AN ABSTRACT OF THE THESIS OF

Xuemei Qiu for the degree of Master of Science in Computer Science

presented on December 5, 2002.

Title:

Exploring the Impact of Test Suite Granularity and Test Grouping Technique on the Cost-Effectiveness of Regression Testing

Redacted for Privacy

Abstract approved: _____

Gregg Rothermel

Regression testing is an expensive testing process used to validate changes made to previously tested software. Different regression testing techniques can have different impacts on the cost-effectiveness of testing. This cost-effectiveness can also vary with different characteristics of test suites. One such characteristic, test suite granularity, reflects the way in which test cases are organized within a test suite; another characteristic, test grouping technique, involves the way in which the test inputs are grouped into test cases. Various cost-benefits tradeoffs have been attributed to choices of test suite granularity and test grouping technique, but little research has formally examined these tradeoffs. In this thesis, we conducted several controlled experiments, examining the effects of test suite granularity and test grouping technique on the costs and benefits of several regression testing methodologies across ten releases of a non-trivial software system, empire. Our results expose essential tradeoffs to consider when designing test suites for use in regression testing evolving systems.

Exploring the Impact of Test Suite Granularity and Test Grouping Technique on the Cost-Effectiveness of Regression Testing

by

Xuemei Qiu

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 5, 2002
Commencement June 2003

Master of Science thesis of Xuemei Qiu presented on December 5, 2002

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Chair of the Department of Computer Science

Redacted for Privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Xuemei Qiu, Author

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my major professor, Dr. Gregg Rothermel, for all of his valuable advice, help and support throughout the course of this work. In addition, special thanks due to my thesis committee, Dr. Margaret Burnett, Dr. Prasad Tadepalli and Dr. Jonathan King, for their insightful suggestions. I also thank Alexey Malishevsky for his help on the experiment tools.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF APPENDIX FIGURES

Figure — Page

LIST OF APPENDIX FIGURES (Continued)

Figure Page

LIST OF APPENDIX FIGURES (Continued)

LIST OF APPENDIX FIGURES (Continued)

Figure | Page

LIST OF APPENDIX FIGURES (Continued)

Figure | Page

LIST OF APPENDIX FIGURES (Continued)

LIST OF APPENDIX FIGURES (Continued)

LIST OF APPENDIX TABLES

EXPLORING THE IMPACT OF TEST SUITE GRANULARITY AND TEST GROUPING TECHNIQUE ON THE COST-EFFECTIVENESS OF REGRESSION TESTING

CHAPTER 1

INTRODUCTION

Regression testing is a testing technique in which previously tested components/functionality of a software system are re-tested to ensure that they function properly as the system evolves. Regression testing attempts to ensure that the operational performance of the system is not degraded due to changes in system components.

One regression testing technique is to rerun all existing test cases after a software modification. When only a small portion of the software has been modified, however this *retest-all* approach can waste resources running unnecessary tests. Techniques for selective software retesting can help reduce these costs. Several different regression test selection techniques [5, 16, 24] have been proposed for use in selecting a subset of an existing test suite. The relative costs and benefits of different selection techniques have also been studied [8, 20, 22, 28]. Although these techniques can produce savings, they can also have substantial costs, and can discard tests that could reveal faults, possibly reducing fault detection effectiveness.

Test case prioritization is another technique that can aid in regression testing [7, 27, 32]. Test case prioritization orders test cases with the intent of improving various performance goals. For example, test cases might be scheduled to place test cases that achieve high coverage of statements or decisions earlier, or exercise features in order of frequency of use. More frequently, test case prioritization techniques are viewed as a way to help increase a test suite's rate of fault detection. An increased rate of fault detection can provide earlier feedback on the system and let developers begin locating and correcting faults earlier.

Finally, test suite reduction techniques [26, 30] can reduce regression testing costs by eliminating redundant test cases from test suites. A reduction in the size of the test suite decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software. However, a potential drawback of this technique is that in reducing a test suite we may significantly alter the fault-detecting capabilities of that test suite [26].

Different regression testing techniques can have different impacts on the cost-effectiveness of testing. This cost-effectiveness can also vary with different characteristics of test suites. One such characteristic is test suite granularity, which reflects the way in which test cases are organized within a test suite. For example, a test suite for a graphic image editor might contain just a few test cases that start up the system, open a file, issue hundreds of editing and formatting commands, and close the file, or it might contain hundreds of test cases that each issue only one or two commands. A test suite for a speech recognition system might contain several test cases that each recognize hundreds of sentences, or hundreds of test cases that each recognize just one sentence. A test suite for a math class might contain a few test drivers that each invoke

dozens of functions, or dozens of drivers that each invoke a few functions.

This illustrates a choice test engineers need to make: use many simple test cases in a test suite or fewer more complex test cases? Kaner et al. [12] suggest that large tests can save time, provided they are not overly complicated, in which case simpler tests may be more efficient. Kit [14] suggests that when testing valid inputs for which failures should be infrequent, large test cases are preferable. Bach [1] states that small test cases cause fewer difficulties with cascading errors, but large test cases are better at exposing system level failures involving interactions between software components. We describe this as a matter of *granularity.* (We define this concept more precisely in Chapter 2.)

Beyond informal suggestions, there has been little formal study examining the effects of test suite granularity on regression testing. Recently, however, Rothermel et al. [21] designed and conducted a controlled experiment to observe the impact of test suite granularity on the cost-effectiveness of several regression testing methodologies: retest-all, regression test selection, test case prioritization and test suite reduction. The results exposed several tradeoffs to consider when designing test suites for use in regression testing evolving systems.

This thesis reports results of a replicated study, based on the empirical study by Rothermel et al. [21]. Several enhancements were implemented for this study to further explore the cost-benefits tradeoffs associated with test suite granularity. Ten versions of a large software system, together with faulty variants of these versions, were studied as subjects. Seven different granularity levels were used in the study instead of the four used in the previous study. In addition to considering random groupings of tests, this study also considers function-based groupings, described further in Chapter 2. Additional regression testing techniques were also added. Together, these enhancements help us extend the

generality of the earlier results and explain additional factors.

Following this introduction, we provide background information and review previous work relevant to this paper. Chapter 3 describes our experiment method, presenting our hypothesis, measures, and experiment materials and design. Chapter 4 presents the experiment results and data analysis. Chapter 5 discusses the results and Chapter 6 concludes and discusses future work.

CHAPTER 2

OVERVIEW AND RELATED WORK

In this chapter, we review related background information and previous work. First, following Binder [3], we define a *test case* to consist of a pretest state of the system under test (including its environment), a sequence of test inputs, and the expected test results. We define a *test suite* to be a set of test cases.

To define *test suite granularity* we begin by drawing on some examples. When test engineers design test cases for a system, first, they need to identify various testing requirements for that system, such as specification items, code elements, or method sequences. Next, they must construct test cases that exercise these requirements. An engineer testing a graphic image editor might specify sequences of editing and formatting commands, an engineer testing a speech recognition system might design sample recognition models, and an engineer testing a math class might develop drivers that invoke functions. The practical questions these engineers face include: how many editing and formatting commands to include per sequence, how many sentences to include in each recognition model, and how many functions to invoke in each driver.

The answers to these questions involve many cost-benefits tradeoffs. For example, if the cost of performing initialization and setup activities for individual test cases dominates the cost of executing those test cases, a test suite containing a few large test cases can be more cost effective than a suite con-

taining many small test cases. Large test cases might also be better than small ones at exposing failures caused by interactions among system functions. Small test cases, on the other hand, can be easier to use in debugging than large test cases, because they reduce occurrences of cascading errors [1] and simplify fault localization [10]. Furthermore, grouping test inputs into large test cases may prevent test inputs that appear later in the test cases from exercising the requirements they are intended to exercise, by causing later inputs to be applied from system states other than those intended.

The above examples somewhat involve *test case size*, a term used informally in [1, 2, 12, 14] to denote notions such as the number of commands applied to, or the amount of input processed by, the program under test, for a given test case. However, there is actually more than just test case size involved: when engineers increase or decrease the number of requirements covered by each test case, this directly determines the number of individual test cases that must be created to cover all the requirements. The interaction between test case size and the number of test cases is a key factor in creating most of the cost-benefits tradeoffs just discussed.

In Rothermel et al. [21], the authors study how test inputs are grouped into test cases within a test suite in the course of designing that suite to cover requirements. The term *test suite granularity* is used to describe a partition on a set of test inputs into a test suite containing test cases of a given size. We use the same term in this study, and Section 3.2.1.2 provides a more precise description and measure of test suite granularity that reflects this usage.

Other definitions of test case, test case size, and test suite granularity are possible. Test engineers might choose to view the individual inputs applied during a single invocation of a graphic image editor, or the individual function

invocations made from within a math class, as individual test cases, each with its own size. Also, in practice, test suites may contain test cases of varying sizes. In this study, however, we follow the same definitions as in Rothermel et al. [21]. These definitions facilitate the controlled study of the cost-benefits tradeoffs outlined above. Moreover, our definitions are suitable for use with the program we use as a subject in our experiments. Alternative definitions could be used in the future to examine other types of subjects and methodologies.

2.1 Regression Testing and Methodologies

In this thesis, we replicate the empirical study of Rothermel et al. [21] with several enhancements. We wish to further investigate the effects of test suite granularity on the costs and effectiveness of testing activities across the software lifecycle, i.e., in relation to regression testing.

Let P be a program, P' be a modified version of P, and T be a test suite developed for P. Regression testing seeks to test P'. To perform such testing, test engineers may re-use some or most of T, but new test cases may also be needed to test new functionalities. Both reuse of T and creation of new test cases are important. However, as test reuse motivates most suggestions about costs and benefits of test case size, it is our main concern in this study.

In particular, we consider four methodologies related to regression testing and test reuse: retest-all, regression test selection, test suite reduction, and test case prioritization.

2.1.1 Retest-all

When P is modified to create P', test engineers may simply reuse all non-obsolete test cases in T to test P'; this is known as the *retest-all* technique [15]. Test cases in T that no longer apply to P' are *obsolete*, and must be reformulated or discarded [15]. The retest-all technique represents typical current practice [18], and thus, serves as our *control technique.*

2.1.2 Regression Test Selection

The *retest-all* technique can be expensive: rerunning all test cases may require an unacceptable amount of time or human effort. *Regression test selection* (RTS) techniques [5, 16, 24] use information about P, P', and T to select a subset of T with which to test P'. (For a survey of RTS techniques, refer to [23].) Empirical studies of some RTS techniques [5, 8, 25, 22] have shown that they can be cost-effective.

One cost-benefits tradeoff among RTS techniques involves *safety* and *efficiency.* Safe RTS techniques guarantee that, under certain conditions, test cases not selected could not have exposed faults in P' [23]. Achieving safety, however, may require inclusion of a larger number of test cases than can be run in available testing time. Non-safe RTS techniques sacrifice safety for efficiency, selecting test cases that, in some sense, are more useful than those excluded.

2.1.3 Test Suite Reduction

As P evolves, new test cases may be added to T to validate new functionalities. As T grows over time, some test cases may become redundant in terms of code or functionality exercised. *Test suite reduction* techniques [4, 9, 17] address

this problem by using information about P and T to permanently remove redundant test cases from T, so that subsequent reuse of T can be more efficient. Test suite reduction thus differs from regression test selection in that the latter does not permanently remove test cases from T, but simply "screens" those test cases for use on a specific version P' of P, retaining unused test cases for use on later releases.

By reducing test-suite size, test-suite reduction techniques reduce the costs of executing, validating, and managing test suites over future releases of the software. However, there is a potential drawback for this technique. The removal of test cases from a test suite may permantly damage the fault-detecting capabilities of that test suite. Some studies [26] have shown that test suite reduction can significantly reduce the fault-detection effectiveness of test suites. Other studies [31] have shown that test-suite reduction can produce substantial savings at little cost to fault-detection effectiveness.

2.1.4 Test Case Prioritization

Test case prioritization techniques [7, 27, 32] schedule test cases so that those with the highest priority, according to some criterion, are executed earlier in the regression testing process than lower priority test cases. For example, software engineers might wish to schedule test cases in an order that achieves code coverage at the fastest rate possible, reveals faults as early as possible, or exercises features in order of expected frequency of use.

Empirical results [7, 27, 32] suggest that several simple prioritization techniques can significantly improve the rate at which test suites detect faults. An improved rate of fault detection during regression testing provides earlier feed-

back on the system under test and lets testers begin locating and correcting faults earlier than might otherwise be possible.

2.2 Related Work

Many papers have examined the costs and benefits of regression test selection, test case prioritization, and test case reduction [5, 7, 8, 13, 26, 31]. Several textbooks and articles on testing [1, 2, 7, 10, 12, 14, 26] have discussed tradeoffs involving test suite granularity. None of these, however, have formally examined tradeoffs involving test suite granularity, much less done so in relation to regression testing, prior to Rothermel et al. [21].

In [25, 22], test suite granularity is specifically treated as a factor in two studies of regression test selection, and test suites constructed from smaller test cases are shown to facilitate selection. These studies, however, measured only numbers of test cases selected, and considered only safe RTS techniques. In [21], the authors present the results of a controlled experiment designed specifically to examine the impact of test suite granularity on the costs and savings associated with several regression testing methodologies, across several metrics of importance. This thesis replicates the granularity study and furthermore, enhances it with more versions of the test subject, more granularities and more regression tesing techniques.

CHAPTER 3

EXPERIMENT DESIGN

3.1 Hypotheses

Informally, the research question investigated in this thesis can be stated as, "how does test suite granularity affect the costs-benefits tradeoffs for different regression testing techniques across software release cycles and test grouping techniques?" More formally, we can express this question as null hypotheses for three methodologies - regression test selection, test suite reduction and test case prioritization - at a 0.05 significance level:

H1 (test suite granularity): Test suite granularity does not have a significant impact on the costs-benefits tradeoffs for different regression testing techniques within each methodology.

H2 (regression testing technique): Different regression testing techniques do not significantly affect the selected costs-benefits measures.

H3 (grouping technique): Different test grouping techniques do not significantly affect the costs and benefits of regression testing methodologies.

In order to test our hypotheses, several controlled experiments were designed and implemented. The following sections present the measures, materials, experiment design and threats to validity.

3.2 Measures

3.2.1 Independent Variables

There were three independent variables manipulated in these experiments: regression testing technique, test grouping technique and test suite granularity.

3.2.1.1 Regression Testing Techniques

There were between two and five techniques studied for each of the regression testing methodologies. These techniques were selected based on three goals: (1) to include techniques that could serve as practical experimental controls, (2) to include techniques that could easily be implemented by practitioners, and (3) to include techniques representative of the main classes of techniques proposed to date.

Regression test selection. Four types of RTS techniques were chosen for the study. They were retest-all, safe, modified non-core entity and minimization techniques. As described in the previous chapter, the retest-all technique executes every test case in T on P', and is our control technique, representing typical current practice. Safe techniques select, under certain conditions, every test in T that could potentially expose one or more faults in P' [8]. As a safe technique we employed the modified entity technique [21], which selects test cases that exercise functions, in P, that (1) have been deleted or changed in producing P', or (2) use variables or structures that have been deleted or changed in producing P'. Our third technique is the modified non-core entity technique [21], which works like the modified entity technique, but ignores "core" functions, which are defined as functions exercised by more than $k\%$ of the test cases in the test suite (we set k to be 80%). Since selecting all test cases through core functions may lead to selecting all of T, this technique trades

safety for savings in the re-testing effort. Minimization is our fourth technique, newly added for this study. This technique attempts to select minimal sets of tests, from T, that yield coverage of modified or affected functions of P.

Test suite reduction. We selected two test suite reduction techniques for use in this study: no reduction and GHS reduction. The no reduction technique is equivalent to the retest-all technique; it represents current typical practice and acts as our control technique. The GHS reduction technique is an heuristic developed by Harrold, Gupta and Soffa [9] that selects a representative set of test cases that is a subset of T, but still provides the same testing coverage of P as T. Coverage here means any method of completeness with respect to a test selection criterion. We used a function coverage criterion in this study.

Test case prioritization. We selected six test case prioritization techniques for use in this study: random prioritization, optimal prioritization, total function coverage prioritization (tcov-func), additional function coverage prioritization (acov-func), and two diff-based prioritization (tcov-bdiff and acov-bdiff). Random prioritization places test cases in T in random order and is our control technique, and is equivalent to the retest-all technique because we always randomly ordered the test cases before using them. For optimal prioritization [27], given program P and a set of known faults for P, if we can determine, for test suite T, which test cases in T expose which faults in P, then we can determine an optimal ordering[1] of the test cases in T for maximizing T's rate of fault detection for that set of faults. In reality, this is not a practical technique,

---

[1] The problem of calculating an optimal ordering is itself intractable, thus a heuristic that calculates an approximation to optimal [27] is employed.

since we do not know which test cases reveal which faults beforehand. However, this technique provides an upper bound on prioritization benefits. Total function coverage prioritization [27] prioritizes test cases in terms of the total number of functions they cover, by counting the number of functions covered by each test case, then sorting the test cases in descending order of that number. When multiple test cases cover the same number of functions, an additional rule is necessary to order these test cases; we order them randomly. Additional function coverage prioritization [27] iteratively selects a test case that yields the greatest function coverage, then adjusts the coverage information on subsequent test cases to indicate their coverage of functions not yet covered, and then repeats this process until every function is covered by at least one test case in T. When all functions have been covered, this process is repeated on the remaining test cases.

Our fifth and sixth techniques, diff-based prioritization techniques, are newly added for this study. These techniques utilize information about modifications between program versions to prioritize test cases. Faults are not equally likely to exist in each function. With regard to regression testing, we are interested in the potential of modifications to lead to regression faults. We use a binary value regarding a function's modification status (BDIFF) to measure the probability this function contains a fault. There are also other types of diff-based prioritization techniques using different measures, such as a fault index (FI), or a number of syntactic changes (DIFF).

3.2.1.2 Test Suite Granularity

Our objective in this study was to quantify the impact of varying test suite granularities on the costs and benefits of regression testing techniques. To do

this, we need to obtain test suites with different granularities. But at the same time, we need to control other factors that might affect our dependent variables.

We considered two possible ways to obtain test suites of varying granularities. The first approach is to obtain or construct test suites for a program, partition them into subsets according to size, and compare the results of executing these different subsets. However, this approach would not let us establish a causal relationship between test suite granularity and measures of costs or benefits, because it does not control for other factors that might influence those measures. For example, suppose we partition T into subsets T_1 containing test cases of size s, and T_2 containing test cases of size ks. Suppose we then compare costs and benefits of T_1 and T_2, and find out that they differ. Under this circumstance, we cannot determine whether this difference was caused by different test suite granularities, or by other factors such as the different number or type of inputs applied in T_1 and T_2. It might be the case that all the modified functions are exercised only by inputs in T_1, and not in T_2.

The second approach is to construct test suites of varying granularities by sampling a single pool or "universe" of *test grains*. A *test grain* is a smallest possible input that could be used as a test case for a target program. Each test grain is applied from a start state to produce a checkable output. We can employ a sampling procedure to randomly select test grains and create test cases of different sizes: a test case of size s consists of s test grains. Applying the above sampling procedure repeatedly to a universe of n test grains, without replacement, until no test grains remain, partitions the universe into n/s test cases of size s, and possibly one smaller test case. This yields a test suite of granularity level s. Repeating this procedure for different values of s gives us test suites of different granularity levels. This approach controls other factors,

like differences in types and numbers of inputs, and makes it possible to compare results across different granularity levels.

Because it provides better control, we selected the second approach for this study. We employed seven granularity levels: 1, 2, 4, 8, 16, 32 and 64, which we refer to as G1, G2, G4, G8, G16, G32 and G64, respectively. This is an enhancement in comparison to the previous study, in which only four granularity levels: G1, G4, G16 and G64, were studied.

3.2.1.3 Grouping Techniques

As just described, we treated all the test grains as a single pool or "universe", and then we employed a sampling procedure to randomly select test grains to create test cases of different sizes. Applying this sampling procedure repeatedly to a universe of n test grains yields a test suite of granularity level s. We call this sampling procedure "random grouping", indicating that the test cases are randomly selected and grouped from the universe. This was the only grouping technique used in the previous study. We added a new grouping technique in this study, "functional grouping".

Functional grouping means that the test suites of different granularity levels are chosen from tests of the same functionality. We considered two possible algorithms for creating functional groupings. One algorithm we could use is: within each bucket (each group of tests testing the same user function), for granularity level s test suites, pick groups of size s without replacement until fewer than s test grains remain, then let the remaining test grains be a group. The problem with this algorithm is that we might have a large number of test suites of size less than s at each granularity level. For instance, if there are M buckets and each bucket contains $s + 1$ test grains, for each bucket we would

have a suite of 1 test, for a total of M suites of 1 test, and also M suites of size s. More suites would be created than in the random partition, and when comparing results across grouping techniques, the effects of functional grouping tests at granularity level s may be lost due to the presence of size 1 tests.

Just as we have controlled, in our random groupings, for using the same set of inputs in test suites at all granularity levels, we need to control for using the same number of test cases at a specific granularity level, and for the suites at granularity level s truly having size s. Thus we considered an alternative algorithm: we begin by selecting as many size s suites from within each bucket as possible, then take the remaining test grains, and group them randomly into size s tests.

We used this alternative algorithm in our study. One problem in choosing test groupings from the same functions for our subject is that some functions have only a few (1, 2 or 3) test grains and some have more (about 30), but few or none have over 30 grains. So we cannot obtain G64 suites by our functional grouping approach, and some suites at lower granularities chosen from within the same functions also cannot be full size. But we still generated test suites all the way up to granularity 64. We expect that as granularity level goes higher, the effects related to grouping per functionality won't be as large because large test cases will include many inputs that are not related to a specific functionality. This might reflect reality, and also show a limiting aspect of large tests that we could not examine in our previous study.

3.2.2 Dependent Variables

We need to measure the costs and benefits of regression testing methodologies in order to investigate our hypotheses. We considered three measures: savings in test execution time, costs in fault-detection effectiveness, and savings in rate of fault detection, to assess the costs and benefits of regression testing techniques. Our first two measures are used for regression test selection and test case reduction techniques, and our third measure is used for test case prioritization.

3.2.2.1 Savings in Test Execution Time

Regression test selection and test suite reduction techniques reduce the number of test cases that need to be executed on P', thereby reducing the costs required to retest P' and achieve savings. We also expect that the use of higher test suite granularities can produce savings in test execution and validation time. In this study, to evaluate these savings, we measure the time used to execute and validate the test cases in the original test suite, the selected test suites from different regression test selection techniques, and reduced test suites from test case reduction, across different granularities. Here, by "validate", we mean using the Unix "diff" utility to compare the output of a set of regression tests on P' with their outputs on P.

3.2.2.2 Costs in Fault-detection Effectiveness

Regression test selection and test suite reduction techniques attempt to lower costs by running a subset of an existing test suite, but one potential cost is that some fault-revealing test cases may be discarded. This cost could also vary with different test suite granularity levels. We can measure costs in fault-detection ef-

fectiveness by investigating programs with known faults. To determine whether test selection or reduction lowers fault detection effectiveness we would like to know which test cases reveal which faults in P'. When dealing with a single fault, one common measure [8, 11] estimates whether a test case t detects fault f in P' by applying t to P' and P, where P' contains f and P does not. If the outputs of P and P' differ on t, we conclude that t reveals f.

In this study, however, we wish to study programs containing multiple faults. When a test case fails on a program that contains multiple faults, it is not always obvious exactly which fault(s) caused the failure. We must also determine which test cases could contribute to revealing which faults. One way to do this [13] is to instrument P' such that when t is run on P' we can determine, for each fault f in P', whether: (1) t reaches f, (2) t causes a change in data state following execution of f, and (3) the output of P' on t differs from the output of P on t.

One drawback of this approach is that it can underestimate the faults that could be found in practice with t. For example, suppose P' contains faults f_1 and f_2, which can each be detected by t if present alone. Suppose, however, that when f_1 and f_2 are both present in P', f_1 prevents t from reaching f_2. This approach would suggest that t cannot detect f_2. A second drawback of this approach is that testing for data state changes can be infeasible in programs that manipulate large data spaces, such as those used in this study.

For the above reasons, we chose a second approach. We activated each fault f in P' individually, executed each test case t (at each granularity level) on P', and determined whether t detects f singly by comparing whether it caused P and P' to produce different outputs. We then assumed that detection of f when present singly implies detection of f when present in combination with other faults.

This approach avoids the drawbacks of the first approach: it considers faults individually and does not need to detect data state changes. But with this approach, a *masking effect* may still be a problem: multiple faults could mask each other's effects such that no failures would occur on t. We investigated the possible magnitude of this problem by also executing our test cases on the multi-fault versions, and measuring the extent to which test cases that cause single-fault versions to fail did not cause multi-fault versions to fail. For both random and funtional groupings, only less than 1% cases have failures under singly activated faults and no failures when all faults have been activated at once. However this check does not eliminate the possibility that some subset of the faults in a multi-fault version might mask one another, and be undetected by test case t in that version even though detected singly by t.

3.2.2.3 Savings in Rate of Fault Detection

The goal of test case prioritization techniques is to improve the test suite's rate of fault detection: how rapidly a prioritized test suite detects faults. To measure rate of fault detection, Rothermel et al. [27] defined a metric, APFD, which represents the weighted average of the percentage of faults detected during the execution of the test suite. APFD values range from 0 to 100; higher values imply faster (better) fault detection rates. More formally, APFD can be expressed as follows: let T be a test suite containing n test cases, and let F be a set of m faults revealed by T. Let TF_i be the first test case in ordering T' of T which reveals fault i. The APFD for test suite T' is given by the equation:

$$\text{APFD} = 1 - \frac{TF_1+TF_2+...+TF_m}{nm} + \frac{1}{2n} \tag{3.1}$$

3.3 Experiment Materials

3.3.1 Subject Program

For this study, we obtained ten releases of `Empire`, a non-trivial software system programmed in C. `Empire` is an infrastructure for testing experimentation partially developed by Brian Davia [6] and completed for use in this study with the assistance of many people.

`Empire` is an open-source client-server internet game. `Empire` is a strategic game of world domination where the setting of the game is a world in which each player has their own country. Throughout the game, the players acquire resources, develop technology, build armies and engage in diplomacy with other players in an effort to overpower all other countries in the world. In addition to the players, each game of `Empire` is setup, started and maintained by one or more "deities" who install and activate a server. A deity has the power to perform almost any imaginable action, such as changing a mountain into a grass plain or bestowing riches upon a nation, and is responsible for using this power in a manner to ensure the game they run is both enjoyable and fair. As may be expected, certain powers are reserved only for a deity; a player doesn't have such power such as changing a mountain into grass plain.

The `Empire` software system is divided into two parts, a server and a client, which are named `emp-server` and `emp-client` respectively. `Emp-server` is essentially a transaction manager that manages all the user commands. First, `emp-server` is initialized and left running on a host server by the deity, then the players dial up using `emp-client` to connect with the server and issue commands to perform their desired actions. When `emp-server` receives a command, it processes the command, returns the appropriate output, and waits for the next

command. In this study, we used the `emp-server` portion of the `Empire` software system to conduct our experiments.

There are various reasons why we select `Empire` as our test subject. First, `Empire` meets our basic requirements for test subjects. It is a large piece of software (approximately 63,000 to 64,000 lines of non-blank, uncommented source code) written in the C programming langugae. It is used in the real world and has multiple versions (29 versions to date) available. A further reason involves the availability of source code and documentation for the system. `Empire` is distributed under the GNU General Public License Agreement, and can be downloaded at no cost. The software comes with a large amount of documentation on how to play the game. This documentation is crucial in designing test cases for use in the experiments.

As mentioned at the beginning of this section, we used ten versions of `emp-server`. Table 3.1 shows the number of functions and non-blank, non-commented lines of executable code in each version, and the number of functions changed for each version from the previous version. We consider changed functions to be those that are modified, use modified header files, are removed from the previous version, or are added to the current version.

3.3.2 Test Cases and Test Automation

To investigate our hypotheses, test cases are required for our test subject. Moreover, these test cases had to be constructed in a way that help facilitate a controlled experiment testing the effects of test suite granularity, following the methodology outlined in Section 3.2.1.1.

We used a semi-formal method called the Category Partition Method [19]

Program	*Version*	*Functions*	*Changed Functions*	*Lines of Code*
emp-server	4.2.0	1,188	—	63,014
emp-server	4.2.1	1,188	51	63,014
emp-server	4.2.2	1,197	245	63,658
emp-server	4.2.3	1,196	157	63,937
emp-server	4.2.4	1,197	9	63,988
emp-server	4.2.5	1,197	101	64,063
emp-server	4.2.6	1,197	32	64,108
emp-server	4.2.7	1,197	156	64,439
emp-server	4.2.8	1,189	52	64,381
emp-server	4.2.9	1,189	12	64,396

TABLE 3.1: The `Emp-server` Experiment Subject.

to construct our functional test cases. The `Empire` documentation files were treated as informal specifications for system functions and used to help structure the test cases. The documentation files describe the 196 commands recognized by `emp-server` together with details about the possible parameters, options and environmental effects associated with each command. Using these, we created a suite of test cases for `emp-server` that exercise each parameter, option, environmental effect, and erroneous condition described in the files.

We tried to create test cases with the smallest possible size, each using the minimum number of commands necessary to cover its target requirement. As a result, each test case consists of a sequence of between one and six lines of characters and constitutes a sequence of inputs to `emp-client`, which are then passed to `emp-server`. Because the complexity of commands, parameters, and

	emp-server
G1	1985
G2	993
G4	497
G8	249
G16	125
G32	63
G64	32

TABLE 3.2: Test Cases per Granularity Level.

environmental effects varies widely between different `Empire` commands, this process yielded between 1 and 38 test cases for each command, and ultimately produced a total of 1985 test cases. These test cases constituted our test grains, as well as our test cases of finest granularity level (G1). We then used the sampling procedure described in Section 3.2.1.1 to create test suites of granularity levels G2, G4, G8, G16, G32 and G64. Table 3.2 shows the number of test cases for each granularity level.

To execute test cases and collect fault information and timing data automatically, various test scripts were used. Given test suite T, for each test case t in T, the low level scripts for executing test cases do the following: (1) initialize the `Empire` database to a start state; (2) invoke `emp-server`; (3) invoke a `client` and issue the sequence of inputs that constitutes the test case to the `client`, saving all output returned to the `client` for use in validation; (4) terminate the `client`; (5) shut down `emp-server` and (6) save the contents of the database for use in validation.

The high level scripts for collecting fault matrices compare saved `client` output and database contents for each version containing each individual fault or all faults together with those archived for that version without faults. The comparisons of results, in which "1" indicates a difference between versions and "0" no difference, were recorded in a specific format and became our fault matrix files.

The high level scripts for collecting timing data record the running time per test, validation time per test, overall running time and overall regression testing time, and also compute the average regression testing time per test. Here when we say regression testing time, we mean the sum of running time and validation time. By design, all these test scripts can be applied to all granularity levels and all versions.

3.3.3 Faults

In this study, we intend to investigate the performance of regression testing techniques with respect to detection of regression faults - faults created in a program version as a result of the modifications that produced that version. While `Empire` comes with extensive documentation with regard to the commands available to users, it does not have adequate information files describing the known regression faults in each version of the software system. Furthermore, for reasons described in Section 3.2.2 with regard to determining fault detection effectiveness, we need to be able to turn faults on and off. This means that for each faulty section of code, we desire an alternative section that is not faulty. We could attempt to locate faults existing in each `Empire` version, but that will require quite a bit of effort and also require us to find a resolution to each fault.

Lacking the knowledge of the software system possessed by the developers, we could risk unknowingly creating additional faults in the system when trying to find resolutions to the faults.

Alternatively, an approach of manually seeding faults into the software system was chosen to perform this task. A seeded fault is a fault that is intentionally introduced into the system, soley for the purpose of investigating the performance of regression testing techniques. To obtain such faults for `emp-server`, we used our programming knowledge to insert faults in sections of the code that are modified from previous versions. The faults were seeded to be as realistic as possible, as might occur in real practice. The faults were surrounded by preprocessor statements and can be toggled on or off depending on the definition of certain variables.

Given ten potential faults seeded in each of the nine versions after the first, we activated these faults individually, and executed the test suite at all granularity levels to determine which faults could be revealed by which test cases. Table 3.3 shows, for each version and fault, the percentage of tests revealing that fault.

In our experiments, we used the fault sets in two ways. For regression test selection and reduction, we use all ten faults. For prioritization techniques, because it is known to be most effective for hard faults, we focused on those faults exposed by less than 1% of tests (35 faults out of the total of 90).

3.3.4 Additional Instrumentation

Some additional instrumentation was required to perform our experiments. The Aristotle program analysis system was used to instrument the source pro-

	v1	v2	v3	v4	v5	v6	v7	v8	v9
Fault 1	5.8	8.0	7.4	0.4	6.7	0.1	0.6	0.9	7.4
Fault 2	0.05	6.8	6.7	6.7	0.1	6.8	7.4	1.1	8.7
Fault 3	2.5	0.9	7.0	6.7	7.4	0.05	0.2	0.4	0.3
Fault 4	0.05	6.8	6.9	7.2	0.4	6.8	7.4	7.2	0.6
Fault 5	6.6	6.7	7.2	0.5	6.7	6.7	7.3	7.2	0.8
Fault 6	0.1	6.7	6.7	7.6	0.1	6.8	7.6	7.2	7.4
Fault 7	0.1	0.2	0.05	0.1	0.05	0.1	7.2	7.2	8.5
Fault 8	0.6	6.7	7.6	1.2	6.6	7.2	7.2	7.2	7.8
Fault 9	6.3	6.7	0.05	0.1	6.7	0.2	0.2	7.2	7.4
Fault 10	5.8	6.9	6.6	0.2	0.5	6.8	7.3	7.6	0.4

TABLE 3.3: Percentage of test cases that detect the faults in each version.

grams in order to provide our test coverage and control-flow graph information. Regression test selection, test suite reduction and tese case prioritization tools were created to implement the regression testing techniques described in Section 3.2.1.1. Unix utilities and manual inspections were used to determine the modified functions and functions using modified structures. Some other tools were also created to generate the functional grouping test suites, collect raw data such as traces, fault matrices, and timing data, and gather the final data.

3.4 Experiment Design

To evaluate our three hypotheses, we designed three sets of experiments with the same format. Each experiment evaluates the hypotheses for one methodology: regression test selection, test suite reduction, or test case prioritization. In

Treat. Comb.	Treat. Comb.	Treat. Comb.	Treat. Comb.	Treat. Comb.	Treat. Comb.	Treat. Comb.
$TQ_1G_1GP_1$	$TQ_1G_2GP_1$	$TQ_1G_4GP_1$	$TQ_1G_8GP_1$	$TQ_1G_{16}GP_1$	$TQ_1G_{32}GP_1$	$TQ_1G_{64}GP_1$
$TQ_1G_1GP_2$	$TQ_1G_2GP_2$	$TQ_1G_4GP_2$	$TQ_1G_8GP_2$	$TQ_1G_{16}GP_2$	$TQ_1G_{32}GP_2$	$TQ_1G_{64}GP_2$
$TQ_2G_1GP_1$	$TQ_2G_2GP_1$	$TQ_2G_4GP_1$	$TQ_2G_8GP_1$	$TQ_2G_{16}GP_1$	$TQ_2G_{32}GP_1$	$TQ_2G_{64}GP_1$
$TQ_2G_1GP_2$	$TQ_2G_2GP_2$	$TQ_2G_4GP_2$	$TQ_2G_8GP_2$	$TQ_2G_{16}GP_2$	$TQ_2G_{32}GP_2$	$TQ_2G_{64}GP_2$

TABLE 3.4: Experiment Design (TQ Stands for Technique, G for Test Suite Granularity Level and GP for Grouping).

addition, each experiment employs three factors (one for each independent variable) with multiple levels to ensure unbiased treatment assignment. Table 3.4 illustrates an example of our experiment design, showing how treatment combinations were applied to subjects. For each methodology, the behavior of various techniques (TQ) under different test suite granularity levels (G) with different test grouping techniques (GP) is investigated. This design let us quantify the impact of test suite granularity and grouping technique on the costs and benefits of different regression tessting techniques as measured by our dependent variables.

3.5 Threats to Validity

A fundamental question concerning results from an experiment is how valid the results are. It is important to consider the question of validity in the planning phase in order to plan for adequate validity of the experiment results. In this section, we describe the possible internal, external, construct, and conclusion threats to the validity of our experiments, and the approaches we used to limit their impact.

3.5.1 Internal Validity

If a relationship is observed between the treatment and the outcome, we must make sure that it is a causal relationship, and that it is not a result of a factor over which we have no control or have not measured. In other words we must make sure that the treatment causes the outcome [29].

In our study, we had to conduct a set of experiments to test our hypotheses. These experiments required a large number of processes and tools. Some of these processes involved programmers (e.g., fault seeding) and some of the tools were specifically developed for this experiment. All of these could have added threats to internal validity to our results. We adopted several procedures and tests to control and minimize the possible threats. For example, the fault seeding process was performed following a specification so that all the faults were seeded in a similar and uniform way. Also, we carefully validated new tools by testing them on small sample programs and test suites, refining them as we targeted our larger subject, and cross validating them across labs.

Having only one test suite at each granularity level might be another source of threat to internal validity. Multiple test suites at each granularity would be ideal, but this would require much more time and effort. Our procedure for generating higher granularity test suites involved randomly selecting and joining test grains, which reduces the chances of bias caused by test suite composition.

3.5.2 External Validity

External validity is concerned with generalization. If there is a causal relationship between the cause and the effect, can the result of the study be generalized beyond the scope of our study? Is there a relation between the

treatment and the outcome in this wider scope [29]?

There are two factors limiting the generalization of our results. The first factor is the quantity and quality of the subject. Using only one subject might lessen the external validity of the study, but the quality of the subject helps compensate for this. There is a large population of C programs of similar size. For example, the linux RedHat 7.1 distribution includes source code for 394 applications; the average size of these applications is 22,104 non-comment lines of code, and 19% have sizes between 25 and 75 KLOC [21]. Still, replication of these studies on other subjects could increase confidence in our results.

The second limiting factor is test process representativeness. Although the random grouping procedure is powerful in terms of control, it constitutes a simulation of the testing procedures used in industry, which might also impact the generalization of the results. Adding another test grouping technique, functional grouping, and comparing the results against random grouping helps address this problem.

3.5.3 Construct Validity

Construct validity is concerned with the relationship between theory and observation. If the relationship between cause and effect is causal, we must ensure two things: 1) that the treatment accurately reflects the construct of the cause and 2) that the outcome accurately reflects the construct of the effect.

The three dependent measures that we have considered are not sufficient to measure all aspects of the costs and benefits of regression testing methodologies thoroughly. Our measures ignore debugging costs such as the difficulty of fault localization, which could favor small granularity test suites [10, 21]. Our

measures do not consider the human costs that might be involved in executing and managing test suites. Our measures also ignore the analysis time required to perform regression testing methodologies, such as selecting, prioritizing, or reducing test cases. However, previous work [26, 22, 27, 21] has shown that for the techniques considered, analysis time is much smaller than test execution time, or analysis can be accomplished automatically in off-hours prior to the critical regression testing period.

3.5.4 Conclusion Validity

Conclusion validity is concerned with the relationship between the treatment and the outcome. We want to make sure that there is a statistical relationship, i.e. with a given significance [29].

The number of versions we considered was large enough to show significance for some of the techniques we studied, but not for others. We could use more versions, but this may not help to increase the power of the experiment.

CHAPTER 4

RESULTS - DATA ANALYSIS

In the following sections, we examine the effects of test suite granularity and test grouping technique on our three regression testing methodologies individually. Graphical representations and statistical analyses are used to present and understand the results.

4.1 Regression Test Selection

We begin by exploring the impact of test suite granularity on regression test selection techniques. To facilitate the comparison between techniques, we present, for random groupings, graphs of fault detection effectiveness for four regression test selection techniques in Figure 4.1, and graphs of test execution time for those techniques in Figure 4.2. We also present similar graphs for functional groupings in Figures 4.3 and 4.4. As discussed in Section 3.2.1.1 and also specified in the subcaptions of the figures, the four regression test selection techniques considered in our study are the retest-all technique as our control technique, the modified entity RTS technique, the modified non-core entity RTS technique, and the minimization RTS technique.

In each graph, the horizontal axis represents test suite granularity, and the vertical axis represents either fault detection effectiveness (Figures 4.1 and 4.3) or test execution time (Figures 4.2 and 4.4). Each graph contains seven data

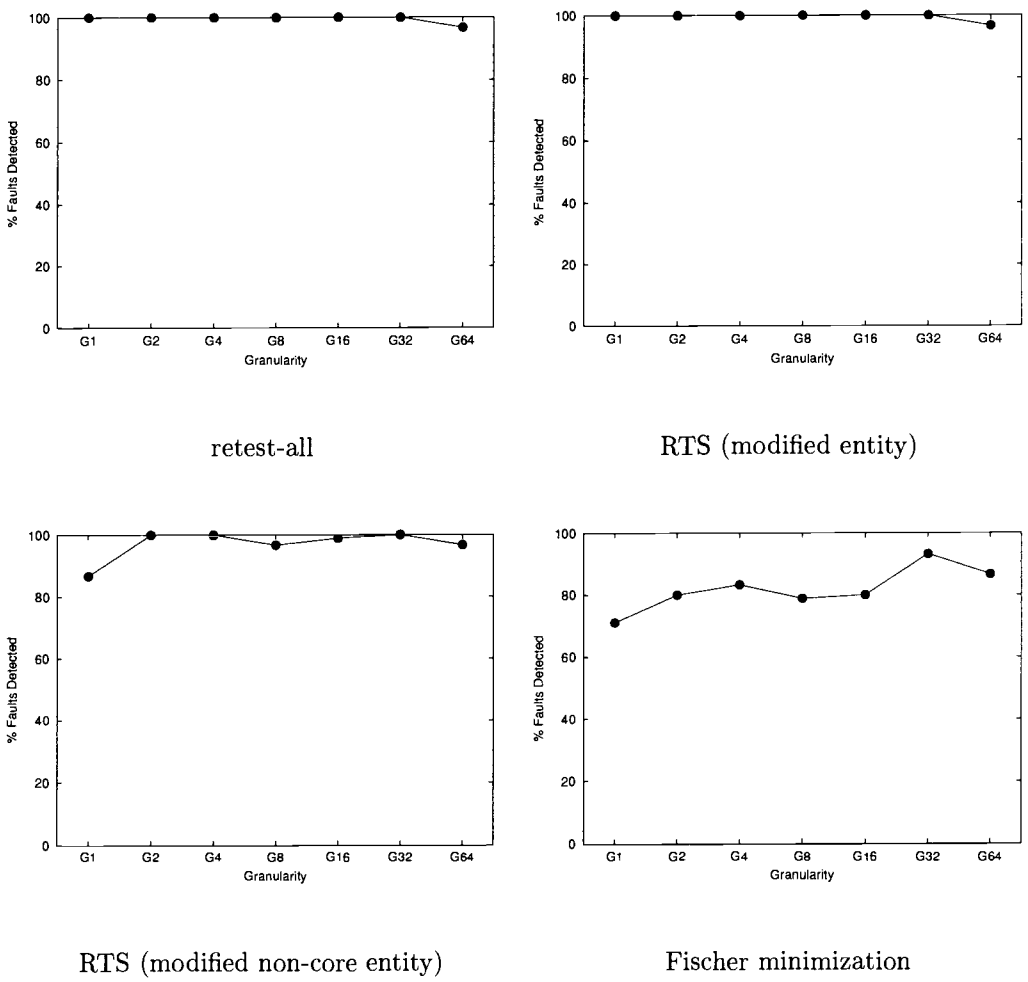
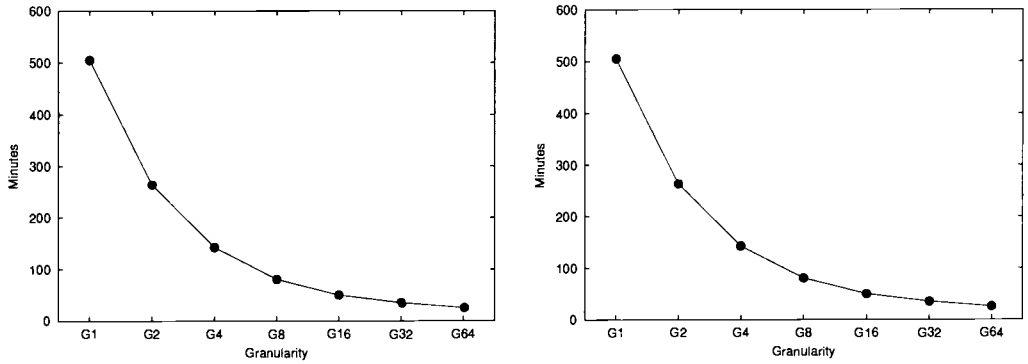
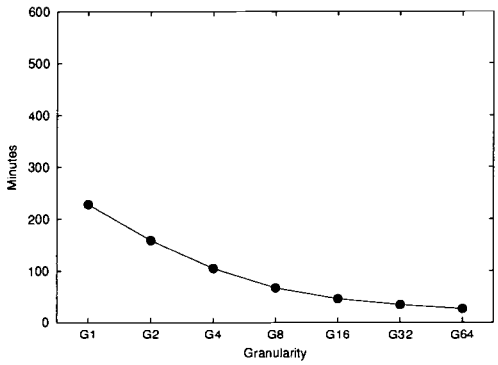


FIGURE 4.1: Fault detection effectiveness for regression test selection techniques across test suite granularities, averaged across versions, for the random grouping of test cases.

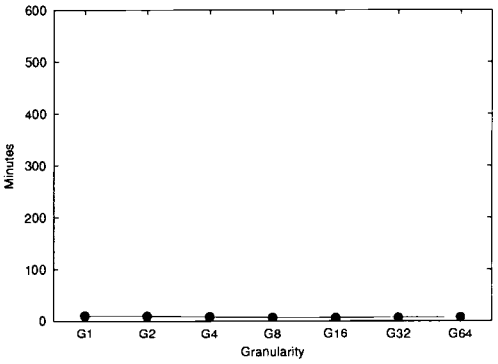


retest-all

RTS (modified entity)



RTS (modified non-core entity)



Fischer minimization

FIGURE 4.2: Test execution time for regression test selection techniques across test suite granularities, averaged across versions, for the random grouping of test cases.

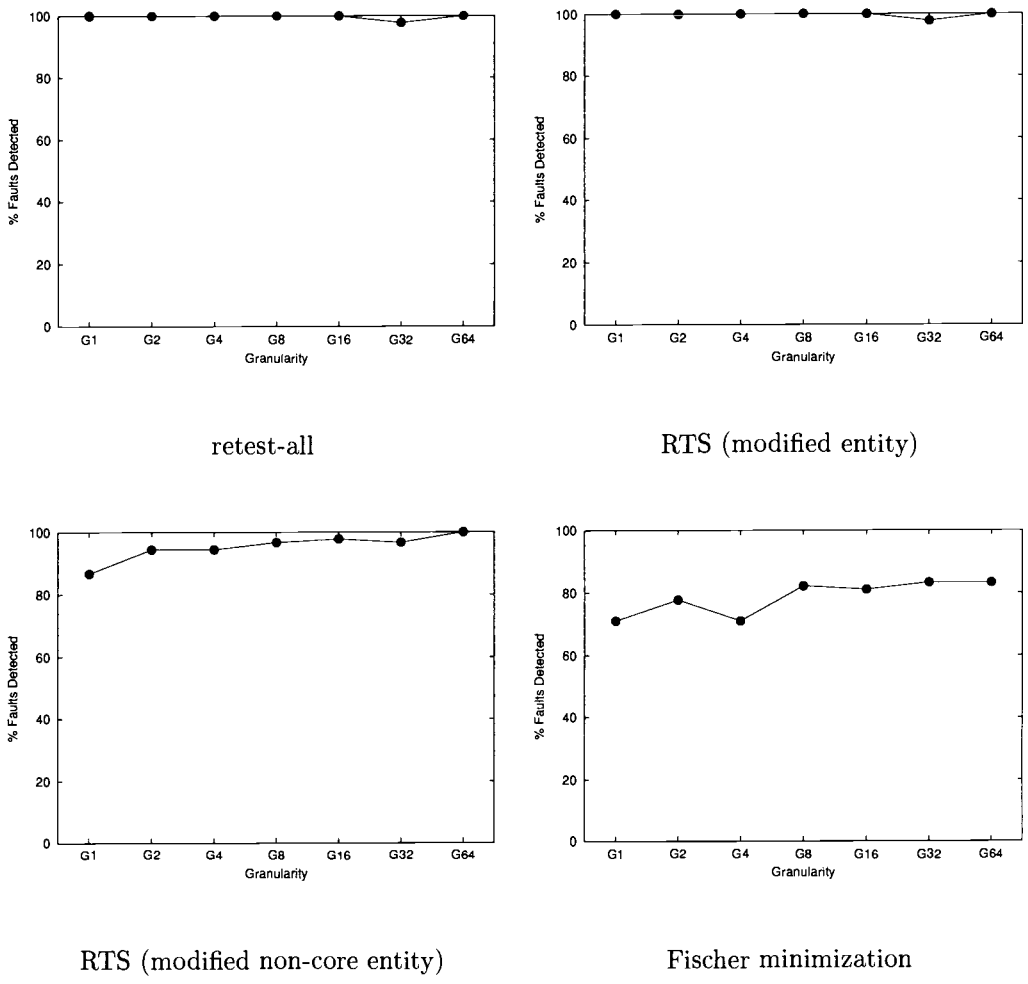


FIGURE 4.3: Fault detection effectiveness for regression test selection techniques across test suite granularities, averaged across versions, for the functional grouping of test cases.

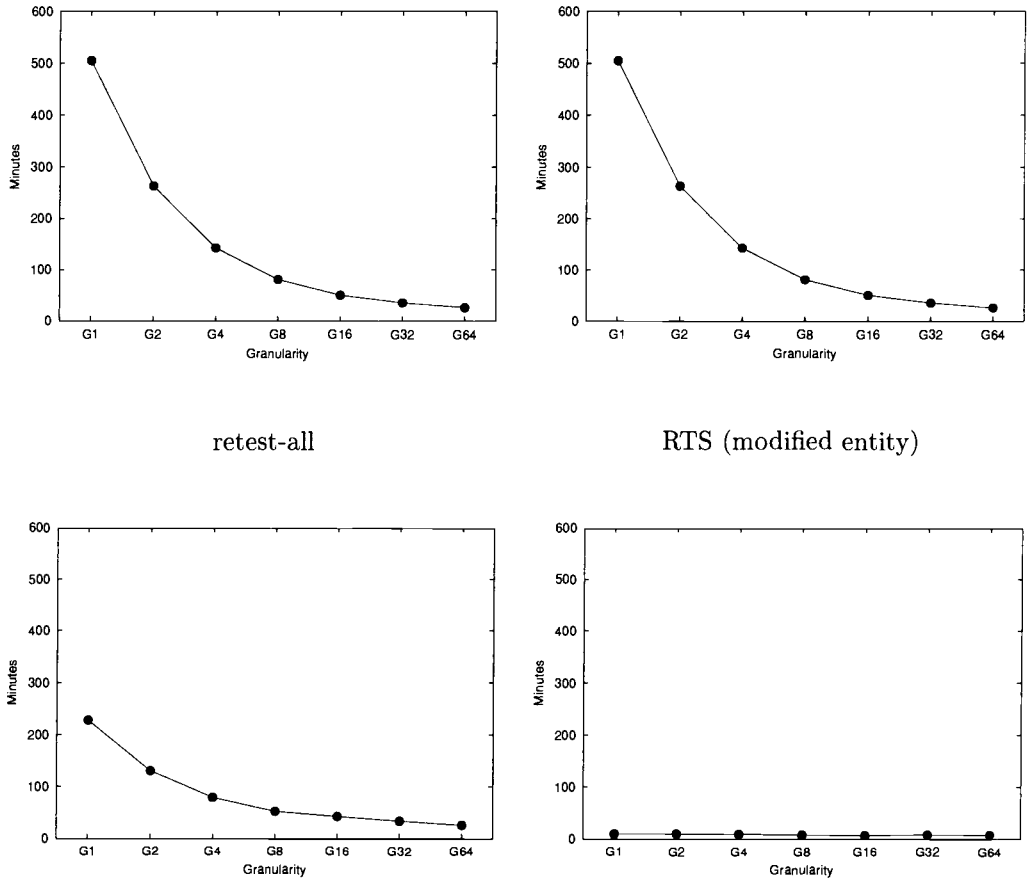


FIGURE 4.4: Test execution time for regression test selection techniques across test suite granularities, averaged across versions, for the functional grouping of test cases.

points, with each point corresponding to the average value of either fault detection effectiveness or test execution time at the specific granularity level, across all nine versions after the base version. The data points were connected with lines to assist interpretation. (Appendix A shows the complete data, listing results per version and granularity.)

For example, the upper-left graph in Figure 4.1 shows that for the retest-all technique, fault detection effectiveness was preserved at 100% from G1 to G32, and dropped to 97% for G64. The upper-left graph in Figure 4.2 shows that for the retest-all technique, test execution times decreased with granularity, ranging from 505 minutes at G1 to 26 minutes at G64.

For both random and functional test groupings, the trends observed across granularities for the modified-entity RTS technique were exactly the same as those observed for the retest-all technique. However, the modified-entity and modified non-core entity RTS techniques behaved quite differently. The modified-entity technique retained the fault detection effectiveness of the retest-all technique, but it achieved no savings in execution times across granularities. This is probably due to the fact that for most versions, changes had been made in core code and all tests went through those changes, forcing selection of all tests. In contrast to the modified entity technique, the modified non-core entity technique achieved substantial savings (55%) in test execution time at G1. These savings decreased, however, as granularity increased, and were barely noticeable (less than 6%) at granularities above G16.

For the random grouping, in terms of fault detection effectiveness, the modified non-core entity technique was nearly equivalent to the retest-all and modified entity RTS techniques at levels G2 through G64. At G1, modified non-core entity's fault detection effectiveness was reduced to 87% compared with 100%

for the other two techniques. However, for the functional grouping, the fault detection effectiveness was reduced at levels G1 through G32, ranging from 87% to 98%, and only at G64 did it achieve 100%.

The minimization RTS technique achieved substantial savings in test execution time at all granularity levels, with all selected suites executing in less than 10 minutes. But the fault detection effectiveness of this technique was lower than that of other techniques, ranging in a somewhat increasing trend from 71.1% at G1 to 86.7% at G64 for the random grouping, and from 71.1% at G1 to 83.3% at G64 for the functional grouping.

To determine whether the impact of test suite granularity on our dependent variables was statistically significant, we performed an analysis of variance (Anova) on the data. Table 4.1 presents the results of this analysis applied to the retest-all and modified non-core entity techniques, for the random test grouping. Table 4.2 presents the random grouping results for the retest-all and minimization RTS techniques. Table 4.3 presents the functional grouping results for the retest-all and modified non-core entity techniques. Table 4.4 presents the functional grouping results for the retest-all and minimization RTS techniques. Since the data for retest-all and the modified entity technique were identical, we omit comparisons of techniques to modified entity. For each dependent variable, we performed one independent analysis that includes the sources of variation considered, the sum of squares, degrees of freedom, mean squares, F value, and p-value for each source. As shown in the Anova tables, granularity and technique are two treatments applied to the dependent variable and considered as two sources of variation, and the "Error" row indicates the variation within the two treatments. The "Total" row shows the sum of all the variations. The p-value represents the smallest level of significance that would

Techniques: Modified non-core Entity and Retest-all					
Variable: *Percentage of faults detected.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	671.43	6	111.90	3.19	0.01
Technique	203.175	1	203.175	5.79	0.02
Error	4141.27	118	35.40		
Total	5015.87	125			
Variable: *Execution time.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	5.99×10^9	6	9.98×10^8	53.81	0.00
Technique	4.48×10^8	1	4.48×10^8	24.17	0.00
Error	2.19×10^9	118	1.85×10^7		
Total	8.62×10^9	125			

TABLE 4.1: Selection Anovas (Modified non-core Entity vs. Retest-all), for the Random Grouping of Test Cases.

Techniques: Minimization RTS and Retest-all					
Variable: *Percentage of faults detected.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	1196.83	6	199.47	1.16	0.33
Technique	9778.57	1	9778.57	56.95	0.00
Error	20260.3	118	171.70		
Total	31235.7	125			
Variable: *Execution time.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	2.99×10^9	6	4.98×10^8	20.05	0.00
Technique	2.55×10^9	1	2.55×10^9	102.56	0.00
Error	2.93×10^9	118	2.48×10^7		
Total	8.46×10^9	125			

TABLE 4.2: Selection Anovas (Minimization RTS vs. Retest-all), for the Random Grouping of Test Cases.

Techniques: Modified non-core Entity and Retest-all					
Variable: *Percentage of faults detected.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	476.19	6	79.36	1.77	0.11
Technique	622.22	1	622.22	13.88	0.00
Error	5288.89	118	44.82		
Total	6387.3	125			
Variable: *Execution time.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	5.92×10^9	6	9.86×10^8	53.19	0.00
Technique	6.09×10^8	1	6.09×10^8	32.83	0.00
Error	2.19×10^9	118	1.85×10^7		
Total	8.72×10^9	125			

TABLE 4.3: Selection Anovas (Modified non-core Entity vs. Retest-all), for the Functional Grouping of Test Cases.

Techniques: Minimization RTS and Retest-all					
Variable: *Percentage of faults detected.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	720.64	6	120.11	0.65	0.69
Technique	14038.9	1	14038.9	76.54	0.00
Error	21644.4	118	183.43		
Total	36404.0	125			
Variable: *Execution time.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	2.98×10^9	6	4.97×10^8	19.93	0.00
Technique	2.53×10^9	1	2.53×10^9	101.37	0.00
Error	2.94×10^9	118	2.49×10^7		
Total	8.44×10^9	125			

TABLE 4.4: Selection Anovas (Minimization RTS vs. Retest-all), for the Functional Grouping of Test Cases.

lead to the rejection of the null hypothesis. As we stated in Chapter 3, we set the significance level to 0.05. So we reject the null hypothesis when p-value is less than 0.05.

For the random test grouping, the results in Table 4.1 indicate that both granularity and technique had a significant effect on fault-detection effectiveness when comparing modified non-core entity and retest-all. They also show that both granularity and technique significantly impacted test execution time. These findings agree with the results shown in the figures (Figures 4.1 and 4.2). Table 4.2 shows that, when comparing minimization and retest-all, granularity had no significant effect on fault-detection effectiveness, but significantly affected test execution time. However, technique significantly impacted both fault-detection effectiveness and test execution time. These also agree with the results shown in the figures.

For the functional test grouping, comparing modified non-core entity and retest-all, Table 4.3 shows that both granularity and technique had significant effect on the execution time. Also, technique significantly impacted fault detection effectiveness, but test suite granularity has no significant impact on it. This is different from what we observed for the random grouping, in which granularity had a significant effect on fault detection effectiveness. We therefore performed an Anova comparing the effect of the grouping techniques themselves on fault detection effectiveness for the modified non-core entity technique, but these Anova results did not show significant difference between the two techniques (see the Anova table for this analysis in Appendix B). The difference we have seen here might simply be due to some outliers in the random grouping results.

For the functional grouping, comparing minimization RTS and retest-all,

Table 4.4 shows results similar to those observed for the random grouping, with both granularity and technique having significant effects on execution time and only technique having significant impact on fault detection effectiveness.

4.2 Test Suite Reduction

Figure 4.5 shows graphs of fault detection effectiveness and test execution time for two test suite reduction techniques, with random test groupings. Figure 4.6 shows the results for the functional groupings. The pair of graphs in the first row of the figure present results for the retest-all technique (no reduction) which is our control technique, the pair of graphs in the second row present results for the GHS reduction technique. Similar to the figures for RTS, the horizontal axes represent test suite granularity, and the vertical axes represent either fault detection effectiveness (left) or test execution time (right).

For both grouping techniques, test suite reduction results were somewhat different from those for regression test selection techniques in terms of fault detection effectiveness. For the random test grouping, in the bottom-left graph in Figure 4.5, it seems like an overall upward trend in fault detection effectiveness occurs as granularity increases, but it is not consistent with a highest point at level G4 and a relatively low point at G8. For the functional test grouping, in the bottom-left graph in Figure 4.6, results display a higher trend in the middle and a lower trend at the two ends.

The test suite execution times measured were similar to those produced by minimization RTS techniques for the two grouping technqiues, as shown in the bottom-right graphs in Figures 4.5 and 4.6, in that time decreased as test suite granularity increased, but savings in time decreased. For example, the GHS re-

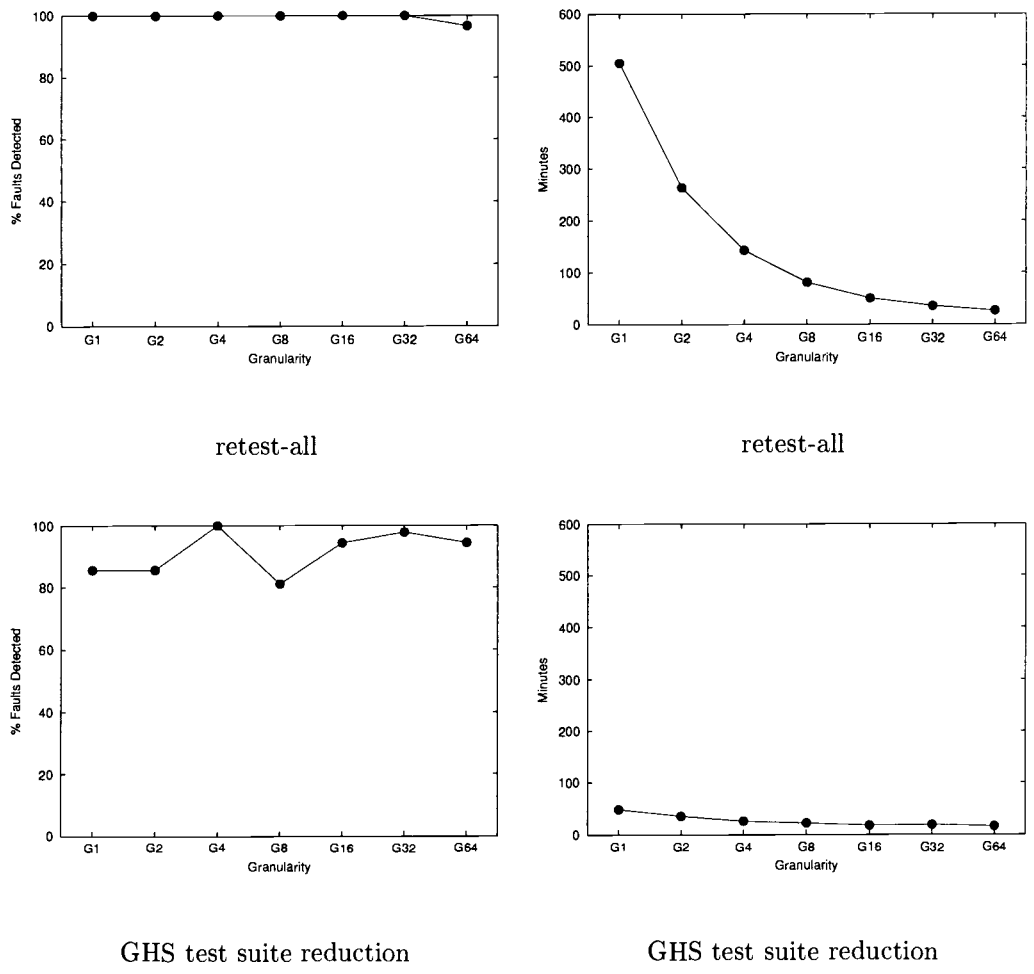


FIGURE 4.5: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, averaged across versions, for the random grouping of test cases.

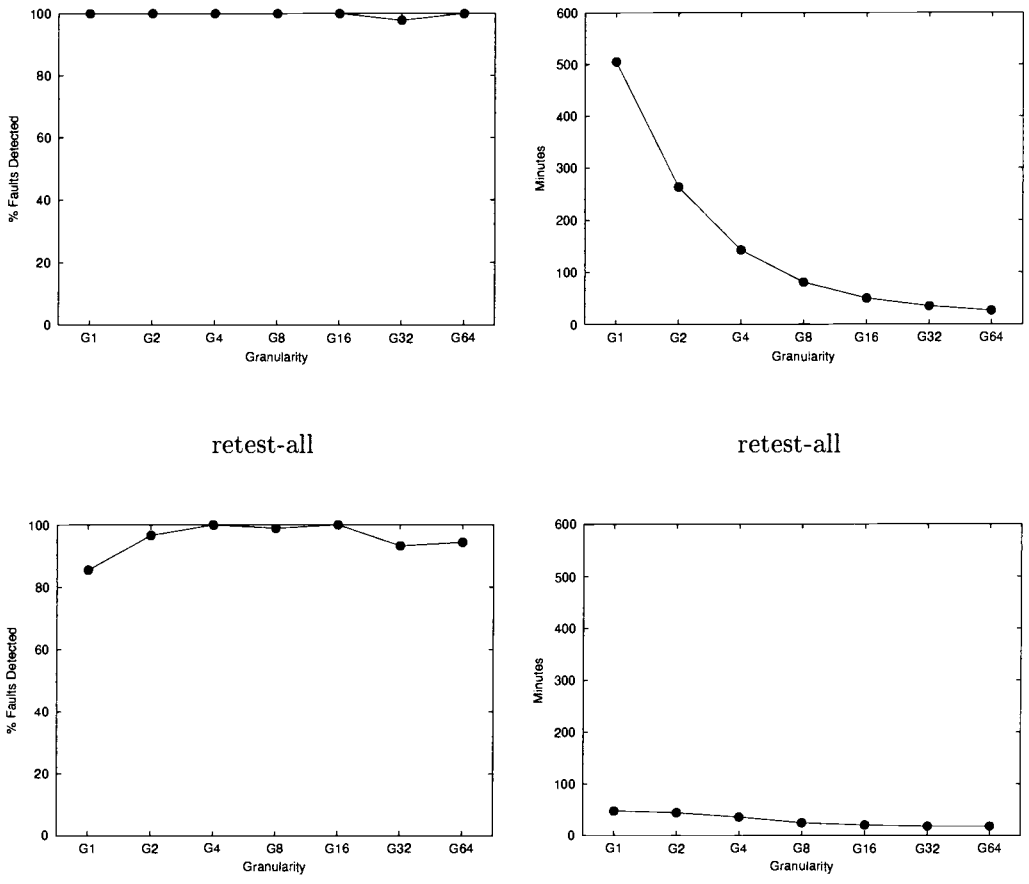


FIGURE 4.6: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, averaged across versions, for the functional grouping of test cases.

duction technique reduced execution time by 90.5% at granularity level G1, but by only 39.7% at granularity level G64, compared to the retest-all technique, for the random grouping. For the functional grouping, the GHS reduction reduced execution time by 90.5% at G1, and 30.7% at G64. It is apparent that as test suite granularity increased, the opportunities to save through reduction decreased.

An Anova was performed to further investigate the statistical impact of granularity and technique on our dependent variables for test suite reduction techniques. The results are shown in Table 4.5 for the random grouping and Table 4.6 for the functional grouping, which follow the same structure as the tables in the previous section for RTS. The Anova tables indicate that granularity and technique both have significant impact on both dependent measures for both grouping techniques.

Techniques: GHS Reduction and Retest-all					
Variable: *Percentage of faults detected.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	1330.16	6	221.69	3.71	0.00
Technique	2146.03	1	2146.03	35.90	0.00
Error	7053.97	118	59.78		
Total	10530.2	125			
Variable: *Execution time.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	3.35×10^9	6	5.59×10^8	25.52	0.00
Technique	1.96×10^9	1	1.96×10^9	89.68	0.00
Error	2.58×10^9	118	2.19×10^7		
Total	7.90×10^9	125			

TABLE 4.5: Reduction Anovas, for the Random Grouping of Test Cases.

Techniques: GHS Reduction and Retest-all					
Variable: *Percentage of faults detected.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	774.60	6	129.10	5.12	0.00
Technique	536.51	1	536.51	21.28	0.00
Error	2974.6	118	25.21		
Total	4285.71	125			
Variable: *Execution time.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	3.35×10^9	6	5.59×10^8	25.51	0.00
Technique	1.85×10^9	1	1.85×10^9	84.23	0.00
Error	2.59×10^9	118	2.19×10^7		
Total	7.79×10^9	125			

TABLE 4.6: Reduction Anovas, for the Functional Grouping of Test Cases.

We also performed an Anova to compare the effect of the grouping techniques on fault detection effectiveness and execution time for the test suite reduction. The results are shown in Table 4.7 and a detailed discussion is presented in Chapter 5.

Grouping Techniques: Random and Functional					
Variable: *Percentage of faults detected.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	2560.32	6	426.72	5.93	0.00
Grouping Technique	578.57	1	578.57	8.04	0.005
Error	8493.65	118	71.98		
Total	11632.5	125			
Variable: *Execution time.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	5.61×10^7	6	9.34×10^6	429.89	0.00
Grouping Technique	1.82×10^6	1	1.82×10^6	83.74	0.00
Error	2.56×10^6	118	2.17×10^4		
Total	6.04×10^7	125			

TABLE 4.7: Grouping Anovas for Test Suite Reduction.

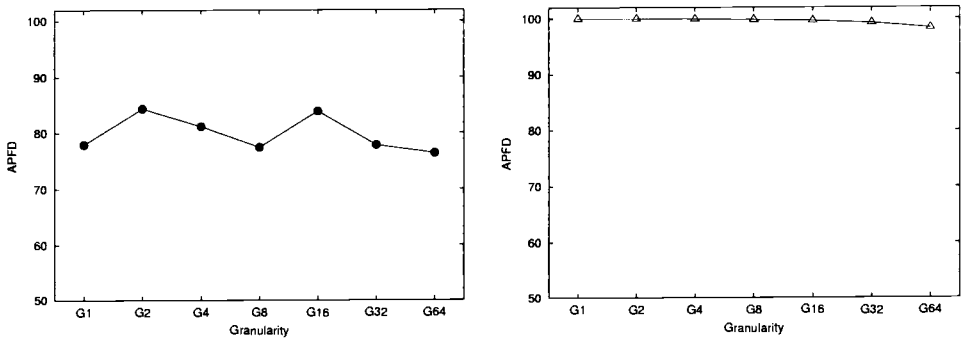
4.3 Test Case Prioritization

Our third experiment considered test case prioritization. Within this methodology we analyze six techniques. When analyzing data, as mentioned in the previous chapter, we focused on the faults revealed by less than 1% of the tests.

Figure 4.7 displays six graphs for the random test grouping, representing the results for each prioritization technique individually, and Figure 4.8 shows the results of all the prioritization techniques together for the purpose of comparison. In each graph, the horizontal axis represents test suite granularity, and the vertical axis represents our measure of rate of fault detection, APFD. Figures 4.9 and 4.10 show the results for the functional test grouping and follow the same structure as the graph for the random grouping.

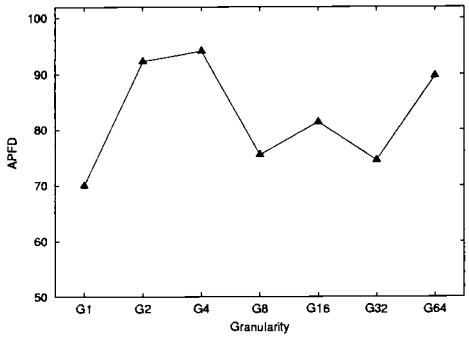
The optimal technique, as expected, achieved the highest rate of fault detection among other techniques for both grouping techniques. Also there was a constant, slight decrease in APFD values for the optimal technique as test suite granularity increased. This is what we expected, since having more test cases provides more opportunities for prioritization; still, the differences were small.

The other prioritization techniques, however, presented great variation in APFD values that cannot be explained based solely on the increase in granularity. The acov-func technique was the second best prioritization technique overall, having higher APFD values than other techniques on all but 3 of the 18 versions, for both the random and functional grouping. For the random grouping, the tcov-func, tcov-bdiff and acov-bdiff techniques achieved fairly similar APFD values to each other across granularities. For the functional grouping, however, the tcov-func, tcov-bdiff and acov-bdiff techniques varied more between one another. For both groupings, random prioritization had lower APFD

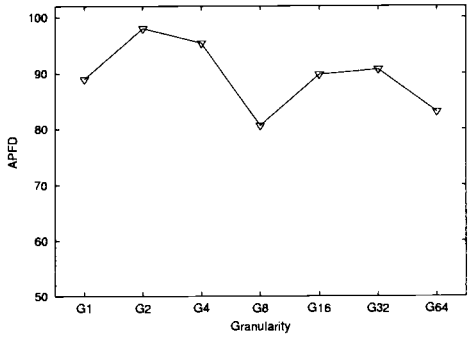


(a) Random (retest-all)

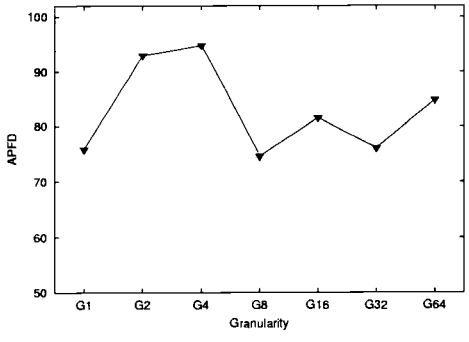
(b) Optimal



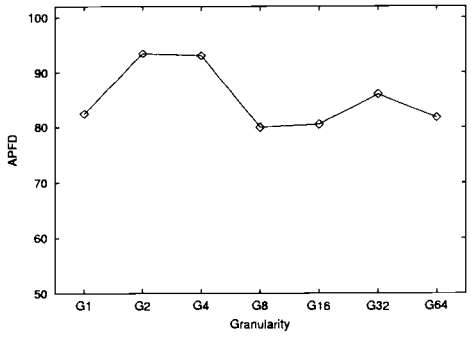
(c) tcov-func



(d) acov-func

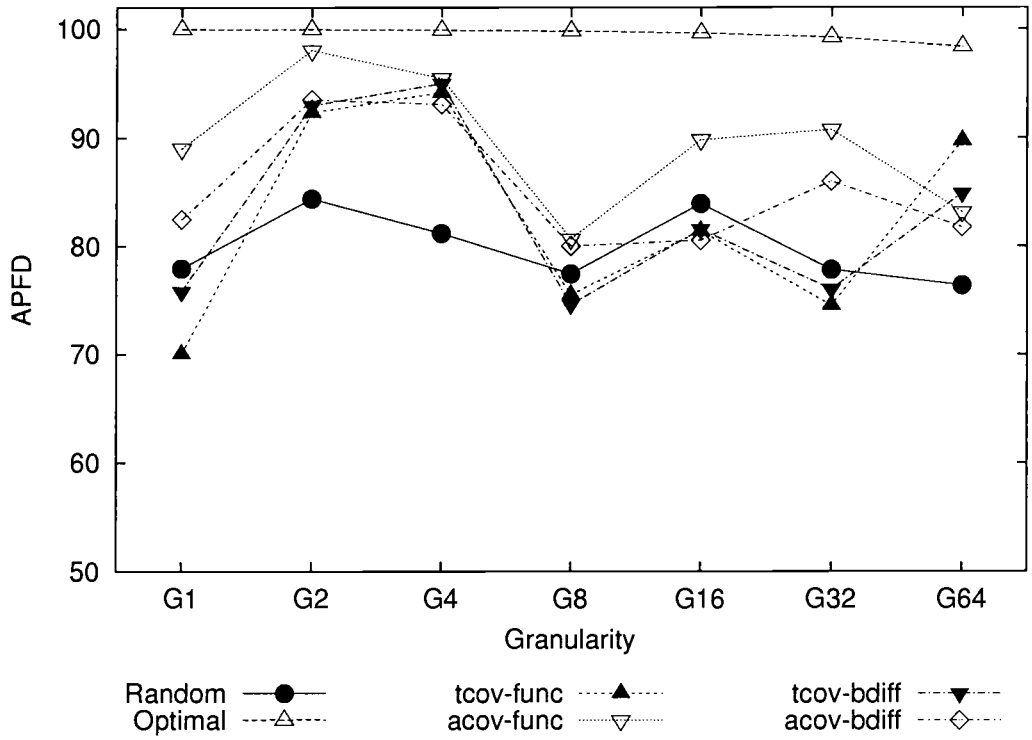


(e) tcov-bdiff



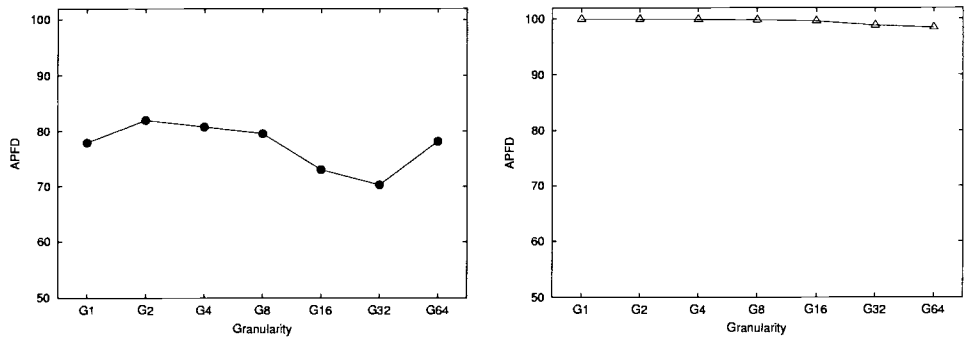
(f) acov-bdiff

FIGURE 4.7: APFD for different test case prioritization techniques across test suite granularities, averaged across versions, for the random grouping of test cases.



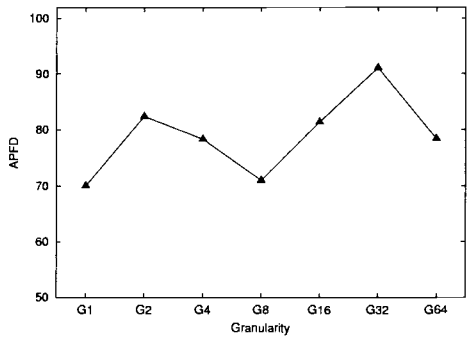
(a) All techniques together

FIGURE 4.8: APFD for all prioritization techniques together, for the random grouping of test cases.

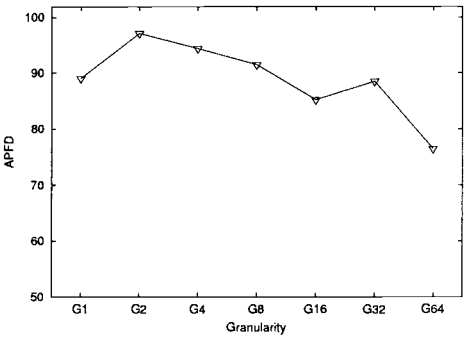


(a) Random (retest-all)

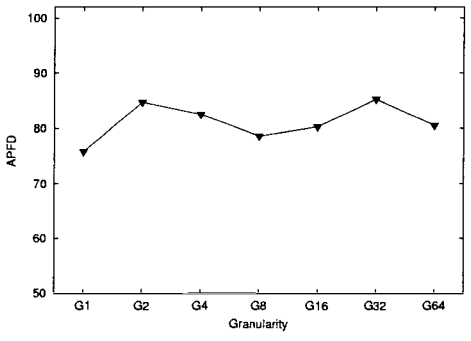
(b) Optimal



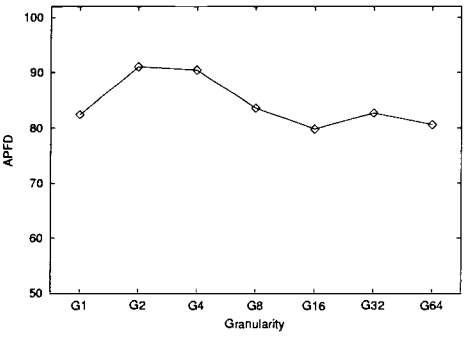
(c) tcov-func



(d) acov-func

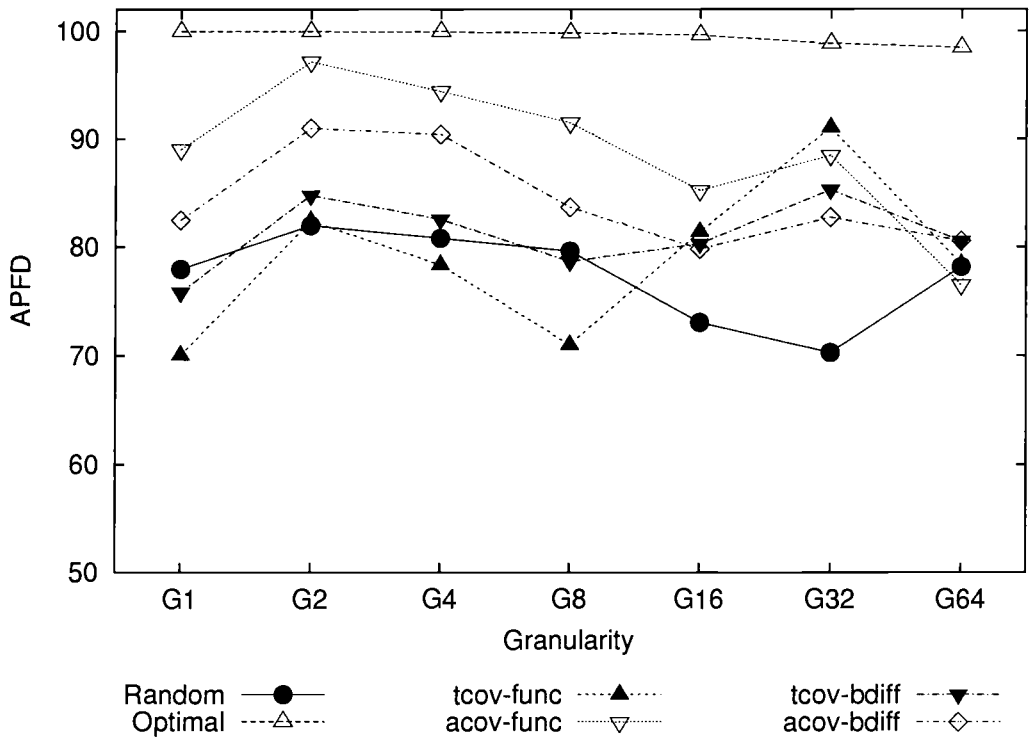


(e) tcov-bdiff



(f) acov-bdiff

FIGURE 4.9: APFD for different test case prioritization techniques across test suite granularities, averaged across versions, for the functional grouping of test cases.



(a) All techniques together

FIGURE 4.10: APFD for all prioritization techniques together, for the functional grouping of test cases.

values than these techniques under most (but not all) circumstances.

In order to further investigate the statistical impact of the test groupings on the rate of fault detection for prioritization techniques, an Anova was performed. The results show that the test groupings did not have significant impact on the APFD values for all six prioritization techniques examined in this study. (See Appendix B for the Anova tables.)

An Anova was also performed between prioritization techniques within each type of the test grouping. Tables 4.8 to 4.12 present the Anova tables comparing different prioritization techniques against random prioritization for the random test grouping. Tables 4.13 to 4.17 present the Anova tables for the functional test grouping. They follow the same structures as the tables for regression test selection and reduction, except with only one dependent measure, APFD.

Across all analyses, granularity was shown to have a significant effect in all but 4 cases (optimal vs random and acov-bdiff vs random for the random grouping, tcov-func vs random and tcov-bdiff vs random for the functional grouping). Similarly, across all analyses, technique was shown to have a significant effect in all but 3 cases (tcov-func vs random and tcov-bdiff vs random for the random grouping, tcov-func vs random for the functional grouping).

Techniques: Optimal and Random					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	352.77	6	58.80	1.60	0.15
Technique	12181.5	1	12181.5	332.21	0.00
Error	4326.79	118	36.67		
Total	16861.1	125			

TABLE 4.8: Prioritization Anovas (Optimal vs. Random), for the Random Grouping of Test Cases.

Techniques: tcov-func and Random					
Variable: *APFD*.					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	3560.95	6	593.49	4.75	0.00
Technique	222.00	1	222.00	1.78	0.18
Error	14755.1	118	125.04		
Total	18538.0	125			

TABLE 4.9: Prioritization Anovas (tcov-func vs. Random), for the Random Grouping of Test Cases.

Techniques: acov-func and Random					
Variable: *APFD*.					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	2126.98	6	354.50	4.39	0.00
Technique	2949.52	1	2949.52	36.55	0.00
Error	9523.02	118	80.70		
Total	14599.5	125			

TABLE 4.10: Prioritization Anovas (acov-func vs. Random), for the Random Grouping of Test Cases.

Techniques: tcov-bdiff and Random					
Variable: *APFD*.					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	3038.21	6	506.37	4.24	0.00
Technique	302.86	1	302.86	2.53	0.11
Error	14098.1	118	119.48		
Total	17439.2	125			

TABLE 4.11: Prioritization Anovas (tcov-bdiff vs. Random), for the Random Grouping of Test Cases.

Techniques: acov-bdiff and Random					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	1700.47	6	283.41	2.14	0.054
Technique	937.66	1	937.66	7.07	0.01
Error	15646.9	118	132.60		
Total	18285.1	125			

TABLE 4.12: Prioritization Anovas (acov-bdiff vs. Random), for the Random Grouping of Test Cases.

Techniques: Optimal and Random					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	566.84	6	94.47	2.54	0.02
Technique	15374.0	1	15374.0	413.48	0.00
Error	4387.51	118	37.18		
Total	20328.3	125			

TABLE 4.13: Prioritization Anovas (Optimal vs. Random), for the Functional Grouping of Test Cases.

Techniques: tcov-func and Random					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	919.14	6	153.19	1.42	0.21
Technique	74.26	1	74.26	0.69	0.41
Error	12687.3	118	107.52		
Total	13680.7	125			

TABLE 4.14: Prioritization Anovas (tcov-func vs. Random), for the Functional Grouping of Test Cases.

Techniques: acov-func and Random					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	2365.68	6	394.28	6.11	0.00
Technique	4162.44	1	4162.44	64.47	0.00
Error	7619.02	118	64.57		
Total	14147.1	125			

TABLE 4.15: Prioritization Anovas (acov-func vs. Random), for the Functional Grouping of Test Cases.

Techniques: tcov-bdiff and Random					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	673.40	6	112.23	1.12	0.35
Technique	435.92	1	435.92	4.35	0.04
Error	11821.5	118	100.18		
Total	12930.8	125			

TABLE 4.16: Prioritization Anovas (tcov-bdiff vs. Random), for the Functional Grouping of Test Cases.

Techniques: acov-bdiff and Random					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	1735.56	6	289.26	2.80	0.01
Technique	1530.29	1	1530.29	14.82	0.00
Error	12182.5	118	103.24		
Total	15448.3	125			

TABLE 4.17: Prioritization Anovas (acov-bdiff vs. Random), for the Functional Grouping of Test Cases.

CHAPTER 5

DISCUSSION

As stated in Chapter 3, the research question investigated in this thesis is, "how does test suite granularity affect the costs-benefits tradeoffs for different regression testing techniques and test grouping techniques?" In more detail, we expressed this question as the following null hypotheses:

H1 (test suite granularity): Test suite granularity does not have a significant impact on the costs-benefits tradeoffs for different regression testing techniques.

H2 (regression testing technique): Different regression testing techniques do not significantly affect the selected costs-benefits measures.

H3 (grouping technique): Different test grouping techniques do not significantly affect the costs and benefits of regression testing methodologies.

Our results let us reject our first hypothesis (supporting the alternative hypothesis): test suite granularity significantly impacts the cost-effectiveness of test suite reduction and some regression test selection and test case prioritization techniques. Also we reject our second hypothesis (supporting the alternative hypothesis): different regression testing techniques significantly affect the costs-effectiveness of test selection, reduction, and some prioritization techniques. However, our results primarily supported H3: the two different grouping techniques did not cause significant differences in our selected cost-benefits measures for RTS and prioritization technqiues, showing significant effects only for test suite reduction. We discuss this in detail in Section 5.3.

These following paragraphs address some of the implications of the results for tradeoffs and factors involved in designing test suites, choosing granularity, and grouping test cases. From a practitioner's point of view, these implications might be important. We also help clarify some practical impacts of the results, taking into account the threats to validity discussed in Section 3.5.

5.1 Reducing the Test Suite Versus Reducing Overhead.

Coarser granularity can greatly reduce execution time, and hence increase the efficiency of a test suite. For example, increasing granularity from G1 to G64 yielded an average savings in test suite execution time of 239 minutes, i.e. a 92% savings. Finer granularity, however, is clearly more supportive of regression test selection and test suite reduction, since the effectiveness of these techniques diminishes as granularity increases. For example, when the modified non-core entity RTS technique was applied to the G1 suite, the suite's execution time was reduced from 505 to 228 minutes (55% time reduction). When the same technique was applied to the G64 suite, the savings were less than 1%. Hence, finer granularity provides greater flexibility through larger numbers of small test cases that can be successfully manipulated by RTS and test suite reduction techniques to reduce the number of test cases to be executed.

Even when RTS and test suite reduction methodologies can save significant execution time at finer granularity, however, increasing test suite granularity by joining small test cases might be still preferable with regard to execution time. Test suites with larger granularity have fewer test cases, which reduces the overall overhead of the suite; this effect was very profound for our subject `emp-server`.

In our experiments, the savings generated by increases in granularity resulted primarily from reduction in the overhead associated with test setup and cleanup. In other cases, the overhead can also include the cost of human intervention. The amount of overhead in test suite execution required for different subject programs might, however, be different. Test suites with low overhead are not likely to yield substantial time savings through increases in granularity. For such suites, potential savings through RTS or reduction may become the dominant factor in choosing granularity.

5.2 Fault Detection Effectiveness Versus Time Savings

Even though execution time was greatly reduced as test suite granularity increased, the fault detection effectiveness for RTS and test suite reduction did not vary a lot over different granularities. In most cases, there was no clear upward trend in fault detection effectiveness as granularity increased and sometimes fault detection effectiveness decreased at higher granularities.

The safe RTS technique (modified entity) did not lose any fault detection effectiveness compared to the retest-all technique; however, it also could not achieve any savings in execution time. The non-core modified entity RTS technique did miss some faults, but it achieved substantial savings in test execution time, especially at lower granularities. The minimization RTS technique, and the test suite reduction technique saved even more, but with the cost of losing more faults. So there exists a tradeoff between time savings and fault detection effectiveness. Practitioners need to be aware of this when deciding which regression testing techniques to use.

5.3 The Effects of Test Grouping Technique on Test Suite Reduction

As we mentioned at the beginning of this chapter, our two test grouping techniques acted significantly differently with respect to fault detection effectiveness for test suite reduction (see Table 4.7 for the ANOVA results). The functional grouping obtained better fault detection effectiveness than the random grouping at granularity levels G1 through G16, but was worse at G32 and G64. The percentage of faults detected for the functional grouping (the bottom-left graph in Figure 4.6) first increased as granularity increased, then reached its highest points in the middle range of granularities, and finally dropped again at the G32 and G64 levels.

We believe that reduction in fault detection effectiveness for the functional grouping at levels G32 and G64 is due to what might be called "impurities" in those groupings. When generating functional test suites, since some *empire* functions had only a few (1, 2 or 3) test grains and some had more (up to 30), but few had over 30 grains, the G32 and G64 functional suites include many tests that are randomly grouped, and not related to a specific functionality. The average impurity (the percentage of randomly grouped tests over the total number of tests) is 27% for G2 through G16 test suites, however, impurity is 88% for the G32 suite and 100% for the G64 suite. This shows a limiting aspect of large tests. In practice, we could omit G32 and G64 test suites from consideration for the functional grouping.

5.4 Prioritization

Among the six prioritization techniques, the optimal technique, as expected, achieved the highest rate of fault detection for both grouping techniques. (Actu-

ally, however, since the optimal prioritization simply considers tests according to their ability to reveal faults, its results are independent of grouping techniques.) Also, as test suite granularity increased, there was a constant, slight decrease in APFD values for the optimal technique. This is what we expected, since having more test cases provides more opportunities for prioritization; still, the differences were small.

Other prioritization techniques, however, presented great variations in APFD values and displayed no clear trend based on the increase in granularity. This may suggest that we need more observations from different test suites. Currently, we only have one test suite at each granularity level for each grouping technique. (There is always only one G1 test suite for both random and functional groupings.) At each granularity level, we used twenty different runs with each run using a different randomized order of that test suite, and computed the average over these runs. By doing so, we reduced the possible bias in our results due to a particular randomized order. However, our results at each level $k > 1$ are still solely based on particular Gk test suites and may vary across other Gk test suites. There are thus possibilities that a particular test suite could bias the results. If we could generate additional test suites at each G level (from G2 to G64), collect results on these test suites individually and then average them, we might find a clearer trend in APFD values over granularities. Unfortunately, approximately 730 hours of machine time is needed to gather a single complete set of fault detection effectiveness data across all the granularities for one version. Due to limitations in time, it was not practical to generate any other test suites and collect results on them for this thesis.

5.5 Unresolved Issues and Opportunities.

Several questions remain unanswered and new questions have been raised as a result of these experiments. First, we must be sensitive to the existence of metrics that capture other meaningful attributes impacted by test suite granularity. For example, test suites with finer granularity might facilitate fault localization. Our metrics do not reflect all possible impacts.

Second, random and functional grouping techniques do not show much meaningful variance on RTS and prioritization techniques. This may suggest that the results for RTS and prioritization techniques are independent of grouping techniques. But given that only two groupings were experimented with in this thesis, we cannot generalize the results to other grouping techniques. More groupings could be examined to verify and further the findings in this study.

Third, we cannot fully explain our prioritization results and we realize that there are factors affecting variation in rate of fault detection that we are not capturing. Creating more test suites at each granularity level could be one way to capture the affecting factors.

CHAPTER 6

CONCLUSION

Writers of testing textbooks have long suggested that test suite granularity can affect the cost-effectiveness of testing. Several test suite design factors, such as test suite size and adequacy criteria, have been empirically studied, but few have been studied with respect to evolving software. Several regression testing methodologies have been empirically studied, but few with respect to issues in test suite design. This study brings the empirical study of test suite design and regression testing methodologies together, focusing on a particular design factor: test suite granularity. This study also investigates the effects of two different test grouping techniques, random and functional, on regression testing methodologies. Our results show that the cost-benefits tradeoffs for test suite reduction, and for some regression test selection and test case prioritization techniques, can be significantly impacted by granularity. The two different test grouping techniques did not show much difference, however, on our selected cost-benefits measures for RTS and prioritization techniques, though they did show significant impact on test suite reduction.

Given these results, there are several things that need to be done next to continue this family of experiments. We could obtain and create additional subject infrastructure to compare with what we have now. We could generate more test suites at each granularity level, compute results on them and use their average results. These steps may lead to more meaningful discoveries. We

could also experiment with a wider range of regression testing techniques and grouping techniques. By doing so, we may be better able to provide guidelines for practitioners to use in designing test suites that can be used more efficiently and effectively across the entire lifecycle of evolving systems.

BIBLIOGRAPHY

[1] J. Bach. Useful features of a test automation system (part iii). *Testing Techniques Newsletter*, October 1996.

[2] B. Beizer. *Black-Box Testing.* John Wiley and Sons, New York, NY, 1995.

[3] R. Binder. *Testing Object-Oriented Systems.* Addison Wesley, Reading, MA, 2000.

[4] T.Y. Chen and M.F. Lau. Dividing strategies for the optimization of a test suite. *Info. Proc. Let.*, 60(3):135–141, March 1996.

[5] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. TestTube: A system for selective regression testing. In *Proc. 16th Int'l. Conf. Softw. Eng.*, pages 211–220, May 1994.

[6] B. Davia. The empire infrastructure for testing experimentation. Master's thesis, Oregon State University, apr 2001.

[7] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proc. Int'l. Symp. Softw. Testing and Analysis*, pages 102–112, August 2000.

[8] T.L. Graves, M.J. Harrold, J-M Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proc. 20th Int'l. Conf. Softw. Eng.*, pages 188–197, April 1998.

[9] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. and Meth.*, 2(3):270–285, July 1993.

[10] R. Hildebrandt and A. Zeller. Minimizing failure-inducing input. In *Proc. Int'l. Symp. Softw. Testing and Analysis*, pages 135–145, August 2000.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. Int'l. Conf. on Softw. Eng.*, pages 191–200, May 1994.

[12] C. Kaner, J. Falk, and H. Q. Nguyeen. *Testing Computer Software.* Wiley and Sons, New York, 1999.

[13] J-M Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proc. 22nd Int'l. Conf. Softw. Eng.*, pages 126–135, June 2000.

[14] E. Kit. *Software Testing in the Real World.* Addison-Wesley, Reading, MA, 1995.

[15] H.K.N. Leung and L. White. Insights into regression testing. In *Proc. Conf. Softw. Maint.*, pages 60–69, October 1989.

[16] H.K.N. Leung and L.J. White. A study of integration testing and software regression at the integration level. In *Proc. Conf. Softw. Maint.*, pages 290–300, November 1990.

[17] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proc. Twelfth Int'l. Conf. Testing Computer Softw.*, pages 111–123, June 1995.

[18] K. Onoma, W-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Comm. ACM*, 41(5):81–86, May 1988.

[19] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Comm. ACM*, 31(6), June 1988.

[20] D. Rosenblum and G. Rothermel. A comparative study of regression test selection techniques. In *Proc. of the 2nd Int'l. Workshop on Empir. Studies of Softw. Maint.*, October 1997.

[21] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. Technical Report 01-60-11, Oregon State University, September 2001.

[22] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Trans. Softw. Eng.*, 24(6):401–419, June 1998.

[23] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, August 1996.

[24] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Meth.*, 6(2):173–210, April 1997.

[25] G. Rothermel, M.J. Harrold, and J. Dedhia. Regression test selection for C++ programs. *J. Softw. Testing, Verif., Rel.*, 10(2), June 2000.

[26] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proc. Int'l. Conf. Softw. Maint.*, pages 34–43, November 1998.

[27] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold. Test case prioritization. *IEEE Trans. Softw. Eng.*, October 2001.

[28] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proc. Int'l. Conf. Softw. Maint.*, pages 44–53, November 1998.

[29] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering, An Introduction.* Kluwer Academic Publishers, Norwell, MA, 2000.

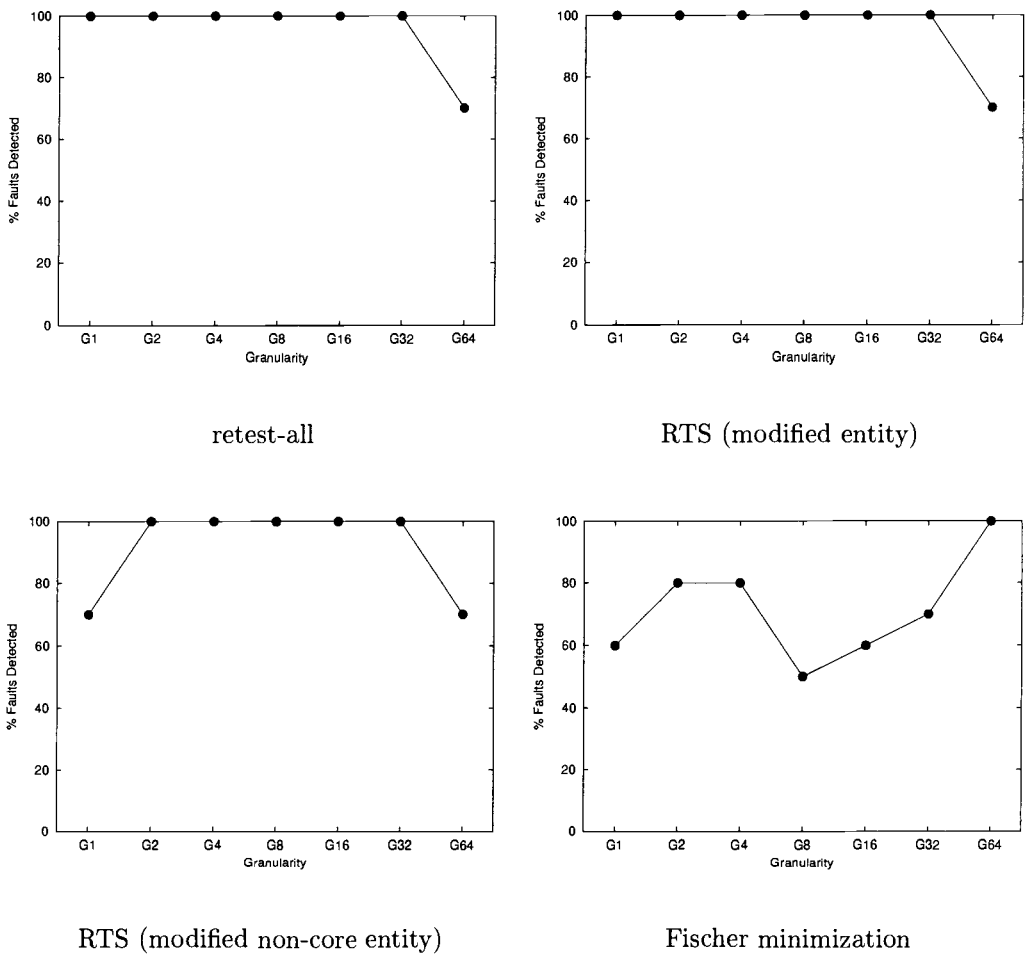
[30] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *17th International Conference on Software Engineering*, pages 41–50, April 1995.

[31] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *Softw. Pract. and Exp.*, 28(4):347–369, April 1998.

[32] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proc. Eighth Intl. Symp. Softw. Rel. Engr.*, pages 230–238, November 1997.

APPENDICES

Appendix A: Complete Data Set for Each Version



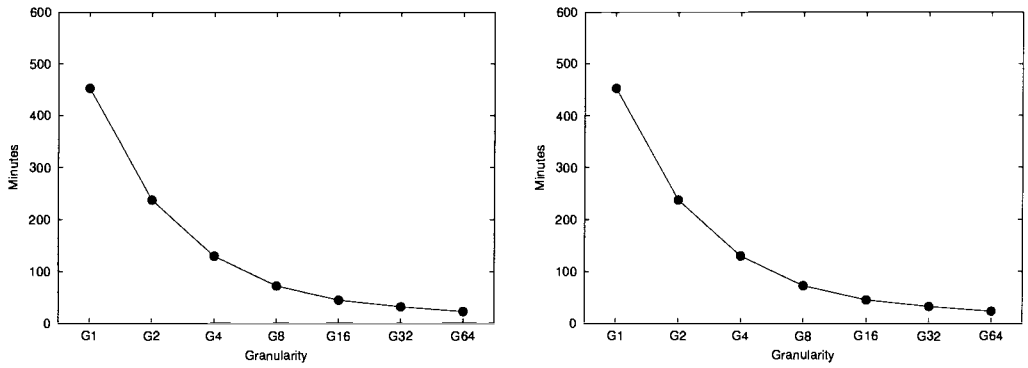
retest-all

RTS (modified entity)

RTS (modified non-core entity)

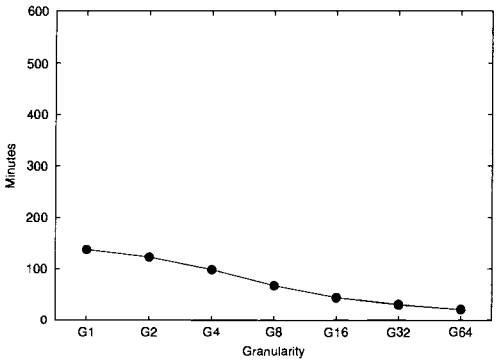
Fischer minimization

FIGURE A.1: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 1, for the random grouping of test cases.

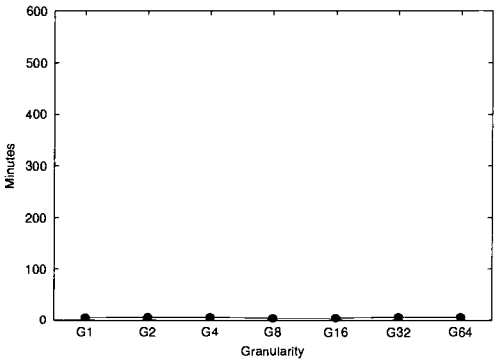


retest-all

RTS (modified entity)



RTS (modified non-core entity)



Fischer minimization

FIGURE A.2: Test execution time for regression test selection techniques across test suite granularities, for version 1, for the random grouping of test cases.

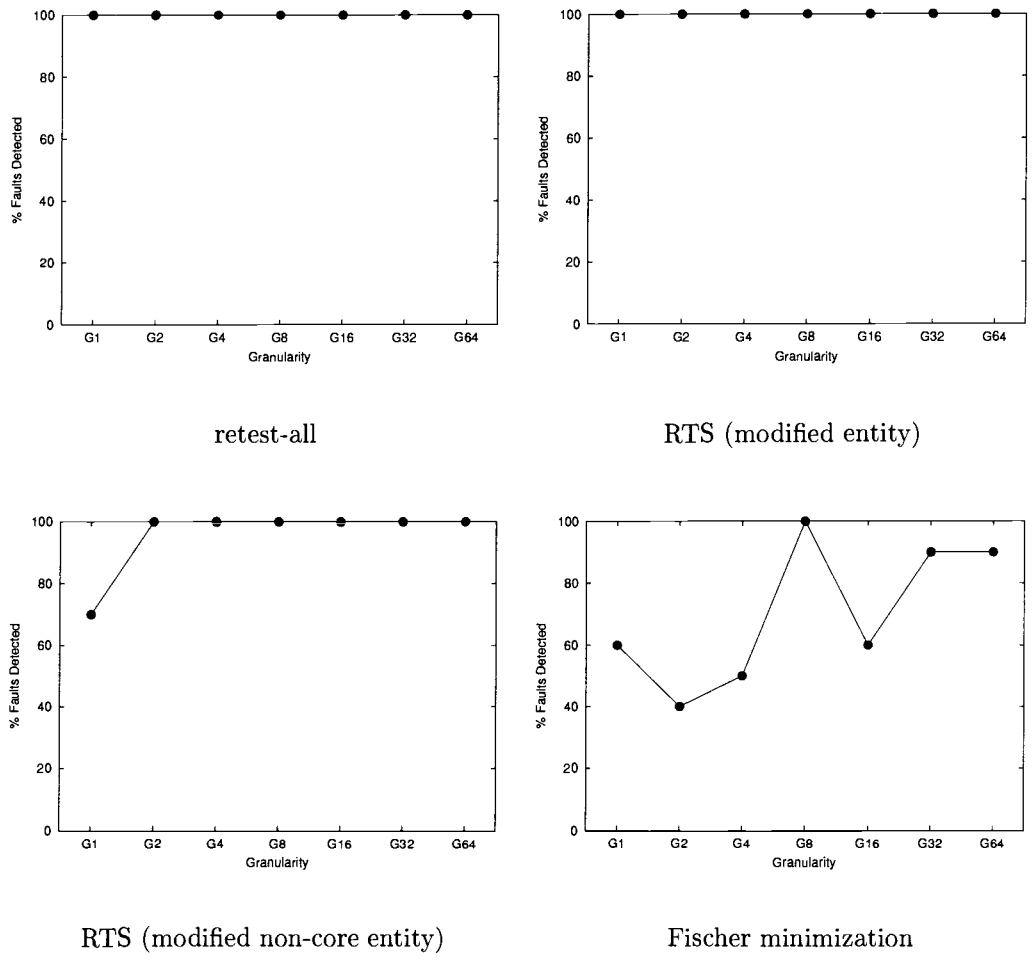
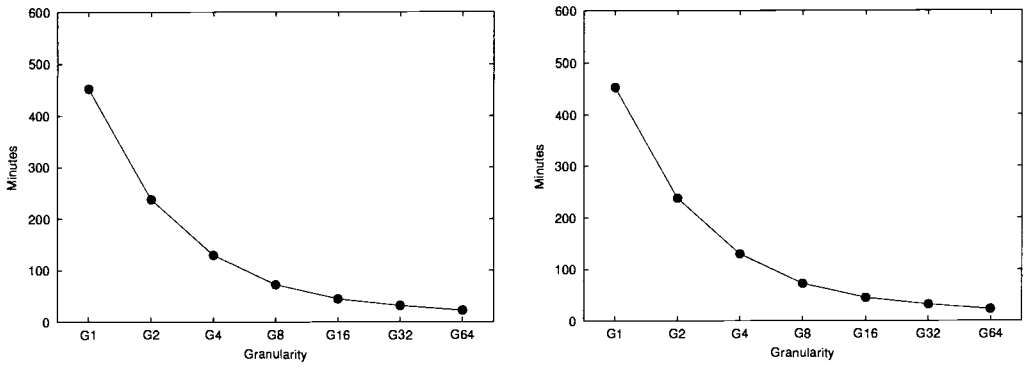
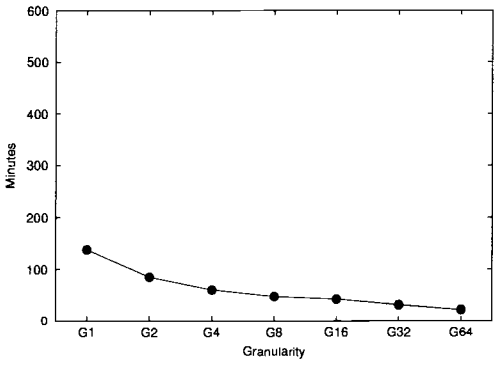


FIGURE A.3: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 1, for the functional grouping of test cases.

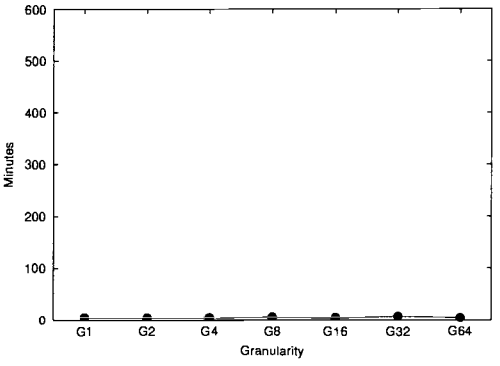


retest-all

RTS (modified entity)

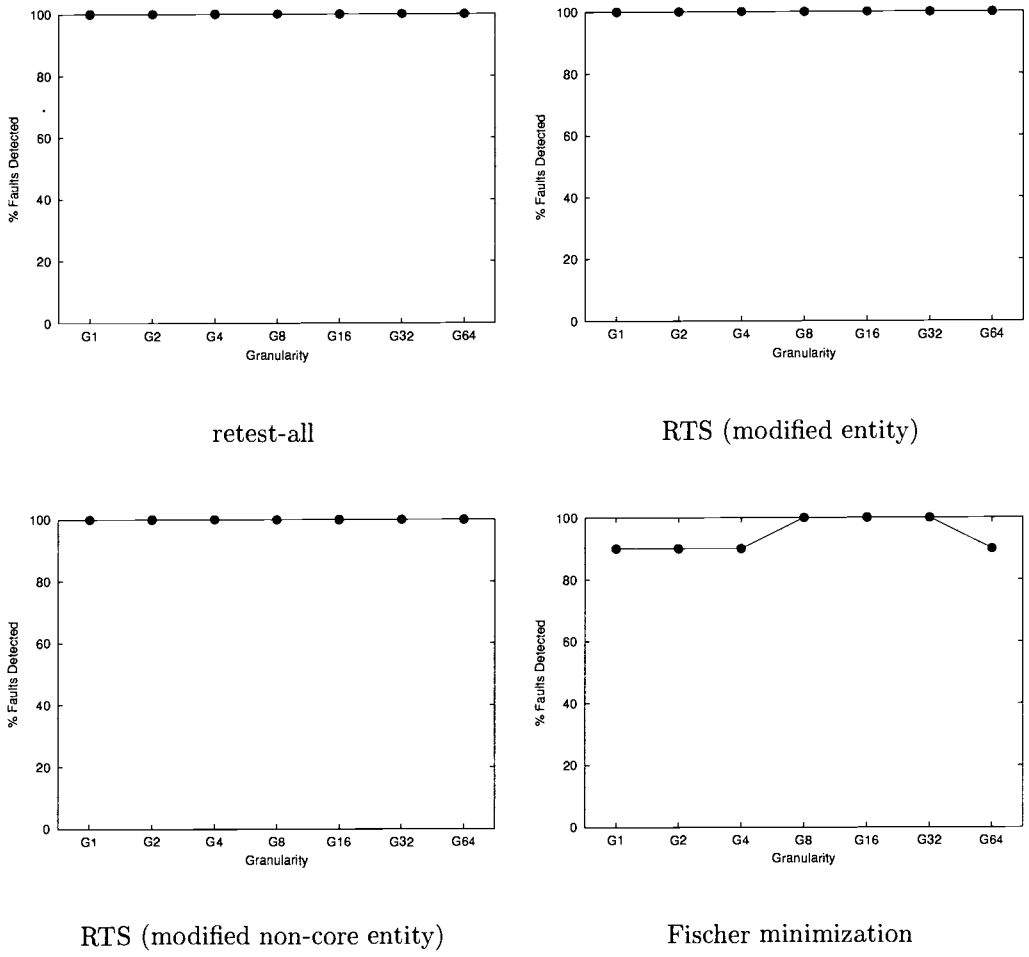


RTS (modified non-core entity)



Fischer minimization

FIGURE A.4: Test execution time for regression test selection techniques across test suite granularities, for version 1, for the functional grouping of test cases.



retest-all

RTS (modified entity)

RTS (modified non-core entity)

Fischer minimization

FIGURE A.5: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 2, for the random grouping of test cases.

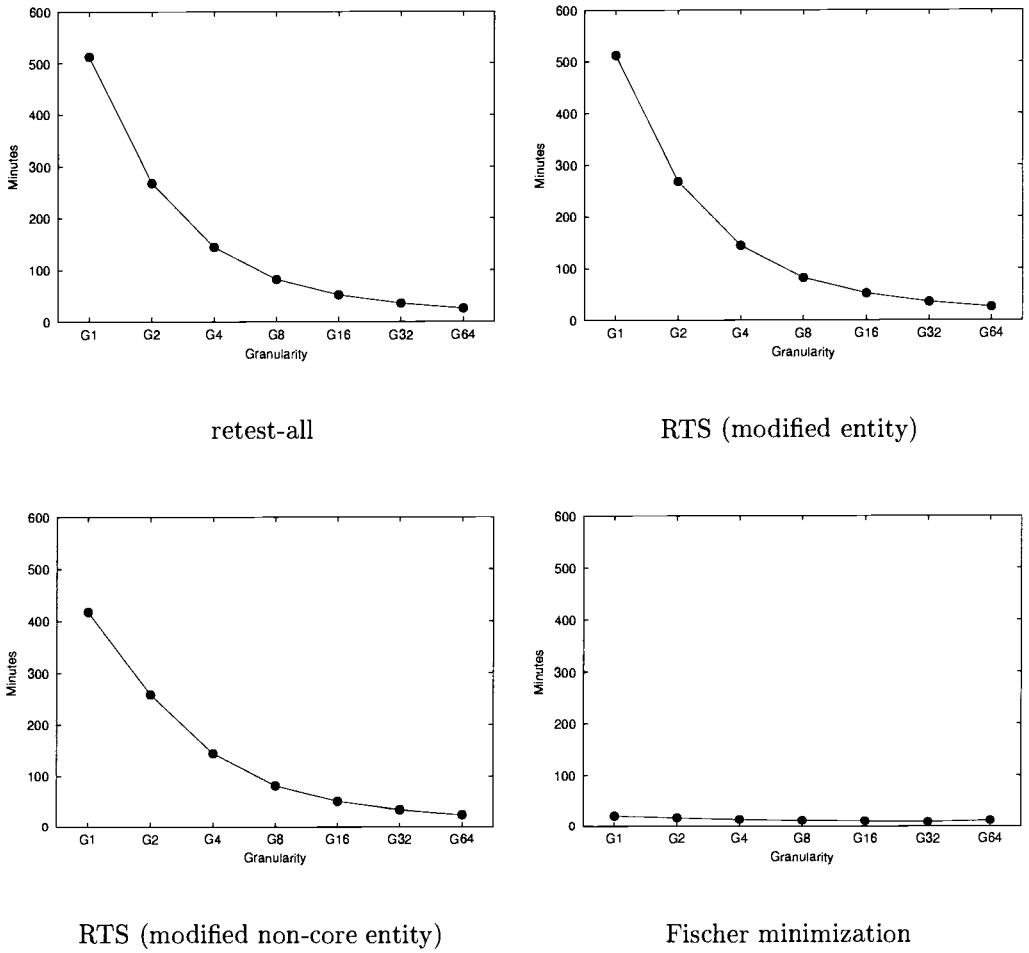


FIGURE A.6: Test execution time for regression test selection techniques across test suite granularities, for version 2, for the random grouping of test cases.

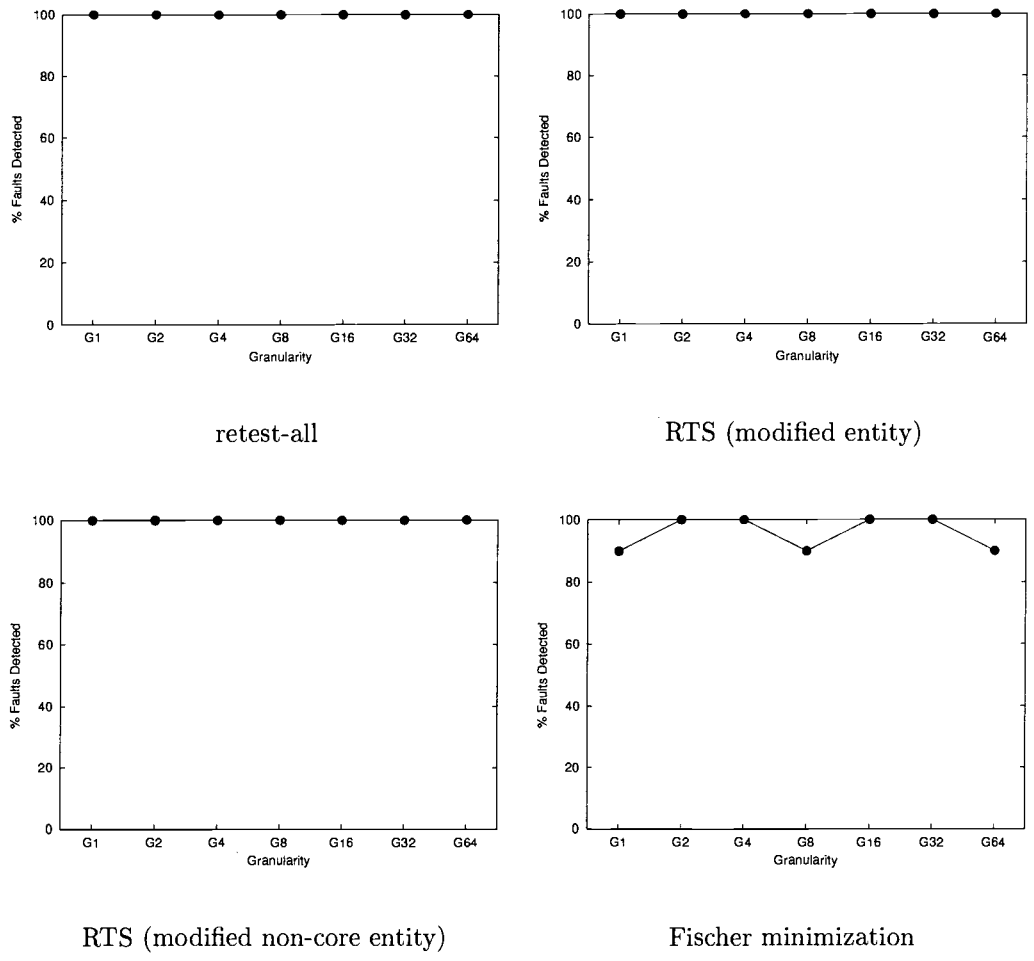


FIGURE A.7: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 2, for the functional grouping of test cases.

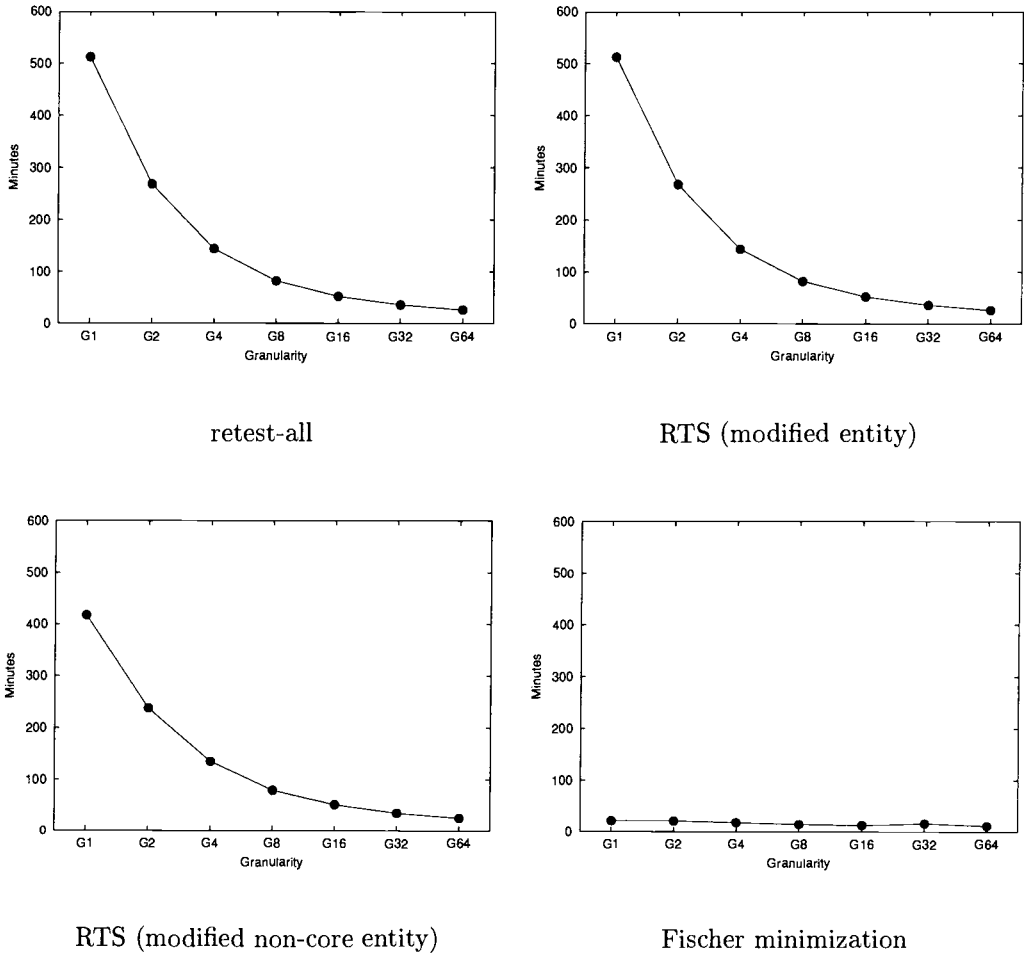


FIGURE A.8: Test execution time for regression test selection techniques across test suite granularities, for version 2, for the functional grouping of test cases.

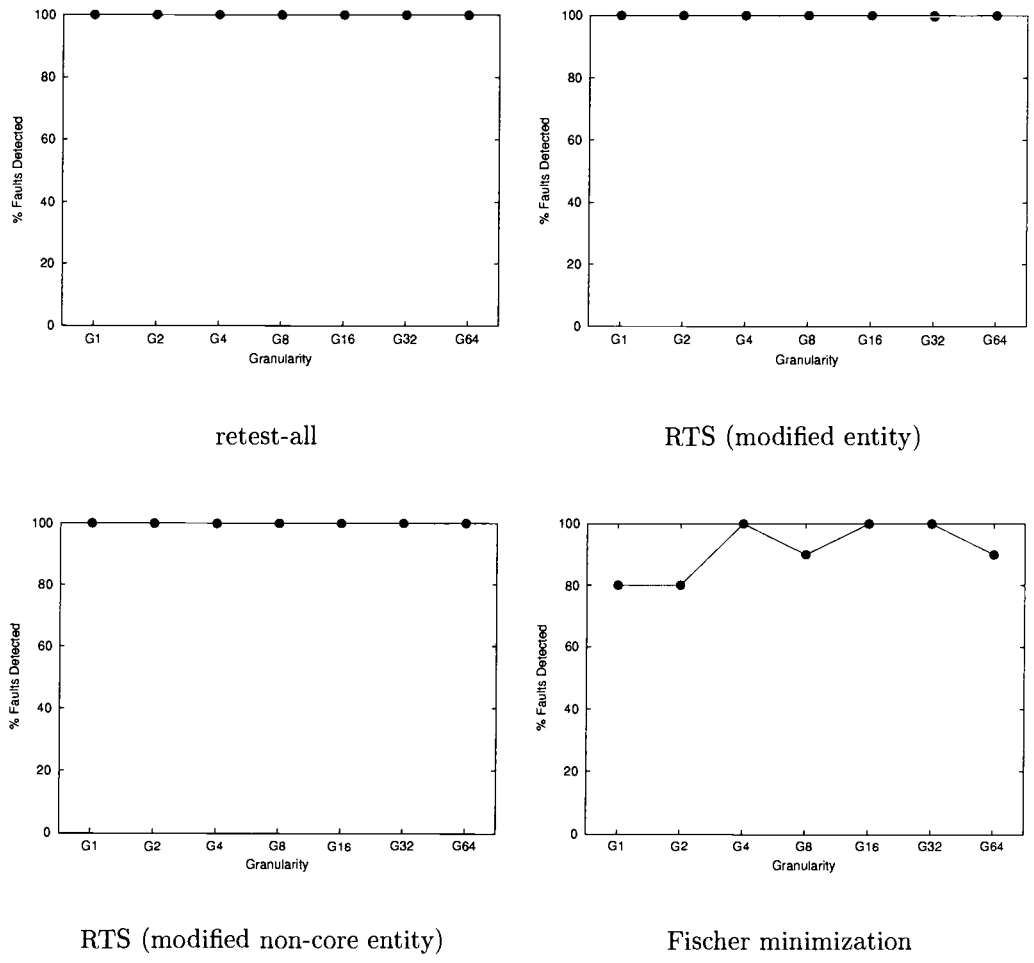


FIGURE A.9: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 3, for the random grouping of test cases.

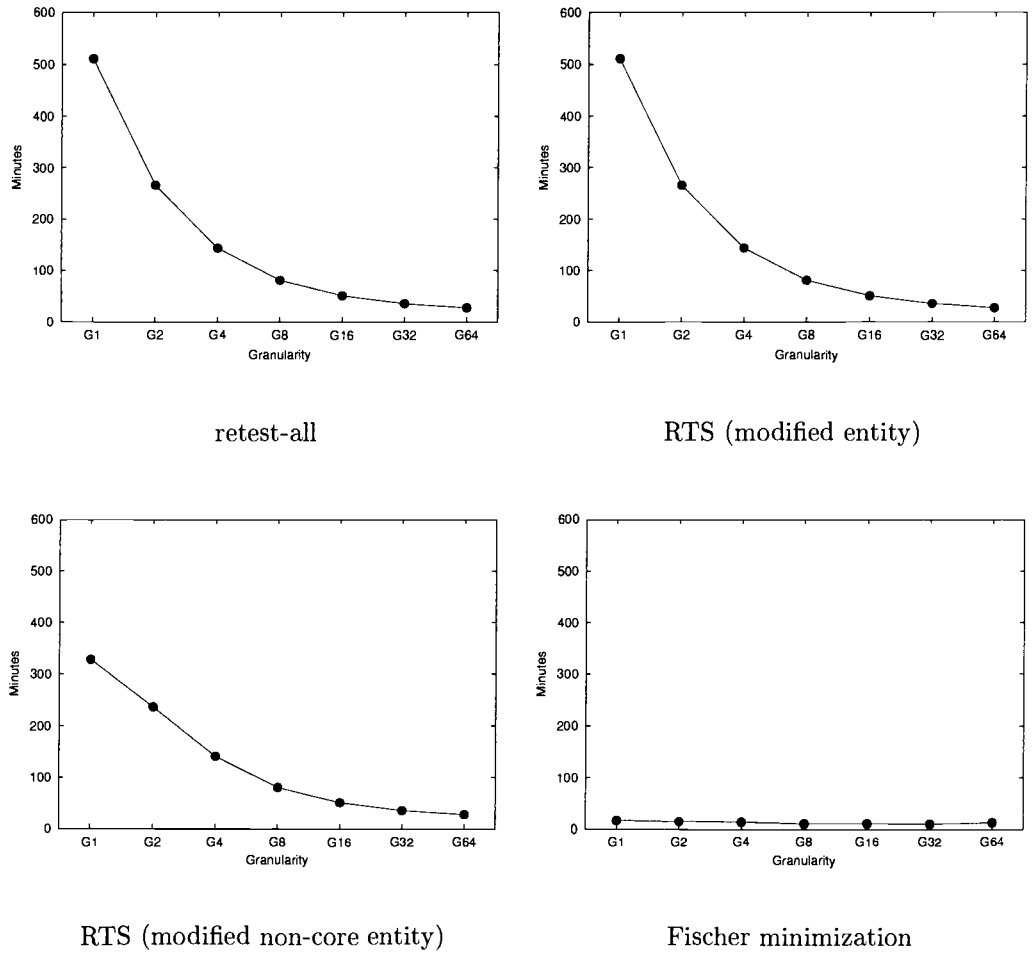
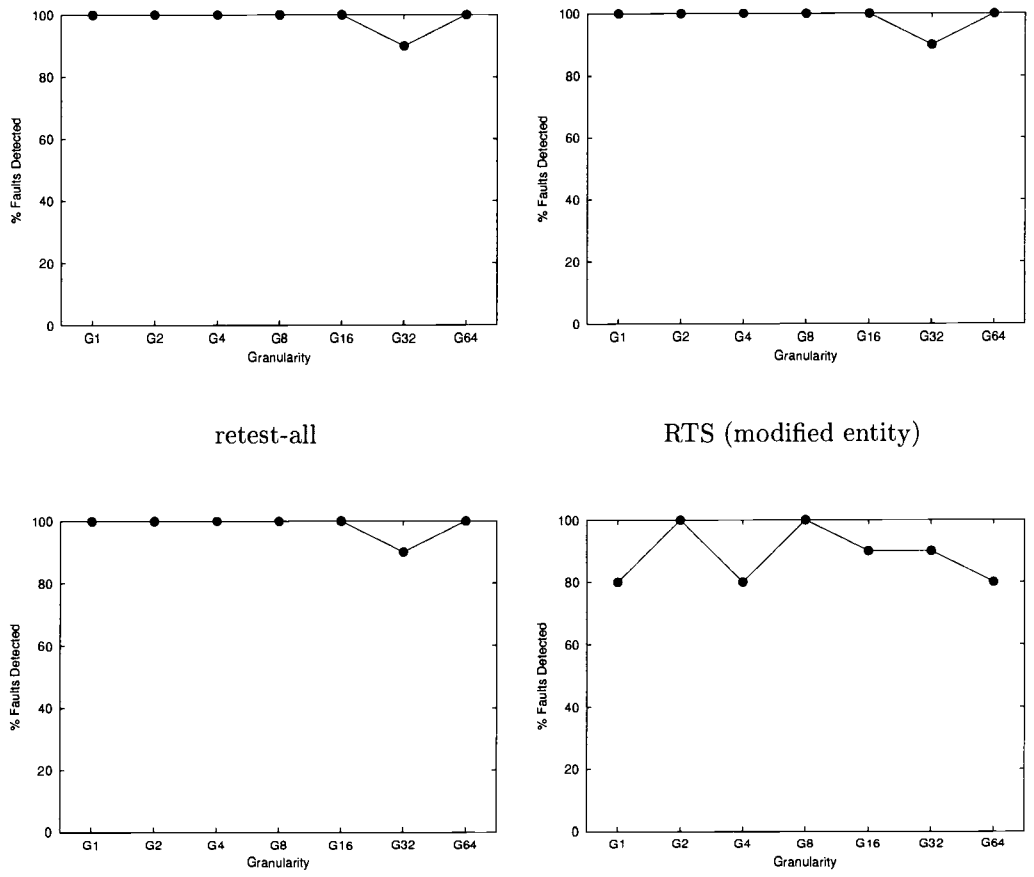


FIGURE A.10: Test execution time for regression test selection techniques across test suite granularities, for version 3, for the random grouping of test cases.



retest-all

RTS (modified entity)

RTS (modified non-core entity)

Fischer minimization

FIGURE A.11: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 3, for the functional grouping of test cases.

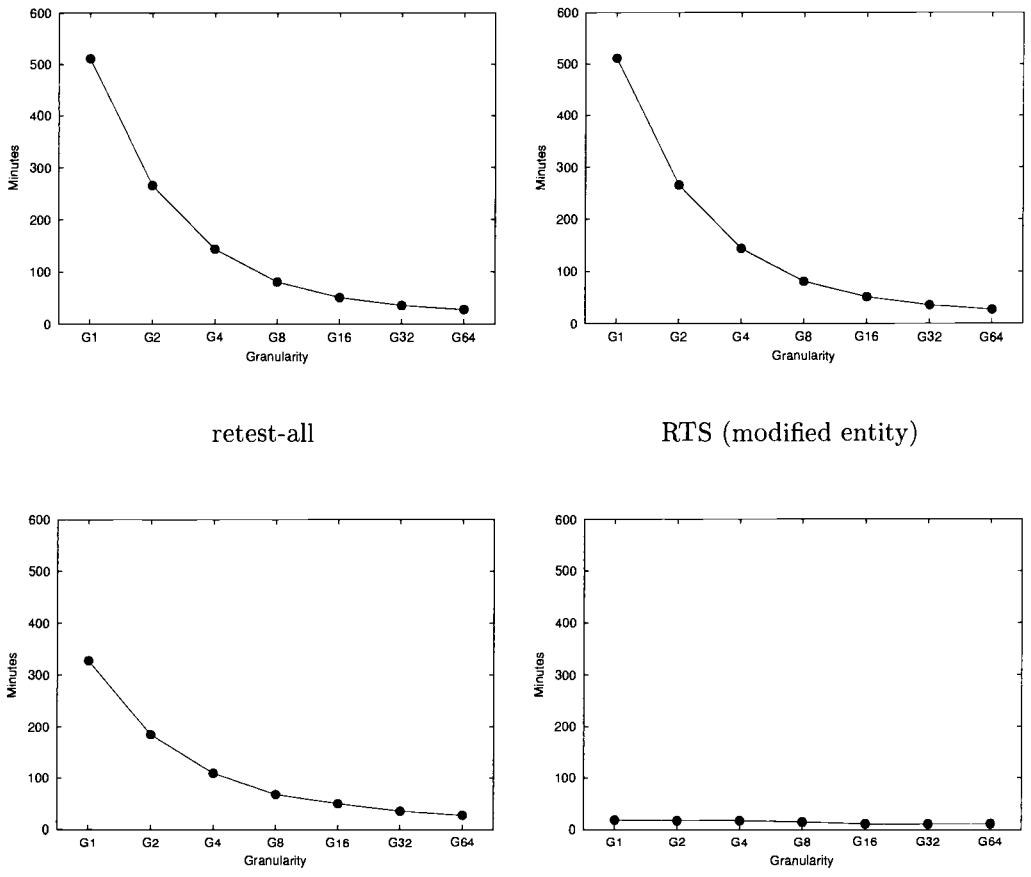
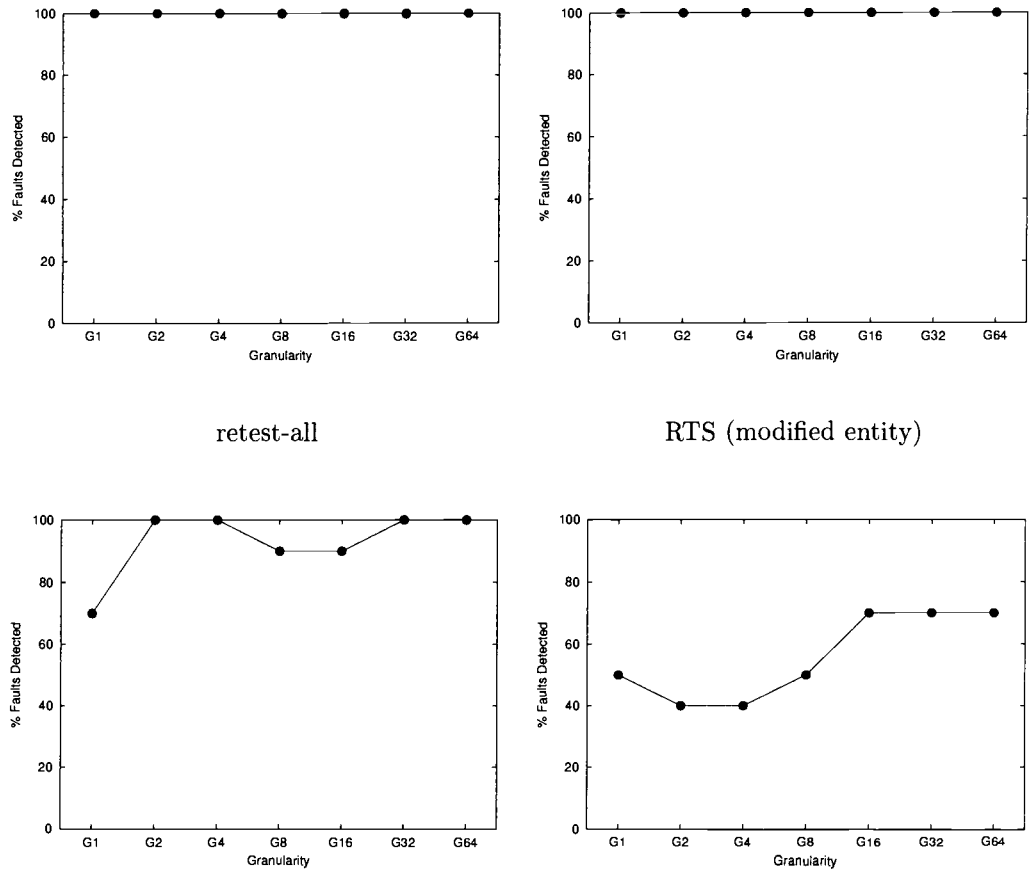


FIGURE A.12: Test execution time for regression test selection techniques across test suite granularities, for version 3, for the functional grouping of test cases.



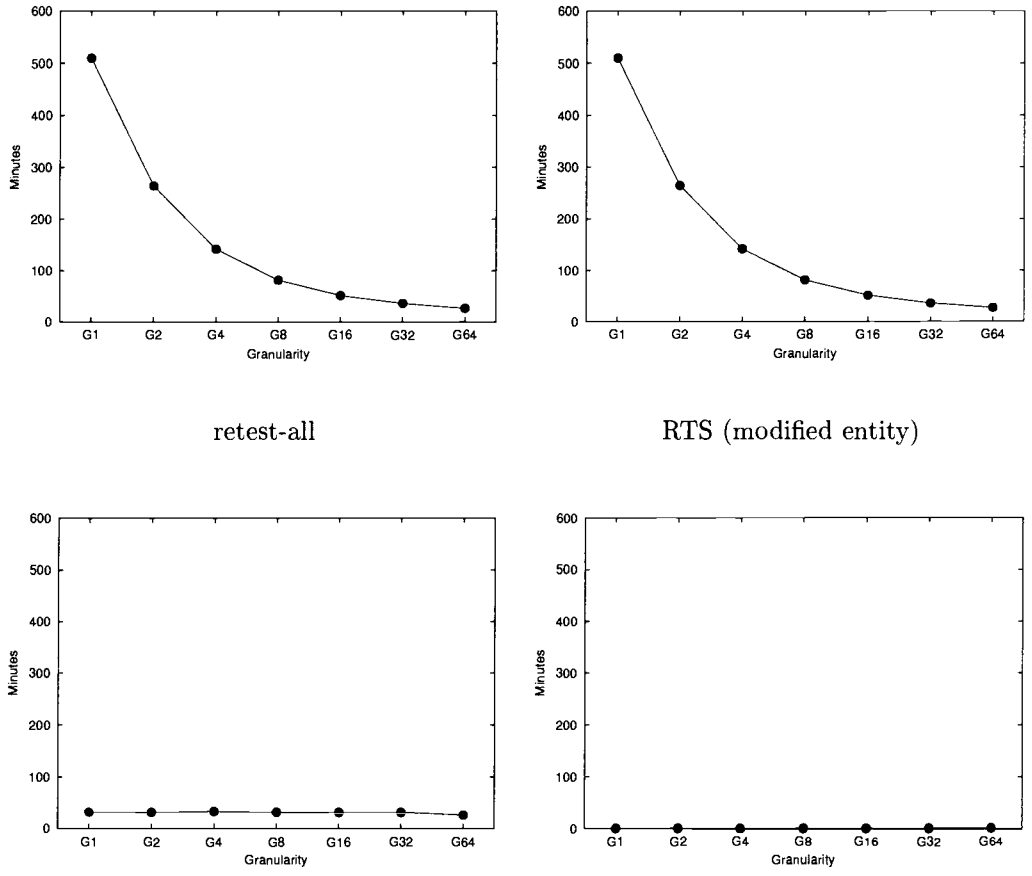
retest-all

RTS (modified entity)

RTS (modified non-core entity)

Fischer minimization

FIGURE A.13: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 4, for the random grouping of test cases.



retest-all

RTS (modified entity)

RTS (modified non-core entity)

Fischer minimization

FIGURE A.14: Test execution time for regression test selection techniques across test suite granularities, for version 4, for the random grouping of test cases.

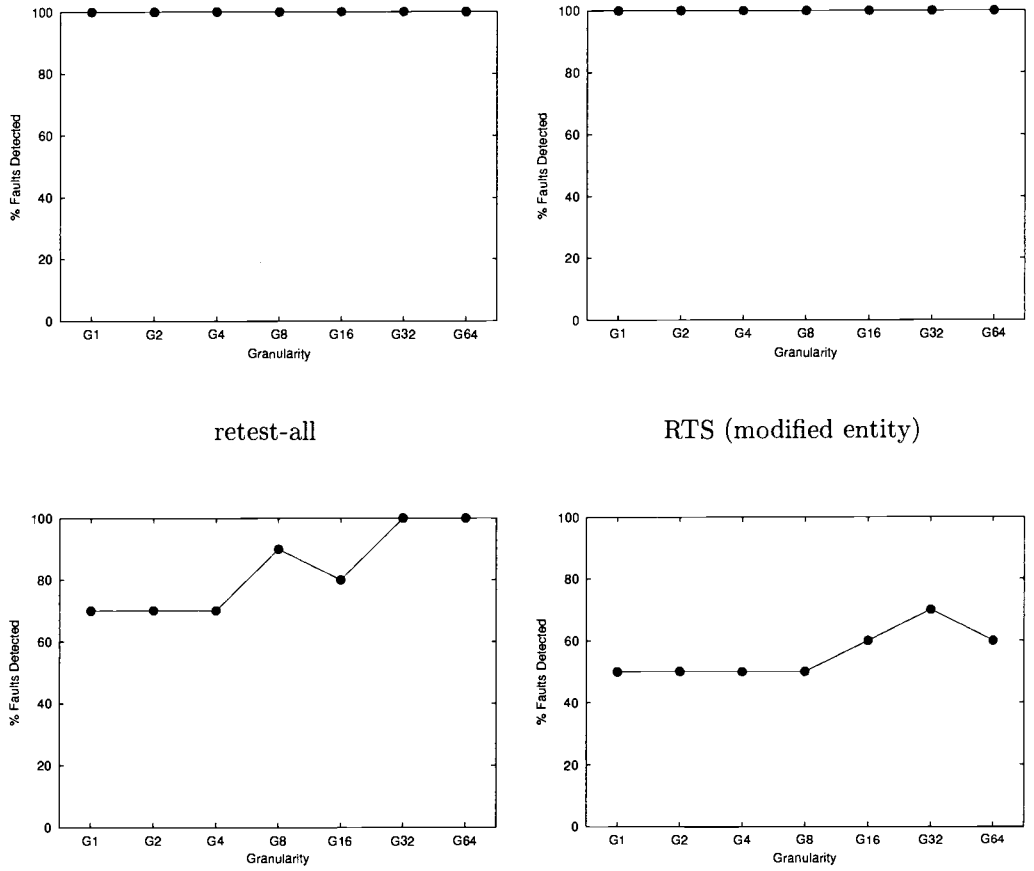


FIGURE A.15: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 4, for the functional grouping of test cases.

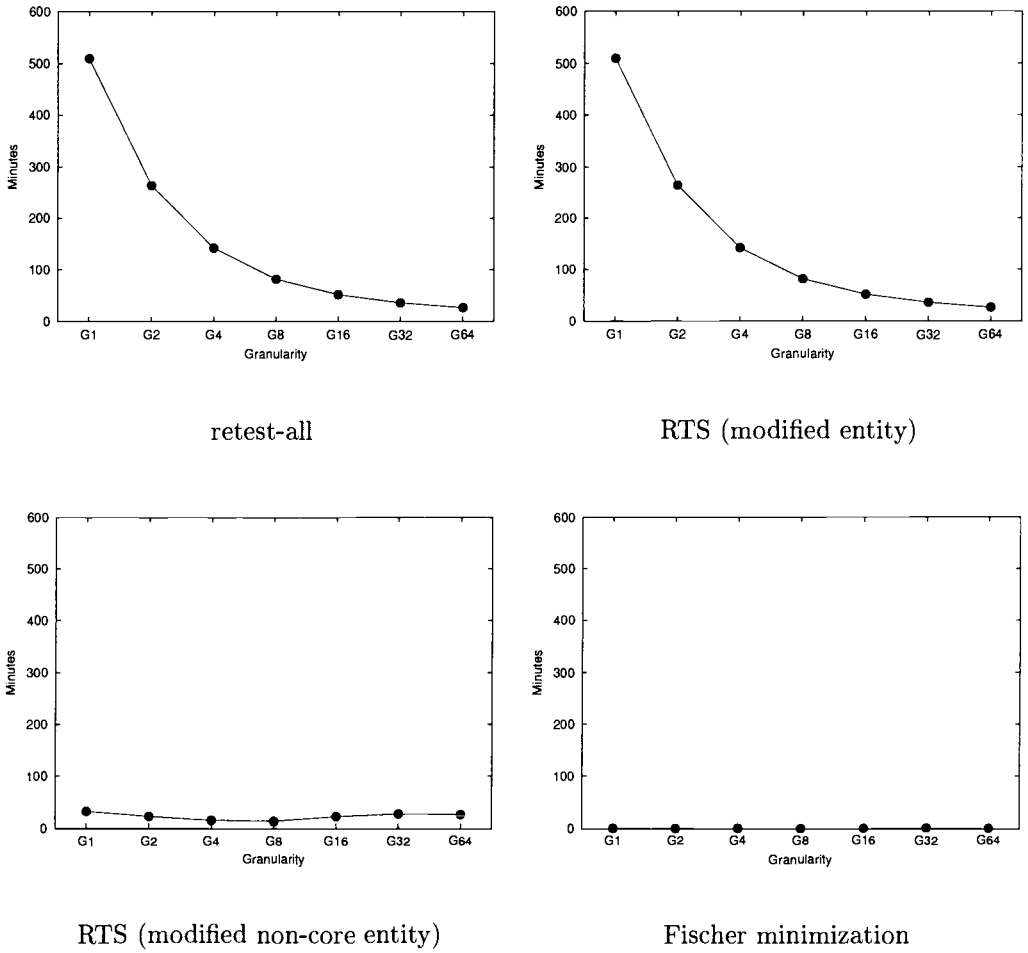
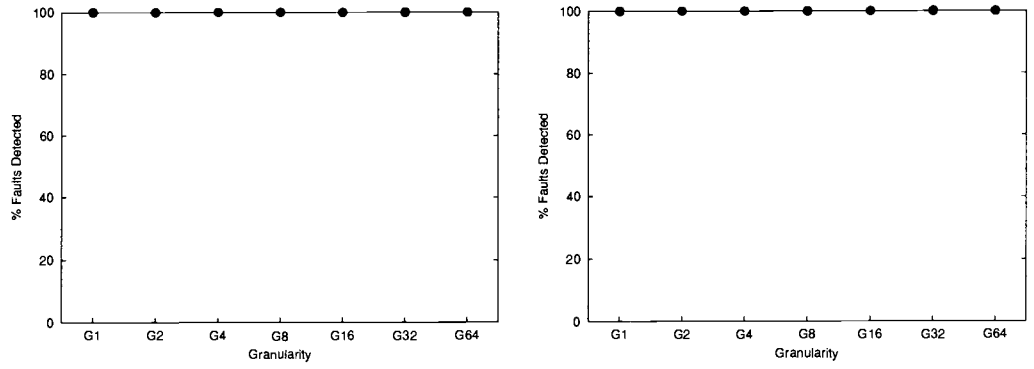
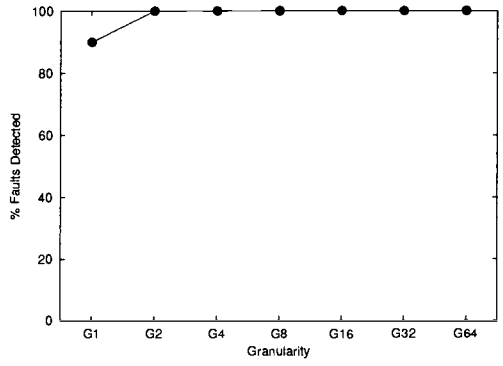


FIGURE A.16: Test execution time for regression test selection techniques across test suite granularities, for version 4, for the functional grouping of test cases.

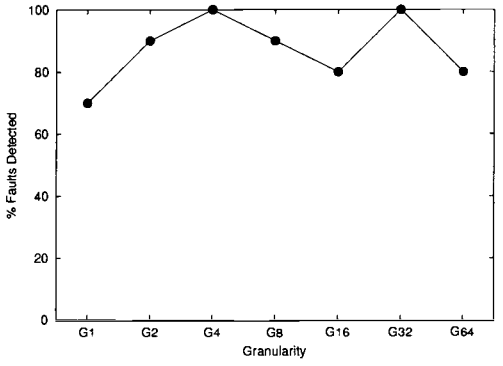


retest-all

RTS (modified entity)

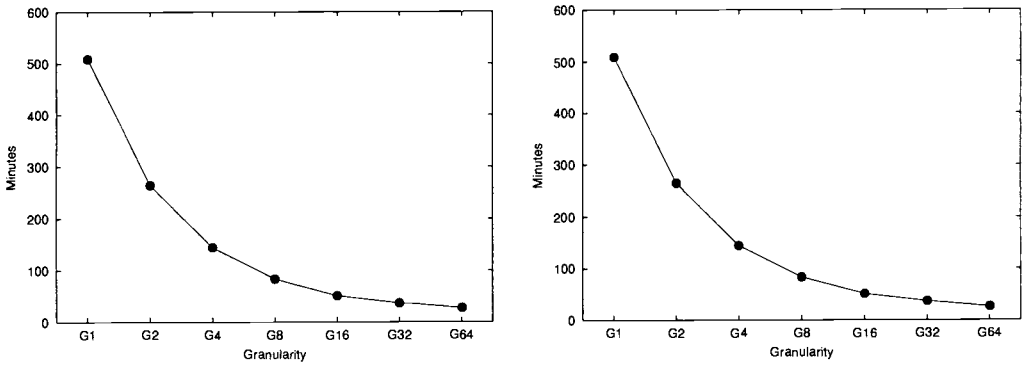


RTS (modified non-core entity)



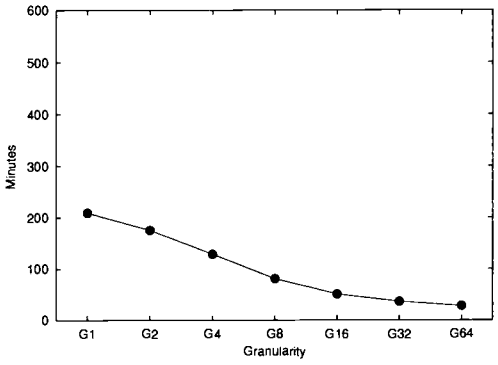
Fischer minimization

FIGURE A.17: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 5, for the random grouping of test cases.

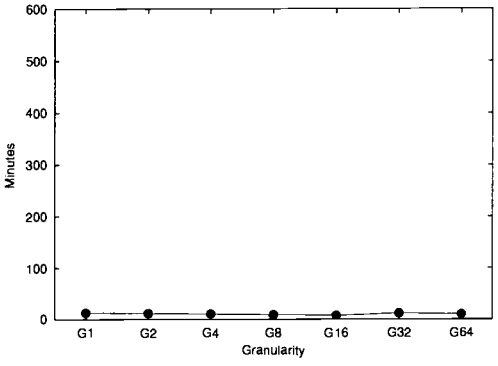


retest-all

RTS (modified entity)



RTS (modified non-core entity)



Fischer minimization

FIGURE A.18: Test execution time for regression test selection techniques across test suite granularities, for version 5, for the random grouping of test cases.

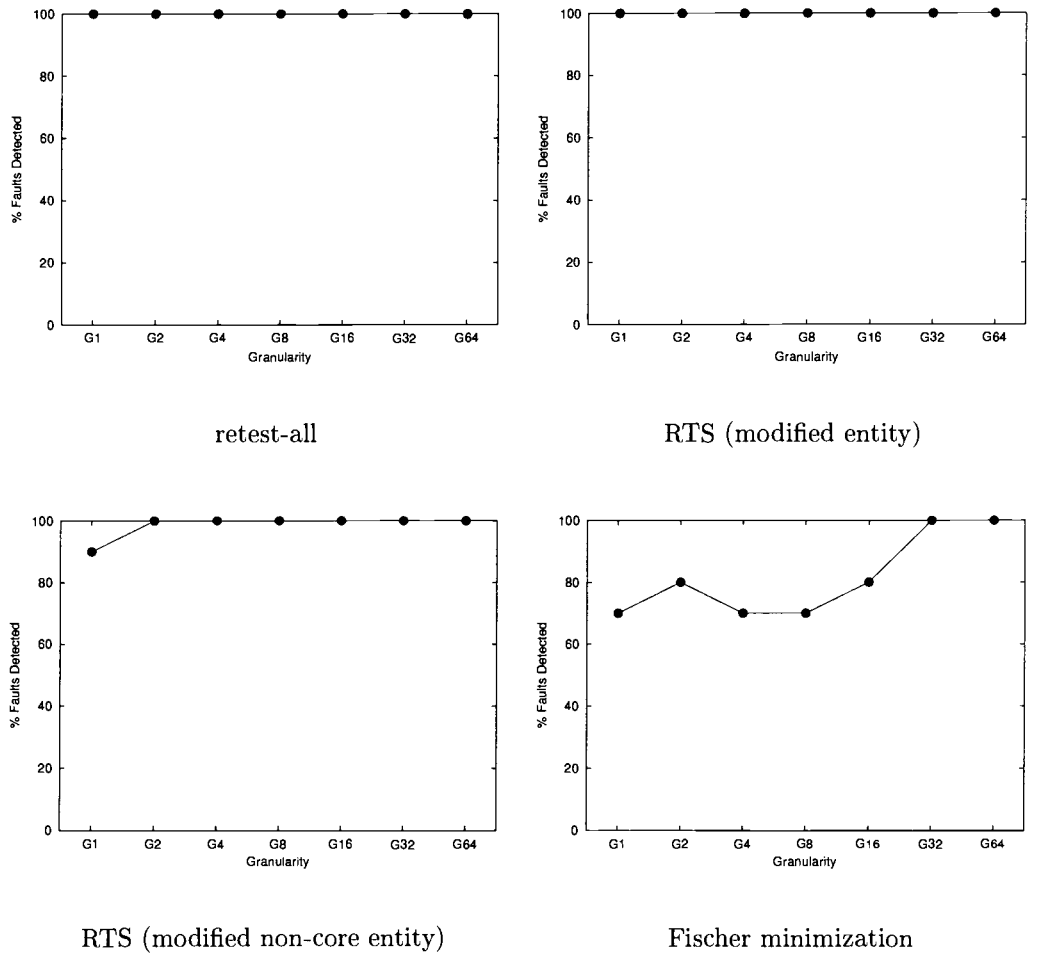


FIGURE A.19: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 5, for the functional grouping of test cases.

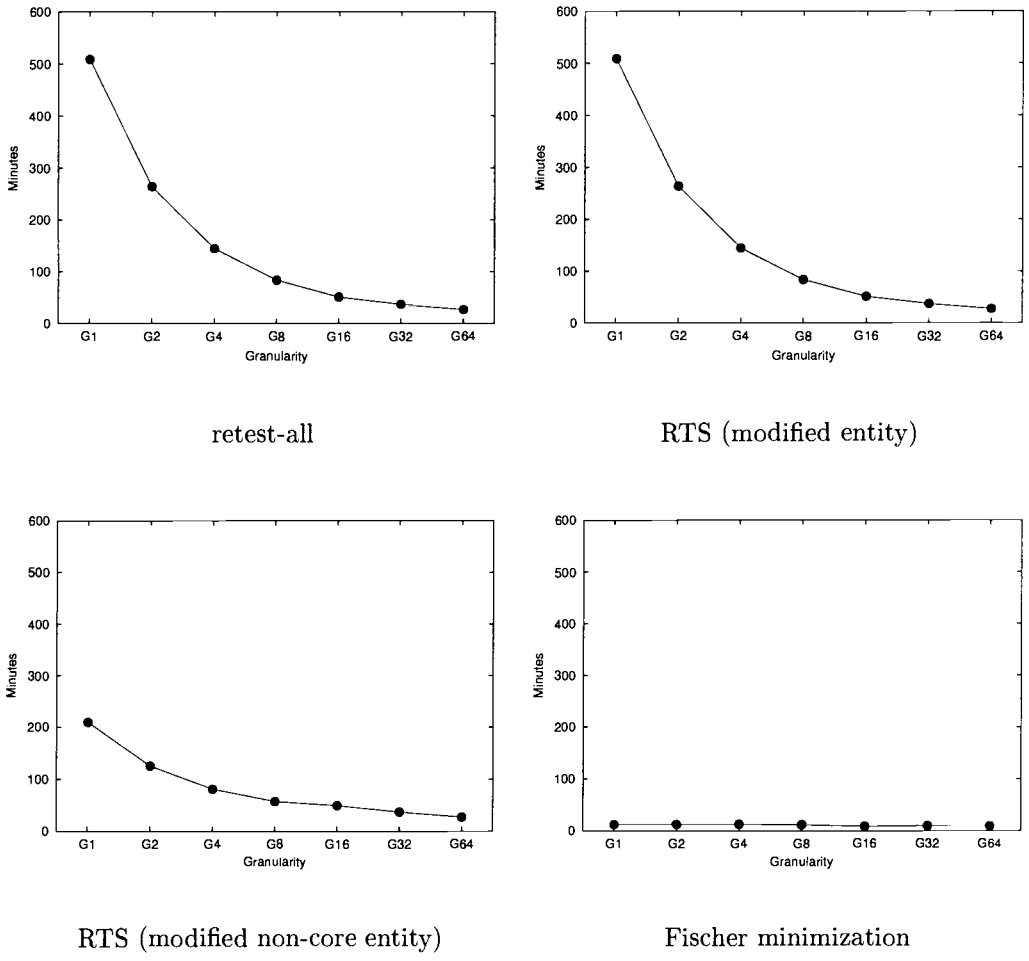
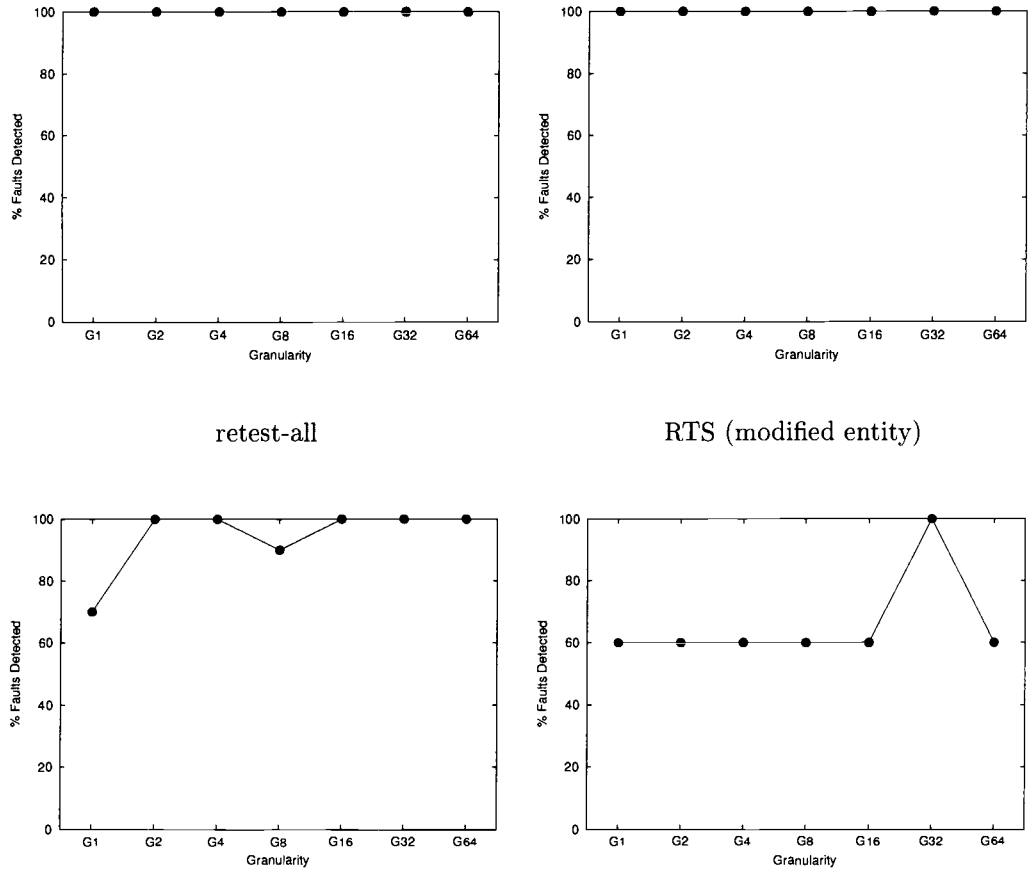


FIGURE A.20: Test execution time for regression test selection techniques across test suite granularities, for version 5, for the functional grouping of test cases.



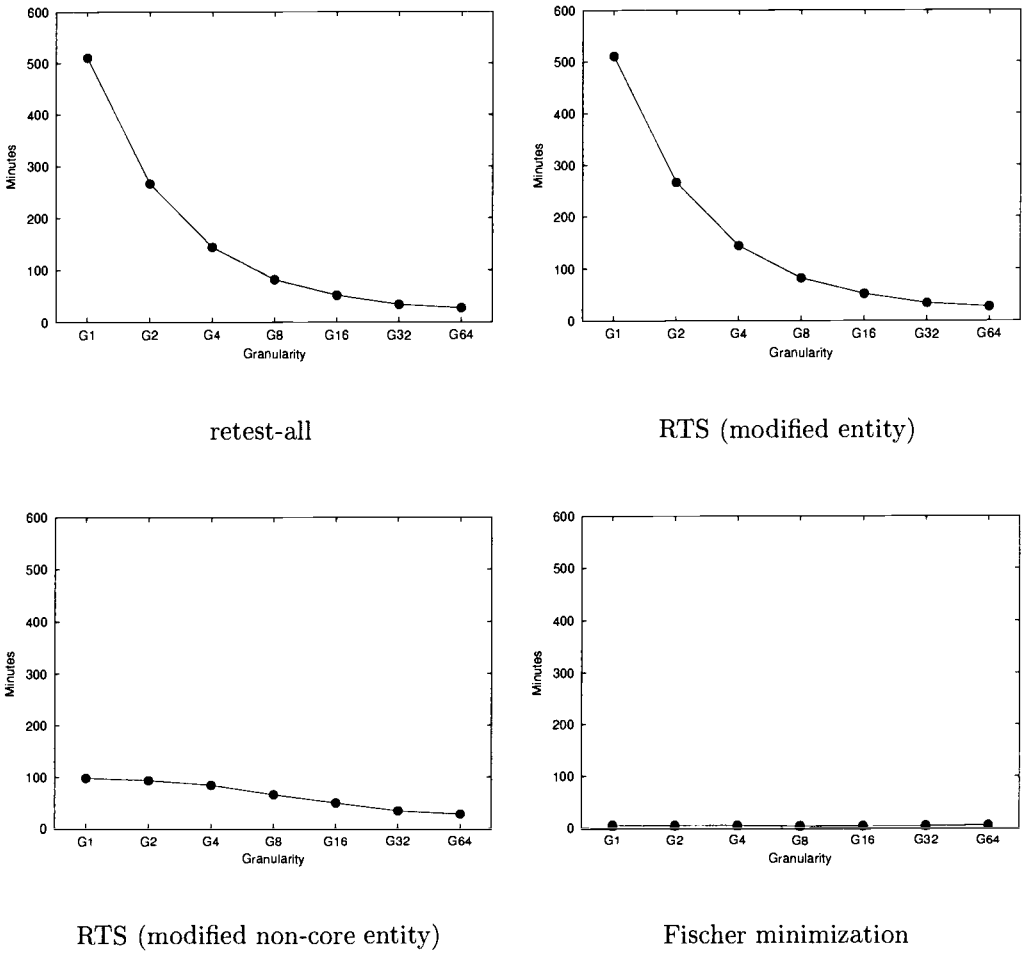
retest-all

RTS (modified entity)

RTS (modified non-core entity)

Fischer minimization

FIGURE A.21: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 6, for the random grouping of test cases.



retest-all

RTS (modified entity)

RTS (modified non-core entity)

Fischer minimization

FIGURE A.22: Test execution time for regression test selection techniques across test suite granularities, for version 6, for the random grouping of test cases.

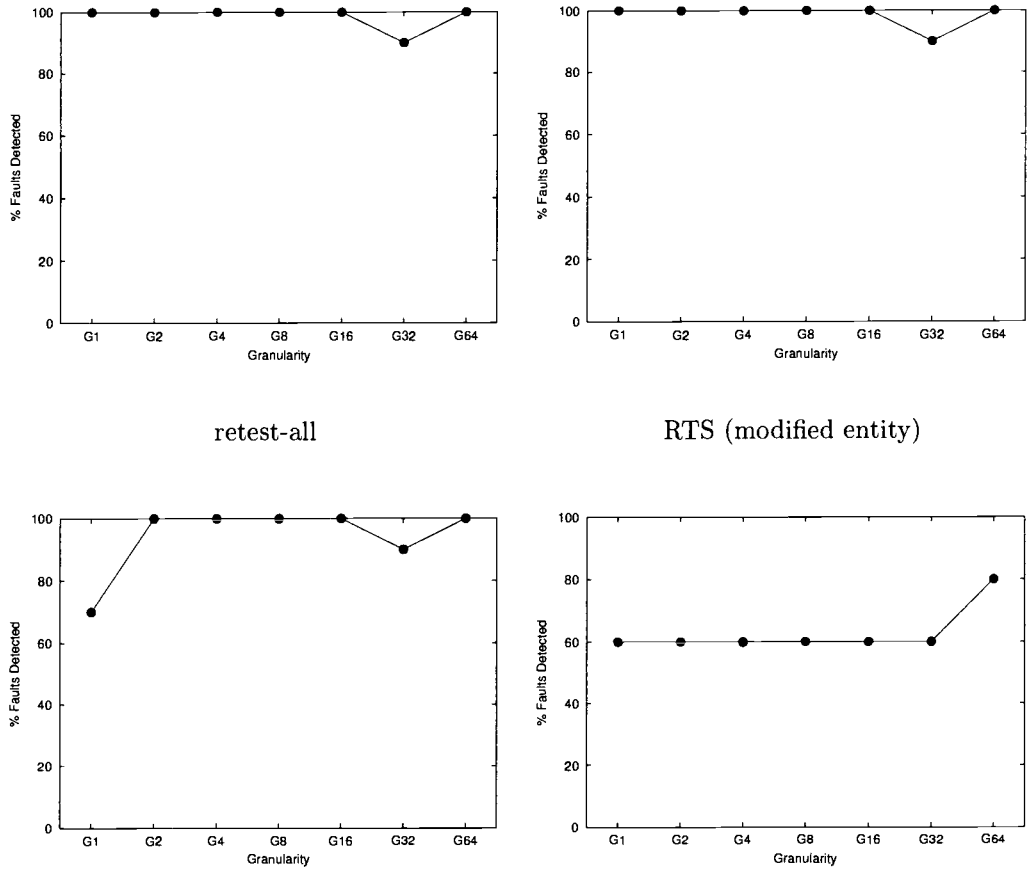
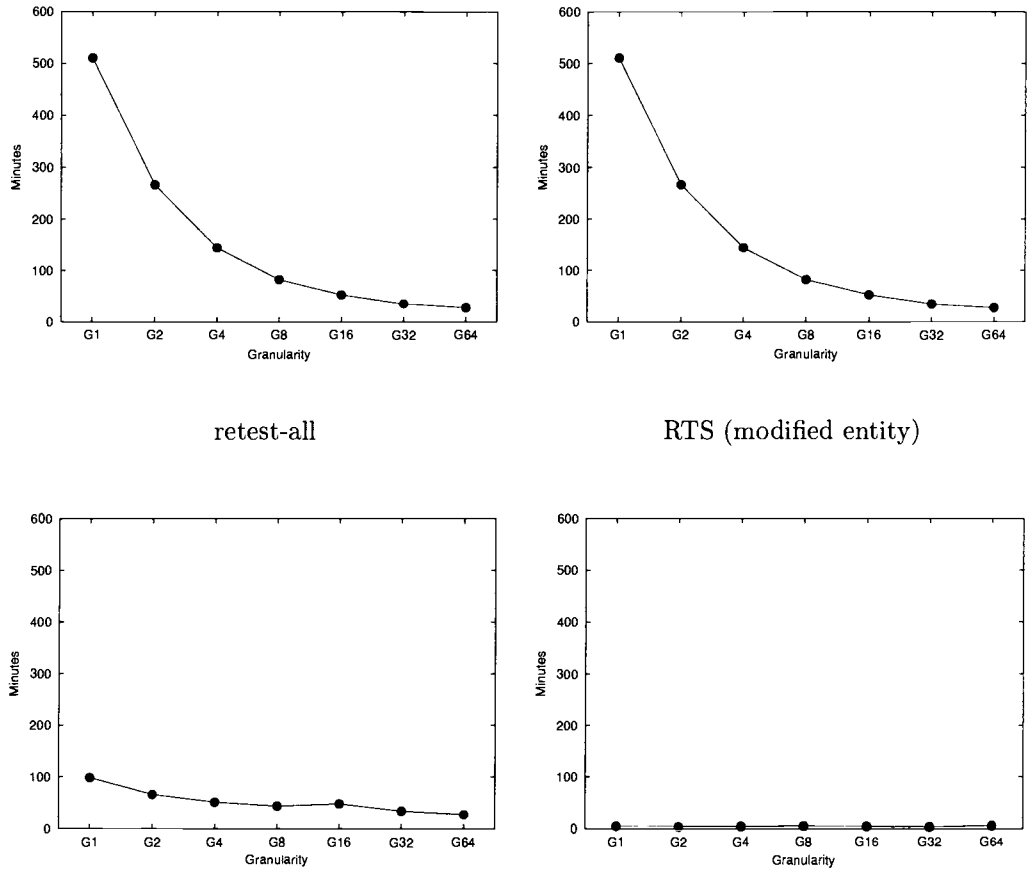


FIGURE A.23: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 6, for the functional grouping of test cases.



retest-all

RTS (modified entity)

RTS (modified non-core entity)

Fischer minimization

FIGURE A.24: Test execution time for regression test selection techniques across test suite granularities, for version 6, for the functional grouping of test cases.

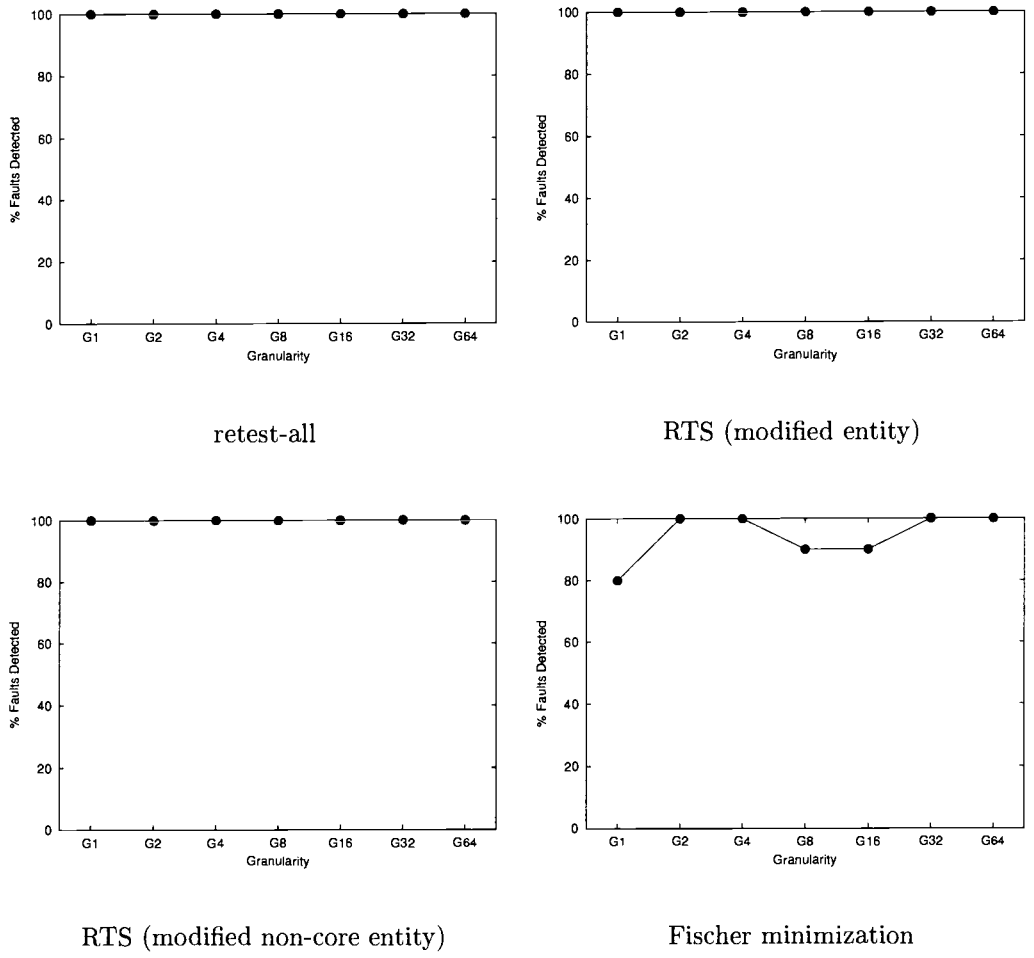


FIGURE A.25: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 7, for the random grouping of test cases.

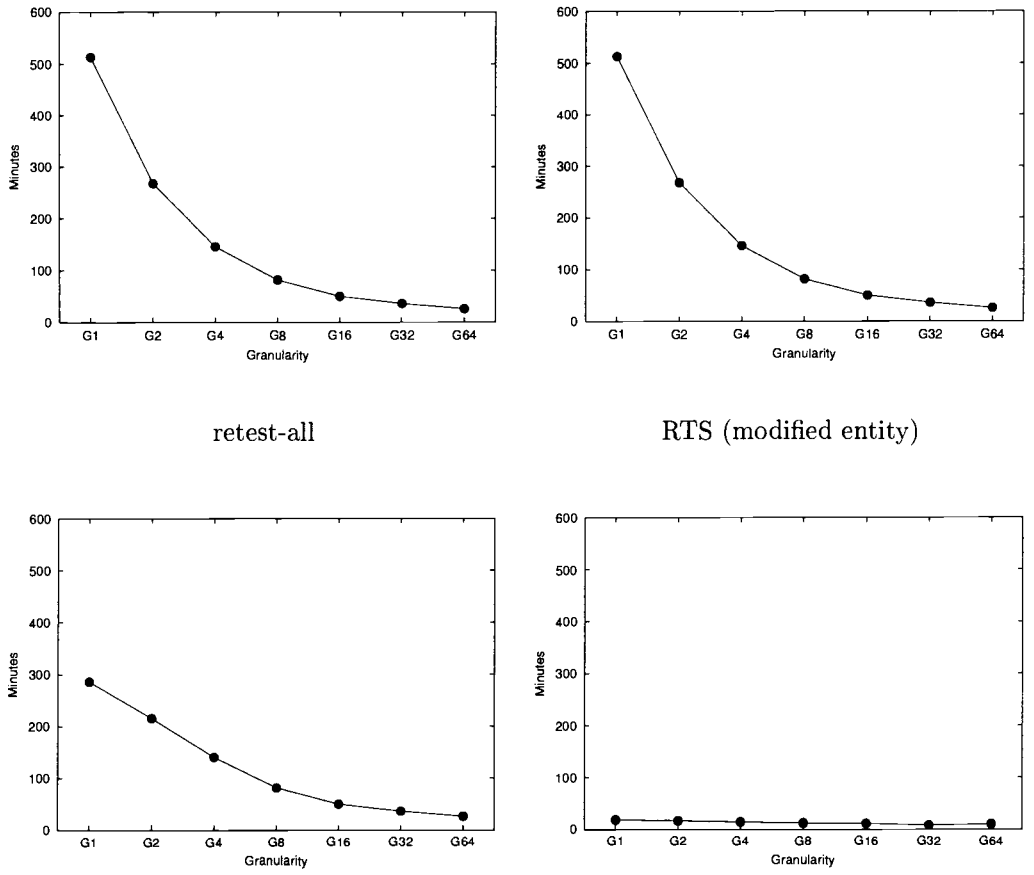
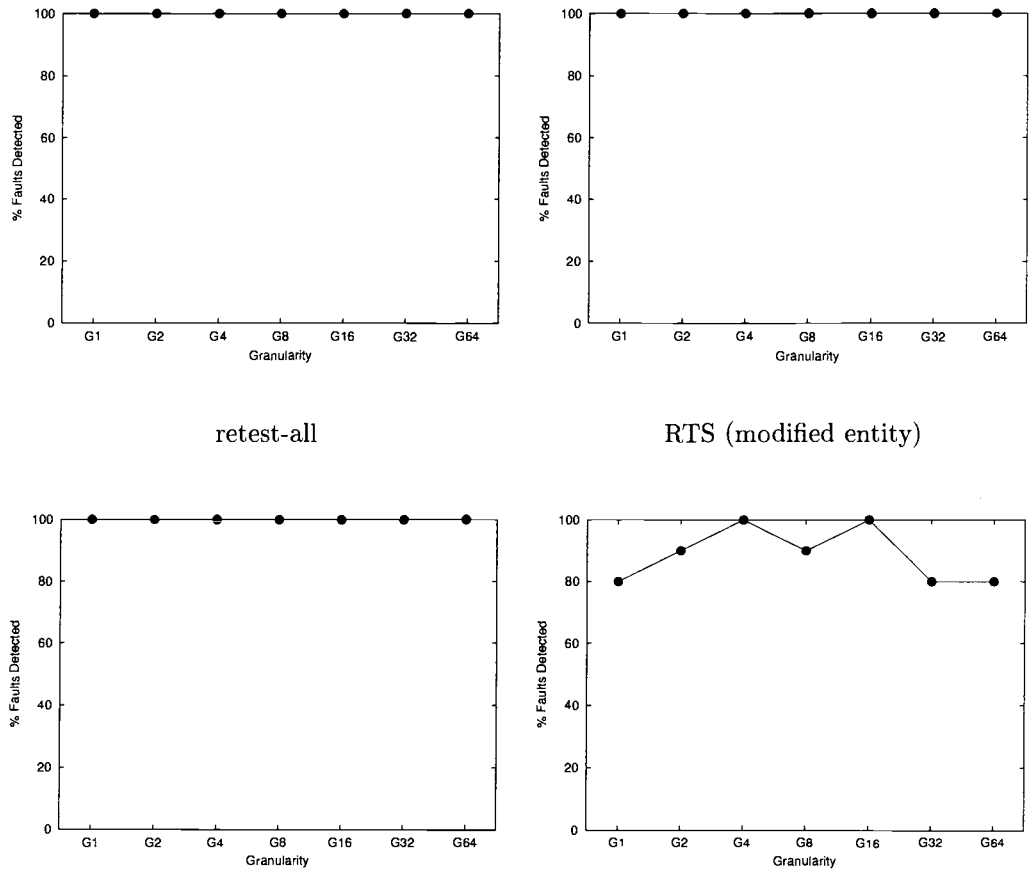


FIGURE A.26: Test execution time for regression test selection techniques across test suite granularities, for version 7, for the random grouping of test cases.



retest-all

RTS (modified entity)

RTS (modified non-core entity)

Fischer minimization

FIGURE A.27: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 7, for the functional grouping of test cases.

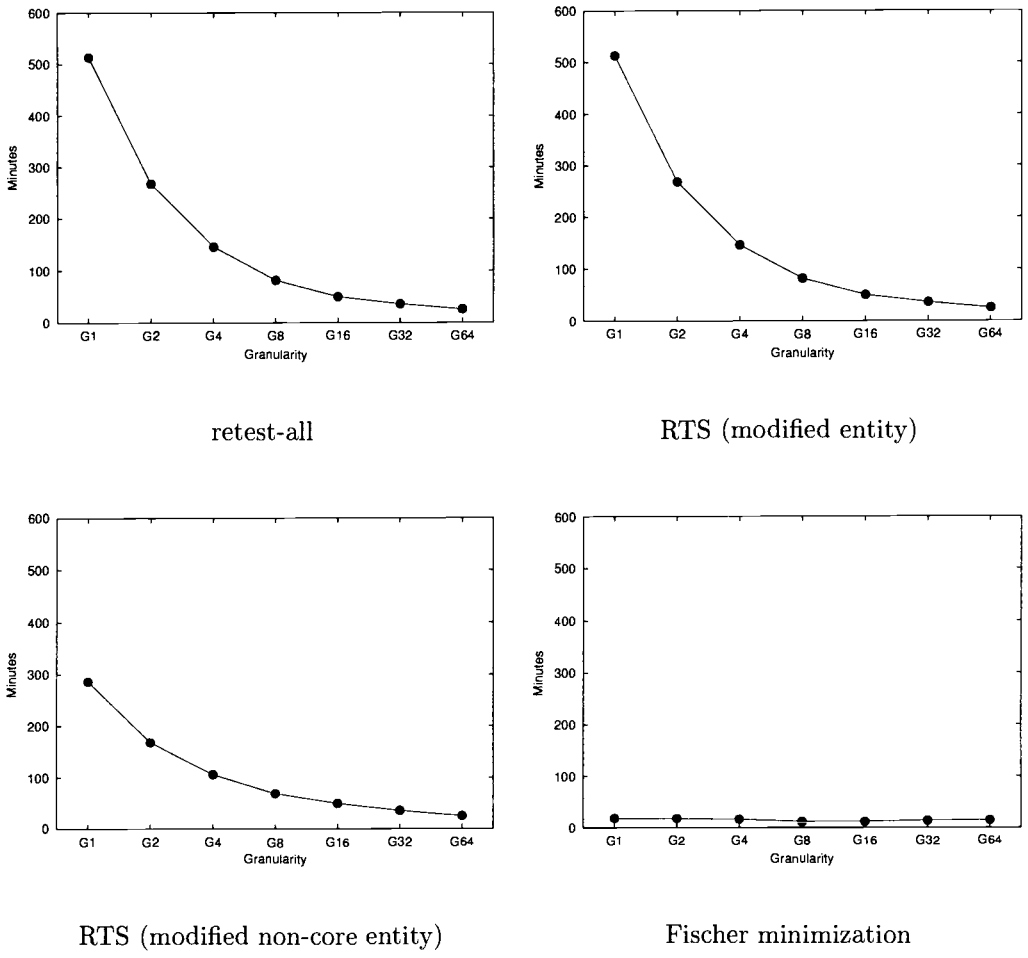
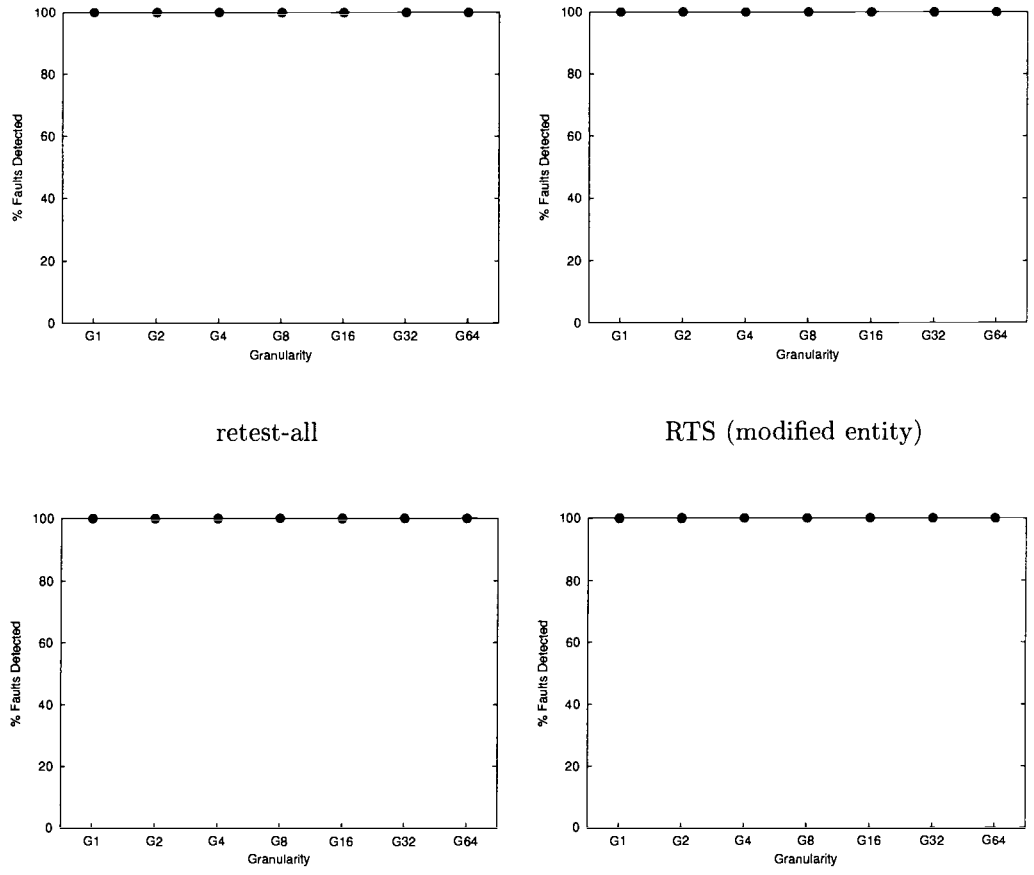


FIGURE A.28: Test execution time for regression test selection techniques across test suite granularities, for version 7, for the functional grouping of test cases.



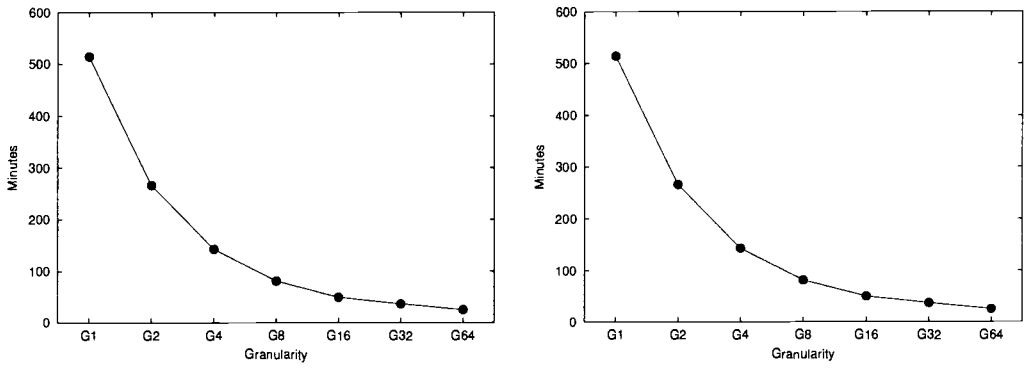
retest-all

RTS (modified entity)

RTS (modified non-core entity)

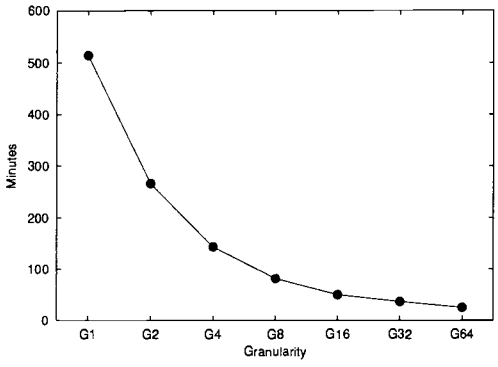
Fischer minimization

FIGURE A.29: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 8, for the random grouping of test cases.

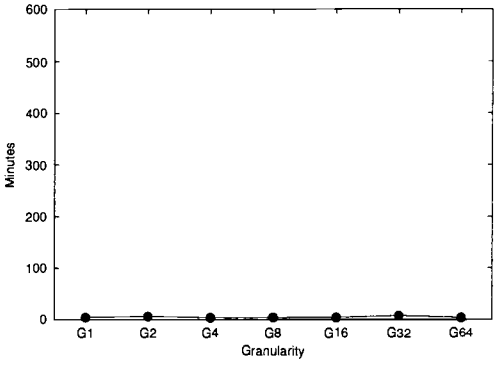


retest-all

RTS (modified entity)



RTS (modified non-core entity)



Fischer minimization

FIGURE A.30: Test execution time for regression test selection techniques across test suite granularities, for version 8, for the random grouping of test cases.

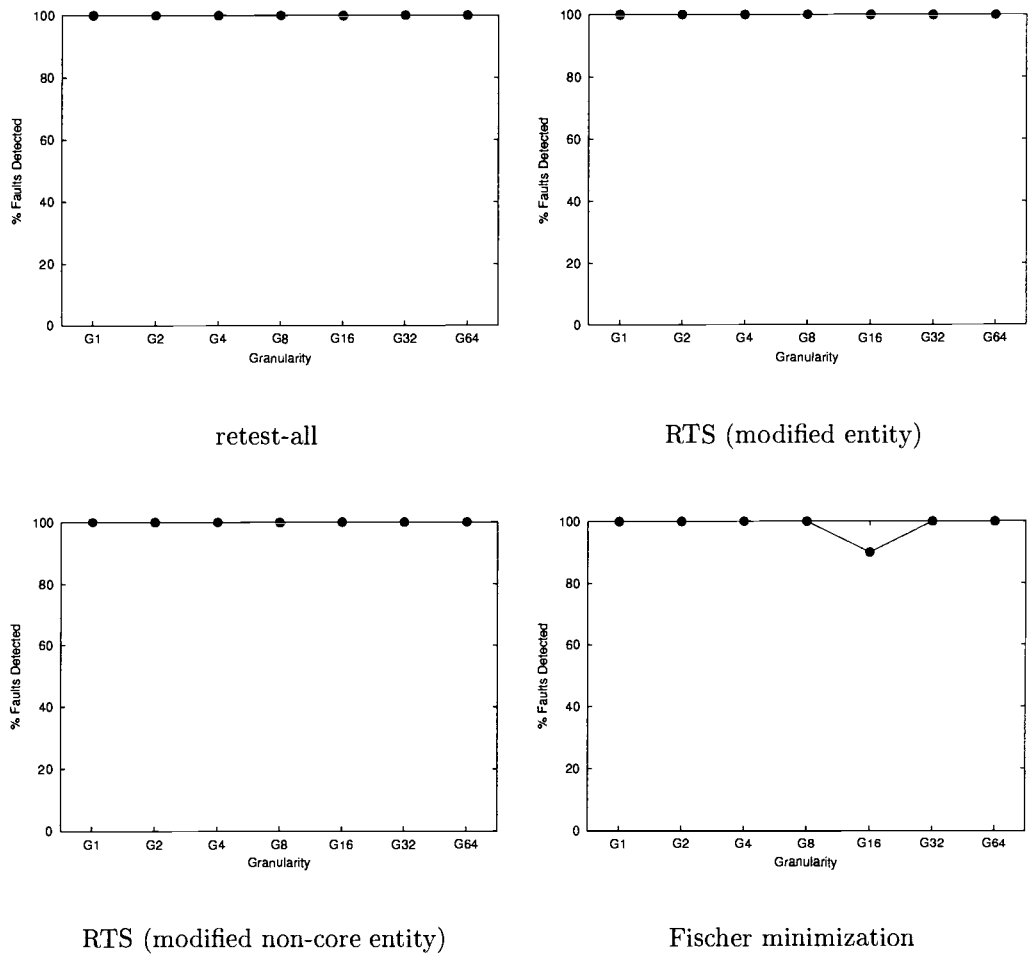


FIGURE A.31: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 8, for the functional grouping of test cases.

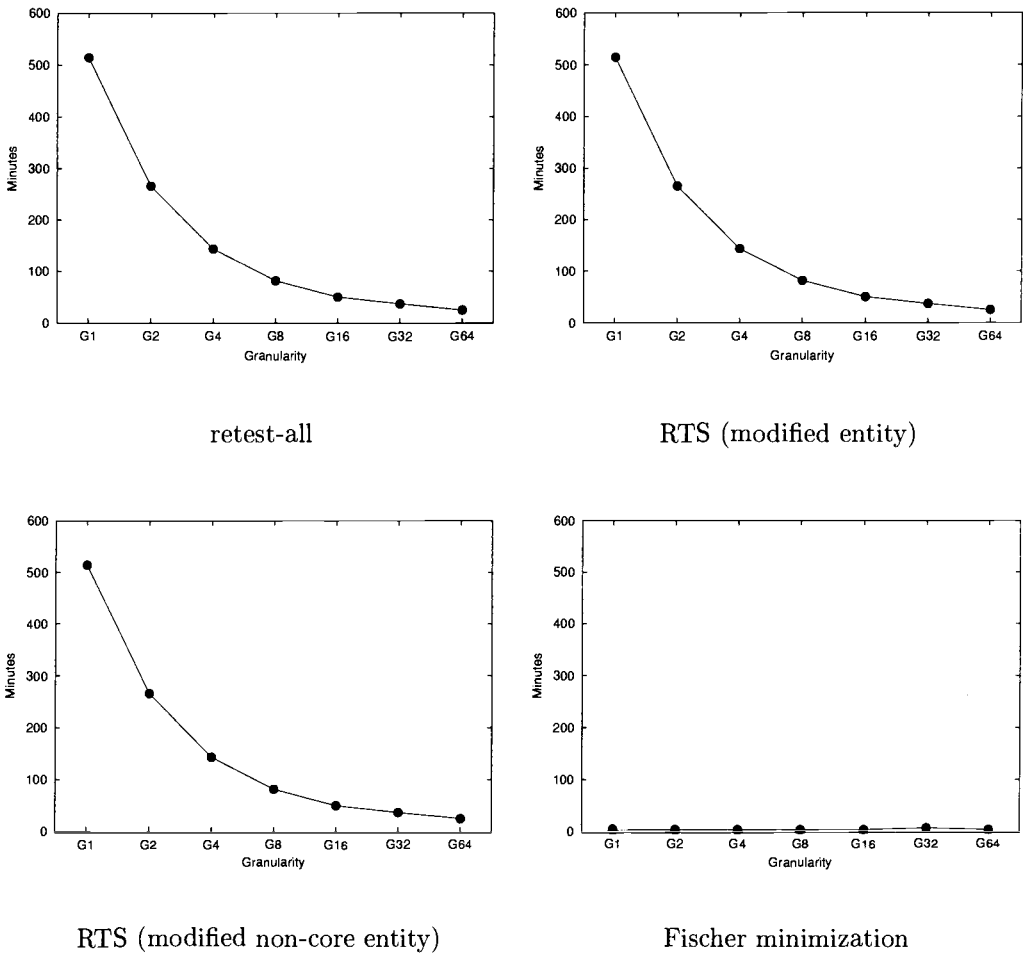


FIGURE A.32: Test execution time for regression test selection techniques across test suite granularities, for version 8, for the functional grouping of test cases.

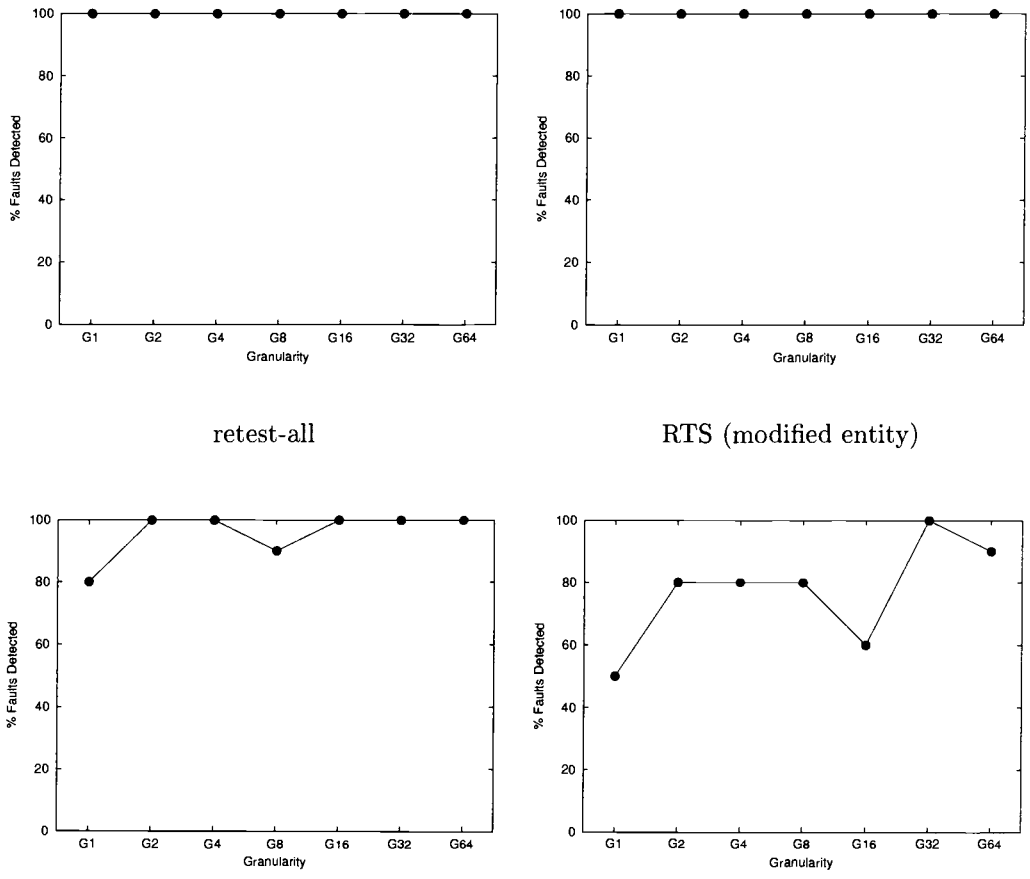


FIGURE A.33: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 9, for the random grouping of test cases.

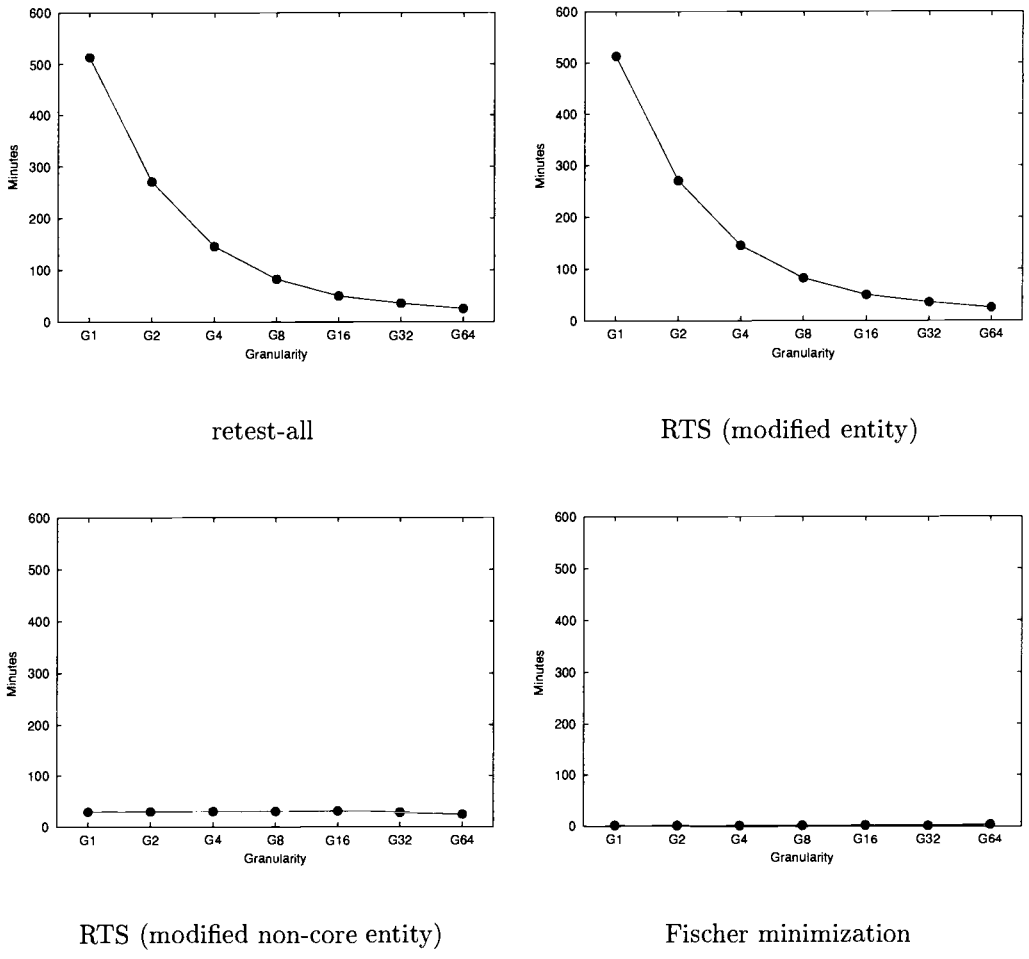
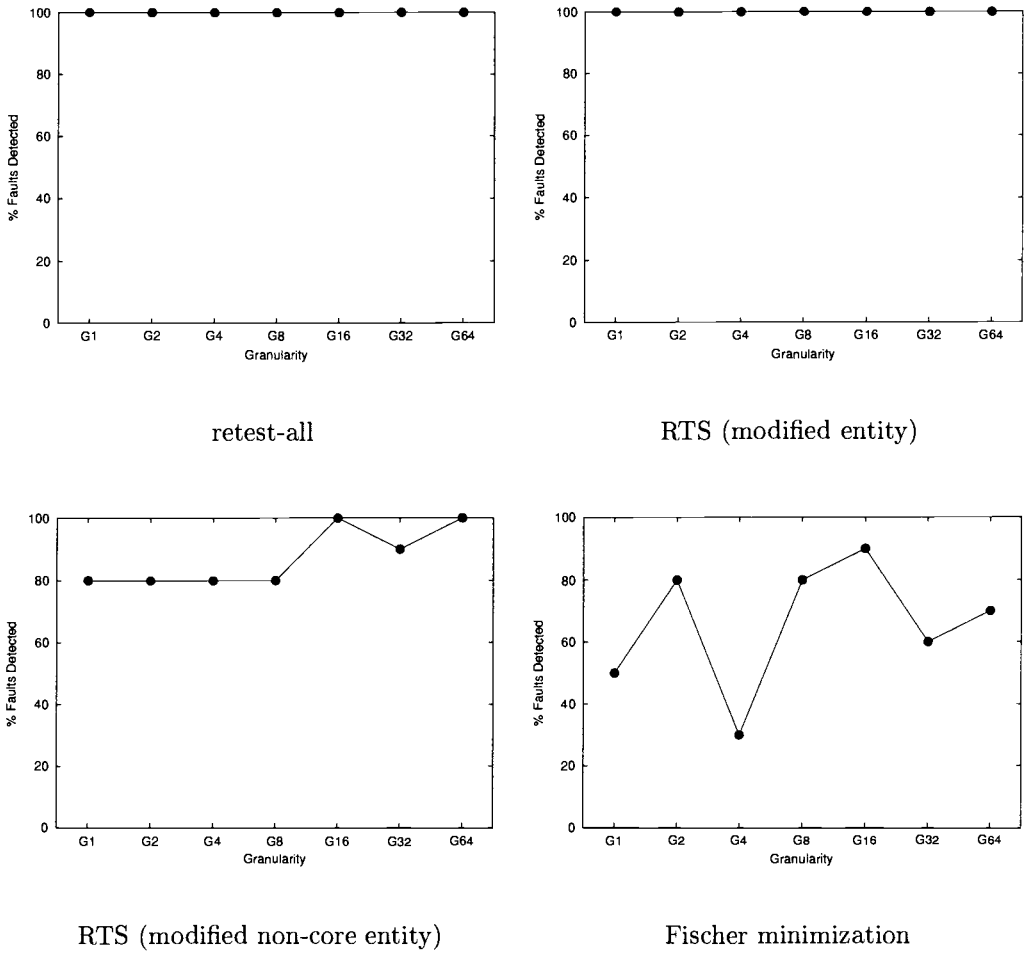


FIGURE A.34: Test execution time for regression test selection techniques across test suite granularities, for version 9, for the random grouping of test cases.



retest-all

RTS (modified entity)

RTS (modified non-core entity)

Fischer minimization

FIGURE A.35: Fault detection effectiveness for regression test selection techniques across test suite granularities, for version 9, for the functional grouping of test cases.

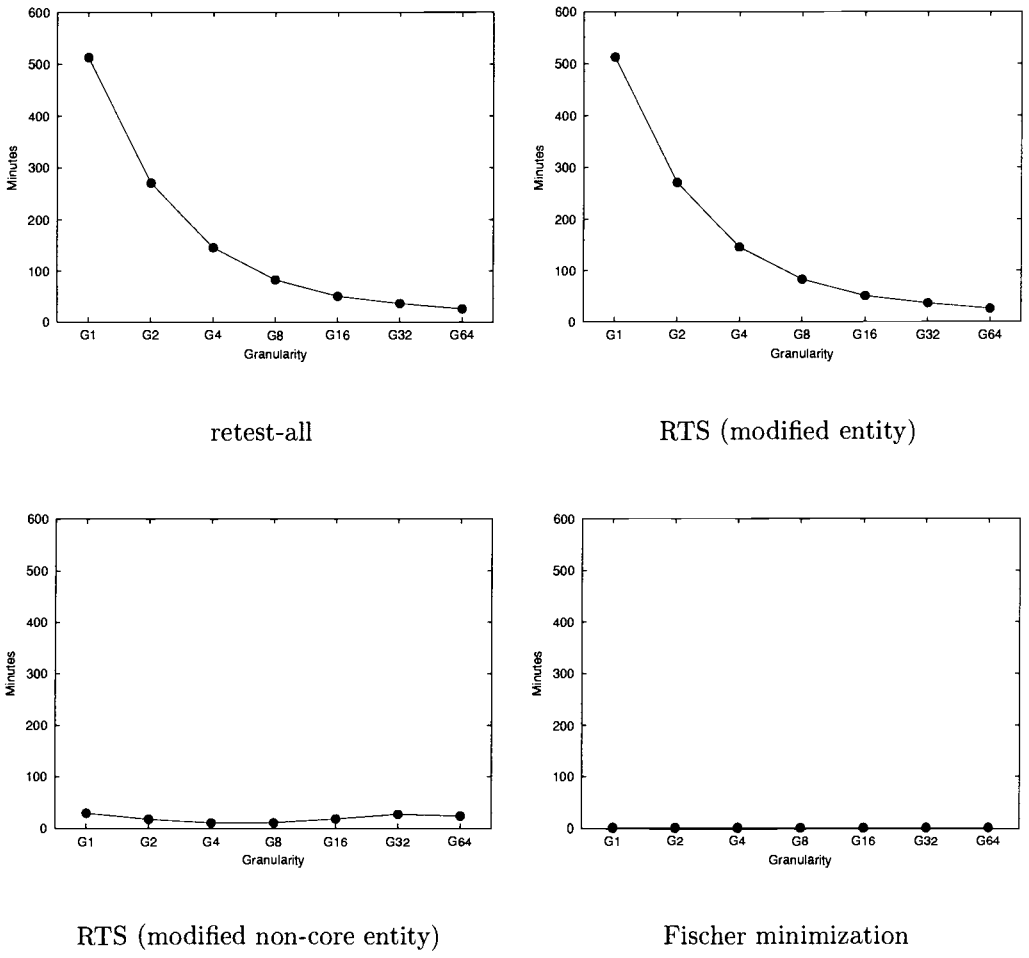
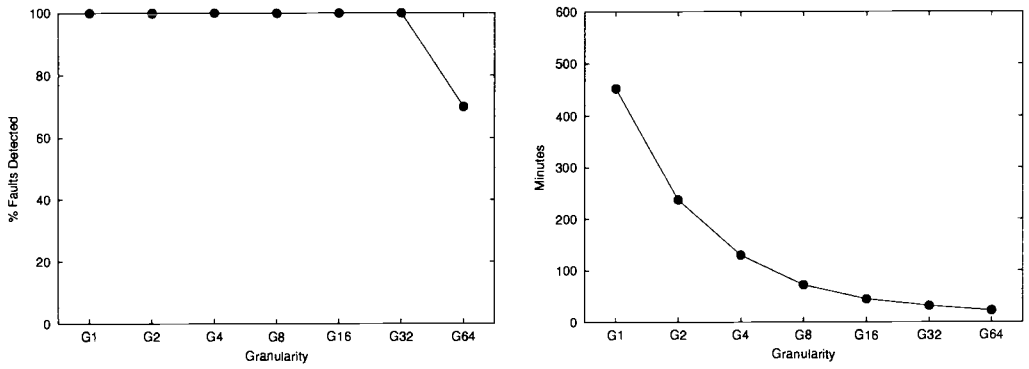
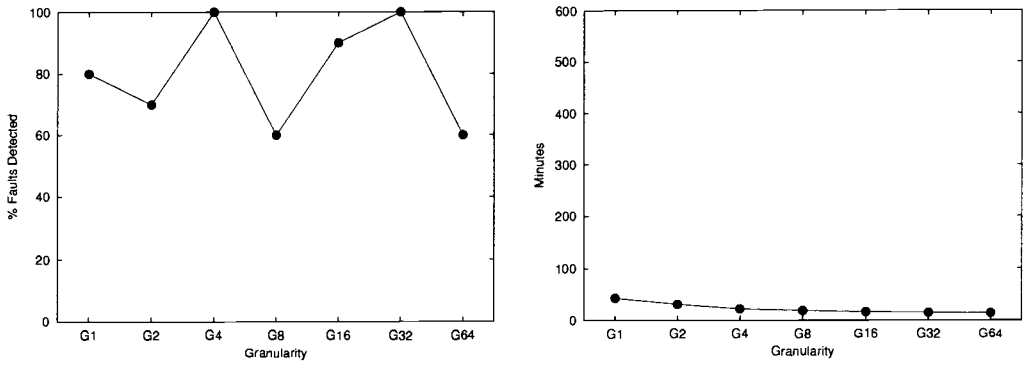


FIGURE A.36: Test execution time for regression test selection techniques across test suite granularities, for version 9, for the functional grouping of test cases.



retest-all

retest-all



GHS test suite reduction

GHS test suite reduction

FIGURE A.37: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 1, for the random grouping of test cases.

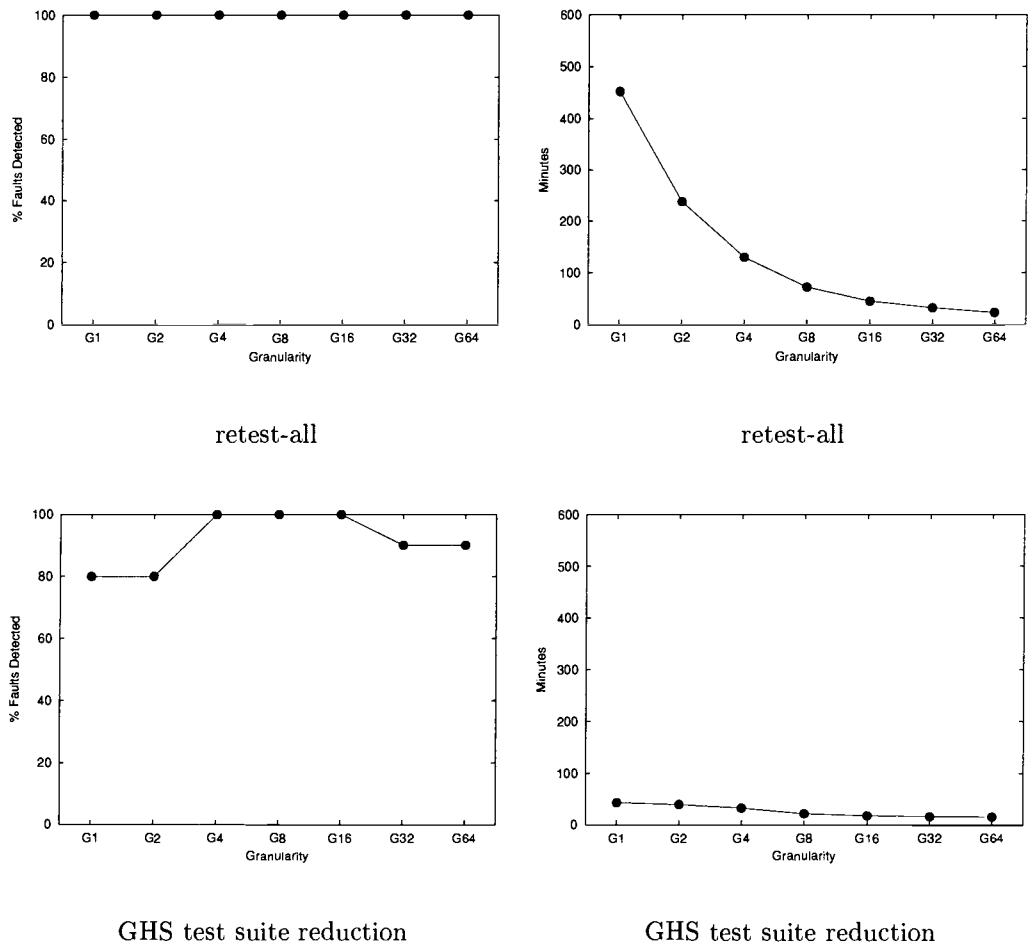
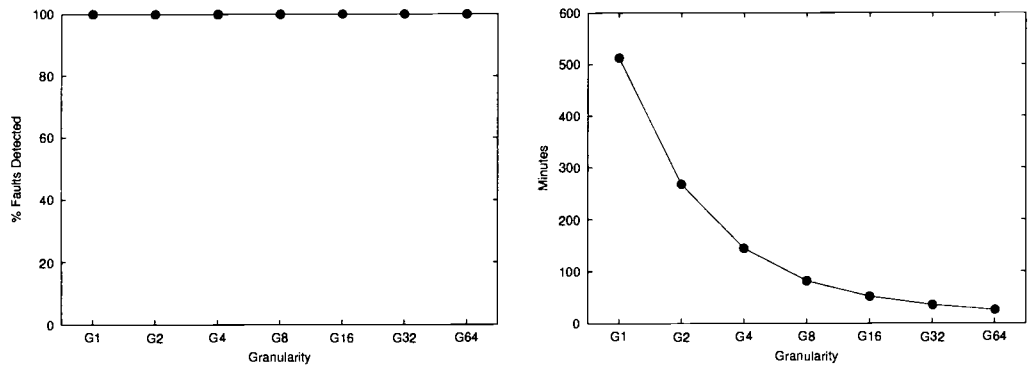
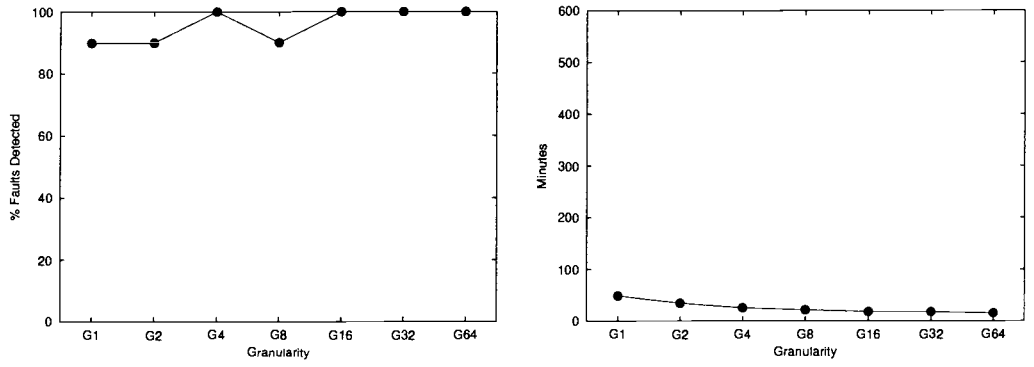


FIGURE A.38: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 1, for the functional grouping of test cases.



retest-all

retest-all



GHS test suite reduction

GHS test suite reduction

FIGURE A.39: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 2, for the random grouping of test cases.

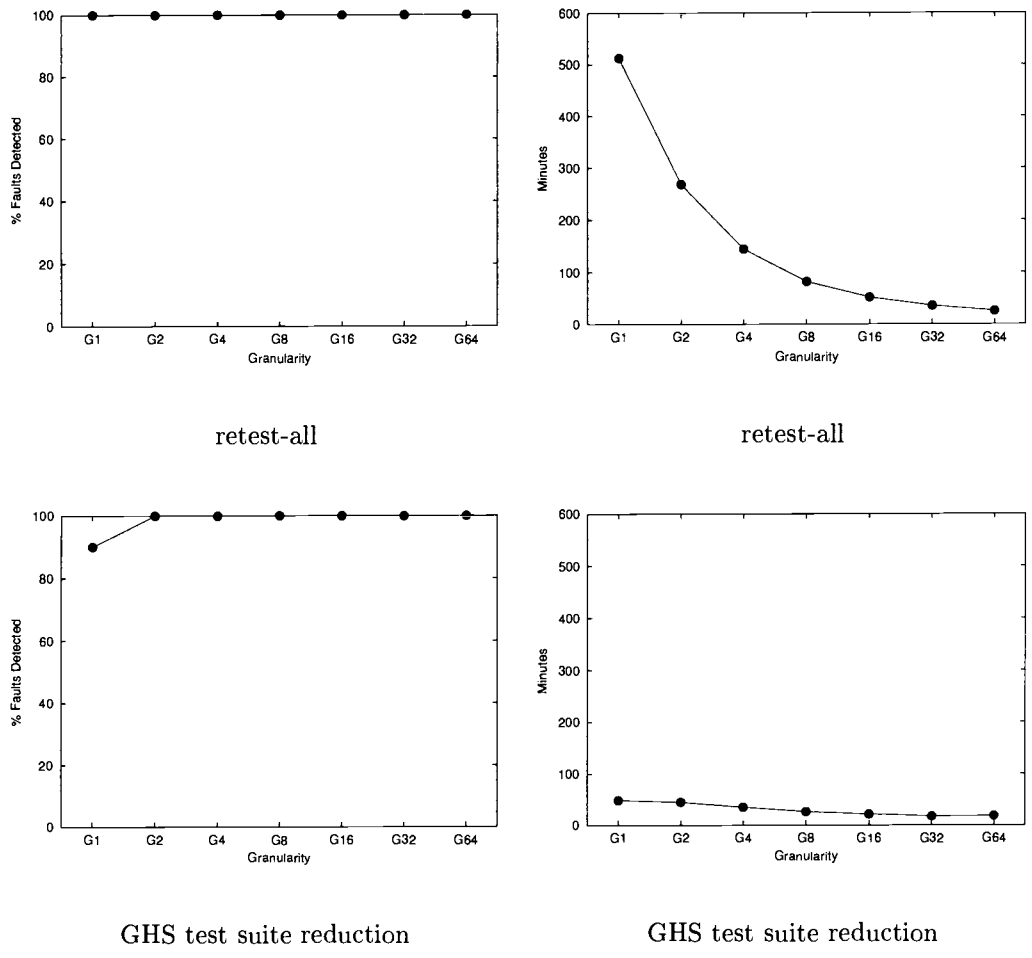


FIGURE A.40: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 2, for the functional grouping of test cases.

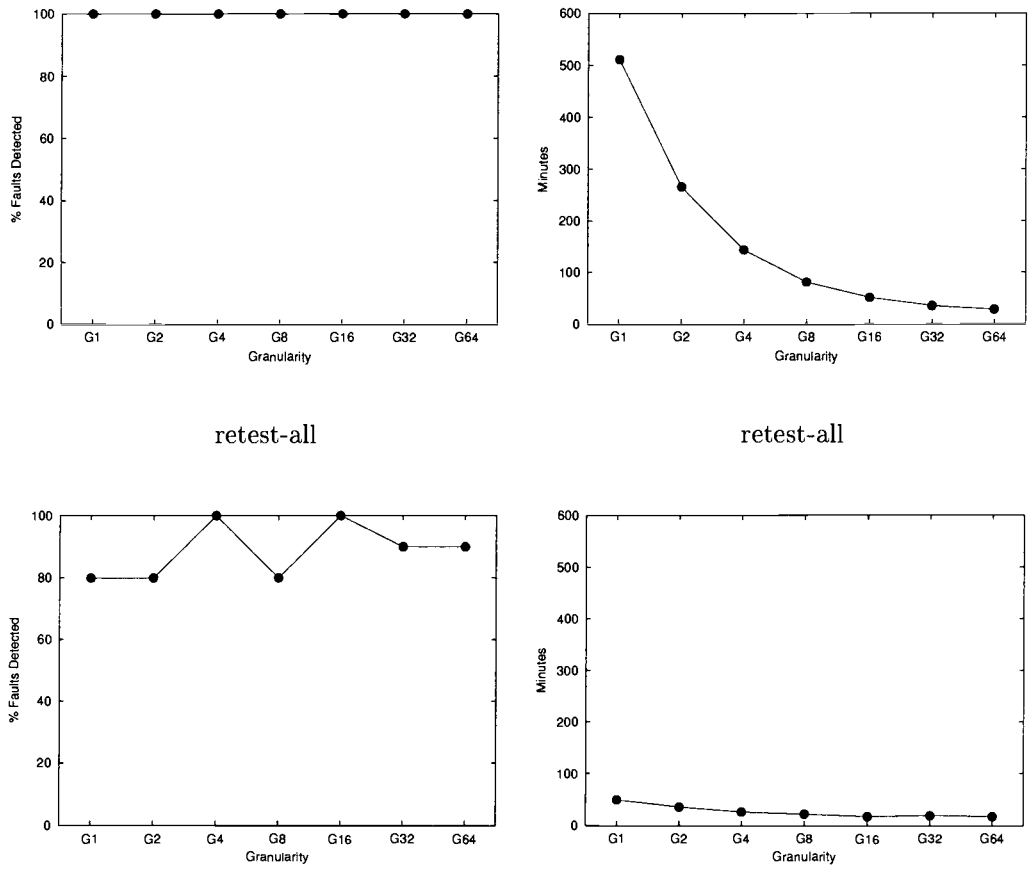


FIGURE A.41: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 3, for the random grouping of test cases.

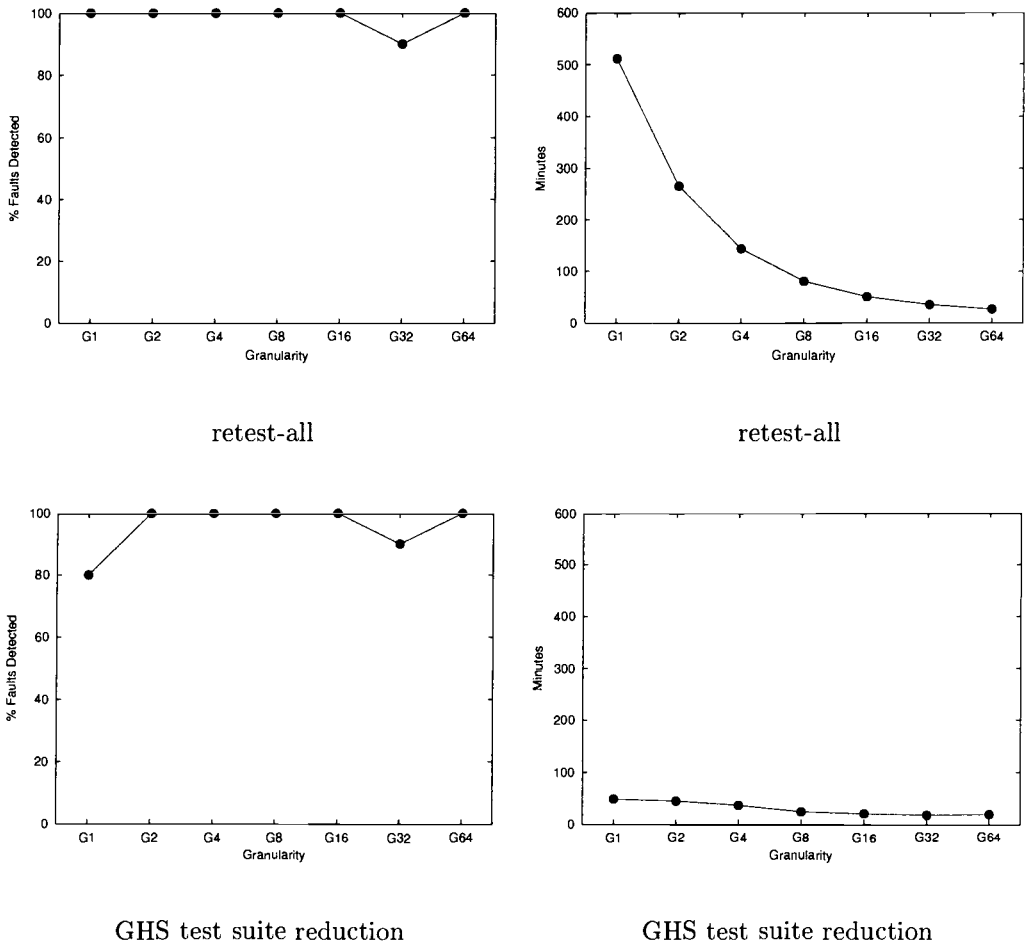


FIGURE A.42: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 3, for the functional grouping of test cases.

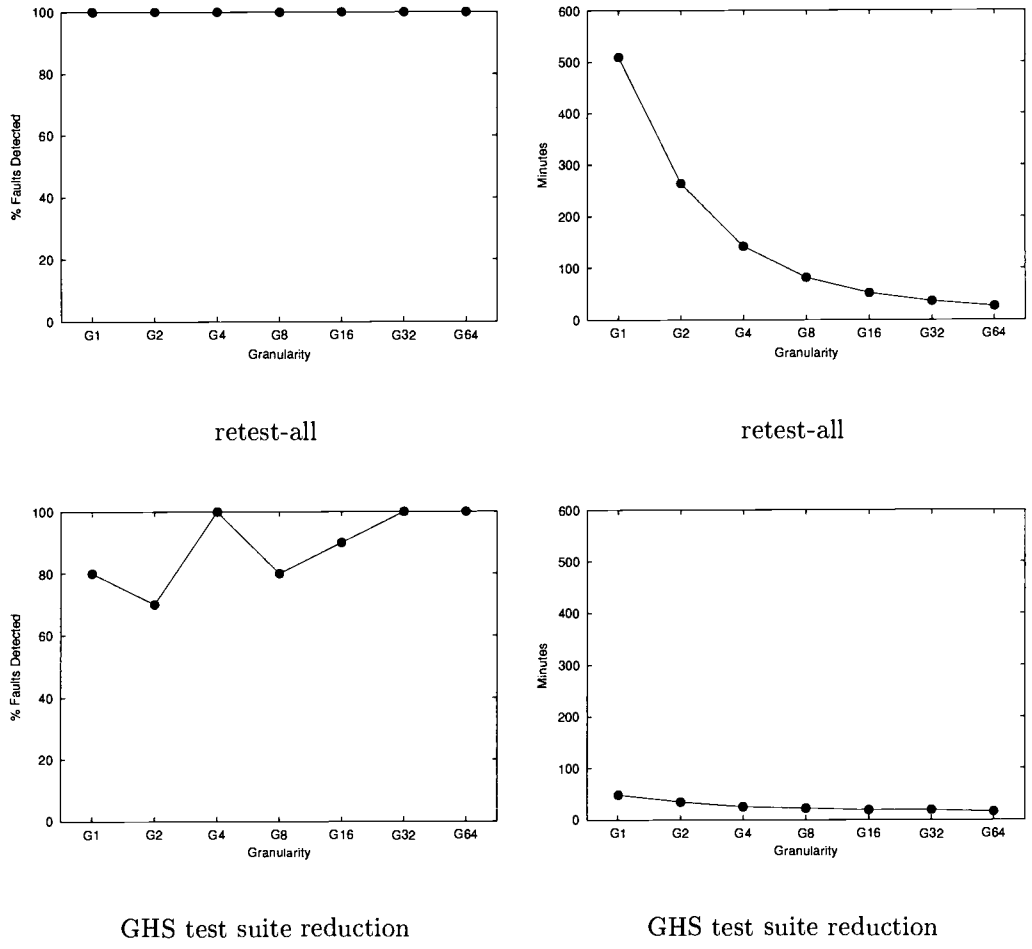


FIGURE A.43: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 4, for the random grouping of test cases.

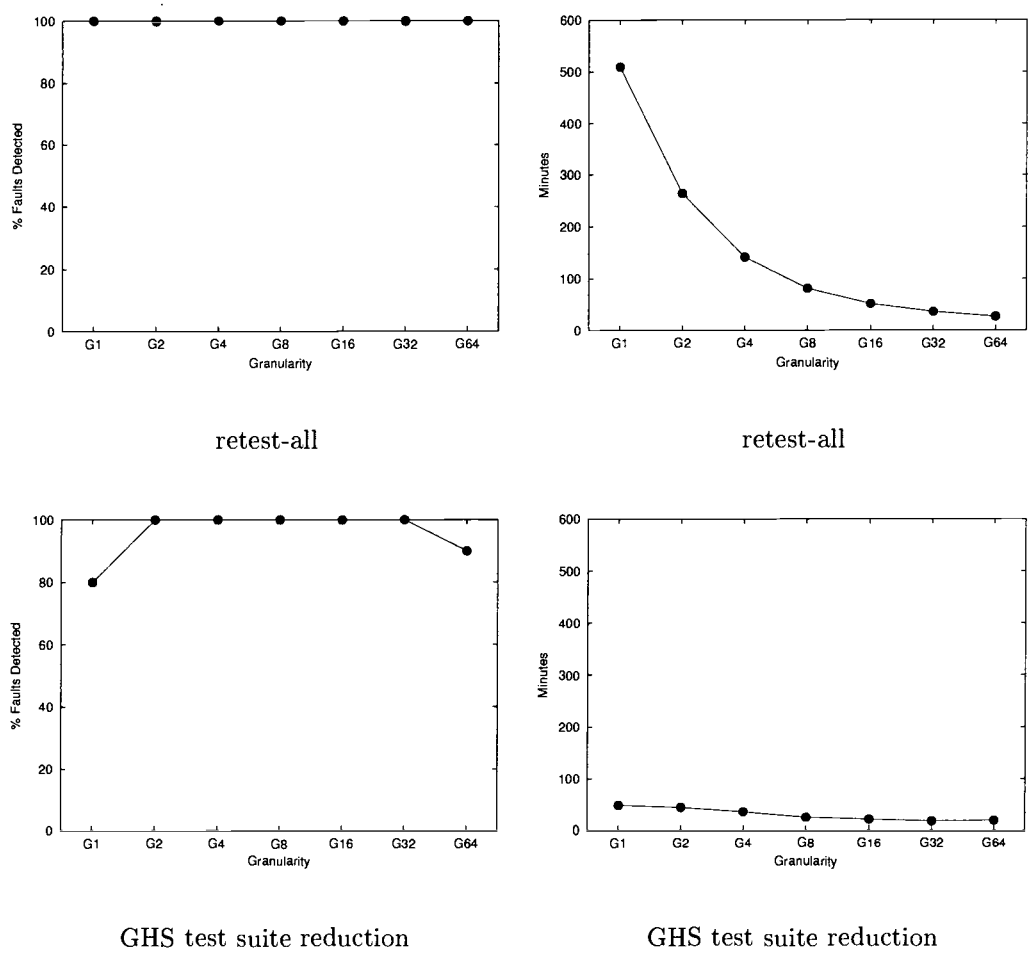
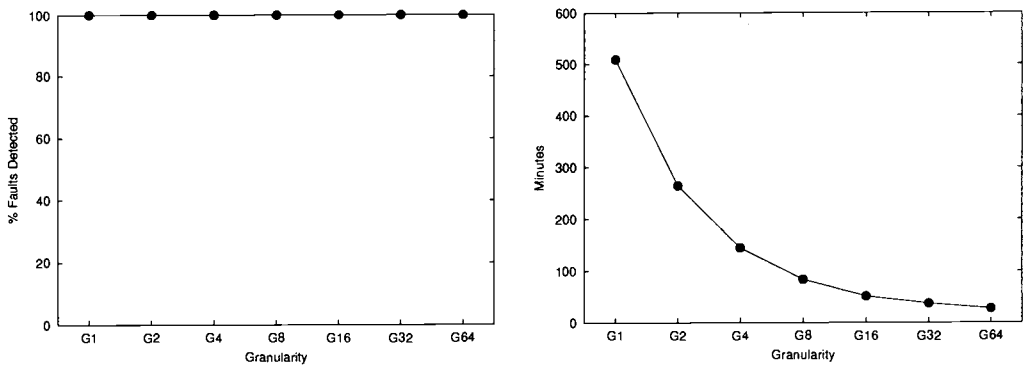
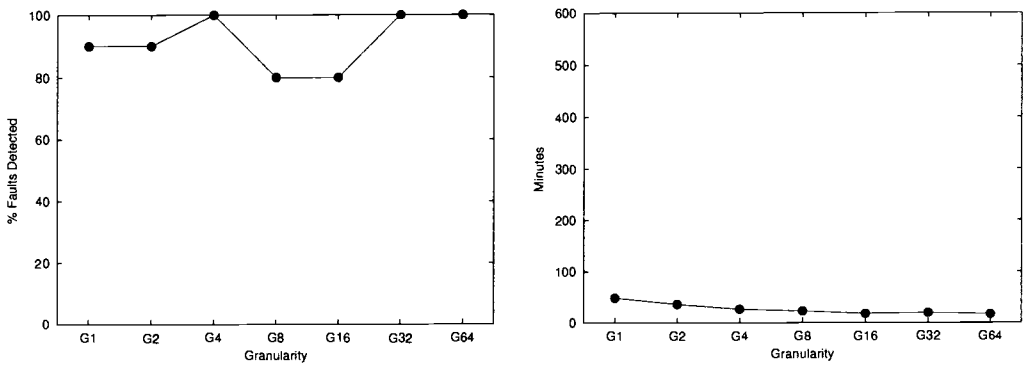


FIGURE A.44: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 4, for the functional grouping of test cases.



retest-all

retest-all



GHS test suite reduction

GHS test suite reduction

FIGURE A.45: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 5, for the random grouping of test cases.

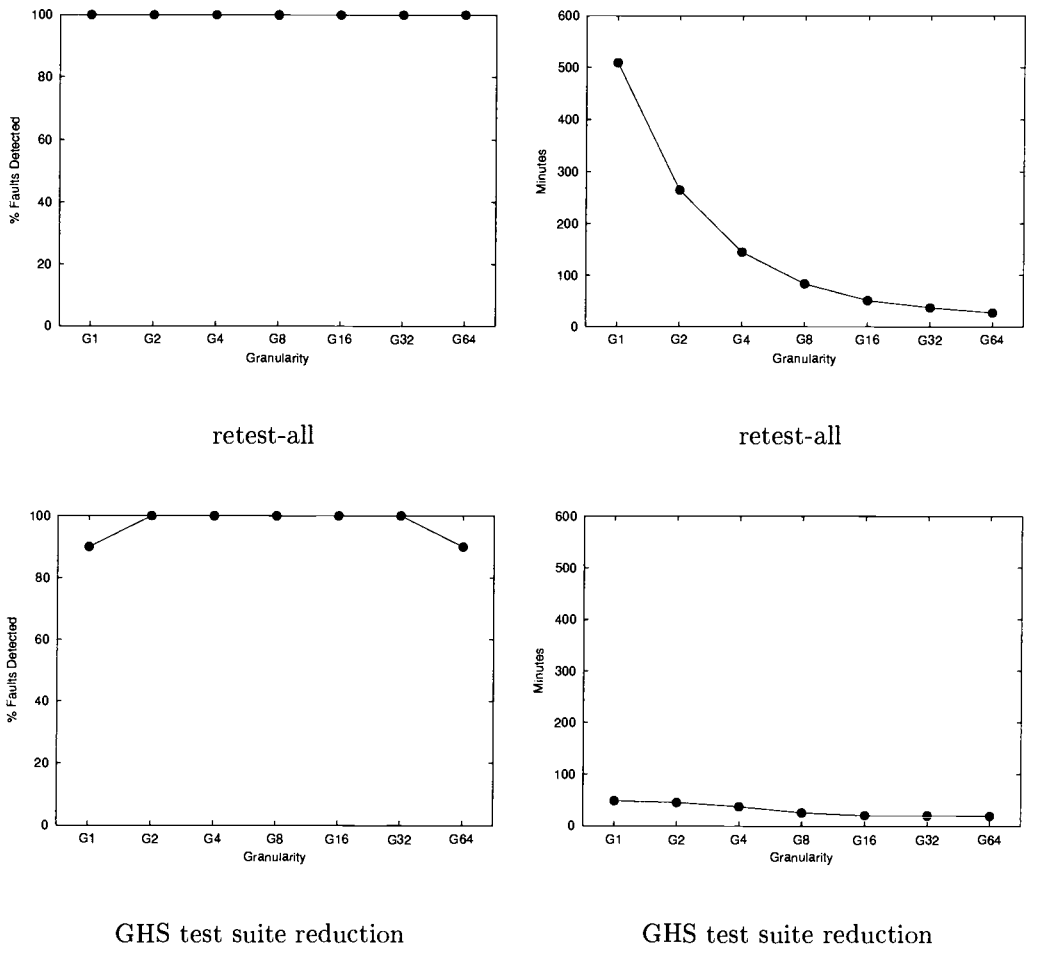


FIGURE A.46: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 5, for the functional grouping of test cases.

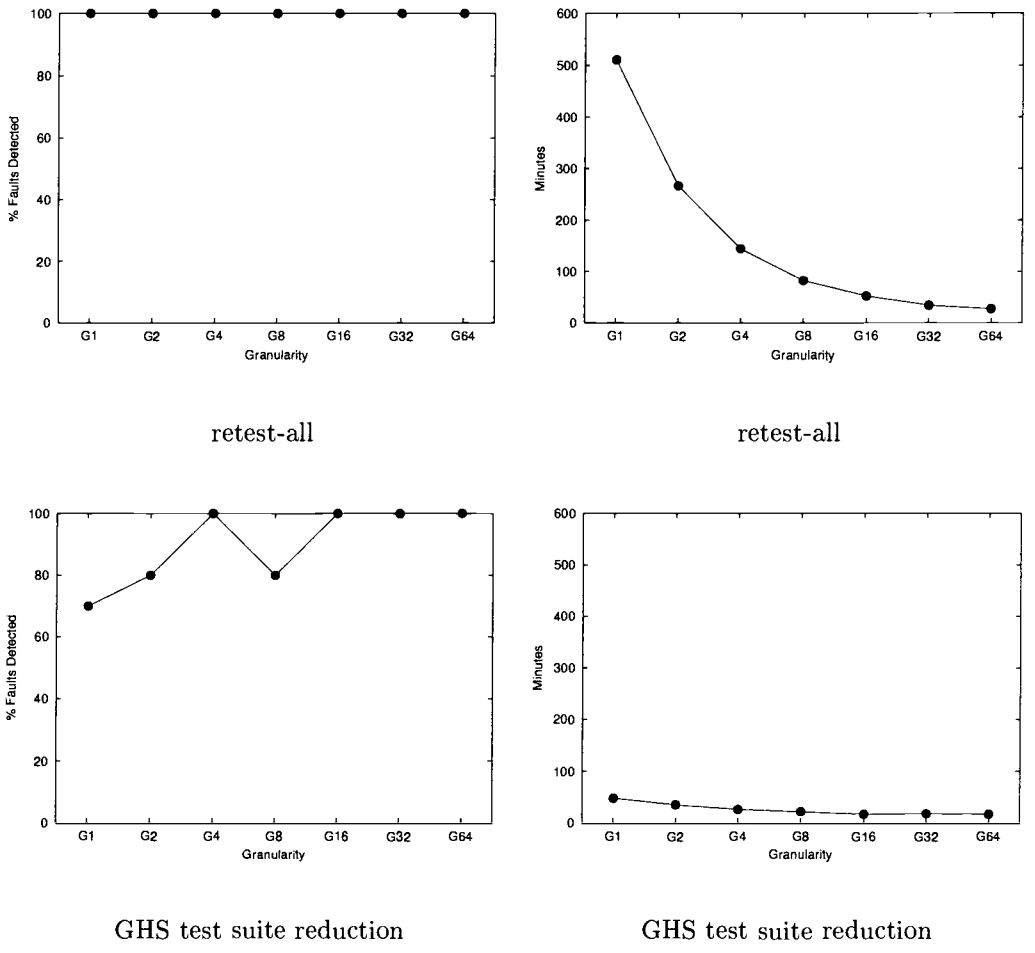


FIGURE A.47: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 6, for the random grouping of test cases.

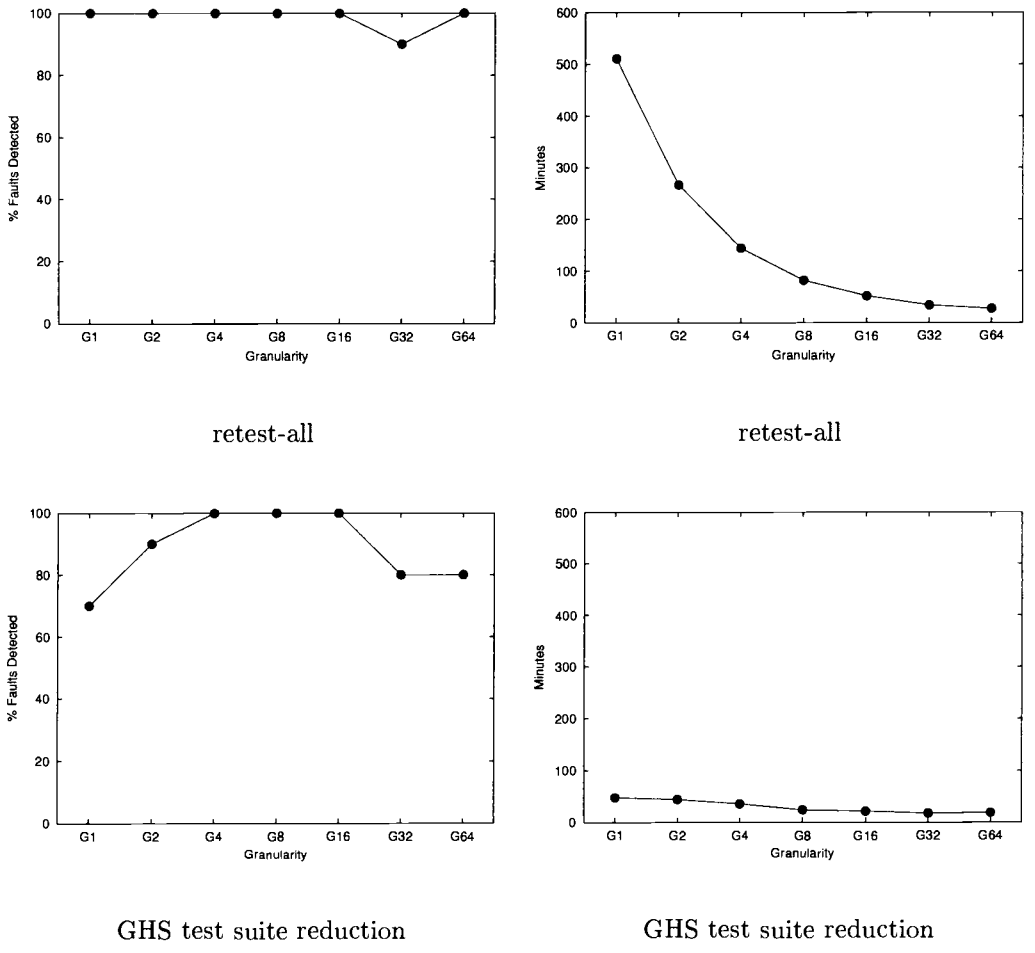


FIGURE A.48: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 6, for the functional grouping of test cases.

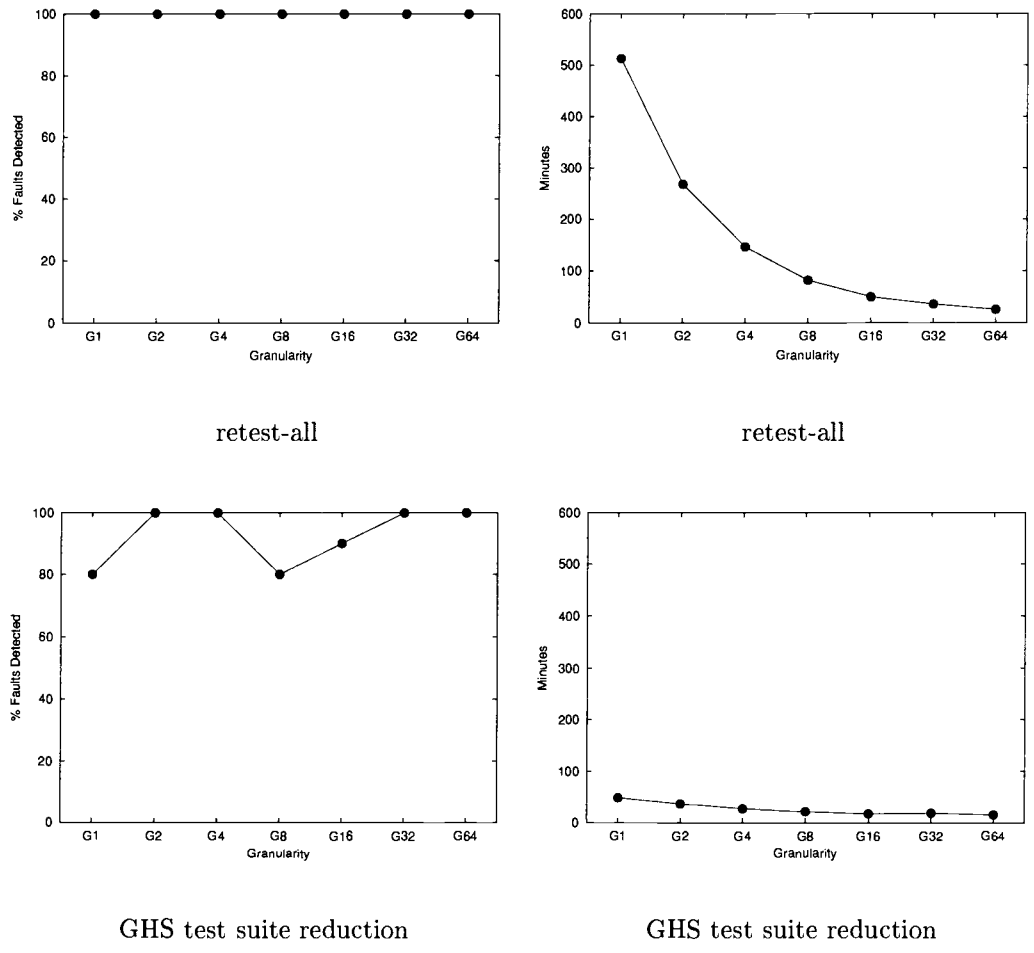


FIGURE A.49: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 7, for the random grouping of test cases.

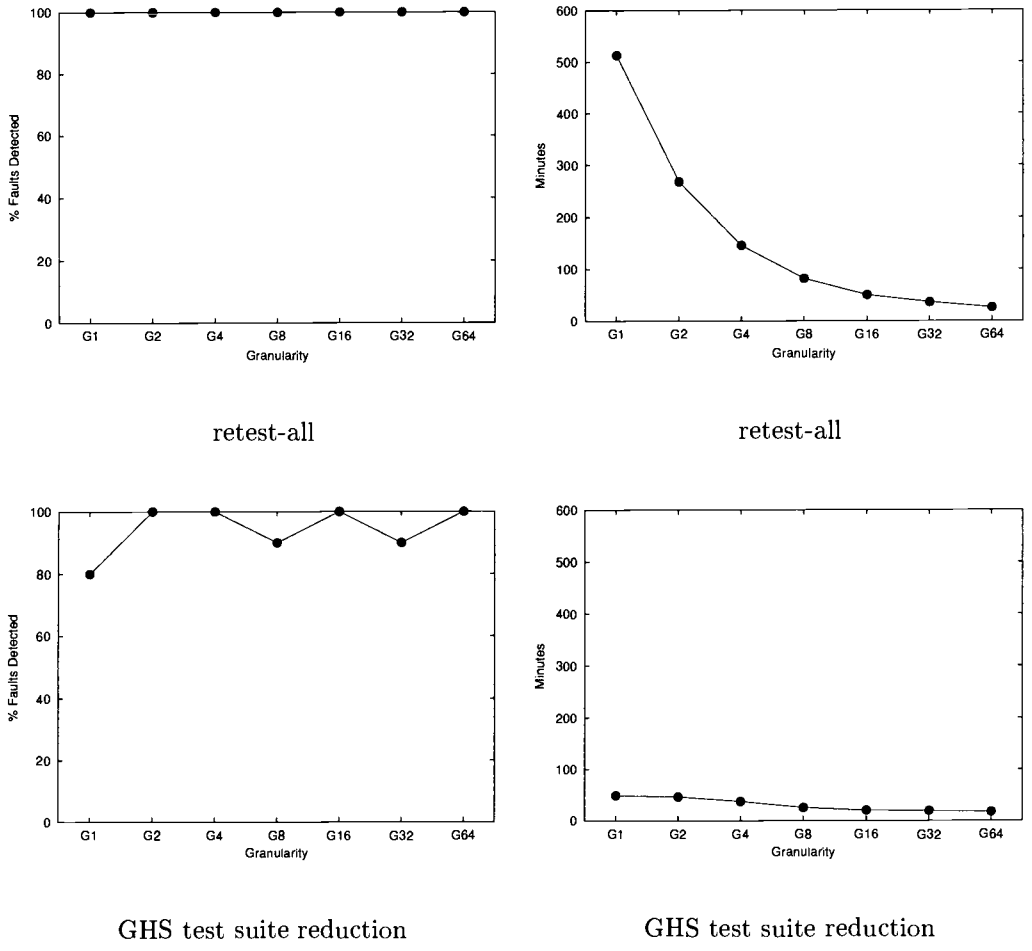


FIGURE A.50: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 7, for the functional grouping of test cases.

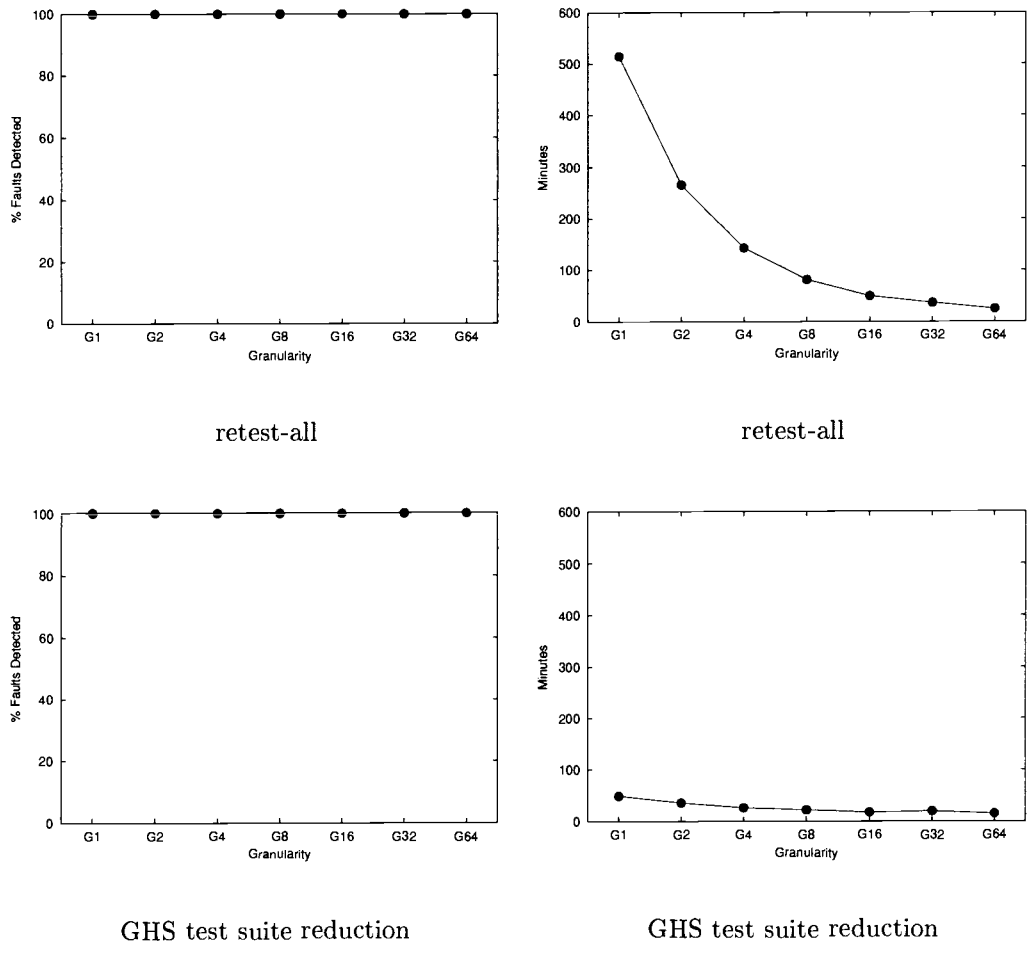


FIGURE A.51: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 8, for the random grouping of test cases.

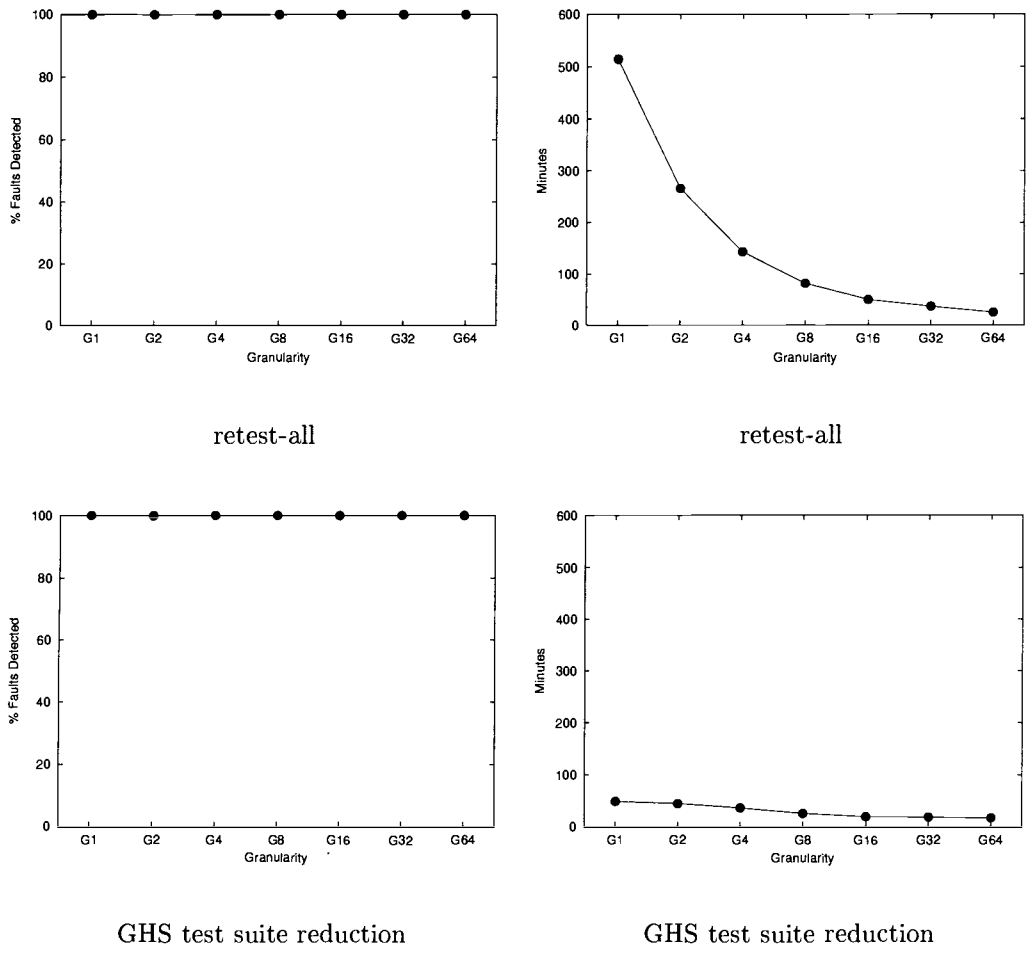


FIGURE A.52: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 8, for the functional grouping of test cases.

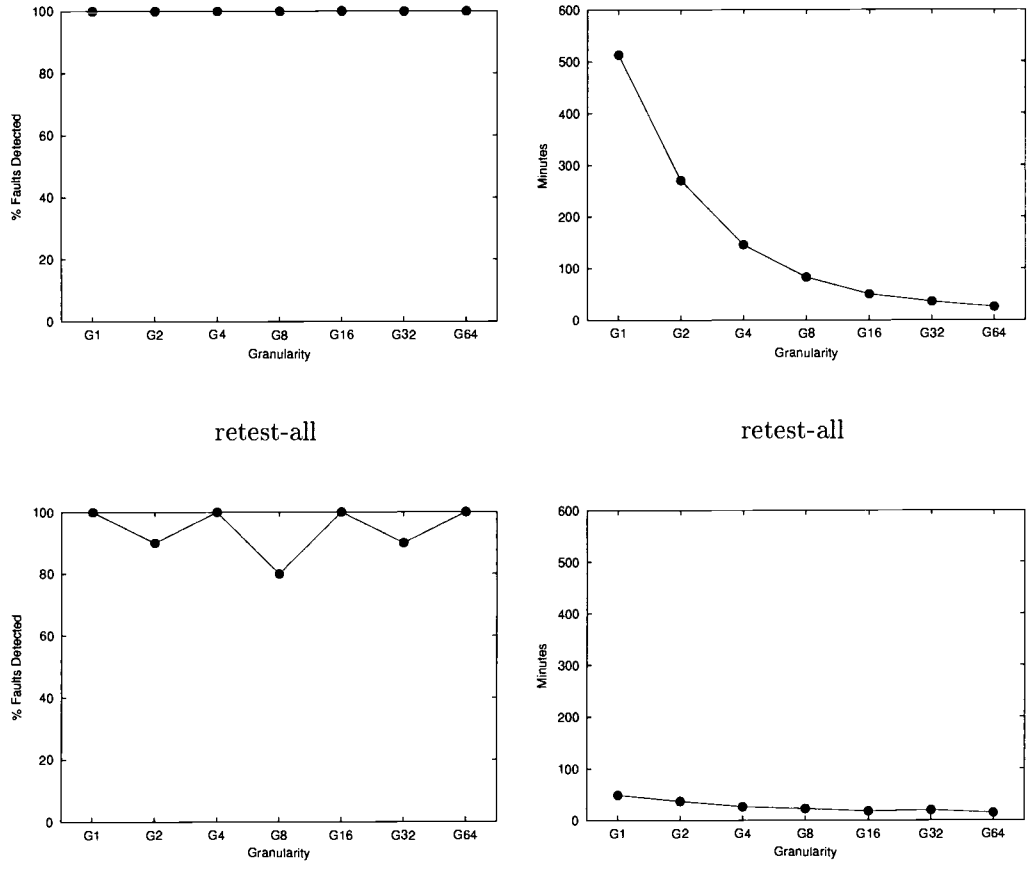


FIGURE A.53: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 9, for the random grouping of test cases.

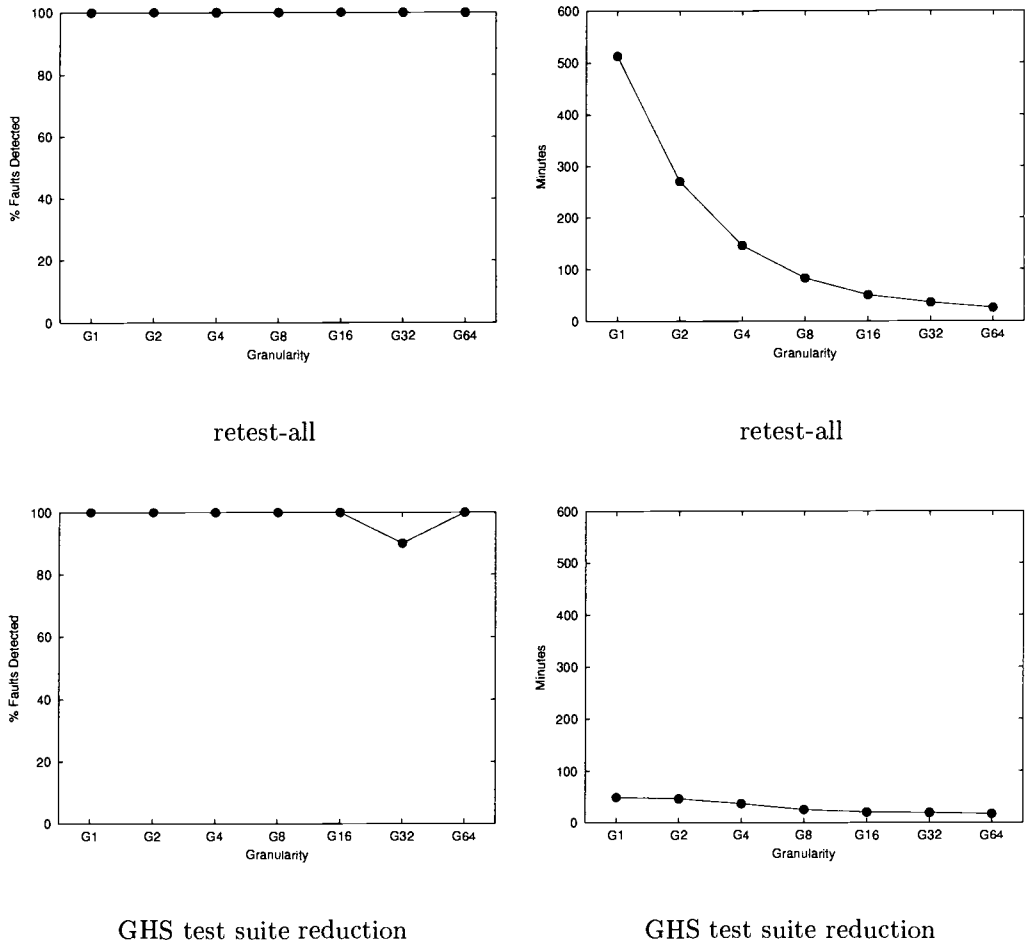


FIGURE A.54: Fault detection effectiveness (left) and test execution time (right) for test suite reduction techniques across test suite granularities, for version 9, for the functional grouping of test cases.

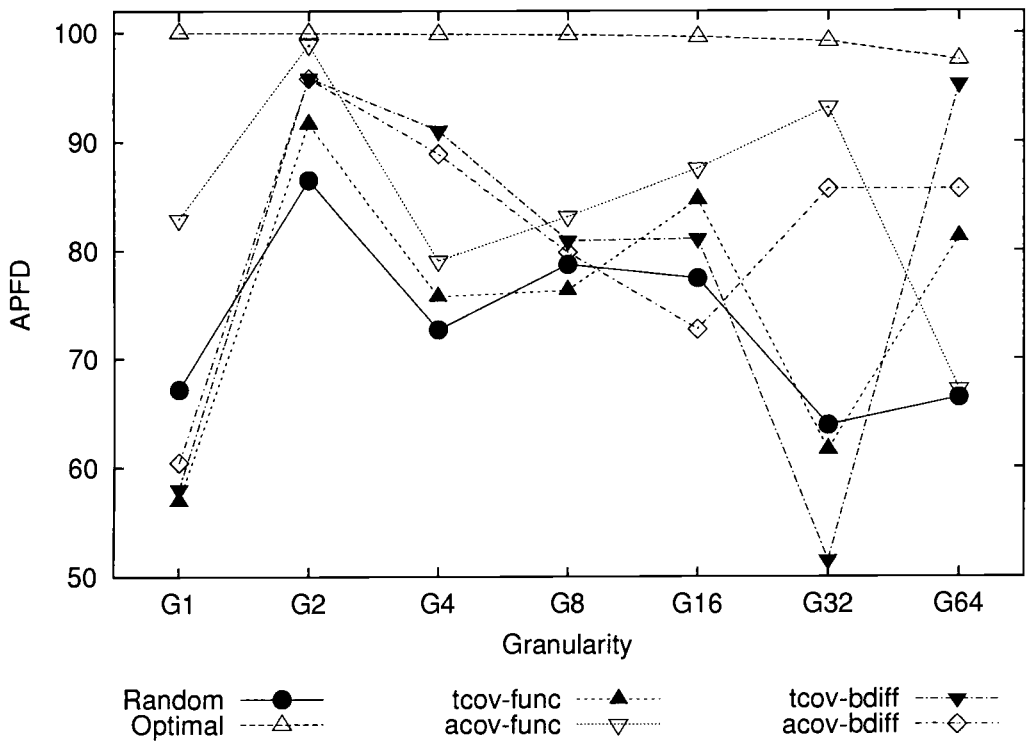


FIGURE A.55: APFD for all prioritization techniques together, for version 1, for the random grouping of test cases.

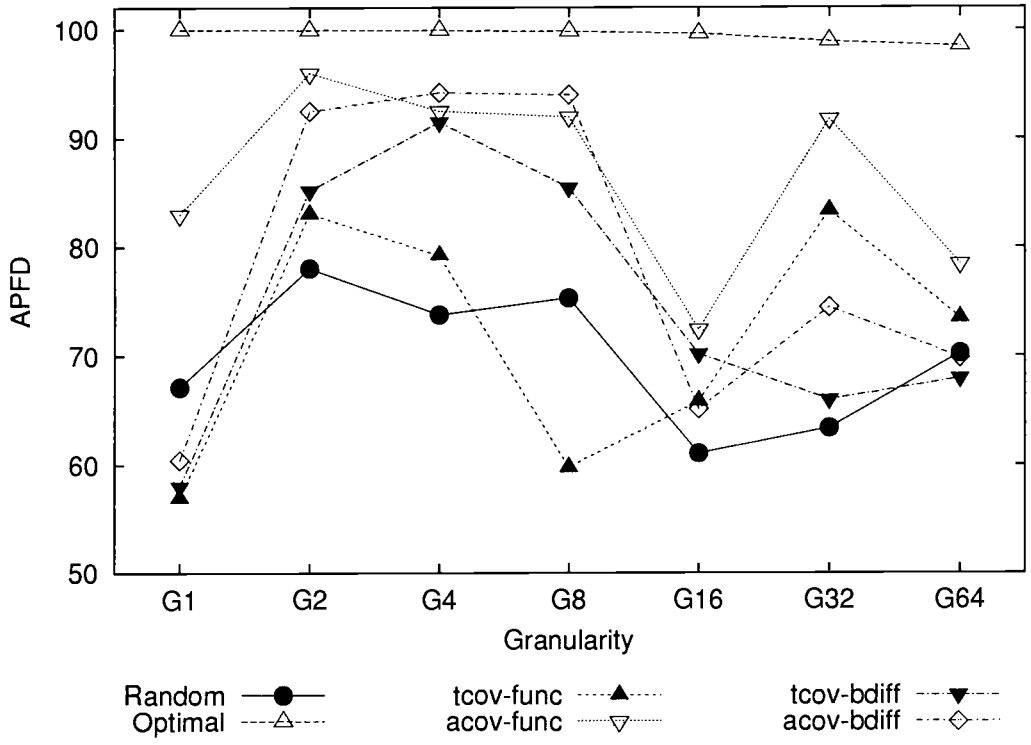


FIGURE A.56: APFD for all prioritization techniques together, for version 1, for the functional grouping of test cases.

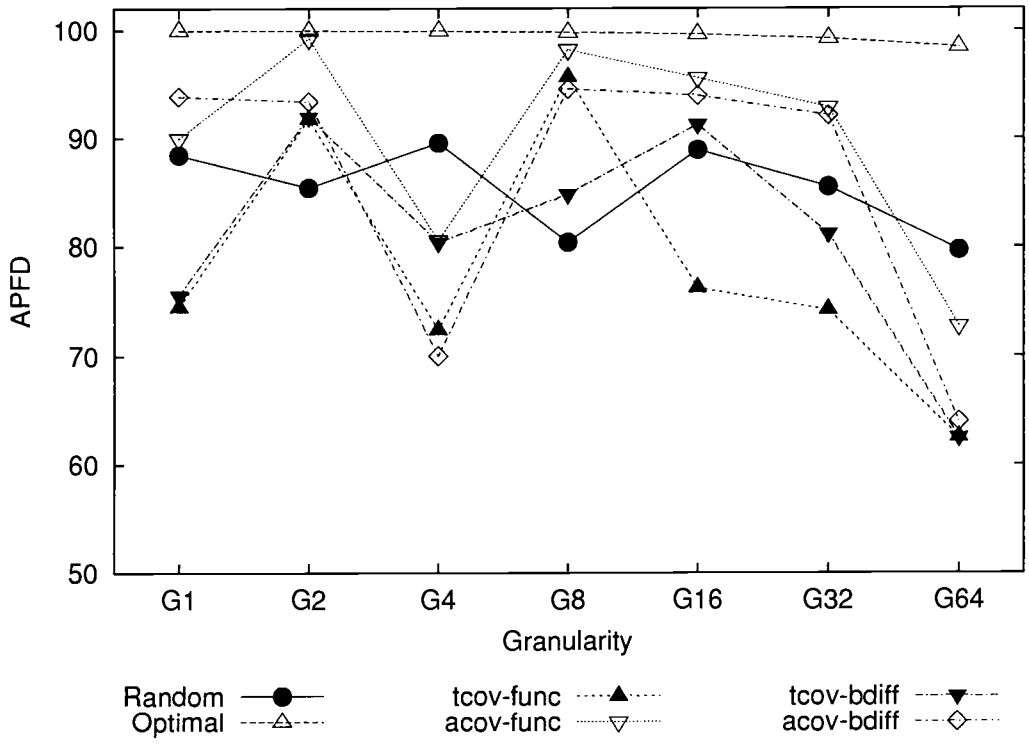


FIGURE A.57: APFD for all prioritization techniques together, for version 2, for the random grouping of test cases.

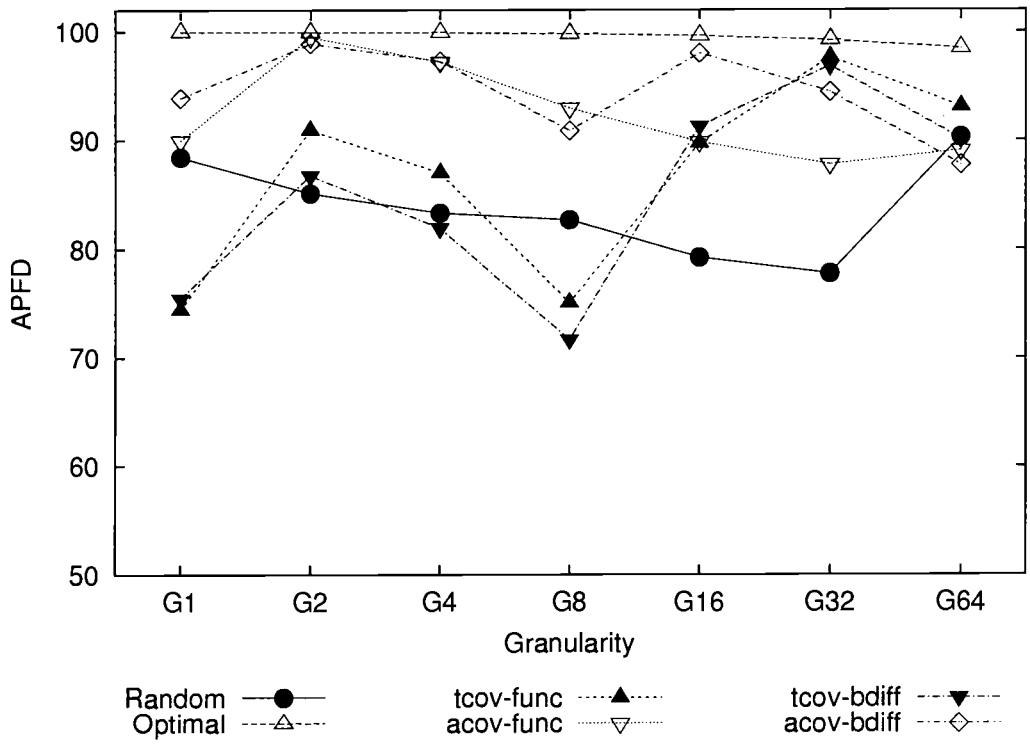


FIGURE A.58: APFD for all prioritization techniques together, for version 2, for the functional grouping of test cases.

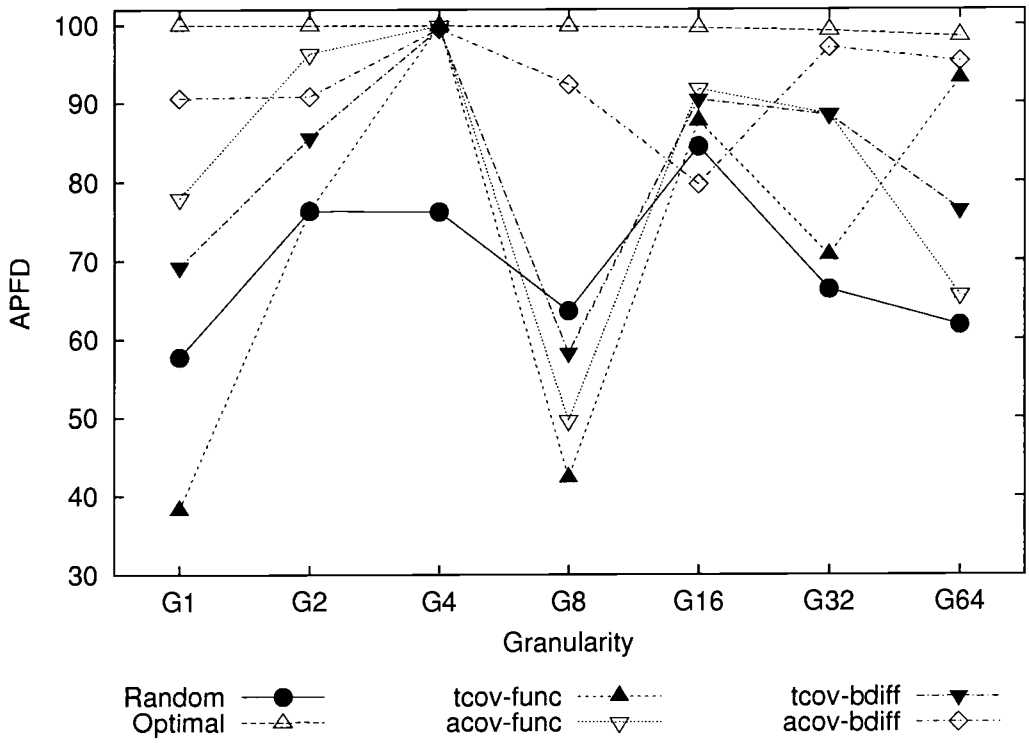


FIGURE A.59: APFD for all prioritization techniques together, for version 3, for the random grouping of test cases.

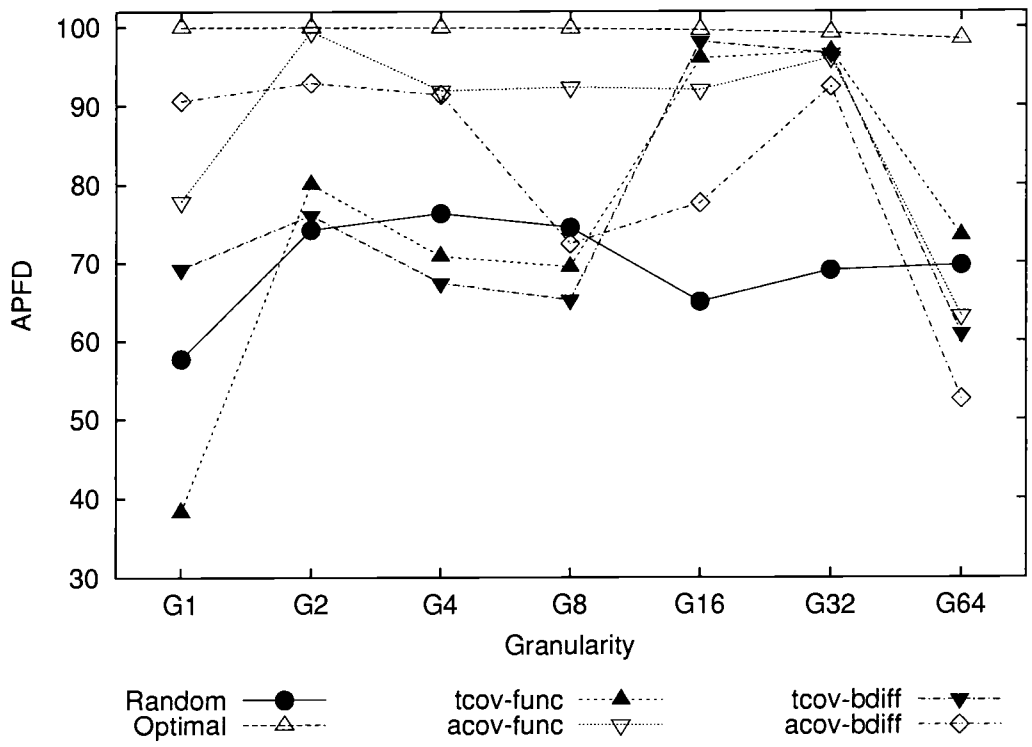


FIGURE A.60: APFD for all prioritization techniques together, for version 3, for the functional grouping of test cases.

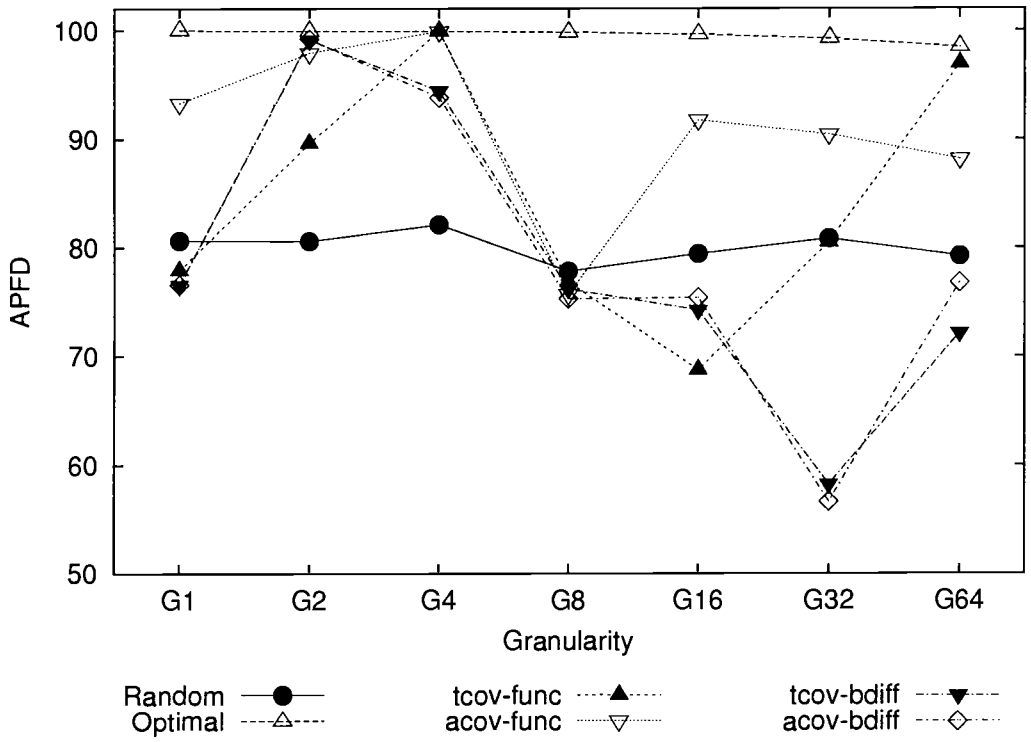


FIGURE A.61: APFD for all prioritization techniques together, for version 4, for the random grouping of test cases.

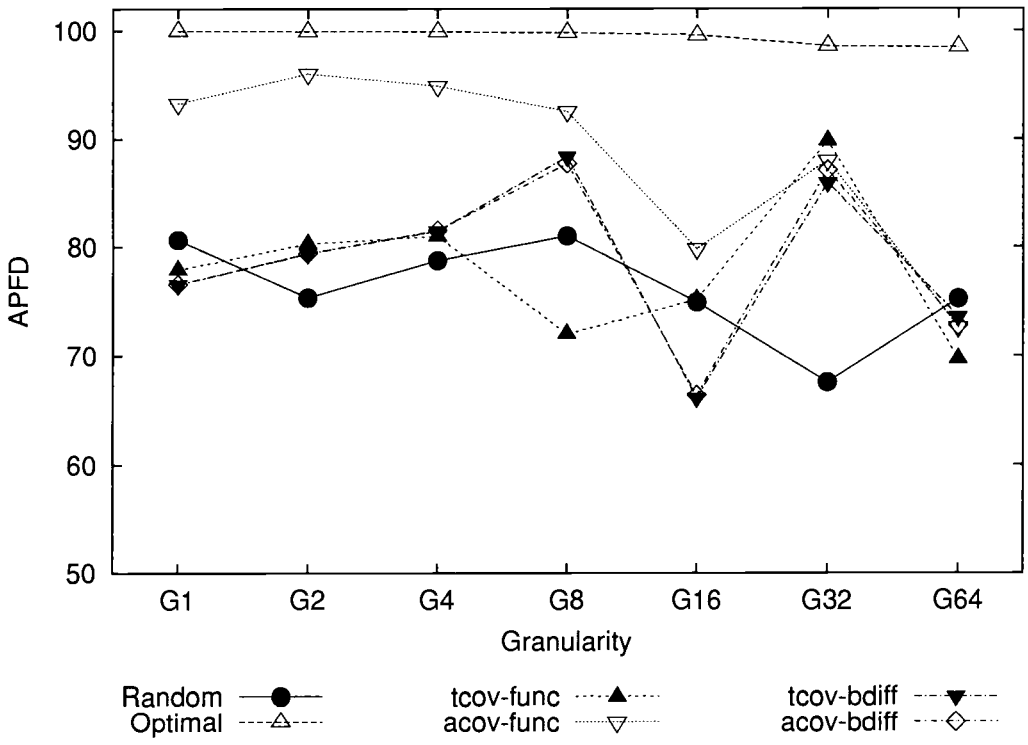


FIGURE A.62: APFD for all prioritization techniques together, for version 4, for the functional grouping of test cases.

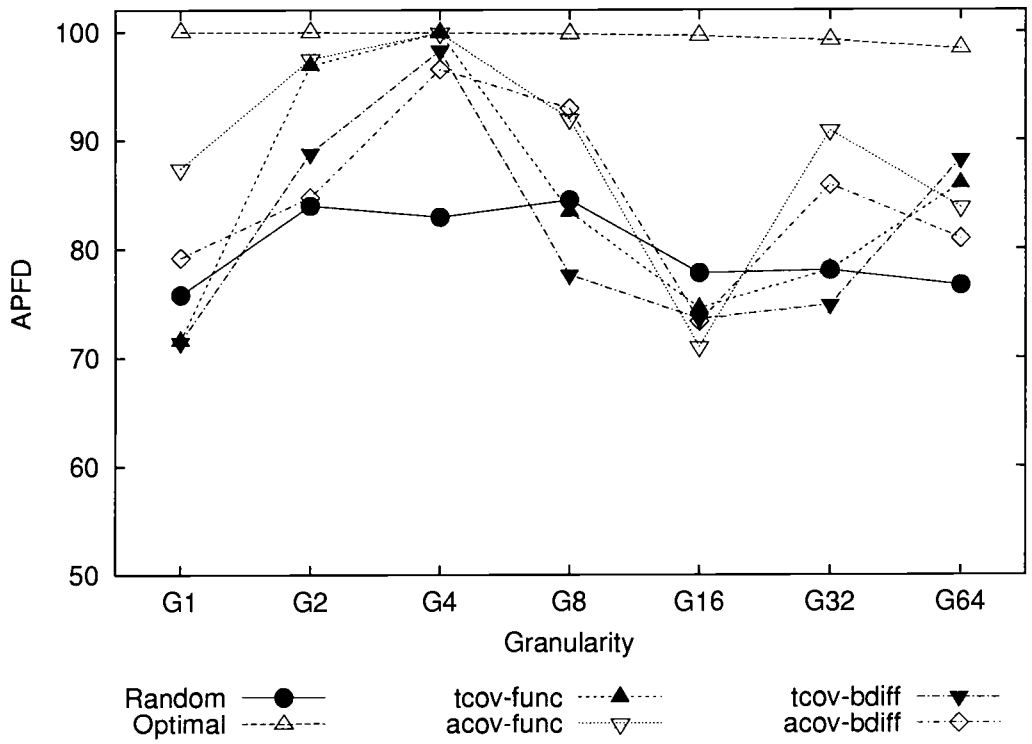


FIGURE A.63: APFD for all prioritization techniques together, for version 5, for the random grouping of test cases.

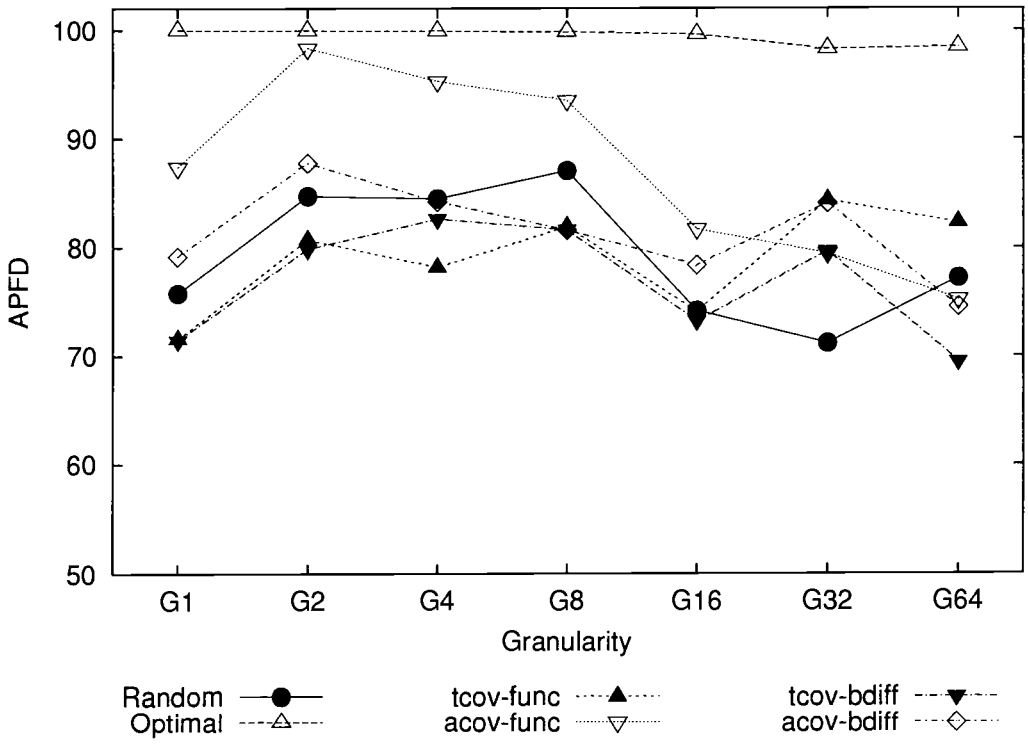


FIGURE A.64: APFD for all prioritization techniques together, for version 5, for the functional grouping of test cases.

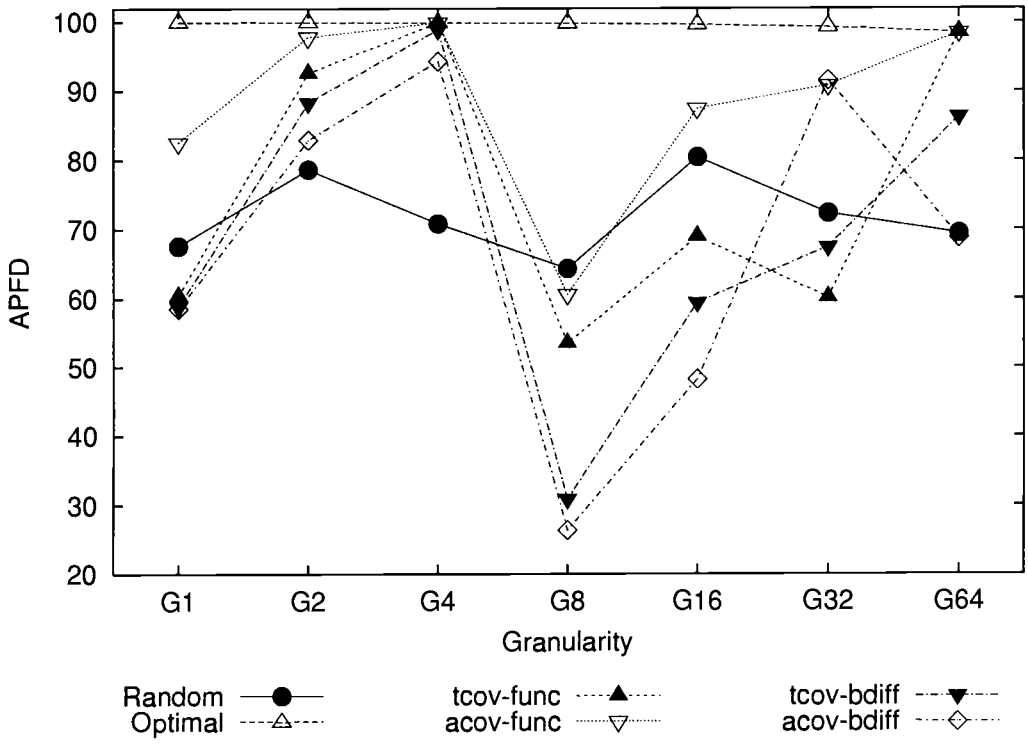


FIGURE A.65: APFD for all prioritization techniques together, for version 6, for the random grouping of test cases.

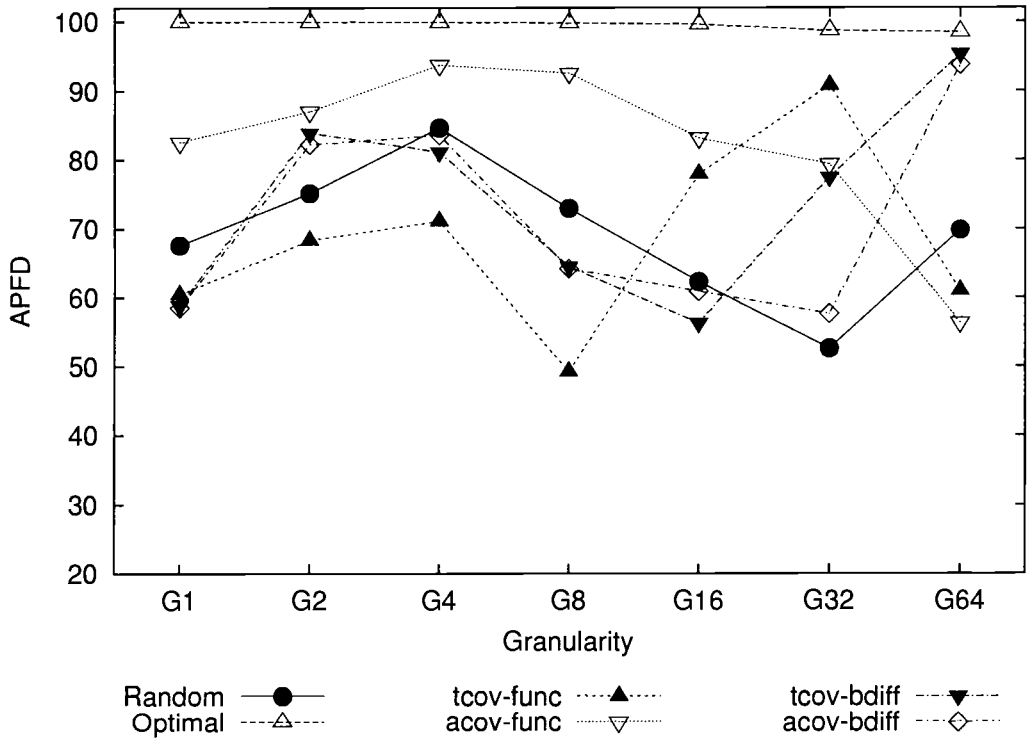


FIGURE A.66: APFD for all prioritization techniques together, for version 6, for the functional grouping of test cases.

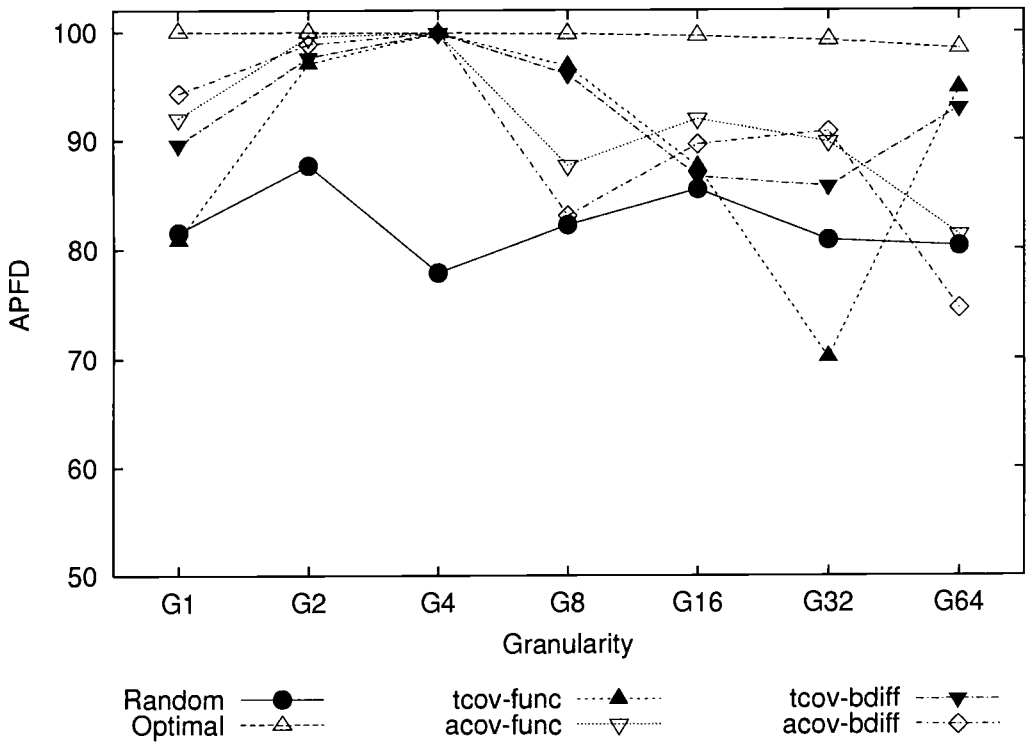


FIGURE A.67: APFD for all prioritization techniques together, for version 7, for the random grouping of test cases.

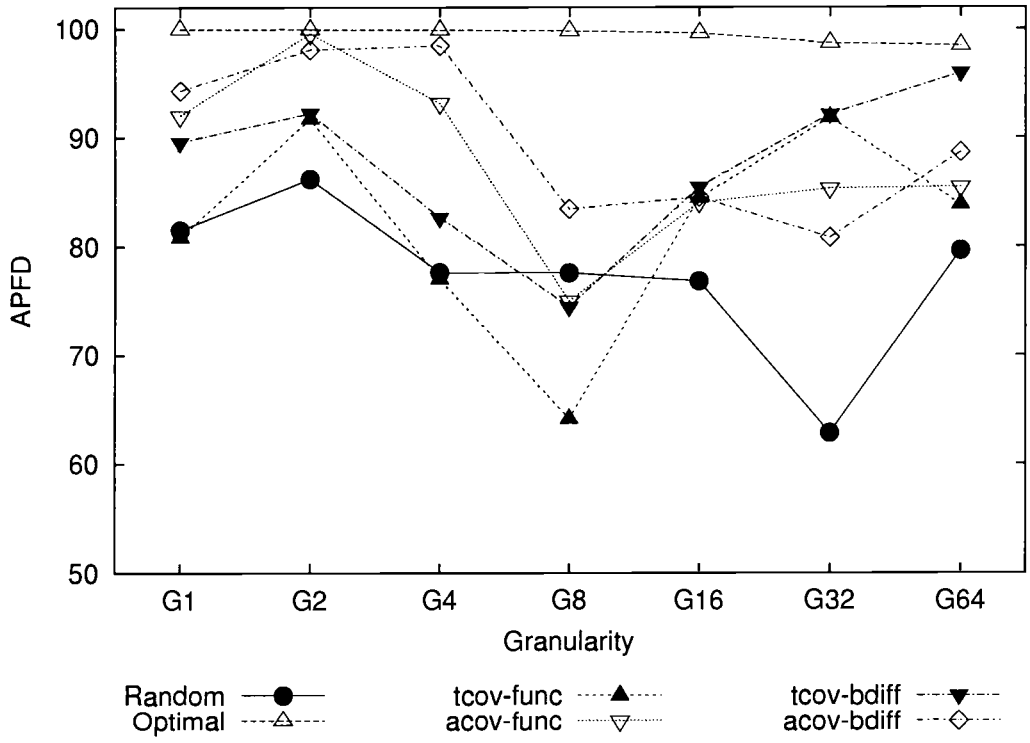


FIGURE A.68: APFD for all prioritization techniques together, for version 7, for the functional grouping of test cases.

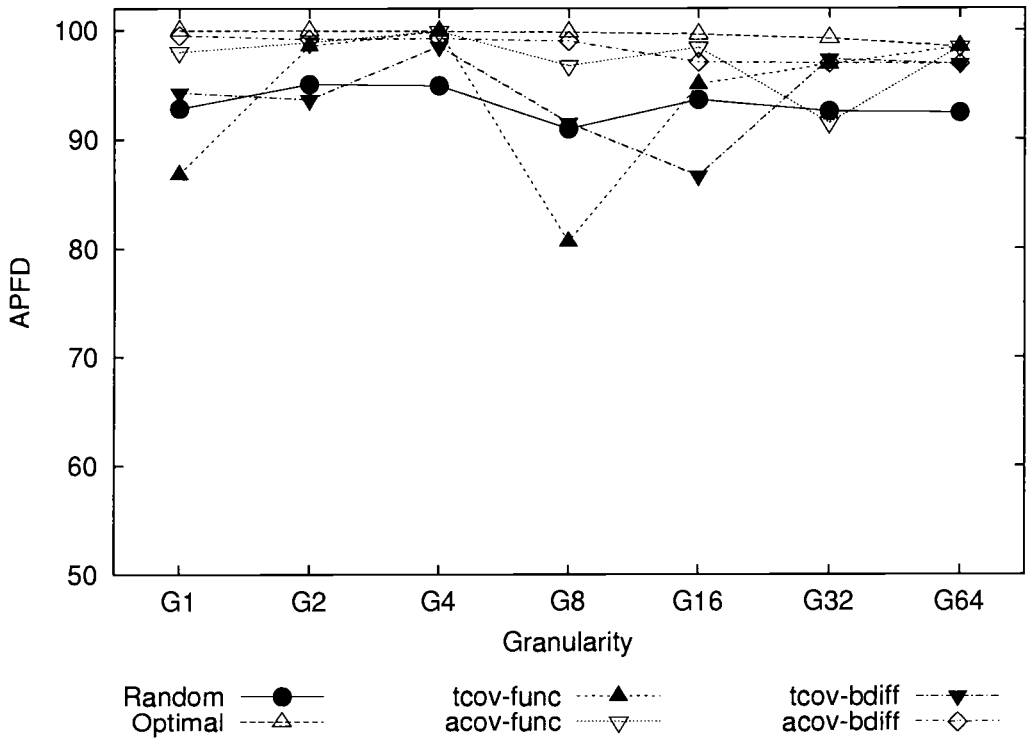


FIGURE A.69: APFD for all prioritization techniques together, for version 8, for the random grouping of test cases.

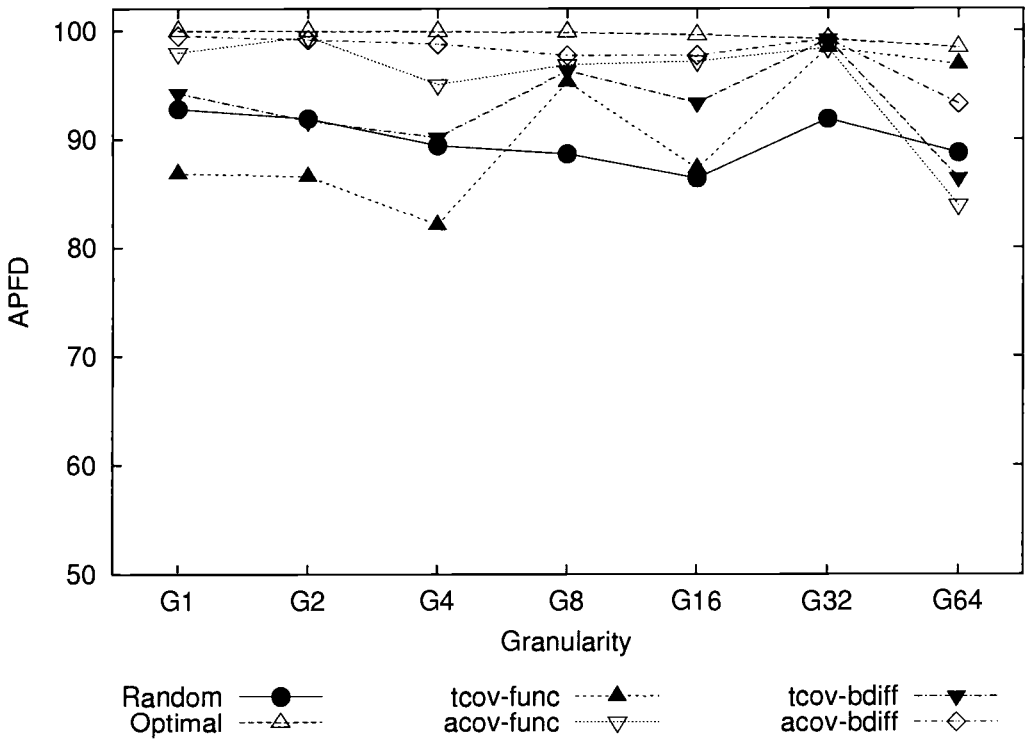


FIGURE A.70: APFD for all prioritization techniques together, for version 8, for the functional grouping of test cases.

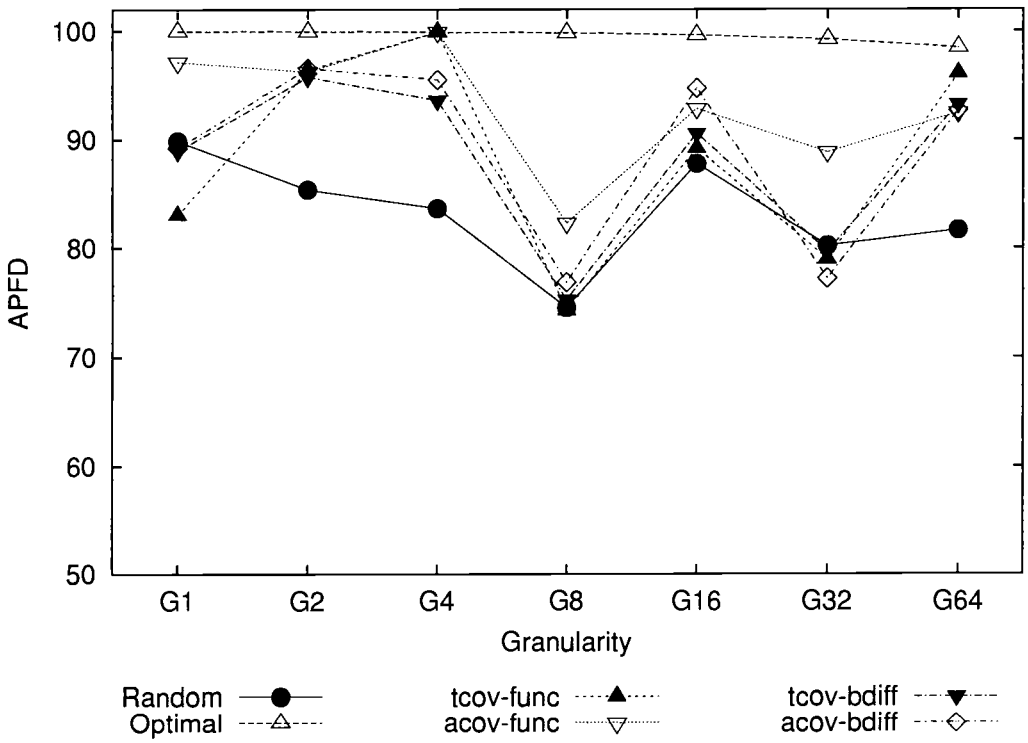


FIGURE A.71: APFD for all prioritization techniques together, for version 9, for the random grouping of test cases.

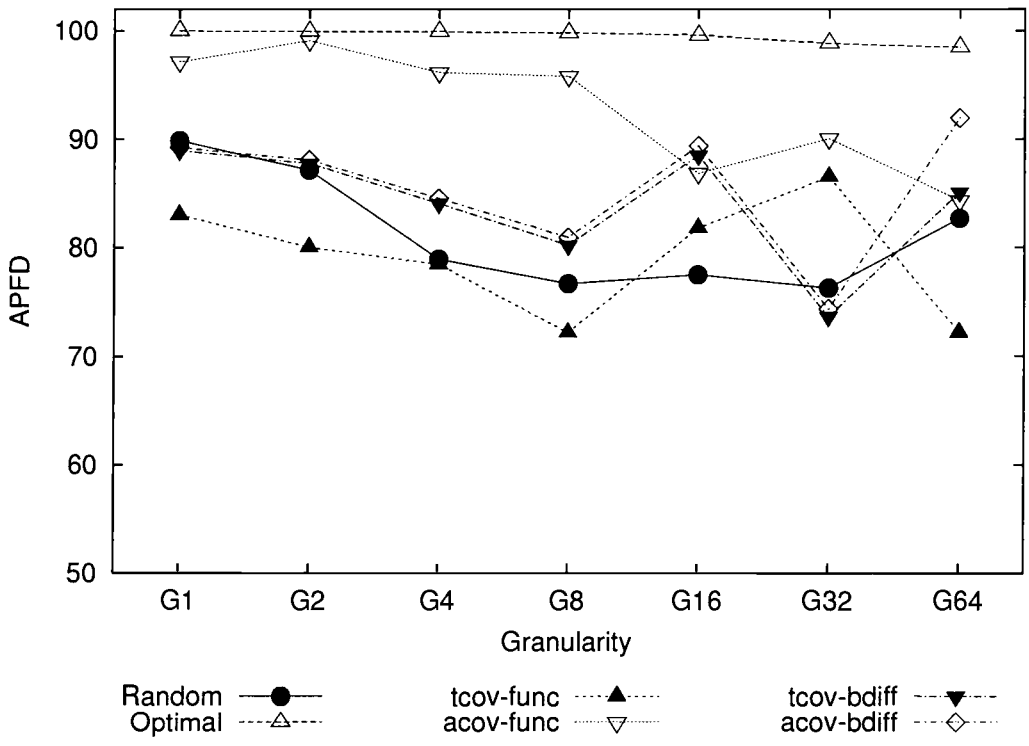


FIGURE A.72: APFD for all prioritization techniques together, for version 9, for the functional grouping of test cases.

Appendix B: Grouping Anovas for Selected Techniques

Grouping Techniques: Random and Functional					
Variable: *Percentage of faults detected.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	1922.22	6	320.37	4.99	0.00
Grouping Technique	96.03	1	96.03	1.50	0.22
Error	7576.19	118	64.20		
Total	9594.44	125			
Variable: *Execution time.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	2.10×10^9	6	3.50×10^8	17.53	0.00
Grouping Technique	1.23×10^7	1	1.23×10^7	0.62	0.43
Error	2.36×10^9	118	2.00×10^7		
Total	4.47×10^9	125			

TABLE B.1: Grouping Anovas for Modified Non-core Entity Technique.

Grouping Techniques: Random and Functional					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	893.23	6	148.87	2.02	0.07
Grouping Technique	191.38	1	191.38	2.60	0.11
Error	8699.61	118	73.72		
Total	9784.22	125			

TABLE B.2: Grouping Anovas for Random Prioritization Technique.

Grouping Techniques: Random and Functional					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	38.80	6	6.47	333.57	0.00
Grouping Technique	0.045	1	0.045	2.32	0.13
Error	2.29	118	0.019		
Total	41.13	125			

TABLE B.3: Grouping Anovas for Optimal Prioritization Technique.

Grouping Techniques: Random and Functional					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	4688.25	6	781.38	5.23	0.00
Grouping Technique	404.66	1	404.66	2.71	0.10
Error	17641.4	118	149.50		
Total	22734.3	125			

TABLE B.4: Grouping Anovas for tcov-func Prioritization Technique.

Grouping Techniques: Random and Functional					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	3679.21	6	613.20	8.65	0.00
Grouping Technique	13.15	1	13.15	0.19	0.67
Error	8362.66	118	70.87		
Total	12055.0	125			

TABLE B.5: Grouping Anovas for acov-func Prioritization Technique.

Grouping Techniques: Random and Functional					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	2935.01	6	489.17	3.37	0.00
Grouping Technique	107.30	1	107.30	0.74	0.39
Error	17103.4	118	144.94		
Total	20145.7	125			

TABLE B.6: Grouping Anovas for tcov-bdiff Prioritization Technique.

Grouping Techniques: Random and Functional					
Variable: *APFD.*					
Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	p-Value
Granularity	2754.74	6	459.12	2.86	0.01
Grouping Technique	28.48	1	28.48	0.18	0.67
Error	18917.9	118	160.32		
Total	21701.1	125			

TABLE B.7: Grouping Anovas for acov-bdiff Prioritization Technique.