Approximate Transformations as Mutation Operators

Farah Hariri[1], August Shi[1], Owolabi Legunsen[1], Milos Gligoric[2], Sarfraz Khurshid[2], Sasa Misailovic[1]
[1] Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801, USA
{hariri2,awshi2,legunse2,misailo}@illinois.edu
[2] Department of Electrical and Computer Engineering
The University of Texas at Austin, TX 78712, USA
{gligoric,khurshid}@utexas.edu

***Abstract*—Mutation testing is a well-established approach for evaluating test-suite quality by modifying code using syntax-changing (and potentially semantics-changing) transformations, called mutation operators. This paper proposes approximate transformations as new mutation operators that can give novel insights about the code and tests. Approximate transformations are semantics-changing transformations used in the emerging area of approximate computing, but so far they were not evaluated for mutation testing. We found that approximate transformations can be effective mutation operators. We compared three approximate transformations with a set of conventional mutation operators from the literature, on nine open-source Java subjects. The results showed that approximate transformations change program behavior differently fromconventional mutation operators. Our analysis uncovered code patterns in which approximate mutants survivedand showed the practical value of approximate transformations for both understanding code amenable to approximations and discovering bad tests. We submitted 11 pull requests to fix bad tests. Seven have already been integrated by the developers.**

I. Introduction

Mutation testing is a well-established approach for evaluating the quality of a test suite [33]. It produces modified versions of the code, called *mutants*, using a set of syntactic transformations that also potentially change program semantics, called *mutation operators*. It then runs the test suite on the mutants to quantify how well the suite detects the program modifications. To provide insights about how to improve the test suites, mutation testing requires both high-quality and diverse mutation operators that lead to different program behaviors.

This paper proposes *approximate transformations* as a new class of mutation operators that lead to different program behaviors from those produced by conventional mutation operators. Approximate transformations were introduced in the emerging area of *approximate computing* for changing program semantics to trade the accuracy of results for improved energy efficiency or performance. Researchers proposed various approximate transformations at the level of programming languages, compilers, and computer systems [5], [21], [22], [26], [27], [49], [50], [56], [57], [59], [65], [67], [73]. For example, loop perforation [49], [65] is a compiler-level approximate transformation that causes amenable loops to execute only a subset of iterations. Degrading floating-point precision is another common language-level [57] and system-level approximate transformation [5], [59], [73].

Mutation operators and approximate transformations both aim to change program semantics. Hence, approximate transformations are an attractive choice for new mutation operators that can provide novel insights about tested code and test suites. Our analysis of three approximate transformations—loop perforation, integer-to-short precision degradation, and double-to-float precision degradation—shows that they often complement conventional mutation operators. Our evaluation on nine open-source Java subjects focuses on the following three research questions:

RQ1: How effective are approximate transformations as mutation operators, compared to conventional mutation operators?
RQ2: What code patterns do approximate transformations as mutation operators reveal?
RQ3: How can approximate transformations as mutation operators help software testing practice?

To evaluate the effectiveness of approximate transformations as mutation operators, we use a combination of techniques established in prior work on mutation testing:

- *Mutation Score:* We compare mutation scores [12], [33] of three approximate transformations with the mutation scores of 14 conventional mutation operators from the PIT framework [3]. Mutation scores are percentages of mutants detected, or *killed*, by the test suite [33].
- *Minimal Mutants:* We check whether approximate transformations generate mutants that are in the minimal mutants set, computed across mutants from both approximate transformations and conventional mutation operators. Minimal mutants dynamically subsume all other mutants (defined in III-B), and they are considered harder to kill than all other mutants [11], [24].
- *Sufficient Mutation Operators:* We check whether approximate transformations are in the set of sufficient mutation operators, computed using *selective mutation analysis* [51], [52]. Tests that kill mutants generated by sufficient mutation operators also kill mutants generated by all other operators.

The results (Section IV) show that approximate transformations are effective mutation operators. Loop perforation has similar mutation scores as conventional mutation operators. However, we observe that mutation score alone is not sufficient for evaluating the effectiveness of approximate transformations. While precision degradation operators have significantly lower mutation scores, our analysis shows that their low mutation scores are not due to the mutants being semantically equivalent to the original code. Rather, the operators expose bad tests that do not exercise the code with boundary values. Approximate transformations also generate mutants in the minimal mutant sets, and they are often in the sufficient mutation operators sets.

To better understand the pattern of computations affected by approximate transformations, we manually inspected a sample of mutants from the approximate transformations (Section V). For loop perforation, we identify four code patterns, e.g., reductions and conditional computation on elements. For precision degradation, we identify three code patterns, e.g., when computed results are within a specified error bound. Further, the identified patterns allow us to draw more general comparisons with a broader set of mutation operators from recent literature [34], [43], [60]. Our analysis (Section V-C) shows that approximate transformations complement the conventional mutation operators.

Based on our inspection (Section VI), we propose a new way of reasoning about surviving (i.e., not killed) mutants generated by approximate transformations. In traditional mutation testing, a mutant can survive either because it is semantically equivalent to the original code, or because of bad (buggy, inadequate, or missing) tests. We discover that, with approximate transformations, there is a third option—*a surviving mutant can indicate the presence of approximable code*. Code is approximable if it can be transformed to produce results different from the original code, but such results still meet the specification. (Note that this third way of interpreting a surviving mutant may apply to other mutants as well.) Our inspection shows that for loop perforation, 63.83% of surviving mutants indicate bad tests, and 19.15% indicate approximable code. We find no equivalent mutants, and the remaining 17.02% are hard to inspect. For precision degradation, 53.13% of surviving mutants indicate bad tests, 14.58% indicate equivalent mutants, and 11.46% indicate approximable code. The remaining 20.83% are hard to inspect.

We identify common testing practices that help to improve bad tests: (i) achieving greater loop coverage, (ii) exercising loop conditions, (iii) exercising boundary values, and (iv) checking correctness of all output elements. We identify the instances of bad tests in all nine subjects. Even though these insights are not new to the testing community, the real value lies in the fact that the approximate transformations help detect those problems and bring them to the attention of the developer who might not have such considerations in mind. We created 11 pull requests to improve the bad tests. The developers already integrated seven pull requests in their code.

```
private void doSwapTest(AMatrix m) {
  if ((m.rowCount()<2)||(m.columnCount()<2)) return;
  m=m.clone();
  AMatrix m2=m.clone();
  m2.swapRows(0, 1);
  assert(!m2.equals(m));
  m2.swapRows(0, 1);
  assert(m2.equals(m));
  ...}
```

Fig. 2: Test of the `swapRows` method in Fig. 1

The paper makes the following contributions:

- **Concept:** We are the first to study the interplay between approximate transformations and mutation testing operators.
- **Framework:** We developed ApproxiMate as an extension to the PIT framework. It supports approximate transformations as mutation operators and integrates analyses from studies on mutation testing.
- **Evaluation:** Our results show that approximate transformations complement conventional mutation operators: they generate mutants in the minimal mutants set and are often in the sufficient mutation operators set.
- **Insights:** We present code patterns revealed by approximate transformations. We discuss how to interpret the results of mutation testing with approximate transformations and improve bad tests. Developers already accepted seven out of 11 pull requests that we submitted for fixing bad tests.

II. Example

This section illustrates mutation testing and approximate transformations and shows a surviving approximate transformation mutant that resulted in an accepted pull request.

A. Code and Test

The snippet in Figure 1a is from `vectorz` [4] (SHA: `9c688f1`), one of the subjects in our study. The snippet shows the instance method `Matrix#swapRows`; the class `Matrix` represents $m{\times}n$ matrices of type `double`. `swapRows` takes integers `i` and `j`, and then it changes the `Matrix` instance by swapping rows `i` and `j`. A parametrized test that directly covers `swapRows` is `doSwapTest`. It operates on instances of `AMatrix`, a superclass of `Matrix` (Figure 2). `doSwapTest` first makes a copy, `m2`, of the input `m` (when `m` is of type `Matrix`, so is `m2`), swaps the first two rows in `m2`, and asserts that `m` and `m2` are not equal. Then, it swaps the first two rows in `m2` again and asserts that `m2` is now equal to `m`.

B. Mutation Testing

Mutation testing proceeds in two steps; it generates mutants, and then runs the tests on each mutant.

Generating mutants. Mutation testing generates *mutants*—code that differ from the original by small syntactic changes, specified by *mutation operators*, e.g., replacing multiplication with division as in Figure 1b (dark background).

Executing mutants. Mutation testing executes the test suite on each mutant. If a test exhibits different behavior when running on a mutant than when running on the original code, that mutant is considered *killed*. Typically, tests pass on the

```
1  public void swapRows(int i, int j) {
2    if (i == j) return;
3    int a = i * cols;
4    int b = j * cols;
5    int cc = columnCount();
6    for (int k = 0; k < cc; k++) {
7      int i1 = a + k;
8      int i2 = b + k;
9      double t = data[i1];
10     data[i1] = data[i2];
11     data[i2] = t;
12   } }
```

(a) Original code.

```
public void swapRows(int i, int j) {
  if (i == j) return;
  int a = i * cols;
  int b = j / cols ;
  int cc = columnCount();
  for (int k = 0; k < cc; k++) {
    int i1 = a + k;
    int i2 = b + k;
    double t = data[i1];
    data[i1] = data[i2];
    data[i2] = t;
} }
```

(b) Killed mutant changes * to /.

```
public void swapRows(int i, int j) {
  if (i == j) return;
  int a = i * cols;
  int b = j * cols;
  int cc = columnCount();
  for (int k = 0; k < cc; k+=2 ) {
    int i1 = a + k;
    int i2 = b + k;
    double t = data[i1];
    data[i1] = data[i2];
    data[i2] = t;
} }
```

(c) Surviving LPM mutant skips iterations.

Fig. 1: Code from `vectorz` [4], a mutation by a conventional mutation operator and a mutation by LPM

original code, so a mutant is killed when a test fails on the mutant. For instance, when `doSwapTest` is run on the mutant in Figure 1b, the mutant computes the index of the second row in the swap as `0`. The first row to swap is also `0`, so no swap happens. The non-equality assertion on `m2` and `m` fails when run on this mutant, suggesting that the test suite is good enough to kill this semantically different mutant.

Mutation score. Mutation testing results in a *mutation score*—the percentage of killed mutants. Higher mutation scores imply higher-quality test suites; a test suite that is strong enough to kill a larger percentage of mutants is likely strong enough to detect more faults in the code under test [13], [35].

C. Approximate Transformations

Loop perforation. Loop perforation is an approximate transformation [49], [65], which transforms loops like `for (int i = 0; i < len; i++) {...}` to execute only a subset of its iterations. In general, perforation can change the value in the initialization expression, the termination condition, or the increment. We consider loop perforations that skip every other loop iteration. Figure 1c shows an LPM (Loop Perforation Mutator) mutant that changes the loop increment, `k++`, to `k+=2` (light background). With this perforation, `doSwapTest` executing on the mutant will only swap every other element (at even-numbered indices) in the specified rows.

Precision degradation. Precision degradation is an approximate transformation that changes the type of a numerical expression or a variable. Specifically, we downcast results of `int` or `double` arithmetic expressions.

The `int`-to-`short` (ITS) transformation changes result of the expression to be of type `short` (values in the range $-32,768$ to $32,767$). An example ITS mutant is replacing `a + k` on line 7 of Figure 1a with `(short)(a + k)`. ITS drops higher-order bits, which may result in a large error magnitude.

If `a` is instead a double-precision variable, the `double`-to-`float` (DTF) transformation changes the expression `a + k` (where the type of `k` is automatically cast to `double`) to `(double)((float)(a + k))`. The cast back to `double` here is necessary in Java to preserve the type. The resulting computation produces imprecise results, usually with a small error magnitude, because it drops lower mantissa bits. Note that our ITS and DTF transformations are finer-grained variants of the actual approximate transformations; we only cast computations as opposed to types as performed by [57].

D. Analysis of Approximate Transformation for Mutation

For the LPM mutant in Figure 1c, `doSwapTest` swaps only elements at even-numbered indices in the specified rows. Since the assertions only check that `m1` and `m2` are not equal after the first swap, and equal after the second swap, `doSwapTest` passes. Since `doSwapTest` is the only test that covers this mutant, the mutant *survives*, i.e., it is not killed. The survival of this LPM mutant suggests that there is some weakness in the test suite, i.e., some tests are "bad" (buggy, inadequate, or missing). Specifically, this surviving mutant indicates that the assertions are not strong enough to detect the skipping of every other element during the swap. We submitted a pull request to check whether elements in the swapped rows are as expected; our pull request was accepted by the `vectorz` developers.

For the ITS mutant on line 7 (not shown in Figure 1 for lack of space), `doSwapTest` is invoked only with small integers (matrices with small dimensions), so the mutant survives. To kill the mutant, one would write a test with large matrices where the column count exceeds the range of `short`.

III. Study Methodology

ApproxiMate is our framework for evaluating approximate transformations as mutation operators. In this section, we describe ApproxiMate's implementation and analyses, the mutation operators studied, and our evaluation subjects.

A. The ApproxiMate Framework

The ApproxiMate framework extends PIT [3], implements approximate transformations as mutation operators, and provides the matrix of tests to killed mutants, as it has been done in previous studies [7], [62], [64]. We implement the approximate transformations as follows:

- We implement the loop perforation mutator (LPM) to skip every other iteration of loops, because other patterns of skipped iterations have similar power to identify approximable code [47]. We use Spoon [66] to find code locations

TABLE I: PIT operators

Type	Name	Acronym
Default	Conditionals Boundary Mutator	CBM
	Increments Mutator	IM
	Invert Negatives Mutator	INM
	Math Mutator	MM
	Negate Conditionals Mutator	NCM
	Return Values Mutator	RVM
	Void Method Calls Mutator	VMCM
Non-Default	Constructor Calls Mutator	CCM
	Inline Constant Mutator	ICM
	Member Variable Mutator	MVM
	Non Void Method Calls Mutator	NVMCM
	Remove Conditionals Mutator	RCM
	Remove Increments Mutator	RIM
	Switch Mutator	SM

of `for` loops that have increment (`i++`) or decrement (`i--`) statements. These locations are passed to our modified PIT extended with LPM, which uses the ASM library [14] to change the `iinc` bytecode instruction so that increments become `i+=2` and decrements become `i-=2`.

- We implement precision degradation, DTF and ITS, using casting. Recall (Section II-C) that ITS is the `int`-to-`short` precision degradation operator; it casts results of `int` arithmetic expressions to `short`. DTF is the `double`-to-`float` precision degradation operator; it casts results of `double` arithmetic expressions to `float` and then back to `double` to preserve the type. The ITS and DTF implementations perform casting at the bytecode level.

ApproxiMate uses all mutation operators available in PIT: seven active-by-default operators and seven non-default operators (Table I), which we enabled to increase the variety of mutation operators in our experiments. ApproxiMate computes mutation scores using only mutants that are covered by the tests. The comparative analyses require the exact mapping from tests to mutants killed. Since PIT cannot capture the test that killed a mutant because of memory or other runtime errors, we exclude such mutants from the mutation score computations and the comparative analyses.

B. Comparative Analyses in ApproxiMate

In mutation testing, it is desirable to use as few mutants as possible while still resulting in the same confidence in the mutation testing results. Prior research investigated means to identify the subset of mutants that are harder to kill and representative of the other mutants [11], [24], [51], [74]. If approximate transformations generate mutants that are harder to kill than mutants generated by conventional mutation operators, it suggests that they are relatively effective as mutation operators. We use two techniques from the literature to compare the mutants from approximate transformations with those from conventional mutation operators: minimal mutants analysis [11], [24] and selective mutation analysis [46], [52].

Minimal mutants analysis. Minimal mutants [11], [24] are used as proxies for finding what mutants are harder to kill compared with the other mutants [9]. We use minimal mutants, which are based on dynamic subsumption:

- ***Definition:*** A mutant m *dynamically subsumes* another mutant m' if the set of tests that kill m is a subset of the set of tests that kill m'. Intuitively, m is harder to kill than m' because only some tests that kill m' can kill m.
- ***Condition:*** If mutants generated from approximate transformations are in the set of minimal mutants, then they subsume (and are therefore harder to kill than) mutants from some conventional mutation operators.
- ***Computation:*** We apply the algorithm proposed by Gopinath et al. [24] to compute the set of minimal mutants.[1]

Selective mutation analysis. Selective mutation analysis is a heuristic technique for reducing the number of mutants to be run [46], [51], [52]. The general idea in selective mutation analysis is to find a set of sufficient mutation operators:

- ***Definition:*** Sufficient mutation operators are a subset of all mutation operators, such that tests which kill mutants generated by the sufficient mutation operators also kill all mutants generated by the operators that are in the complement of the sufficient set.
- ***Condition:*** If approximate transformations are in the set of sufficient mutation operators, it indicates that they are part of operators that are representative of all mutation operators.
- ***Computation:*** We analyze only the mutants killed by the existing tests, assuming that all other mutants cannot be killed [23]. Our algorithm for selecting sufficient operators is close to what was done in prior work using existing test suites [23], but there are two main differences. First, we do not restrict the number of iterations for removing mutation operators. Second, we apply test-suite reduction on each iteration to create a tailored test suite which is sufficient to kill only mutants generated by the currently-selected operators. This is close to previous studies on selective mutation testing [51], [52] where, on each iteration, a test suite is *generated* to kill only mutants from the selected operators, and the generated tests are checked to see that they kill all mutants.
  Each iteration of the algorithm starts by finding and removing the operator that generates the most number of mutants. The second step in each iteration is to apply test-suite reduction [75] to construct a reduced test suite which kills only mutants generated from the remaining operators. If the reduced test suite kills *all* mutants (not just mutants generated from the remaining operators), the algorithm continues to the next iteration by greedily removing the operator which generates the next highest number of mutants. If a reduced test suite that kills all mutants cannot be generated, we continue the same iteration by putting the removed operator back in the set and removing the next highest mutant-generating operator. The algorithm halts when we cannot remove any more operators and still kill all mutants. The operators that remain after the algorithm halts form the set of sufficient mutation operators.

[1]Gopinath et al. refer to minimal mutants as surface mutants in their work.

TABLE II: Subjects Used in Our Study

Subject	SLOC	Tests	Short Description
commons-imaging	31377	169	Imaging library
commons-io	9957	1098	IO library
HikariCP	4256	96	Database connectivity pool
imglib2	31839	337	Image processing library
vectorz	44009	453	Vector and matrix library
jblas	10356	39	Matrix library
OpenTripPlanner	64202	356	Trip planner
la4j	9368	801	Linear algebra library
meka	36512	306	Machine learning library

TABLE III: Number of Mutants Per Operator

Project	Conv. Avg	LPM	ITS	DTF
commons-imaging	1577.43	275	1097	362
commons-io	653.31	37	191	0
HikariCP	192.69	6	17	1
imglib2	646.54	264	296	245
vectorz	2426.93	1009	1991	1466
jblas	323.79	155	147	29
OpenTripPlanner	2265.71	160	623	478
la4j	644.93	311	569	487
meka	593.85	266	192	153
Average	1036.13	275.89	569.22	357.89

C. Evaluation Subjects

We use nine open-source Java subjects in our evaluation of the approximate transformations as mutation operators. Table II shows for each subject the source lines of code (SLOC) it has, the total number of test methods, and a description. The subjects vary widely in size and come from different domains: image processing, machine learning, linear algebra, and databases applications. The subjects are from GitHub and are a mix of (1) subjects used in previous software testing papers [39], [42], and (2) computationally-intensive subjects that may have more opportunities for applying approximate transformations because they come from domains (e.g., linear algebra, image processing, machine learning) that may benefit more from approximate computing techniques [18], [65].

IV. Quantitative Analysis Results

This section contains answers to **RQ1**: how effective are approximate transformations as mutation operators, compared to conventional mutation operators, in terms of mutation scores, minimal mutants analysis, and selective mutation analysis.

A. Effectiveness by Mutation Scores

Table III shows the number of mutants generated and covered by tests per mutation operator for all subjects. There, the "Conv. Avg" column shows the average number of mutants generated by conventional mutation operators for each subject. Columns "LPM", "ITS" and "DTF" show the number of mutants generated by the approximate transformations. (We show only averages for conventional mutation operators due to space limits.) Figure 3 shows the average mutation score per operator across all subjects. Each bar represents a mutation operator; the rightmost three bars for approximate transformations—LPM, ITS, and DTF. The y-axis shows average mutation score

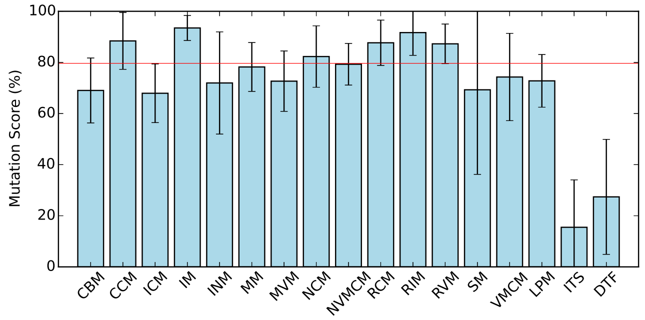

Fig. 3: Mutation scores per operator

per operator across all subjects. The red horizontal line is the average mutation score of all conventional mutation operators across all subjects. The error margin on each bar shows the standard deviation.

Loop perforation. On average, LPM generates only 275.89 mutants, compared with 1036.13 for conventional mutation operators. This is because there are much fewer loops (the only locations that LPM can mutate) relative to the number of locations that conventional mutation operators can mutate. The average mutation score for LPM (72.78%) is *slightly lower* than that of conventional mutation operators (79.65%) but it is not a low outlier, compared to other operators.

Precision degradation. The number of mutants generated by ITS and DTF are similar to that of LPM, relative to conventional mutation operators—an average of 569.22 and 357.89, respectively. These are significantly fewer than the average number of mutants generated by conventional mutation operators (1036.13). The average mutation scores for ITS and DTF are 15.49% and 27.39%, respectively (Figure 3). These are significantly lower than the average score of 79.65% for conventional mutation operators. In fact, ITS and DTF scores are the lowest among all operators (including LPM).

Discussion. The LPM mutation scores are closer to the mutation scores of conventional mutation operators, suggesting that LPM mutants are as easy/hard to kill as mutants generated from conventional mutation operators. The mutation scores for ITS and DTF are very low compared to the scores for conventional mutation operators. A further analysis of survived mutants in Section VI shows that this is not due to a high number of equivalent mutants, but rather to bad tests that do not exercise the code with large values crossing the precision boundaries. We perform a more detailed qualitative analysis on LPM, ITS, and DTF mutants in Section V.

B. Effectiveness by Minimal Sets of Mutants

We compute minimal mutant sets, as described in Section III-B, to see if mutants generated by approximate transformations are in the minimal mutant set, meaning they are not subsumed by other mutants. Table IV shows, for each subject, the breakdown of the counts of the minimal mutants. The column "Conv. Avg" shows the average number of minimal mutants generated from conventional mutation operators; the remaining columns show the number of minimal mutants for

TABLE IV: Minimal Mutants Per Operator

Project	Conv. Avg	LPM	ITS	DTF
commons-imaging	6.79	1	0	0
commons-io	37.07	1	1	0
HikariCP	4.57	1	0	0
imglib2	13.79	4	5	3
vectorz	18.36	14	1	9
jblas	2.21	2	0	1
OpenTripPlanner	15.29	2	0	1
la4j	13.57	17	3	17
meka	7.00	2	2	2
Average	13.18	4.89	1.33	3.67

TABLE V: Selective Mutation Operator Analysis

Project	# Conv. Operators	Approx Operators
commons-imaging	7	n/a
commons-io	9	ITS
HikariCP	8	n/a
imglib2	9	LPM,DTF
vectorz	10	LPM,ITS,DTF
jblas	4	DTF
OpenTripPlanner	8	n/a
la4j	9	LPM,ITS,DTF
meka	5	LPM,DTF

each approximate transformation. (We show only averages for conventional mutation operators due to the space constraints.) Approximate transformations show up in the minimal set of mutants—at least one of the last three columns, LPM, ITS, and DTF, is not `0`—for all subjects The average numbers of mutants contributed by LPM, ITS, and DTF to the set of minimal mutants are 4.89, 1.33, and 3.67, respectively. We conclude that, when used as mutation operators, approximate transformations can generate mutants that are not subsumed by mutants generated from conventional mutation operators.

C. *Effectiveness by Selective Mutation Analysis*

Table V presents the sets of sufficient mutation operators computed using the greedy selective mutation analysis algorithm presented in Section III-B. For each subject, we show the number of conventional mutation operators ("# Conv. Operators") and the selected approximate transformations ("Approx Operators") that are in the sufficient mutation operator set.

Approximate transformations appear among the sufficient mutation operators in six of the nine subjects (`commons-io`, `imglib2`, `vectorz`, `jblas`, `la4j`, and `meka`). The fact that approximate transformations end up in the sufficient mutation operator sets shows that they are important, since sufficient mutation operators are meant to be representative of all operators; tests good enough to kill these mutants are good enough to kill the mutants from all the other operators (Section III-B). Furthermore, when we perform selective mutation analysis with only conventional mutation operators, we find that the sufficient mutation operators for most subjects are the same as those corresponding to the number of conventional mutation operators from the "# Conv. Operators" column in Table V; the only exception was `meka`. From these subjects where approximate transformations are in the set of sufficient mutation

```
Set<Integer> toSet() {
  TreeSet<Integer> ss=new TreeSet<Integer>();
  for (int i=0; i<data.length; i++) {
    ss.add(data[i]);
  }
  return ss; }

void testSetCreate() {
  Index ind=Index.of(1,3,3,3,5);
  Set<Integer> s=ind.toSet();
  assertEquals(3,s.size());
  assertEquals(Index.createSorted(ind.toSet()),
                Index.of(1).includeSorted(s)); }
```

Fig. 4: *Initialization Loop* LPM code pattern from `vectorz` [4] and its corresponding test

operators, it seems approximate transformations are necessary to represent themselves, as the conventional mutation operators do not subsume the approximate transformations.

V. Code Patterns

This section provides answers to **RQ2**, on code patterns that approximate transformations reveal. We describe the results of our qualitative analysis to answer these questions:

RQ2.1: What code patterns do LPM mutants reveal?
RQ2.2: What code patterns do ITS/DTF mutants reveal?
RQ2.3: How are approximate transformations different from conventional mutation operators and how can they help mutation testing?

Answers to these questions help with understanding the type of computations affected by the proposed operators. Further, the answers guide the analysis in Section VI on practical impact.

Methodology. For LPM, we randomly sampled and inspected 5% of killed mutants and 5% of surviving mutants for each subject. ITS and DTF generate significantly higher numbers of mutants than LPM in some subjects, so we sampled and inspected only 1% (121 mutants) of their killed and surviving mutants. Table VI shows code patterns we found during inspection. Sections V-A and V-B further explain these patterns.

A. ***RQ2.1*** *Code patterns for LPM mutants*

Initialization loop. When a loop is used to initialize elements in a data structure, an LPM mutant that skips loop iterations may leave some elements uninitialized. Mutants of this pattern are killed by tests that rely on all elements to be initialized. However, we also find cases where such mutants survived, e.g., in method `Index#toSet()` of `vectorz`, shown in Figure 4. LPM skips some iterations in the loop that initializes elements of set `ss`. The only test for this method, `testSetCreate`, passes when LPM skips an iteration that adds a duplicated value to `ss`. The mutant produces the same result as the original code and reasoning about its survival can help improve the test suite with tests that kill this mutant by not having duplicated data.

Conditional computation on elements. As a loop iterates over all elements in a data structure, the loop body checks whether a property holds before performing some computation. We find examples of this pattern in `commons-imaging`, `vectorz`, and `jblas`. Consider the example in class `DoubleMatrix` of `jblas` shown in Figure 5.

TABLE VI: Code Patterns for Killed and Survived Loop Perforation and Precision Degradation Mutants

Approximate Transformation	**Surviving Code Patterns**	**#Mutants**	**Killed Code Patterns**	**#Mutants**
Loop Perforation	Initialization loop	3	Initialization loop	2
	Conditional computation on elements	14	Conditional computation on elements	22
	Computation on all elements	17	Computation on all elements	56
	Reduction	2	Reduction	9
Precision Degradation	Result is within a precision range	95	Result is outside a precision range	15
	Computing large values	1	Computing large values	8
			Indexing beyond the size of `short`	2
Total		132		114

```
public int argmin() {
  if (isEmpty()) { return -1; }
  double v = Double.POSITIVE_INFINITY;
  int a = -1;
  for (int i = 0; i < length; i++) {
   if (!Double.isNaN(get(i)) && get(i) < v) {
    v = get(i); a = i;
   } }
  return a; }

@Test
public void testArgMinMax() {
  A = new DoubleMatrix(4, 3, 1.0, 2.0, 3.0, 4.0, 5.0,
        6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0);
  assertEquals(0, A.argmin(), eps);
  assertEquals(11, A.argmax(), eps); }
```

Fig. 5: *Conditional Computation on Elements* LPM code pattern and its corresponding test from `jblas` [2]

The LPM mutant is not killed by `testArgMinMax()`, because the index with the minimum element is not skipped. The test suite can be improved by adding more tests with input data where the minimum element(s) are in a variety of different indices. In general, mutants that involve checking a property (or searching for a value) and potentially exiting the loop early tend to survive when tests do not check for both the cases when the property holds and when it does not (e.g., they only `assertTrue` but do not have some `assertFalse` for a different input). We find such surviving mutants in `vectorz`, `jblas`, `meka`, and `OpenTripPlanner`. Killed mutants of this pattern often modify data structures where most elements satisfy the property; skipping iterations misses important computations that affect test outcomes.

Computation on all elements. As a loop iterates over the elements in a data structure, its body performs an independent computation on each element. For example, the computation may involve setting values at corresponding indices in another array, or modifying the current element in the input array. Tests tend to kill LPM mutants for this code pattern when the test assertion iterates over all elements in the resulting array to check that the value at each index is correct. We observe loops of this pattern often in image-processing applications, which process matrices of pixels (e.g., `commons-imaging` and `imglib2`). In math applications with vector and matrix operations (e.g., `jblas`, `la4j`, and `vectorz`), these LPM mutants are commonly killed because the assertions check that every element has the expected value.

```
 public double reduce(double init, double[] data,
                     int offset, int length) {
   double result=init;
   for (int i=0; i<length; i++) {
     result=apply(result,data[offset+i]);
   }
   return result;
}
```

Fig. 6: *Reduction* LPM code pattern from `vectorz` [4]

Reduction. As a loop iterates over all elements in a data structure, the loop body applies a "`reduce`" operation, aggregating all values in the data structure to one representation. This pattern commonly occurs in math applications (e.g., `vectorz` and `jblas`). An example in class `Op2` of `vectorz` is shown Figure 6. The `reduce` method applies an operation `apply` to each element in a subarray of the `data` array. Tests typically kill such LPM mutants because the final result is a single value and the tests assert that the resulting value is equal to an expected value. It is also uncommon that the input array is such that the elements skipped are all identity elements w.r.t the applied operation. Most mutants of this pattern are killed; the few that survived are such that test inputs exercise only one loop iteration; therefore applying LPM is of no effect. To kill these mutants, developers need to add tests that execute the loop with more than one iteration.

B. *RQ2.2 Code patterns for ITS & DTF mutants*

Result is within/outside a precision range. When the specification of the operation is such that, for all allowed inputs, the result is always going to be within the degraded precision range, then such mutants should always survive. An example of a surviving DTF mutant is in `ColorConversions#convertHSLtoRGB` in `commons-imaging`, which converts HSL to RGB pixels by multiplying each pixel (a `double` between 0.0 and 1.0) by `255`. Degrading precision only slightly changes the accuracy of the result, within the tolerance bound of the test. On the other hand, when the test execution leads to values that go outside the precision range, such mutants are killed. We find surviving mutants of this pattern for DTF in `vectorz`, `OpenTripPlanner`, and `la4j`. We find instances where mutants of this pattern are killed in `meka`, `la4j`, `OpenTripPlanner`, `jblas`, and `imglib2`.

Computing large values. When computations involve large numbers, degrading the precision easily leads to different results, e.g., due to overflow. Hence, ITS mutants involving such computations tend to be killed by tests that expect a much larger value than the mutant returns. However, in `OpenTripPlanner`, we find an ITS mutant of this pattern that survived because the tests do not check that computed hash code values are correct and there is no collision in the hash codes computed at lower precision.

Indexing beyond the size of `short`. ITS mutants get killed when they cause the indices of data structures to exceed their bounds when large `int` values are cast to `short` values that overflow. For instance, class `CRSMatrix` in `la4j` has a method `set`. The test that kills the ITS mutant creates a matrix with dimensions greater than `Short.MAX_VALUE`. When the ITS mutant is run on the input matrix, the `int` to `short` precision degradation causes overflow, leading to an `ArrayOutOfBoundsException (AOOBE)`. Also, in class `CellRandomAccess` of `imglib2`, an overflow occurs when an `int` value used to walk through all positions in a large matrix is cast to a `short`, causing an `AOOBE`.

C. *RQ2.3 Comparing approximate transformations with conventional mutation operators*

LPM vs. conventional mutation operators. We check whether mutants generated by conventional mutation operators apply to each loop header on which an LPM mutant was generated. In total, seven conventional mutation operators (CBM, ICM, IM, NCM, NVMCM, RIM, RCM) can be applied on the same lines as LPM. Only two of these seven conventional mutation operators generate mutants that behave somewhat similarly to LPM mutants: Inline Constant Mutator (ICM) and Negate Conditionals Mutator (NCM). ICM changes the constant of the loop initialization to skip only the first iteration. NCM changes the loop condition to skip the entire loop body. LPM falls between the ICM and NCM in terms of the number of skipped iterations.

We conclude that LPM is complementary to conventional mutation operators; reasoning about their killed/surviving mutants helps developers generate new tests that exercise the code in new ways, improving their test suites (Section VI). Our conclusion holds for mutation operators in three other Java mutation tools: MuJava [43], Javalanche [60], and Major [34]. Replace_Constant from Javalanche and Constant Value Replacement from Major produce similar effects as PIT's ICM; Negate_Jump, and Unary_Operator from Javalanche, and Unary Operator Replacement, and Branch Condition Manipulation from Major have similar effects as PIT's NCM. Our understanding of code patterns exercised by LPM mutants enable us to perform such an analysis on mutation operators from other frameworks.

ITS/DTF vs. conventional mutation operators. We do not compare the precision degradation operators with the conventional mutation operators because the mutants they generate are not matched by any of the conventional mutation operators that modify arithmetic expressions. As our inspection in Section VI shows, these mutants provide guidance towards writing better tests that exercise boundary values.

Patterns for approximate transformations and tailored mutation. The patterns we identified open up a research opportunity to achieve additional savings in mutation testing. Our findings related to code patterns can enable performing tailored mutation testing [10] or specialized selective mutation [37] to find (parts of) applications where approximate transformations can be effective as mutation operators.

VI. Impact on Software Testing Practices

This section answers **RQ3**, on the practical impact on software testing of approximate transformations as mutation operators. We describe the results of our qualitative analysis to answer these questions:

RQ3.1: How often do surviving mutants from approximate transformations indicate that tests are bad, mutant is equivalent, or code is approximable?

RQ3.2: Do insights from inspecting surviving mutants from approximate transformations help developers?

A. ***RQ3.1*** *Bad test, equivalent mutant, or approximable code?*

Surviving mutants are traditionally regarded as either (1) signaling buggy, inadequate, or missing tests (*BadTest*) or (2) semantically equivalent to the original code, i.e., equivalent mutants. However, inspecting mutants generated from approximate transformations, we discovered a third possibility: the mutant survived because the original code is approximable (*ApproxCode*). That is, the mutant is semantically different from the original code but produces acceptable outcomes that are within a tolerable range. *This third interpretation applies to mutants from all operators, not just the ones generated by approximate transformations, changing the way mutation testing results should be interpreted in general.*

Of our inspected LPM mutants, 63.83% indicate bad tests (*BadTest*) and 19.15% indicate approximable code (*ApproxCode*); we find no equivalent mutants, and the remaining 17.02% are hard to inspect. Of our inspected ITS and DTF mutants, 53.13% indicate bad tests, 14.58% are equivalent, 11.46% indicate approximable code, and the remaining 20.83% are hard to inspect. Section V discussed the patterns that approximate transformations reveal, explaining the contexts in which those patterns signal approximable code. Section VI-B describes how *BadTest*s inspired better testing, and describes some pull requests we made to fix *BadTest*s.

We find mutants indicating *ApproxCode* in `vectorz`, `la4j`, `jblas`, and `meka`. An example from `la4j` is method `Matrix#shuffle()`, which makes a copy of an input matrix and uses a loop to randomly shuffle elements in the copy. Applying LPM to the shuffling loop is practically not observable, since the specification of the expected output is non-deterministic [63]. For ITS the surviving mutants for *ApproxCode* are equivalent, while for DTF the surviving *ApproxCode* mutants are within the precision range defined in the application.

Determining whether code is approximable is not an easy task. It is highly dependent on the quality of the oracles in the test suites that determine the ranges of acceptable output. Approximate computing often relies on the *usage context* (i.e., specific applications and application-level requirements) to determine if code is approximable. Such usage context is not available for the developers of general-purpose libraries (like most of our subjects) that can be used in a myriad of contexts. Therefore, the tests for these libraries are written in a conservative way, and consequently, our set of identified approximable patterns are necessarily conservative as well.

B. ***RQ3.2*** *Do insights from surviving mutants help improve testing practice?*

We find that surviving mutants of the *BadTest* category can be killed by adding tests that (1) achieve better loop coverage, (2) achieve better coverage of the loop condition, (3) exercise the code with larger inputs that cross the precision boundaries, or (4) check all output elements. Even though these insights are not new to the testing community, the real value lies in the fact that the approximate transformations are able to detect those problems, bringing them to the attention of the developer who might not have such considerations in mind. We also submitted pull requests that fix bad tests, to evaluate whether these insights can help developers improve their test suites. Seven of the 11 pull requests that we submitted were already integrated by developers into `vectorz`, `HikariCP`, `commons-imaging`, `imglib2` and `commons-io`. We next discuss the categories and the pull requests.

Achieve better loop coverage. 11 out of 30 LPM *BadTest* cases have tests that do not achieve full loop coverage, i.e., they do not have tests that exercise zero, one, *and* more than one loop iterations. As Table VII shows, the tests frequently cover either zero or one iteration. We discover the lack of full loop coverage while inspecting surviving LPM mutants in `vectorz`, `jblas`, `OpenTripPlanner`, and `commons-io`. The causes of low loop coverage that we observed are when (1) a test exercises the code with small inputs (e.g., one dimensional matrices) and (2) a test searches for a value that always happens to be the first element in the input data, so that the loop iterates only once before exiting. For example, in `jblas`, the method `argmin()` returns the index of the minimum element in a matrix. All tests that cover `argmin()` use input that is sorted in ascending order, so `argmin()` always returns `0`.

Achieve better coverage of loop condition. While inspecting the 14 surviving LPM mutants for the code pattern "Conditional computation on elements" (Section V-A), we find 12 of them are cases of *BadTest* in two categories: either the conditional check on the elements is never performed, or the conditional check is only performed on even-numbered iterations. In several cases the tests exercise the loop with only valid inputs, so the conditional check for errors that happen in the loop body is never performed. LPM helps direct the developer's attention into those critical parts of the code. An example from `commons-io` (SHA:`733dc26`) is shown in Figure 7. The method `resolveProxyClass()` from the class `ClassLoaderObjectInputStream` is only exercised by the test `testResolveProxyClass`. The test passes only one interface (`Comparable.class`) to the loop in `resolveProxyClass`. Thus, applying LPM to that loop will not cause `testResolveProxyClass` to fail, i.e., the resulting mutant survives, unless more than one interface is passed to `resolveProxyClass()`. Our pull request containing such a test was accepted by the `commons-io` developers.

TABLE VII: Lessons Learned For Better Testing Practices From LPM *BadTest* Cases

Bad Testing Pattern	#Cases	Learned Testing Practice
Zero iterations	7	Better loop coverage
One iteration	4	Better loop coverage
Loop condition (LC) Not Taken	8	Better coverage of LC
LC taken on even iterations	4	Better coverage of LC
Weak or no assertion	6	Check all output elements
Small Inputs	51	Exercise boundary values
Other[2]	1	-
Total	81	

```java
protected Class<?> resolveProxyClass(final String[] ints) {
  final Class<?>[] iClasses = new Class[ints.length];
  for (int i = 0; i < ints.length; i++) {
    iClasses[i] = Class.forName(ints[i], false, loader);
  }
  try {
    return Proxy.getProxyClass(loader, iClasses);
  } catch (final IllegalArgumentException e) {
    return super.resolveProxyClass(ints);
  } }

@Test
public void testResolveProxyClass() throws Exception {
  ...
  ClassLoaderObjectInputStream c =
    new ClassLoaderObjectInputStream(...);
  String[] i = new String[]{Comparable.class.getName()};
  Class<?> r = c.resolveProxyClass(i);
  assertTrue("...", Comparable.class.isAssignableFrom(r));
  c.close(); }
```

Fig. 7: Bad test example from `commons-io` [1]

Exercise boundary values. All *BadTest* cases for ITS and DTF are due to tests using small inputs. This means that the current tests do not use values that exceed the precision bounds of `short` for ITS and `float` for DTF, and we can write a test that can kill the mutant. A DTF example from `vectorz` is in the class `Quaternions`, which represents numbers from the quaternions number system using `double` precision. The method `mul()` computes the product of two quaternions. Mutants casting any of the arithmetic operations involved in the computation survive because the numerical values are very small.

Check all output elements. Multiple mutants are not killed because of the weakness or absence of assertions in the tests. For example, `meka` is a machine learning library. The tests cover the mutants, but most of the tests do not have assertions,

[2]This is a case in `meka`; a setter method resets the values in a matrix, but the new values are almost equal to the old values, so the effect of skipping iterations is not observable. A better test would exercise the function such that the difference between the new and old values is observable.

and coming up with strong assertions is non-trivial. Another example is in `vectorz` (shown in Figure 1a). The only test that covers the method `Matrix#swapRows()` does not check that all elements in the swapped rows are as expected (detailed discussion is in Section II). We submitted a pull request to add assertions and it has been accepted.

VII. Threats to Validity

The conventional mutation operators we use in comparison may not be representative of all mutation operators. Since ours is an initial study of the effectiveness of approximate transformations as mutation operators, we have used the set of conventional mutation operators that are available in PIT, which are used in both research and practice. Furthermore, in our qualitative analysis we examine mutation operators from three other popular mutation frameworks [34], [43], [60] and find our conclusions to still hold for those.

The approximate transformations that we evaluated are a subset of all approximate transformations and they may not be representative. To mitigate that, we model popular transformations that have been widely used in the approximate computing literature. Each transformation that we implement models some key properties of the original approximate transformations, i.e., dropping parts of computation (LPM), large magnitude errors (ITS), and small magnitude errors (DTF).

VIII. Related Work

A. Approximate computing

Approximate computing is an emerging area of research focusing on trading off (slightly) inaccurate results for performance gains (e.g. for energy usage). Some approximate computing techniques involve approximate hardware [40], [48], data types [58], sampling [6], [41], or code perforation [49], [65], all of which obtained significant performance with tolerable errors in specific domains. However, most of the existing work in approximate computing does not make explicit connections to software testing research. While our recent position paper argues in favor of using approximate computing to improve various software testing tasks [22], this work shows that approximate transformations are indeed useful in mutation testing.

Researchers also proposed sensitivity profilers [16], [48], [49], [56], [69], [72], which transform code, run it using representative input/output pairs, compare any differences, and suggest which parts of computations are approximable. Like sensitivity profiling, our approach transforms code and runs them on a set of tests, but our goal is different in several ways: (1) we study approximate transformations for mutation testing and compare with conventional mutation operators, (2) we execute programs on finer-grained unit tests, not coarse-grained integration tests, and (3) our results provide hints for improving tests, not just code.

B. Mutation testing

Mutation testing has been widely-studied for decades [20], [76]; Jia and Harman [33] provide a thorough background. Multiple techniques were developed for mutation testing at different levels and for different languages (e.g., source code [13], [24], [32], [60], intermediate representation [3], [28], [61]), etc. Many tools were also introduced for multiple programming languages, e.g., including C [19], [32], C++ [38], Java [36], [44], [45], [60], and others [15], [17], [29]. Many optimizations have been developed for mutation testing, including mutant schemata [70], weak mutation [54], and higher-order mutation [32]. Researchers have also proposed new mutation operators for different domains and use cases, such as for GUI-based applications [8], [53], embedded systems [68], class diagrams [25], Android applications [71], or fault-localization tasks [30]. We are the first to study approximate transformations in the context of mutation testing.

Researchers have studied how to improve the efficiency of mutation testing by techniques to only use the mutants that are hard-to-kill and representative of all mutants. Some heuristics for finding hard-to-kill mutants include minimal mutant analysis [11], [24], static analysis [55], or use of historical data [31]. Offutt et al. [51], [52] empirically found the set of sufficient operators, operators whose generated mutants are representative of mutants generated by the other operators, and others have extended this idea to various languages and paradigms, like concurrent code [23]. While these works have the goal to improve the efficiency of mutation testing, that is not the goal of our paper. We are focused on improving the *quality* of mutation testing by utilizing new mutation operators that give different insights into improving the test suite. We do, however, use the established existing techniques to evaluate how effective approximate transformations are compared against conventional mutation operators.

IX. Conclusions

We propose approximate transformations as mutation operators, and we compare them with conventional mutation operators. Specifically, we integrated loop perforation and precision degradation into an existing mutation testing framework, and we compared and analyzed the quality of those transformations when used as mutation operators. Our results show that approximate transformations generate mutants that are not subsumed by mutants generated by conventional mutation operators. Our qualitative analysis of a number of killed and surviving approximate transformations uncovered several code patterns that developers could use to enhance their test suites. The surviving mutants inspired proposing better testing practices and helped us submit 11 pull requests to fix bad tests.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and the participants of CS 591SE at the University of Illinois for constructive discussions on the material presented in this paper. This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1409423, CCF-1421503, CCF-1566363, CCF-1629431, CCF-1652517, CCF-1703637, and CCF-1704790.

References

[1] Commons-io. https://github.com/apache/commons-io.

[2] Jblas. https://github.com/mikiobraun/jblas.

[3] Real world mutation testing. http://pitest.org.

[4] Vectorz. https://github.com/mikera/vectorz.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[6] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[7] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen. Can testedness be effectively measured. In *FSE*, pages 547–558, 2016.

[8] E. Alegroth, Z. Gao, R. Oliveira, and A. Memon. Conceptualization and evaluation of component-based testing unified with visual GUI testing: An empirical study. In *ICST*, pages 1–10, 2015.

[9] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce. Evaluating non-adequate test-case reduction. In *ASE*, pages 16–26, 2016.

[10] M. Allamanis, E. T. Barr, R. Just, and C. Sutton. Tailored mutants fit bugs better. *arXiv preprint arXiv:1611.02516*, 2016.

[11] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *ICST*, pages 21–30, 2014.

[12] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

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

[14] ASM. http://asm.ow2.org/.

[15] T. A. Budd, R. J. Lipton, R. DeMillo, and F. Sayward. The design of a prototype mutation system for program testing. In *AFIPS*, pages 623–629, 1899.

[16] M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *ISSTA*, pages 37–48. ACM, 2010.

[17] W. Chan, S. C. Cheung, and T. Tse. Fault-based testing of database application programs with conceptual data model. In *QSIC*, pages 187–196, 2005.

[18] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *DAC*, page 113. ACM, 2013.

[19] M. E. Delamaro and J. C. Maldonado. Proteum-A tool for the assessment of test adequacy for C programs users guide.

[20] 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.

[21] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *PLDI*, 2015.

[22] M. Gligoric, S. Khurshid, S. Misailovic, and A. Shi. Mutation testing meets approximate computing. In *ICSE NIER*, pages 3–6, 2017.

[23] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *ISSTA*, pages 224–234, 2013.

[24] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. Measuring effectiveness of mutant sets. In *ICSTW*, pages 132–141, 2016.

[25] M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and O. Pastor. Mutation operators for UML class diagrams. In *CAiSE*, pages 325–341, 2016.

[26] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. Impact: imprecise adders for low-power approximate computing. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pages 409–414. IEEE Press, 2011.

[27] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *Test Symposium (ETS), 2013 18th IEEE European*, pages 1–6. IEEE, 2013.

[28] F. Hariri, A. Shi, H. Converse, S. Khurshid, and D. Marinov. Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level. In *ISSRE*, pages 105–115, 2016.

[29] S. Hong, T. Kwak, B. Lee, Y. Jeon, B. Ko, Y. Kim, and M. Kim. MUSEUM: debugging real-world multilingual programs using mutation analysis. *IST*, 82:80–95, 2017.

[30] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim. Mutation-based fault localization for real-world multilingual programs (t). In *ASE*, pages 464–475, 2015.

[31] L. Inozemtseva, H. Hemmati, and R. Holmes. Using fault history to improve mutation reduction. In *ESEC/FSE 2013*, pages 639–642, 2013.

[32] Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *TAIC PART*, pages 94–98, 2008.

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

[34] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *ISSTA*, pages 433–436, 2014.

[35] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665, 2014.

[36] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *ASE*, pages 612–615, 2011.

[37] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe. Analyzing the validity of selective mutation with dominator mutants. In *FSE 2016*, pages 571–582, 2016.

[38] M. Kusano and C. Wang. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *ASE*, pages 722–725, 2013.

[39] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. An extensive study of static regression test selection in modern software evolution. In *FSE*, pages 583–594, 2016.

[40] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving DRAM refresh-power through critical data partitioning. In *ASPLOS*, pages 213–224, 2011.

[41] L. Lou, P. Nguyen, J. Lawrence, and C. Barnes. Image perforation: Automatically accelerating image pipelines by intelligently skipping samples. *SIGGRAPH*, 35(5):153:1–153:14, 2016.

[42] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. How does regression test prioritization perform in real-world software evolution? In *ICSE*, pages 535–546, 2016.

[43] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: An automated class mutation system. *STVR*, 15(2):97–133, 2005.

[44] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava: a mutation system for Java. In *ICSE*, pages 827–830, 2006.

[45] L. Madeyski and N. Radyk. Judy-A mutation testing tool for Java. *IET software*, 4(1):32–42, 2010.

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

[47] S. Misailovic. Exploring the Effectiveness of Loop Perforation for Quality of Service Profiling. Technical report, MIT, 2010.

[48] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *OOPSLA*, pages 309–328, 2014.

[49] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *ICSE*, pages 25–34, 2010.

[50] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi. Phase-aware optimization in approximate computing. In *CGO*, 2017.

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

[52] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *ICSE*, pages 100–107, 1993.

[53] R. A. P. Oliveira, E. Algroth, Z. Gao, and A. Memon. Definition and evaluation of mutation operators for GUI-level mutation analysis. In *ICSTW*, pages 1–10, 2015.

[54] M. Papadakis and N. Malevris. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Control*, 19(4):691–723, 2011.

[55] M. Patrick, M. Oriol, and J. A. Clark. Messi: Mutant evaluation by static semantic interpretation. In *ICST*, pages 711–719, 2012.

[56] P. Roy, R. Ray, C. Wang, and W. F. Wong. Asac: Automatic sensitivity analysis for approximate computing. In *SIGPLAN/SIGBED LCTES*, 2014.

[57] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *SC*, page 27, 2013.

[58] A. Sampson, W. Dietl, E. Fortuna, and D. Gnanapragasam. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, pages 164–174, 2011.

[59] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices*, 49(6):53–64, 2014.

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

[61] E. Schulte. llvm-mutate. http://eschulte.github.io/llvm-mutate/.

[62] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *FSE*, pages 246–256, 2014.

[63] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*, 2016.

[64] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *FSE*, pages 237–247, 2015.

[65] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, pages 124–135, 2011.

[66] Spoon. http://spoon.gforge.inria.fr/.

[67] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali. Proactive control of approximate programs. In *ASPLOS*, 2016.

[68] A. Sung, B. Choi, W. E. Wong, and V. Debroy. Mutant generation for embedded systems using kernel-based software and hardware fault simulation. *IST*, 53(10):1153–1164, 2011.

[69] A. Thomas and K. Pattabiraman. Llfi: An intermediate code level fault injector for soft computing applications. In *SELSE*, 2013.

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

[71] M. L. Vásquez, G. Bavota, M. Tufano, K. Moran, M. D. Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. Enabling mutation testing for android apps. In *ESEC/FSE*, pages 233–244, 2017.

[72] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *MICRO*, pages 1–14. IEEE, 2016.

[73] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12. ACM, 2013.

[74] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *ICSE*, pages 919–930, 2014.

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

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