a test is stored in the file 'Status/rgt_status.txt'.

4 Future Direction

Although the test harness is robust, it has a need of few improvements. One need is for better error recovery and restart functionality. For example, the harness moves and copies much data during a run and sometimes tests exceed their batch wall time due to the high load on the files system. These tests will stop and must be restarted by a tedious manual process.

Secondly, the harness needs a more robust set of tests. We have a very limited set of intrinsic hardware tests. We are currently collaborating with other High Performance Computing (HPC) centers to expand our set.

Finally, the harness is not completely portable to other UNIX shells. The harness was written using a BASH shell. However, the user environment is not the same for all UNIX shells on Jaguar (the XT3). This causes problems when other users wish to operate the harness.

5 Conclusions

The NCCS Test Harness is a robust software framework to manage a large number tests. A single user can typically startup all tests (≈ 100) in our harness suite in 2 hours. Thereafter, periodic checking on the tests generally occurs every 6-8 hours.

Their are minimal requirements on users adding new applications and tests which permits very fast and allows integration of numerous types of test. There are sample tests that are easily adaptable for new tests. One example of the variety of tests that can be integrated into the harness is the CCSM smoke test. CCSM has a complicated build procedure which was managed to be integrated, with some difficulty, in our harness.

Lastly, the harness is very extendible. It is written in Python with extensive use of object-oriented design. Nearly all major data-structures are classes. For example, the directory layout of an application is a class. This permits an easy modification, manipulation, or generation of key paths in the directory layout.

Type	**Quantity**
Compute Nodes	5,212 AMD Opteron Dual-Core Processors
Compute Threads	10,424
Memory per Node	2 GB
Global Disk Space	120 TB
Global Disk Bandwidth	14 GB/s
External Network I/O connections	1 Gb/s Ethernet x 38 and 10 Gb/s Ethernet x 2
Login Nodes	8

Table 1: 50 TF hardware configuration

	Sustained Time	**Requirements**	**Applications**
Functionality	24 hrs	Each test must have at least 1 pass	Intel MPI Benchmarks, MPICH Test Suite, FFTW2, FFTW3, HDF5, CCSM, NetCDF, Petsc, Scalapack, Kickstart
Performance	24 hrs	Each test must have at least 1 pass, Meet performance criteria requirements of contract	Intel MPI Benchmarks, MPICH Test Suite, Presta MPI Bandwidth and Latency Benchmark, FFTW2, FFTW3, CCSM, NetCDF, Petsc, Scalapack, LSMS, VH1, GTC, S3D, POP
Stability	72 hrs	Each test must have at least 1 pass, Combined pass rate of 95%	Intel MPI Benchmarks, MPICH Test Suite, Presta MPI Bandwidth and Latency Benchmark, Kickstart, LSMS, VH1, GTC, S3D, POP

Table 2: 50 TF applications and libraries configuration

Type	**Quantity**
Compute Nodes	6,296 AMD Opteron Dual-Core Processors
Compute Threads	12,592
Memory per Node	2 GB
Global Disk Space	767 TB
Global Disk Bandwidth	41 GB/s
External Network I/O connections	1 Gb/s Ethernet x 38 and 10 Gb/s Ethernet x 2
Login Nodes	20

Table 3: 100 TF hardware configuration

	Sustained Time	**Requirements**	**Applications**
Functionality	24 hrs	Each test must have at least 1 pass	Intel MPI Benchmarks, MPICH Test Suite, FFTW2, FFTW3, HDF5, CCSM, NetCDF, Petsc, Scalapack, Kickstart
Performance	24 hrs	Each test must have at least 1 pass, Meet performance criteria requirements of contract	Intel MPI Benchmarks, MPICH Test Suite, Presta MPI Bandwidth and Latency Benchmark, FFTW2, FFTW3, CCSM, NetCDF, Petsc, Scalapack, LSMS, VH1, GTC, S3D, POP
Stability	72 hrs	Each test must have at least 1 pass, Combined pass rate of 95%	Intel MPI Benchmarks, MPICH Test Suite, Presta MPI Bandwidth and Latency Benchmark, Kickstart, LSMS, VH1, GTC, S3D, POP

Table 4: 100 TF applications and libraries configuration

Type	**Quantity**
Compute Nodes	6,296 AMD Opteron Quad-Core Processors
Compute Threads	25,184
Memory per Node	8 GB
Global Disk Space	767 TB
Global Disk Bandwidth	41 GB/s
External Network I/O connections	1 Gb/s Ethernet x 38 and 10 Gb/s Ethernet x 2
Login Nodes	20

Table 5: 250 TF hardware configuration

	Sustained Time	**Requirements**	**Applications**
Functionality	24 hrs	Each test must have at least 1 pass	Intel MPI Benchmarks, MPICH Test Suite, FFTW2, FFTW3, HDF5, CCSM, NetCDF, Petsc, Scalapack, Kickstart, Global Arrays
Performance	24 hrs	Each test must have at least 1 pass, Meet performance criteria requirements of contract	Intel MPI Benchmarks, MPICH Test Suite, Presta MPI Bandwidth and Latency Benchmark, FFTW2, FFTW3, CCSM, NetCDF, Petsc, Scalapack, LSMS, VH1, GTC, S3D, POP, AORSA,Global Arrays
Stability	72 hrs	Each test must have at least 1 pass, Combined pass rate of 95%	Intel MPI Benchmarks, MPICH Test Suite, Presta MPI Bandwidth and Latency Benchmark, Kickstart, LSMS, VH1, GTC, S3D, POP, AORSA

Table 6: 250 TF application and libraries configuration

Type	**Quantity**
Compute Nodes	22,400 AMD Opteron Quad-Core Processors
Compute Threads	89,600
Memory per Node	8 or 16 GB
Global Disk Space	5 to 15 PB
Global Disk Bandwidth	240 GB/s
External Network I/O connections	1 Gb/s Ethernet x 20 and 10 Gb/s Ethernet x 2
Login Nodes	20

Table 7: 1 PF hardware configuration

	Sustained Time	**Requirements**	**Applications**
Functionality	24 hrs	Each test must have at least 1 pass	Intel MPI Benchmarks, MPICH Test Suite, FFTW2, FFTW3, HDF5, CCSM, NetCDF, Petsc, Scalapack, Kickstart, Global Arrays
Performance	24 hrs	Each test must have at least 1 pass, Meet performance criteria requirements of contract	Intel MPI Benchmarks, MPICH Test Suite, Presta MPI Bandwidth and Latency Benchmark, FFTW2, FFTW3, CCSM, NetCDF, Petsc, Scalapack, LSMS, VH1, GTC, S3D, POP, NWChem, AORSA
Stability	72 hrs	Each test must have at least 1 pass, Combined pass rate of 95%	Intel MPI Benchmarks, MPICH Test Suite, Presta MPI Bandwidth and Latency Benchmark, Kickstart, LSMS, VH1, GTC, S3D, POP, NWChem, AORSA

Table 8: 1 PF application and libraries configuration

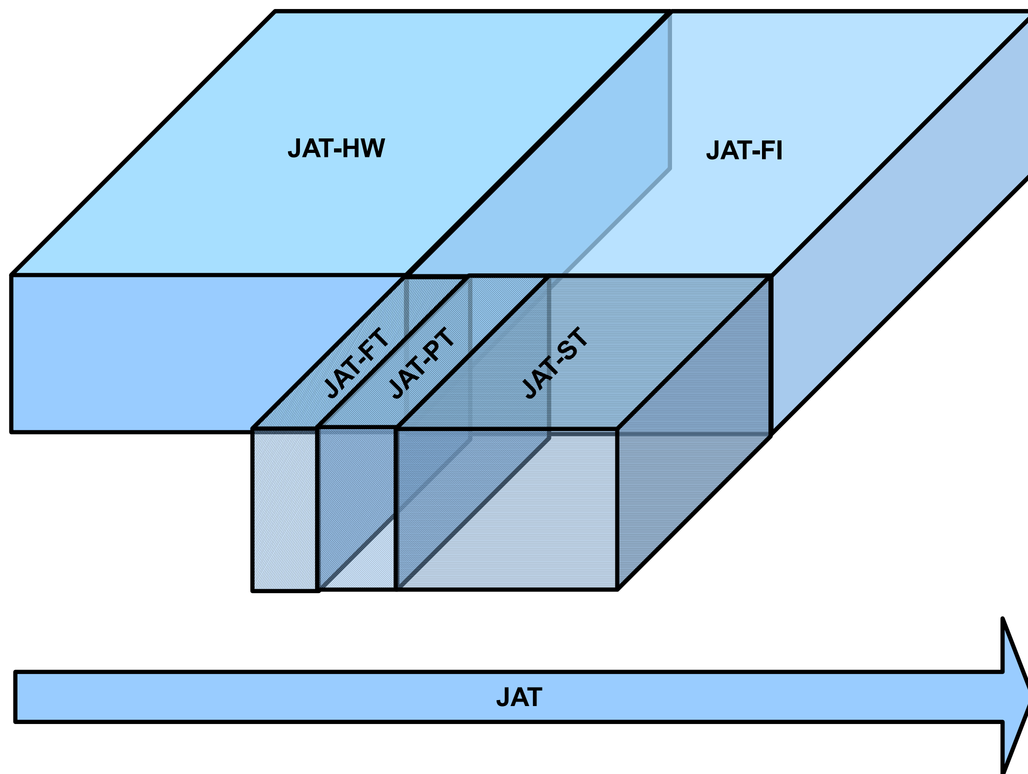

Figure 1: An overview of the JAT for each system or hardware environment. The JAT has 2 parts: the JAT-HW which is the initial hardware acceptance, and the JAT-FI which is the final integration. The JAT-FI contains 3 parts which are the functionality, performance, and stability tests. The arrow symbolizes the order in time the parts are performed

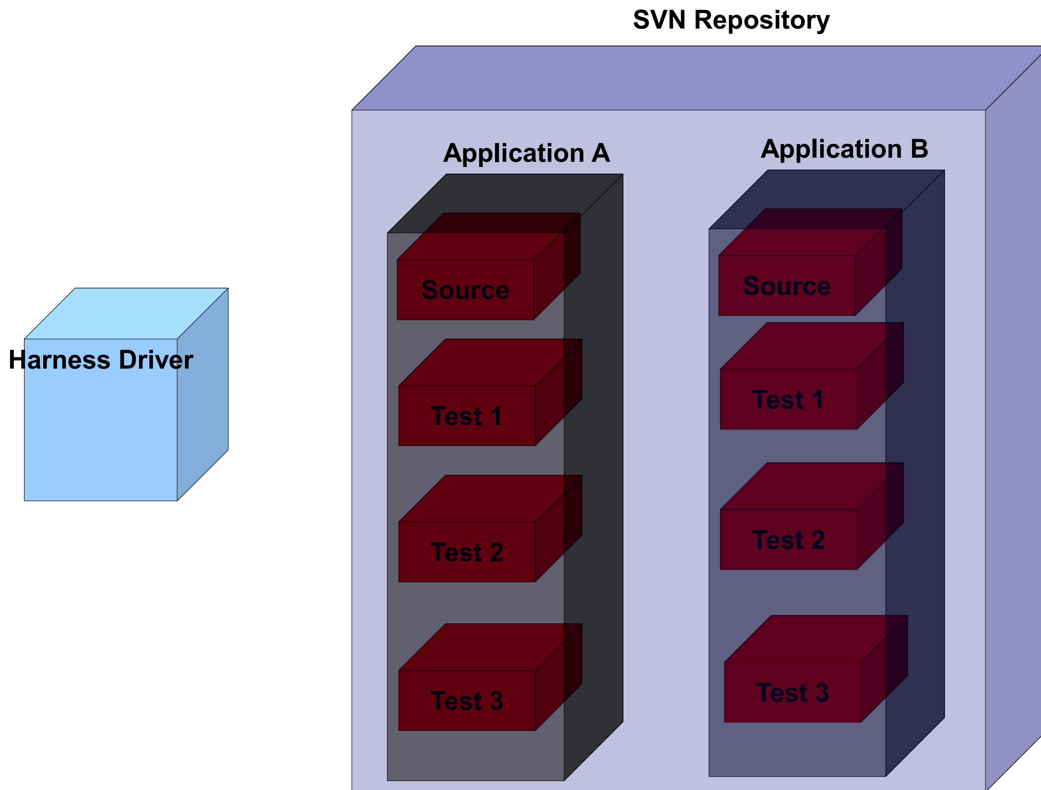

Figure 2: An overview of the harness design. The harness can be logically partitioned into 2 major parts-the Harness Driver and the SVN repository. The Harness driver operates or runs the checked out applications and tests from the SVN repository.

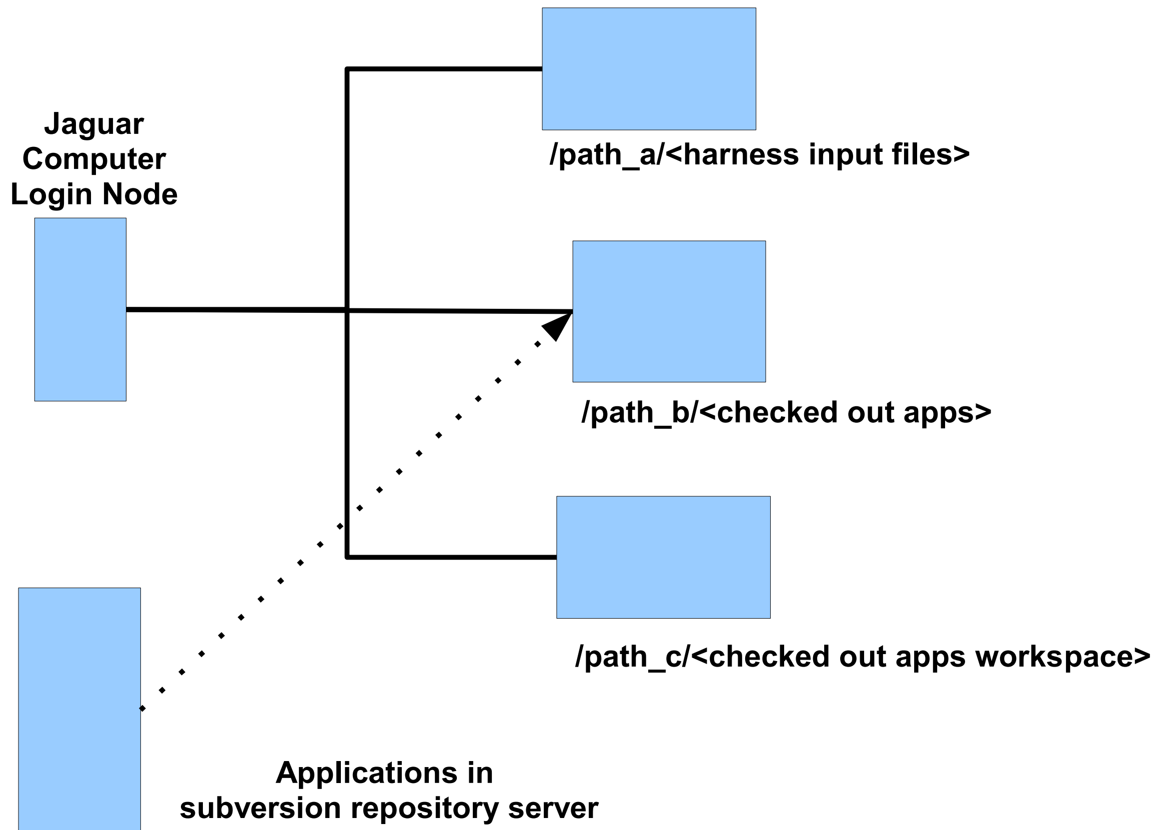

Figure 3: An overview of the directory layout of the applications during a test instance. The dotted line shows the initial checkout of the application and test from the SVN repository. Note the location where the applications and tests are checked out to are distinct form the build and run location.

```
################################################################################
#  Set the path to the top level of the application directory.                 #
#                                                                              #
################################################################################
Path_to_tests = /lustre/scr144/arnoldt/trial_applications_6

################################################################################
#      Name of Applications       Name of Tests                                #
#                                                                              #
################################################################################
Test = IMB_v2.3                   Dual_Core_4096_Sockets

################################################################################
#      Name of Applications       Name of Tests                                #
#                                                                              #
################################################################################
Test = GTC                        Dual_Core_4096_Sockets
Test = GTC                        Dual_Core_8192_Sockets

################################################################################
#      Name of Applications       Name of Tests                                #
#                                                                              #
################################################################################
Test = MPICH-test_v1.1                 Test1

################################################################################
#  The task the harness can perform on the tests.                              #
#                                                                              #
# Simply uncomment the appropiate tasks(s).                                    #
################################################################################

###############################################
# Checks of the test from the SVN repository. #
###############################################
#Harness_task = check_out_tests

################################################
# Starts the test                              #
################################################
#Harness_task = start_tests

################################################
# Display the status of the tests.             #
################################################
#Harness_task = display_status

###############################################
# Stops the tests.                            #
###############################################
#Harness_task = stop_tests
```

Figure 4: A harness input file.

```
#! /usr/bin/env bash

#
# Author: Arnold Tharrington
# Email: arnoldt@ornl.gov
# National Center of Computational Science, Scientifc Computing Group.
#

#
# This file defines and sets user specific environmental variables for the test
# harness.
#


# Add or modify as needed to suit your environment.

    #-----------------------------------------------------------------
    # Absoulte path to scratch space location.                       -
    #-----------------------------------------------------------------
    RGT_PATH_TO_SSPACE='/lustre/scr144/arnoldt/trial_scratch_6'
    export RGT_PATH_TO_SSPACE

    #-----------------------------------------------------------------
    # PBS job account id.                                            -
    #-----------------------------------------------------------------
    RGT_PBS_JOB_ACCNT_ID='stf006bf'
    export RGT_PBS_JOB_ACCNT_ID

    #-----------------------------------------------------------------
    # Set the path to this file                                      -
    #-----------------------------------------------------------------
    RGT_ENVIRONMENTAL_FILE='/spin/home/arnoldt/rgt_environmental_variables.bash.x'
    export RGT_ENVIRONMENTAL_FILE

    #-----------------------------------------------------------------
    # Name of nccs test harness module to load                       -
    #-----------------------------------------------------------------
    RGT_NCCS_TEST_HARNESS_MODULE='nccs_test_harness/0.2'
    export RGT_NCCS_TEST_HARNESS_MODULE
```

Figure 5: A harness environmental definition file. This file is sourced to set critical harness environmental variables.

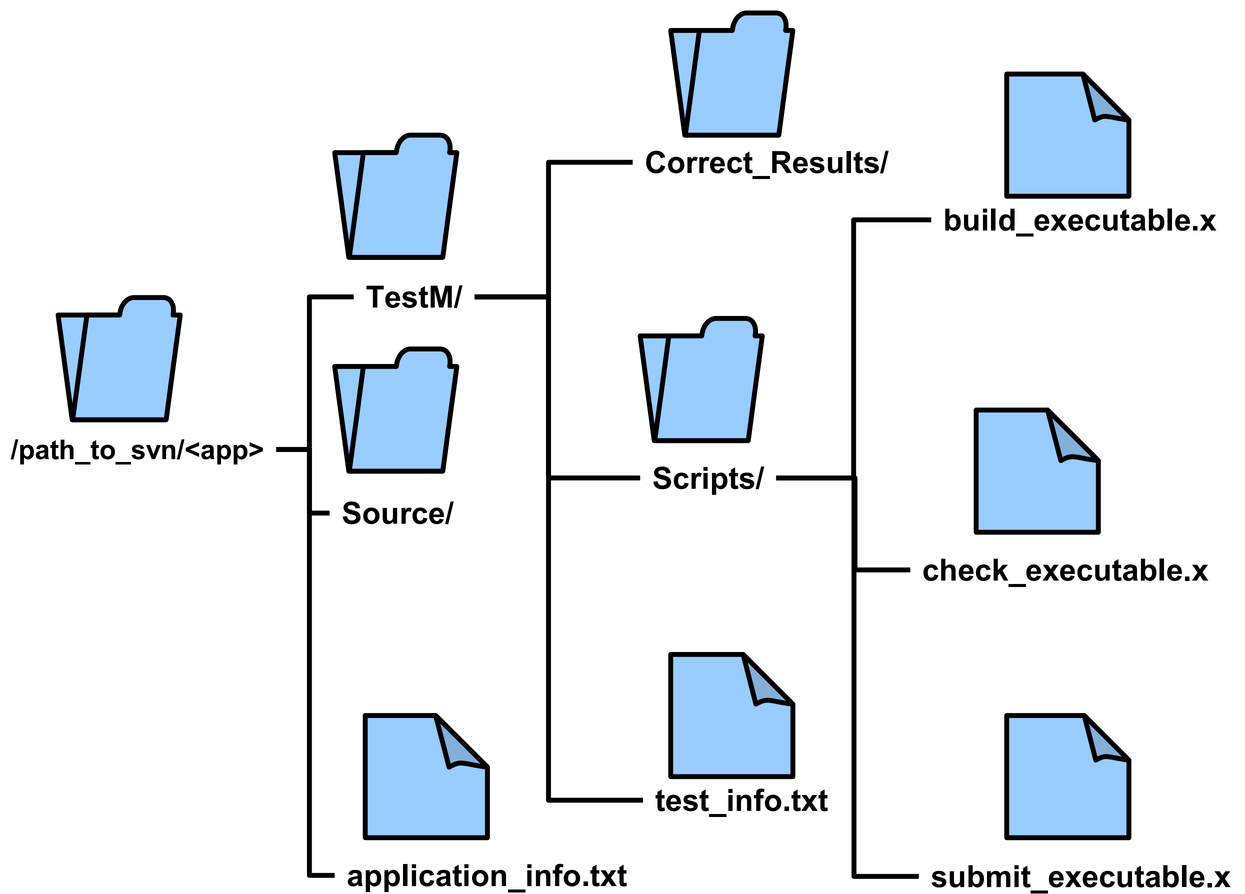

Figure 6: An overview of an application layout in subversion repository for TestM. The directory Source contains the source code for the application. The directory Correct_Results contains the valid results for the test. The Scripts directory contains the build, submit, and check scripts for the test which are the only mandatory files.

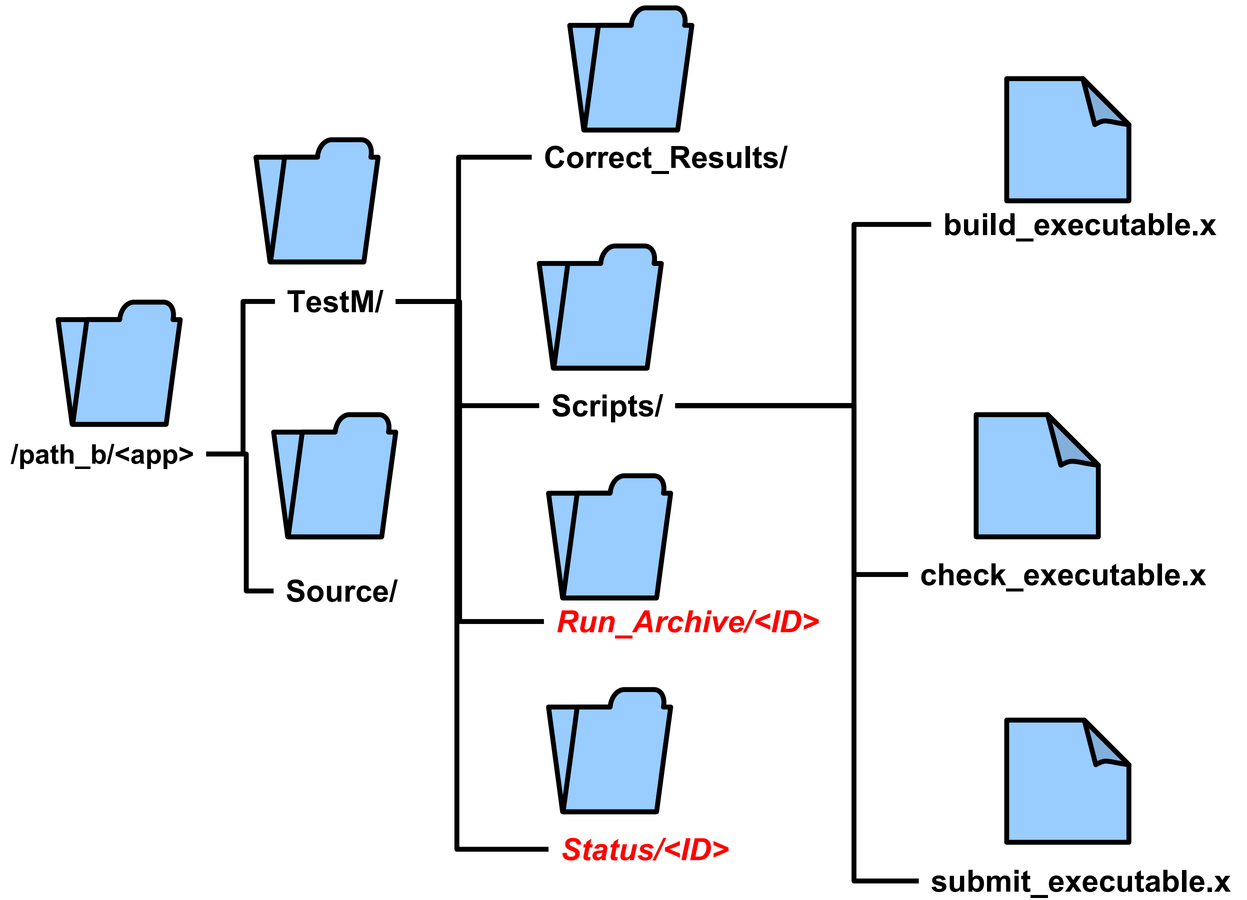

Figure 7: An overview of the directory layout of an application and test during a test instance. For each test instance the harness driver generates a unique id, *ID*, which is the time from epoch. The id is used to make the directories Run_Archive/⟨ID⟩, and Status/⟨ID⟩ respectively contain the critical files to be archived and the results of the test instance.