TESTING IN PARALLEL
A Need for Practical Regression Testing

Zhenyu Zhang
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Zijian Tong
R&D, Sohu.com Inc., Beijing, China

Xiaopeng Gao
School of Computer Science and Technology, Beihang University, Beijing, China

Keywords: Regression Testing, Test Case Prioritization, Continuous Integration, Pipeline Scheduling.

Abstract: When software evolves, its functionalities are evaluated using regression testing. In a regression testing process, a test suite is augmented, reduced, prioritized, and run on a software build version. Regression testing has been used in industry for decades; while in some modern software activities, we find that regression testing is yet not practical to apply. For example, according to our realistic experiences in Sohu.com Inc., running a reduced test suite, even concurrently, may cost two hours or longer. Nevertheless, in an urgent task or a continuous integration environment, the version builds and regression testing requests may come more often. In such a case, it is not strange that a new round of test suite run needs to start before all the previous ones have terminated. As a solution, running test suites on different build versions in parallel may increase the efficiency of regression testing and facilitate evaluating the fitness of software evolutions. On the other hand, hardware and software resources limit the number of paralleled tasks. In this paper, we raise the problem of testing in parallel, give the general problem settings, and use a pipeline presentation for data visualization. Solving this problem is expected to make practical regression testing.

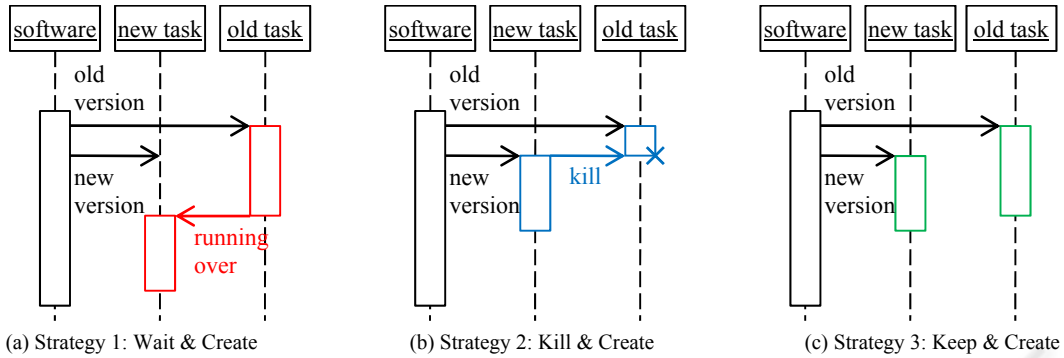
1 INTRODUCTION

Regression testing is a popular technique in software development and maintenance (Elbaum et al., 2000; 2002; 2004). When a program evolves, developers use regression testing to augment, reduce, and prioritize a test suite, before running it to check the functionalities of software in evolution (Onoma et al., 1998; Do et al., 2006; Ramanathan et al., 2008).

Conventionally, a test suite running process is expected to end soon and provide information for developers to ensure the quality of the software build version under test (Rothermel et al., 1997; 2001; 2004). However, from years of realistic industrial experiences, we observed that running a reduced test suite, even concurrently, may cost hours or even longer. On the other hand, in some urgent task or agile continuous integration development pattern (Jiang et al., 2009b), the build versions come freqnently more than once a hour. It makes an unexpected result that a regression testing request for the new build version comes before the last regression testing task for the old build version terminats. We evalute several current strategies to address this issue and find it not a trivial problem.

For example, waiting for the old task to terminate and then starting the new task is not timely enough to find defects in new build version; while killing the old task and immidiately starting the new task has chance to miss the fault in old build version and makes it inharited to new build version. Intuitively, parallaling old and new tasks seems able to run as many as test cases in a unit time and may be effective to reveal faults in both old and new build versions; but the number of paralleled tasks is often limited by the hardware and software resources.

Zhang Z., Tong Z. and Gao X. (2010).
TESTING IN PARALLEL - A Need for Practical Regression Testing.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 344-348
DOI: 10.5220/0003041503440348



(a) Strategy 1: Wait & Create (b) Strategy 2: Kill & Create (c) Strategy 3: Keep & Create

Figure 1: Sequence diagrams for different strategies.

For example, it is not easy to set up many instances to in parallel test a searching service, which occupies a fixed range of ports and consumes up to about 2.5G physical memory. Besides, the test cases run on old build versions may provide reusable information for new build version. Making use of such informations in a paralleled regression testing manner may gain significant progress and generate meaning results.

In this paper, we raise the problem of *testing in parallel*, targeting at refining the test case priorities for test suites parallel running on different build versions, to make effective, efficient, and practical regression testing. We also give a formal problem settings to address this problem and use a pipeline presentation for data visualization.

The contributions in this paper is threefold. (i) It is the first time that such a practical problem on regression testing is reported from industry. (ii) We propose the approach of testing in parallel to address the reported problem. (iii) We give the first formal problem settings to formulate this problem and use a pipeline presentation for data visualization.

The rest of the paper is organized as follows. Section 2 uses a realistic scenario to motivate our work. Section 3 gives formal problem settings and data visualization, followed by an introduction to related work in Section 4. Section 5 concludes the paper. Section 6 foresees some future work.

2 MOTIVATION

In this section, we start from a realistic industrial development scenario and motivate our work.

2.1 A Realistic Scenario

In a searching component project in Sohu.com Inc., the average integration period is about two hours; while the reduced test suite contains more than 2000 test cases and executing the program over such a test suite approximately costs 3.5 hours (some of these test cases have been scheduled and run concurrently). Most of the time, a build version is compiled over and a regression testing request is raised, before the last regression testing task (on the last build version) terminates. In our daily development, we adopt three strategies to address this issue. To ease reader's understanding, we show the sequence diagrams for these three strategies in Figure 1.

[Strategy 1: Wait & Create] The new regression testing task cannot start until the old task terminates, as Figure 1(a).

[Strategy 2: Kill & Create] The old regression testing task is killed at once, and then the new task is created immediately, as Figure 1(b).

[Strategy 3: Keep & Create] The new regression testing task is immediately created and working in parallel with the old task, as Figure 1(c).

In next section, we compare the three strategies.

2.2 Existing Problems

By adopting strategy 1, we wait for the old regression testing task to terminate. As a result, the new build version cannot be tested timely. A program fault, which is responsible for the software defect found during the old regression testing task, may have been fixed in the new build version while we are wasting time testing a old build version. For example, we ever found that a programmer had fixed the fault and committed into new build version.

For strategy 2, we immediately terminate the old regression testing task and start the new task. It is possible that no failure has been revealed yet when we killed the unfinished old regression testing task. Thus, a undetected fault may be inherited by the new build version. Nevertheless, if it is never triggered or

Table 1: Comparison to properties of strategies.

	Effectiveness (*in terms of number of run test cases*)	**Efficiency** (*in terms of speed to run test cases*)	**Limitation** (*in terms of number of paralleled tasks*)
Strategy 1: Wait & Create	***High***	*Low*	***Less***
Strategy 2: Kill & Create	*Low*	*Low*	***Less***
Strategy 3: Keep & Create	***High***	***High***	*More*

encoutners a coincidental correctness case (Wang et al., 2009), the fault cannot be found in the new task.

By adopting strategy 3, we in parallel run test suite on each build version. That maximizes the probability of revealing a failure. This seems the most effective strategy, but the number of paralleled regression testing tasks are commonly limited by the hardware or software resources. For example, we ever needed to test a service on the standard 80 port. It is possible to use conventional methods to in parallel test it at one site. For another example, we ever needed to test a background daemon program that occupies up to 4G memory. It is not feasible to create multiple program instances, limited by the amount of physical memories.

We use Table 1 to summarize the properties of adopting these three different strategies. Our observation is that there is not a universally best strategy among them. Strategy 1 and strategy 3 are more effective than strategy 2 since they run all test cases in the test suites and may have higher probability to reveal faults. Strategy 3 is more efficient than strategy 1 and strategy 2 since it in parallel run the test suite. On the other hand, strategy 1 and strategy 2 have less limitation, compared with strategy 3, since they do not need to create multiple program instances.

2.3 The Idea of Testing in Parallel

In previous section, we have elaborated on the advantages and disadvantages of three current strategies when facing the problem of testing in parallel in our everyday developing work. Our preliminary judgement is as follows.

(i) Paralleling the run of test suite is necessary since it may increase the probability of revealing failure and thus increase the effectiveness of regression testing;

(ii) Blindly paralleling all regression testing tasks may not be feasible because of the limitation of number of paralleled regression testing tasks.

(iii) On the other hand, it is not a economic choice to parallel as many as tasks without scheduling the test case priorities among different test suites because test case run on old build versions may provide useful information for new version.

If a test case run (in a previous regression testing task) on an old build version has revealed a failure, it should be given particularly low priority of running in the regression testing task on a new build version. Let us analyze in different cases. Suppose the running of such a test case on the new regression testing task also reveals a failure, it has high possibility to be due to a same fault because there are generally not huge changes between two adjacent build versions. The counterpart is that the running of such a test case on the new regression testing task reveals no failure, which means that the fault has not been triggered, or there happens coincidental correctness (Wang et al., 2009), or the fault has been fixed in the new build version. None of them takes in new information.

Inspired by such motivation, we raise the problem of testing in parallel, that is, under the situation of paralleled regression testing tasks, with limited number of paralleled tasks, how to conduct regression testing ***effectively*** and ***efficiently***?

2.4 Challenges

In last section, we use an interesting application scenario to demonstrate a motivating example, and raise the problem of testing in parallel. Intuitively, paralleling the test suite runs seems feasible and practical. However, we still foresee many potential challenges when addressing this problem. For example, test case run information on old build version may include reusable information about fault inherited to new version. Such information can be used to prioritize test cases on new build version. How to scientifically reuse those information in a testing in parallel pattern? Paralleling as many as test suite runs may increase the speed of revealing

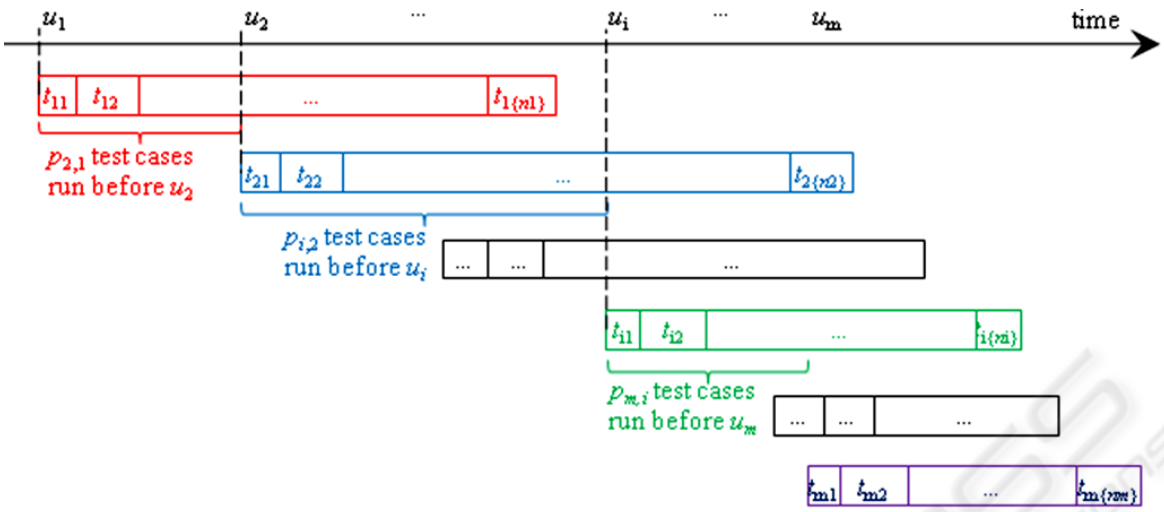


Figure 2: The pipeline presentation of problem settings.

fault. Limited by the number of paralleled test suite runs, how to achieve the goal of maximizing fault-revealing efficiency? Besides, can we find a short cut to visualize this problem and map it to some other forms of familiar problems?

In next section, we shall elaborate on our problem settings and data visualizations.

3 TESTING IN PARALLEL

In this section, we give the problem settings and data visualization.

3.1 Problem Settings

Suppose $v_1, v_2, \ldots, v_m$ are m sequential build versions, respectively released at time $u_1, u_2, \ldots, u_m$. A test suite is initialized as S_1 for version v_1; it contains n_1 test cases. For version v_i, the test suite is accordingly updated to S_i, which contains n_i test cases. Test cases in test suite S_i are run in the order of $t_{i1}, t_{i2}, \ldots, t_{i\{ni\}}$ with respect to version v_i; where the ordered list of $\langle i1, i2, \ldots, i\{n_i\}\rangle$ is a permutation of $\langle 1, 2, \ldots, n_i\rangle$. We further suppose that at time u_j, in total $p_{i,j}$ test cases in S_i has been run on a previous version v_i.

Since different test suites with respect to different build versions may contain identical test cases, we further involve a term ***Identity*** (t_{ix}, t_{jy}) to identify this relationship. If the x-th test case of test suite S_i and the y-th test case of test suite S_j are the same one, we let ***Identity*** (t_{ix}, t_{jy}) = 1; otherwise 0. We further use N to stand for the upper limit for number of paralleled test suite runs.

Suppose that for each test suite, the set of test case priorities is an optimal one that gives maximum efficiency of revealing fault, for that build version. In other words, for each test suite, the test cases are given priorities according to the probability each of them revealing fault. We use P_{ij} with respect to version v_i to stand for the probability of running test case t_{ij} revealing fault. Therefore, we have $P_{i1} > P_{i2} > \ldots > P_{ij} > \ldots > P_{i\{ni\}}$. Our goal is to refine the running order of test cases in each test suites, to maximize the efficiency of revealing fault for all the paralleled regression testing tasks.

3.2 Data Visualization

We further use a pipeline-like structure to visualize the problem settings, as Figure 2.

In Figure 2, each row shows test suite run for one version. Cells in each row mean test cases that are run in order. The slower a test case runs, the wider its corresponding cell. Different row starts from different time point (see the time axis on the top); it means that build versions come one after another sequentially.

Such a data visualization maps the problem to a pipeline scheduling problem. Since the latter has mature technique basis, such visualization is expected to ease the problem. Note that ***Identity*** (t_{ix}, t_{jy}) and N are not shown in this draft.

4 RELATED WORK

Many test case prioritization and regression testing research results have been reported (Rothermel et al.,

1997; Elbaum et al., 2000; Rothermel et al., 2001; Elbaum et al., 2002; 2004; Rothermel et al., 2004). Wong et al. (1997) combined test suite minimization and prioritization. Srivastava et al. (2002) employed a binary matching system to prioritize test cases to maximally program coverage. Do et al. (2006) investigated the impact of test suite granularity. Li et al. (2007) showed that genetic algorithms perform well for test case prioritization, but greedy algorithms are also effective in increasing the code coverage rate. Jiang et al. (2009a) used adaptive random testing concept to facilitate test case prioritization. However, all those work focus on prioritization techniques; they have not started from industrial usage to report the problem of practice.

Our previous work (Jiang et al., 2009b) studied the problem of how prioritization techniques affect fault localization techniques in a continuous integration environment. It inspires this work. Walcott et al. (2006) investigated a time-aware prioritization technique. It is related to resource usage and thus related to our work.

5 CONCLUSIONS

Regression testing is a popular technique used to evaluate the fitness of evolving software. In a regression testing process, a test suite is run to ensure the software functionalities. However, due to the complicated functionality of software and urgent tasks in development, the time used to run a test suite can be much longer than the time interval between two adjacent build versions. There is a need to parallel the test suite runs.

In this paper, we start from a realistic industrial scenario to show the problem. We next investigate the advantages and disadvantages of our previous strategies to address this issue. We finally propose a testing in parallel manner, give the formal problem settings and goals, and use a pipeline presentation to visualize the problem.

6 FUTURE WORK

In the future work, we shall design an algorithm to solve this problem, conduct controlled experiment to evaluate our solution, and implement visualization tools to support industrial usages.

ACKNOWLEDGEMENTS

This research is supported by the National High Technology Research and Development Program of China (project no. 2007AA01Z145).

REFERENCES

H. Do, G. Rothermel, and A. Kinneer (2006). Prioritizing JUnit test cases: an empirical assessment and cost-benefits analysis. *Empirical Software Engineering*.

S. G. Elbaum, A. G. Malishevsky, and G. Rothermel (2000). Prioritizing test cases for regression testing. In *ISSTA 2000*.

S. G. Elbaum, A. G. Malishevsky, and G. Rothermel (2002). Test case prioritization: a family of empirical studies. *TSE*.

S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky (2004). Selecting a cost-effective test case prioritization technique. *Software Quality Control*.

B. Jiang, Z. Zhang, W. K. Chan and T. H. Tse (2009a). Adaptive random test case prioritization. In *ASE 2009*.

B. Jiang, Z. Zhang, T. H. Tse, and T. Y. Chen (2009b). How well do test case prioritization techniques support statistical fault localization. In *COMPSAC 2009*.

Z. Li, M. Harman, and R. M. Hierons (2007). Search algorithms for regression test case prioritization. *TSE*.

A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma (1998). Regression testing in an industrial environment. *Communications of the ACM*.

M. K. Ramanathan, M. Koyuturk, A. Grama, and S. Jagannathan (2008). PHALANX: a graph-theoretic framework for test case prioritization. In *SAC 2008*.

G. Rothermel, S. G. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu (2004). On test suite composition and costeffective regression testing. *TOSEM*.

G. Rothermel and M. J. Harrold (1997). A safe, efficient regression test selection technique. *TOSEM*.

G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold (2001). Prioritizing test cases for regression testing. *TSE*.

A. Srivastava and J. Thiagarajan (2002). Effectively prioritizing tests in development environment. In *ISSTA 2002*.

K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos (2006). Timeaware test suite prioritization. In *ISSTA 2006*.

X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang (2009). Taming coincidental correctness: coverage refinement with context patterns to improve fault localization. In *ICSE 2009*.

W. E. Wong, J. R. Horgan, S. London, and H. Agrawal (1997). A study of effective regression testing in practice. In *ISSRE 1997*.