Build-and-Test Workloads for Grid Middleware: Problem, Analysis, and Applications

Alexandru Iosup and Dick Epema
Delft University of Technology, Faculty EEMCS, Delft University of Technology, NL
Email: {A.Iosup,D.H.J.Epema}@tudelft.nl

Peter Couvares, Anatoly Karp, and Miron Livny
Computer Science Department, U.Wisconsin-Madison, USA
Email: {pfc,akarp,miron}@cs.wisc.edu

Abstract

The Grid promise is starting to materialize today: large-scale multi-site infrastructures have grown to assist the work of scientists from all around the world. This tremendous growth can be sustained and continued only through a higher quality of the middleware, in terms of deployability and of correct functionality. A potential solution to this problem is the adoption of industry practices regarding middleware building and testing. However, it is unclear what good build-and-test environments for grid middleware should look like, and how to use them efficiently. In this work we address both these problems. First, we study the characteristics of the NMI build-and-test environment, which handles millions of testing tasks annually, for major Grid middleware such as Condor, Globus, VDT, and gLite. Through the analysis of a system-wide trace covering the past two years we find the main characteristics of the workload, as well as the performance of the system under load. Second, we propose mechanisms for more efficient test management and operation, and for resource provisioning and evaluation. Notably, we propose a generic test optimization technique that reduces the test time by 95%, while achieving 93% of the maximum accuracy, under real conditions.

1 Introduction

The Grid world is starting to fulfill the promise of a world-scale computing infrastructure for the use of the ever-growing scientific community. Indeed, current systems such as CERN's LCG, the EGEE, the NorduGrid, the TeraGrid, Grid'5000, and the OSG, gather together (tens of) thousands of resources, and offer similar or better throughputs when compared with large-scale parallel production environments [15]. However, the grid paradigm comes with a high price: the problems of the software, in particular, those related to deployability and to core functionality, are much more easily exposed by the dynamicity, the heterogeneity, or simply by the sheer scale of the systems.

The middleware problems are already manifesting in full, with job failure rates in Grids reaching levels from over 10% in controlled environments [14], or 20-45% in a mid-large Grid environment (TeraGrid) without using resubmissions [16], to up to 27% failures, even after 5-10 resubmissions [10]. Deployment success rates are unknown, but grids are notoriously difficult to set-up. A potential solution to the problem of large-scale systems middleware is the adoption of industry practices regarding building and testing, which in light of the failure situation become equally important to designing and developing the middleware.

Throughout development, the middleware must be developed iteratively and incrementally. The middleware needs to be validated through functionality (unit) tests in an environment as close to the target as possible, starting from very early stages. To mitigate development risks, milestones must be clearly defined, and at any moment a distribution package should be available for use (or testing). For all these middleware development goals, a build-and-test environment is required. However, it is unclear what a good build-and-test environment for grid middleware should look like, and how to use it efficiently (the *build-and-test problem*).

Our current work is motivated by the build-and-test problem, which we address as follows. First, we study the characteristics of the NMI build-and-test environment, located at the U. Wisconsin-Madison. The NMI testing facility handles millions of testing tasks annually, for arguably the largest middleware packages in the Grid and otherwise large-scale computing world, e.g., Condor [22], Globus [11], VDT, gLite, BOINC [1], etc. By analyzing the system-wide trace covering the past two years of the NMI operation, we show insights into the load arrival patterns, the load structure, and the performance of a build-and-test environment (Section 3). Then, we propose mechanisms for more efficient test management and operation (Sections 4.1 and 4.2, respectively). Notably, we achieve with generic optimization techniques an 85% time reduction at 5% test accuracy cost, for the given workload. Finally, we present an algorithm and an associate set of tools for build-and-test environment provisioning and setup.

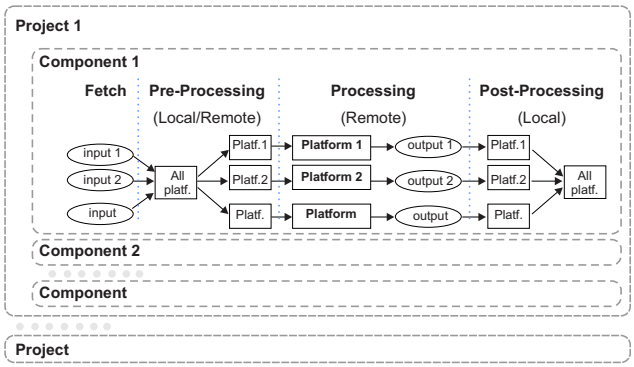

Figure 1. Overview of the The NMI Build-and-Test process for n projects. Only component c_1 of project 1 is detailed.

2 The NMI Build-and-Test Environment

The data used in this work comes from the NMI Build-and-Test Laboratory [21], located at the University of Wisconsin-Madison. In this section we describe this environment, emphasizing on the way it has been designed to address the challenges of the build-and-test problem.

The NMI Laboratory comprises over 100 physical nodes, effectively hosting over 40 different platforms (CPU architecture, and operating system and version combinations). The Laboratory offers access to its nodes through the Condor high-throughput distributed batch computing system [22]. Working on top of Condor is the NMI middleware, which automates the building and the testing of (distributed computing) software in a distributed computing environment.

A *Project* is a set of applications (*Components*) that need to be built and tested. To build and test an application, users explicitly define the workflow of build-and-test tasks, and specify the target platforms on which the workflow is to be run. The workflow description includes not only the specific build-and-test tasks, but also the additional steps that fetch code from existing repositories, and that download, compile, and install external software dependencies, etc. Workflow tasks have inter-dependencies, and may contain several sub-tasks, which in turn may have precedence constraints. A workflow can be of types BUILD (related to building a Component), TEST (testing a Component), or UNKNOWN (other workflows). A *Run* is a workflow execution (instance), which typically comprises several data fetch commands, pre-processing tasks, jobs executed on remote hosts (different platforms), and post-processing tasks (see Figure 1). A *Task* is an individual schedulable unit of a Run. A *Test Job* is a Task that is executed for actual testing, and not for the test setup. For example, fetching source code from the CVS, or the tasks containing sub-tasks are not Test Jobs. A Task that performs unit testing for a module of some Component is a Test Job. Depending on the test setup preferences, a Run fails if one, several, or all of its Tasks fail.

The NMI Build and Test software stores the workflow definition information in a central repository, to ensure every build or test is reproducible. Build or test runs are dynamically deployed to the appropriate computing resources for execution. Users can view the status of their routines as they execute on build-and-test resources. The framework transfers automatically the output produced during the execution to a central repository. Authorized users can pause or remove their routines from the framework at any time.

Currently, the NMI Build-and-Test Laboratory at U.Wisconsin-Madison serves projects such as: core grid middleware (e.g., Condor [22], Globus [11]), grid packages (e.g., VDT[1], gLite, OMII[2]), file and data transferring software (e.g., GridFTP, DataCutter, Replica Location Service (RLS [7]), SRB, UeberFTP), monitoring software (e.g., Network Weather Service (NWS), INCA), and problem solving environments (e.g., APST [5], BOINC [1]).

3 Workload Analysis

In this section we present the analysis of the NMI Laboratory workload.

3.1 The Build-and-Test Workload

We have obtained a system-wide trace covering the past two years of the NMI environment's operation, which stores information about all the Runs (and their Tasks). A total of over 30,000 Runs, and over 2,000,000 Tasks were recorded from 2004/10/01 until 2006/11/01. Table 1 details the workload's size characteristics. While the BUILD and the TEST workflows have similar numbers of runs, the BUILD workflows have a much lower number of Tasks (they typically just compile, while for TEST workflows a large number of individual unit tests must be executed), and the TEST workflows consume a much lower amount of CPUTime (as BUILD tasks have to wait for slow I/O operations, while not yielding the machine's CPU for some operation). The Top-3 Projects dominate the workload in terms of number of Runs, number of Tasks, and consumed CPUTime. As expected, for the Platforms the CPU consumption is more evenly distributed, as building and testing on as many platforms as possible is an important reason for working with the NMI Laboratory.

3.2 Arrival Patterns

We continue our analysis with a description of the arrival patterns.

[1]The Virtual Data Toolkit (VDT), `http://vdt.cs.wisc.edu/`.

[2]The Open Middleware Infrastructure Institute, `http://www.omii.ac.uk/`

Category	First Record	Last Record	No.Runs (% From Total)	No.Tasks (% From Total)	No. Users	CPUTime [Years]	No. Hosts
Total	2004-09-14	2006-10-31	34951(100.00)	2406335(100.00)	54	89.57	122
Per run type							
BUILD	2004-09-14	2006-10-31	16114(46.10)	623435(25.91)	50	54.39	119
TEST	2004-09-17	2006-10-31	18490(52.90)	1775611(73.79)	34	33.39	90
UNKNOWN	2004-09-14	2006-08-24	347(0.99)	7289(0.30)	14	1.78	31
Per project (rank)							
condor (1)	2004-09-14	2006-10-31	21312(60.98)	2029276(84.33)	29	54.91	91
TG (2)	2005-05-04	2006-10-31	847(2.42)	127171(5.28)	2	16.02	40
VDT (3)	2004-10-18	2006-10-31	2438(6.98)	72500(3.01)	11	8.28	52
nmi (4)	2004-09-14	2006-09-02	2014(5.76)	77249(3.21)	9	2.93	47
BOINC (6)	2005-12-20	2006-10-31	302(0.86)	32888(1.37)	1	1.55	57
Per platform (rank)							
X86/Linux-RH/9 (1)	2004-09-14	2006-10-31	9072(25.96)	223982(9.31)	50	8.73	16
HP/HPUnix/10 (2)	2005-04-07	2006-08-09	1967(5.63)	108097(4.49)	24	6.71	5
Sun/Solaris/5 (3)	2004-09-14	2006-10-31	6245(17.87)	237981(9.89)	35	6.68	11
PowerPC/AIX/5 (6)	2004-11-04	2006-10-31	4587(13.12)	120630(5.01)	35	3.9	9
IA64/Linux-RH-AS/4 (8)	2005-09-07	2006-10-31	4712(13.48)	24516(1.02)	18	3.04	5

Table 1. The size characteristics of the NMI Build-and-Test workload. Both Projects and Platforms are ranked by the consumed CPUTime. Note that not all Projects or Platforms are displayed.

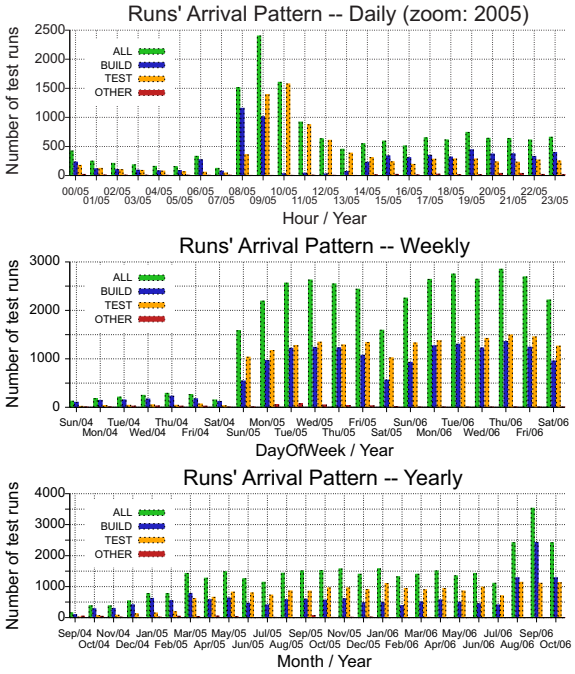

Figure 2. The yearly, weekly, and daily arrival patterns of the test runs, per run type.

Figure 2 depicts the yearly, weekly, and daily Runs' arrival patterns. For a large contiguous part of the data, e.g., for the period between March 2005 to July 2006, the arrivals level remains almost constant throughout the year, with the exception of July, which is a slow month in both 2005 and 2006. Three submission intensities can be observed on the yearly arrival pattern: low, until March 2005, medium, from March 2005 until August 2006, and high, from August 2006 on. Such levels of intensity occur when the Project to which the testing process is associated evolves; here, the main Project grew in approximately one-year steps. The number of submissions increases towards the mid-week, to decrease then towards the week's end. The high-demand part of the day occurs between 09:00 GMT and 10:00 GMT, with the peak part of the day occurring between 08:00 GMT and 22:30 GMT. Note that the local time is GMT-8, for an expected nocturnal, tool-driven, environment. Similar patterns to the global ones can be observed for each of the two major Run types, i.e., Build and Test.

Figure 3 shows a comparative view of the Runs' and Tasks' daily arrival patterns, throughout the whole year 2005. The arrival patterns are similar, but there is a delay of about two hours between the spikes observed for the arrival of Runs, and those observed for the arrival of the Tasks. We ascribe this phenomenon to BUILD Runs (low number of Tasks) being almost always followed by TEST Runs (relatively high number of Tasks). This is confirmed by the breakdown of the type Runs' daily arrival patterns in Figure 2: between 08:00 and 09:00 GMT almost all the Runs are of the type BUILD, from 09:00 to 9:30 GMT the BUILD and TEST Runs are equally present, and from 09:30 to approx. 14:00 GMT the TEST Runs are predominant.

3.3 Individual Workflow Structure

We now detail the structure of individual Runs.

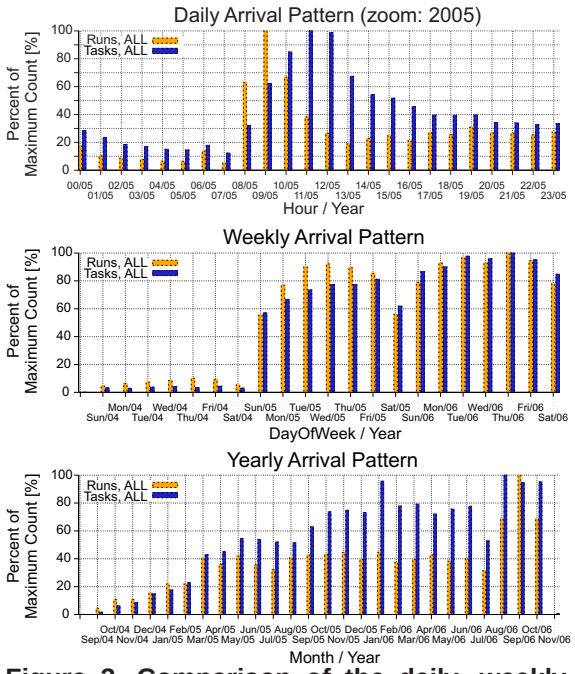

Figure 3. Comparison of the daily, weekly, and yearly arrival patterns of the test runs and tasks.

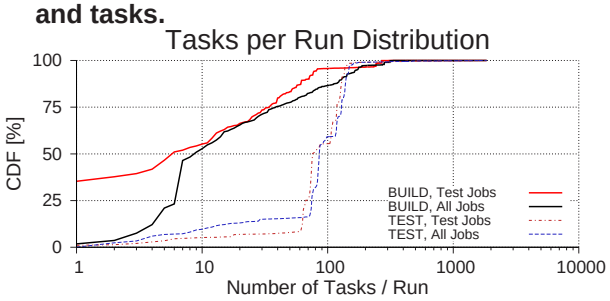

Figure 4. The distribution of the number of Tasks per Run, per Run type. Test Jobs have been introduced in Section 2.

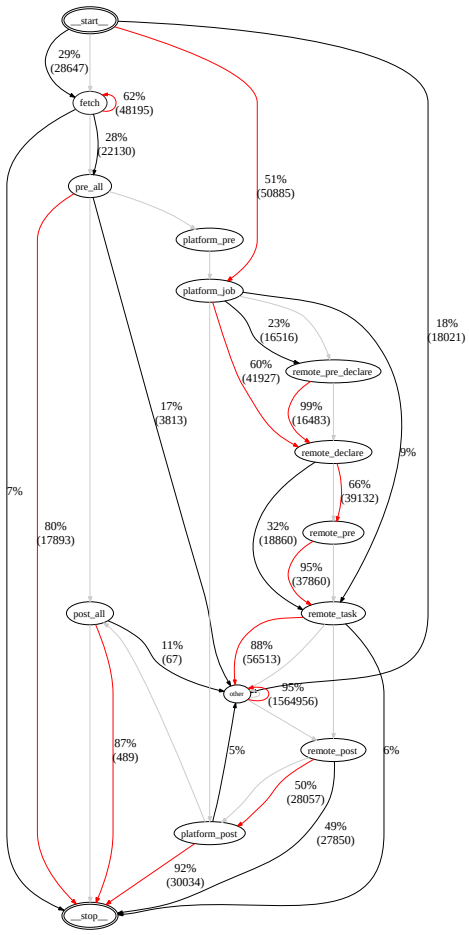

Figure 5. Transition graph for the whole workload. The thin light-gray line shows the transitions, as expected from the NMI workflow definition.

Figure 4 depicts the distribution of the number of Tasks per Run, per Run type. Overall, the average number of Tasks for a BUILD Run is 39 (the standard deviation is 61.3); for a TEST Run, the average number of Tasks is 96 (the standard deviation is 75.3). When considering only Test Jobs, the values follow the same distribution, but are slightly lower.

Figure 5 shows the transition graph averaged over the whole workload. Only transitions with a probability over 10% are displayed. For each node, the highlighted path is composed from the most likely transitions. After entering the system (note `__start__`), a Run is likely to start with a `platform_job`, a composite Task which also contains Test Jobs (the main purpose of the Run's execution, see Section 2). Then, after several more pre-setup Tasks (nodes `remote_declare` and `remote_pre`), a `remote_task` composite Task is called, which in turn calls almost always a string of Test Job (node `other`), which are executed sequentially. When the sequence is shown, with high probability either another `platform_job`, or a post-setup `platform_post` Tasks are executed. The relative chances of `platform_job` vs. `platform_post` are 2:5. Note that neither transition is depicted on the transition graph, as the probability of each is less than 5% (the probability of a Test Job node to transit to itself is 95%). The Run ends and exits the system with a transition to the `__stop__` node.

Figure 6 shows the histogram of the number of components per project (e.g., for each number of components per project, the vertical axis represents the number of occurrences (test runs) having this ratio of components for a given project). Note the use of the logarithmic scale for the vertical axis. We observe that the distribution of the number of components per project looks like a heavy-tail: most of the projects have only one component, 10 projects have 2 to 4 components each, and that the remaining 5 projects have from 5 to 56 components, without the same components per

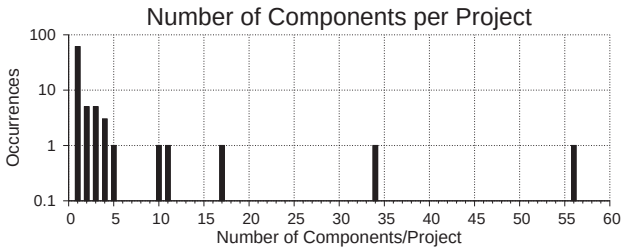

Figure 6. Histogram of the number of components per project.

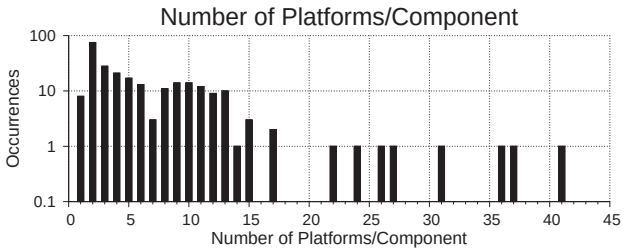

Figure 7. Histogram of the number of platforms per component.

project value repeating.

Figure 7 shows the histogram of the number of platforms per component (e.g., for each number of platforms per component, the vertical axis represents the number of occurrences (test runs) having this ratio of platforms for a given component). Note the use of the logarithmic scale for the vertical axis. Different from the components per project histogram, there is a wide spread of values, with a majority of components being tested on at most 13 platforms. The condor project's main component (condor), is built on the largest number of platforms possible, 41. Components of BOINC and Globus/TG are being built on 26 and 17 platforms, respectively. Note that the Globus toolkit is also being built on other platforms, but by independent projects.

3.4 Correlations Between Characteristics

In this section we investigate the existence of correlations between the characteristics of the workload. We look in particular at the potential correlation between the presence of failures and (i) the duration of the test runs, (ii) the platform where the test tasks are executed.

Figure 8 shows the correlation between the presence of failures and the duration of the test runs. Each point at coordinates (x, y) represents the existence of at least one successful run (dark-colored circles) or that of at lest one failed run (light-colored triangles) at time x, with the duration of the run equal to y hours. We observe that longer runs fail more often. For the workload under study, runs longer than 1000 hours (above the dotted line in Figure 8) always fail. We use this result to optimize the test process, in Section 4.2.

Figure 8 shows the lack of correlation between the test

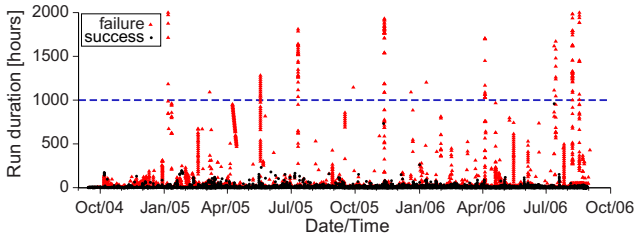

Figure 8. Correlation between the number of observed failures and the duration of the test runs. Longer runs fail more often.

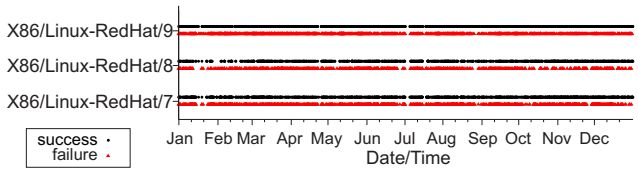

Figure 9. Lack of correlation between the test outcome and the platform where test tasks run. Only data for year 2005 is shown.

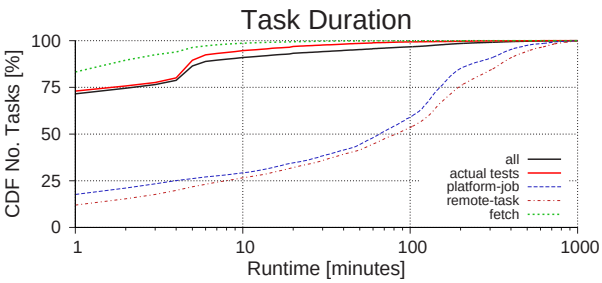

Figure 10. CDF of the tasks' runtimes, per task category.

outcome and the platform where test tasks run. For each platform (vertical axis), each point at horizontal coordinate x represents the existence of at least one successful run (dark-colored circles) or that of at lest one failed run (light-colored triangles) at time x. We only show data for one year (2005) for three platforms which differ only in the version of their OS. While tests were conducted in parallel for all three platforms, there is no significant difference in the occurrence of successful and of failed runs per platform. We have investigated all the other platforms present in the workload, and obtained similar results.

3.5 Build-and-Test Performance

We look now at two performance indicators: the duration of the individual Tasks, and the number of discovered errors.

Figure 10 shows the cumulative distribution function of the tasks' runtime, per task category. For the horizontal axis, note the logarithmic scale; the time unit is 60 seconds (1 minute). To eliminate flurries, we consider all task runtimes exceeding 1000 time units as 1001 time units. Most

Category	Failed Runs All (% runs)	Failed Tasks All (% tasks)	Failed Tasks Machine (% All)
Total	37.99%	5.89%	13.12%
Per run type			
BUILD	22.06%	3.48%	10.32%
TEST	15.31%	2.35%	17.25%
Per project			
Project Rank 1[1]	19.64%	2.87%	16.54%
Project Rank 2[1]	2.12%	1.30%	9.97%
Project Rank 3[1]	4.38%	0.47%	11.03%
Project Rank 4[1]	3.71%	0.54%	7.52%
Project Rank 6[1]	0.74%	0.25%	5.55%
Per platform (rank)			
Platform Rank 1[2]	13.07%	0.59%	13.45%
Platform Rank 2[2]	3.21%	0.38%	5.57%
Platform Rank 3[2]	15.00%	0.92%	11.14%
Platform Rank 6[2]	7.64%	0.31%	14.09%
Platform Rank 8[2]	5.89%	0.35%	8.98%

[1],[2] See Table 1 for the actual project and platform names.

Table 2. Summary of the observed failures for the NMI Build-and-Test workload.

test jobs (category 'actual tests') take less than 5 minutes, with an average of about 4 minutes (the standard deviation is 24.4). The jobs that retrieve data and/or source code (category 'Fetch') are usually very short, with an average of about 1 minute (the standard deviation is 9.2).

Table 2 details the observed failures. The *Failed Runs, All* column shows the percentage of the number of Runs that failed, from all the Runs. The number of failed Runs overall (row *Total*) is around 40%, which underlines the critical importance of the build-and-test system. The BUILD Runs fail more than the TEST Runs, both in absolute terms (22% of the total number of Runs are failed BUILD Runs, whereas below 16% of the total number of Runs are failed TEST Runs), and relative terms (the BUILD Runs are slightly fewer than the TEST Runs, yet they yield more failures). The most thoroughly tested project, Condor, reveals the highest absolute percentage of failed Runs, about 20% of the total number of Runs (note that for Condor a Run fails if *any* of its Tasks fails), but a much lower relative percentage, as the Condor Runs represent over 60% of the workload (see Table 1). The *Failed Tasks, All* column shows the percentage of the number of Tasks that failed, from all the Tasks. A surprisingly low amount of Task errors shows again the importance of the build-and-test environment: when a software package needs to be shipped, it must be functioning correctly under all predicted platforms or uses cases; a small amount of failures, revealed only under cross-platform testing, results in a high number of failed Runs (overall tests). The *Failed Tasks, Machine* column shows the number of failures due to machine unavailability or crashes, relative to the total number of failed tasks. The percentage of failures due to the testing environment is below 20%.

Figure 11. The daily pattern for run and for task failure occurrence. Note the different scale for the right part of the graphic.

4 Applications

Throughout this section, we focus on applications of the build-and-test environment analysis. We assume an incremental development process [3, 4], and we focus on test management, test optimization, and environment provisioning and setup issues. We point out that a more in-depth treatment of these problems is beyond the scope of this work, as the build-and-test area that is rich in research and technical problems.

4.1 Test Management

We investigate here the automated tools that can assist a project manager's decisions. Various performance indicators can be used in practice to estimate how close is the project to a releasable state, to assess the development team's performance, and to manage the test environment (discover faulty machines). We have already introduced in Section 3 a set of analysis tools that characterize the test process, and assess the test process's performance. We add in this section tools for more detailed failure analysis: the occurrence of Run and Task failures, and the observed *mean time to failure* (MTTF) and *mean time to recovery* (MTTR).

Ideally, the project manager would make a shipping decision based on the trends of the number of failures over time (i.e., convergence to 0). A shipping decision can also be taken if the number of observed errors remains at a level below a certain threshold. Figure 11 depicts the daily pattern for run and task failure occurrence. The number of Run failures per day is on average 18 (the standard deviation is 14.0). The number of Task failures per day is on average 183 (the standard deviation is 151.8). A number of outstanding bugs occur daily, but are soon fixed (see also the following discussion on MTTF and MTTR).

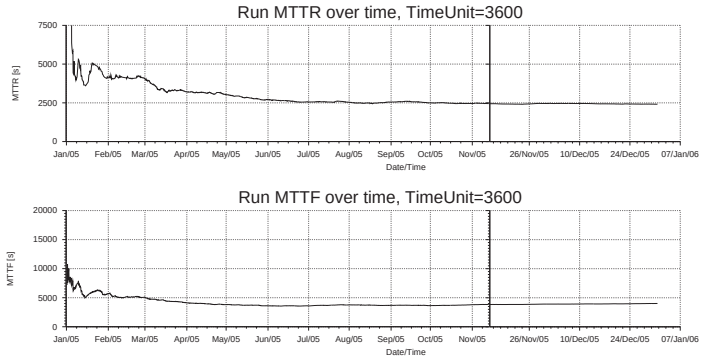

Figure 12. The MTTF and the MTTR for the whole workload, over 2005. Note the different scale for the right part of the graphic.

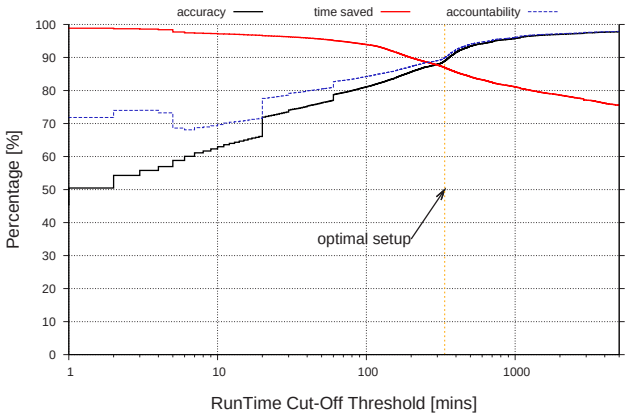

Figure 13. The tradeoff between the tasks' run time and the process outcome: Accuracy, Accountability, and SavedTime vs. the run time cut-off point.

We define MTTF as the average interval between two consecutive failures, and MTTR as the interval between a failed Run (or Task) and the consecutive successful Run (or Task). Figure 12 depicts the MTTF and MTTR for the whole workload, over the year of 2005. The average MTTF is 4013s (cca. $1\frac{1}{2}$ hours); the standard deviation is 9630.47. The average MTTR is 2414s (less than 1 hour); the development team is doing a good work in preventing the long-term existence of important (crash) bugs.

4.2 Test Optimization

The most important optimizations in the testing process concern the time vs. accuracy trade-off. The key question is how to reduce the time needed for testing, while still being able to observe and categorize the failures. While many domain-specific optimization techniques are available, we focus here on a generic optimization technique, by investigating *the tradeoff between the tasks' run time and the process outcome*. We consider a modified test process in which jobs are stopped before their normal finish time, if their runtime exceeds a certain threshold; the result of the stopped jobs is considered to be correct. By stopping the jobs early, the total test time is reduced, at the expense of a lower number of errors observed in the system. This optimization is generic in the sense that it requires only information available to any test process: the duration of the jobs. We aim therefore at answering the question: *If the test tasks are stopped after a certain period, what is the resulting performance?* (from hereon, the *trade-off question*).

We first describe the performance metrics:

$$Accuracy(t) = \frac{NDetected(t)}{NTotal(T)} \times 100[\%] \quad (1)$$

$$Accountability(t) = \frac{\frac{NDetected(t)}{NTotal(t)}}{\frac{NDetected(T)}{NTotal(T)}} \times 100[\%] \quad (2)$$

$$SavedTime(t) = (1 - \frac{UsedTime(t)}{UsedTime(T)}) \times 100[\%] \quad (3)$$

where t is the current system state, e.g., the current time, and T is the final system state, e.g., the end-of-test time. $NDetected(\cdot)$ and $NTotal(\cdot)$ return the number of detected and of total number of failures for the test process up to a given moment of time, respectively.

Accuracy shows how well the test results are predicted under the test conditions (see Eq. 1). The closer the accuracy is to 100%, the better. *Accountability* shows how well the test trends are predicted under the test conditions (see Eq. 2). The closer the accountability is to 100%, the better. We expect the Accountability to converge much quicker to 100% (perfect trends prediction) than the Accuracy. *SavedTime* shows how much of the test time is saved under the modified conditions (see Eq. 3). The closer this is to 100%, the better. There can never be a save of 100%, if any tests are to be performed. Finally, we define the *run-time cut-off point* as the time when tasks are stopped and considered failed (note that no description can be given on this type of failure). Besides answering the trade-off question, we want to also establish the optimal cut-off point (OptCO), that is, the point with the highest combined SavedTime, Accuracy and/or Accountability level. The OptCO depends on the details of the build-and-test workload.

We perform our investigate using the real build-and-test workload presented in Section 2 and analyzed in Section 3). Note that for any time t the values of $NTotal(t)$ are extracted from the input workload, whereas the values of $NDetected(t)$, $SavedTime(t)$, $Accuracy(t)$ and $Accountability(t)$ are computed. OptCO is found to be around 330 minutes, or the equivalent of $5\frac{1}{2}$ hours; the corresponding Accuracy and SavedTime values are of 93%, and 95%, respectively (see Figure 13).

Algorithm 1 Algorithm for generating synthetic Build-and-Test workloads. The steps tagged with $\star$ are optional.

Input:

▷ $2 \times n$, the number of Runs to generate.
▷ D_{1-5}, the distributions depicted in Figures 2, 4, 6, 7, 10 respectively.
▷ D_6, column *Failed Tasks, All*, rows *Per run type*, from Table 2.
▷ TG, the transition graph shown in Figure 5.

Output: A synthetic build-and-test workload.

1: Generate n arrival times for BUILD Runs from D_1.
2: Generate n arrival times for RUN Runs, each 2 hours later than the previously unmatched BUILD Run.
3: **for** each Run r_i **do**
4: Generate the number tasks t_i, from D_2.
5: $\star$ Generate the number of components c_i, from D_3, then foreach component c_j generate the number of platforms $p_{i,j}$, from D_4.
6: $\star$ Split the t_i tasks between all platforms.
7: **for** each Task t_i in a group of tasks **do**
8: Assign the task a type Θ_i, following TG.
9: Assign the task a runtime τ_i, from D_5, using Θ_i.
10: $\star$ Decide if the task would succeed, based on the Run type, and on D_6.

4.3 Test Environment Management

There are many design alternatives when setting up a new build-and-test environment, in the form of hardware, of operating software, of middleware (e.g., a large variety of schedulers), and of software libraries. Using synthetic workloads, the design choices may be compared under realistic load [2, 6, 12]. When a new system is replacing an old one, running a synthetic workload can show whether the new configuration performs according to the expectations, before the system becomes available to users. The same procedure may be used for assessing the performance of various systems, in the selection phase of the procurement process. We propose using a tool like the GRENCHMARK framework [14] for generating and submitting synthetic build-and-test workloads. The GRENCHMARK framework allows its users to plug-in workload generators, and then facilitates the submission and the analysis processes. We therefore focus in the rest of this section on the synthetic generation of build-and-test workloads.

Algorithm 1 provides the means for generating a build-and-test workload. Note that the algorithm is not specifically bound to the data presented in this paper. In the case when the data is not available, the user needs to design the distributions D_{1-6}, and the expected transitions graph TG. Steps 1 and 2 of Algorithm 1 define the arrival times of the test Runs, based on the observations a TEST Run arrives almost always 2 hours after a BUILD Run (see Section 3.2), and that the number of BUILD and TEST Runs are similar (see Table 1). Step 4 fixes the number of Tasks for each Run. Steps 5 and 6 are optional, and should be used only for workloads where the existence of components and platforms is required. Steps 8 and 9 assign the type and the run time of a task. Step 10 assigns whether the task would fail out of its own problems (and not from system failures). This last step is optional: it should be followed only if the investigation for which the workload is generated has steps that depend on the number of failed jobs (e.g., the tests are repeated until less that a fixed number of jobs fail). For example, this step can be skipped the build-and-test workload is used to test whether the tested system can accommodate a certain amount of jobs.

5 Related work

Our work stands at the crossing of two research directions: characterizing workloads and environments of great importance, and performing testing and benchmarking of large-scale software.

The problem of characterizing the workloads from critical environments has received constant attention from both the academic and the industry communities. A significant number of workload and trace analysis papers have dealt with the specifics of request-based (Web) workloads [2, 20], parallel production environments [9, 6, 18], large-scale (grid) computing environments [17, 19, 13]. Here, much effort has been put in proving that realistic workload modeling pays dividends for system-improving work. To the best of the authors' knowledge, ours is the first effort that analyzes the characteristics of a build-and-test workload for the middleware of large-scale computing environments.

The problem of testing and benchmarking large-scale software is a key part of the software engineering discipline. Here, the main question to be answered is what makes a good testing or benchmarking environment [8, 23, 24]. The work of Tian and Palma [23] presents insights into the characteristics of a workload used to test large commercial software products. In grids, efforts have been directed to creating synthetic test suites that operate on grid middleware in real environments, like the GrASP [8, 16], or the GridBench [24] projects. Comparatively, this work shows how a dedicated build-and-test environment is used to effectively control the development and shipping of a large number of software packages for large-scale environments.

6 Conclusion and future work

In this paper we have addressed two problems specific to building-and-testing middleware for large-scale (grid) computing: establishing the characteristics of a build-and-test environment, and improving the efficiency of the build-and-test environments use. To this end, we have first introduced

then analyzed a two-year long trace coming from the NMI Laboratory, which encompasses over 2.4 millions of test tasks. We have established for this environment the overall workload characteristics, the arrival patterns, the individual test workflow structure, and the build-and-test performance. Second, we have proposed mechanisms for more efficient test management and operation, and for resource provisioning and evaluation. Notably, we have proposed a generic test optimization technique that reduces the test time by 95%, while achieving 93% of the maximum accuracy, under real conditions. We have also proposed an algorithm for generating synthetic build-and-test workloads, which shows good promise for test environment design, procurement and setup.

Besides their quantitative value, our results uncover an area that is rich in research and technical problems. We plan to continue investigating generic and grid-specific optimization mechanisms for the testing process, and to extend the NMI Laboratory capabilities, especially in the direction of automated management. Last but not least, we intend to make use of this infrastructure for building and testing our own Grid and P2P middleware.

Acknowledgements

This work was carried out in the context of the Virtual Laboratory for e-Science project (`www.vl-e.nl`), which is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W), and which is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ).

We would like to thank the anonymous reviewers, and to Becky Gietzel and Greg Thain, for their contribution to this paper.

References

[1] D. P. Anderson. Boinc: A system for public-resource computing and storage. In R. Buyya, editor, *GRID*, pages 4–10. IEEE Computer Society, 2004.

[2] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. of ACM SIGMETRICS*, pages 151–160. ACM Press, 1998.

[3] V. R. Basili and A. Turner. Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, 1(4):390–6, 1975.

[4] K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley, Boston, MA, USA, 2000.

[5] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The apples parameter sweep template: User-level middleware for the grid. In *SuperComputing*, 2000.

[6] S. J. Chapin, W. Cirne, D. Feitelson, U. Schwiegelshohn et al. Benchmarks and standards for the evaluation of parallel job schedulers. In *JSSPP*, volume 1659 of *LNCS*, pages 67–90, 1999.

[7] A. L. Chervenak, Ewa Deelman, et al. Giggle: a framework for constructing scalable replica location services. In *SuperComputing*, pages 1–17, 2002.

[8] G. Chun, H. Dail, H. Casanova, and A. Snavely. Benchmark probes for grid assessment. In *IPDPS*. IEEE Computer Society, 2004.

[9] A. B. Downey. A parallel workload model and its implications for processor allocation. In *HPDC*, pages 112–126, 1997.

[10] C. Dumitrescu, I. Raicu, and I. T. Foster. Experiences in running workloads over grid3. In H. Zhuge and G. Fox, editors, *GCC*, volume 3795 of *LNCS*, pages 274–286, 2005.

[11] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *J. of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[12] E. Frachtenberg and D. G. Feitelson. Pitfalls in parallel job scheduling evaluation. In *JSSPP*, volume 3834 of *LNCS*, pages 257–282, 2005.

[13] A. Iosup, C. Dumitrescu, D. H. Epema, H. Li, and L. Wolters. How are real grids used? The analysis of four grid traces and its implications. In *GRID*, pages 262–270. IEEE Computer Society, 2006.

[14] A. Iosup and D. H. J. Epema. Grenchmark: A framework for analyzing, testing, and comparing grids. In *CCGRID*, pages 313–320. IEEE Computer Society, 2006.

[15] A. Iosup, Alexander Papaspyrou, et al. On grid performance evaluation using synthetic workloads. In E. Frachtenberg and U. Schwiegelshohn, editors, *JSSPP*, LNCS. Springer, 2006. (in print).

[16] O. Khalili, Jiahue He, et al. Measuring the performance and reliability of production computational grids. In *GRID*. IEEE Computer Society, 2006.

[17] H. Li, D. L. Groep, and L. Wolters. Workload characteristics of a multi-cluster supercomputer. In *JSSPP*, volume 3277 of *LNCS*, pages 176–193, 2004.

[18] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput.*, 63(11):1105–1122, 2003.

[19] E. Medernach. Workload analysis of a cluster in a grid environment. In *JSSPP*, volume 3834 of *LNCS*, pages 36–61, 2005.

[20] D. Menasce, V. Almeida, R. Fonseca, and M. Mendes. A methodology for workload characterization of e-commerce sites. In *Proc. of ACM Conference on Electronic Commerce (EC)*, pages 119–128, 1999.

[21] A. Pavlo, P. Couvares, R. Gietzel, A. Karp, I. D. Alderman, and M. Livny. The NMI build and test laboratory: Continuous integration framework for distributed computing software. In *The 20th USENIX Large Installation System Administration Conference (LISA)*, Dec 2006. (accepted).

[22] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[23] J. Tian and J. Palma. Test workload measurement and reliability analysis for large commercial software systems. *Annals of Software Engineering*, 4:201–222, 1997.

[24] G. Tsouloupas and M. D. Dikaiakos. GridBench: A workbench for grid benchmarking. In P. M. A. Sloot, A. G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *EGC*, volume 3470 of *LNCS*, pages 211–225. Springer, 2005.