Call Stack Coverage for Test Suite Reduction

Scott McMaster and Atif M. Memon
Department of Computer Science, University of Maryland, College Park, MD 20742
{scottmcm, atif}@cs.umd.edu

Abstract

Test suite reduction is an important test maintenance activity that attempts to reduce the size of a test suite with respect to some criteria. Emerging trends in software development such as component reuse, multi-language implementations, and stringent performance requirements present new challenges for existing reduction techniques that may limit their applicability. A test suite reduction technique that is not affected by these challenges is presented; it is based on dynamically generated language-independent information that can be collected with little run-time overhead. Specifically, test cases from the suite being reduced are executed on the application under test and the ***call stacks*** *produced during execution are recorded. These call stacks are then used as a coverage requirement in a test suite reduction algorithm. Results of experiments on test suites for the* `space` *antenna-steering application show significant reduction in test suite size at the cost of a moderate loss in fault detection effectiveness.*

1. Introduction

Test-suite reduction typically employs sophisticated tools such as source-code analyzers and instrumentors to reduce the number of test cases in a given test suite; the obtained subset yields equivalent coverage with respect to some criterion [8, 12, 13, 16, 17]. Emerging trends in software development present new challenges for existing reduction techniques that may limit their applicability. First, developers rely heavily on reusable components. Source code of these components is usually not available, limiting the application of source-code level instrumentors and analyzers [11]. Second, developers use a combination of programming languages to implement systems. Certain static analyzers and source-code instrumentors may not be available (or may be too complex/expensive to execute) for some of these languages. For example, some static analyses become complex in object-oriented systems due to the presence of virtual function calls. Even if analysis techniques are available for each language, combining the results from different analyses may become complex. Finally, systems such as servers, network protocol implementations, and middleware software have strict quality of service (QoS) requirements. Test cases that check for performance in these systems may not tolerate the overhead of instrumentation needed for test suite reduction. Decreased performance (even during in-house testing) may cause these test cases to fail, hence producing incorrect results [10].

In this paper, we describe and evaluate a new test suite reduction technique based on the set of unique *call stacks* dynamically generated by the test suite. A call stack represents the currently active function calls in a stack-based execution environment, in the order in which the calls occurred. Intuitively, test cases whose execution profiles generate the same call stacks are also repetitive in the application functionality and structure that they test, thus making our criterion a good candidate to have a favorable tradeoff between suite size and fault detection effectiveness. It is possible to collect the set of unique call stacks in many computing environments with minimal or no direct instrumentation of the target program. All that is strictly required is environmental hooks similar to those used by profiling and debugging tools (always available on all development platforms), thus greatly reducing the complexity of employing the technique. Additionally, collecting call stacks and analyzing them does not require access to the source code. Finally, since the actual resolved function activation records appear on the runtime stack, our technique is language independent.

In the next section, we present background on test suite reduction and call stacks. In Section 3, we describe the use of call stacks for test coverage. Section 4 describes the call-stack collection and test suite reduction algorithms. Section 5 contains a detailed description of our experiments and results. Section 6 surveys related work, and Section 7 concludes and proposes future research.

2. Background

Test Suite Reduction: As software is developed, test engineers create test cases to detect defects in the software. Engineers may employ several test case generation techniques to create large numbers of test cases that are potentially beneficial in terms of their defect detection ability. As software is modified, test cases are added to cover its new and modified features. At some point in the software development lifecycle, the time it takes to run the entire test suite against a modified version of the software may become excessive.

Since test cases may be redundant with respect to the statements, functions, paths, or other program elements

they execute, researchers have investigated the problem of *test suite reduction* [8, 12, 13, 16, 17], which focuses on reducing the test suite to obtain a subset that yields equivalent coverage with respect to some criterion. The goal of test suite reduction is to generate a test suite that is smaller (and therefore cheaper to execute and maintain) but that still retains much of its original ability to detect faults.

A variety of static and dynamic program analysis criteria have been proposed as the basis for test suite reduction, including edge coverage [13], all-uses dataflow coverage [18], and dynamic program invariants [7]. The general approach is to completely instrument a set of program entities to record information about each entity's coverage, execute the test suite, and reduce based on the collected coverage information. Since different techniques monitor different program entities or behaviors, they have different costs and application environments. Moreover, they generally select different reduced subsets of test suites and therefore have different tradeoffs between reduced suite size and fault detection effectiveness.

Call Stacks: In a stack-based architecture, a thread in a running program has a call stack as a part of its state. Informally, the call stack is simply the series of currently active calls. Function activation records are pushed onto the call stack when they are called and popped when they return.

An example call stack in this form is shown in Figure 1. The top of the stack is at the top of the figure as indicated by the arrow.

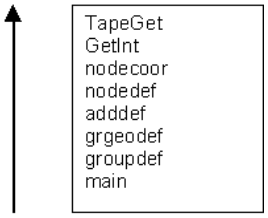

Figure 1: A call stack.

Formally, we define a *call stack c* as an ordered sequence of functions (for the discussion in this paper, we consider the programming language concepts of "methods" and "procedures" to be identical to functions.) $f_1..f_n$:

$$(1)\ \ c = \langle f_1, f_2, \ldots f_{n-1}, f_n \rangle$$

where function f_1 is the entry point of the thread of execution and function f_i calls function f_{i+1}, either directly or indirectly through an intervening function. We allow for indirect calls in our definition of a call stack because in practice, it may not be possible to observe the full call chain due to limitations in the execution environment or program instrumentation technique. In such cases, however, we stipulate that the call stack should consist of all functions that are, in fact, observable. The last element in the sequence, f_n, is the *top* of the stack, and n, the size of the sequence, is the *depth* of the stack. For the purposes of call stack-based analyses, the following two representations of the functions f_i are possible:

1. *Name*: a function may be denoted by just its *Name* (class-qualified in an object-oriented system) or memory address.
2. *Full Signature*: a function may be denoted by its *Name*, *parameter types,* and *return types.* This may be useful for an analysis that can, for example, distinguish between call stacks using different overloads of a function in an object-oriented environment.

A call stack containing at least one recursive function is a *recursive* call stack. A call stack with no such function is a *non-recursive* call stack.

Call Stacks and Program Execution. The *current stack* is the call stack obtained by examining the current set of active functions in an executing program. Each program execution may be viewed as generating a set of current stacks over its lifetime. The set of all such unique stacks shall be denoted as *C*. In general, encoding more details into the function representation by using the full signature rather than just the function name may increase the number of unique call stacks observed in an execution, leading to a finer-grained analysis with an associated increase in analysis costs.

A call stack of depth n implies that at some prior point in the program execution, the *(n-1)* substacks of that stack were themselves the current stack. This is illustrated in Figure 2.

If $c = \langle f_1, f_2, \ldots f_n \rangle$ is a call stack of depth n, we define a *substack* c_s (denoted by a subscript s) and a *superstack* c^s (denoted by a superscript s) as the following ordered sequences, which are themselves call stacks:

$$(2)\ \ c_s = \langle f_1, f_2, \ldots f_i \rangle,\ i < n$$

$$(3)\ \ c^s = \langle f_1, f_2, \ldots f_n, \ldots f_i \rangle,\ i > n$$

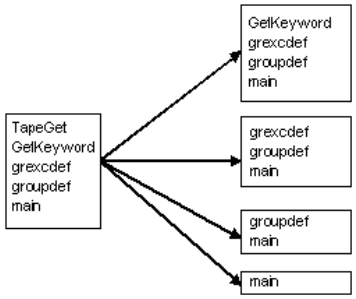

Figure 2: The stack on the left implies that the substacks on the right represented the current sequence of active calls at some earlier point.

For a given call stack c in any program execution, there is associated with c a set of substacks C_s and a set of superstacks C^s. We define the set of deepest, or *maximum depth*, stacks C_{max} in a program execution as follows:

$$(4)\ C_{max} = \{c \in C \mid c^s = \varnothing\}$$

where $\varnothing$ is the empty set. Since each maximum depth stack implies the existence of all of its substacks in C, C_{max} is a more compact representation of the set of all unique call stacks generated by a program execution.

3. Call Stacks and Test Coverage

We define a *test case* as input given to a program in order to test one or more aspects of the program. Running a test case *tc* from a test suite *TS* implies the execution of the program, which itself implies that a set of maximum depth call stacks $C_{max}(tc)$ generated by the execution can be associated with *tc*. We consider two test cases tc_1 and tc_2 to be equivalent if they generate identical sets of maximum depth call stacks.

$$(5)\ tc_1 \sim tc_2 \text{ iff } C_{max}(tc_1) = C_{max}(tc_2)$$

Since a *test suite* is a set of test cases, we denote the union of all C_{max}'s for all the test cases in a test suite *TS* as:

$$(6)\ C_{max}(TS) = \{\cup\ C_{max}(tc) \mid tc \in TS\}$$

Our technique considers a maximum depth call stack to be a *coverage requirement* in the test suite reduction algorithm *ReduceTestSuite* [1]. Thus, execution of a reduced test suite $TS^{reduced}$ will generate the same set of unique maximum depth call stacks as execution of its original (full) counterpart TS^{full}, *i.e.*, $C_{max}(TS^{full}) = C_{max}(TS^{reduced})$.

From a test coverage point of view, examination of call stacks is, by definition, a superset of function coverage since every function covered by an execution must appear in at least one generated call stack. But call stacks encode more semantic information than that, as each function call is not observed in isolation but rather in the context of other calls leading up to it. Intuitively, this suggests that striving to cover each function in as many call stacks as possible would be a more thorough driver of testing than function coverage alone. We later show in Section 5 that, compared to function coverage, call stack coverage does retain more fault detection ability of test cases.

Applying our technique in practice assumes that call stack coverage information is available from prior runs of the test suite, and that such information can be used to generate good testing partitions in modified versions of the software with a fair degree of accuracy. These assumptions are often made in work involving test suite reduction, prioritization, and regression test selection (*e.g.,* [14]).

4. Algorithm

We now present an algorithm to collect call stacks from a series of program executions generated by a test suite, and then show how to use these call stacks to reduce the suite.

Collecting Call Stacks: To write a tool to collect the set of unique call stacks generated by a test case, we need only know when functions are called and when they return. Hence, function addresses are adequate to identify call stacks if we ensure that program and library code segments are loaded at consistent base addresses in each execution. A test designer may also obtain this information in other ways, *e.g.,* by source or binary instrumentation, by attaching a profiler, or by a combination of these approaches. The most appropriate technique for a given application will depend on the availability of source code and the availability of appropriate tools for the target language and platform. Note that symbolic information about program functions may be helpful but is not strictly necessary.

A preliminary approach to building the set of unique call stacks would be to record the dynamic execution trace of the program in the form of calls and returns. Then, the call stacks obtained by each test case could be reconstructed and analyzed to determine their uniqueness in an offline post-processing step. The primary difficulty with this approach is that the volume of data generated by each test case increases linearly with the length of the test case, and this volume of data can easily make the analysis prohibitively expensive in both time and space.

An alternative approach, and the one we have used in our work, is to construct the application's calling context tree, or CCT [2], as a test case is running. The CCT is a tree data structure where the root represents the function that is the entry point of a thread, and each child node represents a call to a specific function made by its parent. It is possible to construct a CCT efficiently at runtime by using the following process, which is discussed in more detail in [2]:

1. Create a node representing the entry point of the thread and make it the current node.
2. When a function is called:
   a. If the current node has a child node representing the called function, make that the current node.
   b. If a node representing the called function is an ancestor of the current node, the call is recursive. Create a *backedge* to that ancestor node and make it the current node. (Note that in the presence of backedges, a dag needs to be used instead of a CCT.)
   c. If the current node does not have a child node representing the called function, create such a node and make it the current node.

3. When a function returns, set the current node to its parent.

This process can be accomplished at roughly the runtime cost of attaching a simple function-level profiler. More importantly, while generally large for non-trivial applications, the size of the CCT data structure depends only on the size of the set of unique call stacks and not on the run-time data generated by each test case, thus making the resulting data volume manageable.

At the end of test case execution, we traverse each path to a leaf in the CCT, not following any backedges caused by recursion. The resulting set of paths gives us the set of unique non-recursive maximum depth call stacks obtained by the test case. To collect the set of unique recursive maximum depth stacks, we can modify the CCT data structure to associate a counter with each backedge that is incremented when a recursive call is made. During traversal, we follow the backedge as many times as the counter indicates. Consideration of recursive maximum depth call stacks in the context of test suite reduction is a subject for future work. Henceforth, when we say "maximum depth stacks", we will be referring to *non-recursive* maximum depth stacks.

Reducing Test Suites: After executing the test suite and collecting the call stacks covered by each test case (encoded as a CCT), we are ready to reduce the suite using a two-step process.

Step 1: Merge Stacks from Different Test Cases. Our goal is to obtain a reduced test suite that covers each unique maximum-depth call stack generated by the full test suite. Thus, we must build a set of all such call stacks before reducing the suite. We can do this by merging the sets of maximum-depth stacks generated by the individual test cases in the original suite.

However, we must consider the situation where a maximum-depth stack of one test case is a substack of a maximum-depth stack of another test case. Consider the example where Test Case 1 (tc_1) has a maximum-depth stack $c_1 = <f_1, f_2, f_3>$ and Test Case 2 (tc_2) has a maximum-depth stack $c_2 = <f_1, f_2>$, *i.e.,* a call from f_1 to f_2 where f_2 never calls f_3. One approach in this case would be to treat c_2 as though it were a unique maximum-depth stack across the entire suite and include it in the merged set. This approach has the advantage of not requiring that we determine the substack relationship when inserting each stack into the merged set. Moreover, the second stack may represent an execution profile where there is a program failure that causes f_2 to not call f_3; removing the test case might potentially remove such fault-detection capability – we will study this effect in the future. The disadvantage to this approach is that it introduces a redundant coverage requirement and therefore may lead to a worse reduction than we might otherwise obtain. Another approach is to check each stack before inserting it into the merged set to ensure that it is not a substack of any previously inserted stack. Obviously this approach is more computationally intensive, but it should yield the best possible reduction. Because one of our goals in this work is to examine the size reduction achieved by the call stack reduction technique (we later show in Section 5 that the time to compute the reduced set, even with this overhead, is very small), we choose the latter approach.

Step 2: Reduce the Test Suite. Finally, we use our call stack coverage information to reduce the size of the test suite to only those test cases necessary to generate the full set of unique maximum depth call stacks. Given a test suite and test case coverage information, the problem of generating a minimum number of test cases that meet the coverage criteria is NP-complete [8]. Thus, existing techniques for coverage-based reduction generate approximations using greedy or heuristic approaches.

In this work, we apply the *ReduceTestSuite* heuristic from [8]. Briefly, the heuristic begins by including all test cases that cover a single requirement. Then it picks a test case that covers the most requirements from the subsets of cases with the next lowest cardinality, marking all of the subsets that contain this case. This process occurs repeatedly for higher cardinality subsets until all subsets are marked and, therefore, all requirements are covered. For a more formal treatment of this algorithm, including an analysis of its running time and an application to data flow testing, see [8]. To utilize *ReduceTestSuite*, we define each unique call stack as a coverage requirement. Then we associate each requirement with the subset of test cases that covered it and run the algorithm as specified in [8].

5. Experiments

We implemented the call-stack collection and reduction algorithms and ran two experiments to evaluate our test suite reduction technique.

Research Questions: We sought to evaluate the call stack reduction technique in terms of the size and fault detection effectiveness of the resulting test suites. Specifically, we wanted to directly compare the call-stack based technique to reduction based on two different types of coverage: *edge* and *function*. We also wanted to investigate whether test suites created by call stack reduction preserved more fault-detecting ability than randomly reduced suites of the same size. Finally, we wanted to determine the cost of applying our technique in terms of execution time. To that end, we designed two experiments that we present next: (1) *Experiment 1*, in which we compared call stack based reduction with edge-coverage and random reduction, and (2) *Experiment 2*, in which we compared call stack reduction to function-coverage based reduction.

Subject Application: The application we used in our experiments is `space` [14]. `Space` is an antenna-steering system developed by the European Space Agency written in C and comprised of about 6200 non-

commentary lines of code. It is well studied in the area of regression test selection because of the availability of a test pool of 13,585 existing test cases as well as 38 program versions containing naturally occurring faults. Of these 38 faulty versions, we identified and used the 34 that were not semantically equivalent to the base version.

Instrumentation: To illustrate the practicality of using binary instrumentation to capture call stacks at runtime, we used the Detours package [9] to instrument the `space` executable. Detours is a library that allows dynamic interception of binary function calls on the Win32 platform without modifying the on-disk program. We used Detours' "dynamic trampoline" functionality to insert hooks at each function entry and exit in `space` to build the CCT. This approach required specific instrumentation code for each function in `space` and the use of a version containing debugging symbols. This instrumentation code was generated by a tool whose input was a list of function prototypes. The generated code was built into a separate code module attached to the space process at runtime using functionality in Detours. Thus, neither the source code nor the on-disk program for `space` was modified. Because `space` does not overload function names, our call stack representation may use function names rather than full function signatures with no change in outcome. Also, because `space` is not a recursive program, considering non-recursive maximum depth stacks as discussed in Section 4 yields identical results to a hypothetical approach that considered recursive stacks as well.

Since `space` uses the Standard C Library, we needed to address instrumenting that code as well. Instead of instrumenting all public and internal functions in the library (which would require examination of the full library source code in our Detours-based approach), we chose to only instrument those functions defined in the public C library headers and called by `space` or a macro used by `space`. Thus, internal library functions do not appear on the call stacks we collected, making them in fact an approximation. There is a tradeoff between the level of detail included in the call stacks (and thus the effectiveness of the technique) on one hand and the practicality of instrumentation and analysis time on the other. In future work, we plan to investigate this tradeoff by repeating the experiments with a fully instrumented Standard Library (possibly using a more amenable instrumentation technique such as the *finstrument-functions* flag in gcc [1] or similar functionality in other compilers), as well as repeating the experiments without considering library function calls in call stacks at all.

Measured Variables: We measured fault detection effectiveness on a per-test-suite basis, *i.e.,* two test suites were considered to be equally effective at detecting a specific fault if they each contain at least one case that exposes the fault. This is the approach adopted in [12] and [17]. For each reduction experiment, we captured the percentage size reduction:

$$(1)\quad 100\times\left(1-\frac{\text{Size}_{\text{Reduced}}}{\text{Size}_{\text{Full}}}\right)$$

and percentage fault detection reduction:

$$(2)\quad 100\times\left(1-\frac{\text{FaultsDetected}_{\text{Reduced}}}{\text{FaultsDetected}_{\text{Full}}}\right)$$

Since we dealt with a fairly small number of discrete faults in our experiments, we took averages of these quantities over large numbers of suites. The precise number of suites is noted in each experiment.

Threats to Validity*: Threats to external validity* are factors that may impact our ability to generalize our results to other situations. Our main threat to external validity in this study is the small sample size. Thus far, we have only run our data collection and test suite reduction process on one program, which we chose for its availability. This program may not be representative of the broader population of programs. An experiment that would be more readily generalized would include multiple programs of different sizes and from different domains. Additionally, we would expect the effectiveness of the call stack minimization process to vary depending on aspects of the programming style used in the target application. In particular, when the application is composed of many small functions, call stacks provide finer-grained dynamic state information. This should increase the effectiveness of our minimization technique relative to what it could do against an application that implemented the same behavior using relatively fewer functions. (Consider the pathological case where a program is composed of a single large function, which would have but a single call stack for all executions.) Finally, characteristics of original test suites (such as their fault detecting ability and how they were constructed) play a role in the size and fault detection reduction results. This threat can be addressed in future work by choosing original test suites satisfying a variety of coverage criteria.

Threats to construct validity are factors in the experiment design that may cause us to inadequately measure concepts of interest. In our experiments, we made several simplifying assumptions in the area of costs. In test suite reduction, we are primarily interested in two different effects on costs. First, there is the cost savings obtained by running fewer test cases. In this study, we assume that each test case has a uniform cost of running (processor time) and monitoring (human time). These assumptions may not hold in practice. The second cost of interest is the cost of failing to find faults during testing as a result of running fewer test cases. Here we assume that

each fault contributes uniformly to the overall cost, which again may not hold in practice. We note that our assumptions are the same as those made in other studies of test suite reduction, including [13] and [16], and thus have precedent as a basis for conclusions about reduction techniques.

Information-Gathering Preprocessing Step & Feasibility Study: We executed each test case in the test pool against the fault-free version of the `space` program, collecting the unique call stacks from each test case. During this process, we did not notice any performance degradation in test case execution. We encountered 143 unique functions comprising 453 unique maximum depth call stacks. Then we executed each test case against each of the 34 faulty versions and recorded the set of faults detected by each case.

The data gathered during this preprocessing step allowed us to create any number of test suites composed of the previously executed test cases and know the set of unique call stacks and faults detected by the suite with no further execution of the program. Hence, it was not necessary to run each test suite under study against each version of the subject program. This simulation approach is similar to one used by Frankl [5, 6] to evaluate adequacy criteria and test effectiveness.

As an initial informal demonstration of the practicality of our algorithm, we evaluated the time taken to reduce randomly generated test suites ranging in size from 50 to 1000 test cases in increments of 50. (We formally evaluate the fault detection effectiveness of suites created in this manner in Section 5). We created five suites of each size and, using the information obtained in the preprocessing step, applied the three-step reduction process described in Section 4. We averaged amount of wall time taken to reduce the five suites and plotted against original suite size. The test platform was a 2.4 GHz Pentium 4 running Windows XP Professional, and our test suite reduction process was implemented as a set of scripts in the interpreted Ruby language [15] with the *ReduceTestSuite* algorithm itself implemented in C#.

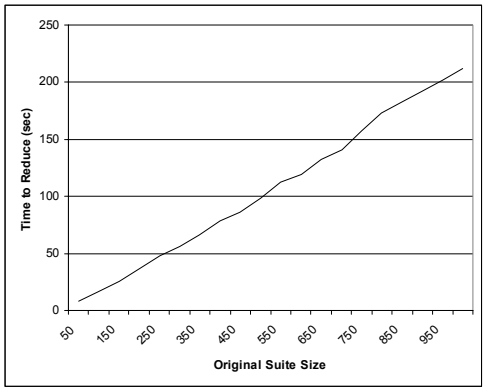

Figure 3: Time to Reduce Suites

The results of this experiment appear in Figure 3. The x-axis shows the original suite size and the y-axis shows the time in seconds. We observe that the amount of time to execute the reduction process scales roughly linearly with the size of the original test suite, and even the largest suites we considered took less than four minutes of wall time on average to reduce on the test platform in the test environment.

Experiment 1: Reducing Edge-Coverage-Adequate Suites: The goal of our first experiment was to apply call stack reduction to a set of edge-coverage-adequate test suites and compare the results to those achieved by edge-coverage-based reduction. We used the 1000 test suites for `space` used by Rothermel *et al.*[13]. Each of these suites consists of a random number of test cases drawn from the pool, augmented with additional test cases to ensure edge-coverage adequacy. They had an average size of 2400 test cases and detected on average 33.5 of the 34 detectable faults.[1] The largest and smallest of these test suites had 4712 and 159 test cases respectively.

For our experiment, these suites have two positive attributes. First, their edge-coverage adequacy arguably makes them representative of well-designed test suites. Second, these suites were made available to us along with a corresponding set of reduced suites that were generated using edge coverage as the criterion in the *ReduceTestSuite* algorithm. Thus, we are able to make a direct comparison between test suite reduction based on edge coverage versus reduction based on maximum depth call stacks.

***Experimental Process*:** We repeated the following steps for each of the 1000 edge-coverage-adequate test suites. The large number of suites helps to control for any effects (such as random fluctuations in effectiveness) caused by the nature of the test suites themselves.

Step 1. *Select a test suite.*

Step 2. *Form the set of faults detected by the full suite.* To form the set of faults detected by the full test suite, we merged the sets of faults detected by the individual test cases in the suite.

Step 3. *Form the set of unique call stacks for the full suite.* The set of unique call stacks that would be generated by a run of the full test suite was created by merging those generated by the individual test cases, eliminating any maximum depth stacks of one test case

[1] Our fault detection reduction numbers for the Rothermel suites will differ from what is presented in [13] because the work in [13] is based on 35 detectable faults. We used a different compiler in which one of those 35 faults was eliminated or masked. Because we evaluated the Rothermel suites against our own fault detection information based on the newer compiler, this does not affect the validity of any of our comparisons.

that are substacks of maximum depth stacks of another test case.

Step 4. *Reduce the test suite using the heuristic***.** Here we applied the heuristic from [8] to reduce the test suite to just those cases necessary to cover each maximum depth call stack.

Step 5. *Form the set of faults detected by the reduced suite***.** Forming the set of faults detected by the reduced suite was carried out by merging the sets of faults detected by the individual test cases selected to be part of the reduced suite.

Step 6. *Calculate results***.** Using the sets of detected faults built in steps 2 and 5, we calculated the percentage size reduction and percentage fault detection reduction when going from the full to the reduced test suite. These quantities were described in Section 5.

***Evaluation and Results*:** Figure 4 shows the absolute sizes of test suites reduced based on both edge (dark data points on the plot) and call stack (light data points) coverage versus original suite size. The plot shows that call stack reduction was able to obtain suites that were roughly half the size of the ones obtained by edge coverage. Both techniques, however, lost some fault-detection ability in their reduced test suites. Figure 5 shows box-plots of the number of faults detected by the full, edge reduced, call stack, and random (we describe "random" later in a subsequent paragraph in this section) reduced test suites. Each box (with tails) represents a data distribution; the box itself contains the central 50% of the data-points in the distribution and each tail represents 25%; the dot inside the box shows the median. Hence, looking at the box for "Full-suite faults", we note that the original (unreduced) test suites were able to detect between 30 and 35 faults; most of them detected 33-34 faults. As noted earlier, the "edge-coverage reduced faults" box shows a reduction in fault-detection ability; on the average 30 faults were detected. The call stack based reduction resulted in test suites that detected a comparable (although slightly smaller) number of faults than their edge-coverage reduced counterparts. Note, however, that the edge-coverage reduced suites were much larger (double) than the call-stack reduced ones.

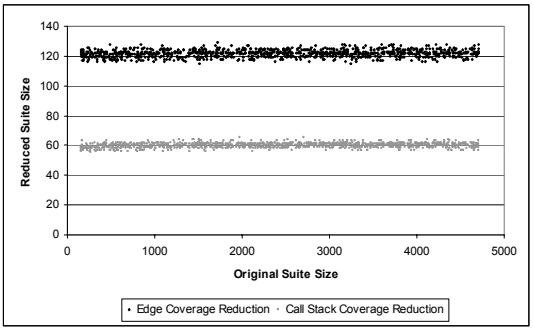

Figure 4: Reduced Suite Sizes

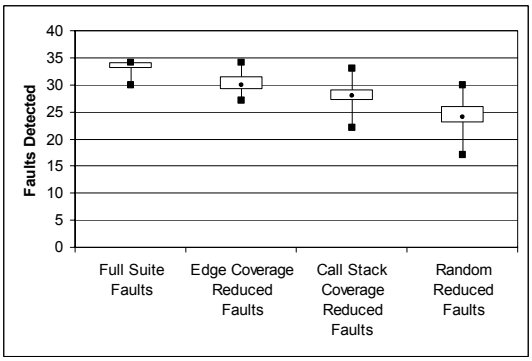

Figure 5: Fault Detection Reduction

Means Over 1000 Test Suites							
Original		Edge-Reduced		Call Stack-Reduced		Random-Reduced	
Size	Faults Detected	Size	Faults Detected	Size	Faults Detected	Size	Faults Detected
2399.5	33.5	121.7	30.4	60.0	28.0	60.0	24.2
% Reduction From Original		90.1	9.2	95.2	16.3	95.2	27.6

Table 1: Reduction results for edge coverage adequate suites

The results of this experiment are summarized in Table 1. The table shows a large average test suite size reduction (95.2%) and a fault detection reduction of approximately 16% when using call stack reduction. In contrast, using edge coverage as the reduction criterion led to a smaller percentage reduction (90.1%) and a smaller loss in fault detection effectiveness (9.2%). In absolute terms, suites resulting from call-stack-based reduction were less than half the size (60.0 versus 121.7 test cases on average) of suites resulting from edge-based reduction while detecting an average of about 2 fewer faults. The practical trade-off between suite size and fault detection capability may favour the use of call stacks as opposed to edges as a coverage criteria for reduction in certain scenarios, such as when edge coverage results in suites too large to run inside time constraints. Additionally, we maintain that the tools to collect call stack coverage information are significantly simpler to develop and use than those necessitated by edge coverage.

To further validate the effectiveness of the call stack reduction technique, we used the approach taken in [13] to permit us to compare call stack reduction to random reduction. Specifically, we compared each call stack reduced suite to a suite of the same size generated by randomly selecting test cases from the original suite. (The random-reduced suites are compared with the edge and call stack reduced suites in Figure 5 and Table 1.) We then performed a *paired-T test* across all 1000 samples. The paired-T test is a statistical test used to determine whether the means of two sets of samples differ in a

statistically significant way, *i.e.,* any differences are not the result of random fluctuations. The null hypothesis (H_0) is that there is no difference between the percentage reductions in fault detection effectiveness between call stack reduced test suites and randomly reduced suites of the same size. The key output of a paired-T test is the *p-value*, where $p < 0.05$ indicates statistical significance. Thus, the null hypothesis would be rejected only if $p < 0.05$, and the alternate hypothesis (H_1), *i.e.,* there is a statistically significant difference between the two sets, would be accepted.

Where the call stack reduced suites lost just over 16% of their fault detection ability, the paired random suites of the same size lost nearly 28%, a difference of over 11%. Because the p-value of the paired-T test is of the order of 10e-221 ($t_{999, 0.975} = 24.11$), which is much less than 0.05, we can safely reject the null hypothesis, accept the alternate hypothesis and conclude that call stack reduction retains fault detection effectiveness better than random reduction to the same suite size.

Experiment 2: Reducing Various Sizes of Test Suites: The goal of this experiment was to evaluate call stack coverage reduction versus function coverage reduction on sets of randomly generated test suites of different fixed sizes. We randomly generated test suites in 20 sizes ranging from 50 to 1000 test cases in increments of 50.

Experimental Process: The approach and evaluation are similar to Experiment 1. The primary differences are that we control the original suite size, and we compare to suites reduced based on function coverage. We repeated the same six steps of Experiment 1 on each test suite created by random selection without replacement from the test pool. In order to minimize the statistical effects of both the relatively small number of faults and the differing difficulty of detecting them with cases from the test pool we took averages over 50 suites of each size. Hence, in all, we report results for 50x20=1000 test suites.

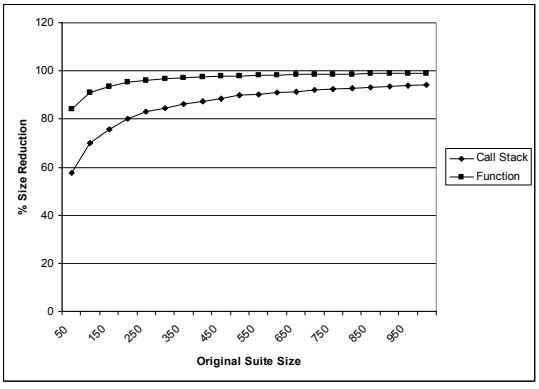

Figure 6: Size of Reduced Suites

Evaluation and Results: Figure 6 and Figure 7 show the reduced test suites size and percentage fault detection reduction, respectively, versus the original test suite size. Note that the relatively small number of available faults makes the analysis highly sensitive to the detection of individual faults, thereby leading to some jaggedness in the graphs.

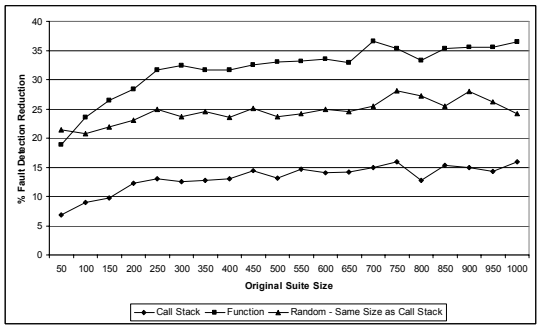

Figure 7: Percent Fault Detection Reduction

Consistent with the findings in [16], fault detection effectiveness generally decreases as the percentage reduction in suite size increases. At the same time, the number of faults detected by the reduced suite increases as the original suite size increases, tapering off as it approaches the maximum number of detectable faults, which is also expected.

We see that reducing based on function coverage yields smaller suites than reducing based on call stack coverage. This is to be expected, since function coverage is a subset of call stack coverage. However, the greater percentage size reduction comes at the high price of more than twice as much percentage fault detection reduction from the original suites. This tradeoff may or may not be acceptable in practice; however, revisiting the edge coverage results in Experiment 1, we suggest that call stack reduction provides a balance between the technical and procedural simplicity and small reduced suite size afforded by the function coverage technique and the greater fault detection effectiveness of the edge coverage technique.

As in Experiment 1, we paired each call stack reduced suite to a randomly reduced suite of the same size and evaluated the significance of our results using a paired-T test with the null hypothesis that call stack reduced and randomly reduced suites perform equally well in terms of percentage reduction in fault detection effectiveness. The results are shown in Table 2. The columns show the original test suite sizes with reduced sizes and loss of fault-detection effectiveness obtained by function, call stack and random reduction. Here again we see a statistically significant difference in performance relative to the fault detection effectiveness metric in favor of the call stack reduced test suites.

	Means and Paired-T Results Over 50 Paired Samples					
Size (Test Cases)	Function Reduced Size (Test Cases)	Function Reduced % Fault Detection Reduction	Call Stack Reduced Size (Test Cases)	Call Stack Reduced % Fault Detection Reduction	Random Reduced % Fault Detection Reduction	P-Value Order ($t_{49, 0.975} = 2.01$)
50	**18.8**	**18.9**	**21.1**	**6.9**	**21.4**	**10e-13**
100	**19.9**	**23.6**	**30.0**	**9.0**	**20.7**	**10e-10**
150	**20.2**	**26.4**	**36.4**	**9.7**	**21.9**	**10e-14**
200	**20.6**	**28.3**	**40.0**	**12.2**	**23.1**	**10e-9**
250	**20.1**	**31.6**	**42.8**	**13.1**	**25.0**	**10e-12**
300	**20.2**	**32.4**	**46.6**	**12.6**	**23.7**	**10-11**
350	**20.5**	**31.6**	**47.7**	**12.8**	**24.5**	**10e-11**
400	**20.8**	**31.6**	**49.8**	**13.1**	**23.5**	**10e-9**
450	**21.0**	**32.5**	**51.8**	**14.4**	**25.1**	**10e-10**
500	**20.5**	**33.0**	**51.3**	**13.2**	**23.6**	**10e-12**
550	**20.9**	**33.2**	**53.5**	**14.6**	**24.1**	**10e-10**
600	**21.1**	**33.5**	**54.1**	**14.0**	**24.9**	**10e-12**
650	**21.2**	**32.9**	**55.4**	**14.2**	**24.6**	**10e-10**
700	**20.3**	**36.5**	**55.8**	**14.9**	**25.4**	**10e-11**
750	**20.9**	**35.3**	**56.7**	**15.9**	**28.1**	**10e-13**
800	**21.4**	**33.3**	**57.4**	**12.8**	**27.3**	**10e-19**
850	**20.8**	**35.4**	**57.5**	**15.4**	**25.5**	**10e-12**
900	**21.0**	**35.6**	**57.8**	**14.9**	**27.9**	**10e-15**
950	**20.8**	**35.6**	**58.1**	**14.3**	**26.2**	**10e-12**
1000	**20.7**	**36.5**	**58.7**	**15.9**	**24.1**	**10e-10**

Table 2: Size and fault detection reduction for different sized suites

Discussion: Clearly, high percentage size reduction and low percentage fault detection reduction are desired when using any test suite reduction technique. However, evaluation of the actual values obtained for percentage size reduction and percentage fault detection reduction is subject to the environment in which reduction is to be employed. For example, in a development or maintenance scenario where faults are less critical, high percentage size reduction may be desired and a higher percentage fault detection reduction may be acceptable. On the other hand, if time is available to run relatively more test cases and/or fault detection is critical, one may want lower percentage size reduction in exchange for lower percentage fault detection reduction. These tradeoffs affect the applicability and effectiveness of different test suite reduction techniques in practice. Furthermore, we see indications in Figures 4 and 7 that beyond a certain point, coverage-based test suite reduction tends to yield similarly sized suites, and the "natural" reduced suite size varies by coverage criterion. Using this concept, it may be possible to empirically derive a taxonomy of coverage techniques in test suite reduction that would guide practitioners in making size and fault detection tradeoffs. Future work may explore this idea in more detail.

6. Related Work

Harder, Mellon, and Ernst [7] use dynamic invariant detection techniques [4] to minimize a test suite. While running a program, they maintain an "operational abstraction", which is a mathematical picture of the program's dynamic behavior. The "operational difference" technique applied to test suite reduction executes each test case in a suite in turn, and if a test case does not change the current operational abstraction of the program, it is discarded. Like call stack reduction (and unlike most other reduction techniques), this approach makes use of dynamic program behavior rather than syntax. However, it has significant performance overhead.

There have been a number of prior studies of the effects of test suite minimization on fault detection effectiveness. Wong *et al.* [16] minimize relative to the all-uses coverage criterion and observe little or no fault detection effectiveness reduction in the reduced suites. They also find a direct relationship between the ease of finding faults and the likelihood that they will be detected after minimization. In contrast, Rothermel *et al.* [12] minimize with respect to all-edges coverage and find significant reductions in fault detection effectiveness. They contrast their results with [16] and suggest possible causes for the different conclusions. However, collecting all-uses and all-edges coverage information generally requires invasive source code instrumentation, and the necessary tools may be difficult to obtain, set up and use for many programming languages. In constrast, call stack coverage information is relatively simple to obtain using tools that we will make available. Additionally, call stack coverage can be analyzed on any stack-based runtime environment, which encompasses most language and system combinations in practical use today.

Ball [3] introduces *concept analysis* as applied to the problem of test coverage. In this application domain, concept analysis relates tests to program entities such as procedures (i.e. functions), edges, or statements that the tests cover. Concepts of this type may be used to compute "dynamic control flow invariants" and dynamic analogies to the static analysis ideas of domination, postdomination, and regions. It would be straightforward to apply concept analysis to the test case minimization problem using a process similar to the one in this paper. Concept analysis of procedures covered by tests is similar to our call stack analysis in that they both consider procedures in the context of other procedures rather than in isolation. Call stack analysis is finer-grained in this sense because it takes into account the actual call chain whereas the concept analysis technique presented by Ball only tracks procedures covered by a test without considering their order.

7. Summary and Future Directions

In this paper, we described a new coverage criterion for use in test suite reduction based on the set of unique call stacks dynamically generated by the test suite. Call-stack based reduction provides a practical alternative to existing reduction techniques that require the use of sophisticated and expensive analyses and data-collection mechanisms. We gave formal definitions of relevant call stack concepts and applied them to create a call stack collection and reduction process. Finally, we empirically evaluated the call stack reduction process on edge coverage adequate suites and on randomly generated suites of different sizes, comparing our reduced suites to those generated by edge-coverage-based reduction and function-coverage-based reduction, respectively. We found that the call stack coverage criterion can produce favorable tradeoffs between the reduction in test suite size and reduction in fault detection effectiveness.

High-priority future work would be to apply the call stack minimization technique to a number of different applications, preferably of different sizes, from different domains, and written by different people. Future work should also evaluate the feasibility of call stack reduction in the presence of certain programming language features. In particular, we consider how the technique may be applied in multithreaded programs. Although we do not consider a multithreaded application in our experiment, multithreading can be handled by keeping separate sets of unique stacks for each thread and merging them either after threads exit or during a post-processing step.

Acknowledgements

Gregg Rothermel provided the `space` program and test artifacts. Portions of the `space` package were previously developed by Alberto Pasquini, Phyllis Frankl, and Filip Vokolos. This research was partially funded by a grant from the National Science Foundation (CCF0447864).

References

[1] GNU Compiler Collection (gcc) information on the web at http://gcc.gnu.org.

[2] G. Ammons, T. Ball, and J.R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. SIGPLAN '97 *Conf. on Programming Language Design and Implementation*, 1997.

[3] T. Ball. The concept of dynamic analysis. Proceedings of the Joint Seventh European Software Engineering Conference (ESEC) and Seventh ACM SIGSOFT International Symposium on the Foundations of Software Engineering, September 1999.

[4] M. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1-25, February 2001.

[5] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. *ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering,* Nov. 1998.

[6] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. Technical report PUCS-100-94, Department of Computer Science, Polytechnic University, Brooklyn, NY, February 1994.

[7] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. *Proceedings of the 25th International Conference on Software Engineering*, pp. 60-71, 2003, Porland, Oregon, United States.

[8] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* July 1993 Volume 2 Issue 3.

[9] G. Hunt and D. Brubacher. Detours: binary interception of Win32 functions. *Proceedings of the 3rd USENIX Windows NT Symposium*, pp. 135-143. Seattle, WA, July 1999.

[10] A. Memon, A. Porter. C. Yilmaz. A. Nagarajan. D.Schmidt and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. *Proceedings of the 26th International Conference on Software Engineering*, pages 459—468, 2004.

[11] M. Morisio, C. B. Seaman, A. T. Parra, V. R. Basili, S. E. Kraft and S. E. Condon. Investigating and improving a COTS-based software development. *Proceedings of the 22nd international conference on Software engineering*, pages 32-41, Limerick, Ireland, 2000.

[12] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault-detection capabilities of test suites. *Proceedings of the International Conference on Software Maintenance*, pages 34-43, November 1998.

[13] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. *Journal of Software Testing, Verification, and Reliability,* V. 12, no. 4, December, 2002.

[14] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization. *IEEE Transactions on Software Engineering* V. 27, no. 10, October, 2001, pages 929-948.

[15] D. Thomas and A. Hunt. Programming Ruby: The Pragmatic Programmer's Guide. Addison-Wesley, 2001.

[16] W. Eric Wong, Joseph R. Horgan, Saul London, Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. *Proceedings of the 17th International Conference on Software Engineering*, p.41-50, 1995, Seattle, Washington, United States.

[17] W. E. Wong, J.R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. *Proceedings of the 21st Annual International Computer Software and Applications Conference*, pages 522-528, August 1997.

[18] W. E. Wong, J. R. Horgan, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, pages 230-238, Monterey, CA, November 1994.