Effective and Efficient Localization of Multiple Faults Using Value Replacement

Dennis Jeffrey
Univ. of California, Riverside
jeffreyd@cs.ucr.edu

Neelam Gupta
guptajneelam@gmail.com

Rajiv Gupta
Univ. of California, Riverside
gupta@cs.ucr.edu

Abstract

We previously presented a fault localization technique called Value Replacement that repeatedly alters the state of an executing program to locate a faulty statement [9]. The technique searches for program statements involving values that can be altered during runtime to cause the incorrect output of a failing run to become correct. We showed that highly effective fault localization results could be achieved by the technique on programs containing single faults. In the current work, we generalize Value Replacement so that it can also perform effectively in the presence of multiple faults. We improve scalability by describing two techniques that significantly improve the efficiency of Value Replacement. In our experimental study, our generalized technique effectively isolates multiple simultaneous faults in time on the order of minutes in each case, whereas in [9], the technique had sometimes required time on the order of hours to isolate only single faults.

1 Introduction

The process of finding, understanding, and fixing software defects ("bugs" or "faults") is called *software debugging*. This is an important and necessary phase of software development, due to the prevalence of bugs in program source code. Software debugging helps to make software more robust and reliable.

An important phase of software debugging, *fault localization*, involves locating faulty program statements. Fault localization research aims to develop automated techniques to assist developers in finding bugs. *Dynamic Program Slicing* [2, 15, 21, 26, 27, 28] can compute a subset of program statements, likely containing a faulty statement, that directly or indirectly affects the erroneous output produced during a failing execution. *Delta Debugging* [4, 23, 24] analyzes differences in program state between successful and failing executions to isolate a bug. Other approaches [14, 16, 17] use runtime information to rank program statements according to likelihood of being faulty.

In our prior work [9], we describe a state-altering technique to locate program bugs called *Value Replacement*. This technique involves replacing the set of values used at a statement instance in a failing run with an alternate set of values, then checking to see whether the failing run changes as a result to become passing. The statement instances where this is true are likely to be associated with a fault. We showed [9] that the Value Replacement technique can achieve highly effective fault localization results using the Siemens benchmark programs [8]. These results were a substantial improvement over those achieved by the *Tarantula* statistical technique [14], which itself was shown [13] to yield better results on the Siemens benchmarks than several other prior approaches. However, one issue with Value Replacement is that the effectiveness of locating faults can be reduced when multiple simultaneous faults are present. This is because the technique considers multiple failing runs; when different runs fail due to different faults, this can make it more difficult to isolate the faulty statements, thereby decreasing the effectiveness of the technique.

In the current work, we show how to generalize the Value Replacement technique into an iterative technique that can effectively handle the situation when multiple faults are present in software. We first consider a minimal-computation approach in which Value Replacement is applied only once to rank program statements, and the search for all faults is performed within the single ranked list (relatively low cost, but relatively low effectiveness). We next consider a full-recomputation approach in which Value Replacement is iteratively invoked to find and fix one fault at a time (relatively high effectiveness, but relatively high cost). Finally, we consider a middle-ground, partial-recomputation approach in which only a part of the required computation for Value Replacement is performed on each iteration to find and fix a fault (lesser cost with effectiveness approximating that of full-recomputation).

A limitation of the core Value Replacement technique is that it may require considerable computation time. Our prior experiments [9] with the single-fault Siemens benchmarks showed that while many faults could be isolated within a matter of minutes, other faults required time on the order of hours to locate, due to relatively long execution traces of failing tests. In the worst case, one particular fault

required over 14 hours to locate. This efficiency issue becomes considerably more pronounced when *iteratively* handling multiple simultaneous faults. To address this concern, we describe two improvements that significantly speed-up the required computation time to locate faults using Value Replacement: (1) removing redundant execution when performing value replacements; and (2) parallelizing the task of performing value replacements. We noticed considerable improvement in the efficiency of the core Value Replacement technique when applying these two improvements: the situation mentioned above that previously required over 14 hours to locate a single fault, now requires only 27 minutes to achieve the same result. These improvements in efficiency have enabled us to generalize Value Replacement into an iterative technique that can address the issue of multiple faults. The main contributions of this paper are:

- A new technique (with three variants) that generalizes Value Replacement to promote *effectiveness* in the presence of multiple faults.
- Two techniques that significantly improve the *efficiency* of Value Replacement.
- An experimental study demonstrating the effectiveness and efficiency of our multiple-fault technique.

In the next section, we describe the core Value Replacement technique and its generalization to handle multiple faults. We then describe two techniques to significantly improve the efficiency of Value Replacement (Section 3). We present an experimental study (Section 4), related work (Section 5), and a summary of our conclusions (Section 6).

2 Value Replacement and its Generalization to Handle Multiple Faults

2.1 Single-Fault Core Technique

We previously presented the core Value Replacement technique for fault localization [9]. The technique involves performing *value replacements* in order to search for *interesting value mapping pairs* in a failing run. A *value replacement* is the act of replacing the set of values used at a statement instance in a failing run with an alternate set of values, then letting execution proceed from that point to observe the effect on the program output. If the output changes to become correct, then an *interesting value mapping pair* (IVMP) is identified and associated with the statement instance, indicating the original and alternate sets of values at that point that caused the failing run to become passing. Alternate sets of values are selected from profiling information taken from the executions of all test cases in an available test suite. Different alternate value sets are used to perform value replacements in each statement instance in each failing run. The key idea of Value Replacement is that the statements associated with IVMPs in more failing runs, are more likely to be faulty. A *suspiciousness* value is associated with each statement exercised by a failing run, which is simply the number of failing runs in which that statement is associated with at least one IVMP. Statements are then ranked in decreasing order of this suspiciousness.

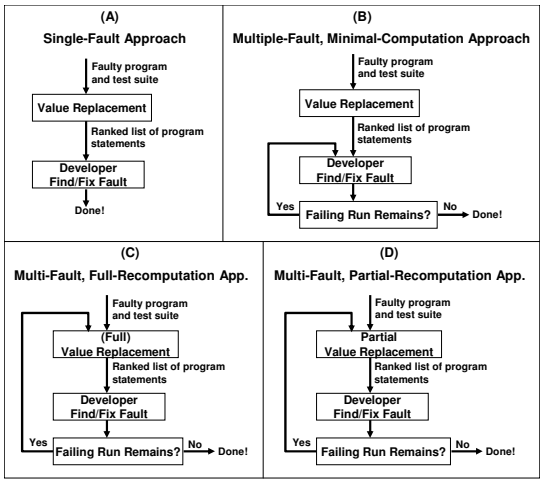

Figure 1. Single-fault core technique (A) and multi-fault generalized techniques (B-D).

We showed [9] that Value Replacement can achieve significantly better fault localization results on the Siemens benchmark programs [8] than another statistical approach called Tarantula [14]. Moreover, Tarantula had been previously shown [13] to be more effective on the same benchmark programs than several other prior approaches. However, the benefits of Value Replacement come at the expense of significantly increased computation time. We address the issue of efficiency in Section 3.

Value Replacement implicitly assumes that all failing runs fail due to a single fault. In the event that runs fail due to different faults, this can lead to faulty statements potentially having lower suspiciousness than other non-faulty statements, limiting the effectiveness of the technique.

2.2 Multi-Fault Generalized Techniques

In the single-fault Value Replacement technique, a ranked list of program statements is computed and presented to a developer. The developer can then search the reported statements in order until the faulty statement is found. Figure 1(A) depicts this scenario. To handle multiple simultaneous faults, we generalize the technique to iteratively present the developer with ranked lists of program statements, each time allowing the developer to find and fix a fault. After each iteration, a new ranked list may be computed as long as at least one failing run remains, so that remaining faults can be located. We consider three variations of the multiple-fault generalized Value Replacement technique. First, we consider a minimal-computation approach where Value Replacement is performed only once, and the developer uses the single ranked list of statements to search

```
input:
    Faulty program P, and test suite T containing at least one
    failing run.
output:
    Program P' such that no tests in T fail.
algorithm MinimalComputationApproach
begin
1:  rankedList := Run_Value_Replacement(P, T);
2:  P' := P;
3:  while ∃ a failing run in T wrt. P' do
4:      F := locate next fault using rankedList;
5:      P' := version of P after fixing F;
    endwhile
end MinimalComputationApproach
```

Figure 2. Minimal-computation approach to locate multiple faults.

for faulty statements as needed (Figure 1(B)). Second, we consider the opposite extreme: a full-recomputation approach in which we perform (regular) Value Replacement, allow the developer to find and fix a fault, then repeat this process on the new version of the program as necessary (Figure 1(C)). Finally, we consider a partial-recomputation approach that performs only partial Value Replacement computation on each iteration (Figure 1(D)).

2.2.1 Minimal-Computation

The minimal-computation approach is presented in Figure 2. The approach takes as input a faulty program and a corresponding set of test cases containing at least one failing run. First, the Value Replacement approach is executed to obtain a ranked list of program statements (line 1). Then, as long as a failing run exists in the test suite with respect to the current version of the program (line 3), the ranked list is used to locate a fault in the program (line 4), which is then fixed (line 5), resulting in a new version of the program with the fault removed. The actual identification and fixing of a faulty statement is done manually by a developer. If at least one run still fails on the new version of the program, then the (same) ranked list is consulted again to find and fix another fault (back to line 3). Under this approach, the computation time is expected to be comparable to that of the original single-fault Value Replacement technique, since a ranked list of program statements need be computed only once. However, the drawback is that the average fault localization effectiveness for each individual fault may be reduced, since the ranked list is never updated as the program is modified and faults are corrected over time.

2.2.2 Full-Recomputation

The full-recomputation approach is identical to the minimal-computation approach, except the invocation of the Value Replacement technique has been moved to *inside* the main loop (in Figure 2, this involves moving line 1 in-between lines 3 and 4). The effect is that a revised ranked list is computed on each iteration when a fault is found and fixed. This ensures that the approach has up-to-date data that can be used to compute a more effective ranking on each iteration. However, the timing requirements increase significantly because Value Replacement must be invoked on every iteration to search for new IVMPs.

```
input:
    Faulty program P, and test suite T containing
    at least one failing run.
output:
    Program P' such that no tests in T fail.
algorithm PartialRecomputationApproach
begin
Step 1:    [Compute ranked lists and find/fix first fault]
1:  RankedLists := {};
2:  Stmt_fail := stmts exercised by failing runs in T;
3:  for each stmt s ∈ Stmt_fail do
4:      F := failing runs in T which do not exercise s;
5:      rankList := Run_Value_Replacement(P, T − F);
6:      RankedLists := RankedLists ∪ {rankList};
    endfor
7:  selectedList := removeFirstList(RankedLists);
8:  F := locate next fault using selectedList;
9:  faultyStmt := the stmt containing F;
10: P' := version of P after fixing F;
Step 2:    [Revise ranked lists and find/fix next fault]
11: while ∃ a failing run in T wrt. P' do
12:     Fail := failing runs in T exercising faultyStmt
            with respect to P';
13:     compute IVMPs for each run in Fail wrt. P';
14:     update any affected lists in RankedLists;
15:     selectedList := removeNextList(RankedLists);
16:     F := locate next fault using selectedList;
17:     faultyStmt := the stmt containing F;
18:     P' := version of P after fixing F;
    endwhile
end PartialRecomputationApproach
```

Figure 3. Partial-recomputation approach to locate multiple faults.

2.2.3 Partial-Recomputation

The partial-recomputation approach is presented in Figure 3. This approach consists of two main steps. First, a set of ranked lists is computed, and they are used to find and fix a first fault in the program. Second, the approach iteratively performs partial Value Replacement re-computation, updates any affected ranked lists, and then uses the revised ranked lists to find and fix a next fault.

Step 1: Initialize ranked lists and handle the first fault (lines 1-10). In this step, the approach first collects together all statements exercised by failing runs to consider for ranking purposes (line 2). Next, for each of these statements s, a ranked list of program statements is computed using Value Replacement by searching for IVMPs in only those failing runs exercising s (lines 3-6). The intuition for

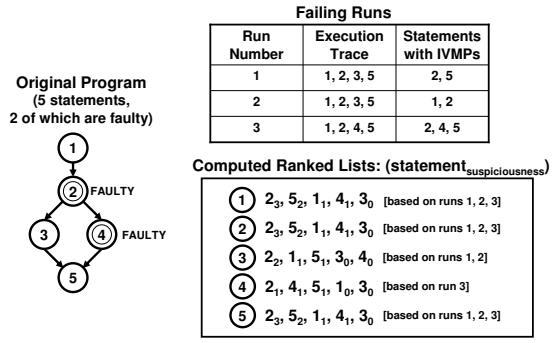

Figure 4. Abstract example for running the partial-recomputation approach, part 1 of 2.

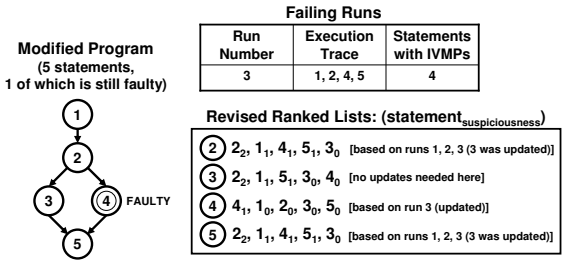

Figure 5. Abstract example for running the partial-recomputation approach, part 2 of 2.

this step is as follows: we know at least one of the statements s is faulty, and we are most likely to achieve maximum suspiciousness for such a statement if we rank statements based on the IVMP information of only those failing runs which exercise s; since we do not know which statements are faulty, we compute a ranked list for each considered statement. Next, from among the set of ranked lists, we select and remove one of them to be used to locate the first fault (line 7). This is determined by choosing the list in which the statement at the front of the ranked list has highest suspiciousness (look at subsequent statements to break ties). This list is most likely to quickly lead a developer to the first fault. The developer uses the selected ranked list to find the first fault (lines 8-9) and fix it (line 10).

Step 2: Iteratively revise ranked lists and handle the next fault (lines 11-18). The second step iterates as long as a failing run still exists (line 11). First, in the new version of the program, the approach identifies the set of failing runs exercising the faulty statement that was just fixed (line 12). For only these failing runs (not for all failing runs), IVMPs are re-computed (line 13). Note that this step only searches for IVMPs in a subset of failing runs; it does not actually compute suspiciousness values and a ranked list as is done in the call to Run_Value_Replacement at line 5. Based on the updated IVMP information, any affected ranked lists from the set of maintained lists are updated so that their rankings may change (line 14). A ranked list is considered affected if it was computed using one of the failing runs for which new IVMPs were just computed. Then, using the revised set of ranked lists, the next one to use for fault localization purposes is selected and removed (line 15). Note that the selection criterion here is different than in line 7. In this case, we select the ranked list for which the statements near the front of the list are the most *different* from those statements near the fronts of the previously-selected lists. We compute this by setting a difference threshold value D, and then we scan all ranked lists in order in parallel, selecting the first ranked list that achieves D different statements as compared to those statements in the previously-selected lists (we used $D = 10$ in our experiments since that gave good results on our subject programs). The intuition for this criterion is that a different fault is likely to have a different set of statements with high suspiciousness, than for those faults already found and fixed. The selected list is finally used to locate (lines 16-17) and fix (line 18) the next fault.

Example. We demonstrate the partial-recomputation approach with a small example. Figure 4 shows an example control-flow graph of a program containing 5 statements, 2 of which happen to be faulty. Suppose a test suite is available that contains 3 failing runs as depicted in the figure, with associated execution traces and sets of statements containing IVMPs as shown (IVMPs were computed as described in [9] and summarized in Section 2.1). In this case, two of the runs fail due to faulty statement 2, and one of them fails due to faulty statement 4. In the first step, we identify the set of statements exercised by failing runs (all statements in this case). Next, we compute a ranked list to associate with each one of these statements, by ordering statements according to suspiciousness value. Recall that the suspiciousness value is the number of (considered) failing runs in which the associated statement has an IVMP. The 5 computed ranked lists for our example are shown in Figure 4. Each of these ranked lists is computed using the IVMP information from only those failing runs exercising the statement that is associated with the ranked list. For example, the ranked list associated with statement 4 is computing using only run 3 (only run 3 exercises statement 4).

Next, the approach identifies the first ranked list (from among the 5 computed lists) to remove and report to a developer. This is the one with highest suspiciousness values at the front of the list. Ranked lists 1, 2, and 5 have the first ranked element with highest suspiciousness; since the lists are identical (no ties can be broken), an arbitrary choice is made from these lists. Suppose list 1 is selected, removed, and reported to the developer. Then faulty statement 2 is immediately identified because it occurs at the front of the selected list; the developer can then fix this faulty statement.

Figure 5 shows how the situation might look after faulty statement 2 is fixed. In this case, statement 4 is the only remaining faulty statement. Assume that run 3 is the only run that still fails. Further assume that on the new version of the program, run 3 is associated with an IVMP

at only statement 4. Next, the second step of the partial-recomputation approach is executed. First, the approach identifies which subset of newly-failing runs need to be re-searched for IVMPs. In our example, failing run 3 exercises statement 2 (the most recently-fixed statement), so run 3 must be re-searched for IVMPs. In practice, not all failing runs may need to be re-searched for IVMPs in this step. Next, from among the remaining ranked lists, only lists 2, 4, and 5 are affected by the new IVMPs and need to be updated (list 3 was not originally computed using run 3). In the original version of the program, run 3 was associated with IVMPs at statements 2, 4, and 5. However, in the new version of the program (with corrected statement 2), run 3 is associated with IVMPs at only statement 4. Thus, ranked lists 2, 4, and 5 are updated to reflect a decrease of 1 in the suspiciousness values for statements 2 and 5 (shown in Figure 5). Now, the next ranked list to remove and report to the developer is selected. In this case, we select the ranked list from among those remaining, that is most different from the first-selected ranked list, in terms of the elements near the fronts of the lists. Since the first-selected ranked list had started with statement 2, then from among the remaining lists, list 4 is the most different because it is the only remaining list that does not also start with statement 2. Thus, ranked list 4 is selected. This allows the developer to immediately fix faulty statement 4 (since it appears at the front of the ranked list). At this point, no failing runs remain since all faults are fixed, and the approach terminates. Overall in this example, two ranked lists were selected and reported to the developer, each list accurately identifying one of the faults with highest suspiciousness.

3 Value Replacement: Improving Efficiency

The efficiency of the Value Replacement fault localization technique is determined by the search for IVMPs in each failing run. Searching for IVMPs can be time-consuming because it is essentially a testing approach where many program executions are performed. On each execution, the state of the executing program is changed at some point (where the value replacement is performed), and then the effect on the program output is observed. Moreover, this must be performed once for each alternate set of values to try at each statement instance, and at each statement instance in multiple failing runs.

We make two observations about the search for IVMPs that allow us to significantly speed-up the search time: (1) within a failing run, the search for IVMPs involves a significant amount of redundant program execution; (2) each value replacement can be performed in isolation, and thus the search for IVMPs is inherently parallelizable.

Removing redundant program execution. There is significant redundant program execution in the search for IVMPs within a failing run. This is because in a value

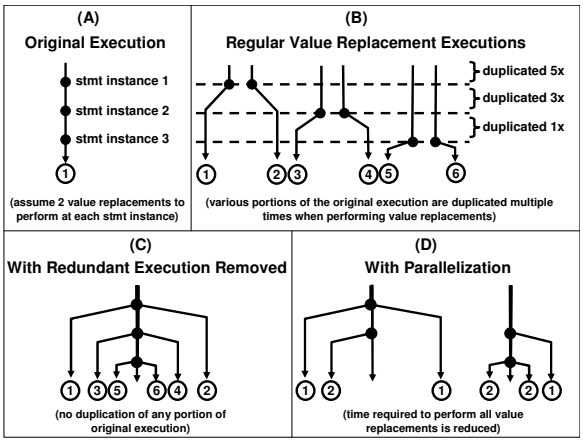

Figure 6. Improving efficiency of Value Replacement. The circled numbers indicate the relative time at which execution terminates.

replacement program execution, the part of the execution *before* the value replacement is the same as in the original failing run (execution is affected only from the point of the value replacement onwards). Many different value replacement executions in the same failing run therefore leads to a significant amount of redundant execution. This is illustrated in Figure 6. In Figure 6(A), an original program execution is shown with 3 statement instances; assume 2 value replacements are performed at each instance, for a total of 6 value replacements. Figure 6(B) shows each of these 6 value replacement executions, along with the duplicated portions of the original execution.

To remove this redundant execution, we require a mechanism that allows for the following: at a statement instance in a failing run at which we need to perform a value replacement, we need to be able to perform the value replacement and then *return directly to the same point*, without re-executing everything before this point, so that we can continue with additional value replacements. In this way, we avoid all redundant execution prior to the point of a value replacement. The `fork()` function in `C` can provide us with this functionality.

Our method for removing redundant program execution is as follows. We execute the failing run once, from beginning to end. At each statement instance during execution at which we need to apply one or more value replacements, we invoke `fork` to create a child process to carry out each value replacement. When the `fork` function call occurs, a new child process is created which is identical to the parent (original) process, except for a new process ID. The parent process then waits at that statement instance as each child completes its value replacement. Afterwards, the parent process moves onto the next statement instance, where more children may be forked to perform more value replacements. This eliminates all redundant portions of execution, as depicted in Figure 6(C).

We had to specially handle the input and output of the parent and child processes in our implementation to ensure they are not affected by the forking, because `fork` duplicates file pointers, which can cause intermixed input/output. This was done by initially capturing all the input and output of the original parent process (representing the original failing execution). When a child process was forked, then its set of input and output was specifically adjusted (such as by setting new file pointers as necessary) to match that of the parent at that point. Reads from "standard-in" were handled by writing these input values to a file after the initial input/output capturing phase, then just reading from this file as necessary during the forking/value replacement phase.

Parallelizing the search for IVMPs. Each time a value replacement is performed when searching for IVMPs, it can be done in isolation from all other value replacements. Thus, multiple available cores can carry out multiple value replacements in parallel. To take advantage of this, we parallelize the search for IVMPs in two ways. First, we partition the set of all value replacements to perform into N task sets, where N is the number of available cores. Then each task set is handed off to an available core, which performs the specified set of value replacements (the IVMP search results from all cores are merged together in the end). Second, when a parent process forks C children to perform C value replacements at a statement instance during execution, then those C children are simply allowed to execute in parallel. This parallelization is illustrated in Figure 6(D). In the figure, the 6 value replacements are partitioned into 2 task sets. The figure also assumes that enough idle cores are available to process all children in parallel at each statement instance. The circled numbers in Figure 6(D) show multiple value replacement executions terminating at the same relative time unit; this is due to multiple value replacements being performed in parallel.

To illustrate the benefit of our efficiency improvements, in our prior work [9], the worst-case situation required over *14 hours* to perform all the necessary value replacements to locate a fault. With the efficiency improvements described above, that same result can be achieved in only *27 minutes*. However, the focus in the current work is on locating multiple simultaneous faults (not single faults as in [9]), so the improvements in efficiency are tempered by the iterative nature of the current multiple-fault approach.

4 Experimental Study

4.1 Setup

Implementation. Our implementation of the core Value Replacement fault localization technique is based on the *Valgrind* infrastructure [7, 19] that allows for dynamic binary translation of an executing program. We created a tool to work in conjunction with Valgrind that performs value replacements to search for IVMPs. On top of this, we used `Java` to implement the three variants of our extended technique for handling multiple faults.

Prog. Name	LOC	# Faulty Versions	# Test Cases	Program Description
tcas	138	41	1608	altitude separation
totinfo	346	23	1052	statistic computation
sched	299	9	2650	priority scheduler
sched2	297	9	2710	priority scheduler
ptok	402	7	4130	lexical analyzer
ptok2	483	9	4115	lexical analyzer
replace	516	31	5542	pattern substituter

Table 1. The Siemens benchmark programs.

Our experiments were conducted using a Dell PowerEdge 1900 server with two Intel Xeon quad-core processors at 3.00 GHz, with 16 GB of RAM. Because of the multiple cores available, we implemented both of the efficiency improvements (removing redundant execution, and parallelization) described in Section 3.

Subject programs and test suites. Our experimental subjects are based on the Siemens benchmark programs and test cases [8] described in Table 1. Each of the Siemens programs are associated with a set of faulty versions – each containing a single seeded fault – and a pool of test cases.

For our experiments, we require a set of programs containing multiple faults. To create a set of multiple-fault faulty versions for each subject program, we randomly selected from among the available seeded faults to create faulty versions that each contain 5 seeded faults. We ensured that each fault in a multiple-fault version is contained in a different statement. We created up to 20 unique 5-fault faulty versions for each subject program, as permitted by the set of available faults. We were only able to create 2 and 11 such versions for programs `ptok` and `ptok2` respectively, due to a limited number of available faults, some of which conflicted by being located in the same statement and could not be incorporated into the same faulty version.

For each multiple-fault version of each subject program, we created a test suite by selecting tests randomly from the associated test case pool until the following criteria were achieved: (1) the suite is statement-coverage adequate (achieving the same statement coverage as the test case pool); (2) for each faulty statement present in the multiple-fault version, there exists a failing run in the suite exercising that statement; and (3) there are at least 5 failing runs and 5 passing runs in the suite (to ensure a good mix of failing and passing runs). Table 2 shows the number of multiple-fault faulty versions created for each subject program, as well as the average test suite size associated with each one.

Approaches and metric for comparison. We compare the fault localization effectiveness and efficiency of the following generalized Value Replacement techniques to handle multiple simultaneous faults.

Prog. Name	# 5-Fault Faulty Versions	Avg. Test Suite Size (# Fail Runs/# Pass Runs)
tcas	20	11 (5 / 6)
totinfo	20	22 (10 / 12)
sched	20	29 (10 / 19)
sched2	20	30 (9 / 21)
ptok	2	32 (8 / 24)
ptok2	11	29 (5 / 24)
replace	20	38 (9 / 29)

Table 2. Multiple-fault experimental subjects.

1. Minimal-computation approach (MIN). The original Value Replacement technique is applied only once to obtain a single ranked list of program statements, which is then consulted as necessary until all faults are found.

2. Full-recomputation approach (FULL). The original Value Replacement technique is iteratively applied to find each fault, one at a time.

3. Partial-recomputation approach (PARTIAL). Multiple ranked lists of program statements are computed and iteratively revised through partial recomputation of IVMPs (from only a subset of failing runs) to find each fault.

4. Ideal situation (IDEAL). We consider the "ideal" situation for finding each fault to be the case where that fault exists in isolation in a program (no other faults present). This situation is most likely to lead to the best fault localization results for each fault, when using Value Replacement. These "ideal" single-fault results are used to compare against the above 3 techniques that handle multiple faults.

To compare the results of each approach, we assign a *score* to each located faulty statement. The score is based upon the position of that statement within the ranked list of statements in which it is found. It represents the percentage of program statements in the ranked list that *need not be examined* if statements are examined in rank order. Let f be a faulty statement, and assume that faulty statement occurs at rank r in a ranked list containing $totalStmts$ total statements. Then the *score* of faulty statement f can be defined as follows.

$$score(f) = \frac{totalStmts - r}{totalStmts} \times 100\% \qquad (1)$$

The ideal situation occurs when the faulty statement has rank value 1. A higher score is preferable because it means that more program statements need not be examined to locate the fault. In the event that ties occur in the ranked list, all tied statements are given a rank value equal to the maximum rank value from among the tied statements. For example, if there are 5 statements tied for highest rank, then all 5 of them are given rank 5. This allows us to conservatively assume that we would have to examine all tied statements before any faulty statement within that tied set can be found.

Prog. Name	Average Score (%)			
	IDEAL	FULL	PARTIAL	MIN
tcas	83.64	82.87	77.98	68.82
totinfo	64.45	63.14	60.29	52.78
sched	88.56	88.28	85.13	84.72
sched2	64.75	58.15	56.81	56.20
ptok	76.28	70.14	65.55	59.00
ptok2	89.62	84.37	84.13	80.69
replace	88.17	86.98	86.50	76.27

Table 3. Average score achieved for each located fault using each approach.

4.2 Results and Discussion

Effectiveness. Table 3 shows the average score values achieved for each fault from among all individual faults contained within the multiple-fault versions associated with each subject program. As shown in the table, the FULL approach is able to achieve average score values that are very close to the IDEAL values in most cases (within one or two percentage points). The exceptions are programs `sched2`, `ptok`, and `ptok2`, in which the FULL approach achieves average results that are about 5% – 6% less than the IDEAL results. We found that for these three programs that contain relatively few distinct faults (shown in Table 1), there were a small number of particular faults in which IVMPs could not be found at the faulty statements, thus resulting in poor ranking results. Since these "problem" faults were repeatedly selected from a relatively small set of total faults, they were present in relatively many of the multiple-fault faulty versions for these programs, affecting the average results.

In all cases, the PARTIAL approach is able to achieve average score values that are within 5% of the FULL approach. In some cases, the difference is quite small. For example, in the `replace` program, the FULL approach has an average score of 86.98%, while the PARTIAL approach yields almost the same average score: 86.50%. For `ptok2`, FULL has an average score of 84.37% while PARTIAL has 84.13%. This suggests that PARTIAL may be effective at approximating the effectiveness of FULL in certain cases.

The MIN approach has the lowest average scores in all cases. When compared to the PARTIAL approach, the MIN results are still sometimes considerably lower. For example, in program `tcas`, the MIN approach yields an average score of 68.82%, which is about 9% less than PARTIAL, 14% less than FULL, and 15% less than IDEAL. For `replace`, MIN yields an average score that is about 10% less than that achieved by PARTIAL.

Table 3 suggests that if effectiveness is the primary concern when locating multiple faults in software, the FULL and PARTIAL approaches may be the best choices. However, the FULL approach may be prohibitively time-consuming in some cases. In these situations, the PARTIAL approach may be preferable to achieve better running time.

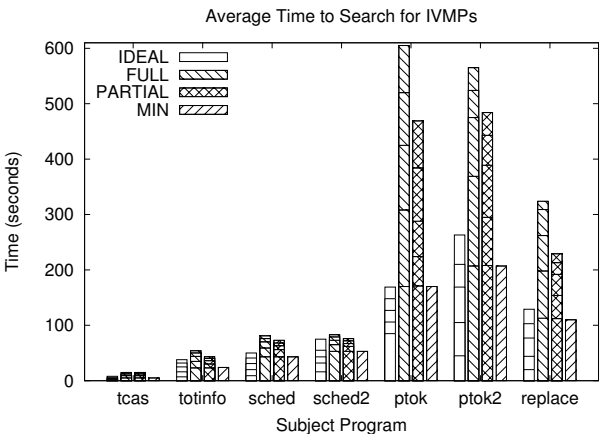

Figure 7. Avg. total time to search for IVMPs.

Efficiency. Figure 7 shows the total time (in seconds) required to search for all IVMPs when running each of the four fault localization approaches. For each subject program, the displayed timing data is the average from among all faulty versions associated with the program. Each bar is stacked to show the average time required for each iteration of the approach (except MIN, in which IVMPs are computed in only one iteration).

It can be seen in Figure 7 that the FULL approach requires more time overall than PARTIAL, which requires more time than MIN. In general, the time required by PARTIAL is slightly closer to the time for FULL as opposed to the time for MIN. This is not surprising, considering that only FULL and PARTIAL iteratively compute IVMPs (though PARTIAL may search fewer total failing runs than FULL). However, in some cases the overall time required by PARTIAL is noticeably less than FULL. In `ptok`, FULL requires about 600 seconds (10 minutes) on average to search for IVMPs in each faulty version, whereas PARTIAL requires only about 470 seconds (under 8 minutes), a 20% reduction in running time. For `replace`, FULL requires about 5.5 minutes while PARTIAL requires about 3.8 minutes, a 30% reduction in running time. Since the effectiveness of PARTIAL is similar to that of FULL (especially for `ptok2` and `replace`), then PARTIAL may be useful in situations where running time is an issue. Note also that the timing results for IDEAL are generally less than for FULL and PARTIAL; this is due to the programs being single-fault versions in IDEAL, with relatively fewer failing runs exposing these faults that need to be searched for IVMPs.

Table 4 shows the actual number of failing runs that are searched for IVMPs in each iteration, on average for each faulty version in a subject program. For example, for the FULL approach in program `replace`, the first iteration required a search for IVMPs in an average of 9 failing runs, the second iteration required a search in an average of 7 failing runs, and so forth, such that an average of 26 runs needed to be searched in total. We leave out the column for approach MIN because the results are identical to the results of the first iteration for approaches FULL and PARTIAL.

Prog. Name	Average Number of Runs to Search for IVMPs		
	IDEAL	FULL	PARTIAL
tcas	1+1+2+2+2=8	5+4+3+2+1=15	5+4+3+2+1=15
totinfo	3+4+4+3+3=17	10+5+4+3+2=24	10+3+3+2+1=19
sched	2+3+3+2+2=12	10+4+3+2+1=20	10+3+2+1+1=17
sched2	5+5+3+4+6=23	9+4+3+2+1=19	9+3+2+2+1=17
ptok	4+1+1+1+1=8	8+7+6+5+4=30	8+2+3+5+4=22
ptok2	1+1+2+1+1=6	5+4+3+1+1=14	5+2+2+1+1=11
replace	2+2+3+2+2=11	9+7+5+4+1=26	9+3+3+2+1=18

Table 4. Average number of runs searched for IVMPs (separated by iteration number).

The results in this table mirror the timing results from Figure 7; PARTIAL requires IVMP searches in fewer total runs than FULL. MIN requires IVMP searches in the fewest number of runs. An interesting observation to make from this table is that as the iteration number increases for both the FULL and PARTIAL approaches, the total number of failing runs that must be searched for IVMPs tends to decrease. This is because as faults are iteratively found and fixed over time, fewer failing runs will tend to remain.

Other points of discussion. The programs used in our experiments, and their associated faults and test cases, were obtained from other researchers [8] and may not be representative of programs and faults that typically occur in practice. In the future, we hope to conduct further empirical evaluation of the effectiveness and efficiency of our technique when applied to larger, more realistic programs.

Despite our improvements regarding the efficiency of Value Replacement, the ability of our approach to scale to larger programs and execution traces remains an important issue: when searching for IVMPs, our approach still considers multiple value replacements in multiple statement instances in multiple runs. We observe that the improvements described in the current work (Section 3) are *lossless*, in the sense that no IVMPs will be missed by applying these improvements. However, there are other *lossy* techniques [9] not considered in this work, that we believe have potential to considerably improve efficiency without significantly degrading the quality of the fault localization results (due to a small number of missed IVMPs). This includes techniques to limit the number of statement instances to search for IVMPs (e.g., consider only those statement instances in the dynamic slice of the incorrect output values), and techniques to limit the number of alternate value sets to consider for value replacements (e.g., consider a subset of alternate values that still span the range of alternate values). We believe that there is still much opportunity to dramatically improve the scalability of Value Replacement in the future.

4.3 Comparison to Clustering

In the paper "Debugging in Parallel" [12], Jones et. al. describe a framework for *parallel debugging* in the presence of multiple faults. In this work, failing runs are clustered according to one of two proposed clustering techniques, and

Prog. Name	Average Score (%)				
	T=0.1	T=0.3	T=0.5	T=0.7	T=0.9
tcas	69.43	69.97	67.82	67.68	59.59
totinfo	54.23	56.37	56.89	56.84	54.66
sched	91.29	91.97	88.96	88.54	88.45
sched2	55.39	55.32	53.79	47.28	42.30
ptok	57.25	57.25	58.38	60.50	59.50
ptok2	80.78	80.64	80.91	78.94	78.13
replace	77.85	77.62	76.26	66.68	63.26

Table 5. Average score value achieved for each located fault using clustering (for different threshold values T).

then used to create specialized test suites that are targeted to a single fault. To study the effect of clustering, we implemented one of their proposed techniques ("Technique 2" in [12], selected based on ease of implementation). In this clustering technique, each individual failing test case in an available test suite is used to compute a suspiciousness ranking. These suspiciousness rankings are then checked against each other for similarity using a metric called *Jaccard Set Similarity*, which is defined as the ratio of the sizes of the intersection and the union between two sets. The failing runs associated with the suspiciousness rankings that are considered "similar" to each other are then clustered together. Each cluster of failing runs is then used to compute a ranked list of program statements that targets a particular fault. To determine whether two suspiciousness rankings are "similar" or not, a threshold value between 0 and 1 is set such that a set-similarity value greater than or equal to that threshold is considered to be "similar".

The above clustering technique is general and can be used with any fault localization approach that computes a suspiciousness ranking. We conducted an experiment where we performed the above clustering technique in conjunction with the Value Replacement technique for computing suspiciousness rankings. The computed clusters were then used to compute ranked lists of program statements that could be used to locate faults. Table 5 shows these results, on average for each subject program. The table shows the results for different similarity threshold values T, which guide how the clusters are formed and can therefore have a significant impact on the overall fault localization results.

From this table, it can be seen that the average score values are generally significantly lower for the clustering technique than for the FULL and PARTIAL approaches as shown previously in Table 3. One exception is for program `sched`, in which the clustering technique is able to achieve slightly higher average score values for the lower threshold values T. Overall, however, these results suggest that the approaches described in the current work, which are aimed specifically at improving the efficiency and effectiveness of Value Replacement in the context of multiple faults, may be preferable to the more general clustering technique when applied in conjunction with Value Replacement. However, the benefit of clustering is that it allows for debugging of multiple faults in parallel, whereas the techniques proposed in the current work are iterative in nature, meant to isolate only one fault at a time.

5 Related Work

The problem of fault localization has been extensively studied, but most techniques are not explicitly designed to address the issue of multiple faults in software. Slicing-based approaches compute subsets of program statements that directly or indirectly influence the value of a variable at some point in a program. *Static Slicing* [22] identifies a conservative subset of program statements that may cause such an influence, based upon the program source code. *Dynamic Slicing* was proposed [2, 15, 21] to identify a more precise subset of statements in the context of an execution of the program. Recent work on Dynamic Slicing [5, 26, 27, 28] has improved its efficiency and efficacy for debugging. To take potential influences into account, the related concept of *Relevant Slicing* [3, 6] has been studied.

Statistical approaches for fault localization [14, 16, 17] use dynamic information obtained from test case executions to rank program statements according to likelihood of being faulty. Jiang and Su [11] proposed a context-aware approach that constructs faulty control flow paths linking bug predictors together, to better describe where and how bugs occur. The *Nearest Neighbor* approach [20] searches for a correct execution that is most similar to an incorrect execution, compares the spectra for these two executions, and identifies the most suspicious parts of a program.

State-altering approaches modify the state of an executing program to isolate bugs. In *Delta Debugging*, failure-inducing input is identified [24] that allows for the computation of cause-effect chains for failures [23] that can be linked to faulty code [4]. This is accomplished by swapping the values of variables between a successful and failing run. Misherghi and Su [18] proposed an improved Delta Debugging algorithm for minimizing failure-inducing inputs. *Predicate Switching* [25] attempts to isolate erroneous code by identifying predicates whose outcomes can be altered during a failing run to cause it to become passing. *Execution Suppression* [10] is a technique that iteratively isolates memory corruption during a failing execution to identify the root cause of a memory-related failure. *Value Replacement* [9], discussed extensively in the current work, is also a state-altering technique.

A few techniques have been proposed for handling multiple faults. Abreu et. al. [1] describe a dynamic model-based approach that can derive explanations for multiple potential faults in software. Jones et. al. [12] describe a framework for *parallel debugging* (discussed in Section 4.3). Their work enables the use of parallel work flows to debug dif-

ferent faults simultaneously, thereby reducing the time-to-release of a program. Our current work differs in that it allows for iterative debugging and simply seeks to achieve the best localization results for each fault.

6 Conclusions

In this paper, we presented a generalized version of our Value Replacement technique for fault localization [9] that has been extended to perform effectively in the presence of multiple faults in software. The overall approach is to iteratively compute a ranked list of program statements such that each ranked list can guide a developer to some fault in the program as quickly as possible. We presented three variants of this approach that have different effectiveness and efficiency tradeoffs. We have also described two techniques that significantly decrease the computation time required by Value Replacement. These efficiency improvements enhance the applicability and usefulness of our generalized Value Replacement technique that handles multiple faults. In our experimental study, the total time required to locate the multiple faults in our benchmark programs was on the order of minutes in the worst case, among all faulty versions. This is a considerable improvement over our previous study [9] in which it took hours in the worst cases to isolate only single faults.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable feedback. This research is supported by NSF grants CNS-0751961, CNS-0751949, CNS-0810906, and CCF-0753470 to UC Riverside.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. A dynamic modeling approach to software multiple-fault localization. *Proc. of the 19th International Workshop on Principles of Diagnosis*, pages 7–14, September 2008.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. *Proc. of the ACM SIGPLAN '90 Conf. on Programming Language Design and Impl.*, pages 246–256, June 1990.

[3] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. *IEEE International Conf. on Software Maintenance*, pages 348–357, September 1993.

[4] H. Cleve and A. Zeller. Locating causes of program failures. *27th Intl. Conf. on Soft. Eng.*, pages 342–351, May 2005.

[5] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. *Intl. Conf. on Automated Software Engineering*, pages 263–272, November 2005.

[6] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. *Foundations of Software Engineering*, pages 303–321, September 1999.

[7] `http://valgrind.org`.

[8] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow and controlflow-based test adequacy criteria. *Proc. of the 16th International Conf. on Software Engineering*, pages 191–200, May 1994.

[9] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. *International Symposium on Software Testing and Analysis*, pages 167–178, July 2008.

[10] D. Jeffrey, N. Gupta, and R. Gupta. Identifying the root causes of memory bugs using corrupted memory location suppression. *Intl. Conf. on Software Maintenance*, pages 356–365, September 2008.

[11] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. *Intl. Conf. on Automated Soft. Eng.*, pages 184–193, November 2007.

[12] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. *International Symposium on Software Testing and Analysis*, pages 16–26, July 2007.

[13] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *Intl. Conf. on Automated Soft. Eng.*, pages 273–282, November 2005.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. *International Conf. on Software Engineering*, pages 467–477, May 2002.

[15] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[16] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. *Conf. on Programming Language Design and Impl.*, pages 15–26, June 2005.

[17] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. SOBER: Statistical model-based bug localization. *10th European Software Engineering Conf. held jointly with 13th Intl. Symp. on Foundations of Soft. Eng.*, pages 286–295, September 2005.

[18] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. *Proc. of the 28th International Conference on Software Engineering*, pages 142–151, May 2006.

[19] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Conf. on Prog. Lang. Design and Impl.*, pages 89–100, June 2007.

[20] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. *Intl. Conf. on Automated Software Engineering*, pages 30–39, October 2003.

[21] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[22] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[23] A. Zeller. Isolating cause-effect chains from computer programs. *10th International Symposium on the Foundations of Software Engineering*, pages 1–10, November 2002.

[24] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[25] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. *Intl. Conf. on Software Engineering*, pages 272–281, May 2006.

[26] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. *Conf. on Programming Language Design and Impl.*, pages 169–180, June 2006.

[27] X. Zhang and R. Gupta. Cost effective dynamic program slicing. *Conf. on Programming Language Design and Implementation*, pages 94–106, June 2004.

[28] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. *IEEE/ACM International Conference on Software Engineering*, pages 319–329, May 2003.