Evaluating Non-adequate Test-Case Reduction

Mohammad Amin Alipour[1], August Shi[2], Rahul Gopinath[1],
Darko Marinov[2], and Alex Groce[1]
[1]School of Electrical Engineering and Computer Science, Oregon State University, USA
[2]Department of Computer Science, University of Illinois at Urbana-Champaign, USA
{alipourm,gopinathr,agroce}@oregonstate.edu, {awshi2,marinov}@illinois.edu

ABSTRACT

Given two test cases, one larger and one smaller, the smaller test case is preferred for many purposes. A smaller test case usually runs faster, is easier to understand, and is more convenient for debugging. However, smaller test cases also tend to cover less code and detect fewer faults than larger test cases. Whereas traditional research focused on reducing *test suites* while preserving code coverage, recent work has introduced the idea of reducing individual *test cases*, rather than test suites, while still preserving code coverage. Other recent work has proposed non-adequately reducing test suites by not even preserving all the code coverage. This paper empirically evaluates a new combination of these two ideas, *non-adequate reduction of test cases*, which allows for a wide range of trade-offs between test case size and fault detection.

Our study introduces and evaluates $C\%$-*coverage* reduction (where a test case is reduced to retain at least $C\%$ of its original coverage) and N-*mutant* reduction (where a test case is reduced to kill at least N of the mutants it originally killed). We evaluate the reduction trade-offs with varying values of $C\%$ and N for four real-world C projects: Mozilla's `SpiderMonkey` JavaScript engine, the `YAFFS2` flash file system, `Grep`, and `Gzip`. The results show that it is possible to greatly reduce the size of many test cases while still preserving much of their fault-detection capability.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

test reduction, test adequacy, coverage, mutation testing

1. INTRODUCTION

Smaller test cases are, in many ways, preferable to larger test cases. For example, smaller test cases tend to run faster, which can improve the efficiency of running test suites [10], i.e., sets of of individual test cases. Smaller, simpler test cases are also easier to understand and enable more effective debugging. This was the initial motivation for delta-debugging [27]—a technique for reducing the size of failing test cases. Because of the advantages of smaller test cases, random test generation is often combined with delta-debugging, making research on effective reduction techniques itself an important topic [5,12,16,19]. Test suites with small test cases (that focus on separate functional properties) also make it possible for test-case selection [7] and prioritization to operate more effectively than when applied to test suites mostly consisting of large, complex test cases.

While smaller test cases have advantages, it is also true that smaller test cases, all else being equal, detect fewer faults than larger test cases [3]. The trade-off between size and effectiveness for individual test *cases* is similar to the trade-off between smaller and larger test *suites*. Researchers have extensively studied test-*suite* reduction [13, 14, 20, 22, 26], which removes entire test cases from test suites.

The problem of *test-suite reduction* is to reduce a given test suite while preserving most of its fault-detection capability. Various techniques have been proposed, and many are summarized in a survey by Yoo and Harman [26]. Test-suite reduction trades off reduced fault-detection capability (most often measured by the number of killed mutants) for reduced test-suite size (typically measured by the number of test cases). Traditional techniques *completely* preserve some property of a test suite, e.g., its code coverage, while removing test cases that are redundant and do not contribute to that property. Recently, we evaluated non-adequate test-suite reduction [22] that only *partially* preserves the property of interest, e.g., preserves 90% of code coverage.

The problem of *test-case reduction* is to reduce an individual test case while preserving most of its fault-detection capability. Reducing a test case essentially requires "slicing and dicing" the atomic parts that make the test case. For example, if a test case is a unit test composed of a sequence of function calls, reduction usually involves removing function calls. If a test case is defined by an input file, reduction can involve removing characters from the file. Note that measuring the *size* of a test case is inherently project-specific, depending entirely on the semantics of test cases, whereas the size of test suites can be defined in a project-agnostic way as the number of test cases in the test suite (though not perfectly correlated with the time to execute the tests). While test-suite reduction has been studied in depth at least since 1993 [14], test-case reduction research is much more recent.

Zeller and Hildebrandt proposed delta-debugging [27], the best known test-case reduction technique, usually applied to reduce a failing test case to a minimal test case that still fails. Recently, we proposed *cause reduction* [10, 11] as a generalization of delta-debugging, and used it to reduce a (passing or failing) test case while preserving its original coverage. Cause reduction *completely* preserves coverage: the reduced test case has to cover all the code elements that the original test case covered (and can potentially cover even more). We call such reductions *adequate* because they preserve 100% of some property. Note that "adequate" in our context refers to the relationship between the reduced and original test cases, although the original test case itself may provide far from adequate code coverage.

The utility of non-adequate reduction for test suites [22] naturally suggests that non-adequate reduction may be useful for test cases as well. Non-adequate reduction, either for test suites or test cases, greatly enlarges the number of points to explore in trading off size and fault-detection capability. For test cases, requiring adequacy limits how much size can be reduced (some test cases cannot be reduced substantially without sacrificing at least some coverage or killed mutants) and increases the time required to reduce test cases (because searching for an adequate reduction is often harder than finding a "good enough" reduction).

Combining the recent ideas of non-adequate reduction for test suites [22] and adequate reduction for test cases [10, 11], this paper empirically evaluates a new combination: *non-adequate reduction for test cases*. To the best of our knowledge, ours is the first such evaluation.

Specifically, we evaluate *C%-coverage* reduction (where a test case is reduced to retain at least C% of its original coverage) and *N-mutant* reduction (where a test case is reduced to kill at least a given set of N mutants it originally killed). Both reductions reduce a larger test case to a smaller test case while only *partially* preserving some property. Hence, we call these reductions "non-adequate" because they do not necessarily preserve completely either the code coverage or all mutants killed. However, the reduced test case *could*, in theory, cover code elements or kill mutants that the original test case does not, even if the reduced test case does not cover all code elements or kill all mutants that the original test case did; in fact, the reduced test case can even cover *more* code or kill *more* mutants.

Non-adequate test-case reduction further generalizes previously proposed test-case reductions. By parameterizing the level to which the reduced test case needs to preserve a property, we allow more freedom to explore trade-offs between size reductions and preservation of fault-detection capability [22, 25]. For example, cause reduction [10, 11] becomes just a special case of our C%-coverage with $C = 100$. Preserving to kill only one mutant that encodes some fault (N-mutant with $N = 1$) can mimic delta-debugging. At the other extreme, setting N to equal the total number of all mutants originally killed results in a very strict test-case reduction that preserves *all* mutants killed; however, such reduction may be prohibitively expensive to perform (and would likely provide very little reduction unless test cases have excessive redundancy), so our evaluation concerns only small values for N.

We evaluate non-adequate test-case reduction on four real-world C projects: Mozilla's `SpiderMonkey` JavaScript engine, the `YAFFS2` flash file system, `Grep`, and `Gzip`. We used manual `Grep` test cases, and automatically generated test cases for the other projects. We evaluate C%-coverage for various levels of C%, from 70% to 100%. We evaluate N-mutant with (1) randomly selected mutants for various values of N from 1 to 32, and (2) mutants that are hard to kill based on the minimal mutant set [1]. We measure size reduction, code coverage, and mutants killed, with the latter two[1] used as proxies for fault-detection capability. Our results show that in many cases, non-adequate test-case reduction can substantially reduce the size of the given test cases while still preserving considerable fault-detection capability. Perhaps most interestingly, when performing C%-coverage reduction, the largest gain in size reduction for all cases comes when C% changes from 100% to 95%; the gain is typically twice as large as for any other C% change. This gain does *not* result in a similarly large loss in mutation detection. In brief, simply giving up on perfection enables a larger reduction in size than the associated reduction in effectiveness. Additionally, preserving even a small number N of mutants killed usually indirectly preserves a large fraction of all other mutants killed, often more than 70%.

This paper makes the following contributions:

- **Novel test-case reduction approach:** We define two types of *non-adequate test-case reduction*: C%-coverage and N-mutant reduction.
- **Evaluation of reduction trade-offs:** Using four real-world C projects, we evaluate the relationship between the size reductions obtained with varying parameters for these reductions, and the code coverage and killed mutants for reduced test cases relative to the original, unreduced test cases.

2. Non-Adequate Test-Case Reduction

We next describe our test-case reductions in more detail. We use t_o to denote the original test case and t_r to denote the reduced test case. We use t to denote an arbitrary test case, $Cov(t)$ to denote the set of statements[2] covered by t, $Mut(t)$ to denote the set of mutants killed by t, $|S|$ to denote the cardinality of the set S, and $Size(t)$ to denote the size of t. Measuring the size of a test case is specific to the project or the format of test cases; Section 4.1.1 precisely defines size for the projects used in our evaluation. Conceptually, we define size as the number of atomic *parts* that a test case has. The parameterized nature of parts is taken from the original delta-debugging work [27]. In some projects, parts are function calls; in other projects, they are lines or characters in a file/string; and in rare cases, they may be much more complex, e.g., defined by a grammar. For example, reduction of test cases that are computer programs (e.g., an input to a compiler) [19] often relies on a semantically involved notion of part.

The high-level goal of test-case reduction is to produce a reduced test case t_r with size smaller than the size of t_o, i.e., $Size(t_r) < Size(t_o)$ (and ideally $Size(t_r) \ll Size(t_o)$), such that t_r still retains (either completely or partially) some desirable property of t_o. That is, for some notion of "quality",

[1] Although they are not ideal proxies, code coverage is often used by developers to evaluate quality of test cases and both are commonly used to evaluate test cases in research.

[2] While we present and evaluate C%-coverage only for statement coverage, it can generalize, e.g., to branch coverage.

t_r has similar quality to t_o. t_o itself may have good or bad quality, but t_r should not have much worse quality than t_o. While in principle the reduction process can stop at various steps (and in the limit, even the original test case can be considered a reduced version of itself), we are interested in so called "1-minimal" test cases [27] where no single part of t_r can be removed without losing some desired property.

2.1 Reduction Algorithm

The test-case reduction algorithm we use is derived from the original delta-debugging [27] algorithm, and we modify it to support non-adequate test-case reduction. Delta-debugging takes as input a *failing* test case and reduces it by removing parts that are not relevant for the failure. A generalized algorithm for cause reduction [10, 11] extends delta-debugging to reduce a test case with respect to *any* property, not just failure, that can be detected when running the test case. The most direct application of cause reduction is to *completely* preserve code coverage.

At a high level, the delta-debugging algorithm (described in detail by Zeller and Hildebrandt [27] and extended in the work on cause reduction [10,11]) iteratively splits a test case into multiple candidate test cases. At each of these steps, the algorithm checks if any candidate satisfies the desired property (which, in traditional delta-debugging, is whether the test case fails). If there is a satisfactory candidate, it becomes the new base test case to be reduced further in the future steps. If no candidate is satisfactory, the granularity for splitting is increased, until the algorithm determines that the test case is 1-minimal: removing any single part produces a test case that does not satisfy the property. In this paper, we further generalize delta-debugging and cause reduction by allowing the candidate test case to only *partially preserve* some property.

2.2 $C\%$-coverage Reduction

We relax the requirement from cause reduction [10, 11]—that the reduced test case t_r preserve all code coverage obtained by the original test case t_o—with the requirement that t_r preserve at least $C\%$ of coverage obtained by t_o. Reducing large test cases to preserve all (statement) coverage can be prohibitively expensive. For example, we previously reported that cause reduction of a single test case for the GCC compiler could take days [10, 11]. Moreover, preserving 100% of the coverage *may not be necessary*, because a test case that preserves less may still have acceptable quality. Hence, we propose $C\%$-coverage reduction:

Definition 1. *$C\%$-coverage test-case reduction produces a reduced test case t_r that covers at least $C\%$ of the statements covered by the original test case t_o:*

$$\frac{|Cov(t_r) \cap Cov(t_o)|}{|Cov(t_o)|} \geq C\%$$

Note that the percentage is determined by the coverage of the *original* test case and *not* by coverage over *all* statements in the code under test. The property is not $\frac{|Cov(t_r)|}{|Cov(t_o)|} \geq C\%$, because t_r could then end up covering statements unrelated to those covered by t_o. Coverage-based cause reduction [10, 11] can be (re)defined as $C\%$-coverage with $C = 100$: $|Cov(t_r) \cap Cov(t_o)|/|Cov(t_o)| = 100\%$, or equivalently $Cov(t_r) \supseteq Cov(t_o)$. $C\%$-coverage does not impose any requirements over statements *not* covered by the original test case: the reduced test case may or may not cover those statements. Also, $C\%$-coverage does not (directly) require any relationship between $|Cov(t_r)|$ and $|Cov(t_o)|$, so it can even happen that $|Cov(t_r)| > |Cov(t_o)|$ if t_r covers some statements that t_o does not cover.

2.3 N-mutant Reduction

We define N-mutant reduction in a similar fashion, but with three important differences: (1) N-mutant uses killed mutants instead of covered statements; (2) N-mutant preserves the ability of a test case to kill an absolute number N of mutants rather than a relative ratio of mutants; and (3) N-mutant considers the *same* set of selected mutants for all steps of the reduction algorithm:

Definition 2. *N-mutant test-case reduction produces a reduced test case t_r that kills a specific set of N mutants selected from the set of $Mut(t_o)$, where typically $N \ll |Mut(t_o)|$.*

The difference (3) from $C\%$-coverage is largely motivated by the cost of determining the complete set of mutants killed for every candidate test case at each step of the reduction algorithm. We did initially experiment with allowing the set of mutants to change, while requiring only that the number of mutants be preserved through reduction steps be at least N. However, by allowing the algorithm to only preserve at least any N mutants, it can be necessary to run a large number of mutants at each step of the reduction algorithm (until at least N mutants are killed or *all* mutants are run and N are not killed). As a result, the time to perform non-adequate test-case reduction was often prohibitively long. Again, we only require the selected N mutants to be a subset of $Mut(t_o)$. Mutants other than those in the selected set may or may not be killed by t_r.

3. METRICS

We describe three metrics for evaluating the effectiveness of test-case reduction: Size Reduction Rate (SRR), Coverage Preservation Rate (CPR), and Mutant(-killing) Preservation Rate (MPR). We define all metrics such that higher values are better and values are normalized to the range 0%–100%.

3.1 Size Reduction Rate (SRR)

The goal of test-case reduction is to reduce the size of a test case. As such, it is important to measure how much smaller the reduced test case is compared to the original test case. Recall that $Size(t)$ denotes the size of a test case t, i.e., the number of the atomic parts that the test case has.

Definition 3. *For an original test case t_o and its reduced test case t_r, Size Reduction Rate (SRR) is:*

$$SRR(t_o, t_r) = \frac{Size(t_o) - Size(t_r)}{Size(t_o)}$$

A higher SRR is desirable as it indicates that more parts have been removed from the test case, resulting in a smaller reduced test case.

3.2 Coverage Preservation Rate (CPR)

Our reduction is non-adequate test-case reduction, so we need some metrics to measure how much fault-detection capability the reduced test case loses compared to the original

test case. Structural code coverage, although not an ideal proxy for fault-detection capability [8, 9, 15], is commonly used to evaluate the quality of test cases: the more code a test case covers, the higher the chance it can detect a fault. We therefore use statement coverage as one way to evaluate quality. Recall that $Cov(t)$ denotes the set of statements covered by a test case t.

Definition 4. *For an original test case t_o and its reduced test case t_r, Coverage Preservation Rate (CPR) measures the ratio between the number of statements covered by both t_r and t_o and the number of statements covered by t_o:*

$$CPR(t_o, t_r) = \frac{|Cov(t_r) \cap Cov(t_o)|}{|Cov(t_o)|}$$

A higher CPR is desirable as it indicates the reduced test case covers a larger subset of statements covered by the original test case. Note that while a reduced test case can potentially cover more statements than the original test case, CPR is limited to 100% as it considers only the statements covered by t_o.

3.3 Mutant Preservation Rate (MPR)

MPR is essentially the same as CPR, except measured with respect to mutants killed, not statements covered:

Definition 5. *For an original test case t_o and its reduced test case t_r, Mutant Preservation Rate (MPR) measures the preservation of mutants killed by t_r relative to the mutants killed by t_o:*

$$MPR(t_o, t_r) = \frac{|Mut(t_r) \cap Mut(t_o)|}{|Mut(t_o)|}$$

A higher MPR is desirable as it indicates the reduced test case is better at killing mutants among those that the original test case kills. Like CPR, MPR is relative to the original test and cannot exceed 100%.

3.4 Reduction Requirements vs. Metrics

Although both of the reduction algorithms and the metrics are based on coverage and mutants, note that the requirements for reduction are *not* the same as the metrics used to evaluate the reduced test cases. Therefore, we cannot *a priori* tell how high or low the metrics will be for all reductions. For C%-coverage reduction, we know that CPR will be at least C%, but it could be much higher (up to 100%), and MPR could in theory range from (literally) 0 to 100%. For N-mutant reduction, we know that MPR will be at least $N/|Mut(t_o)|$, but it could be much higher (in our experiments, even when $N/|Mut(t_o)| < 1\%$, MPR can be quite high), and CPR could range from almost 0 to 100%.

4. EVALUATION METHODOLOGY

We describe the projects, test cases, and mutants used in our evaluaton (Section 4.1) and the experimental setup (Section 4.2). Our experiments ran on a high-performance cluster of commodity computing nodes; each node had 6–12 2.6Ghz Intel Xeon cores.

4.1 Projects

Table 1 lists the projects used in our evaluation. We tabulate the project name, the number of non-comment lines of code, the number of test cases used in our evaluation, what an atomic part is, the total number of mutants used, and the minimum and maximum number of mutants killed by each test case. The last two columns are the number of tests in a randomly generated test pool for each project and the number of minimal mutants determined for each project; these last two columns are metrics relevant for our analysis involving minimal mutants (Section 4.1.3). We use four small- to medium-size C projects: `SpiderMonkey` is Mozilla's JavaScript engine, `YAFFS2` is a popular flash file system used in the early Android versions, `Grep` is the standard Unix utility for searching files, and `Gzip` is the standard Unix utility for compressing/decompressing files.

4.1.1 Test Cases

We use automatically generated test cases for `SpiderMonkey`, `YAFFS2`, and `Gzip`, and we use manually written test cases for `Grep`. The `SpiderMonkey` test cases are JavaScript programs randomly generated using the highly successful `jsfunfuzz` [21] fuzzer. The `YAFFS2` test cases are sequences of API calls to the file system, randomly generated using a publicly available test generator for `YAFFS2` that has been used by several research projects on test generation [4,10,11]. The `Gzip` test cases are files that have 500 to 3,500 random bytes. For `Grep`, we use the manually written test cases obtained from `SIR` [6]; each test case consists of command-line arguments to `Grep`.

How best to measure the *size* of a test case is an open question in software testing research. Researchers use a variety of metrics, such as the number of API calls, execution time, or number of assertions. As in our previous work [10,11], we define size as the number of atomic *parts* of atest case. The concrete part differs from one project to another, as summarized in Table 1. A part is a JavaScript code fragment in the generated program for `SpiderMonkey`, one API call in the generated sequence of API calls for `YAFFS2`, a character in the command-line arguments for `Grep`, and simply one byte in the input file for `Gzip`.

We limit the size of generated test cases to enable experiments to finish in a reasonable amount of time. Time complexity of the basic delta-debugging algorithm is quadratic [27] in the number of parts in a test case. For `SpiderMonkey`, `YAFFS2`, and `Gzip`, we control the number of parts based on the specific limits from our initial experiments, trying to finish most test-case reductions within 30 minutes. In particular, we limit each `SpiderMonkey` test case to be exactly 200 lines of JavaScript code, each `YAFFS2` test case to be a sequence consisting of exactly 200 API calls, and each `Gzip` test case to be a file consisting of at most 3,500 bytes. For `Grep`, we use *all* the test cases manually written by others [6], and we do not limit their sizes; the largest test case for `Grep` has 146 characters in the command-line arguments.

4.1.2 Mutants

We use a mutation-testing tool for C code developed by Andrews *et al.* [2] and used in many previous studies. Quoting [2], the tool provides the following four classes of mutation operators: "(1) Replace an integer constant I by 0, 1, -1, $((I) + 1)$, or $((I) - 1)$; (2) Replace an arithmetic, relational, logical, bit-wise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class; (3) Negate the decision in an if or while statement; and (4) Delete a statement."

Table 1: Four projects used in our evaluation and some statistics of their test cases and mutants

project	NCLOC	# test cases	definition of an atomic part	# mutants	min killed	max killed	test pool	# minimal mutants
`SpiderMonkey`	81,920	99	A statement of JavaScript program	69,067	8101	12825	850	256
`YAFFS2`	10,356	99	One API call	15,046	2071	3439	1000	57
`Grep`	8,433	112	A character in command-line arguments	7,591	19	993	840	99
`Gzip`	5,129	73	A byte in the input file	7,175	1813	2046	1000	32

Each mutant was compiled with *GCC* using the highest optimization `-O3` and compared with other binaries to avoid trivially equivalent mutants [18]. About 15% of the generated mutants were found to be trivially equivalent. Table 1 shows the number of mutants for each project and the minimum and maximum number of mutants killed by each test case. A mutant is considered killed if its output (including `stdout`, `stderr`, and produced files) differs from the output of the original code.

4.1.3 Minimal Mutants

To evaluate N-mutant reduction, we use two methods to select mutants. The first method is simple random sampling: we select N mutants from the set of mutants killed by the test case. In the second method, we wanted to alleviate the impact of redundant and trivial mutants on the results. Redundant mutants are those mutants that are semantically equivalent to one another, albeit syntactically different. Trivial mutants are those mutants that are killed by a majority of test cases. The impact of these two kinds of mutants can be alleviated by using *minimal mutant sets* introduced by Ammann *et al.* [1].

A minimal mutant set is computed based on an original set of killed mutants and a test suite, The first step is to construct a minimal test suite from the original test suite, i.e., a subset of the original test suite that kills all the mutants killed by the original test suite. Removing any test case from the minimal test suite means failing to kill some mutant. Given a minimal test suite, a minimal mutant set is the smallest subset of mutants from the original mutant set such that killing all the mutants from the minimal mutant set (using the minimal test suite) also kills all the mutants from the original set of killed mutants.

We generated minimal mutant sets for projects as follows: first, we generated a large test pool of random test cases for each project. We used these larger pools because minimal mutants require (almost) adequate test suites to ensure that useful mutants are not removed. Then, we obtained the complete set of mutants killed by each test pool. We minimized each test pool with respect to its corresponding project's set of mutants to obtain the minimal set of test cases from the test pool that kill all those mutants, using a greedy test-suite reduction algorithm [26]. Using this minimal set of tests, we minimized the mutant set to obtain the minimal mutant set. Table 1 shows the number of test cases in these pools and the sizes of the minimal mutant sets. We later compare randomly selected mutants with minimal mutants by reducing the test cases taken from the large test pool for each project.

4.2 Experimental Setup

For C%-coverage, we perform experiments with the non-adequacy value C chosen from the set $\{70, 80, 90, 95, 100\}$. For each original test case, we create a reduced test case that preserves at least C% of the statements covered by the original test case. We use GCov to obtain the set of statements covered by each test case.

For N-mutant, we perform experiments with the non-adequacy value N chosen from the set $\{1, 2, 4, 8, 16, 32\}$. For each original test case, we first determine what mutants the test case kills and then randomly select N of those mutants (for a small number of test cases that kill fewer than N mutants, we use all mutants) to create a reduced test case that preserves these N selected mutants. To compare randomly sampled mutants with the harder to kill minimal mutants, we take each test case from the minimal test suite (constructed from the large test pool is described in Section 4.1.3) and reduce the test case while preserving one randomly selected mutant and then reduce it to perserve the one mutant ($N = 1$) from the minimal mutant set that the test case uniquely contributes to the minimal mutant set. If a test case kills no mutants in the minimal mutant set, we do not reduce the test case at all.

Performing test-case reduction can take a long time for some test cases. We limit reduction to 30 minutes per test case. We observed that N-mutant test-case reduction starts having many timeouts when N gets greater than about 40, so we restrict our choices of N to values less than 40. The experiments ignore test cases whose reduction times out.

For each reduced test case, we further generate three randomly reduced test cases that have *exactly the same size* as the reduced test case. We create such a randomly reduced test case by starting from the original test case and iteratively choosing to remove (uniformly randomly selected) one part at a time until the resulting test case has the same number of parts as the reduced test case. We perform random test-case reduction merely as some kind of baseline to show the benefits of preserving benefits of a test case; we do not actually recommend actually using random test-case reduction in practice.

5. RESEARCH QUESTIONS

Our evaluation addresses the following questions about the effects of non-adequate test-case reduction:

- RQ1: How much are test cases reduced (SRR)?
- RQ2: How much are code coverage and mutants killed preserved (CPR and MPR)?
- RQ3: How do SRR, CPR, and MPR trade off?
- RQ4: How do CPR and MPR for our approaches compare to CPR and MPR for *random* test-case reduction?

5.1 RQ1: SRR

Figures 1 and 2 summarize the results for SRR on the test cases reduced using C%-coverage and N-mutant, respectively. For each project and level of C and N, the boxplots show the distribution of SRR. From the figures, we see that both approaches can greatly reduce the size of

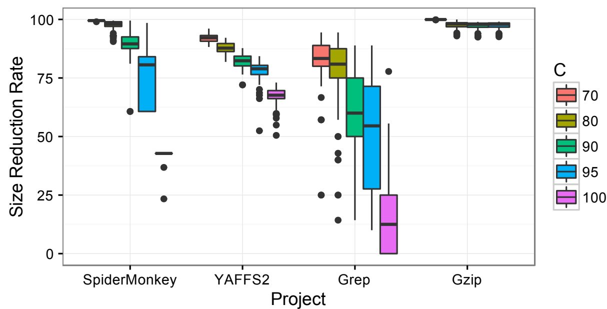

Figure 1: SRR for C%-coverage

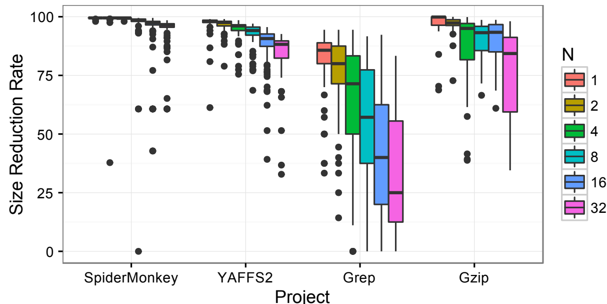

Figure 2: SRR for N-mutant

test cases. In most configurations, the median SRR for all test cases reduced using either C%-coverage or N-mutant is greater than 50%: the size of a reduced test case is usually less than half the size of the original test case. For both reductions, `Grep` behaves somewhat differently, with median SRR. The likely cause is the small size of test cases in `Grep`: most have < 100 characters.

SRR decreases when C or N increases, as expected. We emphasize that SRR for $C = 100$ is particularly low compared with SRR for other values; allowing coverage to miss even a small set of statements increases SRR substantially. For example, for `Grep`, the median SRR for $C = 100$ and $C = 95$ differ by over 30pp[3].

5.2 RQ2: CPR and MPR

Figures 3 and 4 summarize the results for CPR on the test cases reduced using C%-coverage and N-mutant, respectively. CPR is, of course, always at least as high as the C% value given to the reduction. From Figure 3, for `SpiderMonkey` and `YAFFS2`, CPR is almost exactly the given C%, but for the other two projects, CPR is sometimes much higher. Overall, the median CPR across different values of C across all projects ranges from 70.07% to 100%.

Figure 4 illustrates the relation between different values of N and CPR. The range of median CPR here goes from 78.39% to 100%, which is quite high, showing that preserving even just one mutant leads to CPR close to 80%.

Figures 6 and 7 summarize the results for MPR on the test cases reduced using C%-coverage and N-mutant, respectively. For C%-coverage reduction, the median MPR ranges from 41.51% to 100% across all projects and all val-

[3]The "pp" metric (from "percentage points") represents differences in values that are already expressed as percentages.

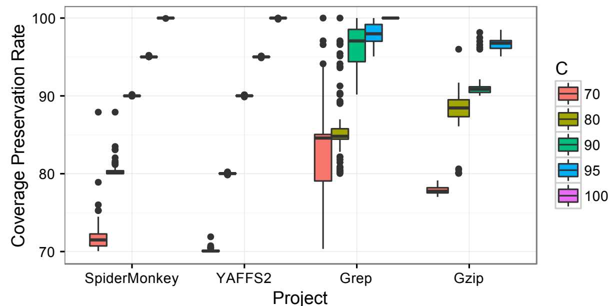

Figure 3: CPR for C%-coverage

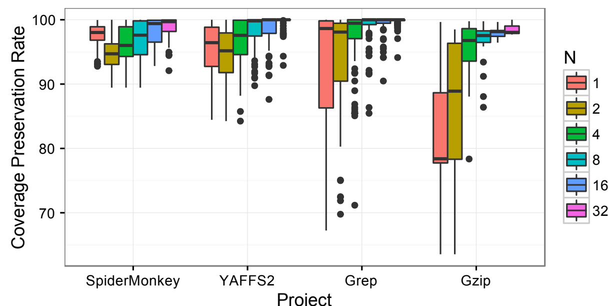

Figure 4: CPR for N-mutant

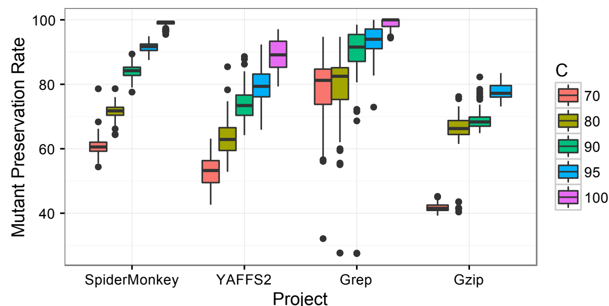

Figure 6: MPR for C%-coverage

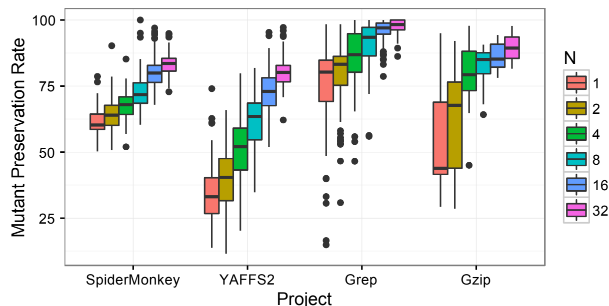

Figure 7: MPR for N-mutant

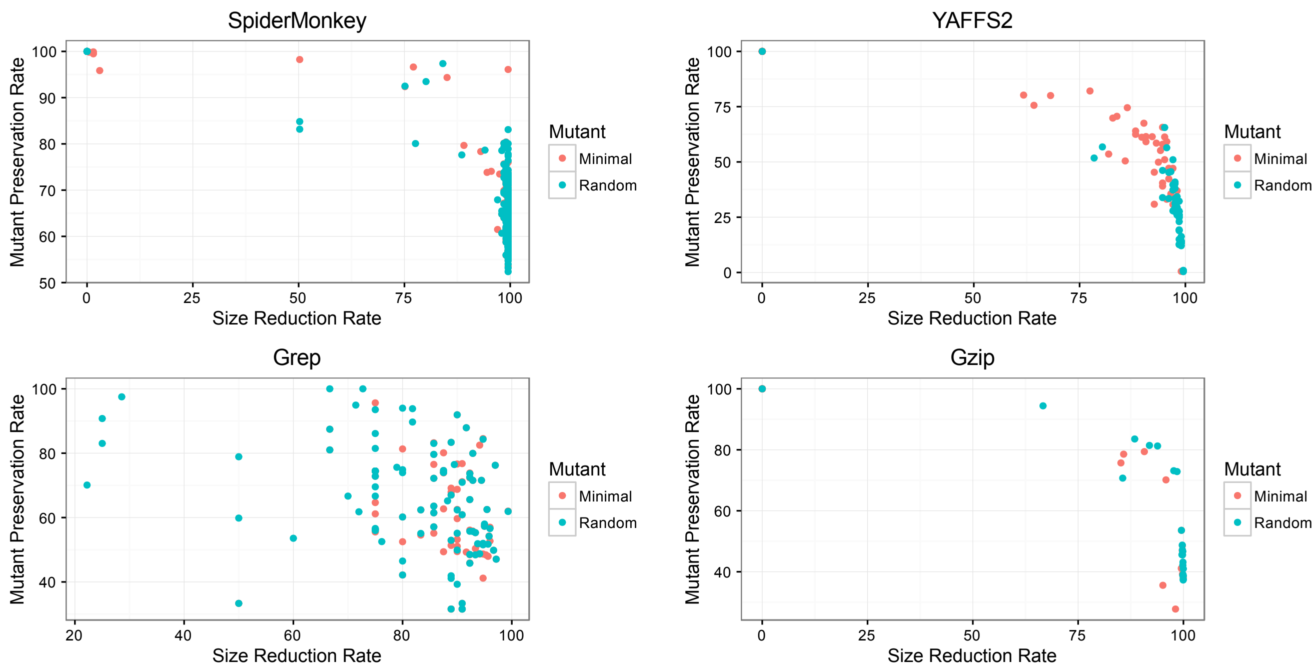

Figure 5: SRR vs. MPR, contrasting minimal mutants against randomly chosen mutants, for N-mutant test-case reduction

ues of C. Concerning general trends, we see that MPR is positively correlated to the value of C: more coverage preserved yields more mutants killed. With C of 95 or higher, the reduced test cases have the median MPR of at least 70% for all projects. Kendall-τ values for `SpiderMonkey`, `YAFFS2`, `Grep`, and `Gzip` were 0.89, 0.80, 0.67, and 0.76, respectively, all with $p < 0.001$, showing a strong positive correlation between the value of C and MPR.

Comparing across the projects, we see that `YAFFS2` has the lowest median MPR when reduced using N-mutant reduction (33.10%). `YAFFS2` test cases are sequences of function calls to the file-system API, such as `mount`, `open`, or `close`. There is little dependency across those functions (e.g., only a few functions call one another), so it is the `YAFFS2` test cases that effectively control the interaction among the functions by the ordering of the API calls. Thus, individual mutants can be isolated reasonably well from the other mutants, due to better decoupling between functions. On the other hand, modules in `SpiderMonkey`, like in any other interpreter or compiler, are deeply intertwined. Thus, each test case exercises multiple functions. As a result, killing a mutant in the parsing module, may also correlate with killing many other mutants in passes before or after parsing, such as lexing or interpretation. Therefore, it is expected that reduced test cases based on even a single mutant in `SpiderMonkey` could still kill a large portion of the mutants killed by the original test cases, with median MPR of 60.26%.

As N grows, MPR of the reduced test cases increases, unsurprisingly: more interdependent mutants can be killed. This observation is validated by the Kendall-τ values: 0.66, 0.69, 0.56, and 0.51 for `SpiderMonkey`, `YAFFS2`, `Grep`, and `Gzip`, respectively, all with $p < 0.001$, suggesting that there is a strong positive correlation between N and MPR. However, a trade-off is that as N increases, the time to perform the reduction increases as well, because intermediate test cases need to be checked against more mutants, and the chance of timeout increases.

In addition to performing N-mutant test-case reduction using N random mutants, we also used minimal mutants. Figure 5 shows for each project the relationship between SRR and MPR for test cases from the large, randomly generated test pool reduced using N-mutant with a randomly selected mutant or a minimal mutant. These plots only show the values for $N = 1$, because each test case can kill at most one minimal mutant. Surprisingly, there is no statistically significant difference ($p < 0.001$), except for `YAFFS2`. For `YAFFS2`, test cases reduced based on minimal mutant often result in a better trade-off between SRR and MPR: for the same SRR, test cases tend to have a higher MPR.

5.3 RQ3: Trade-Offs

Figure 8 shows the trade-off between SRR and CPR for `YAFFS2` test cases reduced using C%-coverage and N-mutant. We show plots only for `YAFFS2` due to space reasons; the plots for the other projects are similar. For C%-coverage, the CPR values cluster very closely with C values, but the SRR values vary, with higher SRR usually corresponding to lower CPR. For N-mutant, many test cases have high SRR, but CPR values vary widely.

Figure 9 shows the trade-off between SRR and MPR for `SpiderMonkey` test cases. Again, we show plots only for `SpiderMonkey`; the other projects are similar. For C%-coverage, we obtain good SRR and MPR without preserving all coverage: many points for $C = 90$ or $C = 95$ cluster in the upper-right of the plot. For N-mutant, more reduced test cases have high SRR, and larger N values have higher MPR.

Finally, Figure 10 visualizes the trade-off between CPR and MPR for all projects. For both plots, we see a linear correlation between CPR and MPR, especially for test cases reduced using C%-coverage. This trend suggests that the

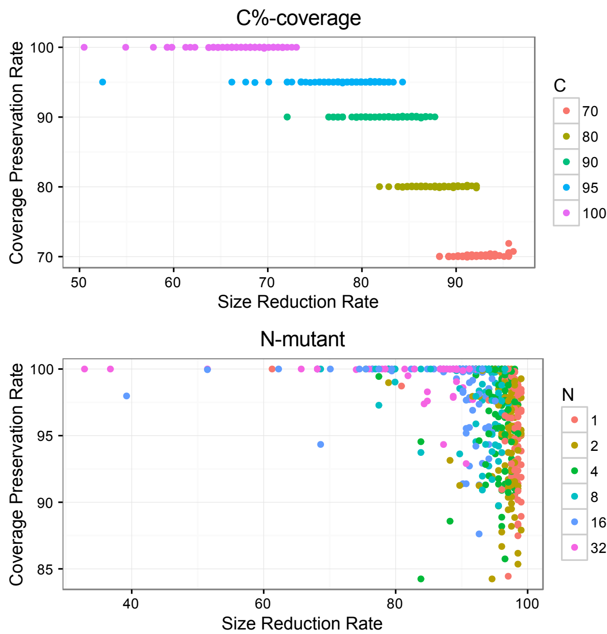

Figure 8: SRR vs. CPR for `YAFFS2`

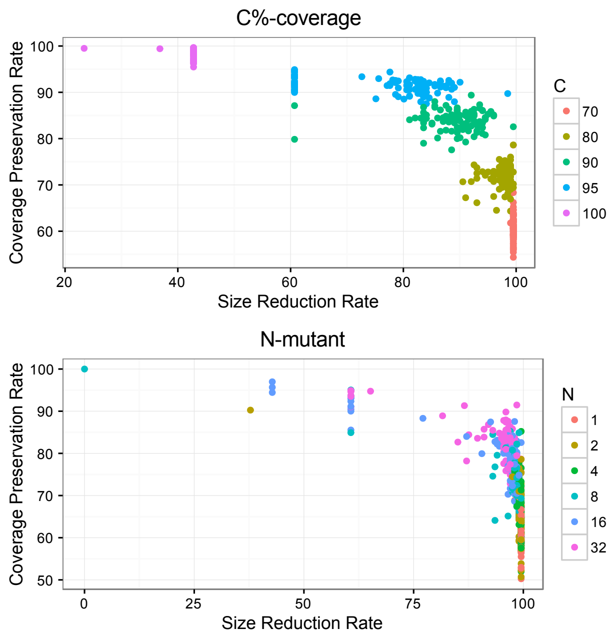

Figure 9: SRR vs. MPR for `SpiderMonkey`

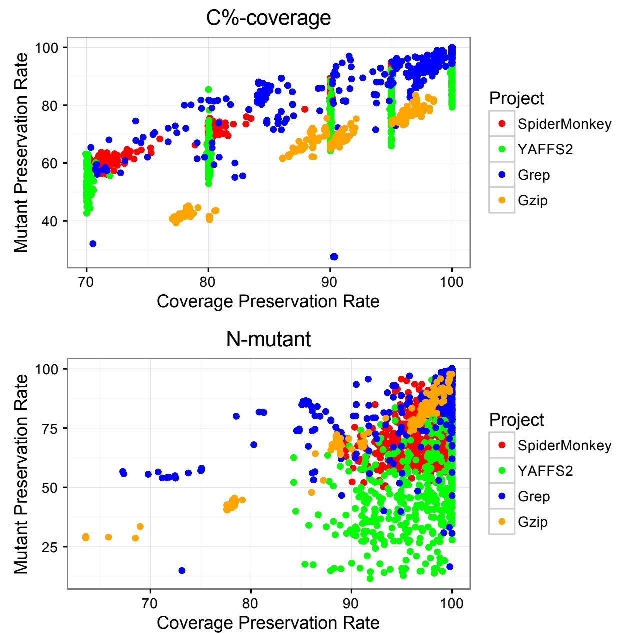

Figure 10: CPR vs. MPR for all four projects

more statements a test case covers the more mutants it kills. For N-mutant, especially for `YAFFS2`, there is more clustering towards the right side of the plot, indicating that even with a high CPR, MPR can still vary widely for the test cases reduced using N-mutant.

5.4 RQ4: Comparison with Random

We also compared our approaches to simple random test-case reduction that simply forces a certain size reduction on test case. For each test case reduced using non-adequate test-case reduction, we generate three reduced test cases of exactly the same size, by randomly removing parts from the original test case. SRR is exactly the same for a randomly reduced test case as for its corresponding test case. Therefore, we measure only CPR and MPR for these randomly reduced test cases.

Figure 11 shows boxplots that compare CPR for test cases reduced using $C\%$-coverage and N-mutant with test cases reduced randomly. We see from these figures that the median CPR computed for test cases reduced by non-adequate test-case reduction is greater than the median CPR computed for the test cases reduced randomly. Figure 12 shows the same comparison for MPR. Once again, we see from these plots that the median MPR computed for test cases reduced by non-adequate test-case reduction is greater than the median MPR computed for the test cases reduced randomly. The median CPR/MPR for test cases reduced using non-adequate test-case reduction is greater than the median CPR/MPR for the test cases reduced randomly. A values of randomly reduced test cases are significantly different ($p < 0.001$) from the test cases reduced using non-adequate test-case reduction.

This is hardly surprising, but confirms that our approaches add value. For `YAFFS2`, there is also a specific cause for the extreme differences due to the validity of the reduced test cases: if the original test case is valid, our non-adequate

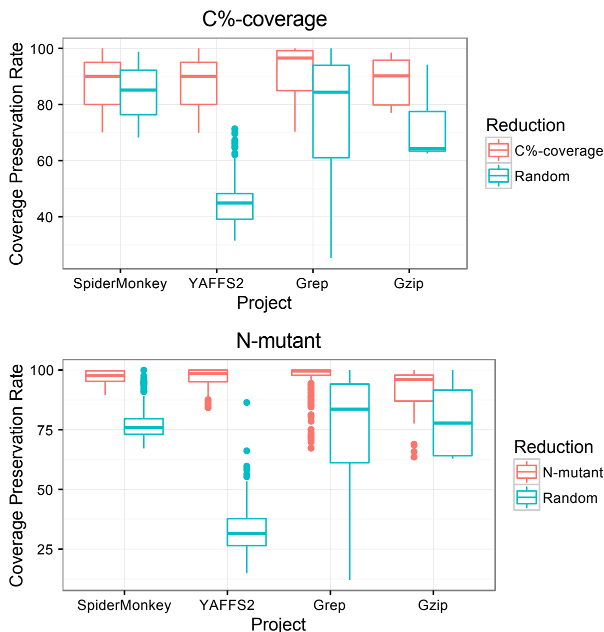

Figure 11: Comparing CPR of non-adequate test-case reduction with random test-case reduction

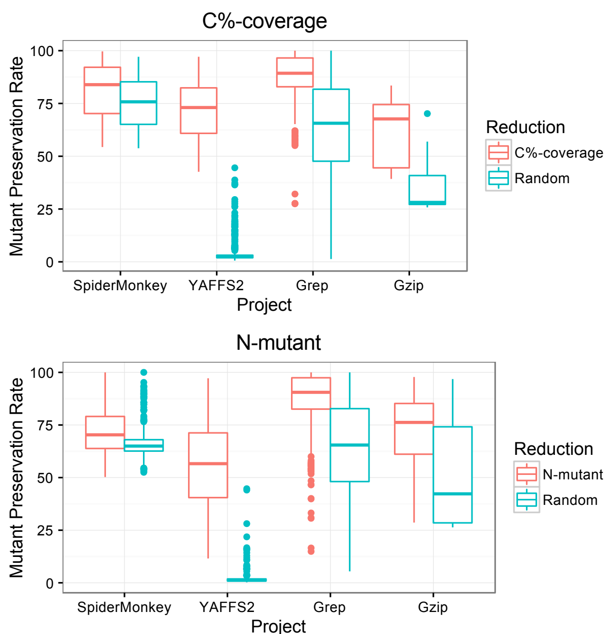

Figure 12: Comparing MPR of non-adequate test-case reduction with random test-case reduction

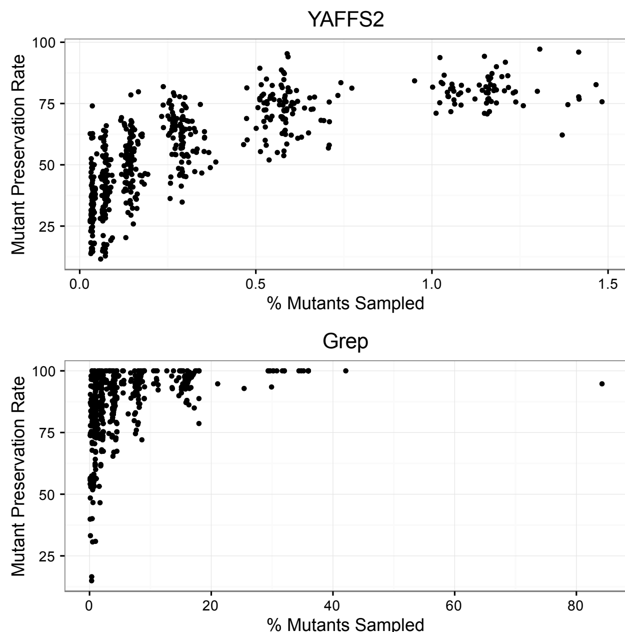

Figure 13: Percentage of mutants used for N-mutant test-case reduction vs. MPR

test-case reduction is unlikely to produce an invalid reduced test case. Each valid test case in `YAFFS2` starts by calling a startup function that prepares for mounting the file system. If a test case does not start with this function, the other function calls in the test case fail. The random test-case reduction is unaware of this, so if it has to reduce a sequence of 200 function calls to 4, each function call, including the startup function, has only $\frac{4}{200} = 2\%$ chance to be in the reduced test case, i.e., there is a high chance the reduced test case does not include the startup call and is invalid.

6. DISCUSSION

Inter-dependencies among mutants. From the MPR values for N-mutant reduction, we see that focusing the reduced test case to preserve only a small number of mutants killed by the original test case still kills a large fraction of all those mutants. For example, by reducing test cases based on only one mutant (i.e., $N = 1$), the median MPR values are 60.26%, 33.10%, 80.28%, and 43.92% for `SpiderMonkey`, `YAFFS2`, `Grep`, and `Gzip`, respectively. These high MPR values for such a small N suggest that many mutants killed by a test case have strong dependencies. Figure 13 illustrates this. The x-axis shows the ratio of N to the total number of mutants killed by the original test case, i.e., $N/Mut(t_o)$, and the y-axis shows the corresponding MPR. For space reasons, we show plots only for `YAFFS2` and `Grep`; the other two projects are similar to `YAFFS2`, but `Grep` is different from all others. We see that a test case reduced based on less than 0.5% of the mutants can still kill more than 50% of originally killed mutants. Note that when a test case reduced to kill some mutant M_1 also kills another mutant M_2, it does not imply that M_1 subsumes M_2 in the sense that *all* tests killing M_1 also kill M_2 [17].

Time for non-adequate test-case reduction. The time for reducing a test case depends on (1) the number of

Table 2: Time in seconds to perform test-case reduction

project	C%-coverage			N-mutant		
	Min	Med	Max	Min	Med	Max
SpiderMonkey	2	74	1003	1	9	1746
YAFFS2	12	102	794	1	24	1700
Grep	1	1	15	1	5	483
Gzip	4	430	1544	1	82	1799

parts in the test case, (2) the time to execute the test case, and (3) the cost of computing coverage or mutants killed. Table 2 summarizes the time required for test-case reduction in our experiments. For 50% of the test cases in SpiderMonkey, YAFFS2 and Grep, both C%-coverage and N-mutant non-adequate test-case reduction finish relatively fast (under two minutes for C%-coverage and under one minute for N-mutant). Gzip has significantly more parts (up to 3,500) than the other projects, which increases the time needed for reduction. In our experiment, all coverage-adequate reduction (i.e. $C = 100$) of Gzip test cases failed due to timeout, but all C%-coverage non-adequate test-case reduction finished within the time limit, with 50% of them being reduced in under ten minutes.

7. THREATS TO VALIDITY

Based on our results, it appears that non-adequate test-case reduction can substantially reduce the size of test cases while still preserving much of test case quality. As usual, experimental result may not generalize to other projects beyond the four we evaluated or even to other test cases and mutants than the ones we used for these projects. A particular threat is how we measure quality. We do not consider some interesting metrics at all (e.g., the execution time of reduced test cases), and the ones used are imperfect.

MPR considers all the mutants killed after performing N-mutant test-case reduction even though the reduction already uses some killed mutants as guidance to reduce the test case; one may argue that by construction the reduced test cases will be good by this metric, or, dually, that this metric is bad. However, we perform non-adequate test-case reduction that does not aim to preserve *all* mutants killed by the original test case, while MPR does consider all mutants killed. Therefore, we do not produce test cases that *necessarily* have a high MPR. Moreover, we also measure the CPR of these reduced test cases, and we do not use coverage to guide N-mutant test-case reduction.

Mutants of C code can introduce undefined behavior. For example, a mutant that removes initialization of a local variable can introduce such behavior. We did not explicitly remove such mutants, but we expect them to be relatively few. Therefore, we generate a large number of mutants for each project, reducing the chance that mutants that introduce undefined behavior significantly bias our results.

Another problem was that of the non-deterministic load on the shared high-performance cluster. Due to nodes having different configurations, and with different loads, a fixed timeout of 30 minutes may not correspond to the same amount of reduction. Hence, some mutants that happened to be evaluated on a slow machine may have been considered killed due to timeout while similarly slow-running mutants evaluated on a fast machine may have managed to complete successfully, thereby not considered killed.

8. RELATED WORK

Test-case reduction aims to reduce the size or complexity of test cases while preserving some desirable properties of these test cases. Reduction is essentially a search in the space of possible modifications to the original test case. In many uses, the only modification allowed is removing a part of the test case [23, 24, 27].

One goal of test-case reduction is to speed up testing, and this goal is shared with many techniques for regression testing, including regression test selection, test prioritization, and test-suite reduction [26]. The most similar to test-case reduction is test-suite reduction. Whereas test-case reduction aims to reduce a single test case, test-suite reduction aims to reduce the size of an entire test suite while preserving some desirable properties for the reduced test suite. Many studies investigated test-suite reduction techniques (e.g., [13,14,20]), including our recent work on non-adequate test-suite reduction [22]. However, this paper presents the first study of non-adequate test-case reduction. Test-case reduction and test-suite reduction can be easily combined [11], either in succession or in tandem.

Delta-debugging [27] is the best known technique for reducing the size of a failing test case: it reduces the test case so that it still fails but no single part can be removed without passing. Cause reduction [10, 11] generalizes delta-debugging by reducing a test case so that it still has the same coverage (or another property) but no single part can be removed without losing coverage (or another property). Our non-adequate test-case reduction further generalizes cause reduction by not requiring a test case to preserve the complete property the original test case satisfies.

9. CONCLUSION

Having smaller test cases is desirable for developers: such test cases run faster and make debugging easier. Test-case reduction reduces the size of test cases. Previous research has studied how to conduct test-case reduction while completely preserving some property of the original test case, e.g., failure or coverage. We evaluate a more general approach to test-case reduction, called non-adequate test-case reduction, that allows only partially preserving a property. Specifically, we propose and evaluate C%-coverage and N-mutant. Our results show that non-adequate test-case reduction can substantially reduce the size of test cases while still preserving a large percentage of all coverage or mutants killed by the original test cases. For C%-coverage in particular, simply giving up on a very small percentage of coverage can greatly reduce reduction time and produce a higher gain in size reduction than the associated loss in coverage. The idea of non-adequate test-case reduction greatly expands the options available in exploring trade-offs between test suite size (measured by adding sizes of individual test cases) for fault-detection capability.

10. ACKNOWLEDGEMENTS

We thank Nicholas Lu and Michael Hilton for comments on an earlier draft of this paper. This research was partially supported by the National Science Foundation Grant Nos. CCF-1054876, CCF-1409423, and CCF-1421503. Darko Marinov and August Shi also gratefully acknowledge the Google Faculty Research Award.

11. REFERENCES

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

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

[3] Andrews, J. H., Groce, A., Weston, M., and Xu, R.-G. Random test run length and effectiveness. In *ASE* (2008), pp. 19–28.

[4] Chen, Y., Groce, A., Zhang, C., Wong, W.-K., Fern, X., Eide, E., and Regehr, J. Taming compiler fuzzers. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 197–208.

[5] Claessen, K., and Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP* (2000), pp. 268–279.

[6] Do, H., Elbaum, S., and Rothermel, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE 10*, 4 (2005), 405–435.

[7] Gligoric, M., Eloussi, L., and Marinov, D. Practical regression test selection with dynamic file dependencies. In *ISSTA* (2015), pp. 211–222.

[8] Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M. A., and Marinov, D. Comparing non-adequate test suites using coverage criteria. In *ISSTA* (2013), pp. 302–313.

[9] Gopinath, R., Jensen, C., and Groce, A. Code coverage for suite evaluation by developers. In *ICSE* (2014), pp. 72–82.

[10] Groce, A., Alipour, M., Zhang, C., Chen, Y., and Regehr, J. Cause reduction for quick testing. In *ICST* (2014), pp. 243–252.

[11] Groce, A., Alipour, M. A., Zhang, C., Chen, Y., and Regehr, J. Cause reduction: Delta debugging, even without bugs. *STVR 26*, 1 (2015), 40–68.

[12] Groce, A., Holzmann, G., and Joshi, R. Randomized differential testing as a prelude to formal verification. In *ICSE* (2007), pp. 621–631.

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

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

[15] Inozemtseva, L., and Holmes, R. Coverage is not strongly correlated with test suite effectiveness. In *ICSE* (2014), pp. 435–445.

[16] Lei, Y., and Andrews, J. H. Minimization of randomized unit test cases. In *ISSRE* (2005), pp. 267–276.

[17] Papadakis, M., Henard, C., Harman, M., Jia, Y., and Le Traon, Y. Threats to the validity of mutation-based test assessment. In *ISSTA* (2016), pp. 354–365.

[18] Papadakis, M., Jia, Y., Harman, M., and Le Traon, Y. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE* (2015), pp. 936–946.

[19] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., and Yang, X. Test-case reduction for C compiler bugs. In *PLDI* (2012), pp. 335–346.

[20] Rothermel, G., Harrold, M. J., von Ronne, J., and Hong, C. Empirical studies of test-suite reduction. *STVR 12*, 4 (2002), 219–249.

[21] Ruderman, J. Introducing jsfunfuzz, 2007. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

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

[23] Slutz, D. R. Massive stochastic testing of SQL. In *VLDB* (1998), pp. 618–622.

[24] Whalley, D. B. Automatic isolation of compiler errors. *TOPLAS 16*, 5 (1994), 1648–1659.

[25] Yoo, S., and Harman, M. Pareto efficient multi-objective test case selection. In *ISSTA* (2007), pp. 140–150.

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

[27] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. *TSE 28*, 2 (2002), 183–200.