Cause Reduction: Minimizing Test Cases with Respect to Code Coverage

Alex Groce,* Amin Alipour,* Chaoqiang Zhang,* Yang Chen,† and John Regehr,†
*School of Electrical Engineering and Computer Science
Oregon State University {grocea,zhangch,alipourm}@onid.orst.edu
†School of Computing
University of Utah
{chenyang,regehr}@cs.utah.edu

***Abstract*—Delta debugging is a critical support for effective (random) testing. As the name "delta debugging" suggests, test case reduction has thus far been applied only to failing test cases for human consumption (or fast regression). However, test case minimization with respect to failure is a special case of the notion of *cause reduction*, where a test case is minimized with respect to any important *effect*. Such reduction for both failing and passing random test cases, with respect to code coverage, can produce suites with *better* properties (e.g. lower runtime and higher efficiency) than the original suite. While the cost of coverage-based minimization is high, minimized test cases can remain effective over modified code for a significant period, justifying the effort and enabling faster testing. The value of coverage-based minimization is shown by using it to solve an important problem in random testing, evaluated using versions of Mozilla's SpiderMonkey JavaScript engine.**

I. Introduction

A. An Anecdote and Problem Statement: Using Previous Random Testing to Generate an Efficient "Quick Test"

In testing a flash file system implementation that eventually evolved into the basis of the file system for the Curiosity Mars Rover [1], [2], one of the authors of this paper discovered that, while an overnight sequence of random tests was extremely effective in shaking out even subtle faults, random testing was not very effective if only a short time 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, the developer found it much more effective to run a regression suite built by applying delta debugging [3] to a representative test case for each fault ever found by the random testing system. In addition to detecting actual regressions (when old bugs re-appeared), these tests obtained good code coverage in less than a minute, while running new random tests typically required many hours to match the same statement coverage level. Unfortunately, the delta debugging-based regression was only modestly effective at detecting *new* faults introduced into the file system unrelated to previous bugs. Inspecting the minimized test cases revealed that, while the tests covered most statements, the tests were extremely focused on strange corner cases that had triggered failures, and sometimes missed even extremely shallow bugs easily detected by a short amount of more traditional random testing. While the bug-based regression suite was effective as a *regression* suite, it was only somewhat effective as a "quick test" — an efficient replacement for running new random tests for a short time period.

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 [4] in Haskell development. 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 [5], [1], swarm testing [6], or other improvements to standard random testing, it is extremely hard to randomly generate effective random tests for complex systems software such as compilers [7] and file systems [1], [2] without a large test budget. Among other factors, a key limitation is that even highly tuned random testers show increasing fault detection with larger tests, which limits the number of tests that can be run in a small budget [8], [7]. The value of the delta debugged regressions at NASA, however, suggests a more tractable problem: *given* the results of previous random testing, generate a truly *quick test* for complex systems software. Discussion with developers and the authors' experience suggested that an ideal quick test maximizes the value of test budgets ranging from 30 seconds to 5 minutes. If more time than 5 minutes is available, it is often reasonable to run overnight tests, in which case simply generating new random tests works.

The *quick test* problem is therefore: given a set of randomly generated tests, produce test suites for test budgets ranging from 30 seconds to 5 minutes that maximize:

1) Code coverage: the most important coverage criterion is probably statement coverage; a quick test that does not execute a statement cannot detect even the least subtle defects (e.g., code that always causes a crash). Branch and function coverage are also clearly desirable;
2) Failures: automatic fault localization techniques [9] often work best in the presence of multiple failing test cases, so more failures is generally a positive attribute of a test suite; more failures also indicate a higher probability of finding a flaw;
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 4 failures, each of which exhibits a distinct fault, than to produce 50 failures that exhibit only 2 different faults [10].

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 even 10 minutes of analysis before each 30 second run is clearly unacceptable; performing 10 hours of analysis once to produce quick test suites that remain useful for a period of months is a very reasonable solution. For quick test purposes, it 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 [11], [12]; the nature of random tests, where tests are all statistically similar (as opposed to human-produced tests which tend to have some particular testing goal) means that in practice selection methods tend to propose running most stored test cases. In addition, some of the most important subjects of aggressive random testing, i.e., compilers and interpreters, tend to pose a difficult problem for change analysis, since optimization passes rely on deep semantic properties of the test case.

B. Delta Debugging

One of the most important developments in the last fifteen years in testing and debugging is a generalized algorithm for reducing the size of failing test cases, widely known as delta debugging or delta-minimization (*ddmin*) [3]. While a variety of improvements to the basic algorithm have been proposed, delta debugging algorithms have retained a common core since the original proposal of Hildebrandt and Zeller [13]: use a variation on binary search to remove individual components[1] of a failing test case t to produce a new test case t_{min} satisfying two properties: (1) t_{min} fails and (2) removing any component from t_{min} results in a test case that does not fail. Such a test case is called *1-minimal*. Because 1-minimal test cases are potentially much larger than the smallest possible set of failing components (and *ddmin* is not guaranteed to find even the smallest 1-minimal test case), we say that *ddmin reduces* the size of a test case, rather than truly minimizing it.

While *ddmin* and similar algorithms have been widely used to reduce test cases produced by humans (either in the process of testing or based on failures reported by users in the field), their most valuable application has perhaps been in random testing, which tends to produce extremely complex, essentially unreadable, failing test cases [14]. Fortunately, random test cases are also typically highly redundant, and the typical reduction for random test cases in the literature ranges from 75% to well over an order of magnitude [14], [15], [1], [16], [10]. Reducing highly-redundant test cases to enable humans to debug a failure is an essential enough component of industrial-strength random testing that some form of automated reduction seems to have been widely applied even before the publication of the *ddmin* algorithm, e.g. in McKeeman's seminal work on differential testing of compilers [17], and better reduction methods for important random testing applications such as compiler testing can be published in top-rank programming language conferences [16]. Recent work has shown that reduction is also critical for automated uses of random testing that do not involve a human in the loop: Chen et. al showed that reduction was required for ranking failing test cases to help users sort out different underlying faults in a large set of randomly produced failures [10], and, as noted above, Groce et al. discovered that a set of 1-minimal failing test cases worked as an efficient regression suite during file system development [2].

The core value of delta debugging across all of these applications can be understood in a simple proposition: given two test cases that achieve the same purpose (in these cases, exhibiting a failure of a software system), the smaller of the two test cases will be more useful for a large variety of purposes, all other things being equal. The smaller test case will typically be easier to understand, have fewer irrelevant features to confuse a machine-learning algorithm (reduction is in this sense a kind of feature selection), and execute more quickly. In other words, delta debugging is valuable because, *given two test cases that both serve the same purpose*, we almost always prefer the smaller of the two test cases.

C. Generalizing Delta Debugging

The definition of delta-debugging is open to many useful generalizations. In fact, Zeller's introduction of the delta-debugging concept [18] is based not on test case components but on changes to the source code itself. Choi and Zeller [19] used a *ddmin*-like algorithm to find failure-inducing thread schedule choices by isolating which changes in scheduling cause a failing test case to succeed (a variation of the basic *ddmin* algorithm also used for other purposes). Cleve and Zeller [20] proposed an algorithm to entire program states to localize faults by "delta-debugging" the contents of memory. Groce and Visser [21] search for minimal transformations between model checking runs, combining test case component and thread-interleaving.

What all of these generalizations have in common is that they generalize the notion of a component or circumstance. The choice of "test fails" as the property to be preserved, however, is taken for granted in all variations of *ddmin* of which we are aware. The assumption has always been that delta debugging is, inherently, a *debugging* algorithm, and is only useful for failures exposing a bug. However, the best way to understand *ddmin*-like algorithms is that they attempt to reduce the size of a *cause* (e.g. a test case, a thread schedule, a model-checking path) while ensuring that it still causes some fixed *effect* (in *ddmin*, the effect is always failure): *ddmin* is a special-case of *cause reduction*[2].

[1]Hildebrandt and Zeller use the term "circumstance" instead of "component" to indicate that the algorithm is not sensitive to the precise nature of the changes defined on a test.

[2]The authors would like to thank Andreas Zeller for suggesting the term "cause reduction."

D. Contributions

The first contribution of this paper is to show that cause reduction is a *useful* generalization of delta debugging. In particular, generalizing the *effect* in *ddmin* from failure to *code coverage* properties makes it possible to apply *ddmin* to improve test suites containing both successful and failing test cases, by reducing their runtime while retaining their code coverage. Retaining code coverage turns out to approximate retaining other important test effects, including fault detection. A case study based on testing Mozilla's SpiderMonkey JavaScript engine uses real faults to show that cause reduction with respect to statement coverage is *more* effective in improving the suite efficiency than test case prioritization based on coverage, 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 possibly detects *more* faults. The second, highly practical, contribution of this paper is therefore *a proposed method for solving the quick test problem posed above*, based on cause reduction (and simple coverage-based test case prioritization). A mutation-based analysis of the YAFFS2 flash file system shows that the effectiveness of cause reduction is not unique to SpiderMonkey's particular faults and/or the chosen test suite: a statement-coverage reduced suite for YAFFS2 ran in a little over half the time of the original suite, but killed over 99% as many mutants, including 6 mutants not killed by the original suite.

II. Coverage-Based Test Case Reduction

A. Cause Reduction

The definitions provided in the core delta debugging paper [3] 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 [3] with a function *reffect* such that *reffect* defines "failure" of a run as preserving an arbitrary effect that holds for the original test case and "success" as not preserving that effect. An actual failure is a particular instance of an effect to be preserved. We call this "new" algorithm *cause reduction*, but, of course, it is almost exactly the same as the original *ddmin* algorithm, and most optimizations or variations applied to *ddmin* can be applied to cause reduction as well[3].

The most interesting consequence of this minor change is that *ddmin* is no longer defined only for failing test cases. If the effect chosen is well-defined for successful test cases, then *ddmin* can be applied to reduce the cause (the test case) in that case also. The question is whether there are any interesting effects that are defined for all test cases and important enough to inspire reduction efforts.

[3]Static analysis based on a slice of a visible failure [15] is an obvious partial exception, depending on the effect in question.

B. Coverage as an Effect

In fact, a large portion of the literature on software testing is devoted to 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 [22], [23], [24], [25]. Seeking high coverage is a goal of many testing methods, as high coverage is expected to correlate well with effective fault detection. Furthermore, producing *small* test suites with high code coverage [26] 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 [11]. 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 the benefit of reduction at the suite level: more efficient testing, in terms of fault detection and 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 amenable to reduction.

C. Choosing a Coverage Criterion for Reduction

In principle, *any* coverage criteria could be used as effects for reduction. In practice, it is highly unlikely that reducing even random tests by extremely fine-grained coverages such as path or predicate coverages [25] would produce much reduction. Moreover, *ddmin* is a very expensive algorithm to run when test cases do not reduce much, since every small reduction produces a new attempt to establish 1-minimality, and repeated small removals tend to result in a massive computational effort to achieve minimal reduction. Moreover, for purposes of a quick test, it seems most important to concentrate on highly efficient coverage of coarse entities, such as statements. Finally, only branch and statement 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 in particular, 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 in the real world, which is one reason random testing is more widely used than concolic testing or software model checking [27]. We therefore decided to investigate cause reduction by statement and branch coverage only.

III. SpiderMonkey JavaScript Engine Case Study

SpiderMonkey is the JavaScript Engine for Mozilla, an extremely widely used, security-critical interpreter/JIT compiler. SpiderMonkey has been the target of aggressive random testing for many years now. A single fuzzing tool, `jsfunfuzz` [28], is responsible for identifying more than 1,700 previously

TABLE I
SPIDERMONKEY UNLIMITED TEST BUDGET RESULTS

Release	Date	Suite	#Tests	Time(s)	ST	BR	FN	Failures	Faults (est.)
1.6	12/22/2006	Full	13,323	14,255.068	19,091	**14,567**	966	1,631	22
1.6	12/22/2006	ST-Min	13,323	**3,566.975**	19,091	14,562	966	1,631	**43**
1.6	12/22/2006	DD-Min	1,019	169.594	16,020	10,875	886	1,019	22
1.6	12/22/2006	GE-ST(Full)	168	182.823	19,091	14,135	966	14	5
1.6	12/22/2006	GE-ST(ST-Min)	171	47.738	19,091	14,099	966	14	8
NR	2/24/2007	Full	13,323	9,813.781	**22,392**	**17,725**	**1,072**	**8,319**	20
NR	2/24/2007	ST-Min	13,323	**3,108.798**	22,340	17,635	1,070	4,147	**36**
NR	2/24/2007	DD-Min	1,019	148.402	17,923	12,847	958	166	7
NR	2/24/2007	GE-ST(Full)	168	118.232	21,305	16,234	1,044	116	5
NR	2/24/2007	GE-ST(ST-Min)	171	40.597	21,323	16,257	1,045	64	3
NR	4/24/2007	Full	13,323	16,493.004	**22,556**	**18,047**	**1,074**	189	**10**
NR	4/24/2007	ST-Min	13,323	**3,630.917**	22,427	17,830	1,070	**196**	6
NR	4/24/2007	DD-Min	1,019	150.904	18,032	12,979	961	158	5
NR	4/24/2007	GE-ST(Full)	168	206.033	22,078	17,203	1,064	4	1
NR	4/24/2007	GE-ST(ST-Min)	171	45.278	21,792	16,807	1,058	3	1
1.7	10/19/2007	Full	13,323	14,282.776	**22,426**	**18,130**	**1,071**	**528**	**15**
1.7	10/19/2007	ST-Min	13,323	**3,401.261**	22,315	17,931	1,067	274	10
1.7	10/19/2007	DD-Min	1,019	168.777	18,018	13,151	956	231	12
1.7	10/19/2007	GE-ST(Full)	168	178.313	22,001	17,348	1,061	6	2
1.7	10/19/2007	GE-ST(ST-Min)	171	43.767	21,722	16,924	1,055	5	2
1.8.5	3/31/2011	Full	13,323	4,301.674	**21,030**	**15,854**	**1,383**	**11**	**2**
1.8.5	3/31/2011	ST-Min	13,323	**2,307.498**	20,821	15,582	1,363	3	1
1.8.5	3/31/2011	DD-Min	1,019	152.169	16,710	11,266	1,202	2	1
1.8.5	3/31/2011	GE-ST(Full)	168	51.611	20,233	14,793	1,338	1	1
1.8.5	3/31/2011	GE-ST(ST-Min)	171	28.316	19,839	14,330	1,327	1	1

unknown bugs in SpiderMonkey [29]. SpiderMonkey is also a frequently modified, actively developed project, with over 6,000 code commits during the period from January 2006 to September of 2011 (averaging almost 4 commits each day). SpiderMonkey is thus ideal for evaluating a quick test approach. All of our SpiderMonkey testing uses the last public release of the `jsfunfuzz` tool, modified for swarm testing [6][4].

The baseline test suite for SpiderMonkey is a set of 13,323 random tests, produced during 4 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[5]. A reduced suite with equivalent statement coverage, referred to as **ST-Min** (STatement coverage Minimized) was produced by performing cause reduction on every test in **Full**, with the effect:

$$\textit{reffect}(t_c, t_b) = \begin{cases} \text{iff } \forall s \in sc(t_b).s \in sc(t_c) & \text{FAIL} \\ \text{else} & \text{SUCCESS} \end{cases}$$

where t is the current candidate reduction, t_b is the original test case from the **Full** suite, and $sc(t)$ is the set of all statements executed by t. Note that this allows a test case to be minimized to a test with *better* statement coverage than the original test; in practice, the minimization algorithm occasionally produces such tests *during* minimization, but the final test case had equivalent statement coverage to the original test in 99% of all cases (because *ddmin* tends to eventually remove the unnecessary extra coverage). The granularity of minimization was based on the semantic units produced by `jsfunfuzz`, with 1,000 such units in each test in **Full**. Each unit is roughly equivalent to one line of JavaScript code. After reduction, the average test case size was just over 122 semantic units, a bit less than an order of magnitude reduction. The computational cost of cause reduction was, on contemporary hardware, similar to the costs of traditional delta debugging reported in older papers, around 20 minutes per test case [14]. The entire process completed in less than 4 hours on a modestly sized heterogeneous cluster (using fewer than 1,000 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, but that BR-based minimization was (1) much slower and (2) only reduced each individual test's size by a factor of 2/3, vs. nearly 10x reduction for ST.

A third suite, referred to as **DD-Min** (Delta Debugging Minimized), 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

[4]Past experience shows that using swarm testing roughly doubles the effectiveness of `jsfunfuzz` in terms of failures and distinct faults.

[5]It is actually slightly better, as test generation is slightly slower than replay; for test periods of 5 minutes, however, the difference is insignificant.

TABLE II
SPIDERMONKEY 30S TEST BUDGET RESULTS (AVERAGED)

Release	Date	Suite	#Tests	ST	BR	FN	Failures	Faults (est.)
1.6	12/22/2006	Full	29.2	17,620.4	12,780.1	920.1	3.3	2.1
1.6	12/22/2006	Full ΔST	28.7	18,306.0	13,088.0	951.0	4.0	3.0
1.6	12/22/2006	Full |ST|	27.0	17,344.7	12,460.0	910.0	1.0	1.0
1.6	12/22/2006	Full ΔBR	28.0	18,153.3	13,219.0	938.0	3.0	2.0
1.6	12/22/2006	Full |BR|	27.0	17,383.0	12,538.3	911.0	0.0	0.0
1.6	12/22/2006	ST-Min	**107.0**	18,129.7	13,406.1	934.7	**13.7**	6.1
1.6	12/22/2006	ST-Min ΔST	104.7	**18,980.0**	13,870.3	**963.0**	12.0	**8.0**
1.6	12/22/2006	ST-Min |ST|	96.0	17,976.0	13,134.0	935.0	1.0	1.0
1.6	12/22/2006	ST-Min ΔBR	99.3	18,842.0	**14,063.3**	961.0	8.0	5.0
1.6	12/22/2006	ST-Min |BR|	93.0	17,919.0	13121.7	932.0	1.0	1.0
NR	2/24/2007	Full	40.7	20,722.2	15,797.8	1026.2	26.0	2.2
NR	2/24/2007	Full ΔST	42.3	19,958.0	14,814.0	1006.0	32.3	2.0
NR	2/24/2007	Full |ST|	41.0	20,110.0	15,057.0	1010.0	28.0	1.0
NR	2/24/2007	Full ΔBR	38.0	20,512.0	15,519.0	1022.0	22.0	3.0
NR	2/24/2007	Full |BR|	37.7	20,203.0	15,164.0	1014.0	24.0	1.0
NR	2/24/2007	ST-Min	108.3	21,170.5	**16,292.6**	1034.4	32.3	2.8
NR	2/24/2007	ST-Min ΔST	**111.0**	**21,364.7**	16,146.3	**1051.0**	**42.7**	2.0
NR	2/24/2007	ST-Min |ST|	97.0	20,715.0	15,724.0	1034.0	29.7	1.0
NR	2/24/2007	ST-Min ΔBR	101.7	21,113.0	16,108.0	1042.0	41.7	**4.0**
NR	2/24/2007	ST-Min |BR|	98.0	20,747.0	15,768.0	1033.0	32.0	1.0
NR	4/24/2007	Full	24.9	20,989.5	16,176.6	1,031.6	0.5	0.5
NR	4/24/2007	Full ΔST	25.3	21,111.7	15,949.0	1,040.3	2.0	**1.0**
NR	4/24/2007	Full |ST|	24.0	20,579.0	15,685.7	1,023.3	2.0	**1.0**
NR	4/24/2007	Full ΔBR	26.0	21,107.0	16,126.3	1,038.0	2.0	**1.0**
NR	4/24/2007	Full |BR|	24.0	20,591.0	15,674.0	1,022.0	0.0	0.0
NR	4/24/2007	ST-Min	**114.0**	21,302.3	16,505.5	1,040.7	2.0	**1.0**
NR	4/24/2007	ST-Min ΔST	112.7	**21,730.0**	16,633.0	**1,057.0**	2.0	**1.0**
NR	4/24/2007	ST-Min |ST|	104.7	20,920.0	16,050.3	1,037.0	2.0	**1.0**
NR	4/24/2007	ST-Min ΔBR	106.7	21,595.0	**16,770.7**	1,056.7	**3.0**	**1.0**
NR	4/24/2007	ST-Min |BR|	96.7	20,908.0	16,057.7	1,034.0	2.0	**1.0**
1.7	10/19/2007	Full	28.9	20,901.0	16,337.7	1028.3	1.1	1.0
1.7	10/19/2007	Full ΔST	30.0	21,113.0	16,141.0	1,042.0	**4.0**	**2.0**
1.7	10/19/2007	Full |ST|	29.0	20,594.0	15,835.3	1,025.0	3.0	1.0
1.7	10/19/2007	Full ΔBR	29.0	21,044.0	16,268.7	1,036.0	2.0	1.0
1.7	10/19/2007	Full |BR|	28.0	20,564.0	15,849.0	1,020.0	1.0	1.0
1.7	10/19/2007	ST-Min	**115.6**	21,278.8	16,681.2	1038.5	2.4	1.2
1.7	10/19/2007	ST-Min ΔST	114.0	**21,661.7**	16,748.3	**1,054.0**	**4.0**	**2.0**
1.7	10/19/2007	ST-Min |ST|	106.7	20,811.0	16,133.3	1,035.3	2.7	1.7
1.7	10/19/2007	ST-Min ΔBR	108.3	21,536.0	**16,906.0**	**1,054.0**	**4.0**	**2.0**
1.7	10/19/2007	ST-Min |BR|	101.3	20,792.0	16,137.0	1,032.0	2.0	1.0
1.8.5	3/31/2011	Full	91.9	**20,047.1**	**14,735.4**	1,322.9	0.0	0.0
1.8.5	3/31/2011	Full ΔST	97.0	19,889.7	14,429.7	1,321.0	**1.0**	**1.0**
1.8.5	3/31/2011	Full |ST|	93.7	19,680.3	14,263.0	1,312.0	0.0	0.0
1.8.5	3/31/2011	Full ΔBR	94.0	19,953.7	14,482.0	1,326.0	**1.0**	**1.0**
1.8.5	3/31/2011	Full |BR|	92.7	19,683.7	14,293.0	1,312.3	0.0	0.0
1.8.5	3/31/2011	ST-Min	132.5	19,630.4	14,293.4	1,311.9	0.0	0.0
1.8.5	3/31/2011	ST-Min ΔST	165.0	19,827.7	14,303.3	1,324.3	**1.0**	**1.0**
1.8.5	3/31/2011	ST-Min |ST|	158.3	19,380.0	14,003.0	1,304.0	0.0	0.0
1.8.5	3/31/2011	ST-Min ΔBR	**170.0**	19,968.7	14,445.7	**1,329.0**	**1.0**	**1.0**
1.8.5	3/31/2011	ST-Min |BR|	160.0	19,459.0	14,081.0	1,307.0	0.0	0.0

TABLE III
SPIDERMONKEY 5M TEST BUDGET RESULTS (AVERAGED)

Release	Date	Suite	#Tests	ST	BR	FN	Failures	Faults (est.)
1.6	12/22/2006	Full	279.1	18,508.0	13,854.5	949.0	31.0	6.0
1.6	12/22/2006	Full ΔST	273.7	**19,091.0**	14,198.0	**966.0**	23.7	7.0
1.6	12/22/2006	Full $\lvert$ST$\rvert$	265.3	18,303.7	13,600.0	947.0	6.0	4.0
1.6	12/22/2006	Full ΔBR	273.0	19,063.0	14,505.7	962.0	24.0	8.0
1.6	12/22/2006	Full $\lvert$BR$\rvert$	264.3	18,297.0	13,599.3	945.0	3.0	2.0
1.6	12/22/2006	ST-Min	1020.4	18,792.7	14,209.3	958.3	124.3	17.6
1.6	12/22/2006	ST-Min ΔST	**1,089.7**	**19,091.0**	14,295.0	**966.0**	138.3	**22.0**
1.6	12/22/2006	ST-Min $\lvert$ST$\rvert$	940.7	18,644.3	13,961.0	957.0	21.3	5.3
1.6	12/22/2006	ST-Min ΔBR	1,063.3	19,089.0	**14,561.7**	964.0	**141.3**	19.3
1.6	12/22/2006	ST-Min $\lvert$BR$\rvert$	959.3	18,647.0	13,976.0	957.0	26.7	9.0
NR	2/24/2007	Full	405.2	21,879.1	17,116.9	1,058.0	250.4	6.8
NR	2/24/2007	Full ΔST	404.3	21,560.0	16,614.0	1,052.0	258.0	6.0
NR	2/24/2007	Full $\lvert$ST$\rvert$	379.0	21,410.7	16,572.7	1,046.0	236.3	2.0
NR	2/24/2007	Full ΔBR	400.0	21,662.0	16,833.0	1,051.0	253.0	7.0
NR	2/24/2007	Full $\lvert$BR$\rvert$	381.7	21,407.7	16,590.3	1,045.0	236.0	2.0
NR	2/24/2007	ST-Min	**1,320.8**	**21,976.1**	**17,260.2**	1058.5	**418.5**	11.9
NR	2/24/2007	ST-Min ΔST	1,222.0	21,888.0	17,012.0	**1,064.0**	374.3	**14.0**
NR	2/24/2007	ST-Min $\lvert$ST$\rvert$	1,147.0	21,669.0	16,870.0	1,057.0	384.3	4.0
NR	2/24/2007	ST-Min ΔBR	1,216.0	21,936.7	17,176.7	1,058.0	353.0	11.0
NR	2/24/2007	ST-Min $\lvert$BR$\rvert$	1,141.7	21,683.7	16,899.3	1,057.0	392.0	6.0
NR	4/24/2007	Full	244.9	21,979.5	17,320.8	1,057.7	7.1	1.3
NR	4/24/2007	Full ΔST	246.0	22,137.0	17,279.0	1,064.0	7.0	1.0
NR	4/24/2007	Full $\lvert$ST$\rvert$	238.3	21,665.0	16,948.0	1,053.0	9.0	1.0
NR	4/24/2007	Full ΔBR	243.0	22,125.0	17,485.3	1,064.0	6.0	2.0
NR	4/24/2007	Full $\lvert$BR$\rvert$	238.0	21,664.0	16,966.0	1,053.0	8.0	1.0
NR	4/24/2007	ST-Min	1067.2	22,073.4	17,435.8	1,061.5	16.7	2.2
NR	4/24/2007	ST-Min ΔST	**1,129.0**	22,114.0	17,320.0	**1,066.0**	**18.0**	**4.0**
NR	4/24/2007	ST-Min $\lvert$ST$\rvert$	1,023.0	21,807.0	17,091.0	1,056.0	9.3	1.0
NR	4/24/2007	ST-Min ΔBR	1,077.0	**22,176.0**	**17,547.0**	1,063.0	17.0	2.0
NR	4/24/2007	ST-Min $\lvert$BR$\rvert$	1,009.3	21,850.0	17,140.0	1,058.0	9.3	1.0
1.7	10/19/2007	Full	283.3	21,857.0	17,430.5	1,056.3	9.9	2.1
1.7	10/19/2007	Full ΔST	286.3	22,072.0	17,440.0	**1,063.0**	18.0	3.0
1.7	10/19/2007	Full $\lvert$ST$\rvert$	264.0	21,615.7	17,114.0	1,055.0	12.0	2.0
1.7	10/19/2007	Full ΔBR	283.7	22,088.0	**17,674.7**	1,061.0	11.0	4.0
1.7	10/19/2007	Full $\lvert$BR$\rvert$	277.3	21,633.0	17,145.7	1,054.0	13.0	2.0
1.7	10/19/2007	ST-Min	1144.3	22,040.2	17,596.2	1,060.1	22.7	3.8
1.7	10/19/2007	ST-Min ΔST	**1,186.0**	22,023.0	17,426.0	**1,063.0**	**26.0**	5.0
1.7	10/19/2007	ST-Min $\lvert$ST$\rvert$	827.7	21,748.0	17,202.3	1057.0	12.0	2.0
1.7	10/19/2007	ST-Min ΔBR	1,166.0	**22,080.0**	17,674.0	1,060.0	24.0	**6.0**
1.7	10/19/2007	ST-Min $\lvert$BR$\rvert$	1,081.7	21,798.0	17,297.3	1,058.0	13.0	2.0
1.8.5	3/31/2011	Full	1,015.0	**20,708.4**	**15,490.4**	1,363.1	0.8	0.8
1.8.5	3/31/2011	Full ΔST	1,004.7	20,509.7	15,183.0	1,349.0	2.0	**1.0**
1.8.5	3/31/2011	Full $\lvert$ST$\rvert$	971.7	20,499.0	15,209.0	1,347.0	2.0	**1.0**
1.8.5	3/31/2011	Full ΔBR	1,006.0	20,637.0	15,310.7	**1,366.0**	**3.0**	**1.0**
1.8.5	3/31/2011	Full $\lvert$BR$\rvert$	982.3	20,520.3	15,247.3	1,350.0	2.0	**1.0**
1.8.5	3/31/2011	ST-Min	1,819.0	20,577.7	15,323.7	1,350.4	0.3	0.3
1.8.5	3/31/2011	ST-Min ΔST	1,806.0	20,400.0	15,069.67	1,343.0	1.0	**1.0**
1.8.5	3/31/2011	ST-Min $\lvert$ST$\rvert$	1,728.67	20,323.0	15,009.0	1,342.0	1.0	**1.0**
1.8.5	3/31/2011	ST-Min ΔBR	**1,820.0**	20,464.0	15,133.3	1,349.0	1.0	**1.0**
1.8.5	3/31/2011	ST-Min $\lvert$BR$\rvert$	1,757.67	20,322.3	15,013.0	1,343.0	1.0	**1.0**

tests, **DD-Min** 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 5 minutes per test case. Results below show why **DD-Min** was not included in experimental evaluation of quick test methods (essentially, it provided extremely poor code coverage, leaving many very shallow bugs potentially uncaught; it also fails to provide enough tests for a 5 minute budget).

Two additional small suites, **GE-ST(Full** and **GE-ST(ST-Min)** were produced by applying Chen and Lau's GE heuristic [30] for coverage-based suite minimization to the **Full** and **ST-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).

The evaluation measures for suites are: statement coverage (ST), branch coverage (BR), function coverage (FN), number of failing tests, and 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 we 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 [10]. 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 detected by a suite. Faults detected by the test suites system would likely have been of interest to developers, as they presumably were not detected by whatever tests were actually run before code commits. 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 during SpiderMonkey development.

In order to produce 30 second and 5 minute test suites (the extremes of the likely quick test budget), it was necessary to choose subsets of **Full** and **ST-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 [11]. While a large number of plausible prioritization strategies exist, we restricted our study 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, we would like to make our methods as lightweight and generally applicable as possible. We therefore chose four coverage-based prioritizations from the literature [11], [31], which we refer to as 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 test with the highest total statement 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 (NR for non-release), 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 we had hoped for 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.

A. Results: An Effective Quick Test?

Table I provides information on the base test suites across the five versions of SpiderMonkey studied. Tables II and III show how each proposed quick test approach performed on each version, for 30 second and 5 minute test budgets, respectively. For random sampling methods, the results are based on averaging 10 runs, and for prioritizations, the results are based on averaging 3 runs (since the only variation was whether one marginal test case managed to execute at the end of the run). 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 best methods).

The results are fairly striking. First, a purely failure-based quick test such as was used at NASA (**DD-Min**) produces very poor code coverage (e.g., covering almost 100 fewer

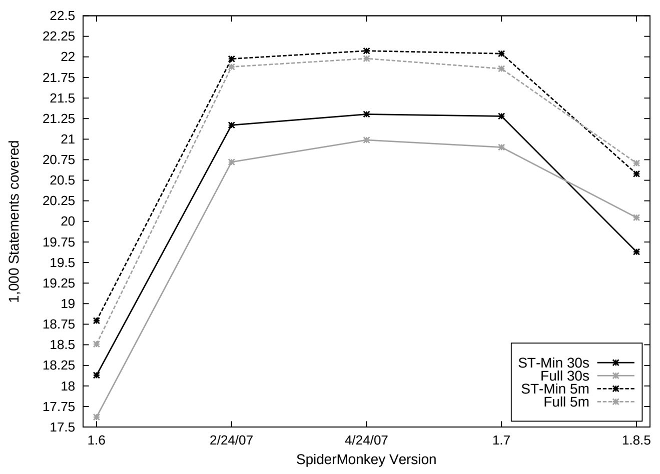

Fig. 1. ST coverage for 30s and 5m quick tests across SpiderMonkey versions

functions than the original suite, and over 300 fewer branches). 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 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(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 5 minute quick test. The benefits of suite minimization by statement coverage (or branch coverage) were represented in the 30 second and 5 minute budget experiments 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 **ST-Min** suite retains (or almost retains) many properties of the **Full** suite that are *not* guaranteed to be preserved by the *ddmin* algorithm. For version 1.6, only 5 branches are "lost", and (most strikingly) the number of failing test cases is *unchanged*. The estimated set of distinct faults, however, is not unchanged: it has grown from ∼22 faults to ∼43 faults. Even if this improvement is an artifact of imprecise measurement, it strongly suggests that the reduced tests are at least slightly more diverse in faults detected than the original suite. Our best hypothesis as to the cause of the remarkable 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 (many do not, however — the numerous 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 a promising area for future research. Even if we assume some problem with fault categorization, 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 [32]).

Perhaps equally surprisingly, for versions of SpiderMonkey up to four months (at least) after the creation of quick test candidate suites, not only is **ST-Min** a better source for quick tests than the **Full** suite, it is better to perform *random selection on the reduced suite than to use coverage-based prioritization on the full suite*. This holds despite the fact that Δ-coverage based prioritization usually improved coverage for 30 second test budgets, and often improved it for 5 minute budgets. The improvement on prioritization is present even for the 1.7 release version, with the exception that Δ-ST prioritization of the **Full** suite is better for fault detection than random selection from **ST-Min**. It is only with the 1.8.5 release, over four years later, that the reduced suite is clearly generally less effective. Even for 1.8.5, however, ΔBR prioritization of **ST-Min** produces the best function coverage of any method. 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 two-month period). Figure 1 graphically exhibits the raw differences in statement coverage for the suites as quick tests, ignoring the effects of prioritization.

The power of coverage-based cause reduction can also be seen by comparing "equivalent" rows for any version and budget: results for each version are split so that **Full** results are the first five rows and the corresponding prioritization the for **ST-Min** tests are the next five 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 the 1.7 version, nearly a year later. The continued utility of quick tests for the 1.7 version is highly surprising, given that this period involves over 1,000 developer commits and the addition of 10,000+ new lines of code (a 12.5% increase).

It is difficult to generalize from one subject, but based on the SpiderMonkey results, we believe that a good initial quick test strategy to try for other projects would be to combine cause reduction by statement coverage with test case prioritization by either additional (Δ) statement or branch coverage (which of these is superior is not at all clear, though statement coverage seems slightly better). Δ prioritization is clearly to be strongly preferred to absolute coverage prioritization in these experiments, and this seems likely to hold in other cases with a small test budget — tests with high absolute coverage also tend to also take a long time to run, and can be highly redundant.

IV. YAFFS 2.0 Flash File System Case Study

YAFFS2 [33] 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. Lacking a large set of real faults in YAFFS2, we applied mutation testing to check our claim that cause reduction not only preserves source code coverage, but tends to preserve fault detection

TABLE IV
YAFFS2 RESULTS

Suite	#Tests	Time(s)	ST	BR	FN	MUT	Notes
Full	4,240	729.032	4,049	1,925	332	616	Kills 11 mutants not killed by **ST-Min**
ST-Min	4,240	402.497	4,049	1,924	332	611	Kills 6 mutants not killed by **Full**
Full 30s	175.9	30.0	4,009.6	1,849.9	332.0	570.0	
Full ΔST 30s	373.0	30.0	**4,049.0**	1,918.0	332.0	594.0	
Full ΔBR 30s	113.0	30.0	4,031.0	1,900.0	332.0	593.0	
ST-Min 30s	315.1	30.0	4,018.9	1,854.0	332.0	560.5	
ST-Min ΔST 30s	514.0	30.0	**4,049.0**	1,912.0	332.0	**601.0**	
ST-Min ΔBR 30s	550.0	30.0	**4,049.0**	**1,924.0**	332.0	575.0	
Full 5m	1,739.9	300.0	4,043.2	1,910.6	332.0	**608.8**	
Full ΔST 5m	2,027.0	300.0	**4,049.0**	1,921.0	332.0	601.0	
Full ΔBR 5m	2,046.0	300.0	**4,049.0**	**1,925.0**	332.0	604.0	
ST-Min 5m	3,156.6	300.0	4,047.0	1,918.9	332.0	607.1	
ST-Min ΔST 5m	3,346.0	300.0	**4,049.0**	1,924.0	332.0	601.0	
ST-Min ΔBR 5m	3,330.0	300.0	**4,049.0**	1,924.0	332.0	605.0	

and other useful properties of randomly generated test cases. The evaluation used 1,992 mutants, randomly sampled from the space of all 15,246 valid YAFFS2 mutants, using the C program mutation approach (and software) shown to provide a good proxy for fault detection by Andrews et al. [34]. Random sampling of mutants has been shown to provide useful results in cases where full evaluation is not feasible, and the sample rate of 13.1% exceeds the 10% threshold suggested in the literature [35]. Sampled mutants were not guaranteed to be killable by the API calls and emulation mode tested. Table IV shows how suites for YAFFS2 compared, omitting |ST| and |BR| in the interest of space; these performed poorly for coverage, though maximized mutant kills to 611/610 for both base suites at 5 minutes. The poorer performance of **ST-Min** at 5 minutes here is probably because 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: the average length of original test cases was 1,004 API calls, while reduced tests averaged 213.2 calls. 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. While we do not know if mutant scores are good indicators of the ability of a suite to find, e.g., subtle optimization bugs in compilers or, mutant kills *do* seem to be a reliable method for estimating the ability of a suite to detect the 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 plus ΔST is the best way to produce a 30 second quick test.

V. GCC: The Potentially High Cost of Reduction

Finally, we attempted to apply cause reduction to test cases produced by Csmith [7] using the GCC 4.3.0 compiler (released 3/5/2008), using CReduce [16] modified to attempt only line-level reduction, since we hypothesized that reducing C programs would be more expensive than reducing SpiderMonkey or YAFFS2 test cases, which have a simpler structure. Our hypothesis proved more true than we had anticipated: after 6 days of execution (on a single machine rather than a cluster), our reduction produced only 12 reduced test cases! The primary problem is twofold: first, each run of GCC takes longer than the corresponding query for SpiderMonkey or YAFFS2 tests, due to the size and complexity of GCC (tests are covering 161K+ lines, rather than only about 20K as in SpiderMonkey) and the inherent start up cost of compiling even a very small C program. Second, the test cases themselves are larger — an average of 2,222 reduction units (lines) vs. about 1,000 for SpiderMonkey and YAFFS — and reduction fails more often than with the other subjects.

While 12 reduced test cases do not make for a particularly useful data set, the statistics for these instances did support the belief that reduction with respect to statement coverage preserves interesting properties. First, the 12 test cases selected all crashed GCC 4.30 (with 5 distinct faults, in this case confirmed and examined by hand); after reduction, the test cases were reduced in size by an average of 50%, and all tests still crashed GCC 4.3.0 with the same faults. For GCC 4.4.0 (released 4/21/2009), no test cases in either suite caused the compiler to fail, and the reduced tests actually covered 419 *more* lines of code when compiled. Turning to branch coverage, an even more suprising result appears: the minimized tests cover an additional 1,034 branches on GCC 4.3.0 and an additional 297 on 4.4.0. Function coverage is also *improved* in the minimized suite for 4.4.0: 7,692 functions covered in the 12 minimized tests vs. only 7,664 for the original suite. Unfortunately the most critical measure, the gain in test efficiency, was marginal: for GCC 4.3.0, the total compilation time was 3.23 seconds for the reduced suite vs. 3.53 seconds for the original suite, though this improved to 6.35s vs 8.78s when compiling with GCC 4.4.0. Even a 50% size reduction does not produce much runtime improvement, due to the high cost of starting GCC.

VI. Threats to Validity

First, we caution that cause reduction by coverage 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 likely that human-produced test cases (or test cases from directed testing that aims to produce short tests) would be not reduce well enough to make the effort worthwhile. The quick test problem is formulated specifically for random testing, though we suspect that the same arguments also hold for model checking traces produced by SAT or depth-first-search, which also tend to be long and redundant.

The primary threat to validity is that 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 a very large system, the GCC compiler.

VII. Related Work

Related work can be divided into two large categories. First, this paper is an extension of the previous work on delta debugging [3], [13], [18] and related methods for reducing the size of failing test cases, including program slicing [36]. While previous work has attempted to generalize the types of circumstances to which delta debugging can be applied [19], [20], [37], this paper replaces the goal of minimizing with respect to failure to the goal of minimizing with respect to any arbitrary effect. Surveying the full scope of work on failure reduction is beyond the scope of this paper, as the period since the introduction of delta debugging has seen a steady thread of work on the topic, in both testing [16], [38], [39] and model checking [40], [41]. Perhaps most relevant to this paper are examinations of the effectiveness of delta debugging in random testing [14], [1], [2], [15]. In a larger sense, work on causality could be considered as related; however, cause reduction makes no particular theoretical commitments other than a general Lewis-like counterfactual [42], [43] assumption that anything that can be "gotten rid of" easily without removing the effect is not causally interesting.

Second, we propose 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 [11]. Namely, while other approaches have focused on minimization [44], [45], [46], [30], selection [12] and prioritization [31], [47], [48] at the granularity of entire test cases, this paper proposes reducing the size of the test cases composing the suite, a "finer-grained" approach that can be combined with previous approaches. 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 [32], [26].

VIII. 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, which we call *cause reduction*, allows us to produce *quick tests*: 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 a third subject, the GCC compiler). Combining cause reduction by statement coverage with test case prioritization by additional statement coverage produced, across 30 second and 5 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 GCC that are up to a year later in development time, a long period for such actively developed projects.

In future work we first propose to further investigate the best strategies for quick tests, across more subjects, to determine if the results in this paper generalize well. Second, it is clear from GCC that cause reduction by coverage is too expensive for some subjects, and the gain in efficiency is relatively small compared to the extraordinary computational demands of reduction. Two alternative mitigations come to mind: first, it is likely that reduction by even coarser coverages, such as function coverage, will result in much faster reduction (as more passes will reduce the test case) and better efficiency gains. Whether cause reduction based on coarse coverage will preserve other properties of interest is doubtful, but worth investigating, as statement coverage preserved other properties much more effectively than we would have guessed. Initial experiments with function coverage based reduction of SpiderMonkey tests showed good preservation of failure and fault detection, but we did not investigate how well preservation carried over to future versions of the software yet. A second mitigation for slow reduction (but not for limited efficiency gains) is to investigate changing *ddmin* to fit the case where expected degree of minimization is much smaller than for failures, and where the probabilities of being removable for contiguous portions of a test case are essentially independent, rather than typically related, which motivates the use of a binary search in *ddmin*.

We also propose other uses of cause reduction. While some applications are relatively similar to coverage-based minimization, e.g., reducing tests with respect to peak memory usage, security privileges, or other testing-based predicates, other possibilites arise. For example, reduction could be applied to a program itself, rather than a test. A set of tests (or even model checking runs) could be used as an effect, reducing the program with respect to its ability to satisfy all tests or specifications. If the program can be significantly reduced, it may suggest a weak test suite, abstraction, or specification. This approach goes beyond simply examining code coverage because examining code that is removed despite being covered by the tests/model checking runs can identify code that is truly under-specified, rather than just not executed (or dead code).

References

[1] 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.

[2] 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*, accepted for publication.

[3] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 183–200, 2002.

[4] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of haskell programs," in *ICFP*, 2000, pp. 268–279.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Automated Software Engineering*, 2005, pp. 273–282.

[10] 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, to appear.

[11] 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.

[12] 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.

[13] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *International Symposium on Software Testing and Analysis*, 2000, pp. 135–145.

[14] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *International Symposium on Software Reliability Engineering*, 2005, pp. 267–276.

[15] 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.

[16] 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.

[17] W. McKeeman, "Differential testing for software," *Digital Technical Journal of Digital Equipment Corporation*, vol. 10(1), pp. 100–107, 1998.

[18] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *ESEC / SIGSOFT Foundations of Software Engineering*, 1999, pp. 253–267.

[19] J. Choi and A. Zeller, "Isolating failure-inducing thread schedules," in *International Symposium on Software Testing and Analysis*, 2002, pp. 210–220.

[20] 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.

[21] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *SPIN Workshop on Model Checking of Software*, 2003, pp. 121–135.

[22] B. Beizer, *Software Testing Techniques*. International Thomson Computer Press, 1990.

[23] T. Ball, "A theory of predicate-complete test coverage and generation," in *Formal Methods for Components and Objects*, 2004, pp. 1–22.

[24] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *Trans. Software Eng.*, vol. 19, pp. 774–787, 1993.

[25] 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, to appear.

[26] 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.

[27] 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.

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

[29] ——, "Mozilla bug 349611," https://bugzilla.mozilla.org/show_bug.cgi?id=349611 (A meta-bug containing all bugs found using jsfunfuzz.).

[30] 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.

[31] 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.

[32] 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.

[33] "Yaffs: A flash file system for embedded use," http://www.yaffs.net/.

[34] 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.

[35] 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.

[36] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, pp. 121–189, 1995.

[37] A. Zeller, "Isolating cause-effect chains from computer programs," in *Foundations of Software Engineering*, 2002, pp. 1–10.

[38] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging," in *International Conference on Software engineering*, 2006, pp. 142–151.

[39] S. McPeak and D. S. Wilkerson, http://delta.tigris.org.

[40] 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.

[41] 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.

[42] D. Lewis, "Causation," *Journal of Philosophy*, vol. 70, pp. 556–567, 1973.

[43] ——, *Counterfactuals*. Harvard University Press, 1973 [revised printing 1986].

[44] 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.

[45] 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.

[46] 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.

[47] 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

[48] 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.