RN/11/15
July 11, 2011

Research
Note

Faster Fault Finding at Google Using Multi Objective Optimisation

S. Yoo, R. Nilsson & M. Harman

Telephone: +44 (0)20 3108 5032
Fax: +44 (0)171 387 1397
Electronic Mail: s.yoo@cs.ucl.ac.uk, robni@google.com, m.harman@cs.ucl.ac.uk
URL: http://www.cs.ucl.ac.uk/staff/S.Yoo/,
http://www.cs.ucl.ac.uk/staff/M.Harman/

Abstract

Companies such as Google tend to develop products from one continually evolving core of code. Software is neither shipped, nor released in the traditional sense. It is simply *made available*, with dramatically compressed release cycles regression testing. This large scale rapid release environment creates challenges for the application of regression test optimisation techniques. This paper reports initial results from a partnership between Google and the CREST centre at UCL aimed at transferring techniques from the regression test optimisation literature into industrial practice. The results illustrate the industrial potential for these techniques: regression test time can be reduced by between 33%–82%, while retaining fault detection capability. Our experience also highlights the importance of a multi objective approach: optimising for coverage and time alone is insufficient; we have, at least, to additionally prioritise historical fault revelation.

Department of Computer Science
University College London
Gower Street
London WC1E 6BT, UK

1 Introduction

Regression testing is performed to gain confidence that the recent modifications made to software system do not interfere with the existing functionalities [10]. Regression test suites tend to grow as the system evolves. Regression testing typically seeks to reduce cost either by selection, minimisation or prioritisation. Selection precisely selects only those tests that are relevant to the most recent changes [1,7]. Minimisation eliminates those tests that do not contribute to the chosen test criterion [4,5]. Prioritisation assigns a priority ordering so that the more effective tests can be executed earlier [8,11].

With good test practice, test suites quickly grow, making timely re-execution of all tests infeasible. This problem establishes a natural and pressing opportunity for some for of test suite optimisation. However, industrial uptake has, hitherto, been rather slow: a recent survey of the field shows that only 8% of the published work evaluates proposed techniques with industrial-scale subjects [10], indicating that further technology transfer work is required. This paper reports our experience in seeking to achieve this technology transfer through the integration of search-based regression testing techniques within Google's test process.

Google's approach to managing the scale of regression testing uses massive parallelism to farm out tests. When a new change is submitted, all tests that could be transitively affected by the change are retested over a period of time governed by a test scheduler. This means that all regression testing will be completed, but the amount of time between submission of a change and a report back on the completion of testing can be considerable, despite parallelism. Google's parallel retesting approach is essential for scalability, not merely because of the size of the test suites, but because of exceptionally high change frequency: Copeland recently reported more than 20 code changes every minute [2].

Regression test optimization can help in this situation because it can be used to identify a set of test cases that could be run locally on the developers' machines before the change is submitted to the code base, with it asynchronous massively parallel regression test infrastructure. To solve this problem we find ourselves in a relatively familiar, well-studied, territory for regression testing research: find a subset of test suites that can achieve early fault revelation with limited resources. This is a compromise scenario for which multi objective regression test optimisation is well-suited [3].

Therefore, we adopted a multi objective search-based test suite selection technique, based on previous work [9], adapting to operate within Google's test environment. Our approach seeks to find a suitable pre-submit test suite that can be run locally with a reasonably high probability of early fault revelation. Though all changes that pass the pre-submit test will be fully tested using the post-submit build system, the pre-submit phase obviates the need for a post-submit phase if it detects a fault.

Our search based optimisation seeks test suites that maximise coverage and historical fault revelation, while minimising execution time. In the literature, selection based on coverage has also been referred to as test suite 'minimisation', but it is important to recognise that our overall approach throws nothing away; ultimately it re-tests all test cases using the massively parallel post-submit test management system. However, the optimised subset is prioritised for early execution (and is run pre-submit). Thus, in order to adapt the regression test optimisation techniques in the literature to Google's regression test environment, our approach combines elements of 'traditional' test suite selection, minimisation and prioritisation.

2 Motivation

In a development environment where changes are frequent and source code is submitted to a shared code repository, faults that are introduced in one sub-system can be propagated rapidly to dependent sub-systems. This causes severe integration problems and lost productivity.

To avoid inadvertent propagation of unexpected faults from a newly submitted component to its dependant components, Google uses a rigorous continuous build and integration system that regression tests every code submission to the shared repository. The set of tests that are run post-submission is currently chosen conservatively using a build dependency graph to compute all automated regression tests that could possibly be affected by a given change. This will include some tests that are only incidentally related to the changed component. The order of execution is comparatively unimportant in this context, since all dependent tests eventually will be executed, and furthermore, the post-submit tests are executed asynchronously to the development process. A valuable benefit of being conservative and complete is that it is easy to track down exactly what change resulted in a regression fault and test failures are unlikely to be missed simply because (possibly transitive) dependencies are missed.

However, when a fault is detected post-submit, manual intervention is required to resolve the issue in the most appropriate way. A fault submitted to the shared repository influences the productivity of other teams because, for example, detection of new faults can be masked by the one previously known. Hence, there is a strong desire to detect potential faults pre-submit, reserving the post-submit build system as last line of detection and a debugging aid.

3 Multi Objective Paradigm

This section introduces the multi-objective formulation of test case selection. Section **??** introduces the Pareto optimal formulation of the test case selection problem. The detailed description of the application of multi-objective regression testing optimisation can be found in Yoo and Harman [9].

Pareto optimality is a notion from economics with broad range of applications in game theory and engineering. The original presentation of the Pareto optimality is that, given a set of alternative allocations and a set of individuals, allocation A is an improvement over allocation B only if A can make at least one person better off than B, without making any other worse off.

Based on this, the multi-objective optimisation problem can be defined as to find a vector of decision variables x, which optimises a vector of M objective functions $f_i(x)$ where $i = 1, 2, \ldots, M$. The objective functions are the mathematical description of the optimisation criteria, which are often in conflict with each other.

Without the loss of generality, let us assume that we want to maximise f_i where $i = 1, 2, \ldots, M$. A decision vector x is said to dominate a decision vector y (also written $x \succ y$) if and only if their objective vectors $f_i(x)$ and $f_i(y)$satisfies:

$$f_i(x) \geq f_i(y) \forall i \in \{1, 2, \ldots, M\}; and$$
$$\exists i \in \{1, 2, \ldots, M\} | f_i(x) > f_i(y)$$

All decision vectors that are not dominated by any other decision vectors are said to form the *Pareto optimal set*, while the corresponding objective vectors are said to form the *Pareto frontier*. Now the multi-objective optimisation problem can be defined as follows:

Given: a vector of decision variables, x, and a set of objective functions, $f_i(x)$ where $i = 1, 2, \ldots, M$

Definition: maximise $\{f_1(x), f_2(x), \ldots, f_M(x)\}$ by finding the Pareto optimal set over the feasible set of solutions.

Identifying the Pareto frontier is particularly useful in engineering because the decision maker can use the frontier to make a well-informed decision that balances the trade-offs between the objectives.

The multi-objective test case selection problem is to select a Pareto efficient subset of the test suite, based on multiple test criteria. It can be defined as follows:

Multi Objective Test Case Selection *Given:* a test suite, T, a vector of M objective functions, f_i, $i = 1, 2, \ldots, M$.

Problem: to find a subset of T, T', such that T' is a Pareto optimal set with respect to the objective functions, f_i, $i = 1, 2, \ldots, M$.

The objective functions are the mathematical descriptions of test criteria concerned. A subset t_1 is said to dominate t_2 when the decision vector for t_1 ($\{f_1(t_1), \ldots, f_M(t_1)\}$) dominates that of t_2.

4 Problem Formulation

We define dependency coverage using a module dependency graph. Figure 1 shows a partial module dependency sub-graph. Google's code repository treats both functional modules and tests in the same manner and the module dependency information is available not only between functional modules (*use* dependency) but also between tests and the modules they test (*test* dependency). Based on dependency coverage, a test t *covers* a module m if t is transitively dependent on m. Let $\mathcal{M}$ be the set of modules that depend on the recently modified module (i.e. the code submission under test). Let $\mathcal{T} = \{t_1 \ldots, t_n\}$ be the full test suite, from which an optimised subset T is produced. The dependency coverage δcov is calculated as follows:

$$\delta cov(T) = \frac{|\{m_i \in \mathcal{M} : \exists t_j \in T \text{ s.t. } m_i \text{ is reached from } t_j\}|}{|\mathcal{M}|}$$

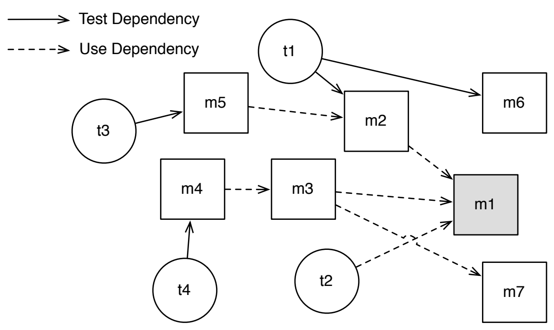


Figure 1: Illustration of Dependency Coverage: if module m_1 is modified, the impacted modules are m_2, m_3, m_4 and m_5. Based on dependency coverage, test t_1 covers m_1 and m_2; t_3 covers m_1, m_2 and m_5.

We use multi-objective optimisation to observe the trade-off between multiple test criteria and testing cost, adapted from our previous work [9]. We optimise the following three objectives in our selection of a test subset $T \subseteq \mathcal{T}$:

Dependency Coverage: we seek to *maximise* the dependency coverage achieved by T. This prioritises the selection of tests that execute the code transitively affected by the change, helping to promote detection of integration faults.

Fault History: we seek to *maximise* the ratio of tests in T that have failed within a fixed time window. This prioritises the selection of tests with previously demonstrated high fault detection capabilities. While there is no guarantee that these will be good at finding new faults, our empirical studies indicate that they are in practice (see Section 5).

Execution Time: we seek to *minimise* the sum of execution time required by the tests in T.

Failing Tests: For web-enabled software products, tests may fail even though there is no fault causing the failure (for example due to temporary unavailability of a third party web service). Fortunately, the sheer scale of Google's test operation provides a wealth of data that helps us to detect many of these false positives. Based on their fault histories, a heuristic decision procedure now widely adopted within Google, was developed for this project to filter out these environmental failures.

We use the Two-Archive Multi-Objective Evolutionary Algorithm (MOEA) as the optimisation engine [6]. The representation of individual solutions is an n-ary bitstring, in which digits mark the inclusion/omission of each test in $\mathcal{T}$. The multi-objective regression test optimisation was been implemented into a tool called `TIPS` (Test Information Prioritisation & Suggestions). The core of `TIPS` is the Two-Archive MOEA written in Python. Various metric collection modules provide the fitness function imnformation to the optimisation engine.

4.1 Benefits of Pre-Submit Test Optimisation

Automated test suite optimisation brings three primary advantages to the developer during the pre-submit phase:

1. It is often costly for a developer to manually identify tests that are relevant to their change, particularly for tests designed for components that use the changed code indirectly. Automated regression test optimization removes this need for manual identification.
2. Early test feedback allows developers to understand how dependent sub-components expect their module to behave so that they can design their changes to be compliant with existing code. Therefore, even when the pre-submit phase passes all tests, the developer receives useful early feedback about the tests executed.
3. Developers lose productivity by having to wait for large sets of test cases to run. If there is a fault, then it is clearly best to learn about it as soon as possible. The pre-submit phase therefore increases the efficiency of the overall process.

4.2 ABBA: Adaption Breeds Better Adoption

In seeking any form of technology transfer from academic research into industry, there is likely to be significant adaption of techniques to make them work in practice. This adaption often involves familiar challenges such as scalability, but there are also more prosaic practical considerations: existing industrial practice may actually work rather well; if it did not then company would not be in business. This is manifestly true at Google, which has achieved dramatic growth and managed the consequent testing scale up in the process of growing. Academic research does have a role to play: it can help to *optimise* existing approaches. Regression test optimisation is all about achieving this goal. However, for it to work in practice, the techniques have to be incorporated with minimum disruption to existing infrastructure, procedures and policies; a kind of optimisation goal in itself.

Given the existing context at Google, our goal was not to reduce the number of tests to execute: the continuous build and integration system will ensure that all relevant tests are eventually executed, retaining full fault detection capability. Rather, the benefits for developers lay in retaining a useful *partial* fault detection capability in a subset of all test cases selected to be prioritised for early execution.

5 Evaluation

In order to evaluate our approach, we have analysed, in detail, 28 randomly sampled changes submitted to Google's code repository[1]. We harvested metrics from Google's continuous build and integration infrastructure: the number of relevant tests, test execution time and (filtered) failure detection. Table 1 summarised test suite sizes and execution time.

[1]A more complete evaluation will be reported upon in a subsequent journal version of this paper.

Table 1: Summary of Test Suites

Property	Min.	Average	Max.
Test Suite Size	4	461	1,439
Execution Time (sec)	115	39,093	116,131

Our Pareto efficient multi-objective optimisation yields a set of solutions that reflect the trade offs inherent in balancing the three competing objectives we seek to optimise: dependency coverage maximisation, historical fault detection maximisation and execution time minimisation. For example, a set of tests, A, may achieve higher dependency coverage than another set of tests, B, but also require longer to execute than B. In such a situation both solutions lie on a Pareto surface, since they are equally valid solutions.

The shape of the surface gives insights to the trade offs between the optimisation objectives. It is hard to depict a 3D Pareto surface in 2D, so we flatten the 3 dimensions to 2, using multiple 2D curves show the different parato fronts corresponding to different levels of historical fault detection.

Figure 2 presents selected illustrations of results from the 28 evaluated code submissions. The complete set of 28 plots is presented in Appendix. Each circle corresponds to a subset of tests that are generated by the optimisation: the shape of each curve is a Pareto front revealing the trade-offs between the execution time (x-axis), the dependency coverage (the left y-axis). Empty circles denote subsets that fail to detect any new faults, whereas solid circles denote subsets that do.

Figures 2(a) and 2(b) illustrate ideal cases for our approach: the optimised subsets detect faults even with relatively low dependence coverage. Both reveal multiple patterns of trade-off between dependency coverage, fault history and cost.

However, Figure 2(c) represents a less ideal case. There is only a single front for this figure because no test has a fault history. In this case optimising for dependency coverage alone does not yield early fault detection: 19 of the 89 tests revealed a new fault for this change, yet none of the optimised subsets contains any of these 19 failing tests.

Overall, our results indicate that fault histories are useful optimisation goals: Out of the 28 evaluated code submission, 23 contained faults. Optimisation resulted in test subsets that detected those faults early for 20 submissions, thereby reducing time to first fault in 86% of cases.

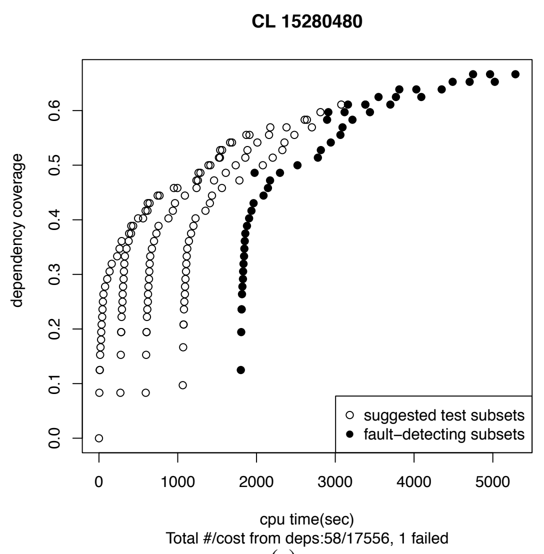
However, when no fault history information was available, optimisation failed to find fault-revealing test subsets. Further research is required to investigate the observations more rigorously. Figure 3 presents boxplot summaries of our sample of 28 changes. Overall, the developer can expect 33%–82% reduction in testing time compared to executing all relevant tests.

6 Future Challenges

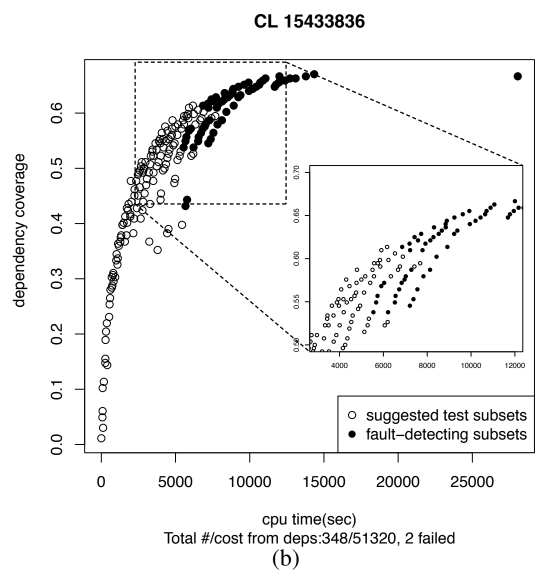
Google is not alone in operating within a rapid release test environment. Many companies release rapid updates from continual changes to a large code base. Our work raises the following research and engineering challenges, the solutions to which may further improve the effectiveness of regression testing in similar highly constrained 'rapid release' scenarios:

Environmental Nondeterminism: tests that are sensitive to environmental factors need to be pre-filtered to improve the metrics that guide optimisation, raising the question of how best to define such filtering decision procedures.

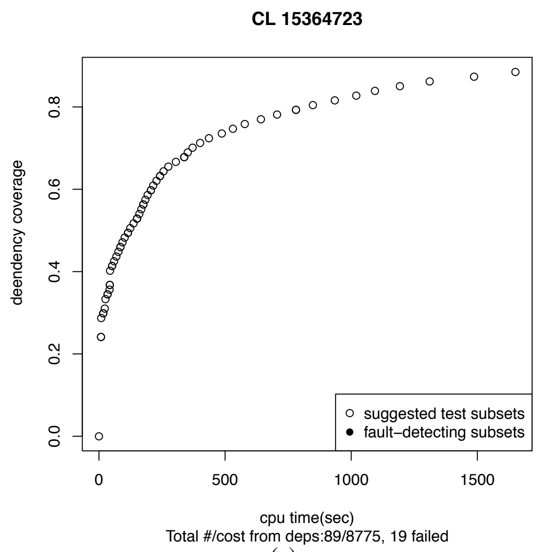
Test Aware Dependence: A build dependence between some module and a test may not mean that the test truly *tests* the dependent module. At this module level of abstraction better metrics are required to capture the tester-testee relationship between modules. Greater dependence precision will provide more accurate fitness information, thereby better guiding the search for good tests.



(a)



(b)



(c)

Figure 2: Results from the test suite optimisation: each point corresponds to a subset of tests proposed by the technique. In Figures 2(a) and 2(b), multiple cost/coverage Pareto fronts can be observed. This is because all three graphs also report the third fault history objective (flattened onto 2D rendering of the 3D Pareto surface). Rightmost Pareto fronts in the 2D rendering consist of subsets that achieve lower dependency coverage but higher fault history coverage per unit cost. Solid circles denote optimised test suites that reveal new faults. The distribution of these highlights the importance of fault history as an objective. The Pareto surface for Figure 2(c) consists of a single Pareto front because no fault history is available in this case.

Dependency Hot-spots: Core libraries are 'dependency hot spots'. Changing them inherently requires a lot of re-testing, raising the question of how can we effectively hybridise existing optimisation and analysis techniques to cope with dependency hot spots.

7 Conclusion

This paper reports experience and initial results from a project to integrate search based regression test optimisation into Google's regression testing processes. We use coarse-grained module dependency coverage to achieve scalability and multiple test objectives to improve practicality by incorporating additional factors into the optimisation process. Initial results indicate that a 33%–82% reduction in testing time may be achieved by optimisation.

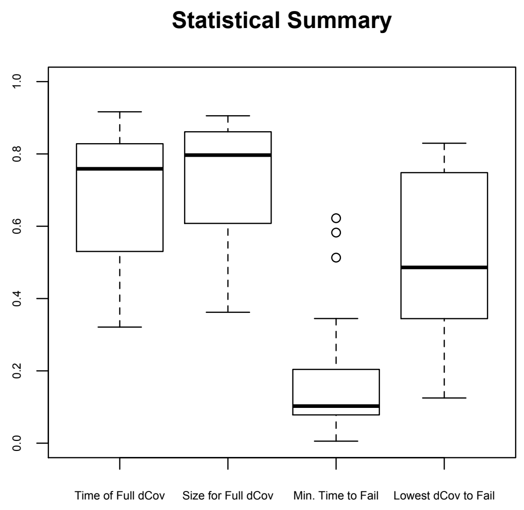


Figure 3: Optimised test subsets compared to complete suites (full testing). Execution time can be reduced by as much as 33%. Earliest fault detection time can be reduced by as much as 82%.

Acknowledgement: This work is funded by the EPSRC project SEBASE and by a Google Research Award.

References

[1] L. C. Briand, Y. Labiche, K. Buist, and G. Soccar. Automating impact analysis and regression test selection based on UML designs. In *ICSM 2002*, pages 252–261. IEEE Computer Society, October 2002.

[2] P. Copeland. Google's innovation factory: Testing, culture, and infrastructure. In *ICST 2010*, ICST '10, pages 11–14. IEEE Computer Society, 2010.

[3] M. Harman. Making the case for MORTO: Multi objective regression test optimization. In *Regression 2011*, Berlin, Germany, Mar. 2011.

[4] S. McMaster and A. Memon. Call-stack coverage for GUI test suite reduction. *IEEE Transactions on Software Engineering*, 34(1):99–115, 2008.

[5] J. Offutt, J. Pan, and J. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the 12th International Conference on Testing Computer Software*, pages 111–123, June 1995.

[6] K. Praditwong and X. Yao. A new multi-objective evolutionary optimisation algorithm: The two-archive algorithm. In *CEC 2006*, volume 4456 of *Lecture Notes in Computer Science*, pages 95–104, November 2006.

[7] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *ICSM 1993*, pages 358–367, September 1993.

[8] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

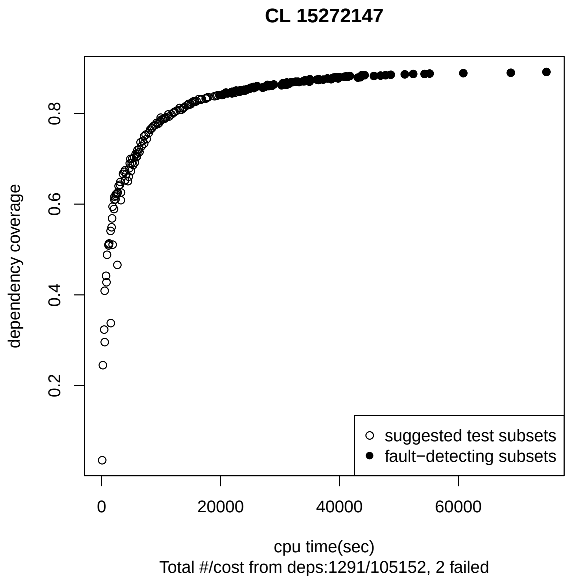
[9] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *ISSTA 2007*, pages 140–150. ACM Press, July 2007.

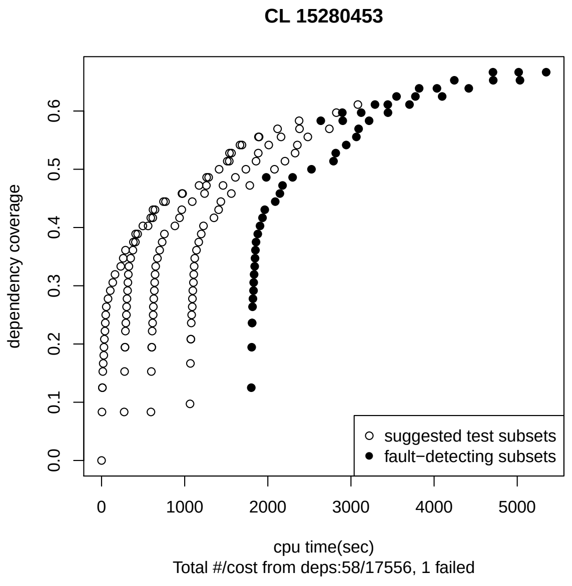
[10] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification, and Reliability, to appear*, 2011.

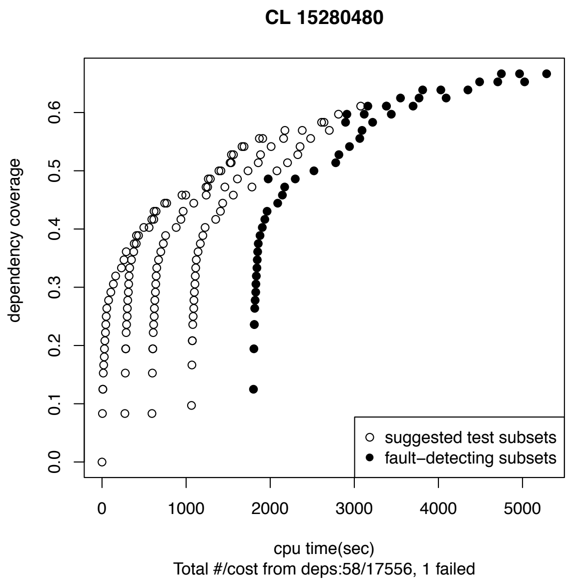
[11] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge. In *ISSTA 2009*, pages 201–211. ACM Press, July 2009.

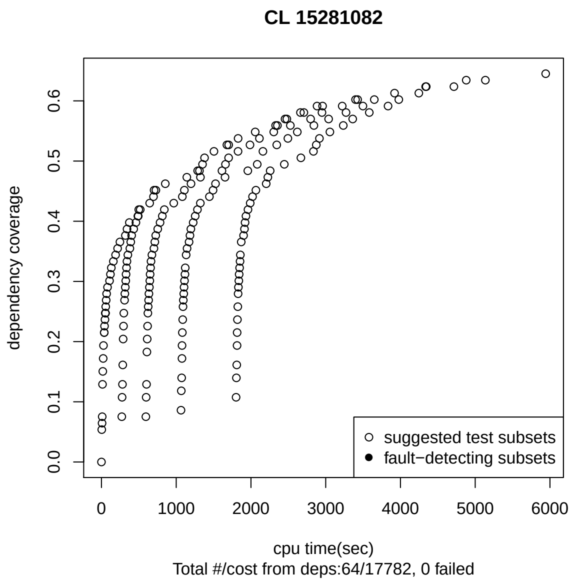
Appendix

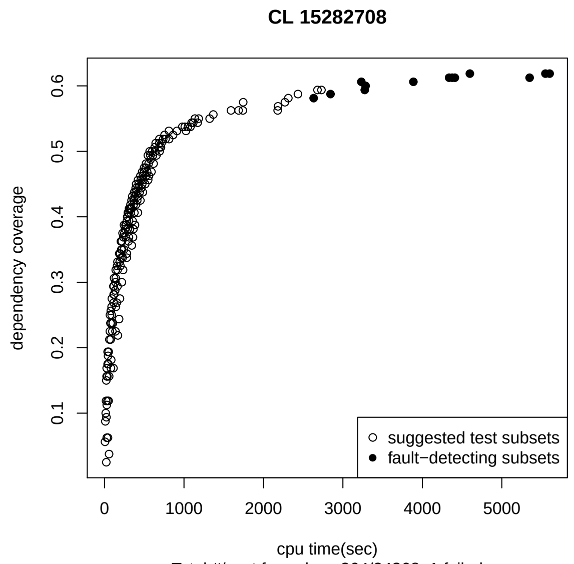
Here are all 28 plots of the Pareto fronts obtained by the optimisation. Each plot corresponds to a Change List (CL), a unit of code submission. Plots for CLs that resulted in no failure (15281082, 15283540, 15358312, 15378688 and 15417409) show no black dots, i.e. there is no failure to detect.



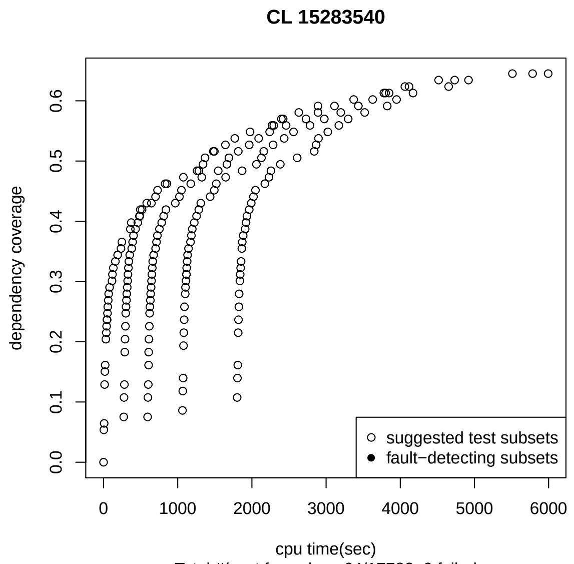




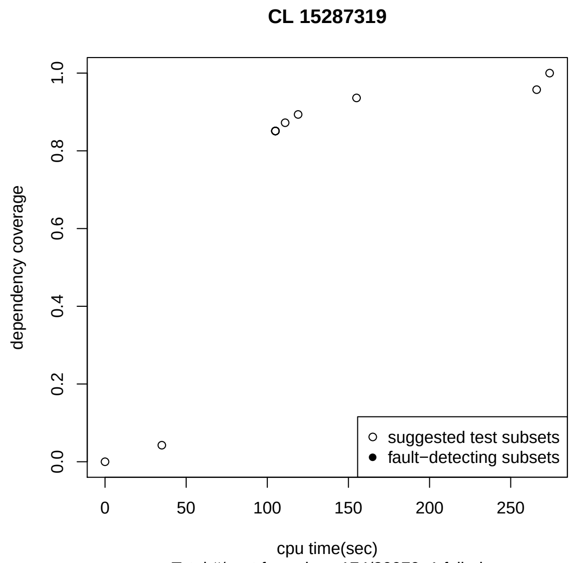




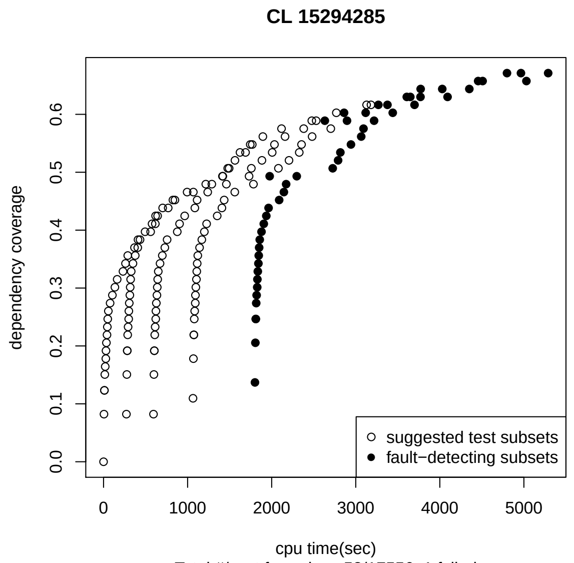
Total #/cost from deps:364/24368, 1 failed



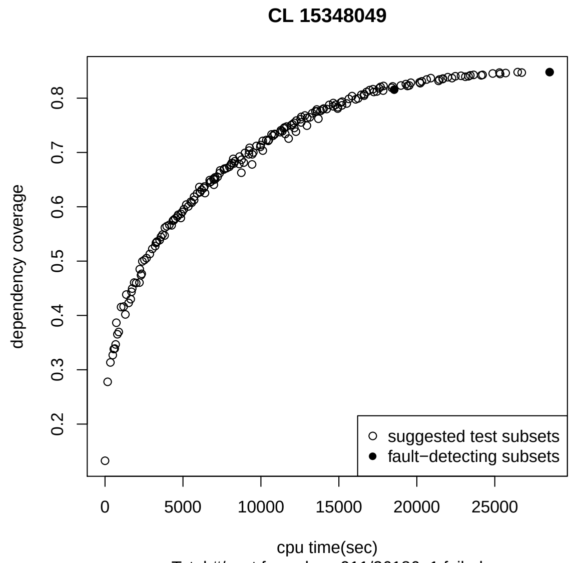
Total #/cost from deps:64/17782, 0 failed



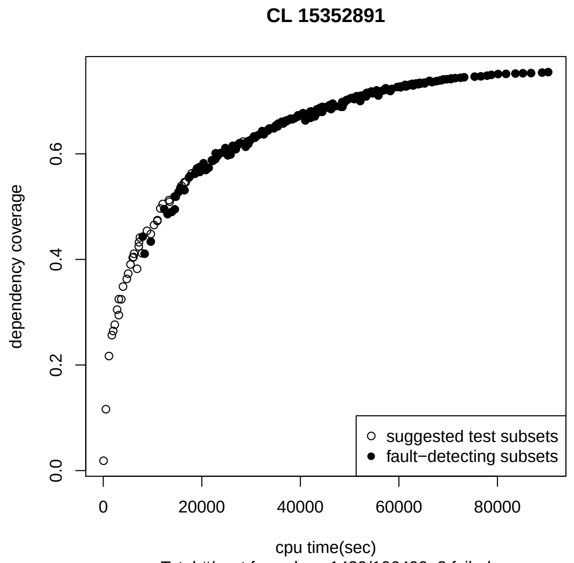
Total #/cost from deps:174/20979, 1 failed



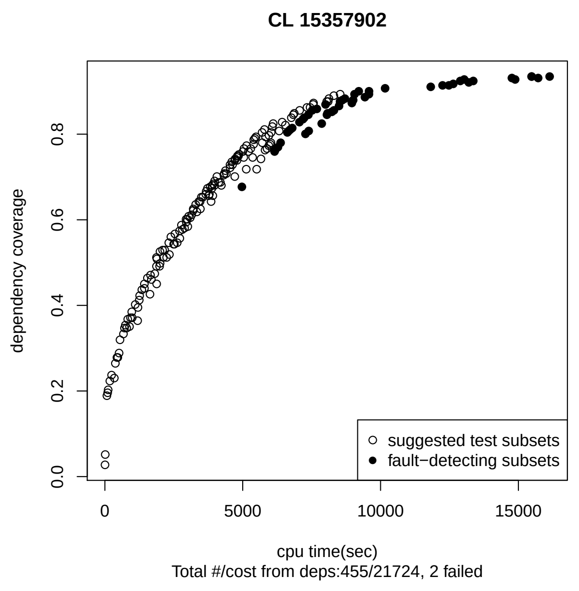
Total #/cost from deps:58/17556, 1 failed

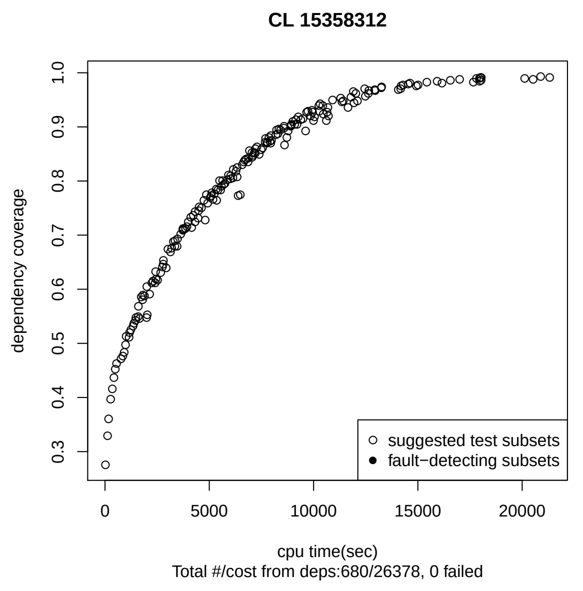


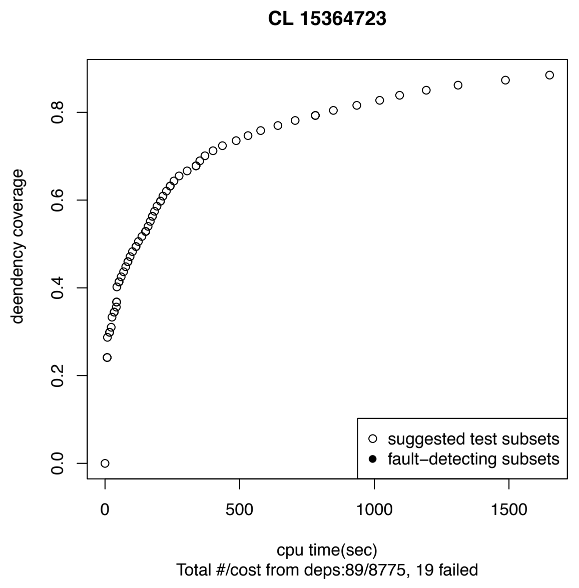
Total #/cost from deps:911/36186, 1 failed

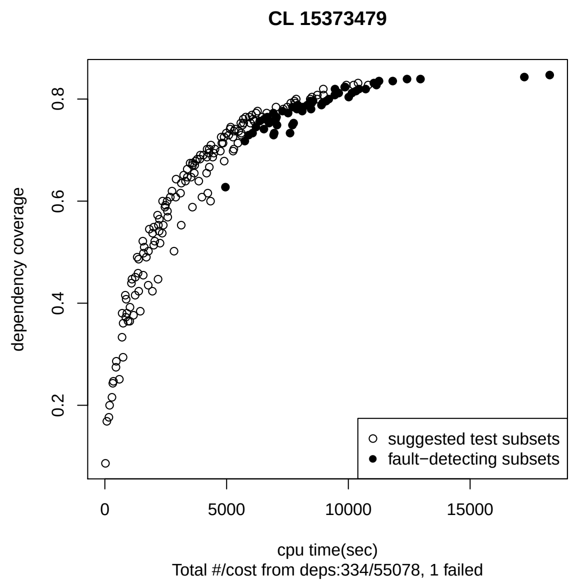


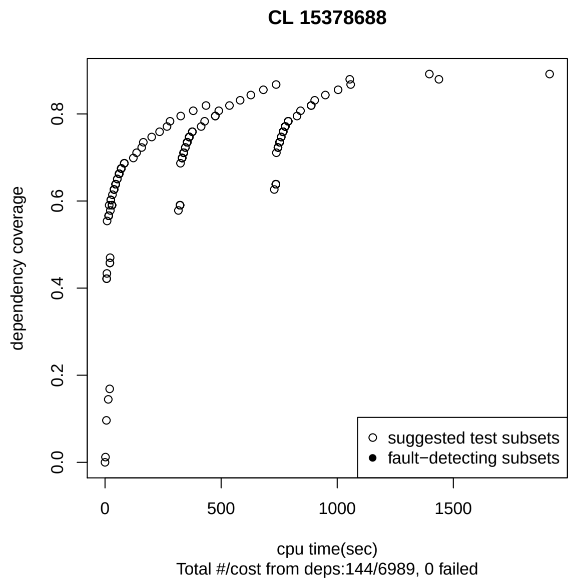
Total #/cost from deps:1439/106499, 3 failed

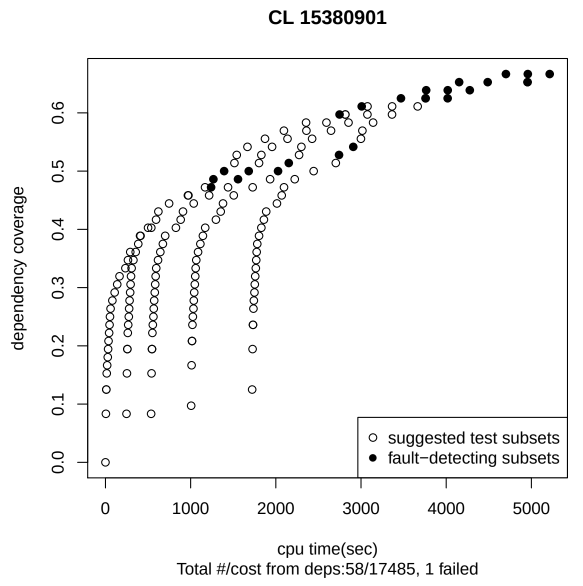

CL 15357902
dependency coverage
0.0
0.2
0.4
0.6
0.8
0
5000
10000
15000
○ suggested test subsets
● fault−detecting subsets
cpu time(sec)
Total #/cost from deps:455/21724, 2 failed

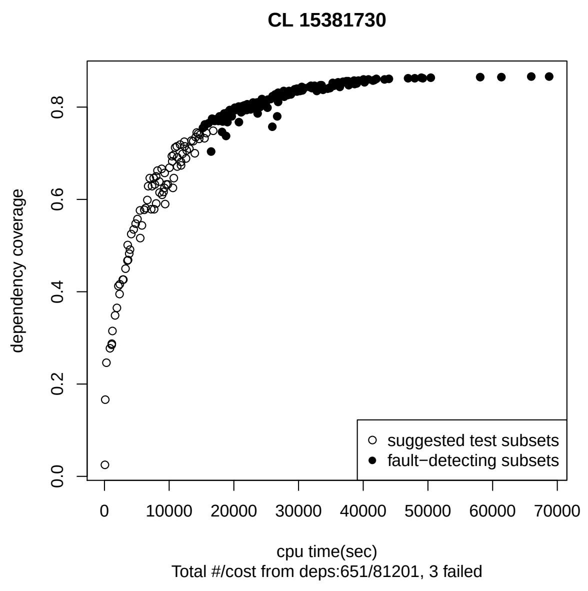

CL 15358312
dependency coverage
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0
5000
10000
15000
20000
○ suggested test subsets
● fault−detecting subsets
cpu time(sec)
Total #/cost from deps:680/26378, 0 failed

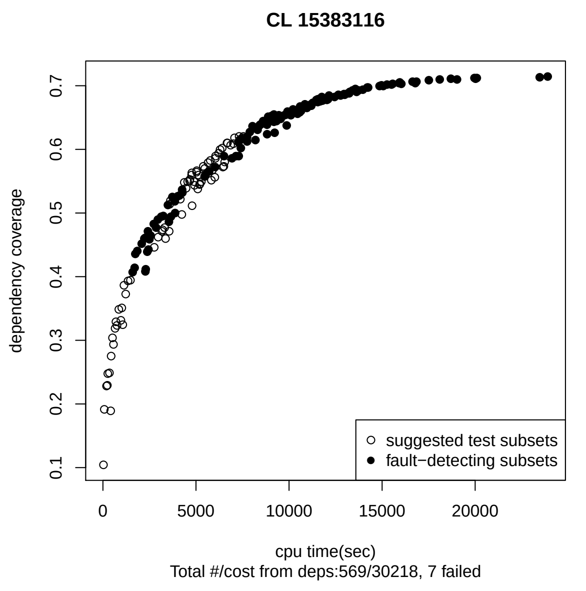

CL 15364723
deendency coverage
0.0
0.2
0.4
0.6
0.8
0
500
1000
1500
○ suggested test subsets
● fault−detecting subsets
cpu time(sec)
Total #/cost from deps:89/8775, 19 failed

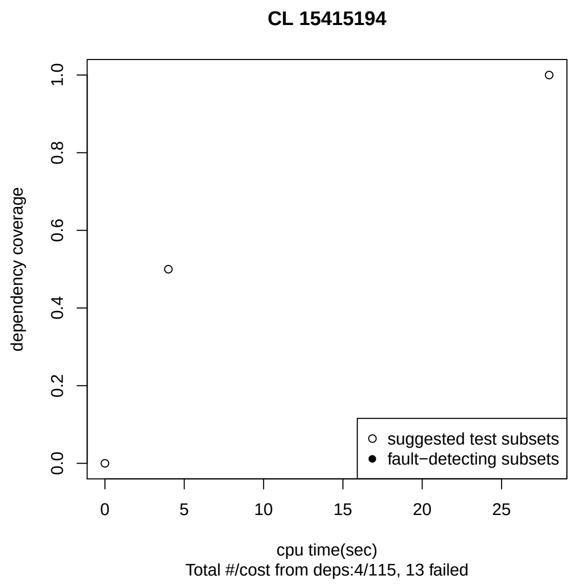

CL 15373479
dependency coverage
0.2
0.4
0.6
0.8
0
5000
10000
15000
○ suggested test subsets
● fault−detecting subsets
cpu time(sec)
Total #/cost from deps:334/55078, 1 failed

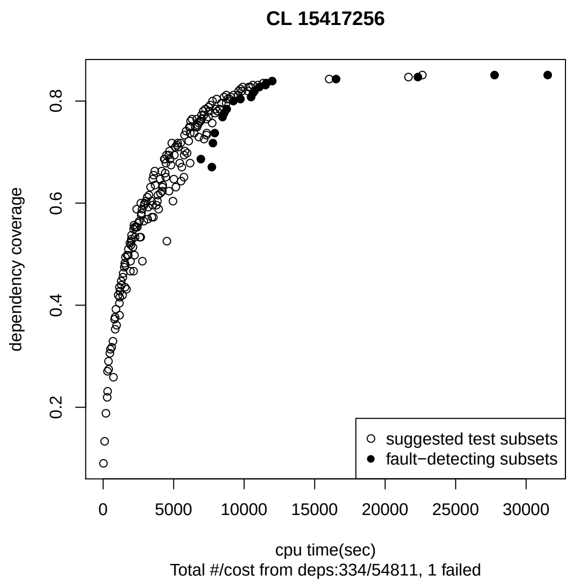

CL 15378688
dependency coverage
0.0
0.2
0.4
0.6
0.8
0
500
1000
1500
○ suggested test subsets
● fault−detecting subsets
cpu time(sec)
Total #/cost from deps:144/6989, 0 failed

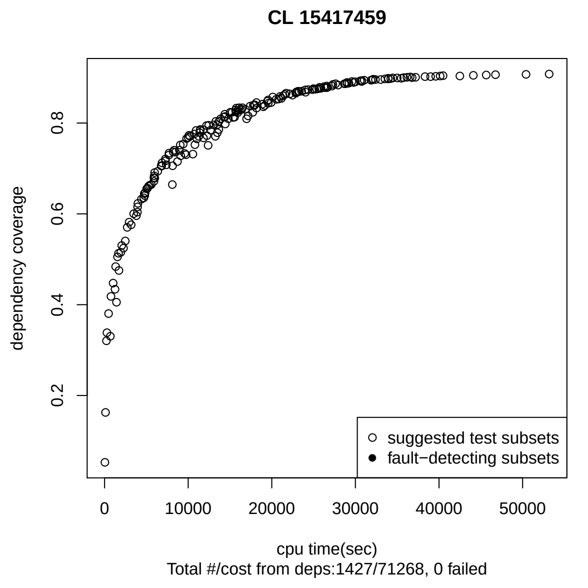

CL 15380901
dependency coverage
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
1000
2000
3000
4000
5000
○ suggested test subsets
● fault−detecting subsets
cpu time(sec)
Total #/cost from deps:58/17485, 1 failed

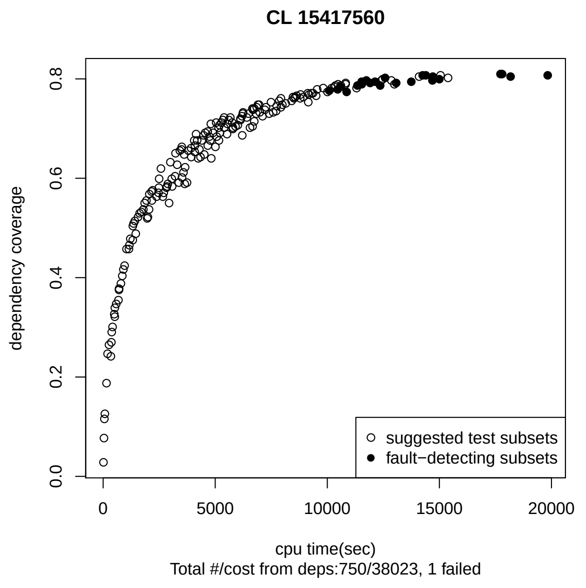

CL 15381730
dependency coverage
cpu time(sec)
Total #/cost from deps:651/81201, 3 failed
suggested test subsets
fault−detecting subsets

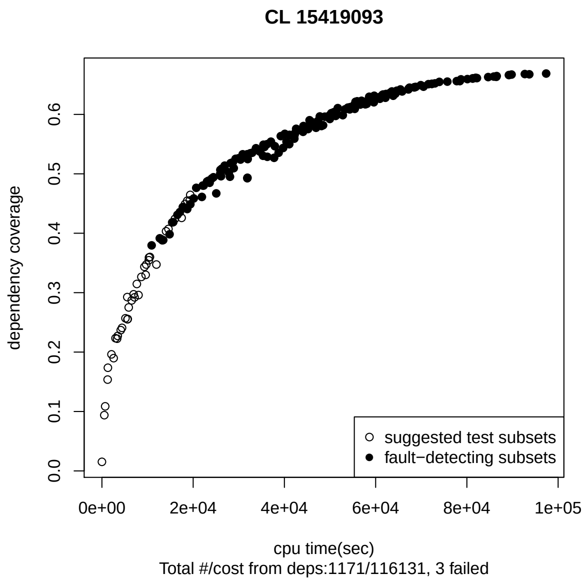

CL 15383116
dependency coverage
cpu time(sec)
Total #/cost from deps:569/30218, 7 failed
suggested test subsets
fault−detecting subsets

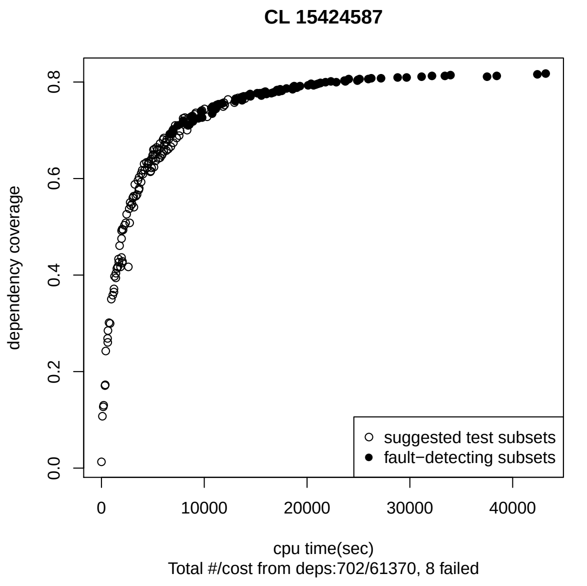

CL 15415194
dependency coverage
cpu time(sec)
Total #/cost from deps:4/115, 13 failed
suggested test subsets
fault−detecting subsets

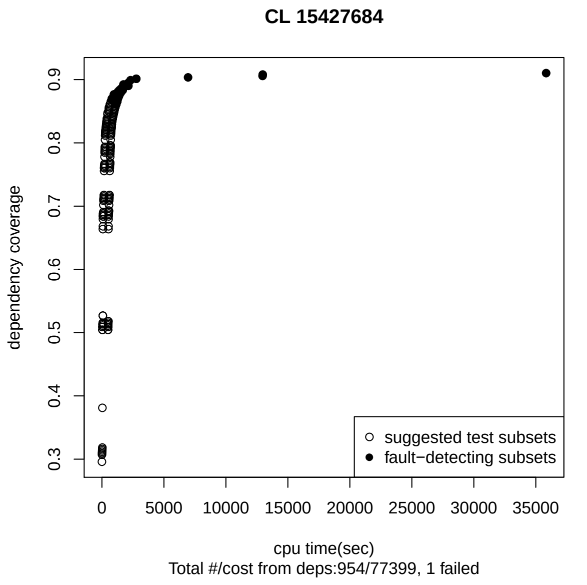

CL 15417256
dependency coverage
cpu time(sec)
Total #/cost from deps:334/54811, 1 failed
suggested test subsets
fault−detecting subsets

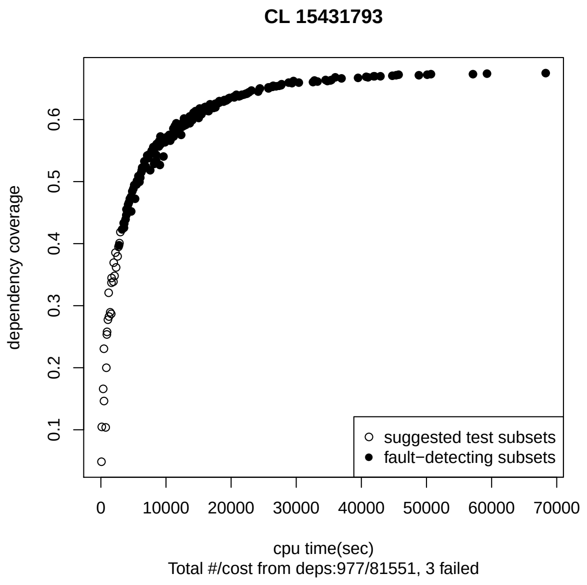

CL 15417459
dependency coverage
cpu time(sec)
Total #/cost from deps:1427/71268, 0 failed
suggested test subsets
fault−detecting subsets

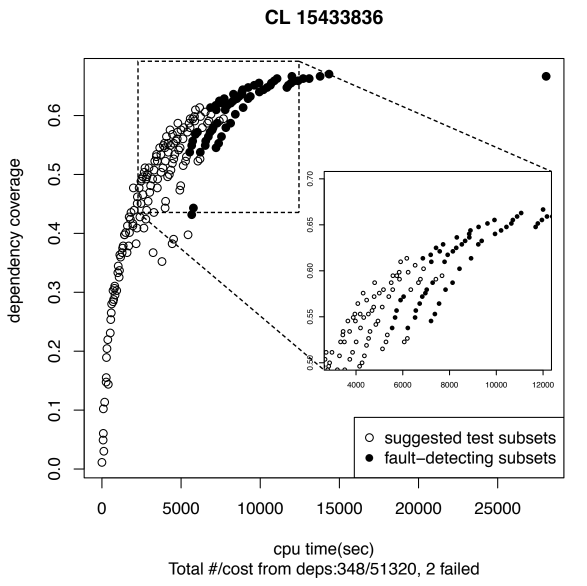

CL 15417560
dependency coverage
cpu time(sec)
Total #/cost from deps:750/38023, 1 failed
suggested test subsets
fault−detecting subsets

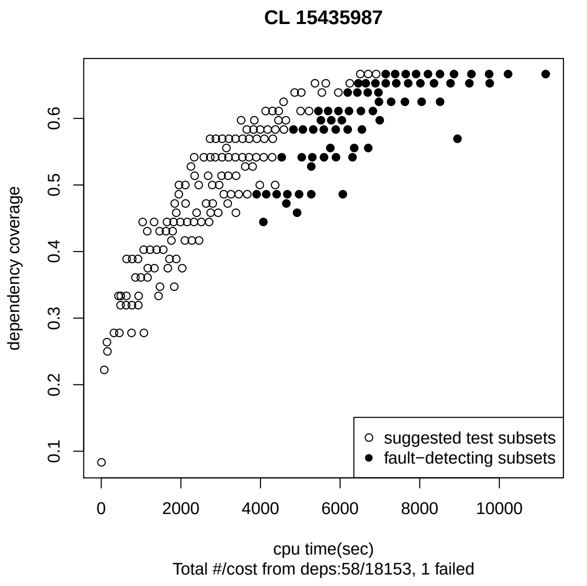

CL 15419093
dependency coverage
cpu time(sec)
Total #/cost from deps:1171/116131, 3 failed
○ suggested test subsets
● fault−detecting subsets


CL 15424587
dependency coverage
cpu time(sec)
Total #/cost from deps:702/61370, 8 failed
○ suggested test subsets
● fault−detecting subsets


CL 15427684
dependency coverage
cpu time(sec)
Total #/cost from deps:954/77399, 1 failed
○ suggested test subsets
● fault−detecting subsets


CL 15431793
dependency coverage
cpu time(sec)
Total #/cost from deps:977/81551, 3 failed
○ suggested test subsets
● fault−detecting subsets


CL 15433836
dependency coverage
cpu time(sec)
Total #/cost from deps:348/51320, 2 failed
○ suggested test subsets
● fault−detecting subsets


CL 15435987
dependency coverage
cpu time(sec)
Total #/cost from deps:58/18153, 1 failed
○ suggested test subsets
● fault−detecting subsets