Mitigating the Confounding Effects of Program Dependences for Effective Fault Localization

George K. Baah
Georgia Institute of Technology
Atlanta, GA 30332
baah@cc.gatech.edu

Andy Podgurski
Case Western Reserve University
Cleveland, OH 44106
podgurski@case.edu

Mary Jean Harrold
Georgia Institute of Technology
Atlanta, GA 30332
harrold@cc.gatech.edu

ABSTRACT

Dynamic program dependences are recognized as important factors in software debugging because they contribute to triggering the effects of faults and propagating the effects to a program's output. The effects of dynamic dependences also produce significant confounding bias when statistically estimating the causal effect of a statement on the occurrence of program failures, which leads to poor fault-localization results. This paper presents a novel causal-inference technique for fault localization that accounts for the effects of dynamic data and control dependences and thus, significantly reduces confounding bias during fault localization. The technique employs a new dependence-based causal model together with matching of test executions based on their dynamic dependences. The paper also presents empirical results indicating that the new technique performs significantly better than existing statistical fault-localization techniques as well as our previous fault localization technique based on causal-inference methodology.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Debugging aids, Diagnostics*

General Terms: Algorithms, Experimentation

Keywords: causal inference, potential outcome model, matching, fault localization, program analysis, debugging

1. INTRODUCTION

The pervasive impact of software systems and applications on society requires software developers to ensure that their products are of high quality. However, software development is a human process, and developers unintentionally introduce faults into software that cause it to fail under certain conditions. When software failures are observed, developers engage in *debugging* to find and fix the fault(s). *Fault localization* is the part of debugging that involves locating faults in a program. Because of the size and complexity of many software products, fault localization is often a difficult and time-consuming task, which contributes significantly to the cost of software development and maintenance.

To reduce the cost of fault localization, researchers have proposed automated statistical fault-localization techniques (e.g., [1, 4, 13, 15, 16, 17]). These techniques compute statistical measures of association between the execution of individual program statements (or other entities) and the occurrence of program failures, using execution data from passing and failing executions (e.g., coverage information). The measures are called *suspiciousness scores*. The techniques rank program statements in decreasing order of their suspiciousness scores for examination by developers. The intuition underlying this approach is that the program statements most strongly associated with failures are the most likely to be faulty. However, statistical association does not necessarily indicate causation [12]. Thus, statements with high suspiciousness scores may not be faulty, and faulty statements may have low suspiciousness scores. Consequently, existing statistical fault-localization techniques can perform poorly with certain programs and faults.

In recent work [5], we showed that statistical fault-localization techniques based on measures of association between the occurrence of certain runtime events (e.g., coverage of a given statement) and the occurrence of program failures are subject to *confounding bias*, because of influences between statements in a program. For example, with a number of coverage-based statistical fault-localization techniques, a correct statement s_c and a faulty statement s_f that happen to be covered by the same subset of test cases will necessarily receive the same suspiciousness score. More generally, any other statement that influences s_c and s_f and also influences the occurrence of failures will confound the suspiciousness scores of s_c and s_f. To accurately assess the distinct effects of covering s_c and s_f on the occurrence of failures, it is necessary to adjust for such confounding factors.

To address the confounding influence of some program statements on the suspiciousness scores of other statements, we presented a preliminary *causal inference model* for software fault localization [5] (see Section 2). We used this model to estimate the *causal effect* of covering a statement s on the occurrence of program failures. To "control" or "adjust" for confounding by other statements, the model incorporates a coverage-indicator variable for the forward control dependence predecessor of s (in addition to a coverage indicator for s itself). The model is grounded in the extensive body of research on making causal inferences from observational data, which spans such diverse fields as econometrics, the social sciences, epidemiology, statistics, and computer science [18, 21]. We contrasted the model analytically with several existing fault-localization metrics that do not address confounding, and presented empirical results indicating that the model (1) is significantly more effective than some of those techniques and (2) can be incorporated into other techniques to improve their performance. These results suggest that existing methodology for making causal inferences from observational data is very relevant to statistical fault localization.

However, it is easy to see that our previous model does not address all possible sources of confounding in fault localization, such as runtime patterns of data dependences (data flows), the values taken on by inputs and program variables, and non-determinism because of concurrency [5]. Making the best use of causal-inference methodology in software fault localization will require determining the relative importance of such factors and devising effective and reasonably efficient ways of controlling for them.

In this paper, we present a new technique that controls for local patterns of both dynamic data dependences and dynamic control dependences that can confound the estimated causal effect of covering a statement s on the occurrence of program failures, which for brevity we call the *failure-causing effect* of s. Static and dynamic data and control dependences have long been recognized as important factors in software testing and debugging, because they contribute both to triggering the effects of faults and to propagating those effects to a program's output [22, 24, 26]. Our technique uses information about dynamic data and control dependences to reduce confounding bias—more than is possible with our previous model—and to thereby rank statements more effectively for fault localization.

When estimating the failure-causing effect of a statement s, confounding bias due to dynamic data and control dependences affecting s can be reduced by considering two groups of executions: those that cover s (referred to as the *treatment group*) and those that do not cover s (referred to as the *control group*). The executions that cover s should have the same pattern of dynamic data and control dependences as the executions that do not cover s, in which case the groups are said to be *balanced*. In practice, the set of test cases or operational executions used in statistical fault localization is usually given, and the subset of them that cover and do not cover a particular statement s cannot be assumed to be balanced.

To achieve relative balance between the treatment and control groups and thus reduce confounding bias, our technique employs a classical causal-inference technique called *matching*. For each program statement s, the executions in the two groups are reorganized so that they are similar with respect to dynamic data and control dependences that directly affect s. Each execution e that covers s is matched with the execution not covering s that is most similar to e, as measured by applying the Mahalanobis-distance metric [18] to dynamic dependence information collected from the executions. After matching, the failure-causing effect of s is estimated based on the outcomes of the matched executions.

In this paper, we also present empirical results indicating that our new technique yields fault-localization rankings that are more effective than those obtained with our previous causal model [5] and with several well-known statistical fault-localization techniques. For example, our new technique performs better than our old technique on 61.4% of 272 faulty program versions, with an average fault-localization improvement of 8.57%. Our new technique also performs better than Tarantula [13] on 72.06% of the versions, with an average fault-localization improvement of 11.44%.

The main benefit of our work is that it provides a practical way to use both data and control dependence information to reduce confounding bias and thereby improve the effectiveness of statistical fault localization. Another benefit is that basing our technique on accepted causal-inference methodology will help to put statistical fault localization on a sounder footing and stimulate further improvements.

The contributions of the paper are

- A new statistical fault-localization technique that uses a new causal model, based on dynamic data and control dependences, together with covariate matching to more accurately estimate the causal effect of covering a particular program statement on program failures.
- Empirical results indicating that our new technique is more effective than our previous causal technique and other statistical fault-localization techniques.

2. BACKGROUND

In this section, we first present background on the potential-outcome model for causal inference and the dynamic program-dependence graph. We then describe our previous causal-inference model for statistical fault localization [5], which is based on the potential outcome model [18], on Pearl's Structural Causal Model [21], and on dynamic program-dependence analysis [10].

2.1 Potential Outcome Model

The *potential-outcome model* [18] provides a theoretical basis for estimating causal effects using data from either observational studies or randomized experiments. Suppose that a researcher wants to estimate the effect of a new or existing treatment, and suppose further that a set of units or subjects are available to the researcher. Let T be a binary variable that takes on the value $t_i = 1$ for a unit u_i if u_i receives the treatment and that takes on the value $t_i = 0$ if u_i does not receive the treatment. Let Y represent the outcome for an arbitrary unit. The potential outcome model associates with Y two *potential-outcome random variables*: Y^1 and Y^0. Variable Y^1 represents the potential outcome for a *treated* unit, and variable Y^0 represents the potential outcome for an *untreated* (control) unit. The *individual-level causal effect* of treatment for a unit u_i is defined as $y_i^1 - y_i^0$, where y_i^1 and y_i^0 are the realizations of Y^1 and Y^0 for unit i. We assume that for any given unit, only one of Y^1 and Y^0 can be observed; the other variable's value is *counterfactual* (counter to the facts) for that unit. This implies it is impossible to estimate $y_i^1 - y_i^0$ directly.[1]

Because individual-level treatment effects cannot be estimated, *average treatment effects* are estimated instead. For the scenario above, the average treatment effect τ is defined to be the difference between the expected outcomes of the two groups:

$$\tau = E[Y^1] - E[Y^0] \tag{1}$$

where E[·] is the expectation operator. Suppose that a *randomized experiment* is conducted in which a researcher ran-

[1]This problem has been called the *fundamental problem of causal inference* [12].

domly assigns units to either the treatment group or the control group, and suppose an outcome y_i is observed for each unit u_i. Then the difference $\bar{y}_1 - \bar{y}_0$ between the sample means for the treatment group and the control group, respectively, is an unbiased estimator of τ. The estimate is unbiased because random treatment assignment tends to ensure that the treatment group and the control group are balanced with respect to the distributions of confounding variables, whether or not the variables are observed. Consequently, the treatment variable T is independent of the potential outcomes Y^1 and Y^0,

$$(Y^1, Y^0) \perp\!\!\!\perp T \tag{2}$$

and so knowing whether a unit is assigned to treatment does not provide any information about Y^1 and Y^0.

It is typically assumed that execution data to be used for statistical fault localization is generated from a set of existing test cases or is collected in the field from operational executions. In both cases, statistical fault localization is a kind of observational study rather than a randomized experiment, because the executions were not randomly assigned to either the treatment group (e.g., executions covering a statement s) or the control group (e.g., executions not covering s). Note that creating test cases specifically to achieve coverage of every conditional branch does not ensure that the test cases that cover a statement s are otherwise similar to the test cases that do not cover s.

To estimate average treatment effects based on observational data, statistical techniques that "control for" or "condition on" the observed values of potential confounding variables [18] are used. An unbiased estimate of the average treatment effects of treatment variable T on outcome variable Y can be obtained by conditioning on a set X of covariates that render T *conditionally independent* of the potential outcome variables Y^1 and Y^0,

$$(Y^1, Y^0) \perp\!\!\!\perp T \mid X \tag{3}$$

In this case, the average treatment effect τ is given by the equation

$$\tau = E[E[Y \mid T = 1, X] - E[Y \mid T = 0, X]] \tag{4}$$

where $\mathrm{E}[\cdot \mid \cdot]$ is the conditional expectation operator. (The outer, unconditional expectation is evaluated with respect to the probability distribution of X.)

One way of controlling for observed confounders when estimating an average treatment effect is to include them as predictors in a *statistical regression model* [18], such as the linear model

$$Y = \alpha + \tau T + \beta X + \varepsilon \tag{5}$$

In this model, Y is the outcome variable, T is the treatment variable, X is a vector of covariates that includes the observed confounders, and ε is a random error term that (ideally) does not depend on the values of T and X. Equation 5 implies that $E[Y \mid T, X] = \alpha + \tau T + \beta X$. The fitted value $\hat{\tau}$ of the coefficient τ is an unbiased average-treatment-effect estimate, assuming the model is correctly specified.

2.2 Dynamic Program Dependences

Informally, in a program P, statement s_2 is *control dependent* [10] on a statement s_1 if, as indicated by the control-flow graph (CFG) for P,[2] s_1 determines whether s_2 is executed. Statement s_1 *dominates* statement s_2 in the CFG if all paths from P's entry to s_2 contains s_1. The *control dependence graph* (CDG) of P is a directed graph where nodes represent statements in P and edges represent control dependences. Statement s_1 is a *forward control-dependence predecessor* of s_2 in the CDG if s_2 is control dependent on s_1 and s_2 does not dominate s_1 in the CFG.

Statement s_2 is data dependent on statement s_1 if there is a definition of a variable v (i.e., a write of v) at s_1, a use of v (i.e., access with no write) at s_2, and there is a least one path from s_1 to s_2 in the CFG on which v is not redefined. The *program-dependence graph* (PDG) of program P is a directed graph whose nodes represent statements in P and whose edges represent control and data dependences. The graph is annotated with two special nodes, START and EXIT, which represent entry to and exit from P, respectively.

Informally, the *dynamic program dependence graph* (Dynamic PDG) is a directed graph constructed from a set of program executions in which nodes represent executed statements and edges represent executed control dependences and data dependences. Statement s_1 is a *dynamic forward control dependence predecessor* of s_2 if s_1 is a forward control-dependence predecessor of s_2 in the Dynamic-PDG. The Dynamic-PDG, which is a subgraph of the PDG, is similar to the graph obtained with Agrawal and Horgan's [2] technique (Approach 1) for dynamic slicing. However, the latter is created using one execution whereas the Dynamic-PDG described here is created using a set of executions. We define the *dynamic forward control dependence subgraph* as the subgraph of the Dynamic-PDG that contains only forward control-dependence edges.

2.3 Previous Fault Localization Model

To specify a causal model for statistical fault localization, it is necessary to indicate the form of the model, the treatment variable, and the covariates. Our initial causal model [5] includes a linear regression model for estimating the failure-causing effect of covering a program statement s:

$$Y = \alpha_s + \tau_s T_s + \beta_s C_s + \varepsilon_s \tag{6}$$

This model is fit separately for each statement in a program, using statement-coverage profiles from a set of passing and failing executions. In the model, Y is a binary variable that is 1 for a given execution if and only if the program failed; T_s is a binary treatment variable, which indicates whether s was covered (executed at least once) during the execution; C_s is a binary covariate, which indicates whether the dynamic forward control dependence predecessor $dfcdp(s)$ of s was covered; α_s is a constant intercept; and ε_s is a random error term that does not depend on the values of T_s and C_s. The average treatment effect of T_s upon Y is estimated by the fitted value $\hat{\tau}_s$ of the coefficient τ_s, which is used as a suspiciousness value for s.

A regression model such as Equation 6 describes only statistical relationships between variables. To obtain a full causal model, it must be augmented with additional information about causal relationships. This information can be represented by a *causal graph* [21], which is a directed acyclic

[2]In a *control-flow graph*, nodes represent program statements, and edges represent the flow of control between the statements.

		t_1	t_2	t_3	t_4	t_5		
		(-1,1)	(2,3)	(1,-3)	(0,5)	(3,4)	$\hat{\tau}$	$\hat{\tau}_{match}$
	void Proc() {							
1	int x=read();	1	1	1	1	1	0.40	NA
2	int y=read();	1	1	1	1	1	0.40	NA
3	if(x > 0){	1	1	1	1	1	0.40	NA
4	if(y < 0)	0	1	1	0	1	0.67	0.67
5	y = 2;	0	0	1	0	0	-1.00	-1.00
6	print("Out:");	0	1	1	0	1	0.67	0.67
7	print(y+y);	0	1	1	0	1	0.67	1.00
8	}							
9	}							
		0	1	0	0	1		

Figure 1: Procedure with test cases, execution data, and causal-effect estimates. Error at statement 7; correct computation should be y×y.

graph (DAG) whose nodes represent random variables (corresponding to causes and effects) and whose edges represent causal relationships. An edge $A \rightarrow B$ in a causal graph indicates that A (potentially) causes B. Equation 6 is implicitly based on the causal graph obtained from a (structured) program's dynamic forward control dependence subgraph, by (1) associating with each node a coverage-indicator variable for the corresponding statement and (2) augmenting the dynamic forward control dependence subgraph with edges from each node to a new node for the failure indicator Y.

The role of the coverage indicator variable C_s for *dfcdp*(s) in Equation 6 is to control for confounding of the estimated failure-causing effect of s, due to coverage of other statements. Intuitively, conditioning on C_s reduces confounding because *dfcdp*(s) is the most immediate cause of s being covered or not being covered on a particular run. Pearl's well-known *Back-Door Criterion* [21] for causal graphs provides some formal justification for Equation 6, because *dfcdp*(s) blocks all "back-door paths" from T to Y in the augmented dynamic forward control dependence subgraph. (Such a path is actually a *semi-path*—one in which edges may point in either direction—whose first edge is of the form $T \leftarrow Z$ for some node Z.) Under the very strong assumption that the augmented dynamic forward control dependence subgraph is sufficient to model failure causation in a program, the Back-Door Criterion implies that T is conditionally independent of the potential outcomes Y^1 and Y^0 given the value of C_s, as required for Equation 4 to hold with T_s and C_s in place of T and X, respectively.

3. ADDRESSING BOTH DATA AND CONTROL DEPENDENCES

The causal model, given in Equation 6, which was used in our previous technique for statistical fault localization relies on the dynamic forward control dependence subgraph as a model of failure causation. However, the dynamic forward control dependence subgraph, and therefore Equation 6, does not adequately reflect failure causation in which data dependences play an important role. In this section, we present our new technique for estimating a statement's failure causing effect, which addresses confounding due to both dynamic data dependences and dynamic control dependences by matching executions based on information about both.

We begin with an example that illustrates the importance of incorporating data dependences into our causal model, and then, present the details of the model.

3.1 Motivating Example

Consider procedure `Proc` in Figure 1, which has a fault at line 7. `Proc` should print the square of the value of `y` at line 7 but instead prints two times the value of `y`. The first column shows `Proc` with line numbers associated with each of its statements. Columns 2 through 6 represent test cases t_1–t_5, respectively. The top entry of each column shows the values of `x` and `y` that are read at lines 1 and 2, respectively. The numbers in the column for a test case indicate whether the corresponding program statement is covered by the test case (1 for covered, 0 for not covered). The bottom row shows the outcome of each test-case execution, with 1 indicating a failing execution and 0 indicating a passing execution.

Suppose we want to compute the failure-causing effect of statement 7 using Equation 6. For statement 7, as Figure 1 shows, test cases t_2, t_3, and t_5 are in the treatment group, because they cover statement 7, and test cases t_1 and t_4 are in the control group, because they do not cover statement 7. The dynamic forward control dependence predecessor of statement 7 is statement 3. Variable C_s in Equation 6 corresponds to a coverage indicator C_3 for statement 3, which is covered by all tests. (Because C_3 is constant, it or the intercept α_s from Equation 6 can be dropped from the model for statement 7.) The column labeled $\hat{\tau}$ in Figure 1 indicates the failure-causing effect estimates obtained with Equation 6 for the statements in procedure `Proc`. The estimate for statement 7, which is faulty, is 0.67. However, the estimates for statements 4 and 6 are also 0.67, even though they are not faulty. Statements 4 and 6 have the same estimate as statement 7 because they are in the same control dependence region [10] and hence, are covered by the same tests. This example illustrates that serious confounding may occur even after conditioning on coverage of each statement's dynamic forward control dependence predecessor.

3.2 New Technique

Our new technique addresses the inadequacy of our previous technique for fault localization. The technique consists of two main components: a new causal model and a matching technique.

3.2.1 Overview

Algorithm `LocalizeFault`, shown in Figure 2, takes as input the dynamic program dependence graph (Dynamic-PDG) of a procedure and the coverage information (Coverage-Info) produced by executing the program containing the procedure on a set of test cases. `LocalizeFault` processes each statement s in the Dynamic-PDG; it initializes the causal estimate of each statement s, ($\hat{\tau}(s)$), to -1.0 (line 2). `LocalizeFault` computes the first major component of our technique, the causal model of s (Model(s)) (line 3). At line 4, the algorithm computes the predecessors of s (Pred(s)) from Model(s). After computing Pred(s), `LocalizeFault` then computes the second major component of our technique by matching on the Pred(s) in the Coverage-Info to produce the matched data, Mdata(s) (line 5). If matching is successful, the $\hat{\tau}(s)$ is estimated using Equation 1 (line 7); if matching is not successful Equation 5 is used to estimate $\hat{\tau}(s)$ (line 9). The algorithm computes $\hat{\tau}(s)$ for every state-

```
   LocalizeFault(Dynamic-PDG, Coverage-Info){
1    foreach s ∈ Dynamic-PDG do
2       $\hat{\tau}$(s) = -1.0
3       Model(s) = Compute causal model of s
4       Pred(s) = Compute predecessors of s from Model(s)
5       Mdata(s) = Match on Pred(s) in Coverage-Info
6       if (Mdata(s) $\neq \emptyset$){
7         Compute $\hat{\tau}$(s) from Mdata(s) using Equation 1
8       }else{
9         Compute $\hat{\tau}$(s) using Equation 5
10      }
11   done
12   sorted_$\hat{\tau}$ = sort $\hat{\tau}$ in descending order
13   return (sorted_$\hat{\tau}$)
14 }
```

Figure 2: Algorithm for new technique

ment in the Dynamic-PDG. After the causal estimate for each statement has been computed, `LocalizeFault` sorts all the estimates in descending order at line 12. It then returns the sorted estimates at line 13 to the developer.

3.2.2 New Causal Model

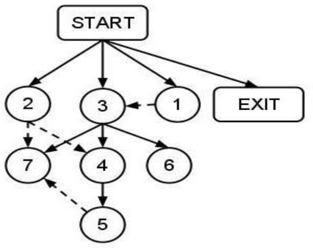
Our new causal model extends our previous causal model (Equation 6) by addressing dynamic data dependences as well as dynamic control dependences. Dynamic data dependences are important because they carry the values that are used at a given statement. Whereas a statement's forward control dependence predecessor determines whether the statement is covered, the statement's data dependences determine the computation it carries out. Conceptually, the additional causal influences that we want to account for can be represented by including dynamic data dependences in the causal graph for a program, in addition to dynamic forward control dependences. A complication is that loop-carried data dependences [10] give rise to directed cycles in dependence graphs, whereas much of the theory of causal inference developed by Pearl [21] and other authors, including the Back Door Criterion, is based on causal graphs that are acyclic. However, to control confounding when estimating the failure-causing effect of a particular statement s_i, it suffices to consider acyclic dependence chains terminating at s, because if there is a causal path from s_j to s_i that contains one or more cycles, there must also be a cycle-free path from s_j to s_i. For example, consider a program loop of the form

```
1 while(...)
2    if (...)
3       x = f(y);
4    else
5       y = g(x);
```

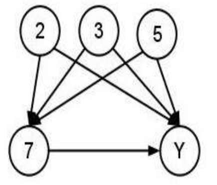
Each edge in the data dependence cycle $3 \rightarrow 5 \rightarrow 3$ may be relevant to localizing a distinct fault. However, in seeking to control confounding while estimating the failure-causing effect of statement 3, we can ignore the edge (3, 5). Similarly, when estimating the failure causing effect of statement 5, we can ignore the edge (5, 3).

The kind of causal graph that is required can be conceptualized in terms of the *transitive reduction* of a digraph [3]. In essence, a transitive reduction of a digraph D is an acyclic, spanning subdigraph H of D with no redundant arcs such that the transitive closures of D and H are isomorphic [6].[3]

[3]Klamt and colleagues employ a form of transitive reduction to model causality in biological networks with cycles [14].



(a) Dynamic-PDG of `Proc`.



(b) ICG of statement 7.

Figure 3: (a) Dynamic program dependence graph (Dynamic-PDG) of procedure `Proc`. (b) Integrated causal graph of statement 7.

Aho and colleagues [3] present a transitive reduction algorithm that, for a digraph with cycles, replaces each equivalence class of vertices appearing in the same cycle with a new vertex. Consider a slight variant of this algorithm, applied to a graph of dynamic data and control dependences between program statements. For a given statement s whose failure-causing effect we want to estimate, this variant preserves all nodes in s's equivalence class (if there is more than one) rather than collapsing them to a single node. However, it breaks any cycles involving s by deleting s's outgoing edges. Therefore, for each statement s in the dynamic program dependence graph, an acyclic subgraph H_s can be constructed that reflects all causal influences on s. H_s can be transformed into a proper causal graph by augmenting it as described in Section 2. We call the augmented H_s the *integrated causal graph* for s and denote it $ICG(s)$.[4]

In the integrated causal graph $ICG(s)$, there will be multiple back doors to s if it is dependent on multiple statements. Observe, however, that the set of predecessors of s ($Pred(s)$) blocks all back-door paths from s to the failure node Y. Thus, by conditioning on the coverage indicator variables associated with $Pred(s)$, we may be able to further reduce confounding when estimating the failure-causing effect of s. For example, Figure 3(a) shows the dynamic program dependence graph of `Proc`; dotted edges represent data dependences and solid edges represent control dependences. Figure 3(b) shows the ICG of statement 7, where nodes 2, 3, and 5 of statement 7 are nodes in the dynamic program dependence graph of `Proc`. As the graph shows, there are three back-door paths in the graph: $7 \leftarrow 2 \rightarrow Y$, $7 \leftarrow 3 \rightarrow Y$, and $7 \leftarrow 5 \rightarrow Y$.

3.2.3 Matching

Matching [18] is a technique that brings some of the benefits of randomized controlled experiments, in terms of reduced confounding bias, to observational studies. Matching involves mapping (if possible) each treatment unit from an observational study to one or more control units that are similar to it, in such a way that *balance* is achieved between the resulting treatment group and control group with respect to covariate values. Matching may involve removing

[4]One subtlety is that if a node n in $ICG(s)$ represents an equivalence class C of nodes from D, a set Z of coverage indicator variables should be associated with n. For each statement t represented in C, Z should contain a coverage indicator for t.

unmatched units or (in effect) duplicating certain units. We first describe a simple form of matching, called exact matching, before describing the more complex type of matching used with our new causal model.

In *exact matching*, each treatment unit is matched with one or more control units that have exactly the same covariate values as the treatment unit, if such control units exist. All matched treatment and control units are retained, and all unmatched units are discarded. The difference in the group means of the resulting treatment group and control group is an estimate of the treatment effect.

We now illustrate exact matching with reference to the procedure and data in Figure 1. Let the covariates for each statement s in procedure `Proc` be the coverage-indicator variables associated with the possible dynamic control and data dependence predecessors of s. For example, for statement 7, there are three predecessors: statement 2, which defines the value of `y` and may be used at statement 7; statement 3, which is statement 7's forward control dependence predecessor); and statement 5, which defines the value of `y` and may be used at statement 7. Figure 4 shows the treatment units and the control units with their associated test cases for statement 7. The treatment units, which are test cases that cover statement 7 are t_2, t_3, and t_5; the control units, which are test cases that do not cover statement 7 are t_1 and t_4. The vector of covariate values generated by the test cases is shown; a vector of the form [1, 1, 0] means that for that test case, statement 2 was covered, statement 3 was covered, and statement 5 was not covered, respectively. There is an arrow from each treatment unit to a matching control unit. Note that there is no match for t_3, because it is the only test case to cover each of statement 2, statement 3, and statement 5. Consequently, t_3 is not used to estimate the failure-causing effect of statement 7. Using Equation 1, the estimate for statement 7 is $(1 + 1)/2 - (0 + 0)/2 = 1.0$, which is the difference between the average outcome values for the treatment group and the average outcome values for the control group. The causal-effect estimates for the other statements are shown in Table 1, in the column labeled $\hat{\tau}_{match}$. (The NA for statement's 1, 2, and 3 reflects the fact that an estimate could not be computed for the statements because there were no control units for the statements.) The estimate for statement 5 is $0 - 1/1 = -1.0$ and the estimate for each of statements 4 and 6 is $(1+1+0)/3 - (0+0)/2 = 0.67$. It can be seen that the faulty statement, statement 7, has the highest estimate.

Unfortunately, as the number of covariates increases, it becomes more difficult to find exact matches for treatment units. This difficulty results in more discarded units, which can increase bias[5] when computing causal estimates. Therefore, other forms of matching are typically used.

Matching with Mahalanobis Distance

To obtain more flexibility in matching treatment units with control units, our technique uses the *Mahalanobis distance* metric [18], which is a measure of the similarity $d_M(\mathbf{a}, \mathbf{b})$ between two random vectors $\mathbf{a}$ and $\mathbf{b}$. Let S be the covariance matrix of $\mathbf{a}$ and $\mathbf{b}$ and let the superscript T denote

[5]This form of bias is different from confounding bias. The bias is the difference in an estimator's expected value and the true value of the parameter being estimated.

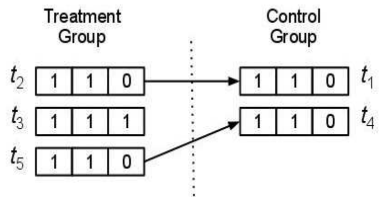


Figure 4: Units in treatment group and control group with their covariate values.

matrix/vector transpose. Then

$$d_M(\mathbf{a}, \mathbf{b}) = \sqrt{(\mathbf{a} - \mathbf{b})^T S^{-1} (\mathbf{a} - \mathbf{b})} \qquad (7)$$

The matrix S is the sample covariance matrix for the treatment and control units. Each treatment unit (test case covering statement s) with covariate vector $\mathbf{a}$ is matched with a control unit (test case not covering s) with covariate vector $\mathbf{b}$ for which $d_M(\mathbf{a}, \mathbf{b})$ is minimal with respect to a threshold. In matching *without replacement*, the two units are then removed from further consideration. In matching *with replacement*, the control unit is retained to be matched with another treatment unit. Replacement is normally used if the number of treatment units exceeds the number of control units; matching with replacement also reduces bias[5] [29] in the causal estimate. A property of Mahalanobis-distance matching is that it regards all the components of the covariate vector as equally important [29]. This property is important when matching on the predecessors of s because in the absence of any prior information about the relative importance of the predecessors, our technique treats all predecessors of s as equally important.

4. EMPIRICAL EVALUATION

To evaluate the effectiveness of our technique for fault localization, we implemented it and performed empirical studies on a set of subjects. In this section, we first describe the set-up and then present the results of the studies.

4.1 Empirical Study Setup

For our studies, we used 16 programs with 272 faulty versions. We used seven programs from the Unix suite (Cal, Col, Comm, Spline, Tr, and Uniq), all programs in the Siemens suite (Print-tokens, Print-tokens2, Replace, Schedule, Schedule2, Tcas, and Tot-info), and programs Sed and Space.[6] Table 1 shows the characteristics of the subjects. For each subject, the first column gives the program name, the second column provides the ratio of the number of versions used to the number of versions available, the third column gives the number of lines of code for the subject, the fourth column gives the number of test cases, the fifth column shows the average number of vertices in the dynamic program dependence graph, and the last column provides a description. We omitted 25 faulty versions because either there were no syntactic differences between the C files of the correct version and the faulty version of the program or none of the test cases failed when executed on the faulty version of the program.

[6]We obtained the Unix programs from Eric Wong of University of Texas at Dallas and the Space and Sed programs from the Software-artifact Infrastructure Repository [9]. We obtained the Siemens suite from the Aristotle Research Lab.

Table 1: Subjects used for empirical studies.

Program	Number of Versions Used / Number of Versions	Number of Lines of Code	Number of Test Cases	Number of Nodes	Description
Cal	19 / 20	202	162	131.5	calendar printer
Col	29 / 30	102	156	240.0	filter-line reverser
Comm	10 / 12	167	186	116.1	file comparer
Look	8 / 14	170	193	140.9	word finder
Spline	13 / 13	338	700	247.0	curve interpolator
Tr	11 / 11	137	870	150.2	character translator
Uniq	17 / 17	143	431	131.3	duplicate line remover
Print-tokens	5 / 7	472	4130	423.6	lexical analyzer
Print-tokens2	10 / 10	399	4115	340.5	lexical analyzer
Replace	28 / 32	512	395	415.9	pattern replacement
Schedule	9 / 9	292	2710	216.3	priority scheduler
Schedule2	9 / 10	301	2650	225.9	priority scheduler
Tcas	41 / 41	141	1608	136.7	altitude separation
Tot-info	23 / 23	440	1052	249.0	information measure
Sed	10 / 10	14K	363	4151.0	stream editing utility
Space	30 / 38	6K	157	3851.9	ADL interpreter

For our implementation, we used the CIL framework [20], which supports the analysis of ANSI C programs, to analyze the subject programs. We implemented the algorithms to instrument the programs and to compute dynamic control-flow graphs in the *Object Caml Language* (OCaml), because OCaml is required to interface with CIL. Recall that our technique uses dynamic control-flow graphs to compute dynamic dependences, because the former contain only statements that are actually executed by a set of test cases.

We implemented all fault-localization algorithms in *R* [23], which is a statistical computation system with its own language and runtime environment. For matching on program dependences, we used the *R* package *Matching* [28]. We used matching with replacement and a default minimal distance of 0.00001. We also computed fault matrices that indicate, for each faulty program, which test cases pass and which test cases fail. We performed our studies on Mac OS X version 10.5.

4.2 Effectiveness Studies

To study the effectiveness of our technique for fault localization, we use a cost metric that has been used in previous studies [7, 25]. The metric, which we denote as *Cost*, measures the percentage of faulty statements a developer must examine before encountering the faulty statement. We assume that the statements are examined in a non-decreasing order of suspiciousness scores. If there are ties, we assume the developer must examine all the tied statements. For example, if there are n statements in a program and all n statements have the same suspiciousness score, we assume that the developer must examine all n statements. Therefore the *Cost* of finding the fault is 100%.

To compare the effectiveness of fault-localization techniques A and B with respect to a given program version P_i, we use B as a baseline, and compute the difference between the *Cost* of applying B to P_i and the *Cost* of applying A to P_i. A positive result implies that A performed better than B on P_i and a negative result implies that B performed better than A. The difference between the *Costs* corresponds to the magnitude of improvement. For example, if the *Cost* of A is 20% and the *Cost* of B is 60%, developers will examine 40% fewer statements if they use A for fault localization.

To reduce the uncertainty in the causal effect computed for each statement because of the matching technique, we computed a statement's causal effect 100 times and took the average of the 100 causal estimates. We found 100 to be an acceptable threshold for providing stable causal effects. Each computed causal effect has an associated standard error. We computed the average of the standard errors, and used it to construct a bound (one standard deviation) on the causal effect for each statement. For example, if the average of the causal effects of a statement on program failure is 0.3 and the average standard error is 0.1, then $[0.3 \pm 0.1]$ is the range of the causal effect over the 100 samples with 0.2 being the lower bound and 0.4 being the upper bound.

For brevity, we denote our new fault-localization technique, which matches executions based on data and control dependences, as PD; we denote a variant of the technique that matches only on data dependences as DD, and we denote our previous causal technique [5], which considered only control dependences, as CD. We also use PDmin and PDmax to represent variants of PD computed using the lower bounds and upper bounds of the causal-effect estimates of PD, respectively. For example, if for a statement the average causal estimate obtained with PD is 0.3 and the average standard error is 0.1 then PDmin and PDmax yield causal estimates of 0.2 and 0.4, respectively.

Table 2 summarizes the results of comparing fault-localization techniques. The first column (Fault Loc. Tech.) shows the two fault-localization techniques (i.e., A vs B) that are being compared; the second technique in each pair (i.e., B) is the baseline. The second column (Positive (%)) shows the percentage of faulty versions for which A performed better than B, the third column (Negative (%)) shows the percentage of faulty versions for which A performed worse than B, and the fourth column (Neutral (%)) shows the percentage of faulty versions for which there was no improvement. For example, the first row, which compares PD to CD, shows that PD performed better than CD on 61.4% of the faulty versions, performed worse on 13.97% of the faulty versions, and performed identically on 24.63% of the faulty versions.

Table 3 shows the minimum (Min), median (Med), maximum (Max), and mean (Mean) values of A's improvement over B. Half the faulty versions with positive improvement values have improvements between the minimum and the median, and the other half have improvements between the median and the maximum. For example, the first row, which compares PD to CD, shows that half of the 61.4%

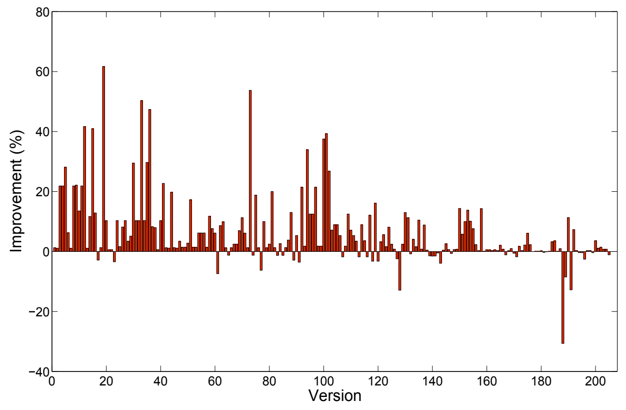


Figure 5: Comparison of new technique (PD) to old technique (CD).

of faulty versions with positive improvement values had improvements between 0.05% and 3.84% while the other half had improvements between 3.84% to 61.71%. The average positive improvement of PD over CD was 8.57%.

4.2.1 Study 1: New Technique vs Old Technique

The goal of this study is to compare the fault-localization effectiveness of our new technique PD to that of our previous causal technique CD, which considers control dependences but not data dependences. To do this, we compared the *Cost* values for the two techniques.

Figure 5 shows the bar graph that summarizes the comparison of PD to CD over all program versions. The horizontal axis (baseline) represents the *Cost* of using our previous technique CD, the vertical axis represents the magnitude of improvement of PD over CD. The graph contains a vertical bar for each faulty version for which there was positive or negative improvement on the horizontal axis; the vertical bar shows the difference in *Costs*. The bars above the horizontal axis represent faulty versions for which PD performed better than CD (positive improvement) and the bars below the horizontal axis represent faulty versions for which PD performed worse than CD (negative improvement). For example, for faulty-version 3 on the graph, PD performed better by about 21.87%. Figure 5 shows that PD performed better than CD overall. As indicated in Table 2, PD performed better than CD on 61.4% of the faulty versions, worse on 13.97% of the faulty versions, and showed no improvement on 24.63% of the faulty versions. The first row of Table 3 characterizes the degree of positive improvement of PD over CD. As the table indicates, half the 61.4% of the faulty versions with positive improvement values had improvements between 0.05% and 3.84%, and the other half had improvements between 3.84% and 61.71%. The average positive improvement of PD over CD was 8.57%.

We also compared the *Costs* of PDmin and PDmax to PD. As Table 2 shows, PDmin performed better than CD on 61.4% of the faulty versions, performed worse on 13.97% of the faulty versions, and performed identically on 24.63% of the faulty versions. The table also shows that PDmax performed better than CD on 60.29% of the faulty versions, performed worse on 14.71% of the faulty versions, and performed identically on 25% of the faulty versions. Table 3 shows that the performance of PD, PDmin, and PDmax are

Table 2: Comparison of fault-localization models.

Fault Loc. Tech.	Positive (%)	Negative (%)	Neutral (%)
PD vs CD	61.40	13.97	24.63
PDmin vs CD	61.40	13.97	24.63
PDmax vs CD	60.29	14.71	25.00
PD vs DD	49.26	23.16	27.57
DD vs CD	43.75	25.74	30.51
PD vs Tarantula	72.06	7.72	20.22
PD vs Ochiai	44.12	21.32	34.56
CO vs Ochiai	53.68	10.29	36.03

Table 3: Distribution of positive improvements.

Fault Loc. Tech.	Min (%)	Med (%)	Max (%)	Mean (%)
PD vs CD	0.05	3.84	61.71	8.57
PDmin vs CD	0.05	4.03	61.71	8.75
PDmax vs CD	0.05	3.94	62.29	8.60
PD vs DD	0.03	3.43	45.71	6.11
DD vs CD	0.05	5.07	59.43	9.54
PD vs Tarantula	0.05	6.41	70.29	11.44
PD vs Ochiai	0.03	4.79	54.86	7.77
CO vs Ochiai	0.03	4.28	54.86	7.45

similar. The results of PDmin and PDmax provide evidence of the stability of the causal estimates computed with PD.[5] Overall, the results indicate that PD improved the accuracy of fault localization over CD by further reducing confounding bias.

Although PD performed better overall, we wanted to see how much of the improvement resulted from data dependences. To do this, we compared DD to CD with respect to *Cost*. Table 2 indicates that DD had lower *Cost* than CD on 43.75% of the faulty versions and higher cost on only 25.74% of the versions. Thus most of the improvement seen with PD is due to considering data dependences. Considering both control and data dependences increases the accuracy of our technique by about 18%. Tables 2 and 3 indicate that causal models based on both data and control dependences are more effective than causal models based on either type of dependence alone. Overall, the results show that by blocking back-door paths created by program dependences in a faulty program, confounding bias can be reduced when estimating the failure-causing effect of a statement.

4.2.2 Study 2: New Technique vs Other Techniques

The goal of this study is to compare PD to two well-known statistical fault-localization techniques, Tarantula [13] and Ochiai [1], with respect to *Cost*. We used the *Costs* of Tarantula and Ochiai as baselines and subtracted the *Cost* of PD.

Figure 6 shows that PD performed better than Tarantula on most faulty versions. Table 2 indicates that PD performed better than Tarantula on 72.06% of the faulty versions, performed worse on 7.72% of the faulty versions, and performed identically on 20.22% of the faulty versions. Table 3 also shows that half the 72.06% with positive improvement values had improvements between 0.05% and 6.41% and that the other half had improvements between 6.41% and 70.29%. The average positive improvement of PD over Tarantula was 11.55%.

In our previous work [5], Ochiai performed better than our older technique (CD). However, Ochiai does not address confounding bias, so we incorporated CD into Ochiai (calling it Causal-Ochiai) and showed that Causal-Ochiai performed better than Ochiai. Here we show that PD performs better than Ochiai even when the techniques are not composed.

Figure 7 shows that PD performed better than Ochiai

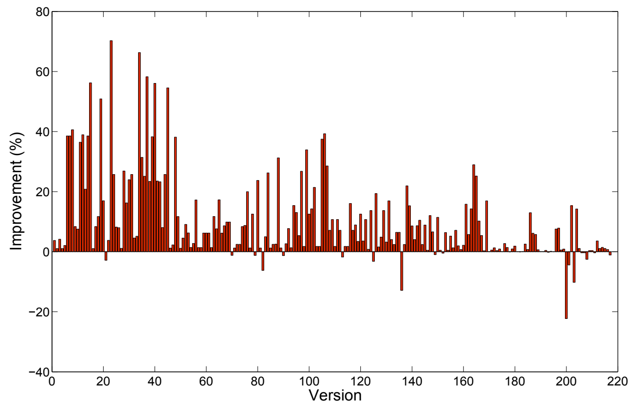


Figure 6: Comparison of our technique (PD) to Tarantula.

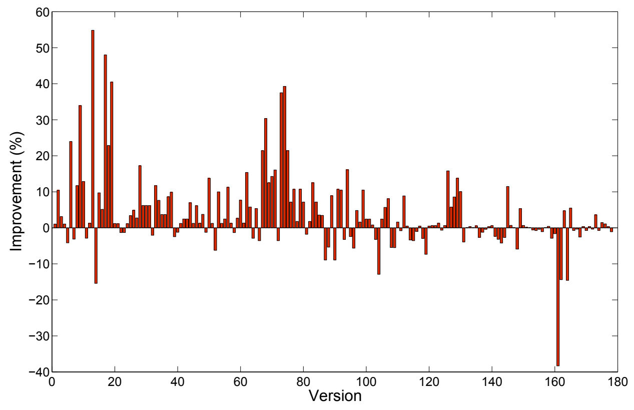


Figure 7: Comparison of our technique (PD) to Ochiai.

on most of the faulty versions. Table 2 indicates that PD performed better than Ochiai on 44.12% of the faulty versions, performed worse on 21.32% of the faulty versions, and performed identically on 34.56% of the faulty versions. Table 3 also shows that half the 44.12% of the faulty versions with positive improvement values had improvements between 0.03% and 4.79% and that the other half had improvements between 4.79% and 54.86%. The average positive improvement of PD over Ochiai was 7.77%. Tables 2 and 3 show that the performance of Ochiai was improved when PD was composed with it to produce Causal-Ochiai (CO). CO performed better on 53.68% of the faulty versions, performed worse on 10.29% of the faulty versions, and performed identically on 36.03% of the faulty versions. Also, half the 53.68% of faulty versions with positive improvement values had improvements between 0.03% and 4.28% and the other half had improvements between 4.28% and 54.86%. The average positive improvement of CO over Ochiai was 7.45%.

Overall, the results indicate that PD performs better than both Tarantula and Ochiai, and improves Ochiai.

4.2.3 Computation Time

In this section, we present the average computation time required to compute all the fault-localization results for one version of some of the subjects. We found that the computation time was largely dependent on the size of the test suite of a program. The Matching package [28] takes considerable time to invert the large covariance matrices. For Tcas, which had 1608 test cases, it took on average 8 minutes. For Schedule, which had 2710 test cases, it took on average 32.73 minutes. For Printtokens2, which had 4115 test cases, it took on average 2.3 hours.

4.3 Threats to Validity

There are three main types of threats to validity that affect our studies: internal, external, and construct. Threats to internal validity concern factors that might affect dependent variables without the researcher's knowledge. The implementations of the algorithms we used in our studies could contain errors. The Matching package we used in our studies is open source and has been used by other researchers for experimentation, which provides confidence that the algorithms in the package are stable. To address potential errors when constructing the dynamic program-dependence graph, we compared manually generated dynamic program-dependence graphs of smaller subjects to graphs generated automatically by our technique.

Threats to external validity occur when the results of a study cannot be generalized. In this work, such threats are greatly alleviated because our work is based on established causal inference theory and methodology. However, more empirical studies on additional subjects are needed to fully address this threat.

Threats to construct validity concern the appropriateness of the metrics used in our evaluation. We used the *Cost* metric to determine the effectiveness of our technique for fault localization. The *Cost* metric is a ranking metric that has been used to compare techniques in many fault-localization studies, though under a different name (*Score*). However, it is not established whether this metric is well suited for presenting fault-localization information to developers.

5. RELATED WORK

There are many techniques that attempt to locate the fault or faults in a failed program. This section discusses those most closely related to ours, and compares our new technique to them.

One category of fault-localization techniques are statistical fault-localization techniques (e.g., [1, 4, 13, 15, 16, 17, 27, 33]). These techniques compute the association between program entities and program failure. However, as we discussed in our previous work and demonstrated with analytical and empirical studies [5], statistical association does not imply causation. In contrast to these existing statistical fault-localization techniques, our technique for finding the cause of program failure is grounded in the theory of causal-inference methodology. Our empirical results in previous work and those presented in this paper demonstrate the effectiveness of our approach over the associative techniques.

Another category of fault-localization techniques are slicing techniques (e.g., [30, 32]). Slicing techniques compute the set of program entities (e.g., statements) that potentially or actually affect the values of variables at a given program point (e.g., the program output). One limitation of slicing techniques is that the techniques do not find the cause of the failure but instead compute the set of program entities associated with the failure. A second limitation is that the slicing techniques do not provide guidance as to how a developer is to examine the statements in the slice. In our work, we have demonstrated with analytical and empirical studies that such association does not imply causation. Our technique, which is grounded in causal theory is more effective at finding causes of failures and as such more effective than slicing techniques. Additionally, our technique provides guidance to developers on how to systematically examine program entities to find the cause of the fault.

A third category of fault-localization techniques are state-altering techniques (e.g., [7, 8, 31]). Like our technique, state-altering techniques are causal techniques. These techniques have a number of limitations that can make them difficult to apply. First, in performing experiments on programs, the techniques must ensure the semantic consistency of memory changes, which can be difficult and expensive. Second, in performing experiments on programs, the techniques require multiple re-executions of the program, which can also be expensive. Third, the techniques require the presence of an oracle (preferably automated) to determine the outcome (success or failure) of the program after each such re-execution, which in practice, can be difficult, if not impossible, to create. The main difference between our technique and state-altering techniques is that our causal technique is observational, whereas they are experimental. Our technique uses observational data (e.g., coverage information) that is often available by executing the program on test cases. Moreover, our technique does not require an automated oracle because the test cases have pre-determined outcomes, and it does not require multiple re-executions of the program. Finally, our technique avoids the semantic consistency problem encountered by state-altering techniques.

One other technique uses causal analysis and program slicing to generate graphs that explain unexpected behaviors produced by computational models [11]. This technique is similar to ours in that both are based on the theory of causal analysis. However, their technique does not find the location of faulty statements. Instead, their technique creates a causal graph that explains an unexpected behavior.

Yet another technique determines whether two versions of a program are the same, given that one of the versions has been transformed (e.g., code has been obsfucated) [19]. To do this, their technique performs dynamic matching on the control-flow graphs of the two versions. Their technique differs from ours in that we match on program dependences to reduce confounding bias for effective fault localization, whereas they match on control flow to determine whether the programs are the same.

Finally, in previous work [5], we presented a causal model that is based on only control dependences. In this work, we have augmented the causal model with data dependences and have shown through empirical studies the effectiveness of the new causal model for fault localization.

6. CONCLUSION AND FUTURE WORK

In this paper we have presented a novel statistically-based causal-inference technique for software fault localization that matches executions based on information about dynamic data and control dependences to obtain more accurate (less biased) estimates of a given statement's effect on the occurrences of program failures. The use of both covariate matching and data-dependence information in our technique are innovations. We also presented empirical results indicating that the new technique is more effective overall than other techniques—including our previous causal-inference technique, which does not consider data dependences.

Although the presented technique performed well overall, it performed relatively poorly on some faulty program versions, which indicates there is room for improvement. We mention three possible reasons for the technique's performance in some cases, and we suggest how it might be improved.

First, the results obtained with the new technique are subject to limitations of the test suite used for fault localization. Even if the test suite has desirable properties overall, it may be inadequate for accurately estimating the failure-causing effects of certain statements. This will be the case for any statement that is covered by very few test cases or for which it is not possible to extract two reasonably well-matched comparison groups of test cases that respectively cover and do not cover the statement. Suspiciousness scores computed for such statements cannot be trusted because they lack adequate support. To address this issue, we plan to investigate the automatic generation of test cases with the appropriate coverage characteristics.

Second, although matching using Mahalanobis distance was generally effective in our studies, it sometimes failed to yield balanced comparison groups. In the latter cases, the rankings produced by our technique are not consistent because they are based on valid causal estimates mixed with biased estimates. Using Mahalanobis distance for matching is also sometimes inefficient, because it is expensive to compute Mahalanobis distance when there are many program dependences. To address these issues, we plan to explore the use of other matching techniques, such as those based on propensity scores [18].

Finally, some faults cannot be effectively localized without considering the variable values carried by data dependences. For example, considering variables values may be the only way to determine that a check for an unusual condition is missing in a certain program location. Although the proposed technique considers data dependences it does not currently consider variable values. We are currently investigating how this can be done.

Acknowledgements

This research was supported in part by NSF award CCF-0725202 and an IBM Software Quality Innovation Award to Georgia Tech, by NSF awards CCF-0820217 and CCF 0702693 and by an award from ABB Corporation to Case Western Reserve University, and by a FACES (Facilitating Academic Careers in Engineering and Science) fellowship award to George Baah. The anonymous reviewers provided many helpful suggestions that improved the presentation of the paper.

7. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. van Gemund. On the Accuracy of Spectrum-Based Fault Localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques*, pages 89–98, September 2007.

[2] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 246–256. ACM, 1990.

[3] A. V. Aho, M. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1:131–137, 1972.

[4] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed Test Generation for Effective Fault Localization. In *Proceedings of the International Symposium for Software Testing and Analysis*, July 2010.

[5] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal Inference for Statistical Fault Localization. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2010.

[6] J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer-Verlag, 2001.

[7] H. Cleve and A. Zeller. Locating Causes of Program Failures. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 342–351, May 2005.

[8] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical Slicing for Software Fault Localization. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 121–134, 1996.

[9] H. Do, S. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[11] R. Gore and P. F. Reynolds, Jr. Causal program slicing. In *IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, 2009.

[12] P. W. Holland. Statistics and Causal Inference. *Journal of American Statistical Association*, 81:945–970, 1986.

[13] 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.

[14] S. Klamt, R. J. Flassig, and K. Sundmacher. TRANSWESD: inferring cellular networks with transitive reduction. *Bioinformatics*, 26:2160–2168, 2010.

[15] 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*, pages 15–26, June 2005.

[16] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical Model-based Bug Localization. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 286–295, September 2005.

[17] Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive Evaluation of Association Measures for Fault Localization. In *Proceedings of International Conference on Software Maintenance (ICSM)*, 2010.

[18] S. L. Morgan and C. Winship. *Counterfactuals and Causal Inference: Methods and Principles of Social Research*. Cambridge University Press, 2007.

[19] V. Nagarajan, R. Gupta, X. Zhang, M. Madou, and B. D. Sutter. Matching Control Flow of Program Versions. In *International Conference on Software Maintenance*, 2007.

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the International Conference on Compiler Construction*, pages 213–228, April 2002.

[21] J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, San Francisco, CA, USA, 2000.

[22] A. Podgurski and L. A. Clarke. A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[23] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008.

[24] S. Rapps and E. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11, NO. 4:367–375, 1985.

[25] M. Renieris and S. Reiss. Fault Localization With Nearest Neighbor Queries. In *International Conference on Automated Software Engineering*, pages 30–39, November 2003.

[26] D. J. Richardson and M. C. Thompson. An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, 1993.

[27] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault localization using multiple coverage types. In *International Conference on Software Engineering*, 2009.

[28] J. S. Sekhon. Multivariate and Propensity Score Matching Software with Automated Balance Optimization: The Matching Package for R. *Forthcoming, Journal of Statistical Software*, 2008. Forthcoming.

[29] E. A. Stuart. Matching Methods for Causal Inference: A Review and a Look Forward. *Statistical Science*, 25:1–21, 2010.

[30] M. Weiser. Program Slicing. In *Proceedings of the International Conference on Software Engineering*, pages 439–449, March 1981.

[31] A. Zeller. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering*, November 2002.

[32] X. Zhang, N. Gupta, and R. Gupta. Pruning Dynamic Slices With Confidence. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.

[33] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing Propagation of Infected Program States. In *Proceedings of European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 43–52. ACM, 2009.