Faster Mutation Testing Inspired by Test Prioritization and Reduction

Lingming Zhang[1], Darko Marinov[2], Sarfraz Khurshid[1]

[1]Electrical and Computer Engineering, University of Texas, Austin, TX 78712, USA
zhanglm@utexas.edu, khurshid@ece.utexas.edu
[2]Department of Computer Science, University of Illinois, Urbana, IL 61801, USA
marinov@illinois.edu

ABSTRACT

Mutation testing is a well-known but costly approach for determining test adequacy. The central idea behind the approach is to generate *mutants*, which are small syntactic transformations of the program under test, and then to measure for a given test suite how many mutants it *kills*. A test t is said to kill a mutant m of program p if the output of t on m is different from the output of t on p. The effectiveness of mutation testing in determining the quality of a test suite relies on the ability to apply it using a large number of mutants. However, running many tests against many mutants is time consuming. We present a family of techniques to reduce the cost of mutation testing by prioritizing and reducing tests to more quickly determine the sets of killed and non-killed mutants. Experimental results show the effectiveness and efficiency of our techniques.

Categories and Subject Descriptors

D2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

Keywords

Mutation testing, Test Prioritization, Test Reduction

1. INTRODUCTION

Test adequacy, i.e., the problem of evaluating the quality of a test suite, plays a central role in software testing. Researchers have developed a number of approaches for measuring test adequacy; Zhu et al. [40] present a comprehensive survey. Among these approaches, mutation testing [2, 3, 8, 12] is often considered the most effective approach. For example, mutation testing is used in numerous research studies to evaluate testing techniques; Jia and Harman [19] present a survey of mutation testing.

The central idea behind mutation testing is to generate *mutants*, each of which contains a small syntactic change to the program under test, and then to determine for a given test suite how many mutants it *kills*. A test t is said to kill a mutant m of program p if the output of t on m is different from the output of t on p. The ratio of the number of mutants killed to the total number of non-equivalent mutants generated is termed the *mutation score*. A test suite with a higher mutation score is of a higher quality. In the limit, a test suite that kills all non-equivalent mutants is 100% adequate.

The key insight into the effectiveness of mutation testing is that a test suite that kills a large number of mutants likely also finds a large number of real faults [2, 8, 12, 19]—even when the mutants are not the same as the real faults. This effectiveness of mutation testing relies on the ability to apply it on a *large* number of mutants. These mutants are generated systematically using *mutation operators*, e.g., to replace an integer constant with 0. This paper focuses on traditional, first-order mutation testing that applies only one operator to generate a mutant; higher-order mutation testing [14, 18] can apply multiple operators to generate a smaller, but still large, number of mutants.

While mutation testing is very effective for evaluating test suite quality, it is also very *expensive* because it requires running many tests against many mutants. For each mutant that can be killed, we potentially run *several* tests that do not kill the mutant until we run one test that does kill the mutant. For each mutant that is not killed, we must run *every* test (that reaches the mutated statement). The cost of mutation testing can be measured in terms of the test-mutant pairs that are run. One way to reduce the cost is to reduce the number of mutants by selective mutation testing [4, 25, 26, 34, 37]. In contrast, our work reduces mutation testing cost in an orthogonal way: we aim to reduce the cost of executing tests for each mutant.

This paper presents *Faster Mutation Testing* (FaMT) to reduce mutation testing cost. FaMT includes novel techniques to prioritize and reduce tests for each mutant. FaMT prioritization and reduction are inspired by *regression test prioritization* [10, 29, 36] and *reduction* [5, 6, 13, 15], which prioritize and reduce tests for regression testing [35]. However, while both FaMT reduction and regression test reduction reduce tests, FaMT reduction does not preserve coverage of test requirements as regression reduction does. Test prioritization has been applied to mutation testing by our previous work, ReMT [39], but ReMT is a specialized technique that (1) only works for evolving code and (2) requires old mutation testing results on previous versions to prioritize tests. In this paper, we present the general FaMT approach to test prioritization for mutation testing which (1) works even for one code version and (2) does not require old mutation testing results. To our knowledge, test reduction has not been previously used for mutation testing.

The goal of our test prioritization is to reorder the tests such that a test that kills the mutant (when it can be killed) is run earlier. The goal of our test reduction is to run only a subset of tests on a mutant

ISSTA '13, July 15–20, 2013, Lugano, Switzerland

to determine that it is not killed if no test from this subset kills it. While test prioritization is *precise* in that it computes exactly the same mutation score as traditional mutation testing but does so faster, our test reduction is *approximate* in that it provides an underapproximation for the mutation score (because a test that was not run could kill a mutant even when no test that was run kills it).

To compute the bounds of mutation testing cost, consider a program $\mathcal{P}$ with a set of mutants $\mathcal{M}$ and a test suite $\mathcal{T}$. Let the number of mutants killed by $\mathcal{T}$ be m_K and the number of mutants not killed by $\mathcal{T}$ be m_N; $|\mathcal{M}| = m_K + m_N$. Let the total number of test executions for the killed mutants be t_K ($t_K \geq m_K$) and the total number of test executions for the non-killed mutants be t_N. The total cost of mutation is $t_K + t_N$. Any *precise* (dynamic) technique that reduces this cost can only reduce the number of test executions to kill the mutants because for each mutant not killed, all tests (that reach the mutant) must be run. An *oracular* technique could kill each mutant (that can be killed) by running only one test per mutant, and therefore has cost $m_K + t_N$. An *approximate* technique could, in principle, have cost 0, by running no test against any mutant, but it would not kill any mutant. Thus, a goal in optimizing mutation testing is to achieve higher precision with lower cost.

This paper makes the following contributions:

- **Idea:** We present the idea of test prioritization and reduction for mutation testing inspired by regression test prioritization and reduction. Our idea addresses both the key elements in the cost of mutation testing—executing some tests for killed mutants and executing every test for non-killed mutants.
- **Techniques:** We design and implement a family of techniques based on coverage information about test execution (e.g., the number of times the test reaches the mutated statement while executing on the original, unmutated program), on the history of test executions on other mutants (e.g., the accumulating number of mutants that the test killed or did not kill before executing the current mutant), and on the history granularity levels (e.g., history at the method level).
- **Evaluation:** We evaluate all 352 variant techniques of FaMT prioritization and 3872 variant techniques of FaMT reduction on 9 medium sized Java programs. The experimental results show that FaMT prioritization can reduce the number of *executions for killed mutants* up to 47.52% over randomized test execution orders (while providing the precise mutation score). The study also shows that FaMT reduction can reduce the number of *executions for all mutants* around 50.0% (while under-approximating the mutation score by only 0.50%). Finally, the study shows that FaMT techniques incur very small runtime overhead.

2. EXAMPLE

This section illustrates our FaMT techniques using the following sample code snippet and its 4 mutants (m_1, m_2, m_3, m_4):

```
1  int abs(int x) {
2    int y = 0;
3    if (x < 0)
4      y = -x;
5    if (x < 0) {//m1:if(y < 0) //m2:if(x <= 0)
6      return y;
7    } else {
8      return x;//m3:return 0; //m4:return -x;
9    }
10 }
```

Note that each mutant is defined by exactly one change to the program, e.g., m_1 replaces the variable `x` with `y` at line 5. In the actual mutation testing, there would be many more mutants even for this simple code, but we use only 4 mutants for ease of exposition. Consider further the following 4 tests for this code:

```
1 assert abs(0) == 0;  // reach: m1, m2, m3, m4
2 assert abs(1) == 1;  // reach: m1, m2, m3, m4
3 assert abs(-1) == 1; // reach: m1, m2
4 assert abs(-4) == 4; // reach: m1, m2
```

The goal of mutation testing is to determine the mutation score for these 4 tests on these 4 mutants. A naïve approach would run all 4 tests on all 4 mutants to determine the mutation score. However, not all these 16 runs are necessary. Traditional mutation testing employs two optimizations. First, it is unnecessary to run tests on a mutant after it gets killed. Second, it is unnecessary to run tests that do not even reach the mutated statement when these tests are run on the original, unmutated program [1,21,31,39]. For our example tests, the comments show this reachability information.

Table 1: Traditional mutation testing

m_1		m_2		m_3		m_4		Total runs of test-mutant pairs
t_1	N	t_1	N	t_1	N	t_1	N	
t_2	N	t_2	N	t_2	K	t_2	K	
t_3	K	t_3	N					11
t_4	-	t_4	N					

Table 2: FaMT test prioritization

m_1		m_2		m_3		m_4		Total runs of test-mutant pairs
t_3	K	t_3	N	t_1	N	t_2	K	
t_4	-	t_4	N	t_2	K	t_1	-	
t_1	-	t_1	N					8
t_2	-	t_2	N					

Table 3: FaMT test reduction

m_1		m_2		m_3		m_4		Total runs of test-mutant pairs
t_3	K	t_3	N	t_1	N	t_2	K	
t_4	-	t_4	N	t_2	-	t_1	-	
t_1	-	t_1	-					5
t_2	-	t_2	-					

Table 1 shows the 11 test-mutant pairs that the traditional mutation runs with these two optimizations. The cells indicate the following: 'N' that the test was run but did not kill the mutant, 'K' that the test was run and did kill the mutant, and '-' that the test was not even run for that mutant, e.g., t_4 is not run for m_1 because m_1 has been killed by t_3 before t_4. Note that t_3 and t_4 are blank for m_3 and m_4 because they cannot reach the mutated statement.

FaMT improves on the optimized traditional mutation testing in two ways. First, FaMT uses test prioritization to reorder the tests that reach each mutant. (Section 3.3 presents the algorithm in detail.) Intuitively, our prioritization uses the dynamic information from test runs on the original program (recall that the tests are already run to find the reachability information) and on the history of test runs on other mutants executed before the current mutant.

Table 2 shows the 8 test-mutant pairs that FaMT runs for our example. For each mutant, the table also shows how FaMT orders the tests for that mutant and the result of test runs. Note that FaMT can run the tests in different orders for different mutants. For m_1, FaMT orders t_3 and t_4 before t_1 and t_2 because t_3 and t_4 execute more instructions before reaching the mutated statement on the original program. (Section 3 describes the rationale for this choice and the additional information that FaMT uses.) For m_4, FaMT orders t_2 before t_1 because the execution history before m_4 (i.e., from m_1 to m_3) shows that t_2 is a "better killer"; t_2 killed m_3, whereas t_1 did not kill any mutant on which it was run before executing on m_4. In sum, the reordering that FaMT makes allows it to run only 8 test-mutant pairs (i.e., a reduction of 27.3%) and still produce the precise mutation score (3 of 4 mutants are killed).

Second, FaMT can further use test reduction to reduce the number of test-mutant pairs run. Intuitively, the reduction runs only a subset of the tests that reach a mutant; if none of these tests kills

the mutant, FaMT considers that the mutant cannot be killed by the given test suite. (Section 3.4 discusses reduction techniques that FaMT can use.) To illustrate, consider that FaMT runs at most 50% of the tests that reach the mutant, where 50% is just one example value; the user can set any ratio here, and our empirical study provides guidelines to choose proper ratios.

Table 3 shows the 5 test-mutant pairs that FaMT runs for our example. Note that different from FaMT prioritization, after t_3 and t_4 are executed on m_2, FaMT directly predicts that m_2 cannot be killed because 50% of the tests that reach m_2 have been executed. However, test reduction is *approximate* and provides an underapproximation for the mutation score because a test that was not executed could kill a mutant even when no executed test kills it. For example, after t_1 is executed on m_3, t_2 will not be executed on m_3. Therefore, FaMT reduction imprecisely predicts that m_3 cannot be killed while actually it can be killed by t_1. In brief, using reduction allows FaMT to run only 5 test-mutant pairs (i.e., a reduction of 54.5%) but produces an imprecise mutation score (2/4 as opposed to the actual score of 3/4). Please note that our experimental study demonstrates that the underapproximations for real-world programs are much smaller than this example.

3. APPROACH

This section presents our FaMT approach for faster mutation testing. We first describe FaMT basics, including initial test ordering (Section 3.1) and adaptive test ordering (Section 3.2). We then present test prioritization that FaMT performs to more quickly compute the precise mutation score (Section 3.3), and test reduction that FaMT performs to more quickly compute an approximate mutation score (Section 3.4). Recall that the cost of mutation testing has two key elements—running some tests for killed mutants and running every test that reaches the mutant for non-killed mutants. While FaMT prioritization addresses only the first element and calculates a precise mutation score, FaMT reduction addresses both elements but calculates an approximate mutation score.

3.1 Coverage-Based Initial Test Ordering

For each mutant m, FaMT first calculates the initial priority values of the tests that execute the mutated statement using test coverage on the unmutated program version. This calculation uses two basic heuristics: (1) tests that execute the mutated statement more times have a higher probability to kill the mutant [39], and (2) tests that execute the mutated statement more closely to the test exit statement have a higher probability to propagate the mutated state to the end and kill the mutant.[1] We also use a third heuristic that combines these two.

Let t be a test for mutant m. Our first heuristic calculates the initial priority value of t for m as:

$$\text{C}_1(t, m) = \text{CovNum}(t, stat_m) \tag{1}$$

where $\text{CovNum}(t, stat_m)$ denotes the number of times that t covers $stat_m$ that is the mutated statement of m.

Our second heuristic calculates the initial priority value of t for m using the ratio of the number of statements executed by t before the *first* execution of $stat_m$ to the number of all statements executed by t:

$$\text{C}_2(t, m) = \frac{\text{CovBefore}(t, stat_m)}{\text{CovBefore}(t, stat_m) + \text{CovAfter}(t, stat_m)} \tag{2}$$

[1]Note that a test can cover the mutated statement multiple times; we measure the distance from the *first* execution of the mutated statement to the test exit statement.

where $\text{CovBefore}(t, stat_m)$ denotes the number of unique statements executed by t before the first execution of the mutated statement $stat_m$, and $\text{CovAfter}(t, stat_m)$ denotes the number of unique statements executed by t after the first execution of $stat_m$. The higher the value is, the closer $stat_m$ may be to the end of t.

Our third heuristic combines the first two and calculates the initial priority value of t for m:

$$\text{C}_3(t, m) = \text{C}_1(t, m) \times \text{C}_2(t, m) \tag{3}$$

3.2 Power-Based Adaptive Test Ordering

During the execution of the tests for a mutant, FaMT also collects on-the-fly history information to adaptively update the test execution order. The basic intuition is that a test that killed more mutants that are *close* to the current mutant has a higher likelihood to kill the current mutant. We refer to this likelihood of a test to kill the current mutant m as the *power* of a test with respect to m. Formally, we denote the mutation testing results as a matrix Matrix, where each cell $\text{Matrix}(t, m)$ denotes the execution result of t on m: K denotes that m is killed by t, and N denotes that m is executed but not killed by t. Matrix is initially empty, and eventually filled with N and K. We define the power of test t with respect to m as the ratio of the number of mutants in m's neighborhood (denoted as $\mathcal{N}_m$, and defined below) which are killed by t to the number of all those in the neighborhood that have been executed by t (whether or not they are killed by t):

$$\text{P}_1(t, m) = \frac{|\{m' \in \mathcal{N}_m | \text{Matrix}(t, m') = \text{K}\}|}{|\{m' \in \mathcal{N}_m | \text{Matrix}(t, m') \in \{\text{K}, \text{N}\}\}|} \tag{4}$$

Intuitively, the higher the ratio, the higher the likelihood that t kills m. Note that the history information used by FaMT is *not* the mutation testing information from *previous program versions* [39]. Instead, it is accumulating execution history of tests on other mutants that have been executed before the current mutants in the same mutation testing task.

Equation (4) calculates the power of a test by taking into account all the mutants which are in $\mathcal{N}_m$ and executed by t. However, the mutants that cannot be killed by any tests may unnecessarily lower the power of a test. Therefore, we propose another formula to calculate the power of a test by excluding the mutants that have not been killed by any test yet:

$$\text{P}_2(t, m) = \frac{|\{m' \in \mathcal{N}_m | \text{Matrix}(t, m') = \text{K}\}|}{|m' \in \mathcal{N}_m | \{\text{Kill}(m') \wedge \text{Matrix}(t, m') \in \{\text{K}, \text{N}\}\}|} \tag{5}$$

where $\text{Kill}(m')$ denotes whether m' has been killed by a test, i.e., $\text{Kill}(m') \Leftrightarrow \exists t, \text{Matrix}(t, m') = \text{K}$.

While we believe there are various ways to define the neighborhood among mutants, here we consider the mutants that share common program locations as neighbors. In particular, we define four levels of neighborhood, each of which can be used for calculating the power of a test with respect to a mutant:

- **Statement-level history**: FaMT groups all the mutants that occur on the same statement with m as $\mathcal{N}_m$, i.e., $\mathcal{N}_m = \{m' | stat_{m'} = stat_m\}$, where $stat_m$ is the statement on which m occurs.
- **Method-level history**: FaMT groups all the mutants within the same method with m as $\mathcal{N}_m$, i.e., $\mathcal{N}_m = \{m' | meth_{m'} = meth_m\}$, where $meth_m$ is the source method in which m occurs.
- **Class-level history**: FaMT groups all the mutants that occur in the same class with m as $\mathcal{N}_m$, i.e., $\mathcal{N}_m = \{m' | clas_{m'} = clas_m\}$, where $clas_m$ is the source class in which m occurs.

Algorithm 1: FaMT Prioritization Algorithm

```
   Input: Program P, mutants M, test suite T
   Output: MATRIX
1  begin
2      Initialize MATRIX as empty
       // Collect coverage information when executing T on P
3      COVNUM, COVBEFORE, COVAFTER ← COVCOLLECT(T, P)
4      for m ∈ M do
           // Detect tests that execute the mutated statement on P
5          T_m ← {t ∈ T | COVNUM(t, stat_m) > 0}
6          for t : T_m do
               // Note that the initial priority calculation is not
                  updated during mutation testing
7              Calculate C(t, m) according to Section 3.1
               // Note that the power calculation is continuously
                  updated during mutation testing
8              Calculate P(t, m) according to Section 3.2
           // Reorder T_m based on the initial priority values, C
9          T'_m ← REORDER(T_m, C)
           // Split T'_m into two lists by comparing the power
              values, P, with Threshold
10         T_1, T_2 ← PARTITION(T'_m, P, Threshold)
           // Iterate over the test list by concatenating T_1 and T_2
11         for t : T_1 ⊕ T_2 do
12             MATRIX(t, m) ← EXECUTE(t, m) // N or K
               // If mutant m is killed, continue to next mutant
13             if MATRIX(t, m) = K then break
       // Return the final mutation testing matrix
14     return MATRIX
```

- **Global history**: FaMT groups all the mutants as $\mathcal{N}_m$, i.e., $\mathcal{N}_m = \mathcal{M}$, where $\mathcal{M}$ is the set of all the mutants for the program under test.

Different levels of neighborhood enable FaMT to use different levels of history information. For each history level, FaMT utilizes the history information of a test t on the mutants that occur in the same neighborhood (e.g., in the same class) with the current mutant m to calculate the likelihood that t kills m. On one extreme, statement-level history calculates the test power based on mutants that are on the same statement because those mutants tend to perform similarly when being tested. However, the number of mutants that are on the same statement is usually too small for sampling. On the other extreme, global-level history records the test power for the entire program and may be imprecise for specific mutants, but the number of mutants is sufficiently large. Therefore, we investigate the impact of all four levels of history information.

3.3 Test Prioritization

The goal of our test prioritization technique is to reorder the tests such that a test that kills the mutant (when it can be killed) is run earlier than by simply following the default or random order of tests. Algorithm 1 gives the pseudo-code for our technique. The algorithm employs a family of ordering functions. These functions are based on coverage information of tests (Section 3.1) and on the accumulating execution history of tests (Section 3.2).

The algorithm takes program $\mathcal{P}$, its mutant set $\mathcal{M}$, and test suite $\mathcal{T}$ as inputs, and returns the mutation testing matrix MATRIX as output. Line 2 initializes MATRIX as empty. Line 3 collects coverage information which is used during later steps. Lines 4-13 iterate over the mutant set to determine whether each mutant is killed. During each iteration, Line 5 identifies the set of tests $\mathcal{T}_m$ that reach the mutated statement of current mutant m on the unmutated program, because the tests which do not reach the mutated statement cannot kill the mutant [1, 21, 31, 39]. Lines 6-8 iterate over all the tests in $\mathcal{T}_m$ and calculate the initial priority (Section 3.1) as well as power (Section 3.2) for each test with respect to m. Note that the initial test priorities are fixed during the process of mutation testing, because they are based on the coverage information of the tests on the unmutated program. However, the test powers are continuously updated during the mutation testing process: the more mutants in the neighborhood of m are executed, the higher the accuracy of the power values.

Line 9 reorders $\mathcal{T}_m$ according to the initial priorities of tests. Line 10 then partitions the reordered list into two sublists, $\mathcal{T}_1$ and $\mathcal{T}_2$, based on the power of a test: if the power is less than the `Threshold`, FaMT puts the test into $\mathcal{T}_2$; otherwise, FaMT puts the test into $\mathcal{T}_1$. Note that each sublist is still ordered by initial test priorities. Lines 11-13 concatenate $\mathcal{T}_1$ with $\mathcal{T}_2$ and iterate over the concatenated list. Line 12 executes mutant m on test t and puts its execution result into the resulting MATRIX: if t kills m, the execution result is `K`; otherwise, the result is `N`. Line 13 terminates the execution for current mutant and continues to the next mutant if the current mutant is killed. Finally, Line 14 returns the mutation testing matrix MATRIX as output and terminates the algorithm.

3.4 Test Reduction

Our test prioritization can reduce the number of executions to kill mutants, but for the mutants that cannot be killed, the test prioritization cannot help. The goal of our test reduction is to run only a subset of tests on a mutant to determine that it is not killed if no test from this subset kills it. In this way, we can reduce the number of executions for all the mutants, regardless of whether they are killed or not. Note that this reduction may cause the mutation testing result to be approximate because some mutant may be mistakenly predicted as not killable due to some tests that kill it not being executed. Therefore, this algorithm needs to be carefully evaluated through an empirical study.

The basic intuition of our test reduction is that if those tests with higher likelihood to kill a mutant cannot kill the mutant, the remaining tests will have little chance to kill the mutant. Recall that our test prioritization also executes first the tests that have higher likelihood to kill mutants. Therefore, we build our test reduction algorithm directly on our test prioritization algorithm (Algorithm 1). Our reduction modifies only Line 11 of Algorithm 1. While our prioritization algorithm always concatenates the entire $\mathcal{T}_1$ and $\mathcal{T}_2$, our reduction algorithm only concatenates $\mathcal{T}_1$ with a prefix of $\mathcal{T}_2$. More specifically, we change Line 11 into the following line to form our reduction algorithm:

$$\textbf{for } t : \mathcal{T}_1 \oplus \textsc{PreFix}(\mathcal{T}_2, \textsc{Max}(0, (|\mathcal{T}_1 \oplus \mathcal{T}_2|) \times \texttt{MinRatio} - |\mathcal{T}_1|))$$

where $\textsc{PreFix}(\mathcal{T}_2, x)$ returns the sublist which contains the first x tests in $\mathcal{T}_2$, and $\textsc{Max}(x, y)$ returns the larger of x or y. The reduction algorithm concatenates $\mathcal{T}_1$ with a prefix of $\mathcal{T}_2$ such that ratio of the concatenated list's length to the length of $\mathcal{T}_1 \oplus \mathcal{T}_2$ is at least `MinRatio`. Note that if the length of $\mathcal{T}_1$ is already larger than or equal to $(|\mathcal{T}_1 \oplus \mathcal{T}_2|) \times \texttt{MinRatio}$, no tests from $\mathcal{T}_2$ will be executed.

4. EXPERIMENTAL STUDY

FaMT aims to reduce the cost of mutation testing by prioritizing and reducing the tests that need to be executed for each mutant. To evaluate FaMT, we implement FaMT on top of Javalanche [31], a state-of-the-art mutation testing tool for Java.

Table 4: Subjects

Subject	Version	Size	#Tests	#Mutants(KillRates)
Time&Money	r207	2681	236	2304 (72.35/87.14)
Jaxen	r1346	13946	690	9880 (46.72/70.54)
Xml-Sec	v3.0	19796	84	9693 (26.41/70.93)
Com-Lang	r1040879	23355	1691	19746 (65.68/86.24)
JDepend	v2.9	2721	55	1173 (68.03/84.62)
Joda-Time	r1604	32892	3818	24174 (66.45/87.16)
JMeter	v1.0	36910	60	21896 (9.24/28.34)
Mime4J	v0.50	6954	120	19111 (23.10/63.39)
Barbecue	r87	5391	154	36418 (2.75/68.40)

4.1 Research Questions

Our experimental study addresses these research questions:

- **RQ1:** How does FaMT prioritization reduce the number of executions?
- **RQ2:** How does FaMT reduction reduce the number of executions and how it approximates the mutant killing ratio?
- **RQ3:** How does FaMT compare with regression test prioritization and reduction in the mutation testing scenario? (While regression test prioritization and reduction are not originally designed for mutation testing, they have a straightforward application to it—to compare with FaMT, we apply regression test prioritization and reduction to mutation testing by using the coverage information of the original program to uniformly prioritize and reduce tests across all the mutants.)
- **RQ4:** What are the runtime overheads for both the test prioritization and test reduction of FaMT?

4.2 Independent Variables

We used the following independent variables (IVs):

IV1: Different Initial Orderings. We considered different test ordering randomizations for each mutant (*DR*) and all our three coverage-based orderings (C_1, C_2, and C_3; Section 3.1).

IV2: Different Test Power Formulas. We considered both choices of test power formulas presented in Section 3.2: (1) using history of all neighbor mutants and (2) using history of only killed neighbor mutants. We denote them as P_1 and P_2, respectively.

IV3: Different History Information Levels. We considered all four levels of history information presented in Section 3.2: (1) statement level, (2) method level, (3) class level, and (4) global level. We denote them as *Stat*, *Meth*, *Clas*, and *Glob*, respectively.

IV4: Different Thresholds. We considered 11 `Threshold` values for Algorithm 1, ranging from 0.0 to 1.0 with increments of 0.1.

IV5: Different MinRatios. We considered 11 `MinRatio` values from Section 3.4, ranging from 0.0 to 1.0 with increments of 0.1.

IV6: Different Regression Testing Techniques. We considered the widely used *total* and *additional* regression test prioritization techniques using statement coverage [10, 29], and the widely used *greedy* regression test reduction technique using statement coverage [38], to evaluate their effectiveness for RQ3.

4.3 Dependent Variables

To evaluate the effectiveness and the efficiency of FaMT, we used the following three dependent variables (DVs):

DV1: Execution Reduction Ratio. This variable denotes the ratio of test-mutant executions reduced by FaMT prioritization or reduction to the number of all test-mutant executions that reach mutants.

DV2: Error Rate. This variable shows the ratio of mutants that are mistakenly predicted as unkillable by FaMT reduction, i.e., $err = \frac{|\mathcal{M}_e|}{|\mathcal{M}_r|}$, where $\mathcal{M}_e$ denotes the set of mistakenly predicted mutants and $\mathcal{M}_r$ denotes all the reached mutants.

DV3: Run Time Overhead. This variable records the runtime overheads incurred by FaMT prioritization or reduction. Specifically, we recorded all the extra setup costs for FaMT prioritization/reduction in comparison with Javalanche, including calculating the coverage-based heuristic and test power information, as well as prioritizing/reducing tests based on them.

4.4 Subjects and Experimental Setup

We evaluated FaMT using nine open-source projects which come from various application domains and have been widely used for mutation testing and regression testing research [30, 31, 36, 39]. Table 4 summarizes the projects. The sizes of the studied projects range from 2.6K lines of code (LoC) to 36.9KLoC (excluding blank lines and test code). Column 4 shows the number of tests for each subject. In the last column of Table 4, we show the number of all generated mutants, the ratio (%) of killed mutants to all the mutants, and the ratio (%) of killed mutants to the reached mutants.

We evaluate all the FaMT techniques on all subjects:

For FaMT prioritization, we studied all the combinations of 4 initial orderings, 2 choices of power calculation, 4 levels of history information, and 11 `Threshold` values, i.e., 4*2*4*11=352 prioritization variant techniques.

For FaMT reduction, we studied all the combinations of 4 initial orderings, 2 choices of power calculation, 4 levels of history information, 11 `Threshold` values, and 11 `MinRatio` values, i.e., 4*2*4*11*11=3872 reduction variant techniques.

As the accumulated history varies for different orders of mutant execution, we randomize the execution order for mutants and apply each FaMT prioritization or reduction technique on each subject for 20 times to evaluate its effectiveness as well as the stability. We also applied all other compared techniques for 20 times on each subject. The experiments were performed on a Dell desktop with Intel i7 8-Core 2.8GHz processor, 8G RAM, and Windows 7 Enterprise 64-bit version.

4.5 Result Analysis

All the detailed experimental data can be found online[2].

4.5.1 RQ1: FaMT Test Prioritization

Evaluation with default threshold. We applied FaMT prioritization techniques with the default `Threshold` of 0.3 on all the subjects. Table 5 shows the detailed experimental results for FaMT prioritization techniques using the history of all neighbor mutants (P_1). Column 1 (*A.*) and Column 2 (*I.*) show the levels of *adaptive* history information and the *initial* test orderings used. Columns 3-20 present the ratios of test executions for killed mutants reduced by the studied techniques (both mean values and standard deviations over 20 runs) compared with different randomized test orders for each mutant (*DR*). Column 21 presents the total execution reduction ratios (the ratio of the sum of all reduced executions to the sum of all executions over all subjects) by each technique. Similarly, Table 6 shows the experimental results of FaMT prioritization using the history of only killed neighbor mutants (P_2). We also compare the baseline *DR* with a more basic random technique, *UR*, which randomizes the entire original test suite and then reorders the tests for each mutant according to their ordering in the randomized original suite (first row in Table 5). The key difference between *UR* and *DR* is that *UR* uses the same random ordering for all mutants whereas *DR* uses different random orderings for different mutants. Our findings are as follows.

[2] `https://webspace.utexas.edu/~lz3548/isstal3support.html`

Table 5: Execution reduction (%) for FaMT prioritization with `Threshold`=0.3 and history of all neighbor mutants (P_1)

A.	I.	*Time&Money*		*Jaxen*		*Xml-Sec*		*Com-Lang*		*JDepend*		*Joda-Time*		*JMeter*		*Mime4J*		*Barbecue*		Total
		Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.
-	*UR*	0.8	9.39	-18.4	61.81	-2.2	18.24	-0.6	3.47	-2.5	21.75	-0.2	7.32	-0.2	3.75	3.9	8.26	2.9	10.15	-1.7
-	*DR*	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0
-	C_1	4.9	4.71	-45.8	60.44	2.0	5.87	2.7	1.43	-3.0	8.12	16.5	2.98	0.6	1.61	6.7	4.50	9.8	5.67	-4.6
-	C_2	0.0	6.19	26.3	27.97	-15.2	11.78	-7.5	1.75	-11.5	10.33	-0.0	4.24	-1.8	2.16	-11.0	4.98	-14.7	7.11	12.2
-	C_3	1.8	4.82	2.4	35.58	3.7	9.61	-1.1	1.60	7.0	8.31	19.5	3.39	-0.6	2.14	7.3	4.07	4.3	7.29	12.8
S	*DR*	14.2	3.28	15.7	5.85	19.1	6.40	8.1	0.79	17.4	2.32	14.5	1.39	4.1	0.82	16.6	1.81	25.0	2.72	13.7
t	C_1	18.4	5.06	-27.4	55.48	21.9	7.54	10.2	1.33	14.8	6.60	25.0	3.35	4.8	1.68	20.0	3.67	28.2	4.77	7.6
a	C_2	13.3	5.30	53.5	17.16	11.5	10.17	3.2	1.49	12.5	7.47	16.9	3.65	2.6	2.09	11.1	4.18	18.0	5.36	31.6
t	C_3	16.8	4.94	20.5	30.04	22.5	8.99	8.2	1.42	21.0	6.77	29.3	3.42	3.8	2.12	20.3	3.55	26.2	5.59	25.5
M	*DR*	18.0	4.19	20.1	8.49	26.9	8.22	9.7	0.91	22.5	3.17	23.1	1.65	6.4	1.55	18.9	2.86	32.6	3.69	18.9
e	C_1	20.7	5.03	-22.3	53.64	28.4	8.61	11.7	1.23	22.2	6.52	29.2	3.06	7.4	1.92	20.6	3.76	35.2	4.30	11.4
t	C_2	17.5	5.23	58.9	15.54	21.7	9.97	6.2	1.46	20.2	6.80	25.0	3.17	5.1	2.02	15.2	4.39	27.0	4.74	37.4
h	C_3	19.8	5.15	24.2	29.30	28.6	9.10	9.9	1.22	27.3	6.57	33.3	3.11	6.6	2.12	20.9	3.49	33.7	4.50	28.8
C	*DR*	17.4	3.57	14.1	5.33	27.2	9.42	8.5	0.97	22.7	8.36	28.2	2.51	6.4	1.47	17.1	3.23	25.9	3.51	18.4
l	C_1	18.3	4.82	-28.1	56.30	28.9	9.06	10.4	1.28	19.4	11.54	34.3	3.28	7.5	1.94	16.6	4.70	32.6	3.76	10.8
a	C_2	18.5	5.29	55.3	18.28	24.6	9.77	6.5	1.44	20.4	8.93	28.0	3.62	5.4	2.03	14.3	4.81	21.2	6.70	37.3
s	C_3	18.3	4.84	19.2	30.76	29.4	9.45	8.6	1.27	16.7	11.29	36.8	2.86	6.6	2.10	17.3	4.13	31.1	4.78	27.8
G	*DR*	14.1	3.04	2.2	1.49	23.2	8.32	7.7	1.17	23.3	5.99	29.4	2.67	4.7	1.61	15.1	4.07	18.8	5.76	14.1
l	C_1	15.2	4.91	-42.6	59.75	24.4	8.92	10.2	1.30	21.2	10.08	34.8	3.31	6.5	1.95	12.1	5.76	24.6	5.20	5.4
o	C_2	15.5	5.88	29.2	27.79	20.1	10.06	5.8	1.41	17.9	10.11	31.0	3.88	5.3	2.03	12.6	6.40	11.6	8.12	28.9
b	C_3	13.9	4.84	4.6	34.98	26.1	8.43	8.5	1.36	19.4	7.78	35.6	3.38	6.1	2.56	13.8	5.62	21.6	5.79	22.0

Table 6: Execution reduction (%) for FaMT prioritization with `Threshold`=0.3 and history of only killed neighbor mutants (P_2)

A.	I.	*Time&Money*		*Jaxen*		*Xml-Sec*		*Com-Lang*		*JDepend*		*Joda-Time*		*JMeter*		*Mime4J*		*Barbecue*		Total
		Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.	Dev.	Red.
S	*DR*	14.5	3.21	16.0	6.03	19.4	6.57	8.3	0.70	17.4	2.35	14.7	1.36	4.2	0.86	17.0	1.78	25.2	2.67	13.9
t	C_1	18.8	4.89	-27.3	55.32	22.4	7.57	10.4	1.32	14.8	6.56	24.9	3.26	4.9	1.73	20.0	3.66	28.3	4.72	7.6
a	C_2	13.7	5.38	53.3	17.30	11.9	10.12	3.3	1.49	12.5	7.41	17.1	3.64	2.6	2.11	11.4	4.21	18.1	5.34	31.6
t	C_3	17.2	4.91	20.5	30.24	23.0	9.00	8.4	1.39	21.0	6.85	29.3	3.44	3.8	2.14	20.4	3.56	26.2	5.58	25.5
M	*DR*	17.9	3.93	65.3	12.99	27.8	8.63	9.9	0.89	22.6	3.39	23.4	2.42	5.9	1.32	21.5	2.79	35.7	3.13	40.1
e	C_1	21.1	4.71	68.0	11.53	29.5	8.92	12.0	1.31	22.7	5.92	29.9	3.08	7.0	1.87	23.7	3.54	37.2	4.31	43.8
t	C_2	17.4	5.03	63.7	13.12	23.7	9.87	6.3	1.34	20.1	6.60	26.1	3.49	4.3	2.08	17.7	3.98	33.7	4.25	39.7
h	C_3	20.0	4.87	69.0	11.45	29.8	8.78	9.7	1.33	28.1	6.10	34.4	3.23	5.8	2.07	23.7	3.67	38.7	4.70	45.5
C	*DR*	15.6	3.63	68.4	12.84	27.5	8.92	8.3	0.89	23.3	6.04	26.8	2.21	6.0	1.45	20.5	2.89	36.3	3.41	42.2
l	C_1	17.1	4.80	66.3	13.33	28.4	9.25	10.7	1.46	24.7	7.84	33.3	2.76	7.0	1.86	23.9	4.15	38.9	4.46	44.3
a	C_2	16.5	5.35	68.1	12.62	26.1	9.47	5.7	1.45	23.2	7.89	30.2	3.51	4.3	1.95	15.6	3.87	33.1	4.74	42.6
s	C_3	16.9	4.85	67.4	13.01	28.8	9.47	8.8	1.42	29.4	5.55	38.3	2.66	5.4	2.03	23.9	3.61	38.6	4.80	46.2
G	*DR*	10.8	2.56	65.5	12.78	17.1	6.80	7.3	0.98	19.5	5.57	30.1	2.55	4.2	1.33	13.1	2.40	31.5	2.89	41.1
l	C_1	13.7	4.53	59.5	15.64	19.0	7.99	10.0	1.50	19.8	7.50	36.5	3.36	5.7	1.94	19.9	4.11	34.8	5.01	42.2
o	C_2	12.6	5.34	68.8	12.65	15.7	10.01	4.2	1.52	19.5	8.12	33.3	3.08	2.7	2.08	6.5	5.22	28.0	5.74	42.7
b	C_3	12.4	4.43	62.7	15.07	21.9	8.30	7.9	1.47	27.5	4.76	39.5	2.54	4.1	2.24	19.2	4.25	33.9	5.09	44.2

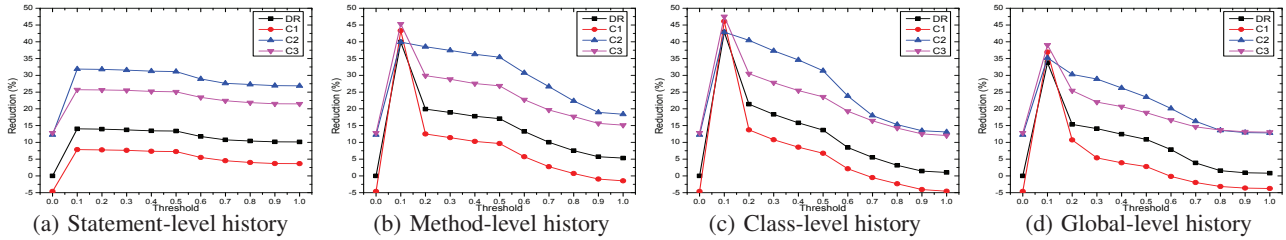
First, FaMT prioritization techniques that embody history information perform better than techniques without history information. For example, *UR*, C_1, C_2 and C_3 without history information reduce the number of executions by -4.6% to 12.8% in total (compared with *DR*). In contrast, all the techniques with history information can effectively reduce the number of executions by 5.4% (C_1 with global-level history and P_1 formula) to 46.2% (C_3 with class-level history and P_2 formula) in total. Another interesting finding is that history power information can even boost the *DR* test order to achieve high reduction ratios. For example, when using the class level history and P_2, even *DR* can reduce the number of executions by 42.2% in total. We also performed a statistical test to confirm the effectiveness of FaMT. Specifically, the Fisher's LSD test [33] shows that all FaMT techniques in Table 6, except the first two, outperform the *DR* random technique at the significance level of 0.05.

Second, for all levels of history information, using P_2 performs better than using P_1 in reducing executions. For example, for the method level history information, techniques using P_2 reduce the number of executions by 39.7% to 45.5%, while techniques using P_1 only reduce the number of executions by 11.4% to 37.4%. The reason is that the non-killable mutants may unnecessarily lower the power of tests and thus delay the execution of some *good* tests. The only exceptions are the techniques using statement-level history.

Third, for both P_1 and P_2, the techniques using the method or class level history information tend to perform the best. For example, FaMT techniques using method level history and P_1 formula reduce the number of executions by 11.4% to 37.4%, and FaMT techniques using class level history and P_2 formula reduce the number of executions by 42.2% to 46.2%. Interestingly, for those techniques, the best initial ordering within the same level of history information depends on using P_1 or P_2. For example, when using P_1, C_2 performs the best. On the contrary, when using P_2, C_3 performs the best.

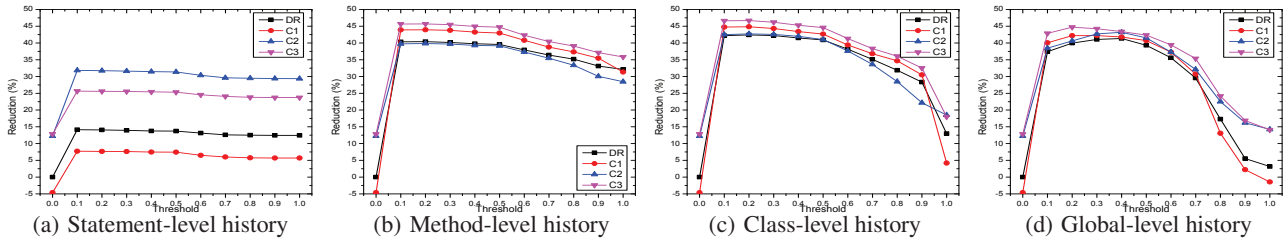
Effectiveness trends when using various thresholds. Figure 1 illustrates the trends for FaMT prioritization techniques with P_1 formula using different thresholds from 0.0 to 1.0. The four plots are for FaMT techniques using statement, method, class, and global levels of history, respectively. In each plot, each line represents using the initial ordering *DR*, C_1, C_2, or C_3. Similarly, Figure 2 illustrates the trends for FaMT prioritization techniques with P_2 formula using different thresholds.

First, when `Threshold`=0.1, almost all FaMT techniques achieve highest reduction ratios. For example, techniques using class-level history and P_1 formula reduce executions by 42.86% to 47.52%, and techniques using class-level history and P_2 formula reduce executions by 42.17% to 46.63%. The only exception is for the techniques using global-level history and P_2 formula, which tend to



(a) Statement-level history (b) Method-level history (c) Class-level history (d) Global-level history

Figure 1: Reduction trends on various `Threshold` values by FaMT prioritization using four levels of history and P_1 formula



(a) Statement-level history (b) Method-level history (c) Class-level history (d) Global-level history

Figure 2: Reduction trends on various `Threshold` values by FaMT prioritization using four levels of history and P_2 formula

Table 7: Execution reduction (%) for killed/all mutants by theoretical techniques and an FaMT prioritization technique.

Subject	Reduction for Killed Mutants			Reduction for All Mutants		
	T-Worst	T-Best	FaMT	T-Worst	T-Best	FaMT
Time&Money	-115.6	34.3	16.9	-66.3	19.8	9.8
Jaxen	-505.7	90.4	67.4	-27.3	5.9	4.6
Xml-Sec	-122.9	41.3	28.8	-28.2	9.8	6.9
Com-Lang	-94.3	24.4	8.8	-56.4	14.6	5.3
JDepend	-179.8	47.7	29.4	-75.6	20.3	12.5
Joda-Time	-981.8	66.6	38.3	-459.3	31.2	18.0
JMeter	-41.8	14.9	5.4	-11.2	4.0	1.5
Mime4J	-131.1	41.4	23.9	-41.0	13.0	7.5
Barbecue	-199.9	66.9	38.6	-72.8	24.5	14.2
Total	-524.6	67.8	46.2	-70.5	9.1	6.2

perform best when using `Threshold` values between 0.2 and 0.4.

Second, when `Threshold` increases from 0.1 to 1.0, FaMT techniques using P_1 drop more dramatically than techniques using P_2 in terms of reduction. For example, when `Threshold` increases from 0.1 to 0.2, the technique using C_1, method-level history, and P_1 formula drops from 43.3% to 12.52%, while the technique using C_1, method-level history, and P_2 formula does not drop at all. The reason is that using the history of all neighbor mutants unnecessarily lowers the power values of tests and the majority of tests will not have power values of greater than 0.2. The only exceptions are the techniques using statement-level history, which remain stable when `Threshold` increases for both P_1 and P_2. The reason is that if one mutant in a statement is killed, there is a high likelihood that all other mutants in the same statement are also killed, making prioritization using P_1 and using P_2 perform similarly.

Comparison of FaMT with two theoretical techniques. To further investigate FaMT's effectiveness, we further present the reduction of executions by the theoretically worst and best techniques, and compare them with FaMT. The theoretically worst technique executes for each mutant all the tests that cannot kill that mutant before it executes a test that can kill that mutant, while the theoretically best technique executes a test that can kill the mutant first. Table 7 presents the reductions achieved by the theoretically worst/best techniques, and an example FaMT technique (i.e., C_3 with the default `Threshold` of 0.3, class-level history, and P_2 formula) over *DR*. Column 1 lists all the subjects, Columns 2-4 present the reduction of executions for killed mutants, while Columns 5-7 present the reduction of executions for all mutants.

First, the example FaMT technique is close to the theoretically best technique. The theoretically best technique reduces the executions by 14.9% to 90.4% with a total reduction of 67.8%. The example FaMT technique reduces the executions by 5.4% to 67.4% with a total reduction of 46.2%, indicating that FaMT performs closely to the theoretically best technique. For all the subjects, the theoretically worst technique reduces the number of test-mutant executions for killed mutants by -981.8% to -41.8% with a total reduction of -524.6%, i.e., in other words, the theoretically worst technique increases the number of executions by about six times.

Second, the executions for all mutants cannot be reduced greatly even using the theoretically best prioritization technique. For all the subjects, the theoretically best technique only reduces the number of executions for all mutants by 4.0% to 31.2% with a total reduction of 9.1%. Therefore, FaMT also cannot reduce the number of executions for all mutants greatly: it reduces the number of executions for all mutants by 1.5% to 18.0% with a total reduction of 6.2%. The reason for the low reduction on executions of all mutants is that all prioritization techniques cannot reduce the executions for the mutants that cannot be killed. This finding also further motivates our second study of FaMT reduction.

4.5.2 *RQ2: FaMT Test Reduction*

Evaluation with default threshold and minratio. We applied FaMT reduction techniques with `Threshold` and `MinRatio` both at the default value of 0.3 on all subject programs. Similar with FaMT prioritization techniques, we compared FaMT reduction techniques with *DR* for 20 times for each subject. Table 8 presents the mean reduction ratios and mean error rates across 20 runs for all FaMT techniques with P_1 formula. Columns 1 and 2 show the levels of *adaptive* history (denoted as *A.*) and *initial* orderings (denoted as *I.*) used. Columns 3-20 present the reduction ratios and error rates achieved by the studied techniques for each subject. Columns 21 and 22 list the total reduction ratios and error rates over all subjects. Similarly, Table 9 presents the experimental results for FaMT reduction using the P_2 formula.

First, all the techniques can reduce the test executions effectively without causing high error rates. In total, when using P_1, the FaMT reduction techniques with both `Threshold` and `MinRatio` of 0.3 reduce test executions by 48.1% to 65.1%, while only causing error rates from 0.44% to 2.46%. When using P_2, the FaMT reduction techniques can reduce the number of test executions by 5.3% to 63.3%, while only causing error rates from 0.07% to 1.22%.

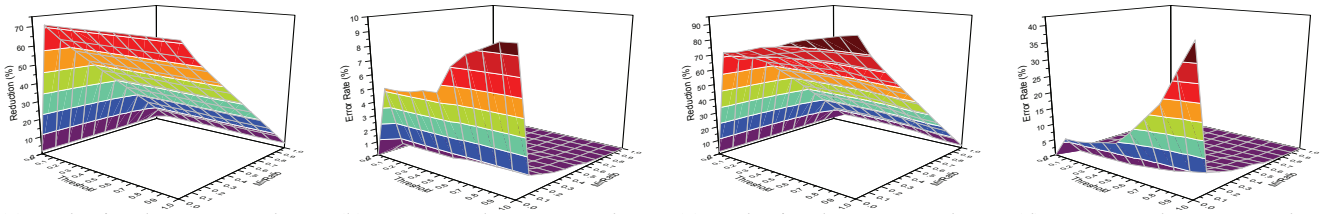
Second, using the P_2 formula gets more conservative reduction than using the P_1 formula. For example, for the statement-level

Table 8: Reduction results (%) for FaMT reduction with `Threshold`, `MinRatio`=0.3, and history of all neighbor mutants (P_1)

A.	I.	*Time&Money*		*Jaxen*		*Xml-Sec*		*Com-Lang*		*JDepend*		*Joda-Time*		*JMeter*		*Mime4J*		*Barbecue*		Total	
		Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.
S	*DR*	15.7	0.93	54.5	1.16	33.1	0.42	16.6	0.51	25.4	0.36	20.7	0.51	8.0	0.06	29.9	1.06	30.7	1.09	48.5	0.61
t	C_1	17.8	0.73	53.4	1.76	33.8	0.50	17.8	0.33	24.6	0.55	25.6	0.44	8.2	0.08	30.6	0.45	31.6	0.62	48.1	0.53
a	C_2	15.2	0.98	56.4	0.30	31.6	0.58	13.9	0.69	23.7	0.79	21.2	0.62	7.6	0.05	28.4	1.03	28.1	1.00	49.9	0.60
t	C_3	16.7	0.77	54.9	0.50	34.0	0.52	16.7	0.42	27.3	0.53	27.1	0.50	7.9	0.05	30.7	0.49	30.8	0.53	49.5	0.44
M	*DR*	22.3	0.93	65.6	1.99	48.8	1.68	23.5	1.29	37.4	1.09	34.6	1.12	11.8	0.22	44.5	2.47	47.0	2.55	59.8	1.36
e	C_1	23.4	0.84	64.3	2.76	49.0	1.99	24.3	1.02	38.0	1.57	37.4	0.96	12.0	0.21	44.8	1.76	47.7	2.32	59.1	1.27
t	C_2	21.8	1.02	68.0	0.96	47.9	2.36	22.1	1.49	36.5	1.71	35.5	1.48	11.6	0.24	43.6	2.51	46.1	3.12	61.8	1.48
h	C_3	22.7	0.89	65.9	1.24	49.1	2.10	23.5	1.17	40.2	1.41	39.0	1.08	11.9	0.18	44.8	1.74	48.0	2.28	60.6	1.17
C	*DR*	32.0	1.51	66.9	2.87	54.0	2.08	25.4	1.50	48.4	1.93	44.1	1.58	12.7	0.31	49.4	2.38	51.8	5.27	62.2	1.76
l	C_1	32.5	1.29	65.6	3.75	54.4	1.55	26.5	1.32	47.7	3.52	46.6	1.35	12.9	0.28	49.4	2.30	52.9	4.61	61.4	1.70
a	C_2	32.5	1.65	69.4	1.87	53.5	2.91	24.8	1.59	47.1	2.60	43.9	1.88	12.5	0.27	48.2	2.52	50.3	5.20	64.2	1.84
s	C_3	32.2	1.30	67.2	2.23	53.9	2.56	25.8	1.49	47.3	3.92	47.1	1.45	12.8	0.25	49.4	2.19	52.9	4.39	62.8	1.65
G	*DR*	29.8	1.99	67.2	4.26	56.8	2.94	25.5	1.65	50.9	2.34	52.1	2.65	12.4	0.52	50.2	2.29	54.2	7.47	63.3	2.40
l	C_1	32.6	1.94	65.7	5.60	57.5	2.05	26.7	1.45	51.1	3.65	54.5	2.40	12.6	0.41	50.8	2.33	55.0	6.85	62.4	2.36
o	C_2	30.9	2.37	69.3	3.24	56.6	3.35	24.9	1.78	50.4	2.60	53.5	3.04	12.6	0.49	49.7	2.03	53.0	8.19	65.1	2.46
b	C_3	31.9	2.10	67.3	3.92	57.5	3.18	26.1	1.61	51.2	3.75	54.9	2.59	12.7	0.36	50.6	2.24	55.0	6.57	63.7	2.33

Table 9: Reduction results (%) for FaMT reduction with `Threshold`, `MinRatio`=0.3, and history of killed neighbor mutants (P_2)

A.	I.	*Time&Money*		*Jaxen*		*Xml-Sec*		*Com-Lang*		*JDepend*		*Joda-Time*		*JMeter*		*Mime4J*		*Barbecue*		Total	
		Red.	Dev.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.	Red.	Err.
S	*DR*	10.0	0.21	10.9	0.05	5.5	0.02	6.6	0.08	9.2	0.06	9.8	0.10	1.4	0.00	7.1	0.05	11.9	0.26	10.6	0.07
t	C_1	12.2	0.19	14.1	0.06	6.3	0.02	7.4	0.07	8.1	0.34	14.4	0.13	1.7	0.00	7.3	0.04	11.9	0.23	13.6	0.08
a	C_2	9.3	0.22	4.6	0.10	4.2	0.02	4.0	0.08	8.1	0.29	10.9	0.12	1.1	0.00	5.7	0.04	9.2	0.23	5.3	0.09
t	C_3	11.5	0.20	9.6	0.06	6.5	0.02	6.3	0.08	10.7	0.36	16.2	0.14	1.4	0.00	7.5	0.04	11.2	0.16	10.1	0.09
M	*DR*	13.2	0.33	28.8	0.37	13.4	0.20	12.2	0.45	16.6	0.13	21.8	0.30	2.8	0.05	18.9	0.45	29.8	1.19	27.2	0.34
e	C_1	14.5	0.22	39.2	0.42	13.7	0.13	12.0	0.35	17.6	0.48	24.4	0.35	3.1	0.05	18.1	0.37	26.1	1.14	35.6	0.33
t	C_2	12.7	0.25	19.9	0.51	14.1	0.19	11.2	0.46	16.7	0.44	21.9	0.43	2.7	0.06	19.3	0.46	29.7	1.31	19.7	0.41
h	C_3	14.0	0.20	28.5	0.36	14.3	0.09	11.0	0.40	19.1	0.47	25.8	0.36	3.1	0.05	18.5	0.36	29.5	0.96	27.0	0.33
C	*DR*	18.5	0.76	46.3	0.91	17.5	0.33	14.9	0.52	37.6	0.72	30.5	0.66	4.0	0.08	31.6	0.76	43.9	2.78	42.9	0.63
l	C_1	17.2	0.55	45.8	1.00	18.8	0.22	15.3	0.46	39.3	0.95	33.0	0.60	4.4	0.06	30.8	0.55	41.0	1.93	42.5	0.55
a	C_2	19.7	0.76	46.4	1.17	20.9	0.21	15.2	0.63	39.8	0.99	30.0	0.78	3.9	0.06	31.9	0.96	44.3	2.72	42.7	0.74
s	C_3	17.9	0.61	37.5	0.91	19.0	0.17	14.9	0.51	40.6	0.93	33.9	0.66	4.1	0.05	31.1	0.58	44.5	2.23	35.8	0.58
G	*DR*	16.4	0.88	69.0	2.37	39.4	0.50	15.8	0.62	35.9	1.04	45.7	1.66	4.7	0.04	33.9	1.09	50.7	4.45	63.0	1.20
l	C_1	16.2	0.84	69.1	2.68	43.5	0.50	16.1	0.55	37.7	1.06	46.0	1.52	4.9	0.03	38.9	0.99	49.7	3.67	63.3	1.14
o	C_2	17.7	0.88	69.1	2.03	43.6	0.54	16.0	0.78	42.2	1.24	46.5	1.63	3.9	0.05	33.5	1.23	49.3	4.74	63.3	1.22
b	C_3	16.8	0.82	68.5	2.41	46.2	0.52	16.2	0.62	41.7	1.03	46.2	1.49	4.2	0.03	37.5	1.00	51.3	3.53	62.9	1.12



(a) Reduction by FaMT reduction using C_3, statement-level history, and P_1 formula

(b) Error rate by FaMT reduction using C_3, statement-level history, and P_1 formula

(c) Reduction by FaMT reduction using C_3, global-level history, and P_2 formula

(d) Error rate by FaMT reduction using C_3, global-level history, and P_2 formula

Figure 3: Reduction and error rate trends on different `Threshold` and `MinRatio` values by FaMT reduction

Table 10: Reduction error rates (%) for example FaMT reduction techniques and corresponding random techniques

Tech.	*Time&Money*		*Jaxen*		*Xml-Sec*		*Com-Lang*		*JDepend*		*Joda-Time*		*JMeter*		*Mime4J*		*Barbecue*		Total
	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.	Dev.	Err.
FaMT1	0.93	0.30	1.16	1.27	0.42	0.27	0.51	0.07	0.36	0.30	0.51	0.07	0.06	0.03	1.06	0.28	1.09	0.40	0.61
Rand.	9.73	1.25	11.59	1.32	8.40	1.15	6.54	0.56	18.41	3.23	13.04	0.68	1.38	0.21	11.29	2.31	17.08	1.78	9.58
FaMT2	0.73	0.27	1.76	1.34	0.50	0.21	0.33	0.06	0.55	0.41	0.44	0.05	0.08	0.03	0.45	0.09	0.62	0.27	0.53
Rand.	10.78	1.09	11.56	1.41	8.97	1.13	7.20	0.24	18.35	3.19	14.70	0.49	1.56	0.18	12.60	2.51	17.26	1.84	10.47
FaMT3	0.98	0.29	0.30	0.07	0.58	0.19	0.69	0.07	0.79	0.35	0.62	0.08	0.05	0.02	1.03	0.17	1.00	0.57	0.60
Rand.	9.69	1.05	12.62	0.68	8.03	0.93	6.16	0.21	17.61	2.91	12.98	0.40	1.35	0.15	10.78	2.09	16.38	1.32	9.47
FaMT4	0.77	0.31	0.50	0.60	0.52	0.17	0.42	0.05	0.53	0.40	0.50	0.06	0.05	0.03	0.49	0.09	0.53	0.31	0.44
Rand.	10.42	1.04	11.90	1.35	9.02	1.06	6.80	0.23	19.71	3.51	14.38	0.47	1.53	0.15	12.55	2.55	17.25	1.53	10.31

history, techniques using P_2 reduce executions from 5.3% to 13.6% and cause less than 0.1% error rates, while techniques using P_1 reduce executions by higher ratios (from 48.1% to 49.9%) and cause slightly higher error rates (from 0.44% to 0.61%). The reason is that using P_2 causes tests to have larger power values and tend to stay in the first priority list (i.e., $\mathcal{T}_1$ from Section 3.4) and thus be run. One interesting finding is that when the history level becomes global, techniques using P_2 achieve similar execution reductions with techniques using P_1, but cause much lower error rates. For example, a technique using global history and P_2 formula reduces all test executions by 63.3% while causing an error rate of 1.14%, while the corresponding technique using global history and P_1 for-

mula reduces executions by 62.4% while causing a twice as high error rate, 2.36%. The reason is that using the global history of all neighbor mutants unnecessarily lowered the power of tests, causing some *powerful* tests to be moved into the secondary list (i.e., $\mathcal{T}_2$ from Section 3.4) and not run later.

Third, there are many techniques that can reduce test executions significantly with negligible error rates. In total, techniques using global-level history and P_2 formula reduce executions by more than 63.0% with less than 1.22% error rates. Techniques using statement-level history and P_1 formula reduce executions by around 50.0% with only around 0.50% error rates. Although the techniques using statement-level history and P_1 formula reduce executions by smaller ratios, their error rates are smaller and more stable. For example, when using the C_3 initial ordering, it only causes error rates of 0.05% to 0.77% across all subjects.

Effectiveness and error rate trends when using various thresholds and minratios. Figure 3 illustrates the total reduction and error rate trends for all subjects when two example FaMT reduction techniques use different `Threshold` and `MinRatio` values from 0.0 to 1.0. The two example techniques are carefully chosen such that they are different enough from each other. The trends for other FaMT reduction techniques should be similar. More precisely, Figures 3(a) and 3(b) present the reduction and error rate trends for FaMT reduction using the C_3 initial ordering, statement-level history, and P_1 formula. Figures 3(c) and 3(d) present the reduction and error rate trends for FaMT reduction using the C_3 initial ordering, global-level history, and P_2 formula. In each sub-figure, the two horizontal axes represent `Threshold` and `MinRatio`, and the vertical axis represents the reductions or error rates.

First, when `Threshold` is fixed, if `MinRatio` increases from 0.0 to 1.0, the reductions achieved by FaMT reduction techniques drop linearly for both techniques, while the error rates drop more dramatically when `MinRatio` increases from 0.0 to 0.1. For instance, for the first example technique with `Threshold`=1.0, when `MinRatio` increases from 0.0 to 0.1, the reduction drops from 71.7% to 63.0%, while the error rate drops from 9.42% to 1.22%. When `MinRatio` increases from 0.1 to 0.2, the reduction ratio drops from 63.0% to 55.9%, while the error rate only drops from 1.22% to 0.93%. A similar observation can be made for the second example technique. This indicates that using the `MinRatio` of 0.0 is not cost-effective, and using `MinRatio` from 0.5 to 1.0 might cause low reduction. Therefore, using `MinRatio` values between 0.1 and 0.5 can be cost-effective choices.

Second, when `MinRatio` is fixed, if `Threshold` increases from 0.1 to 1.0[3], the reductions achieved by FaMT reduction techniques increase linearly for both techniques, while the error rates increase more dramatically when `Threshold` increases from 0.5 to 1.0. For instance, for the first example technique with `MinRatio` of 0.0, when `Threshold` increases from 0.5 to 1.0, the reduction ratio increases from 71.6% to 71.7%, while the error rate increases from 5.68% to 9.42%. Similarly, for the second example technique with `MinRatio` of 0.0, when `Threshold` increases from 0.5 to 1.0, the reduction ratio increases from 81.3% to 93.4%, while the error rate increases from 9.56% to 40.74%. This indicates that `Threshold` values between 0.1 and 0.5 are more cost-effective than `Threshold` values between 0.5 and 1.0 for FaMT reduction.

Comparison of FaMT reduction with random techniques. To further investigate the effectiveness of FaMT reduction techniques, we compare FaMT techniques with random techniques that execute the same number of tests as FaMT reduction for each mutant. Table 10 presents the mean values and standard deviations of error rates caused by the four example FaMT techniques using statement-level history and P_1 formula with corresponding random techniques across 20 runs. Column 1 lists all the compared techniques (each example FaMT technique followed with a random technique). Columns 2-19 list the mean error rates and their standard deviations for each subject. Column 20 lists the overall error rates for all subjects in total.

[3]When `Threshold`=0.0, all the tests are stored in the first priority list and thus executed, and the reduction ratios are close to 0.

Table 11: Comparison between FaMT and regression test prioritization and reduction

Sub	Test Prioritization (%)			Test Reduction (%)			
	FaMT	Tot.	Add.	FaMT		Greedy	
Time&Money	17.0	-15.7	4.7	16.7	(0.77)	30.1	(3.99)
Jaxen	67.4	-144.0	55.8	54.9	(0.50)	72.7	(2.62)
Xml-Sec	28.8	-29.4	-9.2	34.0	(0.52)	38.2	(4.12)
Com-Lang	8.8	-12.5	5.2	16.7	(0.42)	24.3	(1.97)
JDepend	29.4	-4.8	18.3	27.3	(0.53)	45.2	(3.15)
Joda-Time	38.3	5.5	27.0	27.1	(0.50)	50.5	(3.97)
JMeter	5.4	-2.7	3.3	7.9	(0.05)	5.2	(0.09)
Mime4J	23.9	-30.7	-1.0	30.7	(0.49)	31.8	(1.61)
Barbecue	38.6	-46.5	-18.0	30.8	(0.53)	53.4	(16.59)

The error rates caused by random techniques are much larger than those of FaMT techniques although they reduce the executions to the same extent. For example, the first FaMT technique causes an error rate of 0.61% in total for all subjects, while the corresponding random technique causes an error rate of 9.58%. In addition, the error rates caused by FaMT techniques are more stable than those of random techniques. For example, the standard deviations of error rates caused by the first FaMT technique range from 0.03% to 1.27%, while the standard deviations of error rates caused by the corresponding random technique range from 0.21% to 3.23%.

4.5.3 RQ3: Comparison with Regression Techniques

Table 11 summarizes the comparison of two example FaMT techniques and traditional regression testing techniques. Column 1 lists all the subjects. Columns 2-4 present the mean reduction of executions for killed mutants (across 20 runs for each subject) achieved by the example FaMT prioritization technique (using the C_3 initial ordering, class-level history, P_2 formula, and default `Threshold`) with the *total* and *additional* regression test prioritization techniques. Columns 5 and 6 present the mean reduction of executions for all mutants with mean error rates in brackets (across 20 runs) achieved by the example FaMT reduction technique (using C_3 initial ordering, statement-level history, P_1 formula, and default `Threshold` and `MinRatio`) and the *greedy* regression reduction technique.

Both regression prioritization techniques do not always reduce the number of executions for killed mutants. For example, the *total* technique reduces executions by -144.0% to 5.5%, while the *additional* technique reduces executions by -18.0% to 55.8%. In contrast, the example FaMT prioritization effectively reduces executions from 5.4% to 67.4% for all subjects. We believe the reason is that regression prioritization techniques were not originally designed for mutation testing. One interesting finding is that the *additional* regression prioritization technique is able to reduce executions effectively for several subjects. For example, it reduces executions by more than 10% for three subjects. The reason is that diverse tests tend to be executed early against each mutant because the *additional* technique always picks the test that covers most uncovered program elements as the next test. The early execution of diverse tests may have a higher probability to kill a mutant earlier.

The *greedy* regression reduction technique can significantly reduce the number of executions for all subjects (from 5.2% to 72.7%). However, the error rates caused by it can be extremely high for

Table 12: Runtime overhead by FaMT techniques

Sub	Javalanche Time (s)	FaMT Prioritization		FaMT Reduction	
		P_1 (s)	P_2 (s)	P_1 (s)	P_2 (s)
Time&Money	433	0.04	0.04	0.03	0.04
Jaxen	2901	18.34	17.68	17.72	17.47
Xml-Sec	3184	0.91	0.70	0.89	0.59
Com-Lang	4475	0.71	0.70	0.70	0.69
JDepend	182	0.07	0.06	0.05	0.06
Joda-Time	11788	8.55	8.13	8.16	8.28
JMeter	3452	0.22	0.16	0.15	0.13
Mime4J	10880	0.41	0.44	0.47	0.41
Barbecue	455	0.10	0.08	0.09	0.09

some subjects, e.g., 16.59% for *Barbecue*, making it not suitable for reducing the cost of mutation testing. In contrast, although the example FaMT reduction reduces executions by smaller ratios (from 7.9% to 54.9%), it incurs small and stable error rates (from 0.05% to 0.77%), demonstrating that FaMT reduction is more suitable than regression reduction for reducing mutation testing cost.

4.5.4 RQ4: FaMT Efficiency

We measured the runtime overheads for all FaMT prioritization and reduction techniques. Due to the space limitation, we only show the overheads for the most expensive techniques that use C_3 (i.e., the heuristic that combines C_1 and C_2, thus needing more time to calculate) and the global history. The runtime overheads for other FaMT techniques are no more than the ones shown. In Table 12, Column 1 lists the subjects, and Column 2 lists the total execution time for the state-of-the-art Javalanche tool to calculate the mutation score. Columns 3 and 4 list the execution time for the example FaMT prioritization techniques using P_1 and P_2, respectively. Similarly, Columns 5 and 6 list the execution time for the reduction techniques using P_1 and P_2, respectively. All the presented four techniques cause similar overheads. The reason is that all the techniques are based on the same basic algorithm (Algorithm 1). The overhead for each technique varies a lot across different subjects, because FaMT techniques cost more for subjects with a larger number of mutants or larger sets of tests that reach each mutant. The results also show that FaMT costs at most 18.34s (on *Jaxen*), which is negligible compared to the cost of mutation testing (2901s on *Jaxen*) and demonstrates the efficiency of FaMT.

4.5.5 Threats to Validity

Threats to external validity. First, although we used 9 medium-sized Java programs, our results may not be generalizable to other programs. Second, the results may not be generalizable to other test suites. Third, our results based on mutants generated by Javalanche may not be generalizable to mutants generated by other tools.

Threats to internal validity. The main threat to internal validity for our study is that there may be faults in our implementation of the 352 variant prioritization techniques and 3872 variant reduction techniques of FaMT, as well as the other controlled techniques. To reduce this threat, we reviewed all the code that we produced for our experiments before conducting the experiments.

Threats to construct validity. The main threat to construct validity is the metrics that we used to assess the effectiveness and cost of our techniques. To reduce this threat, we used the ratio of executions reduced to assess the techniques' effectiveness, and used the runtime overhead to assess the techniques' cost. We also used error rate to measure the approximation caused by FaMT reduction.

5. RELATED WORK

We present related work on reducing mutation testing cost. Furthermore, as FaMT is inspired by regression test prioritization and reduction, we also discuss related work in regression testing [35].

Table 13: Regression testing techniques for mutation testing

	Regression Testing	Mutation Testing
Test Selection	[16, 27, 28]	[39]
Test Prioritization	[10, 24, 29, 36]	[20, 39], This paper
Test Reduction	[5, 6, 13, 15]	This paper

Table 13 shows three main areas of regression testing and their applications to mutation testing.

5.1 Reducing Cost of Mutation Testing

Selective mutation testing selects a representative subset of all mutants that can achieve similar results as the entire set of mutants. Since its initial proposal by Mathur [22], a large body of research has investigated this topic. Researchers [4, 11, 25, 26, 37] have experimentally investigated various subsets of mutants to ensure that those representative sets of mutants achieve almost the same results as the whole mutant set. *Weak mutation testing*, first proposed by Howden [17], aims to check mutant killing more efficiently by determining that a test kills a mutant if the test produces a different *internal state* (rather than the *output*) when executing the mutant. Researchers also considered various efficient ways to generate, compile, and execute mutants. DeMillo et al. [7] extended compilers to compile all mutants at once to reduce the cost of generating/compiling mutants. Similarly, Untch et al. [32] proposed schema-based mutation, which integrates all mutants into one meta-mutant that can be compiled by a standard compiler. Researchers have also used parallel processing [23] to speed up mutation testing. Our FaMT techniques are orthogonal to the existing techniques that optimize mutation testing and can be directly combined with those techniques to further reduce the cost of mutation testing.

We recently adapted the idea of *regression test selection* for mutation testing, and proposed the ReMT technique [39], which relies on program differences and incrementally collects mutation testing results based on old mutation testing results of a previous version (Table 13). ReMT also includes test prioritization based on program differences, and cannot be applied without mutation testing results on old versions. In contrast, FaMT prioritization does not rely on program differences, and can directly apply to any program without old mutation testing results. After our ReMT, Just et al. [20] used test prioritization for one program version, but their prioritization is quite different from our FaMT as they prioritize tests only once for all mutants by simply executing slower tests later. In contrast, FaMT re-prioritizes tests for each mutant (which provides much better results) and uses different metrics based on execution length, frequency, and the accumulating history of mutant execution. Furthermore, this paper shows that even the theoretically best test prioritization cannot reduce the mutation testing cost significantly, and further introduces the idea of *test reduction* for mutation testing.

5.2 Regression Testing

Here we only discuss the most related regression test prioritization and reduction areas.

Test prioritization reorders regression tests to detect faults in code faster. Rothermel et al. [10, 29] proposed two general strategies for test prioritization: (1) the *total* strategy which prioritizes tests based on the number of code elements they cover, (2) the *additional* strategy which prioritizes tests based on the number of additional elements they cover. Do et al. [9] investigated the suitability of using mutation faults to simulate real faults to evaluate traditional test prioritization techniques. Recently, Zhang et al. [36] proposed unified models for test prioritization which subsume the total and additional strategies as extreme cases, and also contain a spec-

trum of strategies between the two strategies. The basic intuitions for regression prioritization and FaMT prioritization are similar: to reorder the tests to make regression/mutation testing faster. However, their mechanisms are different—regression test prioritization usually aims to cover more program units faster, thus increasing the probability of revealing unknown faults earlier; whereas the locations of mutation faults (i.e., mutated statements) are known for mutation testing and a simple strategy can just execute the tests that reach the mutation faults, making the coverage-based regression test prioritization technique not suitable for mutation testing (also confirmed by our experimental study).

Test reduction executes only a representative subset of regression tests, which can still satisfy all the testing requirements. Harrold et al. [15] were inspired by the fact that some *essential* tests should be picked as early as possible because they test rarely tested requirements, and proposed a heuristic for iteratively picking essential tests. Chen et al. [6] further found that some *redundant* tests, which test only a subset of requirements tested by other tests, should be reduced as early as possible, and proposed to reduce tests by iteratively applying essential test selection and redundant test reduction. Researchers [5, 13] also considered integer linear programming models for reducing regression tests. While the idea of test reduction has been around for decades, to our knowledge, this paper is the first to present a technique for mutation testing inspired by this idea. However, the goals of regression test reduction and FaMT reduction are different, and as our study shows, regression test reduction alone is not suitable for mutation testing.

6. CONCLUSION AND FUTURE WORK

This paper presents the FaMT approach that provides a family of techniques for prioritizing and reducing tests to speed up mutation testing. The basic idea of FaMT was inspired by regression test prioritization and reduction, but the technical details and goal for FaMT are different from regression test prioritization and reduction. The paper reports an empirical study of 352 FaMT prioritization variant techniques and 3872 FaMT reduction variant techniques on 9 real-world Java programs to investigate FaMT's effectiveness and efficiency. The experimental study shows that FaMT prioritization techniques can reduce the number of executions by up to 47.52% for killed mutants. The study of FaMT reduction further shows that some FaMT reduction techniques can reduce all executions for all mutants by around 50.0% while only causing error rates around 0.50%, and some FaMT reduction techniques can reduce all executions for all mutants by more than 63.0% while causing error rates smaller than 1.22%. Finally, the study shows that FaMT incurs negligible runtime overhead.

For future work, we plan to investigate more heuristics for the initial test ordering, and more effective ways to utilize test power values. We also plan to empirically study FaMT with more configurations on more real-world programs. In addition, we plan to apply FaMT for higher-order mutation testing [14, 18].

7. ACKNOWLEDGEMENTS

This work is partially supported by the National Science Foundation under Grant Nos. CCF-0746856 and CCF-0845628.

8. REFERENCES

[1] PIT. http://pitest.org/.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, pages 402–411, 2005.

[4] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *STVR*, 11(2):113–136, 2001.

[5] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proc. ICSE*, pages 106–115, 2004.

[6] T. Chen and M. Lau. A new heuristic for test suite reduction. *IST*, 40(5):347–354, 1998.

[7] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *Proc. COMPSAC*, pages 351–356, 1991.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[9] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE TSE*, 32(9):733–752, 2006.

[10] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE TSE*, 28(2):159–182, 2002.

[11] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *Proc. ISSTA*, page to appear, 2013.

[12] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE TSE*, pages 279–290, 1977.

[13] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *Proc. ICSE*, pages 738–748, 2012.

[14] M. Harman, Y. Jia, and W. Langdon. Strong higher order mutation-based test data generation. In *Proc. FSE*, pages 212–222, 2011.

[15] M. Harrold, R. Gupta, and M. Soffa. A methodology for controlling the size of a test suite. *ACM TOSEM*, 2(3):270–285, 1993.

[16] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proc. OOPSLA*, pages 312–326, 2001.

[17] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE TSE*, pages 371–379, 1982.

[18] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009.

[19] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE TSE*, 37(5):649–678, 2011.

[20] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proc. ISSRE*, pages 11–20, 2012.

[21] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[22] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Proc. COMPSAC*, pages 604–605, 1991.

[23] A. P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. Technical report, Purdue University, 1988.

[24] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *TSE*, 38(6):1258–1275, 2012.

[25] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proc. ICSE*, pages 351–360, 2008.

[26] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM TOSEM*, 5(2):99–118, 1996.

[27] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proc. FSE*, pages 241–252, 2004.

[28] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM TOSEM*, 6(2):173–210, 1997.

[29] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *IEEE TSE*, 27(10):929–948, 2001.

[30] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proc. ISSTA*, pages 69–80, 2009.

[31] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proc. FSE*, pages 297–298, 2009.

[32] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *Proc. ISSTA*, pages 139–148, 1993.

[33] L. J. Williams and H. Abdi. Fisher's least significant difference (LSD) test. *Encyclopedia of Research Design*, 2010.

[34] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *JSS*, 31(3):185–196, 1995.

[35] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *STVR*, 22(2):67–120, 2012.

[36] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proc. ICSE*, pages 192–201, 2013.

[37] L. Zhang, S. S. Hou, J. J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proc. ICSE*, pages 435–444, 2010.

[38] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An empirical study of junit test-suite reduction. In *Proc. ISSRE*, pages 170–179, 2011.

[39] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proc. ISSTA*, pages 331–341, 2012.

[40] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, Dec. 1997.