2-2002

Test Case Prioritization: A Family of Empirical Studies

Sebastian Elbaum
University of Nebraska-Lincoln, selbaum@virginia.edu

Alexey G. Malishevsky
Oregon State University

Gregg Rothermel
University of Nebraska-Lincoln, gerother@ncsu.edu

Test Case Prioritization: A Family of Empirical Studies

Sebastian Elbaum, *Member*, *IEEE*, Alexey G. Malishevsky, *Student Member*, *IEEE*, and Gregg Rothermel, *Member*, *IEEE*

Abstract—To reduce the cost of regression testing, software testers may prioritize their test cases so that those which are more important, by some measure, are run earlier in the regression testing process. One potential goal of such prioritization is to increase a test suite's *rate of fault detection*. Previous work reported results of studies that showed that prioritization techniques can significantly improve rate of fault detection. Those studies, however, raised several additional questions: 1) Can prioritization techniques be effective when targeted at specific modified versions; 2) what trade-offs exist between fine granularity and coarse granularity prioritization techniques; 3) can the incorporation of measures of fault proneness into prioritization techniques improve their effectiveness? To address these questions, we have performed several new studies in which we empirically compared prioritization techniques using both controlled experiments and case studies. The results of these studies show that each of the prioritization techniques considered can improve the rate of fault detection of test suites overall. Fine-granularity techniques typically outperformed coarse-granularity techniques, but only by a relatively small margin overall; in other words, the relative imprecision in coarse-granularity analysis did not dramatically reduce coarse-granularity techniques' ability to improve rate of fault detection. Incorporation of fault-proneness techniques produced relatively small improvements over other techniques in terms of rate of fault detection, a result which ran contrary to our expectations. Our studies also show that the relative effectiveness of various techniques can vary significantly across target programs. Furthermore, our analysis shows that whether the effectiveness differences observed will result in savings in practice varies substantially with the cost factors associated with particular testing processes. Further work to understand the sources of this variance and to incorporate such understanding into prioritization techniques and the choice of techniques would be beneficial.

Index Terms—Test case prioritization, regression testing, empirical studies.

1 Introduction

Regression testing is an expensive testing process used to validate modified software and detect whether new faults have been introduced into previously tested code. Regression test suites can be expensive to execute in full; thus, test engineers may prioritize their regression tests such that those which are more important, by some measure, are run earlier in the regression testing process.

One potential goal of test case prioritization is that of increasing a test suite's *rate of fault detection*—a measure of how quickly a test suite detects faults during the testing process. An improved rate of fault detection can provide earlier feedback on the system under test, enable earlier debugging, and increase the likelihood that, if testing is prematurely halted, those test cases that offer the greatest fault detection ability in the available testing time will have been executed.

In previous work [30], Rothermel et al. formally defined the test case prioritization problem, presented several techniques for prioritizing test cases, and presented the results of empirical studies in which those techniques were applied to various programs. Six prioritization techniques were studied; all were based on coverage of statements or branches in the programs. The test suites produced by these techniques were compared to random, untreated, and optimal test case orders. The studies showed that the techniques improved rate of fault detection and that this improvement occurred even for the least sophisticated (and least expensive) of those techniques.

Building on that work, this article addresses several additional questions. First, [30] examined only "general prioritization," which attempts to select a test case order that will be effective *on average* over a succession of subsequent versions of the software. In regression testing, we are concerned with a particular version of the software and we may wish to prioritize test cases in a manner that will be most effective for that version. We call this "version-specific prioritization" and we are interested in its effectiveness. Although, in many cases, the same techniques may apply to version-specific as to general prioritization, the cost-effectiveness of such techniques with respect to the two forms of prioritization could differ. Thus, in this article, we focus on version-specific prioritization.

Second, the techniques examined in [30] all operated at relatively *fine granularity*—performing instrumentation, analysis, and prioritization at the level of source code statements. For large software systems, or systems in which statement-level instrumentation is not feasible, such techniques may be too expensive. An alternative is to operate at

- *S. Elbaum is with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588-0115. E-mail: elbaum@cse.unl.edu.*
- *A.G. Malishevsky and G. Rothermel are with Department of Computer Science, Oregon State University, Corvallis, OR 97331. E-mail: {malishal, grother}@cs.orst.edu.*

Manuscript received 26 Feb. 2001; accepted 8 Oct. 2001. Recommended for acceptance by A. Bertolino. For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 115155.

a *coarser granularity*—for example, at the function level, where instrumentation and analysis are more efficient. We expect, however, that coarse granularity techniques will be less effective than fine granularity techniques and loss of effectiveness could offset efficiency gains. We wish to examine the cost-benefits trade-offs that hold, for test case prioritization, across granularities. Thus, in this work, we consider four techniques examined in [30], plus 12 new techniques that operate at the function level.

Third, the analysis in [30] revealed a sizable performance gap between the results achieved by the prioritization techniques that we examined and the optimal results achievable. We wish to at least partially bridge this gap and we conjecture that incorporating measures of fault proneness (e.g., [10], [26]) into our techniques might let us do so. Thus, this work involves several techniques that incorporate such measures.

Finally, the empirical studies in [30] considered only eight relatively small programs. In this work, our initial studies involve controlled experiments on these same programs; however, we then extend our focus to include case studies of three larger programs: two open-source Unix utilities and an embedded real-time subsystem of a level-5 RAID storage system, each with a sequence of released versions. Together, this group of varied studies and programs lets us observe the performance of several prioritization techniques in different situations and lets us probe the relative strengths of each technique.

In the next section of this article, we present background material on the test case prioritization problem. Section 3 describes the test case prioritization techniques that we study. Section 4 describes our research questions and overall empirical approach. Section 5 presents our controlled experiments and Section 6 presents our case studies. Section 7 presents an analysis of the practical significance of our results. Section 8 reviews related work and Section 9 summarizes our results and discusses future research.

2 Background: The Test Case Prioritization Problem

Rothermel et al. [30] define the test case prioritization problem and describe several issues relevant to its solution; this section reviews the portions of that material that are necessary to understand this article.

The test case prioritization problem is defined as follows:

The Test Case Prioritization Problem:

Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that $(\forall T'')\ (T'' \in PT)\ (T'' \neq T')\ [f(T') \geq f(T'')]$.

Here, PT represents the set of all possible prioritizations (orderings) of T and f is a function that, applied to any such ordering, yields an *award value* for that ordering.

There are many possible goals for prioritization; [30] describes several. This article, like [30], focuses on the goal of increasing the likelihood of revealing faults earlier in the testing process. This goal can be described, informally, as one of improving a test suite's *rate of fault detection*: A quantitative measure for this goal is provided in Section 4.1.

Rothermel et al. [30] distinguish two types of test case prioritization: general and version-specific. In *general test case prioritization*, given program P and test suite T, test cases in T are prioritized with the goal of finding a test case order that will be useful over a sequence of subsequent modified versions of P. Thus, general test case prioritization can be performed following the release of some version of the program during off-peak hours and the cost of performing the prioritization is amortized over the subsequent releases. The expectation is that the resulting prioritized suite will be more successful than the original suite at meeting the goal of the prioritization, *on average* over those subsequent releases.

In contrast, in *version-specific test case prioritization*, given program P and test suite T, test cases in T are prioritized with the intent of finding an ordering that will be useful on a specific version P' of P. Version-specific prioritization is performed after a set of changes have been made to P and prior to regression testing P'. Because this prioritization is performed after P' is available, care must be taken to prevent the cost of prioritizing from excessively delaying the very regression testing activities it is supposed to facilitate. The prioritized test suite may be more effective at meeting the goal of the prioritization for P' in particular than would a test suite resulting from general test case prioritization, but may be less effective on average over a succession of subsequent releases.

Finally, like [30], this article addresses the problem of prioritizing test cases for regression testing; however, test case prioritization can also be employed in the initial testing of software (see Section 8). An important difference between these two applications is that, in the case of regression testing, prioritization techniques can use information from previous runs of regression test suites to prioritize the test cases for subsequent runs; such information is not available during initial testing.

3 Test Case Prioritization Techniques

We consider 18 different test case prioritization techniques, which we classify into three groups. Table 1 lists these techniques by group. The first group is the *comparator* group, containing two "techniques" that are used in comparisons. The second group is the *statement level* group, containing four fine granularity techniques; these techniques were used in Rothermel et al. [30], but here they are examined in the context of version-specific prioritization. The third group is the *function level* group, containing 12 coarse granularity techniques; four are comparable to statement level techniques and eight add information on the probability of fault existence not used by the statement level techniques. Next, we describe each technique: Because the first six techniques have been presented algorithmically and analyzed in detail in [30], our discussion of these is abbreviated; the reader is referred to that reference for further details. Following this description, Section 3.4 summarizes and further classifies the techniques.

TABLE 1
Test Case Prioritization Techniques Considered in this Paper

Label	*Mnemonic*	*Description*
T1	random	randomized ordering
T2	optimal	ordered to optimize rate of fault detection
T3	st-total	prioritize on coverage of statements
T4	st-addtl	prioritize on coverage of statements not yet covered
T5	st-fep-total	prioritize on probability of exposing faults
T6	st-fep-addtl	prioritize on probability of faults, adjusted to consider previous test cases
T7	fn-total	prioritize on coverage of functions
T8	fn-addtl	prioritize on coverage of functions not yet covered
T9	fn-fep-total	prioritize on probability of exposing faults
T10	fn-fep-addtl	prioritize on probability of faults, adjusted to consider previous test cases
T11	fn-fi-total	prioritize on probability of fault existence
T12	fn-fi-addtl	prioritize on probability of fault existence, adjusted to consider previous test cases
T13	fn-fi-fep-total	prioritize on combined probabilities of fault existence and fault exposure
T14	fn-fi-fep-addtl	prioritize on combined probabilities of fault existence/exposure, adjusted on previous coverage
T15	fn-diff-total	prioritize on probability of fault existence
T16	fn-diff-addtl	prioritize on probability of fault existence, adjusted to consider previous test cases
T17	fn-diff-fep-total	prioritize on combined probabilities of fault existence and fault exposure
T18	fn-diff-fep-addtl	prioritize on combined probabilities of fault existence/exposure, adjusted on previous coverage

3.1 Comparator Techniques

T1: Random ordering. As an experimental control, one prioritization "technique" that we consider is the random ordering of the test cases in the test suite.

T2: Optimal ordering. For further comparison, we also consider an optimal ordering of the test cases in the test suite. We can obtain such an ordering in our experiments because we use programs with known faults and can determine which faults each test case exposes: this lets us determine the ordering of test cases that maximizes a test suite's rate of fault detection.[1] In practice, this is not a viable technique, but it provides an upper bound on the effectiveness of the other heuristics that we consider.

3.2 Statement Level Techniques

T3: Total statement coverage prioritization. Using program instrumentation, we can measure the coverage of statements in a program by its test cases. We can then prioritize test cases in terms of the total number of statements they cover by sorting them in order of coverage achieved. (If multiple test cases cover the same number of statements, we can order them pseudorandomly.)

Given a test suite of m test cases and a program of n statements, total statement coverage prioritization requires time $O(m\,n + m\,log\,m)$. Typically, n is greater than m, making this equivalent to $O(m\,n)$.

T4: Additional statement coverage prioritization. Additional statement coverage prioritization is like total coverage prioritization, but it relies on feedback about coverage attained so far in testing to focus on statements not yet covered. To do this, the technique greedily selects a test case that yields the greatest statement coverage, then adjusts the coverage data about subsequent test cases to indicate their coverage of statements not yet covered, and then iterates until all statements covered by at least one test case have been covered. When all statements have been covered, the remaining test cases are covered (recursively) by resetting all statements to "not covered" and reapplying additional statement coverage on the remaining test cases.

For a test suite and program containing m test cases and n statements, respectively, the cost of additional statement coverage prioritization is $O(m^2\,n)$, a factor of m more than total statement coverage prioritization.

T5: Total FEP prioritization. The ability of a fault to be exposed by a test case depends not only on whether the test case executes a faulty component, but also on the probability that a fault in that statement will cause a failure for that test case [14], [16], [31], [32]. Any practical determination of this probability must be an approximation, but we wish to know whether such an approximation might yield a prioritization technique superior in terms of rate of fault detection than techniques based solely on code coverage.

To approximate the fault-exposing-potential (FEP) of a test case, we used mutation analysis [7], [15]. Given program P and test suite T, for each test case $t \in T$, for each statement s in P, we determined the mutation score $ms(s,t)$ of t on s to be the ratio of mutants of s exposed by t to total mutants of s. We then calculated, for each test case t_k in T, an *award value* for t_k, by summing all $ms(s,t_k)$ values. Total fault-exposing-potential (total FEP) prioritization orders test cases in terms of these award values.

Given the $ms(s,t)$ values for a test suite containing m test cases and a program containing n statements, total FEP prioritization can be accomplished in time $O(m\,n + m\,log\,m)$. In general, n is greater than m, in which case, the cost of this prioritization is $O(m\,n)$, a worst-case time analogous to that for total statement coverage prioritization. The cost of calculating $ms(s,t)$ values, however, could be quite high, especially if these values are obtained through mutation analysis. If FEP prioritization shows promise, however, this would motivate a search for cost-effective approximators of fault-exposing potential.

1. As detailed in [30], the problem of calculating an optimal ordering is itself intractable, thus, we employ a heuristic that calculates an approximation to optimal. Despite this fact, our heuristic provides a useful benchmark against which to measure practical techniques because we know that a true optimal ordering could perform no worse than the ordering that we calculate.

T6: Additional FEP prioritization. Similar to the extensions made to total statement coverage prioritization to produce additional statement coverage prioritization, we incorporate feedback into total FEP prioritization to create additional fault-exposing-potential (FEP) prioritization. In additional FEP prioritization, after selecting a test case t, we lower the award values for all other test cases that exercise statements exercised by t to reflect our increased confidence in the correctness of those statements; we then select a next test case, repeating this process until all test cases have been ordered. This approach lets us account for the fact that additional executions of a statement may be less valuable than initial executions.

3.3 Function Level Techniques

T7: Total function coverage prioritization. Analogous to total statement coverage prioritization, but operating at the level of functions, total function coverage prioritization prioritizes test cases by sorting them in order of the total number of functions they execute. The technique has a worst-case cost analogous to that of statement coverage: $O(m\,n + m\,log\,m)$ for a test suite containing m test cases and a program containing n functions. The number of functions in a program is typically much smaller, however, than the number of statements in a program. Moreover, the process of collecting function-level traces is less expensive and less intrusive than the process of collecting statement-level traces. Thus, total function coverage prioritization promises to be cheaper than total statement coverage prioritization.

T8: Additional function coverage prioritization. Analogous to additional statement coverage prioritization, but operating at the level of functions, this technique incorporates feedback into total function coverage prioritization, prioritizing test cases (greedily) according to the total number of additional functions they cover. When all statements have been covered, we reset coverage vectors and reapply additional function coverage on the remaining test cases. The technique has a worst-case cost of $O(m^2\,n)$ for test suites of m test cases and programs of n functions.

T9: Total FEP (function level) prioritization. This technique is analogous to total FEP prioritization at the statement level. To translate that technique to the function level, we required a function-level approximation of fault-exposing potential. We again used mutation analysis, computing, for each test case t and each function f, the ratio of mutants in f exposed by t to mutants of f executed by t. Summing these values, we obtain award values for test cases. We then apply the same prioritization algorithm as for total FEP (statement level) prioritization, substituting functions for statements.

T10: Additional FEP (function level) prioritization. This technique incorporates feedback into the total FEP (function level) technique in the same manner used for the total FEP (statement level) technique.

T11: Total fault index (FI) prioritization. Faults are not equally likely to exist in each function; rather, certain functions are more likely to contain faults than others. This fault proneness can be associated with measurable software attributes [1], [3], [5], [20], [24]. In the context of regression testing, we are also interested in the potential influence, on fault proneness, of our modifications; that is, with the potential of modifications to lead to regression faults. This requires that our fault proneness measure account for attributes of software change [10]. We can account for the association of changes with fault-proneness by prioritizing test cases based on this measure.

For this technique, as a metric of fault proneness, we use a *fault index* which, in previous studies [10], [26], has proven effective at providing fault proneness estimates. The fault index generation process involves the following steps: First, a set of measurable attributes [9] is obtained from each function in the program. Second, the metrics are standardized using the corresponding metrics of a baseline version (which later facilitates the comparison across versions). Third, principal components analysis [19] reduces the set of standardized metrics to a smaller set of domain values, simplifying the dimensionality of the problem and removing the metrics colinearity. Finally, the domain values weighted by their variance are combined into a linear function to generate one fault index per function in the program.

Given program P and subsequent version P', generating (regression) fault indexes for P' requires generation of a fault index for each function in P, generation of a fault index for each function in P', and a function-by-function comparison of the indexes for P' against those calculated for P. As a result of this process, the regression fault proneness of each function in P' is represented by a regression fault index based on the complexity of the changes that were introduced into that function. Further details on the mechanisms of the method are given in [10], [13]. From this point forward and to simplify the nomenclature, we refer to "regression fault indexes" simply as "fault indexes."

Given these fault indexes, total fault index coverage prioritization is performed in a manner similar to total function coverage prioritization. For each test case, we compute the sum of the fault indexes for every function that test case executes. Then, we sort test cases in decreasing order of these sums, resolving ties pseudorandomly. Given the fault index for each of the n functions in the program, and m test cases, total fault index prioritization can be accomplished in $O(m\,n)$ time. The cost of obtaining the fault indexes for a program is bounded by the number of functions n and the size of the metric set on which the fault index is based. Since the generation of fault indexes does not involve test execution, its computational cost is significantly smaller than the cost of computing FEP values.

T12: Additional fault-index (FI) prioritization. Additional fault index coverage prioritization is accomplished in a manner similar to additional function coverage prioritization, by incorporating feedback into total fault index coverage prioritization. The set of functions that have been covered by previously executed test cases is maintained. If this set contains all functions (more precisely, if no test case adds anything to this coverage), the set is reinitialized to $\emptyset$. To find the next best test case, we compute, for each test

case, the sum of the fault indexes for each function that test case executes, except for functions in the set of covered functions. The test case for which this sum is the greatest wins. This process is repeated until all test cases have been prioritized.[2]

T13: Total FI with FEP coverage prioritization. We hypothesized that, by utilizing *both* an estimate of fault exposing potential *and* an estimate of fault proneness, we might be able to achieve a superior rate of fault detection. There are many ways in which one could combine these estimates; in this work, for each function, we calculate the product of the FI and FEP estimates for that function. We then calculate, for each test case, the sum of these products across the functions executed by that test case. We order test cases in descending order of that sum, resolving ties pseudorandomly.

T14: Additional FI with FEP coverage prioritization. We incorporate feedback into the previous technique to yield an "additional" variant. We again calculate, for each function, the product of its FI and FEP estimates. Next, we repeatedly calculate, for each test case not yet prioritized, the sum of these products across the functions executed by that test case, select the test case with the highest such sum, and reset the values for functions covered by that test case to zero, until all values are zero. If test cases remain, we reset the values for functions and repeat the process on the remaining test cases.

T15: Total DIFF prioritization. DIFF-based techniques are a simpler alternative to FI-based techniques for estimating fault proneness. While FI-based techniques require the collection of various metrics and the use of multivariate statistics, DIFF-based techniques require only the computation of syntactic differences between two versions of the program. With DIFF-based techniques, for each function present in both P and P', we measure degree of change by adding the number of lines listed as inserted, deleted, or changed, in the output of the Unix `diff` command applied to P and P'.

Although this DIFF-based approach does not capture all of the dimensions of complexity included in FI, the wide availability of "diff" tools makes this approach easily accessible to practitioners. Further, comparisons of the DIFF-based and FI-based approaches in terms of effects on rate of fault detection, when employed in prioritization, need to consider immediate practicality.

Total DIFF prioritization, therefore, is performed just like FI prioritization, with the exception that it relies on modification data derived from `diff`.

T16: Additional DIFF prioritization. Additional DIFF prioritization is analogous to additional FI prioritization, except that it relies on modification data derived from `diff`.

T17: Total DIFF with FEP prioritization. Total DIFF with FEP prioritization is analogous to total FI with FEP prioritization, except that it relies on modification data derived from `diff`.

T18: Additional DIFF with FEP prioritization. Additional DIFF with FEP prioritization is analogous to additional FI with FEP prioritization, except that it relies on modification data derived from `diff`.

3.4 Prioritization Techniques Summary

The foregoing test case prioritization techniques represent a broad spectrum of approaches, varying along several dimensions. One dimension mentioned already is granularity, considered here in terms of function-level and statement-level. Granularity affects the relative costs of techniques in terms of computation and storage, but also, we suspect, affects the relative effectiveness of those techniques.

A second dimension involves whether or not a technique employs feedback and is accounted for in the difference between "total" and "additional" techniques. "Total" techniques prioritize test cases based on information available at the outset of prioritization, whereas "additional" techniques adjust their efforts based on the effects of test cases previously positioned in the test case order being developed.

A third dimension involves whether or not a technique uses information from the modified program version. Techniques based solely on coverage information rely solely on data gathered on the original version of a program (prior to modifications) in their prioritizations. Techniques that rely on FEP estimation do not consider the specific modifications present in the modified version of a program; however, they attempt to factor in the potential effects of modifications in general. Techniques that rely on fault indexes, in contrast, explicitly utilize information about the modified program version.

Finally, the techniques we have suggested vary in terms of immediate practicality. Techniques based solely on coverage, at either the statement or function level, could be applied today given existing code instrumentation tools. Techniques utilizing fault index information of the type provided by `diff` could also be immediately applied. Furthermore, the implementation of the non-`diff`-based fault indexes described earlier is feasible, given current technology, and, with relatively little effort, these indexes could currently be utilized. In contrast, our investigation of FEP-based techniques is, due to the lack of a demonstrated, practical method for estimating FEP, purely exploratory. Such an exploration, however, is easily motivated: If FEP prioritization shows promise, this would justify a search for more cost-effective techniques for approximating fault-exposing potential, such as techniques that use constrained mutation [27].

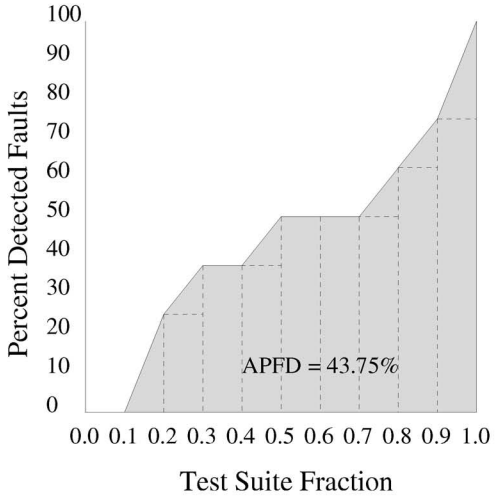
In presenting and discussing our results in subsequent sections, we comment further on each of these dimensions of variance and the effects they have on test case prioritization.

2. Here, a further approach analogous to that used for additional FEP prioritization also suggests itself. If fault indexes are understood to represent (in some sense) probabilities that faults exist in particular functions, then following selection of tests through particular functions, these fault indexes could be adjusted to indicate the reduced probability of a fault existing in those functions. This is analogous to the adjustment performed on reliability estimates when a fault is found in the testing process [25]. In this approach, functions are not ejected from the set of functions considered as they are covered. We leave investigation of this alternative as a topic for future work.

test	fault							
	1	2	3	4	5	6	7	8
A								
B	x	x						
C	x	x	x					
D		x	x					
E	x							x
F								x
G		x						
H				x				x
I	x	x	x	x	x			
J					x	x	x	

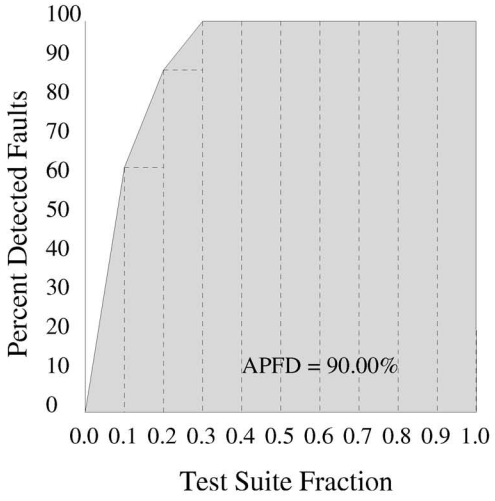
(a)

Test Case Order T1: A–B–C–D–E–F–G–H–I–J



(b)

Test Case Order T2: I–J–E–B–C–D–F–G–H–A



(c)

Fig. 1. Example illustrating the APFD measure.

4 Empirical Studies

In the studies that follow, we address the following specific research questions.

RQ1: Can version-specific test case prioritization improve the rate of fault detection of test suites?

RQ2: How do fine granularity (statement level) prioritization techniques compare to coarse granularity (function level) techniques in terms of rate of fault detection?

RQ3: Can the use of predictors of fault proneness improve the rate of fault detection of prioritization techniques?

4.1 Efficacy and APFD Measures

To quantify the goal of increasing a test suite's rate of fault detection, in [30] we introduce a metric, *APFD*, which measures the weighted average of the percentage of faults detected over the life of the suite. APFD values range from 0 to 100; higher numbers imply faster (better) fault detection rates.

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:

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

We illustrate this metric using an example. Consider a program with a test suite of 10 test cases, **A** through **I**, such that the program contains eight faults detected by those test cases, as shown by the table in Fig. 1a.

Consider two orders of these test cases, order $T1$: **A–B–C–D–E–F–G–H–I–J** and order $T2$: **I–J–E–B–C–D–F–G–H–A**. Figs. 1b and 1c show the percentages of faults detected

versus the fraction of the test suite used, for these two orders, respectively. The area inside the inscribed rectangles (dashed boxes) represents the weighted percentage of faults detected over the corresponding fraction of the test suite. The solid lines connecting the corners of the inscribed rectangles interpolate the gain in the percentage of detected faults. The area under the curve thus represents the weighted average of the percentage of faults detected over the life of the test suite.

On test order $T1$ (Fig. 1b), the first test case executed (**A**) detects no faults, but, after running test case B, two of the eight faults are detected; thus, 25 percent of the faults have been detected after 0.2 of test order $T1$ has been used. After running test case C, one more fault is detected and, thus, 37.5 percent of the faults have been detected after 0.3 of the test order has been used. Test order $T2$ (Fig. 1c), in contrast, is a much "faster detecting" test order than $T1$: The first 0.1 of the test order detects 62.5 percent of the faults and the first 0.3 of the test order detects 100 percent. ($T2$ is, in fact, an optimal ordering of the test suite, resulting in the earliest detection of the most faults.) The resulting APFDs for the two test case orders are 43.75 percent and 90.0 percent, respectively.

4.2 Empirical Approaches and Challenges

Two of the major challenges for this research involve finding adequate objects of study and selecting (and following) the appropriate empirical approaches to address the research questions.

Finding adequate objects of study is difficult because the candidates for empirical studies of prioritization must include programs, subsequent releases of those programs, test suites, and fault data. Obtaining such materials is a nontrivial task. Free software, often in multiple versions, is readily accessible, but free software is not typically equipped with test suites. Free software may be equipped with change logs, but such logs are often not sufficiently detailed. Commercial software vendors, who are more likely to maintain established test suites, are often reluctant to release their source code, test suites, and fault data to researchers. Even when vendors do make such materials available, they typically impose restrictions rendering the sharing of those materials, and their use in replication and validation of studies, infeasible. Finally, even when suitable experimental objects are available, prototype testing tools may not be robust enough to operate on those objects and the effort required to ensure adequate robustness in prototype research tools may be prohibitive.

Choosing the appropriate empirical approach is not simple because each approach presents different advantages and disadvantages and each approach affects and is affected by the availability of objects. One possible empirical approach is to perform controlled experiments on objects drawn partially "from the field" but further manipulated or created in a controlled environment. The advantage of such experiments is that the independent variables of interest (e.g., test suite constitution, modification patterns, and fault types) can be manipulated to determine their impact on dependent variables. This lets us apply different values to the independent variables in a controlled fashion so that results are not likely to depend on unknown or uncontrolled factors. The primary weakness of this approach, however, is the threat to external validity posed by the "manufacturing" of test cases, faults, and modifications.

A second empirical approach is to perform case studies on existing programs, taken "from the field," that have several versions, fault data, and existing test suites. Such objects have the advantage of being "real" and can reduce some investigation costs due to the availability of elements that do not need to be artificially created (e.g., test suites). Under this approach, however, certain factors that may influence prioritization are not controlled, which makes replication much more difficult. For example, test suites may be created by different or even unknown methodologies and there may be only one test suite per program or version. Similarly, modification patterns may differ among programs: Some programs may be released frequently with few changes per release, other programs may be released less frequently with many changes per release. Further, the type of fault data that is available with programs may differ among different programs due to the use of different recording practices. Such differences and their dependency on individual cases may complicate attempts to draw general conclusions, while still requiring careful investigation to avoid misinterpretation.

Thus, each approach—controlled experiments and case studies—has different advantages and disadvantages and, ultimately, a fuller understanding of prioritization techniques requires both.

The foregoing issues have shaped the family of empirical studies presented in this article. We begin by describing a set of controlled experiments on several relatively small programs that perform well-defined tasks. We follow with a set of case studies on two larger programs for which some components (test suites and faults) were (of necessity and by processes designed to limit sources of bias) manufactured, while others (modifications to create new releases) were provided, and on a third program for which all components were provided. This diversity among objects studied and empirical approaches lets us explore and evaluate the performance of various prioritization techniques in different situations. Furthermore, as we expect that the relative effectiveness of techniques may vary across programs, this approach lets us probe the relative strengths of those techniques. Finally, the combination of controlled experiments and case studies lets us begin to address concerns for both external and internal validity.

5 Controlled Experiments

We present our controlled experiments first.

5.1 Experiment Instrumentation

5.1.1 Programs

We used eight C programs, with faulty versions and a variety of test cases, as objects of study. Seven of these programs were assembled by researchers at Siemens Corporate Research for experiments with control-flow and data-flow test adequacy criteria [18]; we refer to these as the *Siemens programs*. The eighth program, `space`, was

TABLE 2
Experiment Objects

Program	Lines of Code	1st-order Versions	Test Pool Size	Test Suite Avg. Size
tcas	138	41	1608	6
schedule2	297	10	2710	8
schedule	299	9	2650	8
tot_info	346	23	1052	7
print_tokens	483	7	4130	16
print_tokens2	402	10	4115	12
replace	516	32	5542	19
space	6218	35	13585	155

developed for the European Space Agency; we refer to this program as the *space program*.

Table 2 provides metrics on the programs; we explain the meaning of these metrics in the following paragraphs. Note that these programs were also used in the earlier studies reported in [30]; here, we reuse these materials to investigate different research questions.

Siemens programs. The Siemens programs perform various tasks: `tcas` models an aircraft collision avoidance algorithm, `schedule2` and `schedule` are priority schedulers, `tot_info` computes statistics, `print_tokens` and `print_tokens2` are lexical analyzers, `replace` performs pattern matching and substitution. For each program, the Siemens researchers created a *test pool* of black-box test cases using the category partition method [4], [28]. They then augmented this test pool with manually created white-box test cases to ensure that each exercisable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 test cases. The researchers also created faulty versions of each program by modifying code in the base version; in most cases, they modified a single line of code and, in a few cases, they modified between two and five lines of code. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. To obtain meaningful results, the researchers retained only faults that were detectable by at least three and at most 350 test cases in the test pool.

Space program. The `space` program is an interpreter for an array definition language (ADL). The program reads a file of ADL statements and checks the contents of the file for adherence to the ADL grammar and specific consistency rules. If the ADL file is correct, `space` outputs an array data file containing a list of array elements, positions, and excitations; otherwise, the program outputs error messages. The `space` program has 35 versions, each containing a single fault: 30 of these were discovered during the program's development, five more were discovered subsequently [30]. The test pool for `space` was constructed in two phases. The pool was initialized to 10,000 test cases randomly generated by Vokolos and Frankl [33]. Then, new test cases were added until each executable edge in the program's control flow graph was exercised by at least 30 test cases. This process produced a test pool of 13,585 test cases.

Test Suites. Sample test suites for these programs were constructed using the test pools for the base programs and test-coverage information about the test cases in those pools. More precisely, to generate a test suite T for base program P from test pool T_p, the C pseudo-random-number generator `rand`, seeded initially with the output of the C `times` system call, was used to obtain integers that were treated as indexes into T_p (modulo $|T_p|$). These indexes were used to select test cases from T_p; each test case t was added to T only if t added to the cumulative branch coverage of P achieved by the test cases added to T thus far. Test cases were added to T until T contained at least one test case that would exercise each executable branch in the base program. Table 2 lists the average sizes of the 1,000 branch-coverage-adequate test suites generated by this procedure for each of the object programs.

For our experimentation, we randomly selected 50 of these test suites for each program.

Versions. For these experiments, we required program versions with varying numbers of faults; we generated these versions in the following way: Each program was initially provided with a correct base version and a *fault base* of versions containing exactly one fault. We call these first-order versions. We identified, among these first-order versions, all versions that do not interfere—that is, all faults that can be merged into the base program and exist simultaneously. For example, if fault f1 is caused by changing a single line and fault f2 is caused by deleting the same line, then these modifications interfere with each other.

We then created higher-order versions by combining noninterfering first-order versions. To limit the threats to our experiment's validity, we generated the same number of versions for each of the programs. For each program, we created 29 versions; each version's order varied randomly between 1 and the total number of noninterfering 1st-order versions available for that program.[3] At the end of this process, each program was associated with 29 multifault versions, each containing a random number of faults.

5.1.2 Prioritization and Analysis Tools

To perform the experiments, we required several tools. Our test coverage and control-flow graph information was provided by the Aristotle program analysis system [17]. We created prioritization tools implementing the techniques outlined in Section 3. To obtain mutation scores for use in FEP prioritization, we used the Proteum mutation system [6]. To obtain fault index information, we used three tools [9], [11]: source code measurement tools for generating complexity metrics, a fault index generator, and a comparator for evaluating each version against the baseline version. To determine the syntactic differences between two versions, we employed a modified version of the Unix `diff` utility. To generate the multiple-fault versions, scripts were written to implement the strategy outlined in the previous section.

3. The number of versions, 29, constitutes the minimum among the maximum number of versions that could be generated for each program given the interference constraints.

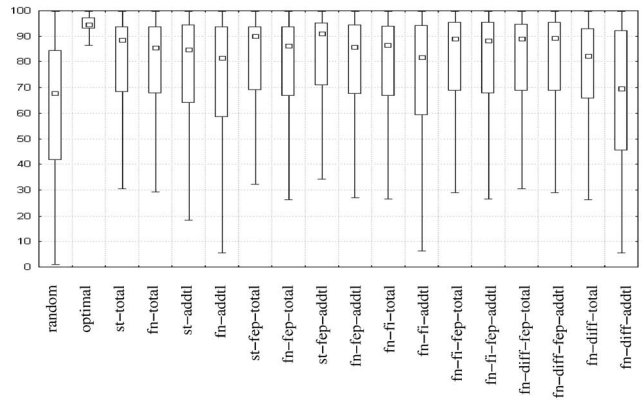


Fig. 2. APFD boxplots for an "all programs" total. The horizontal axis lists techniques and the vertical axis lists APFD scores.

5.2 Experiment Design, Results, and Analysis

We performed several experiments, each addressing one of our research questions. Each experiment included five stages:

1. stating a research question in terms of an hypothesis,
2. formalizing the experiment through a robust design,
3. collecting data,
4. analyzing data to test the hypothesis, and
5. identifying the threats to the experiment's validity.

In general, each experiment examined the results of applying certain test case prioritization techniques to each program and its set of versions and test suites.

To provide an overview of all the collected data,[4] we include Figs. 2 and 3 with box plots.[5] Fig. 2 displays a plot for an "all programs" total and Fig. 3 displays an individual plot for each of the programs. Each plot contains a box showing the distribution of APFD scores for each of the 18 techniques.

The following sections describe, for each of our research questions in turn, the experiment(s) relevant to that question, presenting their design and the analysis of their results.

5.2.1 Experiment 1 (RQ1): Version-Specific Prioritization

Our first research question considers whether version-specific test case prioritization can improve the fault-detection abilities of test suites. Since we conjectured that differences in the granularity at which prioritization is performed would cause significant differences in APFD values, we performed two experiments: Experiment 1a involving statement level techniques st-total, st-addtl, st-fep-total, and st-fep-addtl, and Experiment 1b involving function level techniques fn-total, fn-addtl, fn-fep-total, and fn-fep-addtl. This separation into two experiments gave us more power to determine differences among the techniques within each group.

4. For simplicity, data belonging to separate experiments are presented together.

5. Box plots provide a concise display of a data distribution. The small rectangle embedded in each box marks the mean value. The edges of the box are bounded by the standard error. The whiskers extend to one standard deviation.

Both experiments followed the same factorial design: All combinations of all levels of all factors were investigated. The main factors were program and prioritization technique. Within programs, there were eight levels (one per program) with 29 versions and 50 test suites per program. We employed four prioritization techniques per experiment. Each treatment (prioritization technique) was applied to every viable[6] combination of test suite and version within each program generating a maximum of 46,400 observations (each including an APFD value) per experiment.

We then performed an analysis of variance (ANOVA) on those observations to test the differences between the techniques' mean APFD values. We considered the main effects program and technique and the interaction among those effects. When the ANOVA F-test showed that the techniques were significantly different, we proceeded to determine which techniques contributed the most to that difference and how the techniques differed from each other through a Bonferroni multiple comparison method. This procedure works within the ANOVA setting to compare the techniques' means while controlling the family-wise type of error.

Experiment 1a: Statement Level. Table 3 presents ANOVA results for Experiment 1a, considering all programs. The treatments are in the first column and the sum of squares, degrees of freedom, and mean squares for each treatment are in the following columns. The F values constitute the ratio between the treatment and the error effect (last row). The larger the F statistic, the greater the probability of rejecting the hypothesis that the techniques' mean APFD values are equal. The last column presents the p-values, which represent "the probability of obtaining a value of the test statistic that is equal to or more extreme than the one observed" [21]. Since we selected our level of significance to be 0.05 percent, we reject the hypotheses when the p-value is less than or equal to that level of significance. Otherwise, we do not reject the hypothesis.

The results indicate that there is enough statistical evidence to reject the null hypothesis; that is, the means for the APFD values generated by different statement level techniques were different. However, the analysis also indicates that there is significant interaction between techniques and programs: The difference in response between techniques is not the same for all programs. Thus, individual interpretation is necessary. As a first step in this interpretation, we performed an ANOVA on each of the programs. Each of the ANOVAs was significant, indicating that, within each program, the statement level prioritization techniques were significantly different. (Results of these ANOVAs are presented in [8].)

The ANOVAs evaluated whether the techniques differed, the APFD means ranked the techniques, a multiple comparison procedure using Bonferroni analysis quantifies how the techniques differed from each other. Table 4 presents the results of this analysis for all of the programs, ranking the techniques by mean. Grouping letters indicate

6. Due to characteristics of the FEP calculations, some combinations of test suite and version were not employed.

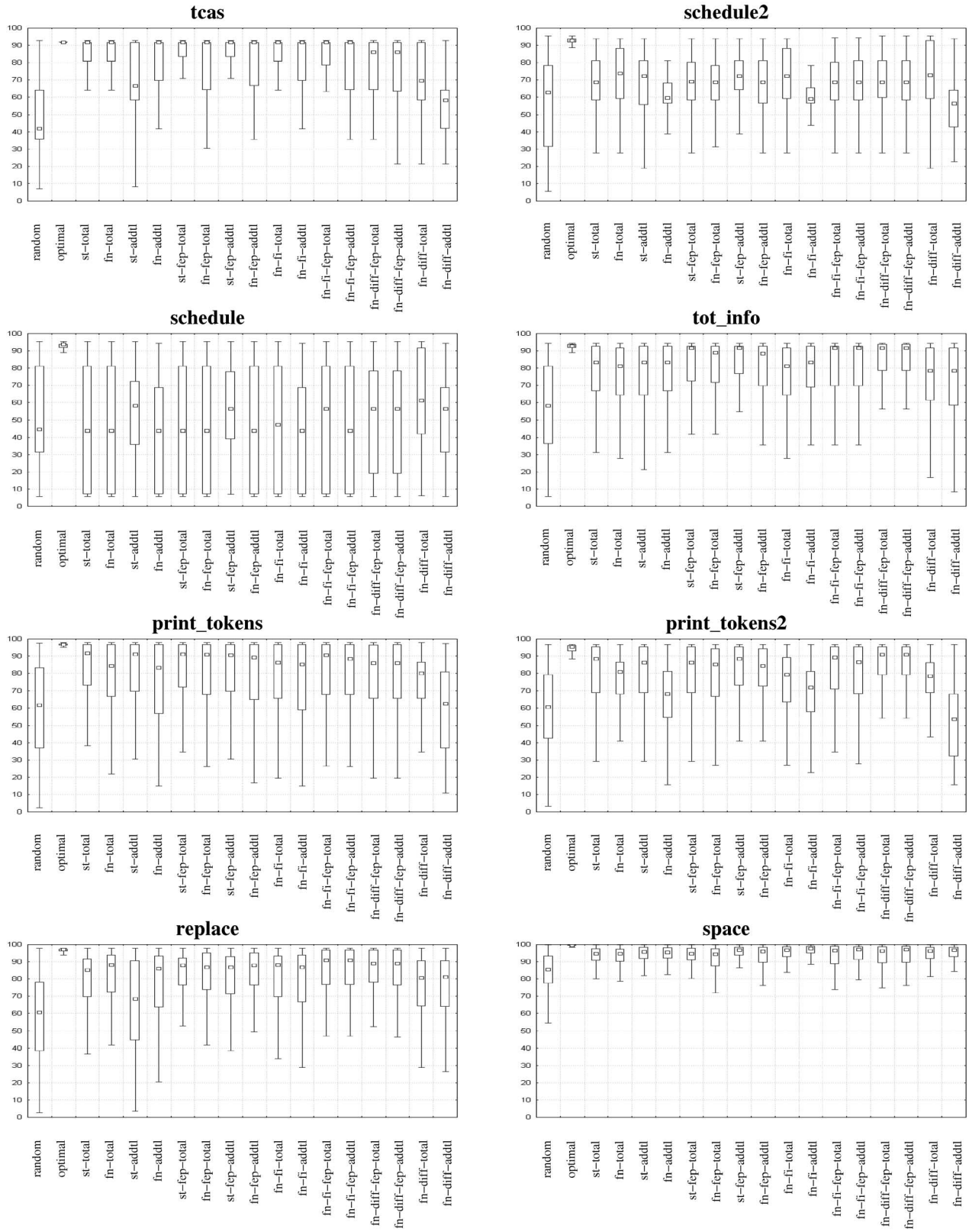


Fig. 3. APFD boxplots for individual programs. The horizontal axes list techniques and the vertical axes list APFD scores.

differences: Techniques with the same grouping letter were not significantly different. For example, st-fep-total has a larger mean than st-total, but they are grouped together because they were not significantly different. On the other hand, the st-fep-addtl technique, which uses FEP information and additional coverage, was significantly better than the other techniques. The last technique ranked is st-addtl, which was significantly weaker than the others.

TABLE 3
ANOVA, Statement Level Techniques, All Programs

	SS	Degrees of freedom	MS	F	p
PROGRAM	3473054	7	496150.60	1358.512	0.00
TECHN	97408	3	32469.20	88.904	0.00
PROGRAM*TECHN	182322	21	8682.00	23.772	0.00
Error	9490507	25986	365.22		

To consider results on a per-program basis, we performed a Bonferroni analysis on each of the programs. (Full results of these analyses are presented in [8]; we summarize those results here.) On `replace`, st-fep-total, st-fep-addtl, and st-total ranked at the top, but were not significantly different from each other. The same scenario held for `schedule2` and `tcas`. On `schedule`, the techniques that use feedback (st-fep-addtl and st-addtl) ranked at the top, but were not significantly different, while the techniques that do not use feedback (st-total and st-fep-total) were significantly inferior. On `space`, st-fep-addtl was significantly better than other techniques, while the rest of the techniques did not differ from each other. `Print_tokens` presented a unique case because the Bonferroni process could not find differences among any pair of techniques, even when the ANOVA specified that there was significant difference when the four of them were considered. On `print_tokens2`, st-fep-addtl ranked at the top, followed by the other techniques among which there was no significant difference. Finally, `tot_info`'s ranking matched the overall ranking for all applications, although no significant difference was found between techniques using and not using feedback.

To summarize, although the rankings of techniques did vary somewhat among programs, similarities did occur across all or across a large percentage of the programs. Specifically, st-fep-addtl ranked in the highest Bonferroni group of techniques independent of the program; st-fep-total and st-total were in the same group (not significantly different) on seven of the eight programs; and, finally, st-addtl ranked significantly worse than all other techniques on four programs.

Experiment 1b: Function Level. Table 5 presents the analysis of variance results for Experiment 1b (function level techniques) considering all programs. The interaction effects between techniques and programs were also significant for function-level techniques and the results revealed significant differences among the techniques. Moreover, the techniques ranked in the same order as their statement-level equivalents, with fn-fep-addtl first, fn-fep-total second, fn-total third, and fn-addtl last.

TABLE 4
Bonferroni Means Separation Tests, Statement Level Techniques, All Programs

Grouping	Means	Techniques
A	80.733	st-fep-addtl
B	78.867	st-fep-total
B	78.178	st-total
C	76.077	st-addtl

However, as shown by the results of Bonferroni analysis (Table 6), the top three techniques were not significantly different from each other.

Following the same steps as in Experiment 1a, we next performed ANOVAs and Bonferroni analyses on a per program basis. (Full results of these analyses are presented in [8]; we summarize those results here.) The results on `replace`, `schedule`, `print_tokens`, and `tot_info` present trends similar to those seen in the Bonferroni results for all programs. On `print_tokens2`, the ranking was identical, but all the techniques produced significantly different averages. `Schedule2`, `tcas`, and `space` present a different perspective. On `schedule2` and `tcas`, fn-total was significantly better than the other techniques. On `space`, fn-addtl was the best, fn-total came second, and the FEP-based techniques followed.

In summary, for the function-level techniques, we observed great variation in the techniques' performance across subjects. The most surprising result was the lack of significant gains observed, for function-level techniques, when using FEP estimates. At a minimum, this suggests that our method for estimating FEP values at the function level may not be as powerful as our method for estimating those values at the statement level. Furthermore, at the function level, except for `print_tokens2`, the two FEP techniques were not significantly different from one another. This implies that feedback had no effect when employing function level FEP techniques. We also observed that using feedback could have a negative impact on APFD values. There is a possible explanation for this. Techniques at the function level employing feedback give higher priority to tests that execute uncovered functions, discarding functions already executed independently of the section or percentage of code in those functions that has actually been covered. If those partially covered functions are faulty, but their faulty sections have not yet been covered and the tests executing those functions are given low priority by techniques with feedback, then APFD values for techniques employing feedback could be lower.

5.2.2 Experiment 2 (RQ2): Granularity Effects

Our second research question concerns the relationship between fine and coarse granularity prioritization techniques. Initial observations on the data led us to hypothesize that granularity has an effect on APFD values. This is suggested by comparing Table 4 to Table 6: For all cases, the mean APFD values for function level techniques were smaller than the mean APFD values for corresponding statement level techniques (for example, the mean APFD for

TABLE 5
ANOVA, Basic Function Level Techniques, All Programs

	SS	Degrees of freedom	MS	F	p
PROGRAM	4139625	7	591375.00	1501.327	0.00
TECHN	60953	3	20317.80	51.581	0.00
PROGRAM*TECHN	158227	21	7534.60	19.128	0.00
Error	10071668	25569	393.90		

fn-fep-addtl was 77.45, but for st-fep-addtl it was 80.73). The radar chart in Fig. 4 further illustrates this observation. In the radar chart, each technique has its own APFD value axis radiating from the center point. There are two polygons, representing the granularities at the statement and function levels, respectively. The radar chart shows that each function level technique had a smaller APFD than its counterpart at the statement level and that statement level techniques as a whole were better (cover a larger surface) than function level techniques. The chart also shows that techniques employing feedback were more sensitive to the shift in granularity.

To formally address this research question, we performed a pairwise analysis among the following pairs of techniques: (st-total, fn-total), (st-addtl, fn-addtl), (st-fep-total, fn-fep-total), and (st-fep-addtl, fn-fep-addtl). The four orthogonal contrasts were significantly different as shown in Tables 7 and 8.[7] That is, for these four pairs of techniques, different levels of granularity had a major effect on the value of the fault detection rate. Thus, in spite of the different rankings obtained in Experiments 1a and 1b, there is enough statistical evidence to confirm that statement level techniques were more effective than function level techniques.

Analyses on a per-program basis present a similar picture. Although, in several cases, statement-level techniques are not significantly better than their corresponding function-level techniques (e.g., on `schedule`, st-total and fn-total do not differ significantly), only two cases occur in which a function-level technique significantly outperforms its corresponding statement-level technique. (These cases all involve st-addtl versus fn-addtl and occur on `tcas` and `space`.) (These results are presented in full in [8].)

5.2.3 Experiment 3 (RQ3): Adding Prediction of Fault Proneness

Our third research question considered whether predictors of fault proneness can be used to improve the rate of fault-detection of prioritization techniques. We hypothesized that incorporation of such predictors *would* increase technique effectiveness. We designed an experiment (Experiment 3) to investigate this hypothesis at the function level. The experiment design was analogous to the design used in Experiment 1b except for the addition of eight new techniques: fn-fi-total, fn-fi-addtl, fn-fi-fep-total, fn-fi-fep-addtl, fn-diff-total, fn-diff-addtl, fn-diff-fep-total, and fn-diff-fep-addtl.

7. We could have performed a series of simple t-tests to compare the contrasts. However, we decided to take a more conservative approach with a post hoc Bonferroni analysis, which is also consistent with the other analyses.

The ANOVA of the data collected in this experiment (see Table 9) indicated that these techniques were significantly different. We then followed the same procedure used earlier, employing a Bonferroni analysis to gain insight into the differences. The results are presented in Table 10. Three techniques combining FEP and fault proneness (fn-diff-fep-addtl, fn-diff-fep-total, and fn-fi-fep-total) were significantly better than the rest. This suggests that some of the combinations of fault-proneness and FEP estimators we employed did significantly improve the power of our prioritization techniques. Fn-fi-fep-addtl and other techniques using either FEP estimates or fault indexes followed. We could not distinguish significant and consistent gains by any particular method (DIFF, FI, or FEP) when used individually. Also, the use of feedback seems to have a negative effect on the techniques using fault proneness, as evidenced by the significant superiority of fn-diff-total and fn-fi-total over fn-diff-addtl and fn-fi-addtl, respectively.

Table 9 shows that the interaction between program and technique was again, in this experiment, significant. So, to better understand the APFD variations, we analyzed the impact of techniques on each program separately. (Full results of these analyses are presented in [8]; we summarize those results here.) First, we performed univariate ANOVAs on each program. The results of those individual ANOVAs were consistent in indicating that all techniques were significantly different.

We next performed individual Bonferroni analyses per program. Several programs (`print_tokens`, `print_tokens2`, `tot_info`, and `replace`) exhibited rankings similar to those seen in the overall analysis, though, in some cases, with fewer significant differences among the techniques. Results on the other programs differed more substantially. On `tcas`, the techniques' APFD values descended gradually, which created overlap among the top ranked techniques. Still, there was a group of significantly best techniques that included fn-total, fn-fi-total, fn-addtl, and fn-fi-fep-total. The techniques using DIFF, however, ranked significantly worse than the others. On `schedule`, in contrast, fn-diff-total performed significantly better than the other techniques and the

TABLE 6
Bonferroni Means Separation Tests, Basic Function Level Techniques, All Programs

Grouping	Means	Techniques
A	77.453	fn-fep-addtl
A	76.957	fn-fep-total
A	76.928	fn-total
B	73.465	fn-addtl

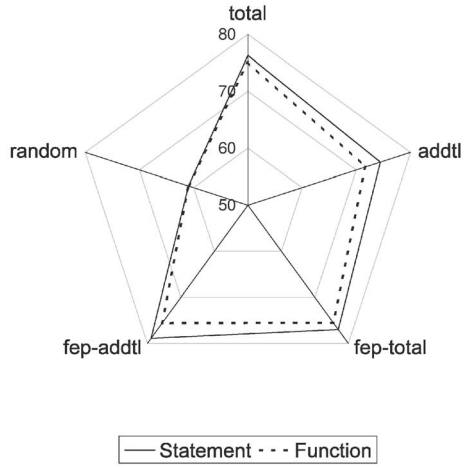


Fig. 4. Radar chart.

remaining techniques fell into a series of overlapping groups. A similar picture occurred for `schedule2`, except that, here, fn-diff-addtl was significantly worse than other techniques. Finally, results on `space` were unique. On this program, techniques using just fault proneness were significantly better than the others. The highest APFD values were generated through fn-fi-addtl, which was significantly superior to the other techniques. Combinations of FEP and fault indexes did not work as well as for other programs. Furthermore, the two techniques using just FEP estimates were ranked last.

In summary, on most programs, techniques combining FEP and FI ranked among the top techniques. However, certain programs presented unique characteristics that impacted the effectiveness of those techniques. Still, on all programs, a subset of the techniques using fault proneness measures were considered significantly better than (or not different from) techniques not using that predictor. It is also interesting that the use of feedback seemed to have a greater impact on simpler techniques, while, on techniques combining FEP and fault proneness measures, the impact of using feedback did not translate into significant gains (e.g., fn-diff-fep-addtl was not significantly different from fn-diff-fep-total).

5.2.4 Overall Analysis

Finally, to gain an overall perspective on all techniques, we performed ANOVAs and Bonferroni analyses on all the techniques including optimal and random (see Tables 11 and 12). As expected, the ANOVAs revealed significant differences among the techniques and the Bonferroni analysis generated groups, which confirmed our previous observations. The most obvious observation is that the optimal technique was still significantly better than all other techniques; this suggests that there is still room for improvement in prioritization techniques. However, all techniques significantly outperformed random ordering. St-fep-addtl remained the best performing technique after optimal. Yet, the group of techniques ranked next included function level techniques combining fault proneness measures and FEP. These function level techniques were significantly better than st-addtl.

TABLE 8
Bonferroni Analysis, Function vs. Statement Level Techniques, All Programs

Grouping				Means	Techniques
A				80.733	st-fep-addtl
B				78.867	st-fep-total
B	C			78.178	st-total
	C	D		77.453	fn-fep-addtl
		D	E	76.957	fn-fep-total
		D	E	76.928	fn-total
			E	76.077	st-addtl
F				73.465	fn-addtl

5.3 Threats to Validity

In this section, we present a synthesis of the potential threats to validity of our study, including: 1) threats to internal validity (could other effects on our dependent variables be responsible for our results), 2) threats to construct validity (are our independent variables appropriate), and 3) threats to external validity (to what extent do our results generalize). We also explain how we tried to reduce the chances that those threats affect the validity of our conclusions.

5.3.1 Threats to Internal Validity

The inferences we made about the effectiveness of prioritization techniques could have been affected by the following factors: 1) Faults in the prioritization and APFD measurement tools. To control for this threat, we performed code reviews on all tools and validated tool outputs on a small but nontrivial program. 2) Differences in the code to be tested, the locality of program changes, and the composition of the test suite. To reduce this threat, we used a factorial design to apply each prioritization technique to each test suite and each object program. 3) FEP, FI, and DIFF calculations. FEP values are intended to capture the probability, for each test case and each statement, that if the statement contains a fault, the test case will expose that fault. We used mutation analysis to provide

TABLE 7
ANOVA, Function vs. Statement Level Techniques, All Programs

	SS	Degrees of freedom	MS	F	p
PROGRAM	7516093	7	1073728.	2829.748	0.00
TECHN	198981	7	28426.00	74.915	0.00
PROGRAM*TECHN	434621	49	8870.00	23.376	0.00
Error	19562175	51555	379.44		

TABLE 9
ANOVA, All Function Level Techniques, All Programs

	SS	Degrees of freedom	MS	F	p
PROGRAM	11860200	7	1694314.00	4580.131	0.00
TECHN	800070	11	72734.00	196.616	0.00
PROGRAM*TECHN	1220361	77	15849.00	42.843	0.00
Error	28551710	77182	369.93		

an estimate of these FEP values; however, other estimates might be more precise and might increase the effectiveness of FEP-based techniques. Similar reasoning applies to our calculations of FI and DIFF.

5.3.2 Threats to Construct Validity

The goal of prioritization is to maximize some predefined criteria by scheduling test cases in a certain order. In this article, we focused on maximizing the rate of fault detection and we defined APFD to represent it. However, APFD is not the only possible measure of rate of fault detection and has some limitations.

1. APFD assigns no value to subsequent test cases that detect a fault already detected; such test cases may, however, help debuggers isolate the fault and, for that reason, might be worth accounting for.
2. APFD does not account for the possibility that faults and test cases may have different costs.
3. APFD only partially captures aspects of the effectiveness of prioritization; we need to consider other measures for purposes of assessing effectiveness. One might not even want to measure *rate* of detection; one might instead measure the percentage of the test cases in a prioritized test suite that must be run before *all* faults have been detected.
4. We employed a greedy algorithm for obtaining "optimal" orderings. This algorithm may not always find the true optimal ordering and this might allow some heuristic to actually outperform the optimal and generate outliers. However, a true optimal ordering can only be better than the greedy optimal ordering that we utilized; therefore, our approach is conservative and cannot cause us to claim significant differences between optimal and any heuristic where such significance would not exist.

TABLE 10
Bonferroni Analysis, All Function Level Techniques, All Programs

Grouping				Means	Techniques
A				79.479	fn-diff-fep-addtl
A				79.450	fn-diff-fep-total
A	B			78.712	fn-fi-fep-total
	B	C		78.167	fn-fi-fep-addtl
		C	D	77.453	fn-fep-addtl
		C	D	77.321	fn-fi-total
		C	D	77.057	fn-diff-total
			D	76.957	fn-fep-total
			D	76.928	fn-total
E				74.596	fn-fi-addtl
E				73.465	fn-addtl
F				67.666	fn-diff-addtl

5.3.3 Threats to External Validity

The generalization of our conclusions is constrained by several threats. 1) Object representativeness. The object programs are of small and medium size and have simple fault patterns that we have manipulated to produce versions with multiple faults. Complex industrial programs with different characteristics may be subject to different cost-benefit trade-offs. 2) Testing process representativeness. If the testing process we used is not representative of industrial ones, the results might not generalize. Furthermore, test suite constitution is also likely to differ under different processes. Control for these two threats can be achieved only through additional studies using a greater range and number of software artifacts.

6 Case Studies

In this section, we present three case studies.[8] These case studies offer us the opportunity to scale up our investigation of prioritization techniques by focusing on larger objects drawn from the field.

6.1 Objects of Study

We considered three programs, including two open-source Unix utilities and an embedded real-time subsystem of a level-5 RAID storage system.

6.1.1 Grep and Flex

`Grep` and `flex` are common Unix utility programs; `grep` searches input files for a pattern and `flex` is a lexical analyzer generator. The source code for both programs is publicly available. For this study, we obtained five versions of `grep` and five of `flex`. The earliest version of `grep` that we used contained 7,451 lines of C code and 133 functions; the earliest version of `flex` contained 9,153 lines of C code and 140 functions. Tables 13 and 14 provide data about the numbers of functions and lines changed (modified, added, or deleted) in each of the versions of the two programs, respectively.

The `grep` and `flex` programs possessed the advantage of being publicly available in multiple versions; however, neither program was equipped with test suites or fault data. Therefore, we manufactured these. To do this in as fair and

8. Two of the programs (`grep` and `flex`) used in these studies, with their versions, faults, and test suites, as well as the data collected about those programs, can be obtained by contacting the authors. The third program (`QTB`) cannot be made available, but portions of the data collected on that program can be obtained by contacting the authors.

TABLE 11
ANOVA, All Techniques, All Programs

	SS	Degrees of freedom	MS	F	p
PROGRAM	15205111	7	2172159.00	6062.476	0.00
TECHN	4654397	17	273788.00	764.140	0.00
PROGRAM*TECHN	2507689	119	21073.00	58.815	0.00
Error	41709550	116400	358.30		

unbiased a manner as possible, we adapted processes used by Hutchins et al. to create the Siemens programs materials [18] (also outlined in Section 5.1 of this article), as follows:

For each program, we used the category partition method and an implementation of the TSL tool [4], [28] to create a suite of black-box tests, based on the program's documentation. These test suites were created by graduate students experienced in testing, but who were not involved in and were unaware of the details of this study. The resulting test suites consisted of 613 test cases for `grep`, exercising 79 percent of that program's functions, and 525 test cases for `flex`, exercising 89 percent of that program's functions.

To evaluate the performance of prioritization techniques with respect to rate of detection of regression faults, we require such faults—faults created in a program version as a result of the modifications that produced that version. To obtain such faults for `grep` and `flex`, we asked several graduate and undergraduate computer science students, each with at least two years experience programming in C and each unacquainted with the details of this study, to become familiar with the code of the programs and to insert regression faults into the versions of those programs. These fault seeders were instructed to insert faults that were as realistic as possible based on their experience with real programs and that involved code deleted from, inserted into, or modified in the versions.

TABLE 12
Bonferonni Analysis, All Techniques, All Programs

Grouping					Means	Techniques
A					94.728	optimal
B					80.733	st-fep-addtl
C					79.479	fn-diff-fep-addtl
C					79.450	fn-diff-fep-total
C	D				78.867	st-fep-total
C	D				78.712	fn-fi-fep-total
	D	E			78.178	st-total
	D	E			78.167	fn-fi-fep-addtl
		E	F		77.453	fn-fep-addtl
		E	F		77.321	fn-fi-total
		E	F	G	77.057	fn-diff-total
			F	G	76.957	fn-fep-total
			F	G	76.928	fn-total
				G	76.077	st-addtl
H					74.596	fn-fi-addtl
H					73.465	fn-addtl
I					67.666	fn-diff-addtl
J					62.100	random

To further direct their efforts, the fault seeders were given the following list of types of faults to consider:

- faults associated with variables, such as with definitions of variables, redefinitions of variables, deletions of variables, or changes in values of variables in assignment statements;
- faults associated with control flow, such as addition of new blocks of code, deletions of paths, redefinitions of execution conditions, removal of blocks, changes in order of execution, new calls to external functions, removal of calls to external functions, addition of functions, or deletions of functions;
- faults associated with memory allocation, such as not freeing allocated memory, failing to initialize memory, or creating erroneous pointers.

After at least 20 potential faults had been seeded in each version of each program,[9] we activated these faults individually, one by one, and executed the test suites for the programs to determine which faults could be revealed by test cases in those suites. We selected, for use in this study, all faults that were exposed by at least one and at most 20 percent of the test cases in the associated test suite. (Exclusion of faults not exposed does not affect APFD results; we chose to exclude faults exposed by more than 20 percent of the test suites on the grounds that easily exposed faults are more likely to be detected and removed during testing by developers and prior to formal regression testing than faults exposed less easily.) The numbers of faults remaining, and utilized in the studies, are reported in Tables 13 and 14.

6.1.2 QTB

`QTB`[10] is an embedded real-time subsystem that performs initialization tasks on a level-5 RAID storage system. In addition, it provides fault tolerance and recovery capabilities. `QTB` contains over 300K lines of C code combined with hundreds of in-line assembly-code statements across 2,875 functions. `QTB` had been under maintenance for several years.

In this study, we considered six `QTB` versions, the first of which we treated as the baseline. Table 15 reports details about these versions. The versions constituted major system releases produced over a six month period. For each version, test engineers employed a regression test suite to exercise system functionalities. The execution of the test

9. On version four of `flex`, due to the small number of modifications in that version, fewer than 20 potential faults were initially seeded.

10. Because our industry partner wishes to remain anonymous, we have changed the original names of the subsystem and versions that comprise this object.

TABLE 13
The `grep` Object

Version	Number of functions changed	Number of lines changed	Number of regression faults
(baseline)	–	–	–
1	81	2488	4
2	40	716	3
3	26	513	3
4	110	1891	1

TABLE 14
The `flex` Object

Version	Number of functions changed	Number of lines changed	Number of regression faults
(baseline)			
1	28	333	5
2	72	1649	4
3	12	40	8
4	6	91	1

TABLE 15
The `QTB` Object

Version	Number of functions changed	Total number of regression faults	Number of regression faults exposed by test cases
(baseline)	–	–	–
1	15	10	8
2	3	1	1
3	98	2	2
4	7	2	2
5	169	7	4

suite required, on average, 27 days. The test suite included 135 test cases that exercised 69 percent of the functions in the baseline version. The coverage information available for `QTB` is exclusively at the function level. (Software instrumentation tools designed to produce finer granularity coverage data caused the system to fail due to timing problems.)

Maintenance activities applied to `QTB` resulted in the unintentional incorporation into the system of 22 (discovered) regression faults. Table 15 summarizes the fault data. Observe that only 17 of the 22 faults were exposed by the regression test suite across the versions; only these faults factor into the calculation of APFD values.[11] Also, note that the execution of a faulty function did not guarantee exposure of faults in that function.

6.2 Design

In each case study, we investigate whether some of our previous conclusions on prioritization hold. More precisely, we focus on prioritization techniques at the function level and their ability to improve rate of fault detection. In addition, we explore instances (extreme in some cases) of the techniques' behavior that were not previously visible, which provide us with additional information on their strengths and weaknesses.

Our case studies evaluate prioritization techniques by adapting the "baseline" comparison method described in [12], [22]. This method is meant to compare a newly proposed technique against current practice, which is used as a baseline. In our case studies, assuming that no particular form of prioritization constitutes typical practice, we consider the random technique the baseline against which other techniques are compared.

There is, however, one aspect in which our studies differ from a "typical" baseline study. In our study, although we do not control the evolution of the programs studied, we can execute multiple techniques on the same version of the same program. In other words, we are studying programs that evolve naturally, but we can control (and replicate) the execution of prioritization techniques and evaluate their impact based on the data we collected from the evolution of those programs.[12] Still, there are several uncontrolled factors that constrain these studies and the aspects of the problem that we can address. We now explain the variables involved and the level of control we had over them.

To minimize the misinterpretation of the results that might occur due to specific types or amounts of change in any particular version, we perform our analysis on several versions in each case study. Confounding factors associated

11. Test case prioritization, in the context in which we consider it, is concerned only with ordering existing test cases; as such, it cannot improve detection of faults not detectable by those existing test cases. A well-rounded regression testing process should include activities aimed at finding faults not detectable by existing tests—such as faults related to new functionality not previously tested. We discuss this further in Section 7.

12. From that perspective, our studies have elements of the software engineering validation models classified as "dynamic analysis and legacy systems" in [35].

with the testing process are not fully controlled. First, we do not control (and do not know) the test generation process employed for QTB. In addition, we have only one test suite in each case study, which may limit our ability to determine whether differences in APFD are due to the techniques or to test suite composition. A similar situation is presented by the faults in the software. Faults were seeded in grep and flex by students not extensively familiar with the application domains, but QTB was used with its original faults. Finally, all the case studies assume that the software development and testing processes remained constant throughout the program evolution.

We investigated eight techniques over each of the units of study. The techniques employed were: random, optimal, fn-total, fn-addtl, fn-fi-total, fn-fi-addtl, fn-diff-total, and fn-diff-addtl. (In other words, we used all techniques not involving statement level instrumentation or FEP estimation. We excluded the former because we did not have statement-level coverage information for QTB and excluded the latter because performing the mutation analysis necessary to estimate FEP for these programs was not feasible.) However, there were two differences involving these techniques due to characteristics of the program data.

First, we obtained the APFD for random by averaging the APFD of 20 random orderings. This differs from the controlled study in which only one ordering per cell was generated. However, in a case study with a much smaller set of observations, we required an "average" random case to avoid extreme instances that could bias our evaluation. Second, the prioritization techniques based on fault proneness that we applied to QTB differed slightly from those used in our controlled experiments and our studies on flex and grep. The DIFF-based technique utilized produced just a binary value indicating whether a function changed or not between versions. The FI technique utilized on QTB used a subset of the metrics incorporated into the FI metric used in previous experiments. These differences might cause the resulting techniques to be less sensitive to modifications in the versions. Nevertheless, for simplicity and despite these differences, in this study, we continue to use the nomenclature used to denote these techniques in earlier studies.

6.3 Evidence Analysis

Fig. 5 provides an overview of the data for the three case studies. We include two graphs for each of the programs studied; these graphs provide complementary information. The box plots on the left present the overall distribution of APFD data per technique, summarized across all versions. This depiction illustrates each techniques' mean and variation, allowing comparisons of overall performance across all versions. The graphs on the right present the APFD values achieved by each of the techniques across each of the versions, allowing comparisons on a per version basis.[13]

We consider overall results (box plots) first. On both grep and flex, in terms of mean APFD, optimal ranks first, fn-addtl ranks second, and fn-fi-addtl ranks third. On both programs, techniques using feedback (addtl) produce APFDs closer to optimal than do techniques not using feedback (total). On QTB, in contrast, the average APFD for techniques using feedback exceeds the average APFD for techniques not using feedback. Further, on grep and flex, techniques using feedback exhibited less variance in APFD than those not using feedback, whereas, on QTB, this relationship was reversed. Another surprise was the high mean APFD value exhibited by the random technique on grep and flex. On QTB, the random technique outperforms the other techniques in some cases (evident in the extents of the tails of the distributions), but, in terms of mean APFD, it is the worst performing technique overall.

The data presented in the graphs of per version results (Fig. 5) also contains several surprises. It seems that the primary constant across different programs is the high degree of change in APFD values across versions. Furthermore, from the figures, it is difficult to understand the "contradictions" that are present in the data. However, when each specific scenario is analyzed in detail, a clearer picture emerges.

We conjecture that the variability in the results observed in these case studies can be attributed, at least in part, to the location of faults in the program and the likelihood that those faults are executed by the test cases in the test suite. (This conjecture is based, in part, on results presented in [32].) We observe that fault location and test coverage patterns varied widely across our programs and versions and this may have contributed to the variability in our results. To investigate this conjecture, we need to understand those factors within each unit of study. Table 16 summarizes relations between faults, fault exposure, and test coverage for each of the programs and versions studied, listing data on the percentage of functions executed by test cases, the percentage of functions executed by fault exposing test cases, the percentage of test cases executing faulty functions, and the percentage of test cases exposing faults.

First, we consider why techniques using feedback did not perform (overall) as well on QTB as on the other two programs. Observe, on the per version graph for QTB, that techniques using feedback performed better than those not using feedback on versions 1, 3, and 4, slightly worse on version 5, and considerably worse on version 2. With only five versions, the influence of one poor performance (version 2) is sufficient to affect the overall rankings of means exhibited across all versions of the program.

To suggest why version 2 exhibited such results, we turn to the data in Table 16. As the table shows, on version 2 of QTB, 98 percent of the test cases for the system execute the faulty function, but only one of those test cases exposes the fault. Also, consider version 4 of grep. Here, as on version 2 of QTB, most of the program's test cases (99.02 percent) execute the faulty function and few of these test cases (only two, or 0.33 percent of the test suite) expose the fault. Despite this similarity, however, these two cases result in different relationships between techniques using and not using feedback.

This difference may be attributed to differences in test case execution patterns. On version 2 of QTB, the test cases

13. Each technique has one value for each version within each program. These values have been connected with lines to facilitate the visualization of patterns.

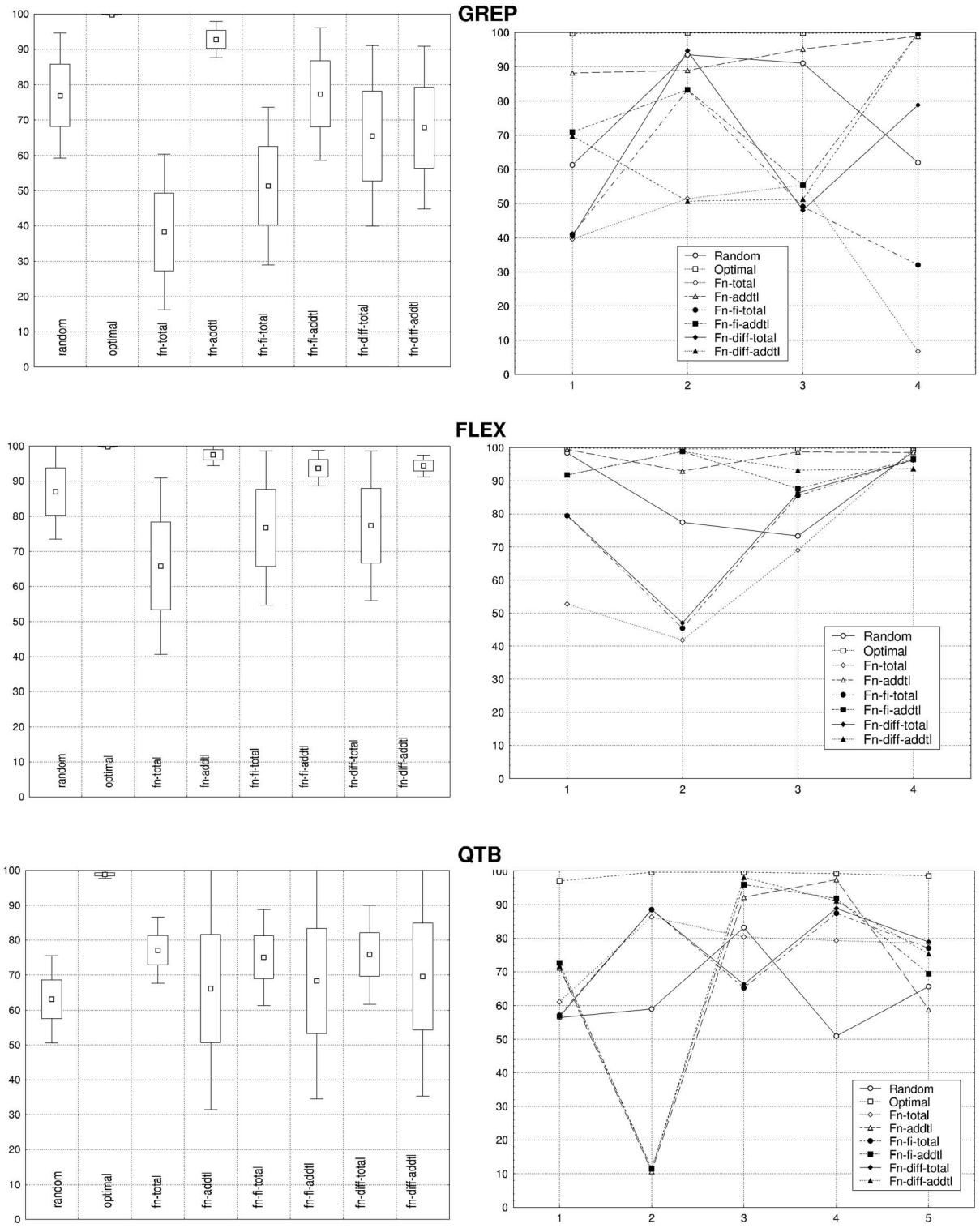


Fig. 5. Overview of case study data. Vertical axes depict APFD values. At left, box plots present the overall distribution of APFD data per technique, summarized across all program versions. At right, graphs show the APFD values obtained by each technique on each version.

exposing the fault execute a larger percentage of functions (37.57 percent) than the average test case (22.91 percent). On version 4 of `grep`, in contrast, the test cases exposing the fault execute a smaller percentage of functions (39.38 percent) than the average test case (46.8 percent).

When test cases that expose faults execute a relatively small percentage of functions, they are likely to be scheduled near the end of test execution by techniques not using feedback (e.g., fn-total). When test cases that expose faults execute a larger percentage of functions, they are likely to be scheduled near the end of test execution by techniques *using* feedback (e.g., fn-addtl). For faults exposed by a small percentage of the test cases that reach them, the postponing of such test cases further postpones

TABLE 16
Fault Exposure and Test Activity Data

Program	Version	Faulty functions	Exposed faults	Pct. of functions executed by tests		Pct. of functions executed by fault exposing tests		Pct. of tests executing faulty functions		Pct. of tests exposing faults	
				Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev
grep	1	4	4	44.22	5.25	41.89	4.25	64.19	46.75	0.25	0.1
	2	2	3	45.64	5.27	42.68	3.77	93.31	8.30	3.37	2.21
	3	3	3	46.82	5.39	45.07	5.65	85.80	23.60	2.29	0.93
	4	1	1	46.80	5.39	39.38	0.48	99.02	0.00	0.33	0
flex	1	4	5	52.52	6.20	59.39	4.46	73.33	38.73	13.29	5.46
	2	4	4	58.37	6.50	63.18	12.38	49.81	57.08	1.29	1.46
	3	6	7	58.37	6.50	66.61	4.85	52.77	51.66	7.48	8.55
	4	1	1	58.37	6.50	72.04	3.60	98.48	0.00	17.52	0
QTB	1	10	8	22.91	10.24	22.99	10.77	74.35	32.42	1.02	0.38
	2	1	1	22.91	10.24	37.57	0.00	97.78	0.00	0.74	0
	3	2	2	22.91	10.24	27.97	6.94	36.67	48.71	1.11	0.52
	4	2	1	22.91	10.24	36.45	10.37	33.33	0.00	1.11	0.52
	5	7	4	22.91	10.24	34.19	7.49	48.64	31.98	1.11	0.43

the exposure of those faults, exacerbating the differences in APFD values achieved by the prioritization techniques.

Summarizing, characteristics involving the coverage achieved by the test suite and the location of the faults affect the results of prioritization techniques using and not using feedback. In our case studies, where each version has significantly different types and locations of faults and test execution patterns with relationship to those faults differ widely, the tests exposing faults change and so does the effectiveness of the techniques across versions.

Next, we consider the situations in which random performs better than some of our prioritization heuristics, by considering differences in the relationship between random and fn-total on versions 1 and 4 of `flex`. (On version 1, random outperforms fn-total; on version 4, the two are nearly equivalent.)

Intuitively, random could be expected to perform well when the chances of exposing a fault with an arbitrary test case are high. Versions 1 and 4 of `flex` reflect this expectation. On both of these versions (for which over 13.29 percent and 17.52 percent of the test cases, respectively, expose faults), random prioritization produces APFD values relatively close to the optimal values. On version 4, this means that it is likely that one of the first six test cases randomly selected will expose the fault in that version. On version 4, however, a relatively large percentage of functions (72.83 percent) are executed by fault exposing test cases and most test cases (98.48 percent) execute faulty functions, rendering it probable that fn-total will also perform well. On version 1, in contrast, a smaller percentage of functions (59.76 percent) are executed by fault exposing test cases and fewer test cases (73.33 percent) execute faulty functions. In this case, the probability that fn-total will postpone execution of fault exposing functions is increased; a random rule thus performs better.

Similar cases can be seen on versions 2 and 3 of `grep`. On the other hand, when faulty functions are not likely to be executed and faults are not likely to be exposed by arbitrary test cases, the random technique performs poorly. For example, on version 1 of `grep`, the likelihood of exposing a fault is very small (0.25 percent), so random performed poorly. A similar situation can be found on version 3 of `flex`, where some faults have a very small probability of being exposed (as suggested by the high standard deviation).

Finally, we were surprised that techniques using fault proneness estimates did not provide more substantial improvements. Although, in many specific instances, incorporation of fault proneness estimates added significant improvements to techniques, the mean APFDs of these techniques across versions is not favorable. A previous study [11] of the FI fault index supported our expectations for techniques using fault proneness estimates; however, that previous study evaluated the predictive abilities of fault proneness indexes, whereas the study reported here evaluates techniques that employ those indexes to schedule test cases. In addition, there are other factors, such as test exposure capability, program domain, particular processes, and technique scalability, that may have not been relevant in the earlier studies, but could have had a significant impact on the fault prediction procedure [23] and on the FI-based prioritization techniques' effectiveness. These limitations might be contributing to some of the differences that we observe across the case studies.[14]

This said, FI-based techniques were observed to result in improved APFD values in our controlled experiments; thus, the difference in results exhibited in these case studies is of interest. We can suggest at least three things that may explain these differences. First, in our controlled experiments, the ratio between the amount of code changed and the number of faults in the program versions utilized was much smaller than in these case studies. The number of lines of code changed in the controlled experiments numbered in the tens, while the average number of changes in the case studies numbered in the hundreds. Since "regression" fault proneness metrics associate evolutionary changes with fault likelihood, they are likely to be more effective when fewer changes are made. Second, the test

14. Repeatability problems such as this, where different studies yield different results, are not unique to testing. For example, Lanubile et al. [23] report that even successful fault proneness prediction models might not work on every data set and that there is a need to take into consideration the context in which they are used.

suites used in the case studies had different characteristics than those used in the controlled experiments. We have suggested ways in which test suite characteristics can impact techniques using and not using feedback differently and the same suggestions apply to techniques employing measures of fault proneness. Third, the fault seeding process used on some objects (small Siemens programs, `grep`, and `flex`) could have played a role in the variance we observed among the techniques' performance, especially on the techniques based on fault proneness. Although we attempted to perform this process as consistently as possible, we recognize that it constitutes an artificial procedure that might not provide an accurate reflection of reality.

It is important to note that the previous interpretations are not always as transparent as presented. For example, in the presence of multiple faults, some of which are exposed by a large number of test cases and some of which are infrequently exposed, interpretation becomes more difficult and results less predictable. Nevertheless, these general patterns can be observed repeatedly. It can also be observed that version specific prioritization can (and most often does) yield considerable gains over random test case orderings and, if we select the proper prioritization technique, those gains can be maximized.

7 Cost-Benefits Analysis

Our results show that there can be statistically significant differences in the rates of fault detection produced by various test case prioritization techniques. But, what practical significance, if any, might attach to statistically significant differences in APFD values such as those we have measured, particularly when those differences are small? In this section, we investigate the practical implications of differences in APFD values and the trade-offs that need to be considered when comparing or selecting from prioritization techniques.

In general, where improving the rate of fault detection is the goal, the decision to use a certain prioritization technique depends on the benefits of discovering faults sooner versus the cost of the technique itself. If the cost of prioritizing with a given technique surpasses the savings generated by the higher rate of fault detection, then the technique is not worth employing.

Further, from a cost-benefits perspective, a technique A is superior to a technique B only if the additional gains achieved by A with respect to the gains achieved by B are greater than the additional costs of using A with respect to the costs of using B. To evaluate the relative cost-benefits of one technique compared with another, we must quantify both the savings generated by increases in the rate of fault detection and the costs of both techniques.

One procedure for savings quantification is to translate each APFD percentage point to a meaningful value scale (e.g., dollars) based on the assessment of the benefits (e.g., faster feedback to developers, earlier evidence that quality goals were not met, value of ensuring that test cases that offer the greatest fault detection ability will have been executed if testing is halted) that the new test prioritization scheme brings to the organization. Then, the comparison of the techniques' performances could be expressed not only in terms of APFD, but also in terms of their economic impact to the testing organization.

We do not possess data for our objects of study that would allow us to provide meaningful value scales for those objects. For example, we do not have execution times for `QTB` tests and, although we could gather execution time for tests of `grep` and `flex`, these do not include validation time. Moreover, our techniques are prototypes, not implemented for efficiency, so measurements of their runtime would not be indicative of the potential runtimes of such techniques in practice. Finally, even if we possessed appropriate cost-benefits data, analyses based on that data would be specific to that data and, though interesting, such analyses would allow us to examine only a limited range of trade-offs.

However, we can still investigate cost-benefits trade-offs and, in fact, do so more generally than the use of specific data would allow. To do this, we first simulate different savings factors that establish relationships between an APFD percentage point and a savings scale. A savings factor (SF) is a weight that translates an APFD percentage point into a measure of benefit. The greater the SF, the greater the benefits generated by an increased rate of fault detection. For example, if we choose to use dollars as our metric (under the assumption that savings in time are associated with savings in dollars through engineer salaries, accelerated business opportunities, etc.), then, under SF 5, a savings of 1 percent in APFD results in a savings of five dollars and, under SF 1000, a savings of 1 percent in APFD results in a savings of 1,000 dollars. (This is just an example; we could instead let our SF units represent person-months, or hundreds-of-dollars, or various other measures relevant to our costs and benefits.)

An SF is, in part, determined by the cost of executing a test suite: As that cost increases, the potential savings yielded by an increase in APFD also increase. Cost is partly a function of time; however, an SF may also take into account environmental factors, such as the availability of developers to take advantage of earlier feedback, the capability of the managers to use quality information, or the reliability expectations for the system. Thus, one situation that may produce a small SF is when test suite execution is fully automated and requires only a few hours of machine time. A second situation is when test suite duration is measured in days and potential gains of early fault detection are in days, but the developers are not able to employ feedback information because they are performing enhancements. On the other hand, if test suite duration is measured in days and managers are able to use test execution information to reset shipping dates before going public and suffering penalties, this may result in large SFs.

The use of savings factors and the following analysis provide further understanding of the circumstances under which a given prioritization technique could make, or fail to make, a practical difference.

In Fig. 6, we use seven savings factors (1, 5, 10, 50, 100, 500, 1,000) to associate differences in APFD values (x-axis) between one and 50 with the savings that can result from those differences under those savings factors (y-axis). It can

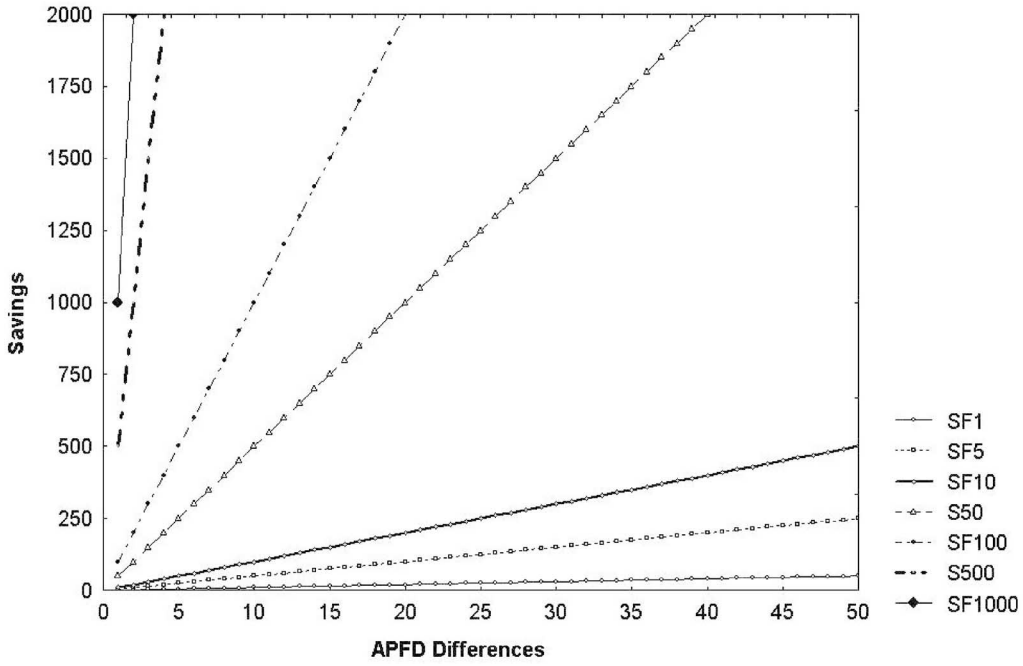


Fig. 6. Simulation of savings factors.

be observed from this figure that, under small SFs, even large differences in APFD values may not translate into practical savings. For example, if SF is measuring savings in dollars, then, at SF 1, a difference of 50 percent in APFD amounts to a savings of only 50 dollars and the same difference at SF 5 amounts to a savings of only 250 dollars. Such savings are so limited that even an optimal prioritization scheme is not likely to have a practical impact. On the other hand, under large SFs, even small APFD gains can be practically significant.

Let SF be the savings factor for a given environment, let A and B be prioritization techniques with costs C(A) and C(B), respectively, and let the APFDs of A and B be APFD(A) and APFD(B), respectively. The decision to use technique A rather than B can be framed as one of determining whether $C(A) - C(B) < SF * (APFD(A) - APFD(B))$. (Note that, by treating technique B in the foregoing as "current practice," we can frame the decision to use technique A rather than current practice.)

We now use the foregoing discussion to illustrate the potential practical significance, or lack thereof, of differences in APFD values such as those we observed in our studies. Columns 2 through 4 of Table 17 present the average APFD values for techniques fn-total and fn-fi-total and the differences between those average APFD values, for each of the eleven objects considered in our studies.[15] These two techniques exhibited different behavior across these objects and, although their exact costs are dependent on the implementation of the algorithms described in Section 3, it is certain that a careful implementation of fn-fi-total would be more expensive than a careful implementation of fn-total because the former technique requires all computations performed by the latter, plus additional work to compute and utilize fault index data.

Using these data, we would like to respond to two questions: 1) When would fn-fi-total be of greater benefit than fn-total and 2) when would those benefits matter to an organization? The first question can be answered by inserting values from Table 17 into the equation C(fn-fi-total) - C(fn-total) < SF $*$ (APFD(fn-fi-total) - APFD(fn-total)) for each of the objects. Even though we do not have specific cost values for the techniques (lefthand side), we can complete the righthand side to compute potential savings; this figure constitutes an upper bound on the differences in cost between both techniques. Table 17 shows the values that result, for each object, for each of the seven SFs shown in Fig. 6.

In the first six rows in Table 17, fn-total is greater than or equal to fn-fi-total and this results in negative potential savings for all SFs. Since the cost of fn-total is less than that of fn-fi-total, the superiority of fn-total holds trivially in these cases. In other cases, however, where fn-fi-total is greater than fn-total, we need to determine whether the difference in rate of fault detection can translate into savings that are greater than the additional cost incurred by fn-fi-total. Mapping the differences from Table 17 onto Fig. 6, we observe that, for `grep` and for a fixed SF, fn-fi-total is likely to provide greater gains than for any other program (13,000 with an SF of 1,000). In fact, for sufficiently large SF, fn-fi-total may be appropriate, even for subjects like `print_tokens`, which exhibit minimal differences in APFD (a 0.6 difference in APFD translates into 600 when SF is 1,000). If SF is small, however, even large APFD differences such as that observed with `grep` may not translate into savings (13 with an SF of 1).

Our second question, whether the benefits of using fn-fi-total rather than fn-total would matter to an organization, requires a somewhat subjective answer. Given that we can estimate the savings that result from a difference in

15. A similar comparison can be performed between any pair of techniques.

TABLE 17
Comparison of fn-total and fn-fi-total Techniques across All Subjects

	APFD			SF * (APFD(fn-total) - APFD(fn-fi-total))						
Object	fn-total	fn-fi-total	difference	1	5	10	50	100	500	1000
QTB	77.11	75.06	-2.1	-2.1	-10.5	-21	-105	-210	-1050	-2100
Print_tokens2	75.24	73.74	-1.5	-1.5	-7.5	-15	-75	-150	-750	-1500
Schedule2	72.08	71.69	-0.4	-0.4	-2	-4	-20	-40	-200	-400
Schedule	72.08	71.69	-0.4	-0.4	-2	-4	-20	-40	-200	-400
Replace	80.80	80.70	-0.1	-0.1	-0.5	-1	-5	-10	-50	-100
Tcas	84.03	84.03	0.0	0	0	0	0	0	0	0
Print_tokens	77.14	77.70	0.6	0.6	3	6	30	60	300	600
Space	93.18	94.96	1.8	1.8	9	18	90	180	900	1800
Tot_info	74.99	79.25	4.3	4.3	21.5	43	215	430	2150	4300
Flex	65.8	76.66	10.9	10.9	54.5	109	545	1090	5450	10900
Grep	38.29	51.32	13.0	13	65	130	650	1300	6500	13000

APFD, we would like to know what would trigger a company to invest in a new prioritization technique (or any prioritization technique at all)? Assume, for the sake of illustration, that the cost of fn-fi-total is effectively equal to the cost of fn-total. Again considering `grep`, if a company's estimated SF is 1,000, where SF is a measure in dollars, then a savings of 13,000 dolars could result from using fn-fi-total on `grep`. Whether such a savings would be considered worthwhile or trivial would depend on the organization. Considering the situation in a different way, given `print_tokens`, an SF of 1,000 would be required to achieve a savings of 600 dollars, whereas an SF of 50 would be sufficient to yield approximately the same savings on `grep`.

Note that there are additional cost-benefits trade-offs not accounted for by the foregoing analysis. For example, our cost-benefits model does not account for the fact that regression testing is performed repeatedly and that savings achieved through the use of a technique can be compounded over the lifetime of a system. Our model also assumes that a savings factor is linear; in reality, other functions (e.g., step functions, logarithmic functions) might be more appropriate. Such factors would need to be considered in adapting the model for use in specific application domains.

8 Related Work

Our conversations with practitioners suggest that, in practice, test engineers—faced with deadlines and excessively expensive test processes—have long applied measures of "relative value" to test cases. To date, however, there has been little mention in the research literature of test case prioritization.

Previous work by Rothermel et al. on prioritization, presented in [30], has been discussed in Section 1 of this article, where its relation to this work has been described, so we do not discuss it further here.[16]

In [2], Avritzer and Weyuker present techniques for generating test cases that apply to software that can be modeled by Markov chains, provided that operational profile data is available. Although the authors do not use the term "prioritization," their techniques generate test cases in an order that can cover a larger proportion of the software states most likely to be reached in the field earlier in testing, essentially, prioritizing the test cases in an order that increases the likelihood that faults more likely to be encountered in the field will be uncovered earlier in testing. The approach provides an example of the application of prioritization to the initial testing of software when test suites are not yet available.

In [34], Wong et al. suggest prioritizing test cases according to the criterion of "increasing cost per additional coverage." Although not explicitly stated by the authors, one possible goal of this prioritization is to reveal faults earlier in the testing process. The authors restrict their attention to "version-specific prioritization" and to prioritization of only the subset of test cases selected by a safe regression test selection technique from the test suite for the program. The authors do not specify a mechanism for prioritizing remaining test cases after full coverage has been achieved. The authors describe a case study in which they applied their technique to the `space` program that we used in the controlled experiments reported in this paper and evaluated the resulting test suites against 10 faulty versions of that program. They conclude that the technique was cost-effective in that application.

9 Summary and Conclusions

In this article, we have focused on the use of test case prioritization techniques in regression testing. Building on results presented in [30] and focusing on the goal of improving rate of fault detection, we have addressed several additional questions raised by that work: 1) Can prioritization techniques be effective when targeted at specific modified versions; 2) what trade-offs exist between fine granularity and coarse granularity prioritization techniques; 3) can the incorporation of measures of fault proneness into prioritization techniques improve their effectiveness? To address these questions, we have performed several new controlled experiments and case studies.

As we have discussed, these experiments and case studies, like any other, have several limitations to their validity. Keeping these limitations in mind, we draw

16. An additional paper, [29], is an earlier conference paper containing results subsumed by those in [30].

several observations from this work, with implications both for practitioners and for researchers.

First, our data and analysis indicate that version-specific test case prioritization can produce statistically significant improvements in the rate of fault detection of test suites. In our controlled studies on the Siemens programs and the larger space program, the heuristics that we examined always produced such improvements overall. In only a few cases did test suites produced by any heuristic not outperform randomly ordered test suites. Our case studies on flex, grep, and QTB, while admitting the possibility for greater variance in such results, illustrate the possibility for similar improvements.

The fact that similar results were observed for both function-level and statement-level techniques is important. The coarser analysis used by function-level techniques renders them less costly and less intrusive than statement level techniques. However, this same coarser level of analysis could also have caused a substantial loss in the effectiveness of these techniques, offsetting efficiency gains. Our results indicate, however, that, on average, function-level techniques were more similar in effectiveness to statement-level techniques than to random ordering and, thus, there could be benefits in using them.

Our investigation of incorporation of measures of fault proneness into prioritization showed that they, too, can (statistically significantly) improve the effectiveness of prioritization, but this improvement was comparatively (and in relation to our expectations, surprisingly) small and did not occur as consistently across our objects of study as did improvements associated with other techniques. This suggests that the benefits of incorporating such information may not be so obvious as intuition and previous successes with fault proneness estimates in other application areas, might lead us to believe.

Statistical significance, however, does not necessarily presage practical significance. As our cost-benefits analysis illustrates, neither the (numerically large) 32 percent difference in average APFD values observed for optimal and random in our controlled experiments nor the (numerically small) 1.25 percent difference in APFD values observed for fn-total and st-total in those experiments is a priori practically significant or insignificant. The practical impact of differences in APFD values depends on the many cost factors related to the expense of regression testing and prioritization processes. Certainly, smaller APFD differences require larger testing costs in order to produce practical differences, but, in practice, testing costs occur across a wide range and we believe that there exist testing processes (e.g., in relation to high-integrity software) in which expenses could justify even relatively small differences in APFD values.

To further complicate matters, both our controlled studies and our case studies suggest that the relative effectiveness of prioritization techniques can vary across programs. Our case studies illustrate that, for specific programs and modification patterns, it is possible for some techniques not to outperform random and the techniques that outperform random may vary. Moreover, our controlled and case studies show that the "best" technique to use may vary across programs.

The implication of these results is that test engineers should not assume that APFD gains will be practically significant nor should they assume that they will not. In the absence of measurement, practitioners who currently employ prioritization heuristics may be doing so to no avail and those who do not may be missing significant opportunities for savings. Then, the process of selecting the appropriate prioritization technique becomes of major interest as a topic for future research.

Our results suggest several avenues for future work. First, to address questions of whether these results generalize, further studies are necessary. Differences in the performance of the various prioritization techniques we have considered, however, also mandate further study of the factors that underlie the relative effectiveness of various techniques. To address these needs, we are gathering additional programs and constructing test suites for use in such studies. One additional desirable outcome of such studies would be techniques for predicting, for particular programs, types of test suites, and classes of modifications, which prioritization techniques would be most effective. We are also investigating alternative prioritization goals and alternative measures of prioritization effectiveness. Further, because a sizable performance gap remains between prioritization heuristics and optimal prioritization, we are investigating alternative prioritization techniques, including different methods for incorporating feedback in the use of fault-index-based techniques. Finally, we are working with our industrial collaborators to better quantify potential savings that can result from increases in rate of fault detection.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) Information Technology Research program under Awards CCR-0080898 and CCR-0080900 to the University of Nebraska, Lincoln, and Oregon State University, respectively. The work was also supported in part by NSF Awards CCR-9703108 and CCR-9707792 to Oregon State University and by a NASA-Epscor Space Grant Award to the University of Nebraska, Lincoln. Tom Ostrand shared the Siemens programs, faulty versions, and test cases. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos shared the space program and test cases. Roland Untch, Mary Jean Harrold, and Chengyun Chu contributed to earlier stages of the work. David Gable revised the techniques based on fault proneness and provided the tools used in DIFF-based prioritization. Adam Ashenfelter, Sean Callan, Dan Chirica, Hyunsook Do, Desiree Dunn, David Gable, Dalai Jin, Praveen Kallakuri, and Joe Ruthruff devoted weeks of often tedious time preparing materials for experimentation. Finally, we thank the anonymous reviewers for comments that substantially improved this paper. A preliminary version of this paper appeared in the *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 201-212, August, 2000.

REFERENCES

[1] IEEE Standards Association, *Software Engineering Standards,* vol. 3 of Std. 1061: Standard for Software Quality Methodology, IEEE, 1999 ed., 1999.

[2] A. Avritzer and E.J. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software," *IEEE Trans. Software Eng.,* vol. 21, no. 9, pp. 705–716, Sept. 1995.

[3] A.L. Baker, J.M. Bieman, N. Fenton, D.A. Gustafson, A. Melton, and R. Whitty, "Philosophy for Software Measurement," *J. System Software,* vol. 12, no. 3, pp. 277–281, 1990.

[4] M. Balcer, W. Hasling, and T. Ostrand, "Automatic Generation of Test Scripts from Formal Test Specifications," *Proc. Third Symp. Software Testing, Analysis, and Verification,* pp. 210–218, Dec. 1989.

[5] L.C. Briand, J. Wust, S.V. Ikonomovski, and H. Lounis, "Investigating Quality Factors in Object Oriented Designs: An Industrial Case Study," *Proc. Int'l. Conf. Software Eng.,* pp. 345–354, May 1999.

[6] M.E. Delamaro and J.C. Maldonado, "Proteum—A Tool for the Assessment of Test Adequacy for C Programs," *Proc. Conf. Performability in Computing Systems (PCS '96),* pp. 79–95, July 1996.

[7] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer,* vol. 11, no. 4, pp. 34–41, Apr. 1978.

[8] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," Technical Report 01-60-08, Oregon State Univ., May 2001.

[9] S.G. Elbaum and J.C. Munson, "A Standard for the Measurement of C Complexity Attributes," Technical Report TR-CS-98-02, Univ. of Idaho, Feb. 1998.

[10] S.G. Elbaum and J.C. Munson, "Code Churn: A Measure for Estimating the Impact of Code Change," *Proc. Int'l Conf. Software Maintenence,* pp. 24–31, Nov. 1998.

[11] S.G. Elbaum and J.C. Munson, "Software Evolution and the Code Fault Introduction Process," *Empirical Software Eng. J.,* vol. 4, no. 3, pp. 241–262, Sept. 1999.

[12] N. Fenton and L. Pfleeger, *Software Metrics–A Rigorous and Practical Approach,* second ed. Boston, PWS-Publishing, 1997.

[13] D. Gable and S. Elbaum, "Extension of Fault Proneness Techniques," Technical Report TRW-SW-2001-2, Univ. of Nebraska, Lincoln, Feb. 2001.

[14] T. Goradia, "Dynamic Impact Analysis: A Cost-Effective Technique to Enforce Error-Propagation," *Proc. ACM Int'l Symp. Software Testing and Analysis,* pp. 171–181, June 1993.

[15] R.G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Trans. Software Eng.,* vol. 3, no. 4, pp. 279–290, July 1977.

[16] R.G. Hamlet, "Probable Correctness Theory," *Information Processing Letters,* vol. 25, pp. 17–25, Apr. 1987.

[17] M.J. Harrold and G. Rothermel, "Aristotle: A System for Research on and Development of Program Analysis Based Tools," Technical Report OSU-CISRC- 3/97-TR17, Ohio State Univ., Mar. 1997.

[18] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. Int'l Conf. Software Eng.,* pp. 191–200, May 1994.

[19] R.A. Johnson and D.W. Wichorn, *Applied Multivariate Analysis,* third ed. Englewood Cliffs, N.J.: Prentice Hall, 1992.

[20] T.M. Khoshgoftaar and J.C. Munson, "Predicting Software Development Errors Using Complexity Metrics," *J. Selected Areas Comm.,* vol. 8, no. 2, pp. 253–261, Feb. 1990.

[21] R.E. Kirk, *Experimental Design: Procedures for the Behavioral Sciences,* third ed. Pacific Grove, Calif.: Brooks/Cole, 1995.

[22] B. Kitchenham, L. Pickard, and S. Pfleeger, "Case Studies for Method and Tool Evaluation," *IEEE Software,* vol. 11, no. 4, pp. 52–62, July 1995.

[23] F. Lanubile, A. Lonigro, and G. Visaggio, "Comparing Models for Identifying Fault-Prone Software Components," *Proc. Seventh Int'l Conf. Software Eng. and Knowledge Eng.,* pp. 312–319, June 1995.

[24] J.C. Munson, "Software Measurement: Problems and Practice," *Annals of Software Eng.,* vol. 1, no. 1, pp. 255–285, 1995.

[25] J. Musa, *Software Reliability Engineering.* New York: McGraw-Hill, 1998.

[26] A.P. Nikora and J.C. Munson, "Software Evolution and the Fault Process," *Proc. 23rd Ann. Software Eng. Workshop,* 1998.

[27] A.J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutation Operators," *ACM Trans. Software Eng. Methods,* vol. 5, no. 2, pp. 99–118, Apr. 1996.

[28] T.J. Ostrand and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Comm. ACM,* vol. 31, no. 6, June 1988.

[29] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Test Case Prioritization: An Empirical Study," *Proc. Int'l Conf. Software Maintenence,* pp. 179–188, Aug. 1999.

[30] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing," *IEEE Trans. Software Eng.,* vol. 27, no. 10, pp. 929-948, Oct. 2001.

[31] M.C. Thompson, D.J. Richardson, and L.A. Clarke, "An Information Flow Model of Fault Detection," *Proc. ACM Int'l Symp. Software Testing and Analysis,* pp. 182–192, June 1993.

[32] J. Voas, "PIE: A Dynamic Failure-Based Technique," *IEEE Trans. Software Eng.,* vol. 18, no. 8, pp. 717–727, Aug. 1992.

[33] F.I. Vokolos and P.G. Frankl, "Empirical Evaluation of the Textual Differencing Regression Testing Technique," *Proc. Int'l Conf. Software Maintenence,* pp. 44–53, Nov. 1998.

[34] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal, "A Study of Effective Regression Testing in Practice," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 230–238 Nov. 1997.

[35] M. Zelkowitz and D. Wallace, "Experimental Models for Validating Technology," *Computer,* vol. 31, no. 5, pp. 23–31, May 1998.

Sebastian Elbaum received the PhD and MS degrees in computer science from the University of Idaho and a degree in systems engineering from the Universidad Catolica de Cordoba, Argentina. He is an assistant professor in the Department of Computer Science and Engineering at the University of Nebraska, Lincoln. He has served on the program committees for the 2000 IEEE International Symposium on Software Reliability Engineering and the 2001 Workshop on Empirical Studies of Software Maintenance. His research interests include software measurement, testing, maintenance, and reliability. He is a member of the IEEE, IEEE Computer Society, IEEE Reliability Society, ACM, and ACM SIGSOFT.

Alexey G. Malishevsky is a PhD student and research assistant in the Department of Computer Science at Oregon State University where he received the MS and BS degrees. His research interests include regression testing and, in particular, prioritization of test suites and testability. He is a student member of the IEEE.

Gregg Rothermel received the PhD degree in computer science from Clemson University, the MS degree in computer science from the State University of New York, Albany, and the BA degree in philosophy from Reed College. He is currently an associate professor in the Computer Science Department at Oregon State University. His research interests include software engineering and program analysis, with emphases on the application of program analysis techniques to problems in software maintenance and testing and on empirical studies. His previous positions include vice president, quality assurance and quality control, Palette Systems, Inc. Dr. Rothermel is a recipient of the US National Science Foundation's Faculty Early Career Development Award and of the Oregon State University College of Engineering's Engelbrecht Young Faculty Award. He has served on the program committees for the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, the 2000 International Conference on Software Engineering, the 2001 International Conference on Software Engineering, the SIGSOFT 2000 Eighth International Symposium on the Foundations of Software Engineering, and the 2000 International Conference in Software Maintenance. He is a member of the IEEE, IEEE Computer Society, ACM, ACM SIGSOFT, and ACM SIGPLAN.