Proceedings of the Thirty-Sixth Australasian Computer Science Conference (ACSC 2013), Adelaide, Australia

Parallel Execution of Prioritized Test Cases for Regression Testing of Web Applications

Deepak Garg  Amitava Datta
School of Computer Science and Software Engineering
The University of Western Australia,
Perth, Western Australia 6009,
email: deepak@csse.uwa.edu.au
email: amitava.datta@uwa.edu.au

Abstract

We present a new approach for automatically prioritizing and distributing test cases on multiple machines. Our approach is based on a functional dependency graph (FDG) of a web application. We partition the test suite into test sets according to the functionalities and associate the test sets with each module of the FDG. The high priority modules and their associated test sets are then distributed evenly among the available machines. Moreover, we further prioritize the test cases by using inter-procedural control flow graphs within individual functional modules. Our suggested approach reduces the test suite execution time and helps in detecting the faults early in a regression testing cycle. We demonstrate the effectiveness of our technique through an experimental study of a web application and measuring the performance of our technique by using the well known APFD metric.

Keywords: Regression testing, Test case prioritization, Web applications, Parallel execution

1 Introduction

Web applications typically use a complex and multi-tiered, heterogeneous architecture including web servers, application servers, database servers and client interpreters. Web applications undergo maintenance at a faster rate than any other software system (Elbaum et al. 2003), as new functionalities are introduced depending on user requirements. The main aim of testing of a web application is to detect faults in the required services and functionalities and fix these faults, to enable it to function according to specifications (Lucca & Fasolino 2006). A test suite for testing a web application consists of many test cases. The serial execution of a test suite on a single machine might take many hours (Lastovetsky 2005) depending on the size of an application, the machine where the test suite is run and its workload.

For every new version of a web application released, regression testing is required to test the compatibility of the new features with the previously tested functionalities. New test cases are generated to perform regression testing. Since web applications typically have a rapid turn around time, it becomes very difficult to execute all the test cases within a specified amount of time. The cost of re-running all test cases may be expensive and not always useful as sometimes only selected functionalities need to be tested. Hence, test cases are usually prioritized during testing in order to discover the likely vulnerable parts of the code early so that developers have more time to identify and debug the faults. Many strategies have been proposed for prioritizing C (Elbaum et al. 2000) (Elbaum et al. 2002) and Java programs (Harold et al. 2001) (Do et al. 2006) (Jeffrey & Gupta 2006). For web applications, Sampath et al. (Sampath et al. 2008) suggested prioritization techniques using user-session based test cases.

Even though prioritization mitigates some of the drawbacks of executing a complete set of test cases, execution of test cases on a single machine may not achieve the rapid testing criterion of large web applications. Also, most organizations can afford to deploy multiple machines for testing. Hence, a possible way of expediting the speed of regression testing is to run disjoint parts of the same test suite on multiple machines. In this paper, we investigate the parallel execution of prioritized test cases. We first construct a functionality dependency graph (FDG) of the entire web application from its UML specification. A node in the FDG is a unique functionality and a directed edge from node m to node n indicates the functional dependency of node n on node m. We partition the entire test suite into test sets such that each test set is associated with a unique functional module in the FDG.

Next, we identify a subgraph S of the FDG for prioritization. This subgraph consists of the nodes that have been modified after the previous regression testing cycle and nodes that are dependent on these modified nodes. We assign priorities to the nodes of S. The test sets are executed according to these priorities when a single machine is used for testing. For multiple machines, we sort the nodes of S in priority order and allocate nodes from different priority groups approximately evenly among the available machines. We further prioritize the execution of test sets in each machine by using code level control flow graphs (CFG). We execute the test sets allocated to each machine simultaneously in parallel. Finally, we collect the test results in a single machine. In this paper, we suggest our approach using web applications but our approach is quite general and can be used for any kind of software application if we can construct a functionality dependency graph for a software system.

Our main contributions in this paper are: 1. Detection of modified functional modules in web application. 2. Prioritization of the test cases using FDG and CFG. 3. Partitioning of the prioritized test suite
into different test sets for parallel execution on different machines. The rest of the paper is organized as follows. Section II presents the background study related to prioritization techniques and parallel execution of test cases. Section III presents the work related to generation of functional test cases from UML Activity diagrams and the generation of the functionality dependency graph. Section IV describes our approach that includes partitioning of test suite into test sets, prioritization framework, collecting test sets into groups for distributing to individual machines, parallel prioritization and parallel execution of test cases. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based on a dependency graph. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based on a dependency graph. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based on a dependency graph. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based on a dependency graph. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based on a dependency graph. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based on a dependency graph. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based on a dependency graph. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based on a dependency graph. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based on a dependency graph. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.

2 Background

While there are many possible goals of prioritization, this paper focuses on the goal of reduction of test suite execution time and early detection of faults. While there are many prioritization techniques available for software testing, very few are available for web application testing. Rothermel et al. (Rothermel et al. 2001) first defined the problem of test suite prioritization. Srikanth et al. (Srikanth et al. 2005) suggested a cost effective test case prioritization technique that improves quality of software by considering defect severity. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code. They called this new approach as PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based on a dependency graph. Section V describes the experimental evaluation. The results are presented in Section VI and the conclusions are presented in Section VII.
3.2 Functionality and Functional Modules

Functionality refers to user actions such as keyboard and mouse events required to navigate through web applications (Sampath et al. 2007) (Di Luca et al. 2003). UML sequence diagrams provide information about functionality and the interaction among different objects in a web application (Cartaxo et al. 2007). Functionality is defined in the specification documents and is provided by the user interface (Heinecke et al. 2010). A functional module is a collection of different functions (at the code level) to enable the functionality to behave according to the specifications.

3.3 Functionality dependency graph

A FDG (Zimmermann & Nagappan 2007) (Samuel et al. 2005) is a directed graph that is used to describe the relationship between functionalities in web applications. For an FDG $G = (V,E)$, the node set $V$ is the set of functionalities in a web application and $E$ is the set of directed edges that represents the dependency relationship among the functionalities. A directed edge from $m \in V$ to $n \in V$ indicates the functional dependency of the module $n$ on module $m$.

![Figure 1: Control Flow Graph (CFG) for the Registration node in the FDG of the Online Bookstore application.](image)

3.4 Control Flow Graph

We assume that each functionality is composed of a class or a combination of classes in the source code of a web application. We extracted CFG for every functional module in the FDG of the Online Bookstore\(^1\) application using the algorithm by Rothermel et al. (Rothermel et al. 2000). Each module in a CFG is either a C# or an ASPX (Active Server Pages Extended) source code module. We show an example CFG in Fig. 1.

4 Our approach

We partition the complete test suite into test sets, each test set is associated with a unique functional module or node in the FDG. We prioritize the test cases within each test set using the CFG of the corresponding functional module.

![Figure 2: The Functionality Dependency Graph (FDG) for the Online BookStore application.](image)

### 4.1 Partitioning of Test Suite

We extract the FDG of the Online Bookstore application from its functional specifications based on our previous work (Garg & Datta 2012). We captured various functional requirements using UML. We captured the interactions between the requirements. We manually extracted the FDG for the Online Bookstore application from the functional specifications as shown in Fig. 2 but depending upon the technologies, other methods of generating FDG could be applied. Fig. 2 shows the various functional modules for this web application.

We generate test cases for all these functional modules and initially randomly store them in a test suite. We partition the entire test suite into test sets, such that each test set is associated with a unique functional module. We automatically identify the test cases that are associated with a particular functional module by reading the test cases and matching the statements. If a test case contains the statements that are required to test some functionality $F_i$, we store that test case separately in a test set $T_i$ that is associated with $F_i$. This computation is done in a single machine that we call as the test server.

### 4.2 Prioritization of test cases for each functional module

Rothermel et al. defined the problem of test suite prioritization in (Rothermel et al. 2001). Given $T$ as a test suite, $P$ is the set of all test suites that are the prioritized orderings of $T$ obtained by permuting the tests of $T$ and $F$ is a function obtained from $P$ to the

---

\(^1\)available freely at www.gotocode.com
reals, the problem is to find a permutation, \( T' \in P \) such that \((\forall T'')(T'' \in P)(F(T'') \geq F(T''))\).
Given a functional module \( F_i \) (a node in the FDG), we prioritize the test cases in the corresponding test set \( T_i \) according to the CFG of \( F_i \). The modules with the dotted circles in Fig. 3 refer to the modified source code modules of the Online Bookstore web application. Each code module in a CFG is associated with a test case or series of test cases. The modified code modules in CFG are executed in priority compared to the unmodified modules. Our prioritization strategy is as follows:

- The modified modules closer to the root or the class definition are given highest priority in execution of test cases in a test set.
- If two modified modules are at the same level in CFG, the modules having more dependent modules will be given priority in execution.
- We randomly order the test cases belonging to the remaining unmodified modules.

Note that, this prioritization is done for all the functional modules in the FDG after we generate the test cases in the test suite. As all the functional modules in the FDG may be tested in different regression test cycles, we need to prioritize the test cases for each module using the corresponding CFG. This computation is done in the test server.

![Control Flow Graph (CFG) with modified modules of the Registration functional module of the Online Bookstore application.](image)

Figure 3: Control Flow Graph (CFG) with modified modules of the *Registration* functional module of the *Online Bookstore* application.

### 4.3 Extraction of the affected subgraph from the FDG

We recall that a directed edge from node \( m \) to node \( n \) in the FDG indicates that node \( n \) is dependent on node \( m \). We also say node \( m \) invokes node \( n \). We assign priorities to the nodes of the FDG in the following way:

1. A newly introduced node in the FDG is given the highest priority. If there are multiple newly introduced nodes, they are assigned the highest priorities in an arbitrary order.
2. The modified nodes in the FDG are given the next lower priorities.
3. The next lower priorities are assigned to the nodes that directly or indirectly invoke the modified or newly introduced nodes. A higher priority among these nodes is assigned to a node that is closer (in terms of path length) to a newly introduced or modified node. In case of multiple paths, the node that is following the shortest path to a newly introduced or modified node will be given priority.
4. All other nodes (except the nodes that are invoked by the modified or newly introduced nodes) are assigned the next lower priority in an arbitrary order.
5. The nodes that are invoked either directly or indirectly by the modified or newly introduced nodes are assigned the least priorities. These nodes have been tested in the previous regression test cycles and they are unchanged. Hence we assume that they need not be tested in the current test cycle. Hence these nodes are called unaffected nodes. All other nodes are called affected nodes.

We use the following observation for extracting the affected subgraph of the FDG.

**Lemma 1** The affected nodes in the FDG form a connected subgraph of the FDG.

**proof 1** Our prioritization scheme ensures that every directed path from the root to a leaf of the FDG has all the affected nodes as a connected sub-path. We prove this by contradiction. Consider three consecutive nodes \( m, n \) and \( p \) on a root to leaf directed path such that \( m \) and \( p \) are affected but \( n \) is not affected. Such a situation will make the affected subgraph disconnected. However, \( p \) is affected and \( n \) invokes \( p \) and hence, our prioritization scheme ensures that \( n \) is also affected, a contradiction.

We extract the affected subgraph \( S \) of the FDG by performing a depth-first search. The depth-first search picks up the subgraph containing only the affected nodes, as the search backtracks whenever it encounters an unaffected node. The search returns with the affected subgraph due to Lemma 1. The nodes in \( S \) are distributed to the available machines in our parallel prioritization scheme. This is discussed below. This computation is done in the test server.

### 4.4 Allocating functional modules to machines

The test server allocates the functional modules to the machines participating in the parallel test execution. It is easy to distribute the test sets for the functional modules to different machines if the number of functional modules \( F \) is less than or equal to the number of available machines \( M \). Each machine can be allocated the test set of one functional module. However, realistically complex web applications consist of a large number of functional modules and
$F$ is usually much greater than $M$. Hence, we need to allocate multiple functional modules and their associated test sets to each available machine. We construct subsets of functional modules in such a way that each machine is allocated approximately an equal number of functional modules as well as the priorities of the different functional modules in each subset are also approximately equal.

We construct the subsets in the following way. Each node in the selected subgraph $S$ of the DFG has a priority associated with it. We sort the nodes according to these priorities and store in an array $A$. From the discussion in Section 4.3, the nodes can have four different priorities $p_i, 1 \leq i \leq 4$. We denote the number of nodes with priority $p_i$ as $|p_i|$. We allocate $\lceil \frac{|p_i|}{M} \rceil$ (1 $\leq i \leq 4$) nodes from the priority class $p_i$ to each of the $M$ machines. Though it is possible that the last machine is allocated less than $\lceil \frac{|p_i|}{M} \rceil$ nodes, nodes from each priority class are approximately evenly distributed among the $M$ participating machines.

### 4.5 Parallel execution strategy

The functional modules and their associate test sets are allocated to the participating machines by the test server according to the strategy discussed in Section 4.4.

Fig. 4 shows the setup of the entire test process. The main machine acts as a hub and stores all the test case execution data and behaves like a test server. The other computing nodes execute the allocated test sets in parallel. Each machine stores the test execution results and these results are sent to the test server for constructing the combined test report.

5 Experimental evaluation

In this section, we discuss our experimental set up using our suggested approach.

To perform the experimental evaluation, we extended some of the functionalities of this original application to make it more complex. This application is an online shopping portal for buying books. This application uses ASPX for its frontend and MySQL for its backend connectivity. The application allows the users to search for books by different keywords, add to the shopping cart and proceed to orders.

We randomly seeded 20 faults in various modified functionalities of the Online Bookstore. The faults were assumed to have similar cost levels. Three different kinds of faults (Guo & Sampath 2008) were seeded in the application: Logical Faults, Form Faults and Appearance Faults. A logical fault in the program code relates to business logic and control flow e.g., if the user inputs the same string for the password and confirm password fields, still the application displays the error message Password and Confirm Password fields don’t match. Form faults in the program code modifies and displays name-value pairs in forms. Appearance faults controls the way in which a web page is displayed. We seeded faults in the various modified functionalities like Registration, Members, My-Info, Login and Books and assumed that the faults behave like real faults.

We used C# to implement our proposed approach. We generated 130 test cases from the UML Activity diagrams for the Online Bookstore and converted them to C# test scripts readable by the Selenium test tool for automatic test suite execution (Torsel 2011). The generated test cases were assumed to be non-redundant and were generated according to the functional specifications.

We partitioned the test suite into different test sets and associated these test sets with the different functional modules. Each test set associated with a functional module is composed of test cases related to that specific functionality. The Online Bookstore application has 14 functional modules (Fig. 2) and hence the test suite was divided into 14 test sets. Though our framework is general and can be applied when only one of the functional modules are modified, we modified all the 14 functional modules to evaluate the worst-case performance of our parallel prioritization scheme. Hence according to the discussion in Section 4.3, all the nodes had the highest priority. We used three computers for running the tests. The first two computers were allocated five functional modules each and the last was allocated the remaining four modules. The test cases were executed in parallel on their respective machines. We performed the experiments using Selenium Grid (Brüns et al. 2009). For comparing our results with a random distribution of test cases, we also randomly distributed the test cases among the three machines and executed them in parallel.

Rothermel et al. (Rothermel et al. 2001) presented the APFD (Average Percentage of Faults Detected) metric for measuring fault detection rates of test suites in a given order. APFD values range from 0 to 1; higher numbers imply faster (better) fault detection rates (Elbaum et al. 2001).

APFD can be calculated using the following formula:

$$\text{APFD} = 1 - \frac{TF_1 + TF_2 + TF_3 + \ldots + TF_n}{mn} + \frac{1}{2n}$$

$TF_i$ is the position of first test in $T$ that exposes fault $i$, $n$= no. of faults $m$= no. of test cases

Informally, APFD measures the area under the curve that is plotted by the percentage of faults detected by prioritized test case order and the test suite fraction.

We collected the test execution results and used the APFD metric to determine whether our approach detects faults earlier and faster compared to the random ordering of the test cases. We applied the APFD metric separately for the test cases run on each of the three participating machines.

**Threats to Validity:** We used three different machines with different hardware configurations to execute the test cases in parallel. All these machines had different workload conditions when we were running our experiments by publishing the web application on the virtual web server of the school. We noticed that the test execution times differed when
we repeated our experiments. We manually seeded the faults in the web application. The faults may not be evenly distributed among the functionalities. Although they are considered to be faults of equal severity, faults with different severity levels may vary the results. The functional test case execution time may differ due to the varying lengths of test cases.

6 Results and Analysis

Table 1: Results for Random Ordering

<table>
<thead>
<tr>
<th>%age of test suite run</th>
<th>Machine 1</th>
<th>Machine 2</th>
<th>Machine 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>21.68</td>
</tr>
<tr>
<td>20%</td>
<td>0</td>
<td>0</td>
<td>61.28</td>
</tr>
<tr>
<td>30%</td>
<td>0</td>
<td>0</td>
<td>66.85</td>
</tr>
<tr>
<td>40%</td>
<td>0</td>
<td>0</td>
<td>66.85</td>
</tr>
<tr>
<td>50%</td>
<td>0</td>
<td>32.44</td>
<td>66.85</td>
</tr>
<tr>
<td>60%</td>
<td>19.80</td>
<td>47.06</td>
<td>66.85</td>
</tr>
<tr>
<td>70%</td>
<td>46.90</td>
<td>47.06</td>
<td>66.85</td>
</tr>
<tr>
<td>80%</td>
<td>46.90</td>
<td>47.06</td>
<td>66.85</td>
</tr>
<tr>
<td>90%</td>
<td>46.90</td>
<td>47.06</td>
<td>66.85</td>
</tr>
<tr>
<td>100%</td>
<td>50.61</td>
<td>47.06</td>
<td>67.18</td>
</tr>
</tbody>
</table>

We compared the test results for the random approach with our suggested approach using the APFD metric. We recall that we distributed the test cases randomly among the three participating machines in the random approach. The first two machines were allocated five modules each and the last machine was allotted the remaining four modules. Table I shows the results for the random approach.

We have shown the results in 10% increments. We use the APFD metric to explain our results. We note that the random ordering of the test set in the first machine has not detected any faults for the first 50% of the test set execution. The APFD results for 100% test set execution is 50.61. Similarly, the random ordering of the test set in the second machine has not detected any fault for the first 40% of the test set execution. The APFD result for 100% test set execution is 47.06.

Table II shows the APFD results using our prioritization approach. The test sets allocated to the first machine are able to detect all the faults in the first 30% of the test set execution. The APFD result for 100% test set execution using our prioritization approach is 92.35. The test sets allocated to the second machine are able to detect all the faults in the first 30% of the test set execution. The APFD result for 100% test set execution in using our prioritization approach is 91.55. Similarly, the test sets allocated to the third machine are able to detect all the faults in the first 10% of the test set execution. The APFD result for 100% test set execution for this test set is 97.17.

Table 2: Results from our approach

<table>
<thead>
<tr>
<th>%age of test suite run</th>
<th>Machine 1</th>
<th>Machine 2</th>
<th>Machine 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>77.64</td>
<td>74.95</td>
<td>97.17</td>
</tr>
<tr>
<td>20%</td>
<td>77.64</td>
<td>74.95</td>
<td>97.17</td>
</tr>
<tr>
<td>30%</td>
<td>92.35</td>
<td>91.15</td>
<td>97.17</td>
</tr>
<tr>
<td>40%</td>
<td>92.35</td>
<td>91.55</td>
<td>97.17</td>
</tr>
<tr>
<td>50%</td>
<td>92.35</td>
<td>91.55</td>
<td>97.17</td>
</tr>
<tr>
<td>60%</td>
<td>92.35</td>
<td>91.55</td>
<td>97.17</td>
</tr>
<tr>
<td>70%</td>
<td>92.35</td>
<td>91.55</td>
<td>97.17</td>
</tr>
<tr>
<td>80%</td>
<td>92.35</td>
<td>91.55</td>
<td>97.17</td>
</tr>
<tr>
<td>90%</td>
<td>92.35</td>
<td>91.55</td>
<td>97.17</td>
</tr>
<tr>
<td>100%</td>
<td>92.35</td>
<td>91.55</td>
<td>97.17</td>
</tr>
</tbody>
</table>

Fig. 5 shows the execution results of random ordering of all three test sets. Fig. 5 shows that many faults were detected after executing close to 100% of the test cases. Fig. 6 shows the execution results of the test sets that are prioritized using our approach. Fig. 6 shows that many of the faults were detected in the first 30% of the test set execution.

The execution for the entire test suite on a single machine took approximately 9 hours. After implementing our suggested approach, we were able to execute all the test cases in less than 3 hours and could detect all the faults in the first hour.
7 Conclusions and Future Work

We have proposed a novel parallel prioritization approach for regression testing of complex web applications in this paper. We prioritize the test case executions at two levels, by choosing and prioritizing the functional modules from the functional dependency graph and then ordering the test cases within each test set by using the control flow graphs at the code level. We then distributed the functional modules and their associated test sets among different machines. We measured the performance of our approach using the APFD metric.

We validated the results using various different test combinations and found that our approach is able to detect the faults early and within a small amount of time. In the future, we will validate our results on several other web applications. We will consider real faults with different cost levels. As we have shown, our first 30% execution of test sets detects most of the faults. In the future, we will suggest a technique that will select the test cases related only to the modified functionalities in web applications and execute only those test cases that will provide maximum fault detection. This may help to reduce the total test execution time even more, as we need to execute only a subset of test cases.

References


