# Debugging in Parallel

James A. Jones
College of Computing
Georgia Institute of Technology
Atlanta, GA
jjones@cc.gatech.edu

James F. Bowring
Department of Computer Science
College of Charleston
Charleston, SC
BowringJ@cofc.edu

Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, GA
harrold@cc.gatech.edu

## ABSTRACT

The presence of multiple faults in a program can inhibit the ability of fault-localization techniques to locate the faults. This problem occurs for two reasons: when a program fails, the number of faults is, in general, unknown; and certain faults may mask or obfuscate other faults. This paper presents our approach to solving this problem that leverages the well-known advantages of parallel work flows to reduce the time-to-release of a program. Our approach consists of a technique that enables more effective debugging in the presence of multiple faults and a methodology that enables multiple developers to simultaneously debug multiple faults. The paper also presents an empirical study that demonstrates that our *parallel-debugging* technique and methodology can yield a dramatic decrease in total debugging time compared to a one-fault-at-a-time, or conventionally *sequential*, approach.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

## General Terms

Algorithms, Experimentation, Reliability

## Keywords

Fault localization, automated debugging, program analysis, empirical study, execution clustering

## 1. INTRODUCTION

Debugging software is an expensive and mostly manual process. This debugging expense has two main dimensions: the labor cost to discover and correct the bugs, and the time required to produce a failure-free program.[1] A developer generally wants to find a good trade-off between these dimensions that reflects the developer's resources and tolerance for delay. Of all debugging activities, *fault localization* is among the most expensive [15]. Any improvement in the process of finding faults will generally decrease the expense of debugging.

In practice, developers are aware of the number of failed test cases for their programs, but are unaware of whether a single fault or many faults caused those failures. Thus, developers usually target one fault at a time in their debugging. A developer can inspect a single failed test case to attempt to find its cause using an existing debugging technique (e.g., [4, 17]), or she can utilize all failed test cases using a fault-localization technique (e.g., [8, 9, 10, 12]). After a fault is found and fixed, the program must be retested to determine whether previously failing test cases now pass. If failures remain, the debugging process is repeated. We call this one-fault-at-a-time mode of debugging and retesting *sequential debugging*.

In practice, however, there may be more than one developer available to debug a program, particularly under urgent circumstances such as an imminent release date. Because, in general, there may be multiple faults whenever a program fails on a test suite, an effective way to handle this situation is to create parallel work flows so that multiple developers can each work to isolate different faults, and thus, reduce the overall time to a failure-free program. Like the parallelization of other work flows, such as computation, the principal problem of providing parallel work flows in debugging is determining the partitioning and assignment of subtasks. To perform the partitioning and assignment requires an automated technique that can detect the presence of multiple faults and map them to sets of failing test cases (i.e., clusters) that can be assigned to different developers.

Other researchers have presented techniques that cluster test cases. Podgurski and colleagues [5, 13] explore the use of multivariate projection to cluster failed executions according to the faults that cause them. Zheng and colleagues [18] cluster failing executions based on fault-predicting predicates. Liu and Han [11] explore the use of two distance measures for failing test cases. Although these techniques provide test-case clustering, they do not fully target or reach our goal of parallelizing the debugging effort.

To parallelize the debugging effort, we have developed, and present in this paper, a new technique—*parallel debugging*—that is an alternative to sequential debugging. Our technique automatically partitions the set of failing test cases into clusters that target different faults, called *fault-focusing clusters*, using behavior models and fault-localization information created from execution data. Each fault-focusing cluster is then combined with the passing test cases to get a *specialized test suite* that targets a single fault. Consequently, specialized test suites based on fault-focusing clusters can be assigned to developers who can then debug multiple faults in

[1] We refer to a program that is being tested as *failure-free* instead of fault-free or bug-free because, although we can know that none of the test cases in a test suite fail, we cannot, in general, know whether faults remain.


*ISSTA'07,* July 9–12, 2007, London, England, United Kingdom.

parallel. The resulting specialized test suites provide a prediction of the number of current, active faults in the program.

In this paper, we also present a new set of metrics that can be used to evaluate the effectiveness of the sequential and parallel debugging modes. Using these metrics, we empirically demonstrate the utility of the parallel mode.

The main benefit of our technique for parallel debugging is that it can result in decreased time to a failure-free program; our empirical evaluation supports this savings for our subject program. When resources are available to permit multiple developers to debug simultaneously, which is often the case, specialized test suites based on fault-focusing clusters can substantially reduce the time to a failure-free program while also reducing the number of testing iterations and their related expenses. Another benefit is that the fault-localization effort within each cluster is more efficient than without clustering. Thus, the debugging effort yields improved utilization of developer time, even if performed by a single developer. Our empirical evaluation shows that, for our subject, using the clusters provides savings in effort, even if debugging is done sequentially. A third benefit is that the number of clusters is an early estimate of the number of existing active faults.

A final benefit is that our technique automates a debugging process that is already naturally occurs in current practice. For example, on bug-tracking systems for open-source projects, multiple developers are assigned to different faults, each working with a set of inputs that cause different known failures. Our technique improves on this practice in a number of ways. First, the current practice requires a set of coordinating developers who triage failures to determine which appear to exhibit the same type of behavior. Often, this process involves the actual localization of the fault to determine the reason that a failure occurred, and thus a considerable amount of manual effort is needed. Our techniques can categorize failures automatically, without the intervention of the developers. This automation can save time and reduce the necessary labor involved. Second, in the current practice, coordinating developers categorize failures based on the failure output. Our techniques look instead at the execution behavior of the failures, such as how control flowed through the program, which may provide more detailed and rich information about the executions. Third, the current practice involves developers finding faults that cause failures using tedious, manual processes such as using print statements and symbolic debuggers on a single failed execution. Our techniques can automatically examine a set of failures and suggest likely fault locations in the program.

The main contributions of this paper are:

- Description of a new mode of debugging—parallel debugging—that provides a way for multiple developers to debug simultaneously a program for multiple faults by automatically producing specialized test suites for targeting individual faults.
- Development of a new set of metrics for evaluating the effectiveness of parallelizing debugging effort that we used to evaluate our technique and that can be used by others to evaluate other parallel-debugging techniques.
- Results of an empirical evaluation of the effectiveness of fault localization for multiple faults in both the default, sequential-debugging mode and the two parallel-debugging modes. For 100 8-fault versions of a program, our results show that parallel debugging yielded a 50% reduction in critical expense to a failure-free program over the traditional mode.

# 2. FAULT LOCALIZATION

In this section, we overview the fault-localization technique that we utilize.

In practice, software developers locate faults in their programs using a highly involved, manual process. This process usually begins when the developers run the program with a test case (or test suite) and observe failures in the program. The developers then choose a particular failed test case to run, and iteratively place breakpoints using a symbolic debugger, observe the state until an erroneous state is reached, and backtrack until the faults are found. This process can be time-consuming and ad-hoc. Additionally, this process uses results of only one execution of the program instead of using information provided by many executions of the program.

In prior work [7, 8], we defined a technique called TARANTULA that addresses these limitations of the current practice of locating faults. TARANTULA assigns a *suspiciousness* to each statement in the program based on the number of passed and failed test cases in a test suite that executed that statement. The intuition for this approach to fault localization is that statements in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases. The suspiciousness of a statement, $s$, is computed by

$$suspiciousness(s) = \frac{\%\text{failed}(s)}{\%\text{failed}(s) + \%\text{passed}(s)} \qquad (1)$$

In Equation 1, $\%\text{failed}(s)$ is a function that returns, as a percentage, the ratio of the number of failed test cases that executed $s$ to the total number of failed test cases in the test suite. $\%\text{passed}(s)$, likewise, is a function that returns, as a percentage, the ratio of the number of passed test cases that executed $s$ to the total number of passed test cases in the test suite. The suspiciousness score can range from 1, denoting a statement that is highly suspicious, to 0, denoting a statement that is not suspicious. A statement with a high suspiciousness score is one that is primarily executed by failed test cases, and likewise, a statement with a low suspiciousness score is one that is primarily executed by passed test cases. TARANTULA can work on any coverable entity such as branches, statements, and invariants. However, in this discussion we apply it at the statement level.

Using the suspiciousness, we sort the coverage entities of the program under test to provide a rank for each statement. The set of entities that have the highest suspiciousness is considered first by the developer when looking for the fault. If, after examining these statements, the fault is not found, the developer can examine the remaining statements in order of decreasing suspiciousness scores. This ordering of suspiciousness scores specifies a *ranking* of entities in the program. For evaluation purposes, each set of entities at the same rank is assigned a rank equal to the greatest number of statements that would need to be examined if the fault were the last statement in that rank to be examined.[2]

To illustrate the TARANTULA technique, consider the example in Figure 1. The program inputs three integers, and outputs the median of the three integers. The program contains two faults: one on line 7 and the other on line 10. To the right of the code is a test suite containing ten test cases. For each test case, its input is shown at the top of the column, its coverage is shown by the black dots in the column, and its pass/fail result is shown at the bottom of the column. The columns to the right of the test-case columns give the suspiciousness and rank, respectively, for the statements. (In the

[2]This rank computation, presented by Renieris and Reiss [14], has been used for evaluation and comparison of fault-localization techniques.

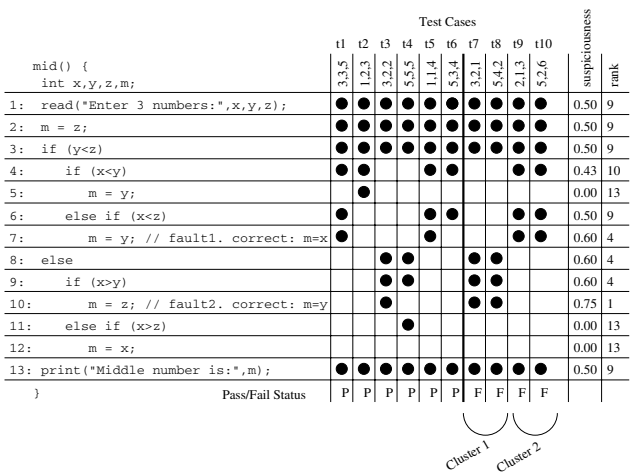


**Figure 1: *mid()* and all test cases before any faults are located.**

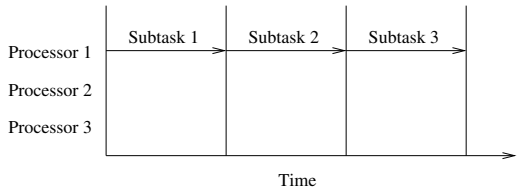


**Figure 2: Sequential processing of a task.**

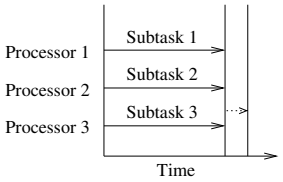


**Figure 3: Parallel processing of a task.**

figure, the first two failed test cases are labeled *Cluster 1* and the second two failed test cases are labeled *Cluster 2*—these clusters will be defined and discussed subsequently.)

To illustrate, consider the computation of suspiciousness for statement 1, which is executed by all ten test cases. The failed percentage is 100% (i.e., 4/4) and the passed percentage is also 100% (i.e., 6/6). Using Equation 1, the resulting suspiciousness is $100/(100+100)$ or 0.5. Next, consider the computation of rank for statements 7-10. Statement 10 is the only statement with suspiciousness of 0.75 (the greatest computed suspiciousness) and thus, the only statement with rank 1. Statements 7-9 have a suspiciousness of 0.6, so each of these statements gets assigned a rank of 4—the maximum number of statements that may be examined.

# 3. SEQUENTIAL AND PARALLEL DEBUGGING

The sequential and parallel debugging modes, described in Section 1, are analogous to many types of sequential and parallel work flows. One such example is the parallelization of computation on multi-processor computers. On a multi-processor computer, a task is divided into subtasks that are processed simultaneously with coordination between the processors. There is a cost of this coordination, and thus, the total processing effort is often higher in the parallel computation than the sequential one. However, because of better utilization of the processors and the divide-and-conquer strategy, the task can often complete faster when computed in parallel.

To illustrate, consider Figures 2 and 3 that represent sequential and parallel computation of a task, respectively. In the figures, the solid arrows represent the cost of the subtasks and the dotted arrow in Figure 3 represents the overhead of performing the tasks in parallel. The figures illustrate that, whereas there is some cost associated with the parallelization of the task, with parallel processing, the overall time to complete the task can be much less than in the sequential processing of the task. Also, Figure 2 shows that in the sequential processing, only one of the processors is utilized in the computation of this task.

Figures 2 and 3 illustrate two dimensions of this parallelization—the completion time of the task and the degree of parallelization that was accomplished for the task. The "width" of these figures depicts the former dimension, and the "height" depicts the latter. In Figure 2, the width shows that the task took a relatively long time to complete, and the height shows that there was little parallelization of the task—in this case, there was no parallelization of the task. In Figure 3, the width shows that the task took a relatively short time to complete, and the height shows that the task was parallelized to a large degree—in this case, the task was fully parallelized. For the parallelization of the debugging task, we can also measure these two dimensions. The completion of the debugging task can be measured as the time to debug the faults causing the failures, and the degree of parallelization of the debugging task can be measured as the number of developers that can simultaneously debug the program.

Like the parallelization of a computation task, some debugging subtasks, such as locating one fault, can dominate other tasks. For example, a program that contains four faults may cause a number of test cases to fail. Upon inspection, we may find that all of the failed test cases fail due to one fault. After that dominating fault is found and fixed, the program is re-tested. This re-testing reveals that there are still a number failed test cases, but these failed test cases are now caused by the remaining three faults. This phenomenon is illustrated in Figure 4. In the example, Fault 1 must be located and fixed before Faults 2, 3, and 4 can be located and fixed because all failed test cases fail due to Fault 1. Only after Fault 1 is fixed do Faults 2, 3, and 4 manifest themselves as failures.

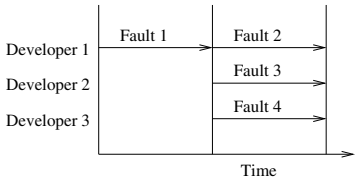


**Figure 4: Fault 1 dominates Faults 2, 3, and 4.**

Unlike the parallelization of a computation task, the cost for each fault subtask can change as a result of the parallelization. In fact, we have found empirically that the fault subtasks are often more efficient in the parallelized version. In the parallelized version, the test suite for each fault subtask is generated specifically for that fault. Thus the fault localization is often more effective at locating that fault than the non-specialized, full test suite.

The key to the parallelization of debugging is the creation and specialization of test suites that target different faults. We create these specialized test suites from the original test suite $T$. Each

| | Test Cases | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 | t10 | suspiciousness | |
| mid() { int x,y,z,m; | 3,3,5 | 1,2,3 | 3,2,2 | 5,5,5 | 1,1,4 | 5,3,4 | 3,2,1 | 5,4,2 | 2,1,3 | 5,2,6 | | rank |
| 1: read("Enter 3 numbers:",x,y,z); | ● | ● | ● | ● | ● | ● | ● | ● | ● | ● | 0.50 | 7 |
| 2: m = z; | ● | ● | ● | ● | ● | ● | ● | ● | ● | ● | 0.50 | 7 |
| 3: if (y<z) | ● | ● | ● | ● | ● | ● | ● | ● | ● | ● | 0.50 | 7 |
| 4: if (x<y) | ● | ● | | | ● | ● | | | ● | ● | 0.67 | 3 |
| 5: m = y; | | ● | | | | | | | | | 0.00 | 13 |
| 6: else if (x<z) | ● | | | | ● | ● | | | ● | ● | 0.73 | 2 |
| 7: m = y; // fault1. correct: m=x | ● | | | | ● | | | | ● | ● | 0.80 | 1 |
| 8: else | | | ● | ● | | | ● | ● | | | 0.00 | 13 |
| 9: if (x>y) | | | ● | ● | | | ● | ● | | | 0.00 | 13 |
| 10: m = y; // fault2 corrected | | | ● | | | | ● | ● | | | 0.00 | 13 |
| 11: else if (x>z) | | | | ● | | | | | | | 0.00 | 13 |
| 12: m = x; | | | | | | | | | | | 0.00 | 13 |
| 13: print("Middle number is:",m); | ● | ● | ● | ● | ● | ● | ● | ● | ● | ● | 0.50 | 7 |
| } Pass/Fail Status | P | P | P | P | P | P | P | F | F | | | |

**Figure 5: Example *mid()* and all test cases after *fault2* was located and fixed.**

specialized test suite is composed of all passing test cases in $T$ and some subset of the failing test cases in $T$. Our technique automatically partitions the failing test cases in $T$ into subsets that exhibit similar failures. With these specialized test suites, our technique applies a fault-localization algorithm to automatically find the likely locations of the faults. These specialized test suites and fault-localization results are assigned to different developers to debug. After each developer has found and fixed a fault, and committed the changes back to the change management system, the program is retested. If the program still exhibits failures, the process is repeated.

Consider the example presented in Figure 1. In the traditional, sequential mode of debugging, the developer would be aware that there were four failed test cases, but would be unaware of the number of faults that caused them. Thus, a typical, sequential process that she follows might be:

1. Examine the statement at the highest level of suspicion: statement 10. She would realize that it was, in fact, faulty and would correct the bug.

2. Rerun the test suite to determine whether all of the faults were corrected. She would witness that two of the failed test cases now pass and two of the formerly failed test cases still fail. Figure 5 depicts the coverage and new, recomputed fault-localization results.

3. Examine the statement at the highest level of suspicion: statement 7. She would realize that it was, in fact, faulty and would correct the bug.

4. Rerun the test suite. In this case, she would witness that all test cases pass.

Consider again the example in Figure 1. To demonstrate the utility of parallel debugging, assume that there exists a technique that can automatically determine that there are two distinct types of failures in this program and can automatically cluster them. The groupings of "Cluster 1" and "Cluster 2" are depicted in Figure 1. Given this clustering, a test suite can be generated for each cluster by combining all passed test cases with each cluster, and the fault-localization results can be calculated on this new, specialized test suite. The specialized test suites are shown in Figures 6 and 7. Each of these test suites and fault-localization results can be given to a different developer to debug. A parallel process that they follow in this circumstance might be:

| | Test Cases | | | | | | | | suspiciousness | |
|---|---|---|---|---|---|---|---|---|---|---|
| | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | | |
| mid() { int x,y,z,m; | 3,3,5 | 1,2,3 | 3,2,2 | 5,5,5 | 1,1,4 | 5,3,4 | 3,2,1 | 5,4,2 | | rank |
| 1: read("Enter 3 numbers:",x,y,z); | ● | ● | ● | ● | ● | ● | ● | ● | 0.50 | 7 |
| 2: m = z; | ● | ● | ● | ● | ● | ● | ● | ● | 0.50 | 7 |
| 3: if (y<z) | ● | ● | ● | ● | ● | ● | ● | ● | 0.50 | 7 |
| 4: if (x<y) | ● | ● | | | ● | ● | | | 0.00 | 13 |
| 5: m = y; | | ● | | | | | | | 0.00 | 13 |
| 6: else if (x<z) | ● | | | | ● | ● | | | 0.00 | 13 |
| 7: m = y; // fault1. correct: m=x | ● | | | | ● | | | | 0.00 | 13 |
| 8: else | | | ● | ● | | | ● | ● | 0.75 | 3 |
| 9: if (x>y) | | | ● | ● | | | ● | ● | 0.75 | 3 |
| 10: m = z; // fault2. correct: m=y | | | ● | | | | ● | ● | 0.86 | 1 |
| 11: else if (x>z) | | | | ● | | | | | 0.00 | 13 |
| 12: m = x; | | | | | | | | | 0.00 | 13 |
| 13: print("Middle number is:",m); | ● | ● | ● | ● | ● | ● | ● | ● | 0.50 | 7 |
| } Pass/Fail Status | P | P | P | P | P | P | F | F | | |

**Figure 6: Example *mid()* with *Cluster 1*.**

| | Test Cases | | | | | | | | | | suspiciousness | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | t1 | t2 | t3 | t4 | t5 | t6 | t9 | t10 | | | | |
| mid() { int x,y,z,m; | 3,3,5 | 1,2,3 | 3,2,2 | 5,5,5 | 1,1,4 | 5,3,4 | 2,1,3 | 5,2,6 | | | | rank |
| 1: read("Enter 3 numbers:",x,y,z); | ● | ● | ● | ● | ● | ● | ● | ● | | | 0.50 | 7 |
| 2: m = z; | ● | ● | ● | ● | ● | ● | ● | ● | | | 0.50 | 7 |
| 3: if (y<z) | ● | ● | ● | ● | ● | ● | ● | ● | | | 0.50 | 7 |
| 4: if (x<y) | ● | ● | | | ● | ● | ● | ● | | | 0.60 | 3 |
| 5: m = y; | | ● | | | | | | | | | 0.00 | 13 |
| 6: else if (x<z) | ● | | | | ● | ● | ● | ● | | | 0.67 | 2 |
| 7: m = y; // fault1. correct: m=x | ● | | | | ● | | ● | ● | | | 0.75 | 1 |
| 8: else | | | ● | ● | | | | | | | 0.00 | 13 |
| 9: if (x>y) | | | ● | ● | | | | | | | 0.00 | 13 |
| 10: m = z; // fault2. correct: m=y | | | ● | | | | | | | | 0.00 | 13 |
| 11: else if (x>z) | | | | ● | | | | | | | 0.00 | 13 |
| 12: m = x; | | | | | | | | | | | 0.00 | 13 |
| 13: print("Middle number is:",m); | ● | ● | ● | ● | ● | ● | ● | ● | | | 0.50 | 7 |
| } Pass/Fail Status | P | P | P | P | P | P | F | F | | | | |

**Figure 7: Example *mid()* with *Cluster 2*.**

1. Examine the statements at the highest level of suspicion: statement 7 for one developer and statement 10 for the other developer. They would each realize that those were, in fact, faulty and would correct them.

2. Rerun the test suite to determine if all of the faults were corrected. In this case, they would witness that all of the test cases pass.

This example demonstrates how an automated technique may reduce the overall time to achieve a failure-free program. Also notice that $fault1$ on line 7 was made more noticeable by the removal of the "noise" generated by $fault2$ on line 10 without the need to actually correct $fault2$.

# 4. TECHNIQUES FOR CLUSTERING FAILURES

To achieve our goal of enabling developers to simultaneously debug multiple faults in parallel, we defined a parallel-debugging process, which is shown by the dataflow diagram[3] in Figure 8. The program under test, $P$, is instrumented to produce $\hat{P}$. When $\hat{P}$ is executed with test suite $T$, it produces a set of passing test cases

[3]Rectangles represent processing components, edges represent the flow of data between the components, and labels on the edges represent the data.

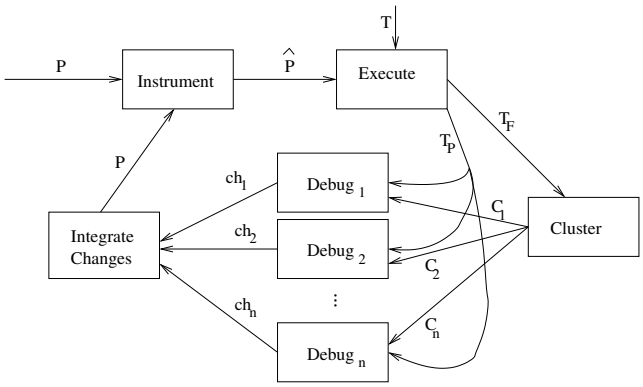


**Figure 8: Technique for debugging in parallel.**

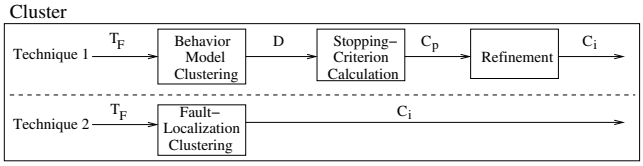


**Figure 9: Two alternative techniques to cluster failed test cases for parallel debugging.**

$T_P$ and a set of failing test cases $T_F$, along with execution information, such as branch or method profiles. $T_F$ and the execution information are input to the clustering technique, $Cluster$, to produce a set of fault-focused clusters $C_1$, $C_2$, ..., $C_n$ that are disjoint subsets of $T_F$. Each $C_i$ is combined with $T_P$ to produce a specialized test suite that assists in locating a particular fault. Using these test suites, developers can debug the program in parallel—shown as $Debug_i$ in the figure. The resulting changes, $ch_1$, $ch_2$, ..., $ch_n$, are integrated into the program. This process can be repeated until all test cases pass.

The novel component of this parallel-debugging process, *Cluster*, is shown in more detail in Figure 9. We have developed two techniques to *Cluster* failed test cases. This section presents details of these techniques.

## 4.1 Clustering Based on Profiles and Fault-localization Results

Our first fault-focused clustering technique, shown as Technique 1 in Figure 9, first clusters behavior models of executions of failed test cases, $T_F$, to produce a complete clustering history (or dendrogram) $D$ (described in Section 4.1.1). The technique then uses fault localization information to identify a stopping criterion for the $D$, and produces a preliminary set of clusters, $C_p$ (described in Section 4.1.2). The technique finally refines $C_p$ by merging those clusters that appear to be focused on the same faults and outputs the final set of clusters, $C_i$ (described in Section 4.1.3). The first step is based on instrumentation profiles, and the second and third steps are based on fault-localization results.

### *4.1.1 Clustering profile-based behavior models*

To group the failed test cases according to the likely faults that caused them, we use a technique for clustering executions based on agglomerative hierarchical clustering [3].[4] For each test case, this technique creates a behavior model that is a statistical summary of data collected during program execution. The specific models are discrete-time Markov chains (DTMCs) and clustering occurs iteratively with the two most similar models merged at each iteration. Every execution is represented by its branching behavior. The branching profile of an execution is represented by the percentage of times that each branch of a predicate statement was taken. Similarity is measured with a similarity metric—in this research, that metric is the sum of the absolute difference between matching transition entries in the two DTMCs being compared. Each pair of executions is assigned a similarity value that is computed by taking the sum of the differences of the branch percentage profile. Our technique initially sets the stopping criterion for the clustering to one cluster, so that the clustering proceeds until one cluster remains.

[4]See Bowring et al. for details of this clustering technique [3].

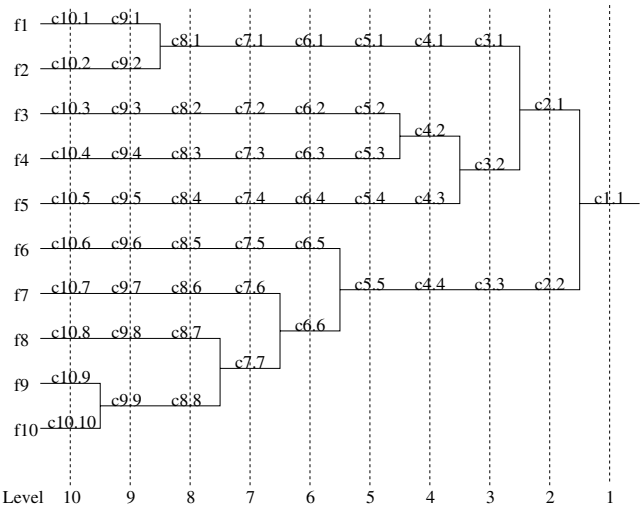


**Figure 10: Dendrogram for 10 failing test cases.**

To illustrate the clustering, consider Figure 10, which shows a *dendrogram* [6][5] that depicts an example clustering of ten execution models. The left side of the figure shows the ten individual failed test cases represented as $f1...f10$. At each "level" of the dendrogram, the process of clustering the two most similar test cases is shown.[6] Initially, at level 10, failed test cases $f1$, ..., $f10$ are placed in clusters $c10.1$ through $c10.10$, respectively. Then, the clustering algorithm finds that $c10.9$ and $c10.10$ have the most similar behavior models, and groups these two clusters to get a new cluster, labeled as $c9.9$, which results in nine clusters at level 9. This clustering continues until there is one cluster, $c1.1$.

Conventionally, a good stopping criterion for the clustering, which is difficult to determine [6], is based on the practitioners' domain knowledge. Because our domain is debugging, we have developed a technique that inputs the dendrogram and computes the stopping criterion based on fault-localization information. We describe this stopping criterion in the next section.

### *4.1.2 Using fault localization to stop clustering*

We use a fault-localization algorithm for this secondary assessment of the clustering. We use our TARANTULA technique to provide a prediction of the location of the fault for each specialized test suite. A number of other fault-localization techniques might also be used for this purpose (e.g., [10, 12])—we chose TARANTULA be-

[5]A *dendrogram* is a tree diagram frequently used to illustrate the arrangement of clusters produced by a clustering algorithm.

[6]If multiple pairs are equally "most similar," one such pair is chosen at random.

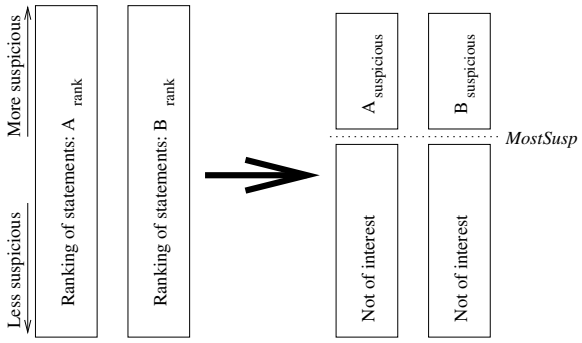


**Figure 11: Similarity of fault-localization results is performed by identifying two sets of interest $A_{suspicious}$ and $B_{suspicious}$ and performing a set similarity.**

cause studies have shown it to be among the most effective [7], and because of its availability.

Figure 10 shows the process of grouping clusters until one cluster remains. Unless there is only one behavior represented by the test cases, at some point during this clustering, two clusters are merged that are not similar. In the context of fault localization, unless there is only one fault, at some point in the clustering process, the failing test cases that fail due to one fault are merged with failing test cases that fail due to another fault. We want our technique to stop the clustering process just before this type of clustering occurs.

Our technique identifies the clustering-stopping criterion by leveraging the fault-localization results. The technique computes the fault-localization ranks (the ranking of all statements in the program from most suspicious to least suspicious based on the Tarantula heuristic) for each individual failed test case (shown at the left side of Figure 10) using a test suite of all passing test cases with that one failed test case. Then, every time a merge is made in the clustering process, our technique calculates the fault-localization ranks using the members of that cluster and the passed test cases. Thus, with regard to a dendrogram, such as Figure 10, our technique computes fault-localization ranks at every merge point of two clusters.

Using these fault-localization ranks at all merge points in the dendrogram, our technique uses a similarity measure to identify when the clustering process appears to lose the ability to find a fault—that is, clusters two items that contribute to find a different suspicious region of the program. To measure the similarity of two fault-localization results, we first define the suspicious area of the program as the set of statements of the program that are deemed "most suspicious" for each of the results. This process is depicted in Figure 11.

To decide whether two fault-localization results identify the same suspicious region of the program, we must establish the threshold that differentiates the *most suspicious* statements from the statements that are *not of interest*. We call this threshold *MostSusp*. For example, we may assign the value of $20\%$ to *MostSusp*—this means that the top $20\%$ of the suspicious statements in the rank are in the most suspiciousness set, and that the lower $80\%$ are not of interest.

To compare the two sets of statements, we use a set-similarity metric called *Jaccard set similarity*. The Jaccard metric computes a real value between 0 (completely dissimilar) and 1 (completely similar) by evaluating the ratio of the cardinality of the intersection of these sets and the cardinality of the union of these sets. The similarity of two sets, $A$ and $B$, is computed by the following equation:

$$similarity(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (2)$$

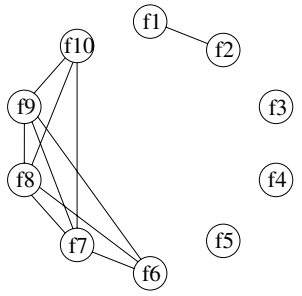


**Figure 12: Pairwise similarity of suspiciousness computed using each failed test case is shown by a connecting line. Clusters are formed by taking a closure of the similar test cases.**

To determine whether the two sets are *similar* or *dissimilar*, we must establish the threshold for the similarity metric. We call this threshold $Sim$. For example, we may assign the value of $0.7$ to $Sim$—this means that two sets of suspicious statements will be in the same cluster if their similarity value is at or above $0.7$. In practice and in our experiments, these thresholds, $MostSusp$ and $Sim$, are determined during a training phase that shadows the debugging process.

To determine where to stop the clustering, our technique traverses the dendrogram in reverse—starting at the final cluster. At each step, the technique examines the merged clusters at that level, and computes the similarity, using Equation 2, of the fault-localization ranks of the merged cluster with its constituent clusters. When at least one of the constituent clusters is dissimilar to the merged cluster, the traversal has found new information, and thus, the traversal continues (i.e., this is not the stopping point for the clustering). For example, in Figure 10, the fault-localization result of $c1.1$ is compared with the fault-localization result of each $c2.1$ and $c2.2$ using Equation 2. If $c1.1$ is dissimilar to either $c2.1$ or $c2.2$, the traversal continues.

### 4.1.3 *Using fault-localization clustering to refine clusters*

After the clusters are identified using profiles and fault-localization results, our technique performs one additional refinement. Occasionally, similar fault-localization results are obtained on multiple "branches" of a dendrogram. To merge these similar clusters, the technique groups clusters that produce similar fault-localization results.

To identify the places where this refinement of the clustering can be applied, we perform a pairwise comparison of the fault-localization results of the clusters at the stopping-point level of the dendrogram. For this comparison, we use the *Jaccard similarity* parameterized for this task. Then we merge the similar clusters.

For example, in Figure 10, consider that the stopping point of the clustering was determined to be best at level 5. A pairwise similarity would be calculated for the five clusters at this level by inspecting the similarity of the suspicious statements that each targets. If it found that clusters $c5.4$ and $c5.5$ were similar, these would be combined to produce the final set of clusters.

## 4.2 Clustering Based on Fault-localization Results

Our second fault-focusing technique, shown as Technique 2 in Figure 9, uses only the fault-localization results. The technique first computes the fault-localization suspiciousness rankings for the individual failed test cases, $T_F$, and uses the *Jaccard similarity*

metric to compute the pairwise similarities among these rankings. Then, the technique clusters by taking a closure of the pairs that are marked as similar.

For example, consider Figure 12, which shows the same ten failed test cases depicted in Figure 10. Each failed test case is depicted as a node in the figure. The technique combines each failed test case with the passing test cases to produce a test suite. The technique uses TARANTULA to produce a ranking of suspiciousness for each test suite, and these rankings are compared using the Jaccard metric in the same way described in Section 4.1.2 and 4.1.3. The technique records the pairs of rankings that are deemed similar (above the similarity threshold). In Figure 12, a pairwise similarity between failed test cases is depicted as an edge. We produce clusters of failed test cases by taking a closure of the failed test cases that were marked similar. Using the example in Figure 12, test case nodes that are reachable over the similarity edges are clustered together. In this example, failed test cases $f1$ and $f2$ are combined to a cluster, $f6$, $f7$, $f8$, $f9$, and $f10$ are combined to a cluster, and $f3$, $f4$, and $f5$ are each singleton clusters.

# 5. METRICS FOR EVALUATION

To evaluate our new parallel-debugging technique, we need to measure how well it reduces costs in each of the two main dimensions of debugging: labor to find and fix the bugs and time to a failure-free program. Thus, we developed two metrics—*total developer expense* and *critical expense to a failure-free program*. This section presents these metrics and required supporting measures.

In previous empirical studies of the effectiveness and comparison of fault-localization techniques, a metric (e.g., [4, 7, 12]), computed using statements' ranks, has been used for measuring developer's effort in locating the fault. This metric was originally presented by Renieris and Reiss [14]. In each of these studies, the technique was used to find just one fault. The metric, $Score$, is defined as the percentage of the program that need *not* be examined to find the fault using the rank described in the preceding discussion, and is computed by the following equation:

$$Score = \left(1 - \frac{\text{rank of fault}}{\text{size of program}}\right) * 100 \quad (3)$$

To evaluate the localization of multiple faults, we use a variation of this metric. Instead of evaluating the fault-localization effectiveness in terms of the percentage of the program that need *not* be examined to find the fault (as described by Equation 3), we use the inverse: the percentage of the program that *must* be examined to find the fault. This value is indicative of the time or effort that the developer would spend in finding a single fault in the program if she examined the program using the ranks computed by the fault-localization technique. This metric, which we call $Expense$, is computed by the following equation:

$$Expense = \frac{\text{rank of fault}}{\text{size of program}} * 100 \quad (4)$$

To illustrate the *Expense* metric, consider again Figure 1. If a developer were debugging the program using the fault-localization results for the ten test cases, she would start by examining the most suspicious statement—statement 10. She would verify that this statement is in fact faulty, and fix the fault. For the fault in statement 10, $Expense$ would be $1/13 * 100$ or $7.69\%$—that is, $7.69\%$ of the program needed to be examined to find the fault.

After the developer finds and fixes the fault in statement 10, she would retest the program to see if any additional faults were detected, and find that two of the previously failing test cases now pass but two test cases still fail. Figure 5 shows the results for this retesting and the new fault-localization results. Using the new ranks, the developer would first examine statement 7, which is in fact faulty. For the fault on statement 7, the $Expense$ would be $1/13 * 100$ or $7.69\%$—that is, $7.69\%$ of the program needed to be examined to find this fault. After the fault on statement 7 was found and fixed, retesting would reveal that all of the test cases now pass.

The debugging process just described is sequential debugging—a single fault was found and fixed, the program was retested, and the process iterated until the program was failure-free. The total developer expense is then the sum of the $Expense$ for each iteration of debugging. In this example, the total $Expense$ cost is $7.69\% + 7.69\% = 15.38\%$.

Now consider that, using the techniques defined in Section 4, we were able to cluster the failing test cases into two clusters—$Cluster1$ (test cases 7 and 8) and $Cluster2$ (test cases 9 and 10)—as in Figure 1. Figures 6 and 7 show the resulting test suites and their fault-localization results. Figure 6 shows the specialized test suite consisting of the failed test cases from *Cluster 1* and the passing test cases. Using this specialized test suite, the fault is successfully located in statement 10, with a rank of 1 and an $Expense$ of $7.69\%$. Figure 7 shows the specialized test suite consisting of the failed test cases from *Cluster 2* and the passing test cases. Using this specialized test suite, the fault is successfully located in statement 7, with a rank of 1 and an $Expense$ of $7.69\%$.

For this example, the total developer expense can be assessed as the sum of $Expense$ for each fault. In both of these cases—sequentially in Figures 1 and 5, and in parallel in Figures 6 and 7—the total developer expense is $7.69\% + 7.69\% = 15.38\%$. The advantage of the parallel debugging scenario over the sequential debugging scenario is that in the parallel debugging scenario, two developers could have been working simultaneously to produce a failure-free program in a shorter time. This example illustrates the need for metrics that can assess both the total developer expense and the time required for a failure-free program.

We calculate a metric to assess the *total developer expense* and denote this metric as $D$. $D$ is used to assess the total of all developers' efforts to find the faults in a program, both in parallel and in sequence. $D$ is computed as the sum of the developer $Expense$ for each fault in the program, and is computed by the following equation:

$$D = \sum_{i=1}^{|faults|} Expense_i \quad (5)$$

The total developer expense, $D$, for the example in Figure 1, both in sequence and in parallel, is $15.38\%$. In the sequential mode, the developer expense occurs in sequence—that is, one developer is active for the first fault, and one developer is active for the second fault. In the parallel mode, two developers can work simultaneously—one on each fault. $D$ captures an important aspect of the cost that can be translated into essential quantities such as employee-hours and payroll-expense to the company. However, it misses another important aspect of these debugging modes—the time to deliver a failure-free program.

Thus, we compute another metric to assess the *critical expense to a failure-free program* and denote this metric as $FF$. $FF$ is used to assess the relative savings in terms of the time to deliver a failure-free program, i.e., the expense of the limiting, or critical path to a failure-free program. $FF$ is computed as the sum of the maximum developer expense at each debugging iteration[7], which is the crit-

[7]Figure 4 shows an example of why more than one debugging iteration may be necessary

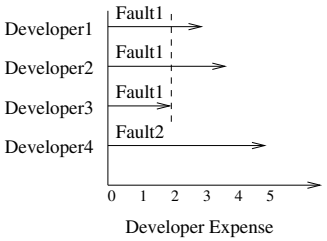


**Figure 13: The cost model accounts for when the clustering technique produces multiple test suites that target the same fault.**

ical path to achieving a failure-free program, and is computed by the following equation:

$$FF = \sum_{i=1}^{|iterations|} \max\{Expense_f | f \text{ is a fault subtask at iteration } i\} \quad (6)$$

Note that, in the sequential mode of debugging, the $D$ and $FF$ values are always equal—the total developer expense and the critical expense to a failure-free program are equal because both are calculated as the sum of the one-at-a-time developer expense.

These two metrics, total developer expense ($D$) and critical expense to a failure-free program ($FF$), capture the two important dimensions of debugging in parallel. The goal of this work is to reduce the critical expense to a failure-free program while not drastically increasing the total developer expense of parallel debugging over sequential debugging. For the example in Figure 1, the $FF$ value is reduced from $15.38\%$ in the sequential mode to $7.69\%$ in the parallel mode, while the $D$ value stays the same in both. Thus, for this example, we are successful in accomplishing this goal.

Note that both $D$ and $FF$ might exceed $100\%$. This is a consequence of our metrics representing multiple faults and our desire to define *Expense* related to the *Score* metric that has been used in many previous experiments. However, for simplicity, we drop the percentage notation from the *D* and *FF* values and use them as metrics for relative comparison in the following sections.

It is worth noting at this point that the $Expense$ metric also accounts for errors in the clustering process. Particularly, when the clustering approach produces multiple clusters for finding the same fault, multiple developers would be unnecessarily expending effort when a single developer would be a more efficient use of developer effort. We assume that the developers that are simultaneously debugging communicate with one another when a fault has been found. This communication limits the expense expended by any other developer that may be working to find the same fault. When a developer receives notice from another developer that a fault has been found, he can check to see if that fault is the one causing his failures, and if so, stop his debugging efforts. Thus, the expense metric is calculated as the product of the minimum of the redundant effort and the number of developers working on that fault. Figure 13 shows an example of where the clustering technique produces three clusters that target the same fault and a fourth cluster that targets another fault. In this example, the expense required to find fault 1 is calculated as $Expense_1 = 3 * \min(2, 3, 4) = 6$, where the minimum of these three developers' expenses is depicted with the dotted line at a value of 2. The total developer expense for this example is calculated as $D = Expense_1 + Expense_2 = 6 + 5 = 11$. Thus, we capture the inherent inefficiencies that sometimes occur because of inaccurate clustering.

# 6. EMPIRICAL STUDY

To compare the sequential mode of debugging with the two parallel-mode-debugging techniques, described in Section 4, we implemented our technique by integrating three existing systems. The Aristotle Analysis System [2], written in C, analyzes and instruments C programs. Argo, written in C#, models and clusters program executions. TARANTULA, written in Java, provides fault-localization and visualization of results. We also wrote Perl and shell scripts for cross-platform scaffolding and results processing.

With this integrated system, we conducted an empirical study. This section overviews the empirical set up and presents the results of the study.

## 6.1 Variables and Measures

Our studies manipulated one independent variable—the technique for creating the fault-focusing clusters that drive the debugging process. The techniques that we examine are

**S** : sequential
**P1** : parallel technique 1 that clusters using execution profiles with fault-localization-result refinement (Section 4.1)
**P2** : parallel technique 2 that clusters using fault-localization results (Section 4.2)

Our studies measure each technique's effectiveness using the two dependent variables described in Section 5—the total developer expense (*D*) and the critical expense to a failure-free program (*FF*). To compare these techniques, we calculate the sample means of these measures for each technique. Then, we evaluate the statistical significance of the differences of the sample means of the three techniques.

## 6.2 Object of Analysis

Our object of analysis is *Space*, a program developed by the European Space Agency. *Space* is a program written in C and consists of 3,660 executable statements. There are 33 known faults for *Space*, discovered during the program's development. A test suite for *Space* was constructed from 10,000 test cases generated randomly by Vokolos and Frankl [16] and 3,585 test cases created by researchers in the Aristotle Research Group [2]. This test suite guarantees that each executable branch program is exercised by at least 30 test cases.

## 6.3 Experimental Protocol

Our experimental protocol created 100 8-fault versions of *Space* by choosing from the available faults at random. We simulated a developer's test suite for each version by choosing a test suite at random from a collection of 1000 branch-adequate test suites, each with an average of 156 test cases. Over the course of evaluating all debugging modes, we generated a total of 1147 derivative multi-fault versions. For example, in the sequential mode, we started with an 8-fault version, ran it with the test suite, and detected and removed one fault. Then we generated the 7-fault version using the remaining seven faults and ran it with the same test suite. The sequential fault-removal process repeated, creating the 6-, 5-, 4-, 3-, 2-, and 1-fault versions until no executions in the test suite failed. In the two parallel modes, we also started with the 8-fault version, determined the number of clusters and found the faults that they focused, removed those faults, and repeated the process with another iteration of debugging in parallel with a derivative faulty version containing only the remaining faults.

To determine the best threshold parameterization, as described in Section 4, we sampled various parameters using ten 8-fault versions. We selected from these the best candidates for use in our

| Source | Mean | Std. Dev. | 99% lower | 99% upper |
|---|---|---|---|---|
| $D_S$ | 36.63 | 22.35 | 31.06 | 42.20 |
| $D_{P1}$ | 31.50 | 26.63 | 24.86 | 38.14 |
| $D_{P2}$ | 26.43 | 22.42 | 20.84 | 32.02 |
| $D_S - D_{P1}$ | 5.13 | 15.49 | 1.27 | 8.99 |
| $D_S - D_{P2}$ | 10.20 | 13.54 | 6.82 | 13.57 |
| $D_{P1} - D_{P2}$ | 5.07 | 13.14 | 1.80 | 8.34 |

**Table 1: Total developer expense, $D$.**

| Source | Mean | Std. Dev. | 99% lower | 99% upper |
|---|---|---|---|---|
| $FF_S$ | 36.63 | 22.35 | 31.06 | 42.20 |
| $FF_{P1}$ | 18.19 | 13.74 | 14.76 | 21.61 |
| $FF_{P2}$ | 20.95 | 15.00 | 17.21 | 24.69 |
| $FF_S - FF_{P1}$ | 18.44 | 13.96 | 14.96 | 21.92 |
| $FF_S - FF_{P2}$ | 15.68 | 12.03 | 12.68 | 18.68 |
| $FF_{P1} - FF_{P2}$ | -2.76 | 7.72 | -4.69 | -0.84 |

**Table 2: Critical expense to failure-free, $FF$.**

study of the remaining 90 8-fault versions. This "training" of the parameters for a program is similar to how we would prescribe training in the field. For both clustering techniques, *P1* and *P2*, we used only the top $20\%$ of the most suspicious lines—$MostSusp = 20\%$. For determining the stopping criterion for the clustering, we used a threshold of $Sim = 68\%$ (roughly two standard deviations) in the *Jaccard* similarity scores. However, for clustering based on sets of suspicious code, we used a threshold of $Sim = 50\%$ in the *Jaccard* similarity scores.

We gathered the $D$ and $FF$ scores for each of the 90 versions and report their mean and standard deviation. We also compute the pairwise difference of the three different techniques' scores. The $D_S$, $D_{P1}$, $D_{P2}$, and their pairwise differences (and likewise for $FF$) can be taken as a sample of the entire population of all 8-fault versions for our subject program. Because our sample size is adequately large, their distribution approximates a normal distribution. Thus, we compute a two-sided $t$-interval with a confidence level of 99%. Interpret this statistic to mean that with 99% confidence, the mean of the sample will be in the range defined by the lower and upper bounds. For the samples calculated by differencing two data points, if both bounds have the same sign, then we have confidence (with 99% certainty) that one mean is always larger than the other for the entire population.

## 6.4 Results and Analysis

As detailed in Section 5, our two principal metrics for comparing the costs of the three investigated modes are total developer expense *D* and critical expense to failure-free *FF*.

### 6.4.1 Total developer expense

Table 1 (and Figure 14) shows the comparative results for total developer expense *D*. The columns show the sample source, mean, and standard deviation, followed by the lower and upper 99% confidence interval bounds calculated for a two-sided $t$-interval for the mean of the sample. The first three rows show statistics derived by measuring *D* for each of the three modes. For example, for the sequential debugging mode, the sample mean of $D_S$ for the 90 8-fault versions of *Space* is $36.63$ with standard deviation of $22.35$. The two-sided $t$-interval with a confidence level of 99% for this mean is between $31.06$ and $42.20$.

The last three rows show statistics about the pair-wise differences among the individual means of the three debugging modes. For example, in the fourth row, the difference between the means $D_{P1}$ and $D_S$ is $5.13$, with a standard deviation of $15.49$. The two-sided $t$-interval for the difference of the means $D_S - D_{P1}$ is between $1.27$ and $8.99$

The results show that the most expensive developer expense is for the *S* mode and the least is for the *P2* mode. When comparing the means of these two debugging modes in row five, we see that $D_S$ is expected, with a $99\%$ confidence, to be greater than $D_{P2}$ by a value between $6.82$ and $13.57$. These results mean that the use of fault-focusing clusters and the resultant test suites yields reduced total developer expense even if the debugging is done by a single developer.

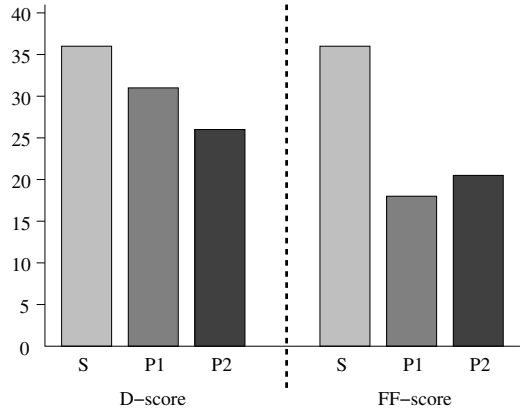


**Figure 14: Mean score for the total developer expense, $D$, and the critical expense to failure-free, $FF$, for the three techniques.**

### 6.4.2 Critical expense to a failure-free program

Table 2 (and Figure 14) presents the comparative results for the critical expense to a failure-free program *FF*. The table is constructed identically to Table 1. Here, for example, for the sequential debugging mode, the sample mean for $FF_S$ for the 90 8-fault versions of *Space* is $36.63$ with standard deviation of $22.35$. The two-sided $t$-interval with a confidence level of 99% for this mean is between $31.06$ and $42.20$. Note that $D_S$ and $FF_S$ are necessarily identical, as explained in Section 5.

The results show that the greatest expense to failure-free is for the *S* mode and the least is for the *P1* mode. When comparing the means of these two debugging modes in row four, we see that the mean of $FF_S$ is expected, with a $99\%$ confidence, to be greater than the mean of $FF_{P1}$ by a value between $14.96$ and $21.92$. Furthermore, when we compare the means of the two parallel debugging modes in row six, we see that $FF_{P1}$ is expected, with a $99\%$ confidence, to be *less* than $FF_{P2}$ by a value between $0.84$ and $4.69$ (negating the values shown.) These results mean that both parallel modes are better than the sequential mode, and that for this subject *P1* outperforms *P2* in terms of $FF$. The results show that the use of the *P1* debugging mode yields a 50% reduction in the critical expense to a failure-free program over the *S* mode.

## 6.5 Discussion

Notable in the results is that the *S* mode is the most expensive both in the developer expense, *D*, and in the critical expense to failure-free, *FF*. Both parallel techniques provide a savings over the sequential mode. This means that the fault-focusing ability of either clustering technique has economic benefits as measured in expense.

These results demonstrate that even for a single developer, clustering failing test cases will be beneficial. The *D* score shows the expense that would be incurred if the developer were to debug the program *sequentially* using each of these methods (*S*, *P1*, and *P2*). For example, suppose the developer were to use the *P2* technique of clustering, but were to debug the program one fault at a time. Tables 1 and 2 show both total developer expense (*D*) and the total expense to failure free (*FF*) that he would expend. The results show that, regardless of whether debugging in sequence or in parallel, clustering failed test cases reduces developer expense in finding the faults.

The choice of *P1* or *P2* may depend on the development organization's resources and circumstance. If the goal is to deliver a failure-free program as fast a possible, then *P1* may be a better choice than *P2*. However, if the goal is to minimize development expense, then *P2* may provide a net savings. We investigated this trade-off to determine the reason that each technique demonstrated different strengths. The *P2* technique seems to cluster more aggressively than *P1*. *P1* and *P2* respectively have an average of 2.08 and 1.62 parallel fault subtasks across all versions and iterations. *P1* may incur more total expense due to the under-clustering situation described in Section 5 and depicted in Figure 13. Moreover, because each of the techniques that we implemented has merit, we reason that clustering executions for the purpose of fault-localization may be conducted in a number of ways to good effect. Although more research is necessary to determine the best clustering technique, we have demonstrated the promise of parallelizing the debugging effort in such an automated way.

## 6.6 Threats to Validity

Although this empirical study provides evidence of the potential usefulness of the parallel-debugging techniques developed in this research, there are several threats to the validity of the empirical results that should be considered in their interpretation.

Threats to the external validity of an experiment limit generalizing from the results. The primary threat to external validity for this study arise because only one medium-sized C program has been considered. Thus, we cannot claim that these results generalize to other programs. In particular, no generalization can be made as to the effectiveness of parallel debugging. However, a variety of faults were randomly combined to produce the 100 8-fault versions used in this research and thus, these versions are useful for exploring the presented techniques.

Threats to the internal validity occur when there are unknown causal relationships between independent and dependent variables. In this study, we have postulated a simplistic development scenario that removes these causal relationships. However, for real developers, there will be causal relationships between total expense and the debugging mode chosen. For example, developers will interact with each other, which may change the expense in either direction.

Also, we assume that a developer can identify the fault by inspecting the code—that is, she can follow the order of statements that is specified and determine at each one whether it is faulty. We do think that the amount of code that must be examined while following the prescribed order of the fault-localization technique is indicative of the technique's effectiveness. This issue must be explored further with human studies.

The integration of multiple bug fixes may be more error-prone than one-at-a-time bug fixing. This may cause new bugs to be introduced as the parallel debugging proceeds. Our experiment does not address this difficulty; further studies are needed to explore this difficulty.

# 7. RELATED WORK

The main component of our technique is the automatic clustering of failing executions according to their causes. Dickinson and colleagues show that clustering of executions can isolate failing executions from passing executions [5]. In later work, Podgurski and colleagues show that profiles of failing executions can be automatically clustered according to similar causes or faults [13]. Their approach depends on a supervised classification strategy informed by multivariate visualizations that assist the practitioner. In contrast, our technique is completely automated and attempts to cluster failed executions according to their root cause by combining information from execution profiles with information about the relative failure-causing suspiciousness of lines of code.

Zheng and colleagues present an approach to finding bug predictors in the presence of multiple faults [18]. The authors show that test runs can be clustered to give a different bug-predictor profile or histogram. They also present a result that is similar to our findings: that some bug predictors dominate others—they call these super bug predictors. We found a similar results although from a different perspective: we found that some faults prevent others from being active. Beyond this, our work differs from theirs in that we present a methodology for debugging multiple faults in parallel. Also, we present an experiment that presents the costs of debugging multiple faults.

Liu and Han present two pairwise distance measures for failing test cases [11]. They demonstrate the difference of a profile-based distance measure (usage mode) and fault-localization-based distance measure (failure mode) by means of multidimensional-scaling plots. For their subject programs and plots that they present, they propose that the failure-mode distance measure is better able to isolate failures caused by different faults. Our work differs from theirs in a number of ways. First, unlike their multidimensional plots of executions, our work provides an automatic way to cluster failed test cases without interpretation by the developer. Second, our experiments do not confirm their finding that usage-mode or profile-based distances is inferior to failure-mode or fault-localization-based distances. Although, we are not able to generalize to other programs, our experiments show that each type of clustering may have its own strengths. Finally, their work targets a sequential-mode of debugging by removing faults that are creating noise for finding the most dominant fault at each iteration. Our work aims to enable the parallelization of the debugging task.

# 8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a description of a parallel debugging process that offers an alternative to the conventional sequential debugging process by both reducing total developer cost and reducing the time to a failure-free program. We presented two parallel-debugging techniques that create specialized sets of test cases that can be assigned to different developers for simultaneous debugging. We also have developed a novel technique for establishing a useful stopping criterion in the clustering of failing executions, providing an early prediction of the number of active faults.

We also presented an empirical study that demonstrates the cost-saving potential of these two parallel-debugging techniques. Notable in the empirical results is that, for the object studied, sequential debugging is the most expensive both in developer expense, *D*, and in the critical expense to failure-free, *FF*. Each of the parallel-debugging techniques provides a savings over sequential debugging. This means that the fault-focusing ability of either clustering technique has potential economic benefits. It also means that even a single developer will benefit by using fault-focused test suites as

a debugging aid. Both of our clustering techniques showed merit. This result demonstrates the promise of clustering executions for the purpose of localizing multiple faults. Other clustering technique may be shown to have their own merits.

Finally, the two metrics—total developer expense and critical expense to a failure-free program—can provide parameters for a cost model for parallel debugging where a developer can decide the combination of factors that are best for his resources and time requirements. These metrics can be utilized to evaluate future research into the parallelization of debugging.

We have identified a number of research directions for future work that result from this work. First, during parallel debugging, one developer could finish his debugging while another developer is still debugging. In this situation, fixes could be distributed to other developers or one bug fix may affect the debugging efforts of another developer. We plan to investigate enhancing our parallel debugging technique so that it will provide recommendations for such situations. Second, organizational and situational constraints will likely dictate the best way to debug in parallel. For example, an imminent release date may call for a more aggressive parallelization if redundant developers can be afforded in an effort to quickly resolve critical bugs. Also, an organization may have a limited number of developers—the parallelization should take this into account. We plan to develop a cost model that is informed by the program and test suite as well as organizational constraints to customize the technique. Third, assignment of suspected faults and specialized test suites to the developers that will debug them can be automated. Based on information such as source-code revision history, ownership or familiarity of the suspected faulty code can be mapped to developers. We plan to explore the possibility of automatically assigning developers to faults when multiple faults can be debugged simultaneously in the same spirit as presented in Reference [1]. Finally, we plan to perform further experiments on more subject programs of larger sizes with a varying number of faults.

## 9. ACKNOWLEDGMENTS

This work was supported in part by NSF awards CCF-0541049, CCF-0429117, and CCF-0306372 to Georgia Tech, by Tata Consultancy Services, Ltd. The anonymous reviewers provided many helpful suggestions to improve the paper. Andy Podgurski also provided many suggestions to improve the paper.

## 10. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceeding of the International Conference on Software Engineering*, pages 361–370, May 2006.

[2] Aristotle Research Group. ARISTOTLE analysis system, 2007. `http://www.cc.gatech.edu/aristotle/`.

[3] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 195–205. ACM Press, July 2004.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the International Conference on Software Engineering*, pages 342–351, May 2005.

[5] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the International Conference on Software Engineering*, pages 339–348, May 2001.

[6] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley and Sons, Inc., 2001.

[7] J. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the International Conference on Automated Software Engineering*, pages 273–282, November 2005.

[8] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, pages 467–477, May 2002.

[9] J. Jones, A. Orso, and M. Harrold. Gammatella: Visualizing program-execution data for deployed software. *Information Visualization*, 3(3):173–188, Autumn 2004.

[10] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2005.

[11] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 286–295, November 2006.

[12] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of European Software Engineering Conference and Foundations on Software Engineering*, pages 286–295, September 2005.

[13] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the International Conference on Software Engineering*, pages 465–474, May 2003.

[14] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering*, pages 30–39, October 2003.

[15] I. Vessey. Expertise in debugging computer programs. *International Journal of Man-Machine Studies: A process analysis*, 23(5):459–494, 1985.

[16] F. Vokolos and P. Frankl. Empirical evaluation of the textual differencing regression testing techniques. In *Proceedings of the International Conference on Software Maintenance*, November 1998.

[17] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the International Conference on Software Engineering*, pages 272–281, May 2006.

[18] A. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *Proceedings of the International Conference on Machine Learning*, pages 1105–1112, June 2006.