SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2015; **00**:1–30
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Cause reduction: delta debugging, even without bugs

Alex Groce,[1]* Mohammad Amin Alipour,[1] Chaoqiang Zhang,[1]
Yang Chen,[2] John Regehr[2]

[1]*Oregon State University, Corvallis, OR 97331, USA*
[2]*University of Utah, UT 84112, USA*

SUMMARY

What is a test case *for*? Sometimes, to expose a fault. Tests can also exercise code, use memory or time, or produce desired output. Given a desired effect a test case can be seen as a *cause*, and its components divided into essential (required for effect) and accidental. Delta debugging is used for removing accidents from failing test cases, producing smaller test cases that are easier to understand. This paper extends delta debugging by simplifying test cases with respect to *arbitrary effects*, a generalization called *cause reduction*. Suites produced by cause reduction provide effective *quick tests* for real-world programs. For Mozilla's JavaScript engine, the reduced suite is possibly *more* effective for finding faults. The effectiveness of reduction-based suites persists through changes to the software, improving coverage by over 500 branches for versions up to four months later. Cause reduction has other applications, including improving seeded symbolic execution, where using reduced tests can often double the number of additional branches explored.
Copyright © 2015 John Wiley & Sons, Ltd.

Received ...

1. INTRODUCTION

What is a test case *for*? This question motivates much of current software testing research. In the final analysis, the goal of testing is to improve software reliability and reduce risk of failure. As a means to this end, it is often asserted that the purpose of a test is to detect faults. However, for most Software Under Test (SUT), for most iterations of the testing process, most test cases *will not* expose a fault. Many test cases will *never* expose a fault. To say that these test cases are without value seems obviously incorrect. Testing, in both research and practice, therefore has also assumed that a test case is valuable because it *exercises certain behavior of the SUT*. The concept of code coverage [1], for example, equates a test case's value with the structural elements of the SUT it exercises. In stress testing, a test case is often valued for the memory, processor time, or bandwidth it requires. Understanding the precise details of each test case, or each test suite, is generally impossible. The essential effects (code coverage, load, use case, failure) of a test case are used in numerous ways:

- Test cases are often *generated*, either manually or automatically, to achieve one of these effects. E.g., unit tests are often produced to cover certain code paths, concolic testing [2] generates tests with complete branch coverage as a goal, and stress tests are obviously devised to produce stress.
- Test cases are often *selected* based on these purposes. Regression suites are often composed of tests that have previously failed. Work on test suite reduction and prioritization [3] typically chooses tests based on their structural coverage.

```
int f(int a, int b) {
  int x = (a * 40) + 2;
  int y = a + b;
  int z = (a * 40) + 161;
  return (a > 0 ? x : z);
}
```

Figure 1. A test case for common sub-expression elimination

- Test cases are often *evaluated* by these purposes. Quality assurance processes often use some form of structural coverage as a requirement for an "adequate" test suite [4], and of course testing research usually validates proposed techniques by fault detection or code coverage.

Based on this understanding, a test case can, for any given purpose, be understood as having both an *essential* effect (what makes the test case suitable for the purpose at hand—detecting some fault, covering some code, inducing some computational or network load, etc.) and an *accidental* aspect. The *accidents* of a test case are those components of the test that are not necessary to achieve the purpose of the test case. These accidental aspects of a test case are unfortunate in two respects. First, as noted, human understanding of test cases generally is limited; the task of debugging is arguably simply the effort of distinguishing the essence of a test case (or set of test cases) from the accidental aspects. Fault localization [5] can be thought of as automatically extracting a failure-based essence, in terms of source code locations. Second, time spent executing the accidental aspects of a test case is, in terms of a given essential property of a test case, wasted computational effort. This wasted effort is precisely why test case selection and prioritization often aims to avoid execution of tests that are redundant in terms of their essence (e.g., to compose minimal suites with complete coverage).

As an example, consider the C compiler test case in Figure 1. Assuming that the test case was written in order to test the compiler's common sub-expression elimination (CSE) optimization, it is clear that the test has both essential portions (required for the intended effect) and accidental elements. The variables and computation associated with `b` and `y` in particular do not seem to be required. The computation in the `return` statement, on the other hand, could be simplified, but if `x` and `z` are not potentially involved in the return value they may be eliminated as dead code by the compiler, avoiding CSE. While the runtime penalty of the accidental code is trivial, it is easy to imagine such elements as imposing a serious burden on testing, over longer tests or a very large number of tests. In this case, the additional code imposes a mental burden on someone attempting to understand the purpose of the test.

1.1. Delta Debugging: Reducing a Test Case for Human Understanding

Delta debugging (or delta-minimization) is a greedy algorithm (called *ddmin*) for reducing the size of failing test cases by eliminating accidental components. Delta debugging algorithms have retained a common core since the original proposal of Hildebrandt and Zeller [6]: use a variation of binary search to remove individual components of a failing test case t to produce a *new* test case t_{1min} satisfying two properties: (1) t_{1min} fails and (2) removing any component from t_{1min} results in a test case that does not fail.[*] Such a test case is called *1-minimal*. Because 1-minimal test cases can potentially be much larger than the smallest possible set of failing components, *ddmin* technically *reduces* the size of a test case, rather than truly minimizing it. While the precise details of *ddmin* and its variants are complex, delta debugging algorithms can be simply described. Ignoring caching and the details of an effective divide-and-conquer strategy for constructing candidate test

[*]In their original formulation, Hildebrandt and Zeller refer to "circumstances" rather than components, to emphasize the generality of the idea, but in all of the uses in this paper, non-overlapping components of a test case are the circumstances in question.

cases, *ddmin* for a base failing test case t_b proceeds by iterating the following two steps until a fixpoint is reached:

1. Construct the next candidate simplification of t_b, denoted by t_c. Terminate if no t_c remain (t_b is 1-minimal).
2. Execute t_c by calling $rtest(t_c)$. If $rtest$ returns ✗ (the test fails) then it is a simplification of t_b. Set $t_b = t_c$.

The purpose of this iteration is to reduce a test case until it has as few accidental components as possible, with respect to its essential nature as a *failing* test case. Note that the algorithm ignores any properties of a candidate test case except requiring 1) that the size, as measured by some notion of test components, must be smaller than the original test case and 2) the test case must still fail (in most practical cases, this is additionally restricted to "fail in the same way as the original test case"). These two conditions may be thought of as the *cause* condition and the *effect* condition. All other aspects of the candidate test cases are ignored: for instance, the algorithm allows the effects of a reduced test case to be more complex or numerous than for the original test case, so long as the desired effect is preserved. In practice, it appears reduction seldom introduces more effect-side behavior, however.

1.2. Cause Reduction: Delta Debugging with a Parameterized Effect

Delta debugging is an extremely useful algorithm, and good delta debugging tools are practically required in order to do meaningful random testing on complex systems [7, 8]. Unfortunately, traditional delta debugging reduces tests *too much* for any goal not related to simple success or failure, discarding all behavior not related to the failure of a test. Even more critically, traditional delta debugging cannot be applied to successful tests at all. The assumption has always been that delta debugging is a *debugging* algorithm, only useful for reducing failures. However, the best way to understand *ddmin*-like algorithms is that they reduce the size of a generic *cause* (e.g. a test case, a thread schedule, etc.) while ensuring that it still causes some fixed *effect*[†] (in *ddmin*, the effect is always test failure) related to the essence of a test case: *ddmin* is a special-case of *cause reduction*.[‡] The core value of delta debugging can be understood in a simple proposition: given two test cases that achieve the same purpose, the smaller of the two test cases will typically execute more quickly and be easier to understand and mechanically analyze. Delta debugging is valuable because, *given two test cases that both serve the same purpose*, the smaller of the two is almost always preferable. There is no reason why the essence of a test case should be limited to fault detection.

The definitions provided in the core delta debugging paper [9] are *almost* unchanged in cause reduction. The one necessary alteration is to replace the function *rtest*, which "takes a program run and tests whether it produces the failure" in Definition 3 (in the DD journal paper [9]) with a function *reffect* such that *reffect* simply tests whether the test case satisfies an arbitrary predicate representing some interesting effect. Failure is just one instance of an effect to be preserved.

This "new" algorithm is called *cause reduction* but, of course, it is almost exactly the same as the original *ddmin* algorithm, and most optimizations, variations, and restrictions still apply. For example, in traditional delta debugging, it is critical to define candidate test cases for reduction such that the validity of the original test case is preserved. In some settings, this is trivial: e.g., many random testing systems produce test cases where each test component is independent with respect to validity of the test case. In other cases, such as detecting incorrect compilation (vs. compiler crashes) of C code [8], preserving test case validity requires complex checks, and is a time-consuming aspect of setting up a delta debugging system. In this paper, it is assumed that the conventional engineering methods for preserving test case validity have been applied. In all experiments, even for human-produced test cases, line or character based reduction always produces valid test cases.

[†]In practice, again, note that the test may cause some additional effect(s) as well, but it is guaranteed by the algorithm to maintain the desired effect.
[‡]The authors would like to thank Andreas Zeller for suggesting the term "cause reduction."

```
int f(int a) {
  int x = (a * 4) + 2;
  int z = (a * 4) + 1;
  return (a ? x : z);
}
```

Figure 2. A reduced test case for common sub-expression elimination

The most interesting consequence of the "minor change" introduced by the concept of cause reduction is that *ddmin* is no longer defined only for failing test cases. If the effect chosen is well-defined for all test cases, then *ddmin* can be applied to reduce the cause (test case) for any test case. As an example, imagine taking off-the-shelf character-level delta debugging tools, plus some static analysis to guarantee that a C program remains in the defined subset of the language [8] and applying these tools to the code in Figure 1.[§] Assuming this test case succeeds, traditional delta debugging cannot reduce it. However, it is easy to instrument the compiler to log whether it successfully applied common sub-expression elimination (or a value numbering optimization performing the same function), and consider a run to be "failing" if CSE succeeds. With this criteria, the *ddmin* algorithm reduces the program to the smaller version shown in Figure 2 (reformatted to insert removed line breaks).

The idea of cause reduction is, in some sense, quite trivial. The core question is whether cause reduction is simply a novel but pointless generalization of a useful algorithm, or whether the concept of cause reduction enables practically important advances in software testing. This paper introduces at length one such advance, the use of cause reduction with respect to *code coverage* to produce very small, highly efficient, regression suites ("quick tests") for use in large-scale random testing. This paper shows that reduction that preserves *statement coverage* can approximate retaining other important effects, including *fault detection and branch coverage*. A large case study based on testing Mozilla's SpiderMonkey JavaScript engine uses real faults to show that cause reduction is effective for improving test efficiency, and that the effectiveness of reduced test cases persists even across a long period of development, without re-running the reduction algorithm. Even more surprisingly, for the version of SpiderMonkey used to perform cause reduction and a version of the code from more than two months later, the reduced suite not only runs almost four times faster than the original suite, but detects *more* distinct faults. A mutation-based analysis of the YAFFS2 flash file system shows that the effectiveness of cause reduction is not unique to SpiderMonkey. Moreover, cause reduction can be used to aid complex program analyses that are based on test cases. Reducing by code coverage, even when it does not significantly improve the runtime of a test case, can make the test case a more suitable basis for symbolic execution. In many cases, spending 20% of a symbolic execution budget on reducing test cases, rather than on symbolic exploration, can result in covering more than twice as many additional branches as symbolic execution alone [10].

This paper is an extension of the ICST 2014 [11] paper proposing the idea of cause reduction for quick testing, providing more, and more detailed, experimental results as well as new applications of cause reduction. Section 2 presents the problem of quick testing and introduces a detailed study of how cause reduction with respect to code coverage can address this problem. Section 3 shows how cause reduction can aid seeded symbolic execution. Sections 4 and 5 introduce two more narrow applications of cause reduction that use different cause reduction criteria, in order to suggest to the reader the flexibility of the idea. Finally, in Section 6 the paper's results are placed in the context of other work on delta debugging, test suite optimization, and causality. Section 7 presents some final thoughts on cause reduction and summarizes key results.

[§]In practice, character level delta debugging is not recommended for C programs, but in this case it is not problematic; moreover, for this toy example, no code is produced that introduces undefined behavior and still compiles, making the static analysis step unnecessary.

2. COVERAGE-BASED CAUSE REDUCTION FOR QUICK TESTING

2.1. The Quick Test Problem

In testing a flash file system implementation that eventually evolved into the file system for the Mars Science Laboratory (MSL) project's Curiosity rover [12, 13], one of the authors of this paper discovered that, while an overnight sequence of random tests was effective for shaking out even subtle faults, random testing was not very effective if only a short time (less than an hour) was available for testing. Each individual random test was a highly redundant, ineffective use of testing budget. As a basic sanity check/smoke test before checking a new version of the file system in, it was much more effective to run a regression suite built by applying delta debugging [9] to a representative test case for each fault previously found.

In addition to detecting actual regressions of the NASA code, *ddmin*-minimized test cases obtained close to 85% statement coverage in less than a minute, which running new random tests often required hours to match. Unfortunately, the delta debugging-based regression was often *ineffective* for detecting *new* faults unrelated to previous bugs. Inspecting minimized test cases revealed that, while the tests covered most statements, the tests were extremely focused on corner cases that had triggered failures, and sometimes missed very shallow bugs easily detected by a short amount of random testing. While the bug-based suite was effective as a pure *regression* suite, it was ineffective as a quick way to find *new* bugs; on the other hand, running new random tests was sometimes very slow for detecting either regressions or new bugs.

The functional programming community has long recognized the value of very quick, if not extremely thorough, random testing during development, as shown by the wide use of the QuickCheck tool [14]. QuickCheck is most useful, however, for data structures and small modules, and works best in combination with a functional style allowing modular checks of referentially transparent functions. Even using feedback [12, 15], swarm testing [16], or other improvements to standard random testing, it is extremely hard to randomly generate effective tests for complex systems software such as compilers [17] and file systems [12, 13] without a large test budget. For example, even tuned random testers show increasing fault detection with larger tests, which limits the number of tests that can be run in a small budget [17, 18]. The value of the *ddmin* regressions at NASA, however, suggests a more tractable problem: *given* a set of random tests, generate a truly *quick test* for complex systems software. Rather than choosing a particular test budget that represents "the" quick test problem, this paper proposes that quick testing is testing with a budget that is at most half as large as a full test budget, and typically more than an order of magnitude smaller. Discussion with developers and the authors' experience suggested two concrete values to use in evaluating quick test methods. First, tests that take only 30 seconds to run can be considered almost without cost, and executed after, e.g., every compilation. Second, a five minute budget is too large to invoke with such frequency, but maps well to short breaks from coding (e.g. the time it takes to get coffee), and is suitable to use before relatively frequent code check-ins. The idea of a quick test is inherent in the concept of test *efficiency*, defined as coverage/fault detection *per unit time* [19, 20], as opposed to absolute effectiveness, where large suites always tend to win.

The *quick test* problem is therefore: given a set of randomly generated tests, produce test suites for test budgets that are sufficiently small that they allow tests to be run frequently during code development, and that maximize:

1. Code coverage: the most important coverage criterion is probably statement coverage; branch and function coverage are also clearly desirable;[¶]
2. Failures: automatic fault localization techniques [5] often work best in the presence of multiple failing test cases; more failures also indicate a higher probability of finding a flaw;

[¶]The importance of various coverages is based on recent work on the effectiveness of coverage criteria for test suite evaluation [21, 22].

3. Distinct faults detected: finally, the most important evaluation metric is the actual number of *distinct* faults that a suite detects; it is generally better to produce four failures, each of which exhibits a distinct fault, than to produce 50 failures that exhibit only two different faults [23].

It is acceptable for a quick test approach to require significant pre-computation and analysis of the testing already performed if the generated suites remain effective across significant changes to the tested code without re-computation. Performing 10 minutes of analysis before *every* "30 second run" is unacceptable (in that they are no longer 30 second runs); performing 10 hours of analysis *once* to produce quick test suites that remain useful for a period of months is fine. For quick test purposes, it is also probably more feasible to build a generally good small suite rather than perform change analysis on-the-fly to select test cases that need to be executed [3, 24]. In addition compilers and interpreters tend to pose a difficult problem for change analysis, since optimization passes rely on deep semantic properties of the test case.

Given the highly parallel nature of random testing, in principle arbitrarily many tests could be performed in five minutes. In practice, considerable effort is required to introduce and maintain cloud or cluster-based testing, and developers often work offline or can only use local resources due to security or confidentiality concerns. More critically, a truly small quick test would enable testing on slow, access-limited hardware systems; in MSL development, random tests were not performed on flight hardware due to high demand for access to the limited number of such systems [25], and the slowness of radiation-hardened processors. A test suite that only requires 30 seconds to five minutes of time on a workstation, however, would be feasible for use on flight testbeds. The desire for high quality random tests for slow/limited access hardware may extend to other embedded systems contexts, including embedded compiler development. Such cases are more common than may be obvious: for instance, Android GUI random testing [26] on actual mobile devices can be even slower than on already slow emulators, but is critical for finding device-dependent problems. Quick testing's model of expensive pre-computation to obtain highly efficient execution is a good fit for the challenge of testing on slow and/or over-subscribed hardware.

In some cases, the quick test problem might be solved simply by using test-generation techniques that produce short tests in the first place, e.g. evolutionary/genetic testing approaches where test size is included in fitness [27, 28, 29], or bounded exhaustive testing (BET). BET, unfortunately, performs poorly even for file system testing [30] and is very hard to apply to compiler testing. Recent evolutionary approaches [28] are more likely to succeed, but (to the authors' knowledge) have not been applied to such complex problems as compiler or interpreter testing, where hand-tuned systems requiring expert knowledge are typical [17, 31]. Even random testers for file systems are often complex, hand-tuned systems with custom feedback and hardware fault models [12].

One approach to quick testing is to use traditional test suite prioritization or minimization techniques [3]. However, these techniques provide no mitigation for the extreme redundancy and inefficiency of most random tests. If a typical generated random test requires more than a second to execute, a 30 second quick test will never execute more than 30 tests; 30 random tests is, for most systems, not a very effective test, even if those tests are well chosen. Cause reduction offers the potential to actually increase the number of tests run with the same computational effort. However, to apply cause reduction, the user needs to find an effect that maintains the desirable properties of the original test case: fault detection and, more generally, exploration of SUT behavior. Does such an effect exist?

2.2. *Coverage as an Effect*

In fact, a large portion of the literature on software testing is devoted to proposing precisely such a class of effects: running a test case always produces the effect of *covering* certain source code elements, which can include statements, branches, data-flow relationships, state predicates, or paths. High coverage is a common goal of testing, as high coverage correlates with effective fault detection [32]. Producing *small* test suites with high code coverage [20] has long been a major goal of software testing efforts, inspiring a lengthy literature on how to minimize a test suite with respect to coverage, how to select tests from a suite based on coverage, and how to prioritize a test suite by coverage [3]. Coverage-based minimization reduces a *suite* by removing test cases; using cause reduction, a suite can also (orthogonally) be reduced by minimizing each test in the suite (retaining all tests) with the effect being *any chosen coverage criteria*. The potential benefit of reduction at the test level is the same as at the suite level: more efficient testing, in terms of fault detection or code coverage per unit of time spent executing tests. Cause reduction with respect to coverage is a promising approach for building quick tests, as random tests are likely to be highly reducible.

As described in the introduction, *ddmin* algorithms proceed by generating "candidate" tests: tests that are smaller than the original test case, but may preserve the property of interest, which in the original algorithm is "this test fails." When evaluating the preservation check on a candidate reduced test case returns ✗ (indicating the test failed) *ddmin* essentially starts over, with the candidate test case as the new starting point for reduction, until no candidates fail. Preservation is formulated as follows for coverage-based reduction:

$$\mathit{reffect}(t_c, t_b) = \begin{cases} \text{iff } \forall s \in c(t_b).s \in c(t_c) & \text{✗} \\ \text{else} & \checkmark \end{cases}$$

where t_c is the currently proposed smaller test, t_b is the original test case, and $c(t)$ is the set of all coverage entities executed by t. While it may be confusing that a valid reduction of the test case returns ✗ the terminology of the original papers is maintained to show how little difference there is between generalized cause reduction and the *ddmin* algorithm; recall that in *ddmin* the point of preservation is to find tests that fail. Returning ✗ in this context means that the new test has preserved coverage and can therefore be used as the basis for further reduction efforts, while ✓ means that the candidate test does *not* preserve coverage, and should be discarded (the fate of any successful test case in the original *ddmin* algorithm). Note that this definition allows a test case to be minimized to a test with *better* coverage than the original test. In practice, improved coverage seems rare: if a smaller test that does not preserve the added coverage can be found, *ddmin* removes gained coverage. As a variant, it would be possible to force preservation of any gains in coverage encountered during reduction; the authors did consider this approach, but found that such gains were rare enough that it made little difference. *Forcing* reductions to *improve* coverage resulted in most test cases not reducing at all, at high computational cost, and very slight gains in coverage at high cost for other test cases (with little reduction). However, the authors did not experiment extensively with these variants.

In principle, *any* coverage criteria could be used as an effect. In practice, it is highly unlikely that reducing by extremely fine-grained coverages such as path or predicate coverages [32] would produce significant reduction. Moreover, *ddmin* is a very expensive algorithm to run when test cases do not reduce well, since every small reduction produces a new attempt to establish 1-minimality: small removals tend to result in a very large computational effort proportional to the reduction. Additionally, for purposes of a quick test, it seems most important to concentrate on coverage of coarse entities, such as statements. Finally, only branch and statement (which obviously includes function) coverage are widely enough implemented for languages that it is safe to assume anyone interested in producing a quick test has tools to support their use. For random testing, which is often carried out by developers or by security experts, this last condition is important: lightweight methods that do not require static or dynamic analysis expertise and are easy to implement from scratch are more likely to be widely applied [33]. This paper therefore investigates only statement, branch, and function coverage.

2.3. Combining Test-Based and Suite-Based Reductions

While the primary interest is in reduction at the test case, rather than test suite, level, it is also interesting to consider the possibility of combining the two approaches to suite reduction. Figure 3 shows an algorithm for combining the approaches, where `T` is the original test suite and `M` is the output selected and reduced suite. First, some test suite selection method is applied to produce a suite $B \subseteq T$ such that B has the same total coverage as T. Any of the various algorithms [3] proposed for this task can be used. The tests in B are then, in the order of the coverage they contribute, minimized with the requirement that they *cover all statements that are their additional contribution*. That is,

```
B = Select(T)   compute test cases for complete coverage
M = $\emptyset$           output: set of reduced test cases
C = $\emptyset$           entities covered so far
while (B $\neq$ $\emptyset$)
      $t$ = $t$ ∈ B such that $c(t)$−C is maximized
      B = B − $t$
      $t'$ = ddmin($t$, $c(t')$ $\supseteq$ $c(t)$−C)
      M = M + $t'$
      C = C + $c(t')$
return M
```

Figure 3. Algorithm for combining suite-based selection and cause reduction

rather than requiring each test to preserve its original statement coverage, each test is only required to preserve coverage such that the entire selected suite will still have the same overall coverage.

2.4. SpiderMonkey JavaScript Engine Case Study

In order to determine the value of cause reduction for quick testing, this paper relies on one large case study and two smaller studies. The core research questions to be answered are:

- **RQ1:** How does minimizing an entire suite using cause reduction with respect to coverage affect other properties of the suite (runtime, different coverages, failures, fault detection)?
- **RQ2:** How does this impact the effectiveness (by the same evaluation measures) of quick tests based on these suites, here represented by 30 second and 5 minute test budgets?
- **RQ3:** How do the results for RQ1 and RQ2 change as the software is modified over time, but the minimized suite is not re-computed?
- **RQ4:** What is the computational cost of cause reduction?

In addition to these core questions, this paper also considers the effects of combining cause reduction with traditional prioritization and selection methods. To answer the questions, the only approach is to construct a large number of different test suites and compare them: given a baseline test suite produced by random testing, apply cause reduction by coverage measures and some traditional suite prioritization algorithms, and measure key suite properties with varying testing budgets and for different versions of the software under test. Briefly put, the independent variables are program version, suite construction method, and testing budget, and the dependent variables are the various evaluation measures listed below. Answers to research questions 1, 2, and 4 are broken out into separate sections below; results for RQ3 are spread through other sections.

2.4.1. Object of Study The large case study is based on SpiderMonkey, the JavaScript Engine for Mozilla, an extremely widely used, security-critical interpreter/JIT compiler. SpiderMonkey has been the target of aggressive random testing since 2007. A single fuzzing tool, `jsfunfuzz` [31], is responsible for identifying more than 1,700 previously unknown bugs in SpiderMonkey [34]. SpiderMonkey is (and was) very actively developed, with over 6,000 code commits in the period from January 2006 to September 2011 (nearly four commits/day). SpiderMonkey is thus ideal for evaluating the effects of coverage-based reduction, using the last public release of the `jsfunfuzz` tool, modified for swarm testing [16]. Figures 4 and 5 show cause reduction by statement coverage in action. A `jsfunfuzz` test case consists of a sequence of JavaScript constructs, given as arguments to a JavaScript function, `tryItOut`. The `tryItOut` function compiles its string argument and executes it in a sandbox, ensuring that unbounded loops or uncaught exceptions don't terminate the test, and checking some deeper semantic properties of execution (e.g., calling `uneval` if an object is returned by the code fragment). The first figure is a short test generated by `jsfunfuzz`; the second is a test case based on it, produced by *ddmin* using statement coverage as effect. These

```
tryItOut("with((delete __proto__))
          {export __parent__;true;}");
tryItOut("while((false for (constructor in false))){}");
tryItOut("throw __noSuchMethod__;");
tryItOut("throw undefined;");
tryItOut("if(<><x><y/></x></>) {null;}else{/x/;/x/g;}");
tryItOut("{yield;export __count__; }");
tryItOut("throw StopIteration;");
tryItOut("throw StopIteration;");
tryItOut(";yield;");
```

Figure 4. `jsfunfuzz` test case before statement coverage reduction

```
tryItOut("with((delete __proto__))
          {export __parent__;true;}");
tryItOut("while((false for (constructor in false))){}");
tryItOut("throw undefined;");
tryItOut("if(<><x><y/></x></>) {null;}else{/x/;/x/g;}");
tryItOut("throw StopIteration;");
tryItOut(";yield;");
```

Figure 5. `jsfunfuzz` test case after statement coverage reduction

tests both cover the same 9,625 lines of code. While some reductions are easily predictable (e.g. the repeated `throw StopIteration`), others are highly non-obvious, even to a developer.

2.4.2. Evaluation Measures The evaluation measures for suites are: size (in # tests), statement coverage (ST), branch coverage (BR), function coverage (FN), number of failing tests, and estimated number of distinct faults. All coverage measures were determined by running gcov (which was also used to compute coverage for *reffect*). Failures were detected by the various oracles in `jsfunfuzz` and, of course, detecting crashes and timeouts.

Distinct faults detected by each suite were estimated using a binary search over all source code commits made to the SpiderMonkey code repository, identifying, for each test case, a commit such that: (1) the test fails before the commit and (2) the test succeeds after the commit. With the provision that the authors have not performed extensive hand-confirmation of the results, this is similar to the procedure used to identify bugs in previous work investigating the problem of ranking test cases such that tests failing due to different underlying faults appear early in the ranking [23]. This method is not always precise. It is, however, uniform and has no obvious problematic biases. Its greatest weakness is that if two bugs are fixed in the same check-in, they will be considered to be "one fault"; the estimates of distinct faults are therefore best viewed as *lower* bounds on actual distinct faults. In practice, hand examination of tests in previous work suggested that the results of this method are fairly good approximations of the real number of distinct faults (as defined by fixes applied by developers) detected by a suite. Some bugs reported may be faults that developers knew about but gave low priority; however, more than 80 failures result in memory corruption, indicating a potential security flaw, and all faults identified were fixed at some point.

2.4.3. Test Suites Constructed for Evaluation The baseline test suite for SpiderMonkey is a set of 13,323 random tests, produced during four hours of testing the 1.6 source release of SpiderMonkey. These tests constitute what is referred to below as the **Full** test suite. Running the **Full** suite is essentially equivalent to generating new random tests of SpiderMonkey. A reduced suite with equivalent statement coverage, referred to as **Min**, was produced by performing cause reduction with respect to statement coverage on every test in **Full**. The granularity of minimization was based

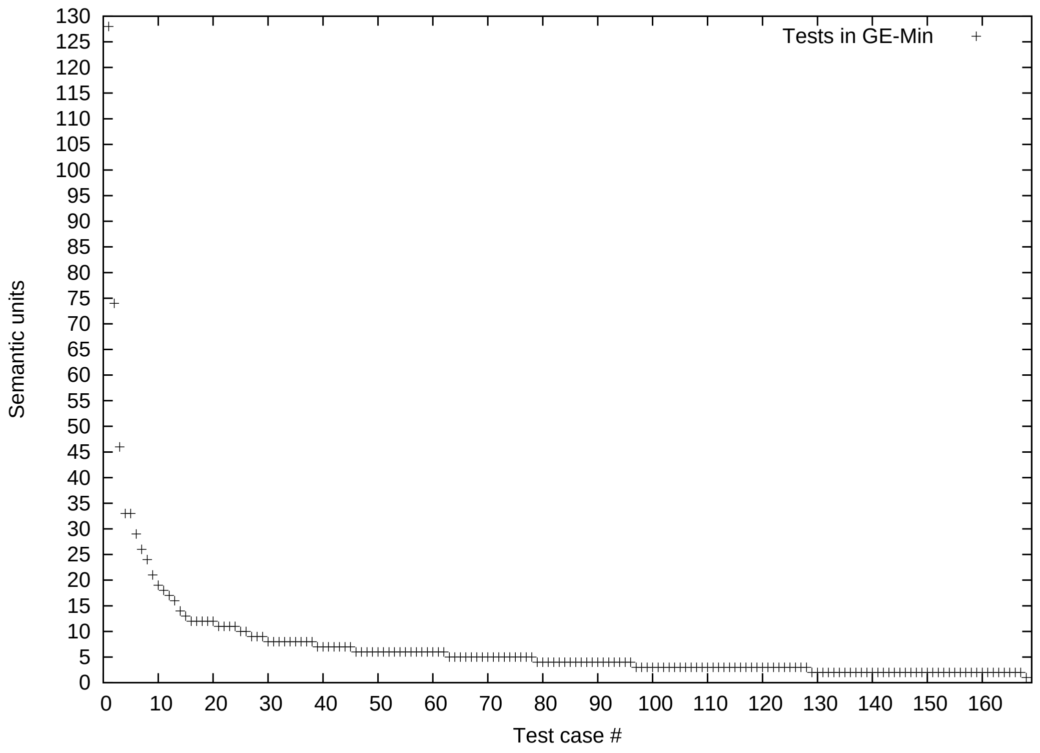

Figure 6. Semantic units in **GE-Min** test cases

on the semantic units produced by `jsfunfuzz`, with 1,000 such units in each test in **Full**. A unit is the code inside each `tryItOut` call—approximately one line of code. After reduction, the average test case size was just over 123 semantic units, a bit less than an order of magnitude reduction; while increases in coverage were allowed, in 99% of cases coverage was identical to the original test. The computational cost of cause reduction was, on contemporary hardware, around 30–35 minutes per test case, somewhat higher than the costs of traditional delta debugging reported in older papers [7]. The entire process completed in less than four hours on a modestly sized heterogeneous cluster (using fewer than 120 nodes). The initial plan to also minimize by branch coverage was abandoned when it became clear that statement-based minimization tended to almost perfectly preserve total suite branch coverage. Branch-based minimization was also much slower (by a factor of two or more) and typically reduced size by a factor of only 2/3, vs. nearly 10x reduction for statements.

A third suite, $\mathbf{Min_{FN}}$ was produced by the same method, only applying cause reduction with respect to function, rather than statement, coverage. The cost of reduction was improved to less than 10 minutes on average, and the average length in semantic units was lowered to only 27.6 units.

A fourth suite, $\mathbf{Min_{Fail}}$, was produced by taking all 1,631 failing test cases in **Full** and reducing them using *ddmin* with the requirement that the test case fail and produce the same failure output as the original test case. After removing numerous duplicate tests, $\mathbf{Min_{Fail}}$ consisted of 1,019 test cases, with an average size of only 1.86 semantic units (the largest test contained only 9 units). Reduction in this case only required about five minutes per test case.

Two additional small suites, $\mathbf{GE}_{ST}$**(Full)** and $\mathbf{GE}_{ST}$**(Min)** were produced by applying Chen and Lau's GE heuristic [35] for coverage-based suite minimization to the **Full** and **Min** suites. The GE heuristic first selects all test cases that are essential (i.e., they uniquely cover some coverage entity), then repeatedly selects the test case that covers the most additional entities, until the coverage of the minimized suite is equal to the coverage of the full suite (i.e., an additional greedy algorithm, seeded with test cases that must be in any solution). Ties are broken randomly.

GE_{ST} and cause reduction with respect to statement coverage were combined to produce the **GE-Min** suite, by applying the algorithm in Figure 3. This resulted in a test suite with the same coverage as the entire original test suite, but a runtime of less than 30 seconds—a full coverage 30 second quick test. The total number of semantic units for all tests in **GE-Min** is only 1,212, slightly larger than one original unreduced test case. Figure 6 shows the size in semantic units of tests in **GE-Min**. After the first few test cases, reduction is soon targeting only a very small number of statements per test case, resulting in very small and highly specialized test cases.

In order to produce 30 second and five minute test suites (the extremes of the likely quick test budget), it was necessary to choose subsets of **Full** and **Min**. The baseline approach is to randomly sample a suite, an approach to test case prioritization used as a baseline in numerous previous test case prioritization and selection papers [3]. While a large number of plausible prioritization strategies exist, this study is restricted to ones that do not require analysis of faults, expensive mutation testing, deep static analysis, or in fact any tools other than standard code coverage. As discussed above, a primary goal here is to prefer methods as lightweight and generally applicable as possible. On this basis, the strategies are limited to (simple) four coverage-based prioritizations from the literature [3, 36], which are denoted by ΔST, |ST|, ΔBR, and |BR|. ΔST indicates a suite ordered by the incremental improvement (Δ) in statement coverage offered by each test over all previous tests (an additional greedy algorithm), while |ST| indicates a suite ordered by the absolute statement coverage of each test case (a pure greedy algorithm). The first test executed for both ΔST and |ST| will be the one with the highest total coverage. ΔBR and |BR| are similar, except ordered by different coverage.

Finally, a key question for a quick test method is how long quick tests remain effective. As code changes, a cause reduction and prioritization based on tests from an earlier version of the code will (it seems likely) become obsolete. Bug fixes and new features (especially optimizations in a compiler) will cause the same test case to change its coverage, and over time the basic structure of the code may change; SpiderMonkey itself offers a particularly striking case of code change: between release version 1.6 and release version 1.8.5, the vast majority of the C code-base was re-written in C++. All experiments were therefore performed not only on SpiderMonkey 1.6, the baseline for cause reduction, but applied to "future" (from the point of view of quick test generation) versions of the code. The first two versions are internal source commits, not release versions, dating from approximately two months (2/24/2007) and approximately four months (4/24/2007) after the SpiderMonkey 1.6 release (12/22/2006). When these versions showed that quick tests retained considerable power, it indicated that a longer lifetime than expected might be possible. The final two versions of SpiderMonkey chosen were therefore the 1.7 release version (10/19/2007) and the 1.8.5 release version (3/31/2011). Note that all suites were reduced and prioritized based on the 1.6 release code; no re-reduction or re-prioritization was ever applied. Strictly speaking, therefore, this involved no creation of new suites other than the random selection of 30s and 5m samples.

2.4.4. SpiderMonkey Results for RQ1 (and RQ3): Effect of Reduction on Baseline Suite Table I provides information on the base test suites across the five versions of SpiderMonkey studied. A purely failure-based quick test such as was used at NASA (**Min**$_{\textbf{Fail}}$) produces very poor code coverage (e.g., covering almost 100 fewer *functions* than the original suite, and over 3,000 fewer branches), as expected. It also loses fault detection power rapidly, only finding ∼7 distinct faults on the next version of the code base, while suites based on all tests can detect ∼20-∼36 faults. Given its extremely short runtime, retaining such a suite as a pure regression test suite may be useful, but it cannot be expected to work as a good quick test. Second, the suites greedily minimized by statement coverage (**GE**$_{ST}$**(Full)** and **GE**$_{ST}$**(Min)**) are very quick, and potentially useful, but lose a large amount of branch coverage and do not provide enough tests to fill a five minute quick test. The benefits of suite minimization by statement coverage (or branch coverage) were represented in the 30 second and five minute budget experiments shown below by the Δ prioritizations, which produce the same results, with the exception that for short budgets tests included because they uniquely cover some entity are less likely to be included than with random sampling of the minimized suites.

The most important total suite result is that the cause reduced **Min** suite retains (or improves) many properties of the **Full** suite that are *not* guaranteed to be preserved by the modified *ddmin* algorithm. For version 1.6, only five branches are "lost", and (most strikingly) the number of failing

Table I. SpiderMonkey Unlimited Test Budget Results

Suite	Size (# Tests)	Time(s)	Statement Coverage	Branch Coverage	Function Coverage	# Failing Tests	Estimated # Faults
Release 1.6, 12/22/06							
Full	13,323	14,255.068	19,091	**14,567**	966	1,631	22
Min	13,323	**3,566.975**	19,091	14,562	966	1,631	**43**
Min$_{\text{FN}}$	13,323	2,636.648	18,876	14,280	966	1,627	39
Min$_{\text{Fail}}$	1,019	169.594	16,020	10,875	886	1,019	22
GE$_{ST}$(Full)	168	182.823	19,091	14,135	966	14	5
GE$_{ST}$(Min)	171	47.738	19,091	14,099	966	14	8
GE-Min	168	25.443	19,091	13,722	966	12	8
Non-release snapshot, 2/24/07							
Full	13,323	9,813.781	**22,392**	**17,725**	**1,072**	**8,319**	20
Min	13,323	**3,108.798**	22,340	17,635	1,070	4,147	**36**
Min$_{\text{FN}}$	13,323	2,490.859	22,107	17,353	1,070	1,464	32
Min$_{\text{Fail}}$	1,019	148.402	17,923	12,847	958	166	7
GE$_{ST}$(Full)	168	118.232	21,305	16,234	1,044	116	5
GE$_{ST}$(Min)	171	40.597	21,323	16,257	1,045	64	3
GE-Min	168	25.139	20,887	15,424	1,047	8	6
Non-release snapshot, 4/24/07							
Full	13,323	16,493.004	**22,556**	**18,047**	**1,074**	189	**10**
Min	13,323	**3,630.917**	22,427	17,830	1,070	**196**	6
Min$_{\text{FN}}$	13,323	2,522.145	22,106	17,449	1,066	167	5
Min$_{\text{Fail}}$	1,019	150.904	18,032	12,979	961	158	5
GE$_{ST}$(Full)	168	206.033	22,078	17,203	1,064	4	1
GE$_{ST}$(Min)	171	45.278	21,792	16,807	1,058	3	1
GE-Min	168	24.125	21,271	15,822	1,052	2	2
Release 1.7, 10/19/07							
Full	13,323	14,282.776	**22,426**	**18,130**	**1,071**	**528**	**15**
Min	13,323	**3,401.261**	22,315	17,931	1,067	274	10
Min$_{\text{FN}}$	13,323	2,474.55	22,022	17,565	1,065	241	11
Min$_{\text{Fail}}$	1,019	168.777	18,018	13,151	956	231	12
GE$_{ST}$(Full)	168	178.313	22,001	17,348	1,061	6	2
GE$_{ST}$(Min)	171	43.767	21,722	16,924	1,055	5	2
GE-Min	168	24.710	21,212	15,942	1,045	4	4
Release 1.8.5, 3/31/11							
Full	13,323	4,301.674	**21,030**	**15,854**	**1,383**	**11**	**2**
Min	13,323	**2,307.498**	20,821	15,582	1,363	3	1
Min$_{\text{FN}}$	13,323	1,775.823	20,373	15,067	1,344	3	**2**
Min$_{\text{Fail}}$	1,019	152.169	16,710	11,266	1,202	2	1
GE$_{ST}$(Full)	168	51.611	20,233	14,793	1,338	1	1
GE$_{ST}$(Min)	171	28.316	19,839	14,330	1,327	1	1
GE-Min	168	21.550	18,739	13,050	1,302	1	1

Legend: Full = Original Suite; Min = *ddmin*(Full, Statement Coverage); Min$_{\text{FN}}$ = *ddmin*(Full, Function Coverage); Min$_{\text{Fail}}$ = *ddmin*(Full, Failure); GE$_{ST}$ = Greedy Selection for Statement Coverage

Minimization of original (**Full**) suite test case:

```
tryItOut("L: {prototype%=new ({}).__lookupGetter__(); }");
tryItOut("__noSuchMethod__ = __parent__;");
tryItOut("__iterator__ = prototype;");
tryItOut("try { throw constructor; } catch(__parent__ if (function()
          {gc()})()) { try { throw prototype; } catch(__proto__)
          { throw __proto__; } finally {return (prototype =
          <x><y/></x>.__noSuchMethod__.unwatch("constructor")); }
          } finally { export *; } ");
```

Minimization of reduced (**Min**) suite test case:

```
tryItOut("do {<><x><y/></x></>; } while(({}) && 0);");
tryItOut("try { throw constructor; } catch(__parent__ if (function()
          {gc()})()) { try { throw prototype; } catch(__proto__)
          { throw __proto__; } finally {return (prototype =
          <x><y/></x>.__noSuchMethod__.unwatch("constructor")); }
          } finally { export *; } ");
```

Figure 7. Change in minimized tests for change in fault example

test cases is *unchanged*. Most surprisingly, the estimated distinct fault detection is *improved*: it has grown from $\sim$22 faults to $\sim$43 faults. The difference in results is statistically significant: dividing the test populations into 30 equal-sized randomly selected test suites for both full and minimized tests, the average minimized suite detects 11.83 distinct faults on average, while the average full suite only detects 7.6 faults, with a p-value of $5.2 \cdot 10^{-10}$ under U-test. It is unclear how any bias in the fault estimation method produces this strong an effect, given that the tests produced are similar in structure and nature, and the estimation method was applied in the same way to all tests. One likely hypothesis as to the cause of the failure preservation level is that *ddmin* tends to preserve failure because failing test cases have unusually *low* coverage in many cases. Since the *ddmin* algorithm attempts to minimize test size, this naturally forces it to attempt to produce reduced tests that also fail; moreover, some failures execute internal error handling code (some do not, however—many test cases violating `jsfunfuzz` semantic checks, for example).

The apparent increased diversity of faults, however, is surprising and unusual, and suggests that the use of *ddmin* as a test mutation-based fuzzing tool might be an interesting area for future research. In retrospect, it is obvious that *ddmin* takes as input a test case and generates a large number of related, but distinct, new test cases—it is, itself, a test case generation algorithm. It seems safe to say that the new suite is essentially as good at detecting faults and covering code, with much better runtime (and therefore better test efficiency [19]). In order to better understand the reasons why the number of distinct faults identified increased, the authors examined by hand a number of the cases where the identified bug changed from the original suite to the reduced suite. Understanding exactly what is taking place in 1,000 (or even 100) line`jsfunfuzz`-produced test cases was unfortunately almost impossible without intimate understanding of the code involved. A tool to identify cases where the difference in estimated fault also resulted in a change in minimized (by traditional failure-based delta debugging) test cases was developed and applied to these results. The pattern in the 53 identified instances was similar: the cases had considerable overlap in actual JavaScript code, but the result for the coverage-minimized tests differed in one or two lines—it was more often shorter or simpler, but sometimes larger or more complex. The last line of the test case, triggering failure, was the same in all but two cases. Figure 7 shows an example of a typical case. This case appears to be an instance of two partial fixes for one underlying fault, though the code changes in both instances are complex (and lengthy) and involve enough other modification (e.g. what appear to be refactorings) that determining this for certain would require considerable SpiderMonkey code base expertise.

It appears that the phenomenon of detecting "more faults" may be strongly related to the very buggy nature of SpiderMonkey 1.6. For many faults, there may be several fixes, made at different times, that affect different manifestations of a single underlying conceptual "bug." The binary-search

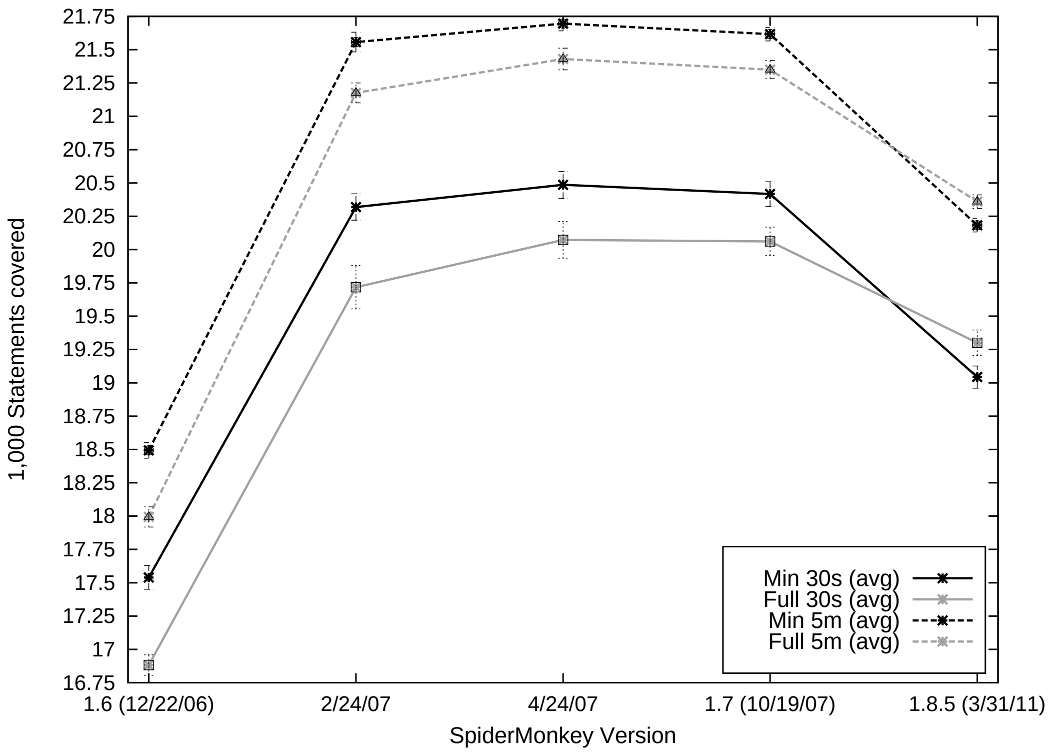

Figure 8. Statement coverage for 30s and 5m quick tests across SpiderMonkey versions

based approach is unable to generalize these into a single fault since, in fact, various aspects of "the fault" were fixed at different times, affecting different exposing test cases. The coverage-based minimization increases the set of "fixes" for one underlying fault that are needed to be included in order to reach a state where no test cases fail, because the preconditions that come before the statements inducing the failure become more varied. Why are the preconditions more varied? One guess, which seems to match hand examination, is that some preconditions appear much more frequently in test cases than others, due to the combinatorics of `jsfunfuzz`. These tend to be the ones that induce failure in the non-minimized suite test cases. However, these multiple appearances of the more-common precondition to failure appear to be coverage-redundant with respect to each other, so the cause reduced suite eliminates most of the appearances of these common preconditions, allowing other preconditions to trigger failure in a different way. While understanding precisely what is going on would probably require the expertise of SpiderMonkey developers, or very long investigation, this hypothesis seems to match the hand examination of particular test cases. In short, it is probably only true in a somewhat over-formal sense (based on the actual fixes applied to SpiderMonkey, rather than a reasonable notion of "different bugs") that the coverage minimized suite detects *more faults*. However, it is also true that the reduced suite serves as a considerably better regression test in that a larger portion of the needed fixes to SpiderMonkey 1.6 must be applied to produce a successful run of the whole suite.

2.4.5. SpiderMonkey Results for RQ2 (and RQ3): Effectiveness for Quick Testing Tables II and III show how each proposed quick test approach performed on each version, for 30 second and five minute test budgets, respectively. All nondeterministic or time-limited experiments were repeated 30 times. The differences between minimized (Min) suites and Full suites for each method and budget are statistically significant at a 95% level, under a two-tailed U-test, with only one exception: the improvement in fault detection for the non-prioritized suites for the 4/24 version is not significant. The best results for each suite attribute, SpiderMonkey version, and test budget combination are shown in bold (ties are only shown in bold if some approaches did not perform as well as the

Table II. SpiderMonkey 30s Test Budget Mean Results

Suite	Size (# Tests)	Statement Coverage	Branch Coverage	Function Coverage	# Failing Tests	Estimated # Faults
Release 1.6, 12/22/06						
Full	27.1	16,882.1	11,895.4	897.0	2.8	2.8
Full+ΔST	27.6	18,270.5	13,050.1	949.9	4.2	4.0
Full+ΔBR	25.7	18,098.2	13,144.7	936.4	2.8	2.6
Min	102.2	17,539.4	12,658.6	916.6	12.6	7.1
Min+ΔST	**106.9**	**18,984.7**	**13,873.6**	**963.1**	12.0	**9.0**
Min+ΔBR	77.3	18,711.6	13,860.9	958.8	7.3	5.4
Min$_{FN}$	135.7	17,320.5	12,300.2	922.6	**17.0**	6.0
Non-release snapshot, 2/24/07						
Full	37.8	19,718.0	14,644.9	991.6	23.9	3.1
Full+ΔST	45.1	19,958.0	14,813.9	1,006.0	35.1	3.0
Full+ΔBR	39.4	20,502.2	15,511.6	1,021.8	23.5	4.4
Min	105.0	20,319.3	15,303.5	1,013.2	32.2	4.0
Min+ΔST	92.9	**21,238.1**	15,984.8	**1,049.1**	35.6	2.7
Min+ΔBR	**117.2**	21,167.2	**16,183.9**	1,042.0	**46.4**	5.0
Min$_{FN}$	137.8	19,896.1	14,783.4	1,017.6	14.9	**5.1**
Non-release snapshot, 4/24/07						
Full	23.8	20,072.8	15,108.5	999.0	0.6	0.6
Full+ΔST	25.3	21,111.7	15,948.7	1,040.3	2.0	**2.0**
Full+ΔBR	25.8	21,101.4	16,122.2	1,037.8	2.0	**2.0**
Min	100.8	20,485.7	15,564.3	1,016.6	1.6	1.6
Min+ΔST	**113.5**	**21,731.8**	16,631.1	**1,056.9**	2.0	**2.0**
Min+ΔBR	105.4	21,583.7	**16,763.8**	1,056.4	**3.0**	**2.0**
Min$_{FN}$	145.1	20,001.4	14,983.1	1,017.4	1.8	1.2
Release 1.7, 10/19/07						
Full	27.5	20,061.6	15,288.4	1,002.0	1.4	1.4
Full+ΔST	30.0	21,112.9	16,140.3	1,042.0	**4.0**	**3.0**
Full+ΔBR	29.2	21,047.3	16,280.5	1,036.3	2.0	2.0
Min	103.5	20,416.8	15,675.1	1,015.7	1.8	1.8
Min+ΔST	**116.4**	**21,668.4**	16,762.6	**1,054.0**	**4.0**	**3.0**
Min+ΔBR	109.7	21,535.6	**16,908.7**	1,053.8	**4.0**	**3.0**
Min$_{FN}$	148.6	19,967.7	15,099.4	1,019.0	2.9	1.6
Release 1.8.5, 3/31/11						
Full	83.4	19,300.8	13,907.5	1,291.4	0.0	0.0
Full+ΔST	98.8	19,876.9	14,430.8	1,320.4	**1.0**	**1.0**
Full+ΔBR	98.0	19,963.1	**14,494.2**	1,326.0	**1.0**	**1.0**
Min	140.8	19,043.3	13,621.1	1,286.0	0.0	0.0
Min+ΔST	**179.4**	19,848.2	14,338.0	1,325.0	**1.0**	**1.0**
Min+ΔBR	178.3	**19,975.8**	14,453.0	**1,329.0**	**1.0**	**1.0**
Min$_{FN}$	158.4	18,389.2	12,882.8	1,278.3	0.0	0.0

Table III. SpiderMonkey 5m Test Budget Mean Results

Suite	Size (# Tests)	Statement Coverage	Branch Coverage	Function Coverage	# Failing Tests	Estimated # Faults
Release 1.6, 12/22/06						
Full	269.4	17,993.2	13,227.5	933.2	32.6	7.4
Full+ΔST	270.2	19,093.0	14,195.9	966.0	23.0	8.0
Full+ΔBR	272.1	19,064.2	14,504.3	962.0	24.0	9.0
Min	1,001.2	18,493.2	13,792.4	949.8	121.1	18.8
Min+ΔST	1,088.9	**19,093.0**	14,298.4	**966.0**	138.7	**22.9**
Min+ΔBR	**1,093.1**	19,091.0	**14,563.2**	964.0	146.3	20.9
Min$_{FN}$	1,662.2	18,367.0	13,592.1	951.7	**204.7**	20.9
Non-release snapshot, 2/24/07						
Full	381.4	21,175.5	16,308.2	1,037.6	237.8	8.3
Full+ΔST	404.5	21,554.0	16,612.1	1,051.0	258.9	7.0
Full+ΔBR	398.7	21,664.2	16,833.1	1,051.0	252.6	8.0
Min	1,124.9	21,556.8	16,711.3	1,051.1	347.9	10.6
Min+ΔST	**1,255.6**	21,899.8	17,021.9	**1,064.0**	**383.6**	15.0
Min+ΔBR	1,227.7	**21,940.0**	**17,180.0**	1,058.1	356.5	12.0
Min$_{FN}$	1,769.4	21,322.6	16,430.6	1,052.1	190.9	**15.5**
Non-release snapshot, 4/24/07						
Full	237.8	21,430.2	16,663.0	1,043.8	7.8	2.7
Full+ΔST	244.7	22,139.0	17,279.3	1,064.0	7.0	2.0
Full+ΔBR	241.2	22,126.8	17,483.3	1,064.0	6.1	3.0
Min	1,085.6	21,695.8	16,960.1	1,051.4	16.0	2.9
Min+ΔST	1,113.8	22,106.9	17,308.0	**1,065.3**	18.0	**5.0**
Min+ΔBR	**1,135.1**	**22,178.0**	**17,550.5**	1,063.0	17.1	3.0
Min$_{FN}$	1,790.1	21,380.9	16,568.2	1,050.6	**23.0**	2.6
Release 1.7, 10/19/07						
Full	263.7	21,350.0	16,796.8	1,042.2	10.9	3.6
Full+ΔST	282.1	22,074.0	17,438.1	**1,063.0**	17.8	4.0
Full+ΔBR	278.6	**22,087.5**	**17,670.1**	1,061.0	11.0	5.0
Min	1,072.9	21,616.9	17,070.0	1,050.4	22.2	4.8
Min+ΔST	**1,186.3**	22,025.0	17,425.7	**1,063.0**	26.1	6.0
Min+ΔBR	1,165.8	22,082.3	17,676.6	1,060.0	24.0	**7.0**
Min$_{FN}$	1,794.4	21,306.7	16,673.1	1,049.0	**33.6**	4.9
Release 1.8.5, 3/31/11						
Full	908.8	20,359.8	15,057.9	1,344.2	0.6	1.1
Full+ΔST	982.6	20,514.6	15,182.8	1,349.0	2.0	1.0
Full+ΔBR	1,001.4	**20,638.6**	**15,312.7**	**1,366.7**	**3.0**	**2.0**
Min	1,649.1	20,182.0	14,850.2	1,333.5	0.4	1.2
Min+ΔST	**1,851.3**	20,402.3	15,067.5	1,343.0	1.0	1.0
Min+ΔBR	1,661.0	20,445.5	15,108.4	1,348.4	1.0	1.0
Min$_{FN}$	1,819.6	19,727.7	14,306.2	1,323.7	0.5	0.3

best methods). For number of tests executed, the best value for suites *other than* **Min$_{\mathbf{FN}}$** are shown in bold, as **Min$_{\mathbf{FN}}$** produced much shorter tests and thus consistently executed the most test cases. Results for absolute coverage prioritization are omitted from the table to save space, as Δ prioritization always performed much better; absolute often performed worse than random.

Figure 8 graphically exhibits the raw differences in statement coverage for the suites sampled as quick tests, ignoring the effects of prioritization, with one-standard-deviation error bars on points. The power of coverage-based cause reduction can be seen in Tables II and III by comparing "equivalent" rows for any version and budget: results for each version are split so that **Full** results are the first three rows and the corresponding prioritization for the **Min** tests are the next three rows. For the first three versions tested, it is almost always the case that for every measure, the reduced suite value is better than the corresponding full suite value. For 30s budgets this comparison even holds true for version 1.7, nearly a year later. Moving from 1.6 to 1.7 involves over 1,000 developer commits and the addition of 10,000+ new lines of code (a 12.5% increase). In reality, it is highly unlikely that developers would not have a chance to produce a better baseline on more similar code in a four year period (or, for that matter, in any one month period). The absolute effect size, as measured by the lower bound of a 95% confidence interval, is often large—typically 500+ lines and branches and 10 or more functions, and in a few cases more than 10 faults.

It is difficult to generalize from one subject, but based on the SpiderMonkey results, a good initial quick test strategy to try for other projects might be to combine cause reduction by statement coverage with test case prioritization by either Δ statement or branch coverage. In fact, limitation of quick tests to very small budgets may not be critical. Running only 7 minutes of minimized tests on version 1.6 detects an average of twice as many faults as running 30 minutes of full tests and has (of course) indistinguishable average statement and branch coverage. The difference is significant with p-value of $2.8 \cdot 10^{-7}$ under a U-test. In general, for SpiderMonkey versions close to the baseline, running N minutes of minimized tests, however selected, seems likely to be much better than running N minutes of full tests. The real limitation is probably how many minimized tests are available to run, due to the computational cost of minimizing tests.

The **GE-Min** suite is also a highly competitive 30 second quick test; it only detects one less fault than the average 30 second ST-prioritized **Min** quick test, and arguably is more efficient (given that it requires five fewer seconds to execute). For later versions of SpiderMonkey, while it does not perform well in terms of branch coverage (invariably producing less coverage than all prioritized **Min** suites), it is *equal to or superior to* the best 30 second quick test in terms of fault detection, the most important attribute. Running **GE-Min** as a precursor to other quick tests seems likely to be a sound rule of thumb for efficient regression.

The **Min$_{\mathbf{FN}}$** suite also performs adequately as a quick test. For 30s and 5m tests, it is not as effective (by any measure) as the prioritized **Min** suites, but outperforms **Full** based suites in fault detection for the first two versions, by a substantial (and statistically significant) number of faults.

2.4.6. SpiderMonkey Results for RQ4: Runtime for Cause Reduction These results are based on fully reduced test cases. However, examining the reduction histories of a sample of 33 test cases showed that much of the reduction is completed in the early stages of the process. The cost to reach full 1-minimality is high, but large runtime reductions can be obtained considerably faster. Figures 9 and 10 show how the runtime of the current minimization, as a fraction of the original test case's runtime, decreased over time (in seconds) spent running the *ddmin* algorithm. Data points represent intermediate results (most minimal preserving test cases thus far seen) during reduction of the 33 sample test cases. In a very few cases, early reductions actually increased runtime by up to 20% (recall that cause reduction removes test case components, rather than targeting actual runtime), but generally early stages showed rapid improvement in runtime, followed by a slow convergence on 1-minimality. Table IV shows the average times to achieve complete 1-minimality, 80% of total reduction, 50% of total reduction, and 20% of total reduction, respectively, for statement and function coverage. In traditional delta debugging, it is usually worth spending the time to produce 1-minimal tests, given that human effort is very expensive compared to compute-time. For quick tests, this is also probably the case, in that every lost opportunity for reduction results in a repeated

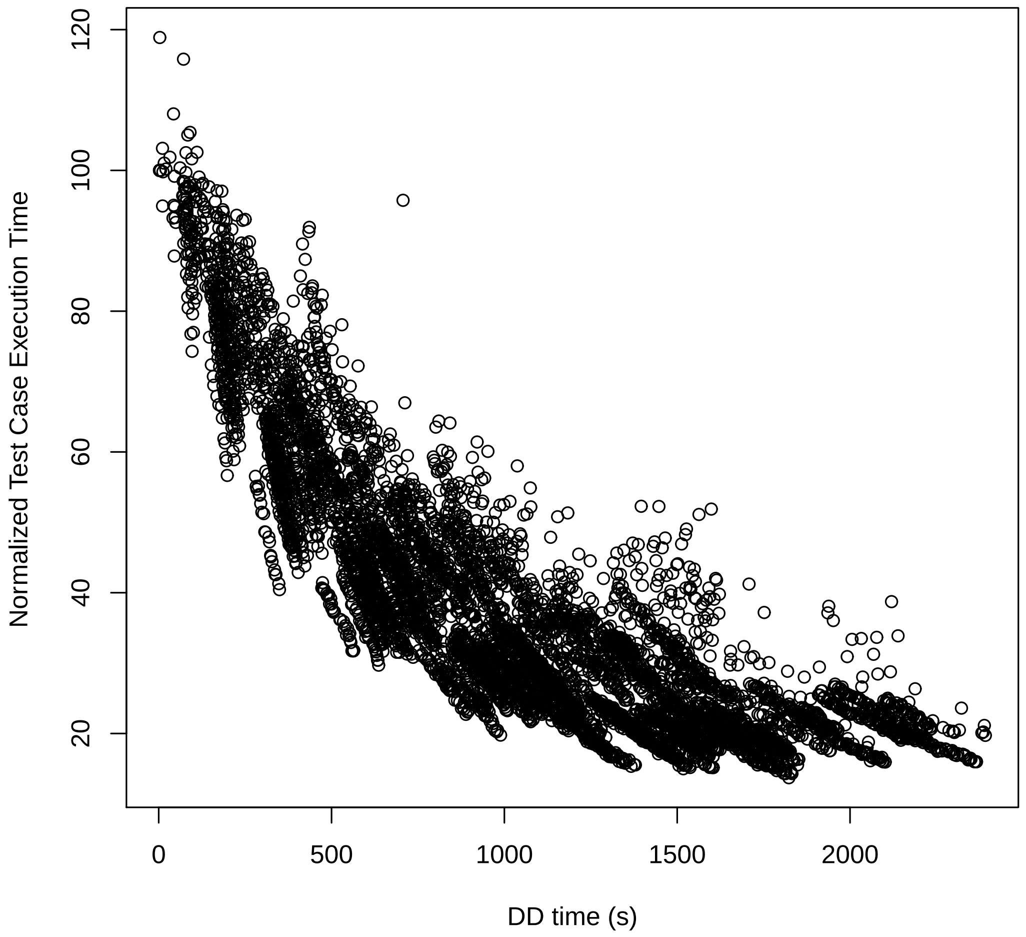

Figure 9. Time spent reducing vs. test case runtime, statement coverage

Table IV. Time to reduce test cases, SpiderMonkey

	Average time (s) to obtain % of total runtime reduction			
Reduction criteria	100% (1-minimal)	80%	50%	20%
Statements	1,781.7	1,553.1	542.3	207.2
Functions	448.1	258.6	108.6	48.0

penalty to future test efficiency. However, in some applications, as discussed in Section 3, a quick reduction without 1-minimality may be extremely useful.

2.5. YAFFS 2.0 Flash File System Case Study

YAFFS2 [37] is a popular open-source NAND flash file system for embedded use; it was the default image format for early versions of the Android operating system. Due to the lack of a large set of real faults in YAFFS2, mutation testing was used to check the claim that cause reduction not only preserves source code coverage, but tends to preserve fault detection and other useful properties of randomly generated test cases. The initial evaluation for the ICST 2014 paper [11]

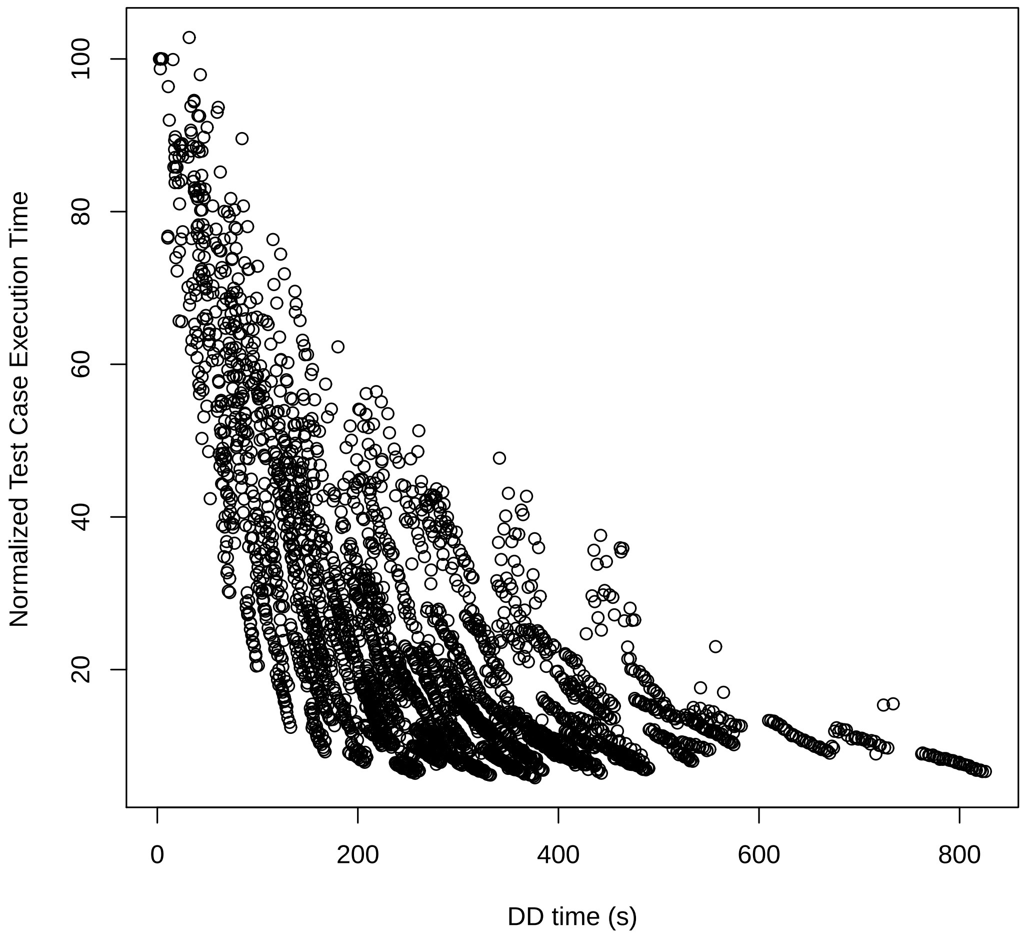

Figure 10. Time spent reducing vs. test case runtime, function coverage

used 1,992 mutants, randomly sampled from the space of all 15,246 valid YAFFS2 mutants, using the C mutation software shown to provide a good proxy for fault detection [38], with a sampling rate (13.1%) above the 10% threshold suggested in the literature [39]. Sampled mutants were not guaranteed to be killable by the API calls and emulation mode tested. Table V shows how full and quick test suites for YAFFS2 compared. MUT indicates the number of mutants killed by a suite. Results for |BR| are omitted, as absolute prioritization by branch and statement coverage produced equivalent test suites. Runtime reduction for YAFFS2 was not as high as with SpiderMonkey tests (1/2 reduction vs. 3/4), due to a smaller change in test size and higher relative cost of test startup. The average length of original test cases was 1,004 API calls, while reduced tests averaged 213.2 calls. The most likely cause of the smaller reduction is that the YAFFS2 tester uses a feedback [12] model to reduce irrelevant test operations. Basic retention of desirable aspects of **Full** was, however, excellent: only one branch was "lost", function coverage was perfectly retained, and 99.1% as many mutants were killed. The reduced suite killed 6 mutants not killed by the original suite. It is not known whether mutant scores are good indicators of the ability of a suite to find, e.g., subtle optimization bugs in compilers. Mutant kills are, however, a plausible method for estimating the ability of a suite to detect many of the shallow bugs a quick test aims to expose before code is committed or subjected to more testing. Even with lesser efficiency gains, cause reduction

Table V. YAFFS2 Results for ICST 2014 Experiment

Suite	Size (# Tests)	Time(s)	Statement Coverage	Branch Coverage	Function Coverage	Mutants Killed
Full	4,240	729.0	4,049	1,925	332	616
Min	4,240	402.5	4,049	1,924	332	611
Full	174.4	30.0	4,007.4	1,844.0	332.0	568.3
Full+ΔST	372.5	30.0	**4,049.0**	1,918.0	332.0	594.0
Full+ΔBR	356	30.0	**4,049.0**	**1,925.0**	332.0	**596.0**
Full+|ST|	112.5	30.0	4,028.0	1,889.0	332.0	589.0
Min	315.8	30.0	4,019.7	1,860.5	332.0	559.0
Min+ΔST	**514.7**	30.0	**4,049.0**	1,912.0	332.0	571.0
Min+ΔBR	500.0	30.0	**4,049.0**	1,924.0	332.0	575.0
Min+|ST|	255.0	30.0	4,028.0	1,879.0	332.0	552.0
Full	1,746.8	300.0	4,044.7	1,916.0	332.0	608.7
Full+ΔST	2,027.0	300.0	**4,049.0**	1,921.0	332.0	601.0
Full+ΔBR	2,046.0	300.0	**4,049.0**	**1,925.0**	332.0	604.0
Full+|ST|	1,416.0	300.0	4,042.0	1,916.0	332.0	**611.0**
Min	3,156.6	300.0	4,048.1	1,920.0	332.0	607.1
Min+ΔST	**3,346.0**	300.0	**4,049.0**	1,924.0	332.0	601.0
Min+ΔBR	3,330.0	300.0	**4,049.0**	1,924.0	332.0	605.0
Min+|ST|	2,881.7	300.0	**4,049.0**	1,924.0	332.0	**611.0**

Table VI. YAFFS2 Results: New Experiment

Suite	Size (# Tests)	Time(s)	Statement Coverage	Branch Coverage	Function Coverage	Mutants Killed	Average Length
Full	5,000	798.7	4,098	**1,925**	337	**2,990**	1,000
Min	5,000	**443.2**	4,098	1,924	337	2,949	213
Min$_{\text{FN}}$	5,000	275.5	4,091	1,902	337	2,518	86.6
GE$_{ST}$(Full)	13	4.6	4,098	1,914	337	2,882	1,000
GE$_{ST}$(Min)	13	2.1	4,098	1,906	337	2,440	213
GE-Min	13	1.0	4,099	1,893	337	2,630	44
Full	153.1	30.0	4,055.1	1,843.6	337	2,842.5	1,000
Full+ΔST	155	30.0	**4,098**	**1,918**	337	**2,889**	1,000
Full+|ST|	110	30.0	4,078	1,889	337	2,818	1,000
Min	**267.7**	30.0	4,068.6	1,856.2	337	2,787.7	213.0
Min+ΔST	260	30.0	**4,098**	1,912	337	2,887	213.0
Min+|ST|	211	30.0	4,078	1,879	337	2,828	213.0
Full	1,676.5	300.0	4,094.1	1,915.6	337	**2,971.0**	1,000
Full+ΔST	1,685	300.0	**4,098**	**1,925**	337	2,967	1,000
Full+|ST|	1,336	300.0	4,092	1,916	337	2,950	1,000
Min	**3,012**	300.0	4,097	1,921.3	337	2,941.7	213.0
Min+ΔST	2,689	300.0	**4,098**	1,921	337	2,935	213.0
Min+|ST|	2,346	300.0	**4,098**	1,924	337	2,933	213.0

plus *absolute* coverage prioritization is by far the best way to produce a five minute quick test, maximizing five-minute mutant kills without losing code coverage. *All* differences in methods were significant, using a two-tailed U-test (in fact, the highest p-value was 0.0026).

Table VII. Time to reduce test cases, YAFFS2

		Average time (s) to obtain % of total reduction			
Reduction criteria	Measure	100% (1-minimal)	80%	50%	20%
Statements	Runtime	627.9	627.9	612.9	215.9
Statements	Length	611.0	563.6	265.3	91.7
Functions	Length	84.4	58.9	18.2	3.4

For this paper, an additional set of experiments were performed on YAFFS2, using a new set of 5,000 randomly generated test cases and a much larger set of 10,078 mutants. The new experiment added function coverage minimization and the GE-Min algorithm application as well. Table VI shows the results of this experiment. The results are somewhat strange. As before, the reduced **Min** suite has a little over half the runtime of the **Full** suite, but preserves statement and function coverage perfectly (of course) and only loses 1 branch. It fails to kill as many mutants as the **Full** suite, however, though only by 1.4% of total killed mutants. The **Min$_{\mathbf{FN}}$** suite produces good coverage for such a short runtime, but loses 15% of mutation killing power and more than 20 branches. The **GE**$_{ST}$ and **GE-Min** suites are extremely efficient; the inadequacy of the **Min$_{\mathbf{FN}}$** suite can be seen by the fact that the **GE**$_{ST}$**(Full)** and **GE-Min** suites both kill more mutants, in less than 5 seconds (just 1 second for **GE-Min**). Incidentally, **GE-Min** also added, as a result of reduction, one additional statement covered by no test in the original 5,000 tests (remember, delta debugging only guarantees preservation, but can in principle increase the effect of a test case).

The disparity between **GE-Min**'s lower branch coverage and higher mutation kill rate compared to **GE**$_{ST}$**(Min)** brings us to an interesting point: YAFFS2 results are somewhat difficult to understand because, unusually [32], coverage (across multiple measures) and mutation kill are not in agreement for many suite pairs. This disparity is most evident in the results for 30s and 5m quick test suites, with random sampling. For these, **Min**-based suites consistently, if narrowly, perform better than **Full** suites for statement and branch coverage, but perform consistently worse for mutation kills. Similarly, ordering by absolute statement coverage decreased mutation kill rates in all but one case, despite (as expected) increasing coverage. Using **Full** suites misses more code, but seems to find more faults, in these settings. In fact, the un-prioritized **Full** suite was the worst 30s quick test for statement and branch coverage, and the third best for 5m, but killed the most mutants for 5m tests, and performed better than all but the Δ prioritized suites for 30s. This somewhat unexpected result merits examination in future work: perhaps coverage alone is particularly weak in its ability to produce varied behavior in YAFFS2, due to the complex state of the file system itself, and the advantage longer tests have in producing complex state [18]. Some other peculiar interaction of coverage and mutation testing (since there aren't enough real YAFFS2 faults to investigate) [40] may also be involved. For now, it seems that while cause reduction is not ideal for YAFFS2 quick testing, the basic preservation of most properties holds even here. Most results in the table are statistically significant at a much lower than 0.001 p-value by U-test; the exception is the difference in branch coverage for randomly sampled 30s quick tests, which only has a p-value of 0.0518.

YAFFS2 reduction rate data was also of interest. Because runtimes for YAFFS2 tests are more similar, there was no clear curve (and for function reduction, runtimes were almost immediately too small to either measure well or to make much difference to test suite runtime). However, examining test case length, the data was more interesting, as shown in Table VII. Again, the time to 50% reduction (by API calls, not runtime) is considerably less than half of the total time for reduction.

2.6. *GCC: The Potentially High Cost of Reduction*

The SpiderMonkey and YAFFS case studies both featured systems where reduction is a simple matter of off-the-shelf delta debugging and computing coverage on tests is not much of a burden, due to source simplicity. This section reports on some preliminary experiments in applying cause reduction to test cases produced by Csmith [17] using an older version of GCC (4.3.0), the most recent version (4.9.0), and also an LLVM development snapshot from June 19 2014 (used because

Table VIII. C Compiler Results

GCC

Size (bytes)			Compile time (s)			Time (m) to obtain % of reduction			
Unreduced	Reduced	Reduction	Unreduced	Reduced	Reduction	100%	80%	50%	20%
66930	14673	78.1 %	0.15	0.14	6.9 %	740.6	93.7	3.3	0.8
126187	17942	85.8 %	0.15	0.15	4.5 %	1705.0	200.7	28.8	3.7
135585	52488	61.3 %	0.33	0.23	29.3 %	3032.1	357.9	24.8	2.1
192307	39291	79.6 %	0.64	0.37	41.6 %	1629.1	134.9	47.0	5.3
220127	66856	69.6 %	1.18	0.82	30.3 %	1781.4	440.0	77.2	7.6
236253	38107	83.9 %	0.73	0.37	49.2 %	1046.7	169.4	48.7	6.4
258698	19019	92.6 %	0.36	0.20	43.1 %	1298.3	257.0	49.0	7.9
279633	36899	86.8 %	0.48	0.43	10.2 %	2648.9	593.0	77.4	8.3

LLVM

Size (bytes)			Compile time (s)			Time (m) to obtain % of reduction			
Unreduced	Reduced	Reduction	Unreduced	Reduced	Reduction	100%	80%	50%	20%
84660	33473	60.5 %	0.30	0.08	73.7 %	1555.4	345.7	62.5	6.1
112469	44436	60.5 %	0.32	0.08	74.7 %	3855.8	409.9	91.9	10.2
118224	34680	70.7 %	0.30	0.08	72.9 %	3512.8	260.3	96.0	17.1
118342	23166	80.4 %	0.26	0.08	69.5 %	3107.9	192.2	81.7	13.7

recent released versions failed to build with code coverage enabled). C programs are harder to reduce effectively without using more sophisticated delta debugging methods, and coverage extraction imposes a higher overhead. Each Csmith output was reduced using C-Reduce [8] modified to support coverage constraints, which turned out to be expensive, often requiring more than 24 hours on a modern Core i7. Several factors explain this poor reduction performance. First, an individual run of a C compiler takes longer than a corresponding run of SpiderMonkey or YAFFS2, and produces hundreds of coverage files that must be processed. Second, the test cases themselves are larger: an average of 3,659 reduction units (lines) vs. about 1,000 for SpiderMonkey and YAFFS.

The first experiment, using GCC 4.3.0, started with 12 test cases (C programs generated by Csmith) that triggered five different bugs in GCC 4.3.0 that cause the compiler to crash. For this experiment only C-Reduce's line-based reduction passes were enabled, disabling its finer-grained components. Even so, the test cases were reduced in size by an average of 37%. After reduction, each test case still crashed the compiler. However, none of the reduced or unreduced test cases in this set crashed the next major release of GCC, 4.4.0, although they did manage to cause 419 more lines of GCC code to be covered while being compiled by the newer version. Turning to branch coverage, an even more surprising result appears: the reduced test cases cover an additional 1,034 branches in GCC 4.3.0 and an additional 297 in 4.4.0, relative to the unreduced test cases. Function coverage is also slightly improved in the minimized suite for GCC 4.4.0: 7,692 functions covered by the 12 minimized tests vs. only 7,664 for the original suite. In this experiment, test case efficiency improved only marginally: the total compilation time for the 12 reduced programs was 3.23 seconds vs. 3.53 seconds for the original tests.

The results of the experiment with newer compilers—the latest GCC and a recent LLVM snapshot—are summarized in Table VIII. In this case C-Reduce was not restricted to use only its line-based reduction passes, and consequently the reduction ratio improved, to an average of 75%, meaning that the average reduced test case was a quarter of its unreduced size. Reduction improved test case execution time by an average of 42%. The right-most columns of Table VIII show that most of C-Reduce's reduction occurs early. For example, 50% of its total reduction is usually obtained within the first 90 minutes. The last 20% of the reduction benefit, on the other hand, requires many hours. Since this experiment used very recent versions of GCC and LLVM, it was not possible to explore the effect of reduced test cases on later versions of the compilers.

In the end it is not entirely clear that large and complex artifacts such as GCC and LLVM can be adequately smoke-tested within a five-minute budget, even after test case reduction and prioritization.

2.7. Threats to Validity

First, cause reduction by coverage for quick tests is intended to be used on the highly redundant, inefficient tests produced by aggressive random testing. While random testing is sometimes highly effective for finding subtle flaws in software systems, and essential to security-testing, by its nature it produces test cases open to extreme reduction. It is possible that human-produced test cases (or test cases from directed testing that aims to produce short tests) would not reduce well enough to make the effort worthwhile, or that the effort to maintain test validity would be too onerous in many settings. The quick test problem is formulated specifically for random testing, though many of the same arguments also hold for model checking traces produced by SAT or depth-first-search, which also tend to be long, redundant, and have independent components. The primary threat to validity is external: experimental results are based on one large case study on a large code base over time, one mutation analysis of a smaller but also important and widely used program, and a few indicative tests on two extremely critical larger systems: the GCC and LLVM compilers. The primary internal threat to validity is that the code is in error; however, the experiments produce a number of easy opportunities for sanity checks, including the output of test case execution, and the authors have taken care to avoid implementation errors. Moreover, much of the implementation is taken from existing code for `jsfunfuzz` and the delta debugging scripts provided online, or has been used and tested for years. There is also a construct threat in the method for counting faults for SpiderMonkey, but the raw failure rates and coverage values alone support a belief that cause reduction is useful.

3. CAUSE REDUCTION FOR EFFECTIVE SEEDED SYMBOLIC EXECUTION

Quick testing is primarily intended to improve the efficiency of highly redundant randomly generated test cases, based on a reduction in test case runtime. Can cause reduction serve any purpose for human-generated tests, and/or when actual runtime reduction is minimal? In fact, the accidents of a test case can be costly, even when their impact on test case execution time is small, and such accidents exist even in human-produced tests [10].

Seeded dynamic symbolic execution is a variation of symbolic testing [2] that uses existing test cases as a basis for symbolic exploration [41, 42, 43, 44]. For example, the `zesti` (Zero-Effort Symbolic Test Improvement) extension of KLEE is based on the observation that symbolic-execution can take advantage of regression test cases [45]. While seeded symbolic execution is generally more scalable than non-seeded symbolic execution, it still faces the fundamental problems of path explosion and complex constraints. Seeded symbolic execution takes a test case and attempts to cover new program behavior based on it; for example, if a conditional is false in the original test case, and the symbolic engine finds the true branch to be feasible, it explores to find a test case covering the true branch. The essential quality of the seed test suite is therefore its code coverage. In general, given two test cases similar other than in length, symbolic execution has more difficulty with the longer test case: longer tests have more nearby paths to explore, and constraint complexity accumulates over an execution. Therefore, given two test cases with the same code coverage, it seems likely that symbolic exploration based on the "smaller" test case will be more effective, though it is possible that such a change *could* reduce the value of the seed test case (for example constraints may change and branch feasibility in context may change).

In order to test the hypothesis that cause reduction can improve the efficiency of seeded symbolic execution, this paper relies on reduced test suites for six C programs (five taken from the SIR repository [46]), with respect to statement coverage, and compares the additional branch coverage obtained by running KLEE `make-zesti` [45] on the reduced tests vs. the original tests [10]. In order to make the comparison fair, the runtime of cause reduction was counted against the total time budget for symbolically exploring each test case using a timeout of 20% of the budget instead of reducing to 1-minimality. Table IX shows the subject programs and data on their test pools. For these tests, the startup cost for each test case dominated individual runtimes. For most programs the difference in test suite execution time before and after reduction was negligible, less than 1/10th of

Table IX. Subject programs used in the evaluation

Subject	NBNC	LLVM instructions	# test cases	runtime (s)
Sed	13,359	48,684	370	15
Space	6,200	24,843	500	16
Grep	10,056	43,325	469	9
Gzip	5,677	21,307	214	85
Vim	107,926	339,292	1,950	65
YAFFS2	10,357	30,319	500	11

Table X. Reduction rates

Subject	Path length before reduction			Path length after reduction			Reduction rate (%)		
	Min	Median	Max	Min	Median	Max	Min	Median	Max
Sed	6	88,723	496,723	6	6,167	80,418	0	94	99
Space	530	4,154	24,147	530	3,798	18,199	0	8	80
Grep	740	103,097	622,223	691	26,466	424,388	0	75	99
Gzip	24	752,257	36,351,281	24	231,629	1,732,247	0	60	100
Vim	201,222	221,219	481,749	201,083	213,957	475,421	0	2	50
YAFFS2	32,632	53,139	91,252	23,719	40,339	71,942	2	23	50

Table XI. Branch coverage increment (mean values over 150 test suites) on 100 random tests (Before: Before test case reduction, After: test case reduction)

Subject	Time	DFS		Random path		Random state		MD2U		Combined	
		Before	After	Before	After	Before	After	Before	After	Before	After
Sed	10m	150.66	**211.73**	265.03	**274.46**	213.75	**248.56**	211.42	**249.59**	279.65	**289.00**
	20m	163.15	**229.53**	**298.83**	292.77	229.23	**267.65**	239.15	**265.14**	**311.49**	302.33
Space*	10m	3.21	**9.18**	5.00	**5.33**	3.58	**6.77**	3.58	**7.24**	5.00	**9.26**
	20m	2.24	**10.81**	4.23	**5.13**	2.75	**6.78**	2.75	**6.71**	4.23	**10.82**
Grep	10m	20.73	**160.37**	120.70	**139.35**	116.22	**183.65**	112.65	**189.71**	157.87	**212.66**
	20m	21.13	**185.68**	177.65	**201.23**	114.07	**205.58**	110.14	**209.36**	194.85	**232.85**
Gzip	10m	93.35	**103.95**	220.55	**226.33**	113.10	**129.07**	134.71	**158.12**	222.59	**228.77**
	20m	153.66	**157.25**	233.47	**236.50**	176.10	**182.41**	*193.81*	*193.89*	239.44	**242.59**
Vim	10m	*312.17*	*310.36*	111.71	**116.44**	302.77	**308.35**	357.60	**365.79**	*540.42*	*542.99*
	20m	513.45	**558.27**	118.60	**123.79**	345.81	**358.17**	421.97	**442.37**	769.95	**821.35**
YAFFS2	10m	**78.14**	76.28	98.21	**100.18**	93.40	**104.80**	93.99	**105.27**	115.47	**125.35**
	20m	78.54	**79.51**	99.15	**100.47**	94.58	**103.89**	95.09	**104.33**	117.98	**126.39**

a second; for `vim` the reduction was on the order of three seconds. However, test case runtime is not the full story; for symbolic execution, the number of branch choices made during a run is the more important cost of a test case. Table X shows how cause reduction affected path lengths (total branch choices during execution) for these programs, with a two-minute timeout. The test cases in these pools, with the exception of `YAFFS` and `space`, are human-generated tests taken from the SIR [46]. For YAFFS and `space` test cases are produced randomly, for YAFFS as described above, and for `space` the SIR tests are random.

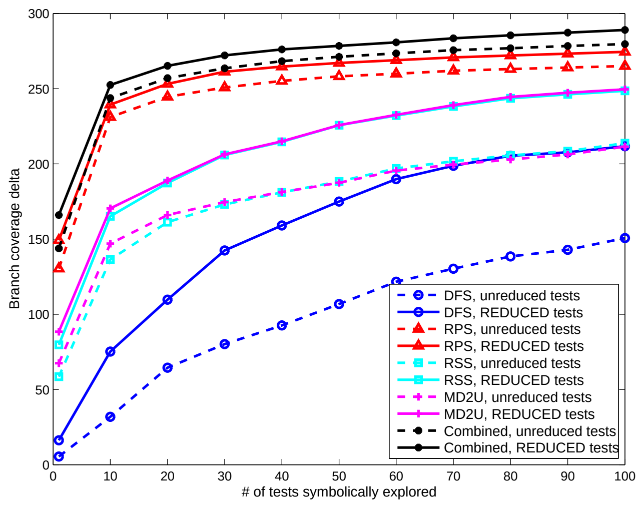

Figure 11. Additional branch coverage on `sed` program during seeded symbolic execution with unreduced tests and reduced tests (test cases are in random order)

Path length or runtime reduction itself, however, is not the goal. Table XI shows how cause reduction affected additional branches explored by symbolic execution, using four of KLEE's symbolic search strategies, the actual goal of seeded testing. In almost all cases differences were significant by a U-test, with p-value ≤ 0.02; the few exceptions are shown in italics. Figure 11 shows graphically how reduction improved coverage for the `sed` program with a 10 minute budget for symbolic exploration of each test case. In several cases, test reduction resulted in an improvement in additional branch coverage of over 100%, and in the three cases where reduction gave worse results, the penalty in lost branches was small (less than 10%). Cause reduction more than pays for itself. Additionally, the effectiveness of reduction with a timeout shows the practical implications of the majority of reduction occurring long before 1-minimality is reached. Even where reduction needs to be performed on-the-fly and included in the testing budget, reduction can be feasible.

4. CAUSE REDUCTION FOR BUILDING SPECIALIZED TEST SUITES

The previous applications of cause reduction have primarily been aimed at improving the effectiveness or efficiency of an entire test suite, for general testing purposes. It is also possible to target specific behaviors tested by a suite. For example, a developer may wish to produce a quick test that is focused only on assertions related to their own code. Given a suite of general-purpose tests, the aim is to produce what is essentially a unit test for some subset of system functionality.

Of course, cause reduction by coverage could be applied with only a single statement, or a small set of statements, as the reduction criteria, as in producing the **GE-Min** suite. However, such reduction only preserves structural coverage; while a test case reduced by this approach may *cover* an assertion, the behavior that preceded the execution and determined the success or failure of the assertion may be discarded. As a concrete example, consider the assertion on line 3,934 of `jsregexp.c` in SpiderMonkey 1.6:

```
JS_ASSERT((size_t)(ncp - nstart) == length);
```

This assertion is executed by 264 of the 13,323 test cases in the **Full** suite. Executing these tests takes just over five minutes, which is potentially far too much time if testing for multiple

*The reason branch coverage for 10 minutes is higher than for 20 minutes is that KLEE crashed in some 20 minute runs, losing some coverage data.

assertions is needed. However, the set of 264 tests is highly redundant: only 10 different values of `ncp - nstart` and `length` actually are represented in these tests. Reducing 10 tests covering these values with respect to coverage only unfortunately loses some covered values. However, this can be avoided by changing the reduction criteria to require that the candidate test (1) execute the assertion and (2) have the same values for the tuple `(ncp - nstart, length)` when the assertion executes. Performing reduction by these criteria results in 10 tests, three of which contain 5 semantic units, two of which contain 2 semantic units, and five of which contain only 1 semantic unit. It is the additional semantic units beyond the call to the `RegExp` operator in the longer tests that are lost by coverage-only reduction. Running these tests, which represent all values reaching the assertion, requires only 1.6 seconds. Moreover, these tests had the same ability to detect three hand-seeded faults in the `regexp_compile` function as the whole set of 264 tests.

Program slicing [47] is a technique that determines all statements that *might* affect the value of a variable at some point in execution; dynamic slicing [48] only returns statements that *did* affect the value in a given program execution. At first glance, dynamic slicing appears to be an alternative method to address the problem of not only preserving the execution of an assertion, but of the exact values that reach it. In fact, slicing seems to be an attractive alternative for cause reduction in general. Leitner et al. [49] applied slicing to standard delta debugging, to dramatically speed reduction for unit tests for Eiffel programs; their technique would presumably work just as well for cause reduction, including for the purpose of specialized test suites (if variable values are used in addition to reachability). However, slicing can aid delta debugging or cause reduction only when the test case itself is in the executable language that is the target of slicing. This works fine for unit tests consisting of API calls in the language of the SUT, as in the work of Leitner et al., but many testing systems generate input values for a program, and are thus outside the domain of slicing. When the test case is not in the object language, slicing can only report what portions of the SUT can be ignored, not which portions of the test case are irrelevant. Of the testing systems considered in this paper, only YAFFS2 could easily incorporate slicing-based cause reduction. In the SpiderMonkey case, while `jsfunfuzz` tests are technically written in JavaScript itself, the fact that the JavaScript for the test is a string given to the `tryItOut` function would likely frustrate most static slicing approaches. Dynamic slicing, however, could still be useful in this case (though not, as far as the authors can tell, for C compiler inputs or most of the SIR subjects). Unfortunately slicing tools are not generally as widely available or easily used as delta debugging tools, and can require capturing the entire execution trace for analysis, which is likely to be very expensive for typical SpiderMonkey test cases, and the dynamic slice may be much larger than the behavior required by preserving the reaching values with cause reduction.

5. CAUSE REDUCTION FOR QUANTITATIVE RELATIONSHIPS

Cause reduction can be used to aim for small test cases with "unreasonable" behavior. For example, given a large random test that produces unusually large peak memory usage, cause reduction can attempt to find a smaller test case with the same peak memory usage. In addition to preserving such a feature, cause reduction could be used to search for test cases that *increase* some quantifiable property of execution beyond the original test.

C++ compilers sometimes produce unreasonably lengthy error messages for invalid programs; this problem is well known enough and irritating enough that it inspired a contest (`http://tgceec.tumblr.com/`) for the shortest program producing the longest error message using version 4.8.1 of the GNU C++ compiler. The authors took four C++ files from open source projects and used C-Reduce to reduce them with an effect designed to maximize the ratio of error message length to source code length. In one case, C-Reduce eventually produced this C++ fragment:

```
struct x0 struct A<x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(_T1,x0(_T1>
  <_T1*, x0(_T1*_T2> binary_function<_T1*, _T2, x0{ }
```

The version of `g++` specified by the contest, and all subsequent versions as of June 2014, produce about 4.6 MB of error output when asked to compile this program. Furthermore, the size of the error output approximately doubles for each additional occurrence of the `(x0` substring. This result—exponential error message size even in the absence of templates (the usual cause for huge errors from C++ compilers)—is a bit surprising. A cleaned up version of C-Reduce's output was entered into the contest where it was declared to be one of seven winning entries.

6. RELATED WORK

This paper extends results originally presented in conference papers, primarily the ICST 2014 paper [11] introducing cause reduction for quick testing, as well as summarizing and providing additional details on some results from the ISSTA 2014 paper [10] on using test case reduction and prioritization in seeded symbolic execution.

This paper follows previous work on delta debugging [9, 6, 50] and other methods for reducing failing test cases. While previous work has attempted to generalize the circumstances to which delta debugging can be applied [51, 52, 53], this paper replaces preserving failure with any chosen effect. Surveying the full scope of the work on failure reduction in both testing [8, 54] and model checking [55, 56] is beyond the scope of this paper. The most relevant work considers delta debugging in random testing [7, 12, 13, 49], which tends to produce complex, essentially unreadable, failing test cases [7]. Random test cases are also highly redundant, and the typical reduction for random test cases in the literature ranges from 75% to well over an order of magnitude [7, 49, 12, 8, 23]. Reducing highly-redundant test cases to enable debugging is an essential enough component of random testing that some form of automated reduction seems to have been applied even before the publication of the *ddmin* algorithm, e.g. in McKeeman's early work [57], and reduction for compiler testing is an active research area [8]. Recent work has shown that reduction has other uses: Chen et. al showed that reduction was required for using machine learning to rank failing test cases to help users sort out different underlying faults in a large set of failures [23].

In a larger sense, all work on causality [58] in testing and debugging is relevant to this paper's approach, which is explicitly focused on a notion of a cause (the test case) inducing effects [59]. The notion of causes in testing can focus on inputs, code, or program state (see Chapter 12 of Zeller's book on debugging [59]); this paper locates causes in inputs, though these inputs can be test code. The validity of Zeller's claim that "among all scientific disciplines, *debugging is the one that can best claim to deal with actual causality*" is less important in this application, as there is no expectation that any human will attempt to verify the relationship of an input to a particular coverage target: the set of effects is too large for human conceptualization.

Second, this paper proposes an orthogonal approach to test suite minimization, selection and prioritization from that taken in previous work, which is covered at length in a survey by Yoo and Harman [3]. Namely, while other approaches have focused on minimization [60, 61, 62, 35], selection [24] and prioritization [36, 63, 64] at the granularity of entire test suites, this paper proposes reducing the size of the test cases composing the suite, a "finer-grained" approach that can be combined with previous approaches. Previous work on suite minimization has shown a tendency of minimization techniques to lose fault detection effectiveness [65]. While the experiments in this paper are not intended to directly compare cause reduction and suite-level techniques, it is true that for SpiderMonkey, at the 30 second and 5 minute levels, fault detection was much better preserved by the proposed approach than by prioritizations based on suite minimization techniques.

The idea of a quick test proposed here also follows on work considering not just the effectiveness of a test suite, but its *efficiency*: coverage/fault detection per unit time [19, 20]. Finally, as an alternative to minimizing or prioritizing a test suite, tests can be constructed with brevity as a criteria, as in evolutionary testing and bounded exhaustive testing [27, 28, 29, 30]. However, the applications where random testing is most used tend to be precisely those where "small by construction" methods have not been shown to be as successful, possibly for combinatorial reasons.

Seeded dynamic symbolic execution takes an initial test case and tries to cover a branch that test has not covered. This initial test case is called the seed. In the literature, the seed is usually chosen arbitrarily from a pool of test cases, which are not modified in the process [2, 66, 67, 68]. The most relevant literature on symbolic execution for this paper is the ISSTA 2014 paper extended [10]; that approach makes use of the `zesti` variation of the KLEE tool [45].

Cause reduction can be viewed as an imprecise, non-unique, kind of "dynamic slice" [48] with respect to an arbitrary property of execution, but slicing a test case rather than a program. The underlying approach, however, is radically different than any program slicing method [47, 69]. The closest related work is that using slicing methods to improve fault localization and error explanation in model checking [70], as it was also inspired by delta debugging and a causal view [56, 71] of model checking counterexamples (which are essentially test cases).

7. CONCLUSIONS AND FUTURE WORK

This paper shows that generalizing the idea of delta debugging from an algorithm to reduce the size of failing test cases to an algorithm to reduce the size of test cases with respect to *any* interesting effect, called *cause reduction*, is a useful concept. This paper provides detailed data on the application of cause reduction to the *quick test* problem, which proposes generating highly efficient test suites based on inefficient randomly generated tests. Reducing a test case with respect to statement coverage not only (obviously) preserves statement and function coverage; it also approximately preserves branch coverage, test failure, fault detection, and mutation killing ability, for two realistic case studies (and a small number of test cases for C compilers). Combining cause reduction by statement coverage with test case prioritization by additional statement coverage produced, across 30 second and five minute test budgets and multiple versions of the SpiderMonkey JavaScript engine, an effective quick test, with better fault detection and coverage than performing new random tests or prioritizing a previously produced random test suite. The efficiency and effectiveness of reduced tests persists across versions of SpiderMonkey and the GCC compiler that are up to a year later in development time, a long period for such actively developed projects. Cause reduction also has other potential applications, discussed in brief in this paper; in all cases, the central idea is that a test causes some desired behavior, not necessarily meaning program failure. Reducing the size of the cause while preserving the effect has many benefits, such as decreased execution time.

As future work, the authors hope to explore other uses of cause reduction. For example, reduction could be applied to a program itself, rather than a test. A set of tests (or model checking runs) could be used as an effect, reducing the program with respect to its ability to pass tests/satisfy properties. If the program can be significantly reduced, it may suggest a weak suite or oracle, and identify code that is under-specified, rather than just not executed.

Delta debugging is widely used and appreciated because it applies the insight that short test cases are usually more useful than long test cases to the special case of failing test cases. Cause reduction takes that insight and generalizes it to the wide variety of other "purposes" for test cases seen in both research and practice: code coverage, dynamic behavior, "stress properties," and so forth.

Acknowledgments: The authors would like to thank the anonymous reviewers, Andreas Zeller, Weng-Keen Wong, Xiaoli Fern, and Jamie Andrews. A portion of this work was funded by NSF grants CCF-1217824 and CCF-1054786.

REFERENCES

1. B. Beizer, *Software Testing Techniques*. International Thomson Computer Press, 1990.
2. P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Programming Language Design and Implementation*, 2005, pp. 213–223.
3. S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.
4. RTCA Special Committee 167, "Software considerations in airborne systems and equipment certification," RTCA, Inc., Tech. Rep. DO-1789B, 1992.
5. J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Automated Software Engineering*, 2005, pp. 273–282.
6. R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *International Symposium on Software Testing and Analysis*, 2000, pp. 135–145.

7. Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *International Symposium on Software Reliability Engineering*, 2005, pp. 267–276.
8. J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Conference on Programming Language Design and Implementation*, 2012, pp. 335–346.
9. A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 183–200, 2002.
10. C. Zhang, A. Groce, and M. A. Alipour, "Using test case reduction and prioritization to improve symbolic execution," in *International Symposium on Software Testing and Analysis*, 2014, pp. 160–170.
11. A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction for quick testing," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 243–252.
12. A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *International Conference on Software Engineering*, 2007, pp. 621–631.
13. A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu, "Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning," *Annals of Mathematics and Artificial Intelligence*, vol. 70, no. 4, pp. 315–349, 2014.
14. K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *ICFP*, 2000, pp. 268–279.
15. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.
16. A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *International Symposium on Software Testing and Analysis*, 2012, pp. 78–88.
17. X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.
18. J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu, "Random test run length and effectiveness," in *Automated Software Engineering*, 2008, pp. 19–28.
19. A. Gupta and P. Jalote, "An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing," *Journal of Software Tools for Technology Transfer*, vol. 10, no. 2, pp. 145–160, 2008.
20. M. Harder, J. Mellen, and M. D. Ernst, "Improving test suites via operational abstraction," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 60–71.
21. R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *International Conference on Software Engineering*, 2014, pp. 72–82.
22. M. Gligoric, A. Groce, C. Zhang, R. Sharma, A. Alipour, and D. Marinov, "Guidelines for coverage-based comparisons of non-adequate test suites," *ACM Transactions on Software Engineering and Methodology*, accepted for publication.
23. Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013, pp. 197–208.
24. J. Bible, G. Rothermel, and D. S. Rosenblum, "A comparative study of coarse- and fine-grained safe regression test-selection techniques," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 149–183, 2001.
25. A. Groce, K. Havelund, and M. Smith, "From scripts to specifications: The evolution of a flight software testing effort," in *International Conference on Software Engineering*, 2010, pp. 129–138.
26. Android Developers Blog, "UI/application exerciser monkey," http://developer.android.com/tools/help/monkey.html.
27. L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: an evolutionary test approach for Java," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2010, pp. 185–194.
28. G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 416–419.
29. J. H. Andrews, T. Menzies, and F. C. Li, "Genetic algorithms for randomized unit testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 1, pp. 80–94, 2011.
30. J. Andrews, Y. R. Zhang, and A. Groce, "Comparing automated unit testing strategies," Department of Computer Science, University of Western Ontario, Tech. Rep. 736, December 2010.
31. J. Ruderman, "Introducing jsfunfuzz," 2007, http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.
32. M. Gligoric, A. Groce, C. Zhang, R. Sharma, A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *International Symposium on Software Testing and Analysis*, 2013, pp. 302–313.
33. A. Groce, A. Fern, J. Pinto, T. Bauer, A. Alipour, M. Erwig, and C. Lopez, "Lightweight automated testing with adaptation-based programming," in *IEEE International Symposium on Software Reliability Engineering*, 2012, pp. 161–170.
34. J. Ruderman, "Mozilla bug 349611," https://bugzilla.mozilla.org/show_bug.cgi?id=349611 (A meta-bug containing all bugs found using jsfunfuzz.).
35. T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Inf. Process. Lett.*, vol. 60, no. 3, pp. 135–141, 1996.
36. G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.
37. "Yaffs: A flash file system for embedded use," http://www.yaffs.net/.
38. J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *International Conference on Software Engineering*, 2005, pp. 402–411.
39. L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *International Conference on Software Engineering*, 2010, pp. 435–444.
40. A. Groce, M. A. Alipour, and R. Gopinath, "Coverage and its discontents," in *Onward! Essays*, 2014, pp. 255–268.
41. Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software*

Engineering, ser. FSE '10, 2010, pp. 257–266.
42. Z. Xu, Y. Kim, M. Kim, and G. Rothermel, "A hybrid directed test suite augmentation technique," in *ISSRE*, 2011, pp. 150–159.
43. Y. Kim, M. Kim, Y. Kim, and Y. Jang, "Industrial application of concolic testing approach: A case study on libexif by using crest-bv and klee," in *International Conference on Software Engineering*, 2012, pp. 1143–1152.
44. Y. Kim, Z. Xu, M. Kim, M. B. Cohen, and G. Rothermel, "Hybrid directed test suite augmentation: An interleaving framework," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 263–272.
45. P. D. Marinescu and C. Cadar, "make test-zesti: a symbolic execution solution for improving regression testing," in *International Conference on Software Engineering*, 2012, pp. 716–726.
46. H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Engg.*, vol. 10, pp. 405–435, 2005.
47. M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/358557.358577
48. H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Programming Language Design and Implementation*, 1990, pp. 246–256.
49. A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *International Conference on Automated Software Engineering*, 2007, pp. 417–420.
50. A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *ESEC / SIGSOFT Foundations of Software Engineering*, 1999, pp. 253–267.
51. J. Choi and A. Zeller, "Isolating failure-inducing thread schedules," in *International Symposium on Software Testing and Analysis*, 2002, pp. 210–220.
52. H. Cleve and A. Zeller, "Locating causes of program failures," in *ICSE*, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM, 2005, pp. 342–351.
53. A. Zeller, "Isolating cause-effect chains from computer programs," in *Foundations of Software Engineering*, 2002, pp. 1–10.
54. G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *International Conference on Software engineering*, 2006, pp. 142–151.
55. P. Gastin, P. Moro, and M. Zeitoun, "Minimization of counterexamples in SPIN," in *In SPIN Workshop on Model Checking of Software*. Springer-Verlag, 2004, pp. 92–108.
56. A. Groce and D. Kroening, "Making the most of BMC counterexamples," *Electron. Notes Theor. Comput. Sci.*, vol. 119, no. 2, pp. 67–81, Mar. 2005.
57. W. McKeeman, "Differential testing for software," *Digital Technical Journal of Digital Equipment Corporation*, vol. 10(1), pp. 100–107, 1998.
58. D. Lewis, "Causation," *Journal of Philosophy*, vol. 70, pp. 556–567, 1973.
59. A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.
60. S. McMaster and A. M. Memon, "Call-stack coverage for GUI test suite reduction," *Software Engineering, IEEE Transactions on*, vol. 34, no. 1, pp. 99–115, 2008.
61. A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *In Proc. Twelfth Int'l. Conf. Testing Computer Softw*, 1995.
62. H.-Y. Hsu and A. Orso, "Mints: A general framework and tool for supporting test-suite minimization," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 419–429.
63. K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 2006 international symposium on Software testing and analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1146238.1146240
64. S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004.
65. G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Journal of Software Testing, Verification, and Reliability*, vol. 12, pp. 219–249, 2007.
66. K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Foundations of Software Engineering*, 2005, pp. 262–272.
67. P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for c/c++ using concolic execution," in *International Conference on Software Engineering*, 2013, pp. 132–141.
68. R. Majumdar and K. Sen, "Hybrid concolic testing," in *International Conference on Software Engineering*, 2007, pp. 416–426.
69. F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, pp. 121–189, 1995.
70. A. Groce, "Error explanation with distance metrics," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004, pp. 108–122.
71. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. J. Trefler, "Explaining counterexamples using causality," *Formal Methods in System Design*, vol. 40, no. 1, pp. 20–40, 2012.