Statistical Debugging Using Compound Boolean Predicates*

Piramanayagam Arumuga Nainar Ting Chen Jake Rosin Ben Liblit
Computer Sciences Department
University of Wisconsin–Madison
{arumuga,tchen,rosin,liblit}@cs.wisc.edu

ABSTRACT

Statistical debugging uses dynamic instrumentation and machine learning to identify predicates on program state that are strongly predictive of program failure. Prior approaches have only considered simple, atomic predicates such as the directions of branches or the return values of function calls. We enrich the predicate vocabulary by adding complex Boolean formulae derived from these simple predicates. We draw upon three-valued logic, static program structure, and statistical estimation techniques to efficiently sift through large numbers of candidate Boolean predicate formulae. We present qualitative and quantitative evidence that complex predicates are practical, precise, and informative. Furthermore, we demonstrate that our approach is robust in the face of incomplete data provided by the sparse random sampling that typifies post-deployment statistical debugging.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*statistical methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, distributed debugging, monitors, tracing*; I.5.2 [**Pattern Recognition**]: Design Methodology—*feature evaluation and selection*

General Terms

Experimentation, Reliability

Keywords

statistical bug isolation, three-valued logic, debugging effort metrics, dynamic feedback analysis

1. INTRODUCTION

Statistical debugging improves software quality by identifying program (mis)behaviors that are highly predictive of subsequent program failure. As embodied in the Cooperative Bug Isolation Project (*CBI*) [14], these techniques find bugs in programs by analyzing reports collected from software executing in the hands of end users. First the CBI instrumenting compiler injects extra code that evaluates simple Boolean expressions (called *predicates*) at various program points. Predicates are designed to capture potentially interesting program behaviors such as results of function calls, directions of branches, or values of variables. Upon termination of an instrumented program, a feedback report is generated that records how often each predicate was observed, and how often each was both observed and found to be true. Given many such feedback reports, e.g., from a large user community, statistical debugging is used to find predicates that are predictive of failure [13, 19]. These predicates are then ranked and presented to the developer.

CBI gathers execution reports by using valuable CPU cycles at end user machines. It is essential to make those cycles worthwhile by extracting every bit of useful information from them. Instead of considering predicates in isolation from one another as in previous work [13, 16, 19], this paper explores useful relations between predicates. Predicates are expressions involving program variables at different program points and hence may be related by control and data dependences. We propose to capture these relations by building *complex predicates* from the set of currently-instrumented predicates (which we refer to as *simple predicates*). Since predicates are Boolean expressions, they are combined using logical operators (such as conjunction and disjunction). We construct complex predicates and include them in the input to the statistical analysis algorithms.

There are two approaches to combine predicates using logical operators:

1. Change the instrumenting compiler to explicitly monitor each complex predicate at run time.

2. Estimate the value of each complex predicate from the values of its components.

The first approach will yield a precise value but needs significant modifications to existing infrastructure. The second approach will be less precise (as described later) but requires only few modifications to existing infrastructure (and none to the instrumenting compiler). In the present work, we implement the second approach, which serves as a proof of concept for using complex predicates, as well as a justification for incorporating them into the instrumenting compiler in the future.

The remainder of this paper is organized as follows. Section 2 reviews statistical debugging and motivates the present effort to find complex failure predictors. Section 3 gives a precise definition of complex predicates and discusses how complex predictors can be computed efficiently. Section 4 defines two metrics to evaluate the

*This research was supported in part by AFOSR Grant FA9550-07-1-0210 and NSF Grant CCF-0621487.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'07, July 9–12, 2007, London, England, United Kingdom.
Copyright 2007 ACM 978-1-59593-734-6/07/0007 ...$5.00.

usefulness of a complex predicate. Section 5 discusses two case studies that demonstrate the usefulness of complex predicates.

Section 6 presents the results of experiments conducted on a large suite of buggy test programs, including an assessment of the effect of sparse random sampling on complex predicates. Sparse random sampling is a technique used by CBI to reduce the runtime overhead of instrumentation. Section 7 discusses related work and Section 8 concludes and offers directions for future work.

2. BACKGROUND

CBI uses lightweight instrumentation to collect feedback reports that contain truth values of predicates in an execution as well as the outcome (e.g., crash or non-crash) of the execution. Large numbers of reports are collected, then analyzed using statistical debugging techniques. These techniques identify *bug predictors*: predicates that, when true, herald failure due to a specific bug. Bug predictors highlight areas of the code that are related to program failure and so provide information that is useful when correcting program faults. Feedback reports can be collected from deployed software in the hands of end users, who may encounter bugs not identified in program testing. CBI can therefore be used to monitor software after its release and help direct program patches by identifying bugs as they manifest in the field.

2.1 Finding Bug Predictors

The feedback report for a particular program execution is formed as a bit-vector, with two bits for each predicate (*observed* and *true*), and one final bit representing success or failure. If generated in experimental or testing conditions these feedback reports are likely to be complete; when instrumented code is distributed to end users predicates are usually sampled infrequently to reduce computational overhead. Previous experiments [12] have determined that sampling rates of 1/100 to 1/1,000 are most realistic for deployed use.

Using these reports, CBI assigns a score to all available predicates and identifies the single best predictor among them (see Section 2.2). It is assumed that this predictor corresponds to one important bug, though other bugs may remain. This top predictor is recorded, and then all feedback reports where it was true are removed from consideration under the assumption that fixing the corresponding bug will change the behavior of runs in which the predictor originally appeared. The next best predictor among the remaining reports is then identified, recorded, and removed in the same manner. This iterative process terminates either when no undiagnosed failed runs remain, or when no more failure-predictive predicates can be found.

This process of iterative elimination maps each predictor to a set of program runs. Ideally each such set corresponds to the expression of a distinct bug; unfortunately this is not always the case. Due to the statistical nature of the analysis, along with incomplete feedback reports resulting from sparse sampling rates, a single bug could be predicted by several top-ranked predicates, and predictors for less prevalent bugs may not be found at all.

The output of the analysis is a list of predicates that had the highest score during each iteration of the elimination algorithm, as well as a complete list of predicates and their scores before any elimination is performed. These lists may be used by a programmer to identify areas of the program related to faulty behavior. Liblit et al. employed this method to discover previously unknown bugs in several widely-used applications [12, 13].

CBI output can also be used as input to an automated analysis tool, such as BTRACE [11]. BTRACE finds the shortest control- and dataflow-feasible path in the program that visits a given set of bug predictors. This analysis allows a programmer to examine the fault-predicting behavior even if the connection to a bug is not easily identifiable, or if the predictors are numerous or complex enough to overwhelm a programmer examining them directly.

2.2 Scoring Predicates

This section provides a brief overview of the numeric scores used to identify the best predictor from a set of predicates. For a detailed discussion on this topic, the reader should refer to Liblit et al. [13]. A good predictor should be both *sensitive* (accounts for many failed runs) and *specific* (does not mis-predict failure in successful runs). Assigning scores based on sensitivity will result in *super-bug predictors*, which include failures from more than one bug. Super-bug predictors are highly non-deterministic, since they are not specific to any single cause of failure, and rarely provide useful debugging information. Scoring predicates based on specificity instead results in *sub-bug predictors*. A sub-bug predictor accounts for a portion of the failures caused by a bug, but not all. Unlike super-bug predictors, sub-bug predictors that account for a significant sub-set of failures can be useful in debugging, although perfect predictors are of course preferred. Sensitivity and specificity are balanced using a numeric *Importance* score computed as follows.

The truth values of a predicate p from all the runs can be aggregated into four values:

1. $S(p \text{ obs})$ and $F(p \text{ obs})$, respectively the number of successful and failed runs in which the value of p was evaluated.
2. $S(p)$ and $F(p)$, respectively the number of successful and failed runs in which the value of p was evaluated and was found to be true.

Using these values, two scores of bug relevance are calculated:

Sensitivity: $\log(F(p))/\log(NumF)$ where $NumF$ is the total number of failing runs. A good predictor must predict a large number of failing runs.

Specificity: $Increase(p)$. The amount by which p being true increases the probability of failure over simply reaching the line where p is defined. It is computed as follows:

$$Increase(p) \equiv \frac{F(p)}{S(p)+F(p)} - \frac{F(p \text{ obs})}{S(p \text{ obs})+F(p \text{ obs})} \quad (1)$$

Taking the harmonic mean combines these two scores, identifying predicates that are both highly sensitive and highly specific:

$$Importance(p) \equiv \frac{2}{\frac{1}{Increase(p)} + \frac{1}{log(F(p))/log(NumF)}} \quad (2)$$

The *Importance* score is calculated for each predicate, and the top result selected. After all runs in which the top predicate is true are eliminated scores are recalculated for all remaining predicates in the remaining sets of runs. This process of eliminating runs continues, as described above, until there are no remaining failed runs or no remaining predicates.

2.3 Expected Benefits of Complex Predicates

A single predicate can be thought of as partitioning the space of all runs into two subspaces: those satisfying the predicate and those not. The more closely these partitions match the subspaces where the bug is and is not expressed, the better the predicate is as a bug predictor. If a bug has a cause which corresponds well to a simple predicate then a simple analysis is sufficient, but analysis of more complex bugs will produce only super- and sub-bug predictors.

A richer family of predicates can describe more complex shapes within the space of runs. This allows good predictors for bugs with more complicated causes. Some bugs may have causes connected to simple predicates, but that no single predicate can accurately predict. Complex predicates formed from these simpler ones would be more accurate predictors than any component predicate. *Partial predictors* are predicates that predict some aspect of a bug that is necessary, but not sufficient, for program failure. Partial predictors and sub-bug predictors are two classes of simple predicates which can be combined into more accurate predictors.

A partial predictor will correctly partition all (or most) expressions of the bug, but would also predict the bug in a large number of runs where it did not occur. Because partial predictors are highly non-deterministic with respect to the bug, they are likely to be outscored by a sub- or super-bug predictor. Partial predictors can be improved by eliminating false positives. This can be accomplished by taking a conjunction with a predicate that captures another aspect of the bug. The case study presented in Section 5.1 describes a bug that is predicted best by a conjunction involving a partial predictor.

Sub-bug predictors correctly partition some expressions of a bug, but not all. They are useful in identifying a bug because, though they do not predict the bug in a general sense, they are extremely good predictors of some special case where the bug is expressed. Combining two such predictors with a disjunction will reduce false negatives and result in a predicate that correctly partitions more manifestations of the bug. Combine enough special cases in this manner and the resulting predicate will predict the bug in the general case. It is important to note that the analysis may find a disjunction of predictors of individual bugs as a predictor for the whole set of failures. This it is not as problematic as it seems: for such a disjunction to be high-ranked each component predicate must be a good predictor for a specific bug, providing useful information on all bugs involved.

The bug predictors that result from combining simple predicates can be conjoined or disjoined again, eliminating more false positives and false negatives to approach a perfect predictor. This process can continue, eventually finding a good predictor for any bug that can be expressed in terms of the simple predicates measured during the construction of the feedback reports. Even if some aspect of the bug is uncovered by the simple predicates, it's likely that a sub-bug predictor may still be constructed. The introduction of complex predicates to CBI analysis greatly increases the number of shapes that can be described within the set of runs, thereby increasing the chances of finding an accurate predictor for a bug.

3. COMPLEX PREDICATES

A complex predicate C is defined as $C = \phi(p_1, p_2, \ldots p_k)$ where $p_1, p_2, \ldots p_k$ are simple predicates and ϕ is a function in conjunctive normal form (CNF). Evaluating C requires combining predicates using $\wedge$ ("and") and $\vee$ ("or"). A negation operator is not required because, by design, the negation of every CBI predicate p is also a predicate. For N predicates there are 2^{2^N} such Boolean functions [18]. There may be hundreds of simple predicates involved in the analysis, and so this a prohibitively large number.

To reduce complexity we consider only functions of two predicates. Out of the 16 (2^{2^2}) such functions, we consider only conjunction and disjunction since they are likely to be most easily understood by programmers. Our revised definition is $C = \phi(p_1, p_2)$ where $\phi \in \{\vee, \wedge\}$. Conjunction and disjunction are commutative, and the reflexive cases ($p_1 \wedge p_1$ and $p_1 \vee p_1$) are uninteresting. This reduces the number of complex predicates to just $\binom{N}{2} = \frac{N(N-1)}{2}$ binary conjunctions and an equal number of binary disjunctions. These may be evaluated for a set of R runs in $O(|R|N^2)$ time. The revised definition for a complex predicate is used throughout; related definitions may be trivially extended to the general case.

3.1 Measuring Complex Predicates

For a predicate p and a run r, $r(p)$ is true if and only if p was observed to be true at least once during run r. Similarly we could define $r(C)$ as follows:

Definition 1. For a complex predicate $C = \phi(p_1, p_2)$, $r(C)$ is true iff at some point during the execution of the program, C was observed to be true.

The difficulty with this notion of complex predicates is that C must be explicitly monitored during the program execution. For example, $r(p_1) =$ true and $r(p_2) =$ true does not imply that $p_1 \wedge p_2$ is ever true at a single program point. p_1 and p_2 may be true at different stages of execution but never true at the same time. Furthermore, when p_1 and p_2 appear at different source locations, there may be no single point in time at which both are even well-defined and therefore simultaneously observable. In order to be able to estimate the value of C from its components, we adapt a less time-sensitive definition as follows:

Definition 2. For a complex predicate $C = \phi(p_1, p_2)$, $r(C)$ is true iff $\phi(r(p_1), r(p_2))$ is true.

In other words, we treat r as distributive over ϕ, effectively removing the requirement that all p_i be observed simultaneously. This can lead to false positives, because $r(C)$ may be computed as true when C is actually false at all moments in time. False negatives, however, cannot arise. The impact of this assumption on the score of C may be either positive or negative depending on whether r failed or succeeded.

3.2 Three-Valued Logic

This section explains how conjunctions and disjunctions are actually computed. Three-valued logic is required because the value of a predicate in a run may not be certain. This can arise in two situations:

1. The predicate was not observed in a run because the run did not reach the line where it was defined.
2. The program reached the line where the predicate was defined but was not observed because of sampling.

In such cases, the value of a predicate p is considered unknown. For the analysis introduced in Section 2, it is enough to consider whether $r(p)$ was true or $\neg$true (either false or unknown). When constructing compound predicates, however, considering the subcases of $\neg$true separately allows additional run information to be derived. Constructing a compound predicate requires generating two bits for each program run: whether C was observed at least once during run r and whether C was observed true at least once during run r. The case where a compound predicate is observed but not true can only be generated by considering whether the same was true for its components.

Consider a complex predicate $C = p_1 \wedge p_2$. If either $r(p_1)$ or $r(p_2)$ was false in a run r, then $r(C) =$ false, since one false value disproves a conjunction. If both $r(p_1)$ and $r(p_2)$ were true, then C was observed to be true. Otherwise, the value of $r(C)$ is unknown. This is shown using a three-valued truth table in Table 1a.

Similarly one true value proves a disjunction, and so a disjunction is false only if both of its components were observed false. The truth table for the predicate $D = p_1 \vee p_2$ is shown in Table 1b.

Table 1: Three-valued truth tables for complex predicates

(a) Conjunction: $p_1 \wedge p_2$

$p_1 \searrow p_2$	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

(b) Disjunction: $p_1 \vee p_2$

$p_1 \searrow p_2$	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

3.3 Interesting Complex Predicates

Even using the revised definition for C, the number of complex predicates is quadratic in the number of simple predicates. A large number of complex predicates formed by this procedure are likely to be useless in the analysis of the program. Certainly a complex predicate that has a lower score than one of its components is useless: the component (simple) predicate with a higher score is a better predictor of failure, and so the complex predicate adds nothing to the analysis.

Definition 3. A complex predicate $C = \phi(p_1, p_2)$ is "interesting" iff $Importance(C) > \mathbf{max}(Importance(p_1), Importance(p_2))$

In the case where the complex predicate has the same score as a component predictor, the simpler one is preferable. Keeping only interesting combinations of predicates reduces the memory burden of storing them, and helps ensure the utility of a complex predicate that is presented to the user.

3.4 Pruning

Constructing complex predicates from simple ones requires generating two bits of information (according to the three-valued logic discussed in Section 3.2) for each program run. As CBI is meant to analyze deployed software, program runs potentially number in the hundreds of thousands, if not millions. The measure of a predicate's relevance to a bug is its *Importance* score (see Section 2.2), which for complex predicates can be computed only after the three-valued bits are generated for all the runs. Even for small test datasets with around a thousand runs, this computation takes around ten minutes. However, it turns out that most of these complex predicates are not interesting (i.e., they have low *Importance* scores) and are not presented to the programmer. In such cases, the effort to compute scores has been wasted. This suggests that one should prune combinations before run information is generated based on an estimate of their resulting scores. If this estimate falls below the threshold required for the predicate to be presented to the programmer, the exact score is not computed. This reduces the complexity of constructing run information for the common case from $O(|R|)$ to $O(1)$.

The threshold *Importance* value, which complex predicates must potentially exceed to be constructed, can be calculated in two ways:

1. Only interesting complex predicates are retained; for a complex predicate $\phi(p_1, p_2)$ the threshold for $Importance(C)$ is therefore $\mathbf{max}(Importance(p_1), Importance(p_2))$.

2. During redundancy elimination, only the predicate with the highest score is retained at each iteration. The threshold for $Importance(C)$ is therefore the highest score yet seen (including those of simple predicates).

To simplify the formulae derived in this section, we introduce some new terms and notations. If $R \in \{F, S\}$ is the set of program runs under consideration, then $R(p \text{ obs})$ denotes the number of runs in which p was observed at least once. $R(p)$ is the number of runs in which p was observed true at least once. $R(\bar{p})$ is the number of runs in which predicate p was observed at least once but never observed true. It is equal to $R(p \text{ obs}) - R(p)$. For some unknown quantity x, let $\uparrow x$ and $\downarrow x$ denote estimated upper and lower bounds on the exact value of x, respectively.

Using an upper bound as the estimate of *Importance* guarantees that no predicate is pruned erroneously. This upper bound can be computed by maximizing $Increase(p)$ and $log(F(p))/log(NumF)$ under constraints based on the Boolean operation. $Importance(p)$ (Equation 2), being a harmonic mean of these two terms, will likewise be maximized. From Equation 1, an increase in $F(p)$ or $S(p \text{ obs})$ or a decrease in $S(p)$ or $F(p \text{ obs})$ will increase the value of $Increase(p)$. So an upper bound on $Increase(p)$ is

$$\uparrow Increase(p) \equiv \frac{\uparrow F(p)}{\downarrow S(p) + \uparrow F(p)} - \frac{\downarrow F(p \text{ obs})}{\uparrow S(p \text{ obs}) + \downarrow F(p \text{ obs})} \quad (3)$$

The second component of *Importance* is $log(F(p))/log(NumF)$. *NumF* is a constant, so maximizing $F(p)$ yields an upper bound of this quotient. Note that $\uparrow F(p)$ is also required for the upper bound of the first term, $Increase(p)$.

Consider a conjunction $C = p_1 \wedge p_2$ and a set of runs R. From Table 1a, C is observed true if both p_1 and p_2 were observed true. Equivalently, the set of runs in which C was observed true is the intersection, and thus cannot exceed the sizes, of the sets of runs in which p_1 was observed true and p_2 was observed true.

$$\uparrow R(C) = \mathbf{min}(R(p_1), R(p_2))$$

Likewise, $R(C)$ is minimized when there is minimum overlap between the set of runs in which p_1 was observed true and p_2 was observed true. The minimum overlap is 0 as long as $R(p_1) + R(p_2)$ does not exceed $|R|$, the total number of runs. Otherwise, both p_1 and p_2 must be observed together in at least $R(p_1) + R(p_2) - |R|$ runs. Thus,

$$\downarrow R(C) = \mathbf{max}(0, R(p_1) + R(p_2) - |R|)$$

Next consider $R(C \text{ obs})$. C is observed when (1) both p_1 and p_2 are observed true, or (2) either p_1 or p_2 is observed false. To maximize $R(C \text{ obs})$, instances of both of these cases should be maximized. There are $\mathbf{min}(R(p_1), R(p_2))$ applications of Rule 1 when the set of runs in which p_1 and p_2 are observed true completely overlap. There are $R(\bar{p_1}) + R(\bar{p_2})$ applications of Rule 2 when the sets of runs in which p_1 and p_2 are observed and never true are non-overlapping. However, there has to be an overlap between these cases if their sum exceeds $|R|$, the total number of runs. Thus,

$$\uparrow R(C \text{ obs}) = \mathbf{min}(|R|, R(\bar{p_1}) + R(\bar{p_2}) + \mathbf{min}(R(p_1), R(p_2)))$$

Finally, to minimize $R(C \text{ obs})$, applications of the two rules mentioned above are minimized. This can happen when

1. the false observations of p_1 and p_2 completely overlap, contributing $\mathbf{max}(R(\bar{p_1}), R(\bar{p_2}))$ to $R(C \text{ obs})$, and

2. the true observations of p_1 and p_2 do not overlap. However, the outcomes of $\mathbf{min}(R(\bar{p_1}), R(\bar{p_2}))$ runs have been fixed in the previous rule, effectively reducing the number of runs to $|R'| = |R| - \mathbf{min}(R(\bar{p_1}), R(\bar{p_2}))$. Among these runs, $R(p_1) + R(p_2)$ runs must be selected without overlap, failing which there would be an $R(p_1) + R(p_2) - |R'|$ overlap between the two sets.

Thus,

$$\downarrow R(C \text{ obs}) = \mathbf{max}(R(\bar{p_1}), R(\bar{p_2})) + \mathbf{max}(0, R(p_1) + R(p_2) - |R'|)$$

Table 2: Bounds required in Equation 3 for a conjunction

Quantity	Bounds for $C = p_1 \wedge p_2$
$\uparrow F(C)$	$\mathbf{min}(F(p_1), F(p_2))$
$\downarrow S(C)$	$\mathbf{max}(0, S(p_1) + S(p_2) - \lvert S\rvert)$
$\uparrow S(C \text{ obs})$	$\mathbf{min}(\lvert S\rvert, S(\bar{p_1}) + S(\bar{p_2}) + \mathbf{min}(S(p_1), S(p_2)))$
$\downarrow F(C \text{ obs})$	$\mathbf{max}(F(\bar{p_1}), F(\bar{p_2}))+$ $\mathbf{max}(0, F(p_1) + F(p_2) - \lvert F'\rvert)$ where $\lvert F'\rvert = \lvert F\rvert - \mathbf{min}(F(\bar{p_1}), F(\bar{p_2}))$

Table 3: Bounds required in Equation 3 for a disjunction

Quantity	Bounds for $D = p_1 \vee p_2$
$\uparrow F(D)$	$\mathbf{min}(\lvert F\rvert, F(p_1) + F(p_2))$
$\downarrow S(D)$	$\mathbf{max}(S(p_1), S(p_2))$
$\uparrow S(D \text{ obs})$	$\mathbf{min}(\lvert S\rvert, S(p_1) + S(p_2) + \mathbf{min}(S(\bar{p_1}), S(\bar{p_2})))$
$\downarrow F(D \text{ obs})$	$\mathbf{max}(F(p_1), F(p_2))+$ $\mathbf{max}(0, F(\bar{p_1}) + F(\bar{p_2}) - \lvert F'\rvert)$ where $\lvert F'\rvert = \lvert F\rvert - \mathbf{min}(F(p_1), F(p_2))$

For simplicity and generality, the preceding discussion derives the bounds for a general set of runs R. Table 2 lists the specific bounds required in Equation 3 for conjunctions by substituting the set of successful runs S or the set of failed runs F in place of R. The approach for disjunctions is similar. In interest of space, we omit the derivation and just list the bounds for a disjunction $D = p_1 \vee p_2$ in Table 3.

As a concrete example, consider $C = p_1 \wedge p_2$. Assume that there are 1,000 successful runs, 1,000 failed runs, and that p_1 and p_2 are observed in all runs. Furthermore, assume $F(p_1) = 500$, $F(p_2) = 1000$, $S(p_1) = 250$ and $S(p_2) = 500$. Substituting $R(\bar{p}) = 1000 - R(p)$ and computing the bounds listed in Table 2 we get $\uparrow F(C) = 500, \downarrow S(C) = 0, \downarrow F(C \text{ obs}) = 1000$ and $\uparrow S(C \text{ obs}) = 1000$. Therefore the upper bound of $Increase(C)$ is $\frac{500}{500} - \frac{1000}{2000} = 0.5$ and the upper bound of $Importance(C)$ is

$$\frac{2}{\frac{1}{0.5} + \frac{1}{\log 1000 / \log 1000}} = 0.666\ldots.$$

If the complex predicate C is generated during the iterative elimination step (see Section 2.1) and we already know a predicate C' which scored better than 0.666, we can prune C before computing its exact score. The estimate of $Importance(C)$ computed above is an upper-bound on the actual score of C, making it mathematically impossible for it to beat C' even if fully evaluated.

All upper- and lower-bound estimates are conservative. Pruning complex predicates using the above calculations and the appropriate threshold value reduces the computational intensity of the analysis without affecting the results. The effectiveness of pruning in practice is examined in Section 6.2.

4. USABILITY METRICS

In our experiments, we often observe hundreds of complex predicates with similar or even identical high scores. The redundancy elimination algorithm will select the top predicate randomly from all those tied for the top score; a human programmer finding a predicate to use in debugging is likely to make a similar choice. *Importance* measures predictive power, so all high-scoring predicates should be good bug predictors. However, even a perfect predictor may be difficult for a programmer to use when finding and fixing a bug.

Debugging using a simple predicate is aided by understanding the connection between the predicate and the bug it predicts. For a complex predicate, the programmer must also understand the connection between its components. Given a set of complex predicates with similar high scores, those that can be easily understood by a human are preferable. However, the notion of usability is hard to quantify or measure. In this section we propose two metrics for selecting understandable predicates from a large set of high-scoring predictors. Only predicates selected by the metrics are presented to the user. (If the data is analyzed by an automated tool it may not be advantageous to employ these metrics.) Both metrics use criteria unrelated to a predicate's *Importance* score, making them orthogonal to pruning as discussed in Section 3.4. As mentioned earlier, the objective function approximated by these metrics is hard to measure experimentally. However, we find that these metrics work well in practice.

4.1 Effort

The first metric models the debugging effort required from the programmer to find a direct connection between component predicates. We adapt the distance metric of Cleve and Zeller [1] for this purpose. In this metric, the score of a predicate is the fraction of code that can be ignored while searching for the bug. We use a similar metric called *effort* for a complex predicate.

Definition 4. The effort required by a programmer while using a complex predicate $\phi(p_1, p_2)$ is inversely proportional to the smaller fraction of code ignored in a breadth-first bidirectional search for p_1 from p_2 and vice-versa.

The idea behind this metric is that the larger the distance between the two predicates, the greater the effort required to understand their relationship. Also, if a large number of other branches are seen during the search, the programmer should keep track of these dependencies too. Per Cleve and Zeller [1], we use the program dependence graph (PDG) to model the program rather than the source code. We perform a breadth-first search starting from p_1 until p_2 is reached and count the total number of vertices visited during the search. The fraction of code covered is the ratio of the number of visited vertices to the total number of PDG vertices.

4.2 Correlation

The second metric considers the correlation between the two predicates. Two predicates may be easily reached from one another without having an apparent connection; a complex predicate formed from them would provide little help to the programmer. On the other hand, two predicates which are both affected by some shared area of code may have a connection which a programmer can easily discern. The correlation between two predicates is defined based on the program dependency graph. Given a single predicate p, we define the *predecessor set* of p as the set of vertices in the PDG that can influence p.

Definition 5. The correlation between two predicates of a complex predicate is defined as the number of vertices in the intersection of the two predecessor sets.

The idea behind this metric is that a larger intersection between the predecessor sets means it is possible that they are closely related. We expect correlation to mitigate the issue of disjunctive predicates raised in Section 2, namely that the disjunction of predictors for two separate bugs will be scored very highly. The predictors of two unrelated program faults are likely to reside in different areas of the program, and therefore the intersection of their predecessor sets would be smaller than two related predictors for the same bug, which are likely to be in closer proximity.

4.3 Proactive and Reactive Pruning

The above two metrics can be applied both *proactively* and *reactively*. Proactive use of the metrics removes complex predicates whose metric values fall below a certain threshold of usefulness. This avoids computing their scores and hence improves performance. Reactive use of the metrics retains all the predicates but breaks ties by giving higher ranks to those with better values for the metrics. This is desirable if neither computing time nor space is a concern.

5. CASE STUDIES

This section discusses two cases where complex predicates prove to be useful. The first study concerns a memory access bug in `exif` 0.6.9, an open source image manipulation program. A complex predicate is useful in increasing the score of an extremely useful bug predictor. The second study uses an input validation bug in `ccrypt` 1.2 to explain how complex predicates can be used to identify partial predictors automatically.

Both studies present predicates found during automated analysis. It should be noted that in most cases the predicates discussed were not top-ranked, and in fact many were removed by the redundancy elimination algorithm. These predicates were manually identified from the full list of predictors. This demonstrates the need for techniques to effectively filter through large numbers of complex predicates, as discussed in Section 4.

5.1 `exif`

`exif` 0.6.9 crashes while manipulating a thumbnail in a Canon image. The bug is in function `exif_mnote_data_canon_load` in the module handling Canon images. The following is a snippet from said function:

```
for (i = 0; i < c; i++) {
    ...
    n->count = i + 1;
    ...
    if (o + s > buf_size) return;     // (a)
    ...
    n->entries[i].data = malloc(s);  // (b)
    ...
}
```

If the condition `o + s > buf_size` is true on line (a), then the allocation of memory to the pointer `n->entries[i].data` on line (b) is skipped. The program crashes when other code reads from `n->entries[i].data` without checking if the pointer is valid. This is an example of a non-deterministic bug as the program succeeds as long as the uninitialized pointer is not accessed somewhere else.

We generated 1,000 runs of the program using input images randomly selected from a set of Canon and non-Canon images. As the bug being studied rarely manifests, this set of images was designed to trigger sufficient failed executions. Each run was executed with randomly-generated command line arguments, omitting arguments which would have triggered two unrelated bugs. There are 934 successful executions and 66 crashes. Applying the redundancy elimination algorithm with only simple predicates produces two predicates that account for all failed runs as shown in Table 4. Studying the source code of the program does not show any obvious relation between the two predictors and the cause of failure. Although the second predictor is present in the crashing function, it is a comparison between two unrelated variables: the loop iterator `i` and the size of the data stored in the traversed array `s`. Also it is true in only 31 of the 66 failures.

The analysis assigns a very low score of 0.0191528 to the predicate p_1: `o + s > buf_size` despite the fact that it captures the exact source of the uninitialized pointer. Because the bug is non-deterministic, p_1 is also true in 335 runs that succeeded, making p_1 a partial predictor. Including complex predicates in the analysis produces one complex predicate shown in Table 5. (The second row is the second component of a complex predicate, which is a conjunction as indicated by the keyword *and* at the start.) Conjunction of p_1 with the second predicate p_2: `offset < len` eliminates all false positives and thereby earns a very high score. This is an example of how a conjunction can improve the score of a partial predictor. p_2 is in function `exif_data_load_data` that calls `exif_mnote_data_canon_load` indirectly. It is possible that p_2 is another partial predictor, capturing another condition that drives the bug to cause a crash. If it does, it has to be a deep relationship as we could not find such a relation even after spending a couple of hours trying to understand the source code. However this does not reduce the importance of this result as the conjunction has a very high score compared to p_1 and p_2 individually.

At the point where the uninitialized pointer is actually used, a hypothetical predicate p_3: `n->entries[i].data == 0` ought to be a perfect bug predictor. However, the CBI instrumenting compiler does not actually instrument this condition or any direct equivalent. Furthermore, this assumes that `n->entries[i].data` is zero-initialized even when `exif_mnote_data_canon_load` returns early without filling in this field. Predicate p_1 provides critical additional information, as it identifies the initial trigger (skipping the `malloc`) that sets the stage for eventual failure (use of an uninitialized pointer). Thus one role for complex predicates is to capture those program behaviors, like p_1, that are necessary but not sufficient preconditions for failure.

5.2 `ccrypt`

`ccrypt` 1.2 contains a known bug that causes a crash when `EOF` is received at a confirmation prompt before overwriting an existing file. `EOF` in other contexts does not cause failure, however, and an examination of the source code quickly reveals why:

```
/* read a yes/no response from the user */
int prompt(void) {
  ...
  line = xreadline(fin, cmd.name);    // (a)
  return (!strcmp(line, "y") ||
     !strcmp(line, "yes"));
}

char *xreadline(FILE *fin, char *myname) {
  ...
  res = fgets(buf, INITSIZE, fin);
  if (res==NULL) {                    // (b)
    free(buf);
    return NULL;
  }
  ...
  return buf;
}
```

Calls to `xreadline`, the function used to get user input, can return `NULL` under some circumstances. In most cases the value is checked before being dereferenced. In `prompt`, however, it is used immediately after the call on line (a). `xreadline` returning `NULL` in `prompt` should thus be a perfect predictor of failure, occurring in no successful runs and in every failure related to this bug. The branch taken on line (b) in `xreadline` is important as well, serving as the moment failure in `prompt` becomes inevitable. This branch is only taken when the user enters `EOF` on

Table 4: Results for `exif` with only simple predicates

Score	Predicate	Function	File:Line
0.704974	new value of len == old value of len	`jpeg_data_load_data`	jpeg-data.c:224
0.395001	i == s	`exif_mnote_data_canon_save`	exif-mnote-data-canon.c:176

Table 5: Results for `exif` with complex predicates

Score	Predicate	Function	File:Line
0.941385	o + s > buf_size is TRUE *and* offset < len	`exif_mnote_data_canon_load` `exif_data_load_data`	exif-mnote-data-canon.c:237 exif-data.c:644

the command line. In mapping the cause of failure, a programmer without a clear understanding of the code is likely to spend time tracking the user-entered `EOF` through `xreadline` to the `NULL` dereference in `prompt`, requiring either a visual inspection of the source or use of an interactive debugger. Knowledge of the connection between program events such as these is necessary to make good debugging decisions, e.g., adding a `NULL` check to `prompt` versus ensuring `xreadline` always returns a valid pointer. Automated bug analysis should ideally reveal as much of this chain of causation to the programmer as possible.

We generated 1,000 runs of `ccrypt`, again using randomly-selected command line arguments. Input files include images and text archived from the online documentation of a remote desktop display system. There are 658 successful executions and 342 crashes. All failing runs crash due to the `NULL` dereference described above. No other bugs were visible to our test suite.

An initial analysis of only simple predicates (Table 6) finds p_1: `xreadline == 0` as the top predictor of failure. It is true in all 342 failed runs and no successful runs, verifying our assumptions. The related predicate p_2: `res == (char *)0` scores substantially lower, appearing in all failures but a large number of successes. p_2's reported score is low enough that without knowledge of the nature of the bug a programmer would be likely to overlook its significance. Additionally, because of its relationship to p_1, it is removed by the redundancy elimination algorithm. More importantly, traditional CBI analysis reveals no connection between the two predictors to the programmer, despite the fact that p_2, a necessary but not sufficient condition for failure, is subordinate to p_1 in predicting a crash.

When complex predicates are included in the analysis (Table 7), a conjunction of p_1 and p_2 is among the top predictors. This provides little help in finding the bug, which is easily identified by traditional CBI analysis, but it does reveal the nature of p_2 as a partial predictor. The conjunction $p_1 \wedge p_2$ is observed in more successful runs than p_1 alone, but is true in the same number of successes and failures. That p_1 can be conjoined with p_2 without affecting p_1's predictive power demonstrates a connection between the two predicates, in this case suggesting that $p_1 \implies p_2$.

This implication is detectable because the experiment is run using complete data collection. Results taken using sparse sampling rates would have made this detection impossible, given the likelihood of p_2 being unobserved in a run where p_1 was true. Additionally, it is detected in the absence of other bugs. Intuitively an unrelated bug would cause faults in different program runs, allowing the analysis to distinguish between unrelated sets of predictors, but this was not tested.

This result provides evidence that complex predicate analysis can automatically group related predicates in ways traditional CBI analysis does not, including the discovery of partial, sub-bug, and perfect predictor hierarchies and implications. Grouping related predictors statistically provides insight into program structure and execution features that can be used in debugging. This example reiterates that complex predicates can collaborate with tools like BTRACE that produce an execution trace from a set of predicates. Cooperative Bug Isolation can therefore utilize techniques that previously required detailed execution information by generating a facsimile from statistical data.

6. EXPERIMENTS

This section presents quantitative data about the ideas presented in previous sections. This data was collected using the Siemens test suite [7] as maintained by the Galileo Software-artifact Infrastructure Repository (SIR) [4, 17]. There are two configurable parameters for the experiments: the sampling rate and the *effort* cutoff (described in Section 4). Unless specified, the default sampling rate is 1 (i.e., complete data collection) and the default *effort* is 5% (only predicates that are reachable from each other by exploring less than 5% of the program are considered). Each Siemens application is available in multiple variants with different bugs: as few as 7 variants of `print_tokens` and as many as 41 variants of `tcas`. We report aggregate results by averaging the relevant measures across all variants of each Siemens application. We use CodeSurfer to build program dependence graphs and to compute effort and correlation metrics as described in Section 4.

6.1 Top-Scoring Predicates

Table 8 shows the number of buggy variants of each Siemens application used in our experiments. For each of these 130 buggy programs, we perform a statistical debugging analysis using the iterative redundancy elimination algorithm discussed earlier, at a sampling rate of 1. We then determine whether the top-scoring bug predictor is a simple predicate, a complex conjunction, or a complex disjunction. Table 8 reports how often each of these three kinds of predictors appears with the highest score.

As can be seen, conjunctions dominate, with 86% of programs tested selecting a conjunction as the top bug predictor. This confirms that complex predicates can diagnose failures more accurately than simple predicates alone. Because each Siemens program variant has only a single bug, it is to be expected that disjunctions are not needed as frequently. Thus, disjunctions play a smaller but significant role, especially in the case of `schedule`. Even in single-bug programs, disjunctions can be helpful if no one simple predicate perfectly aligns with the condition that causes failure.

Section 6.4 explains that the chance of observing a complex predicate shrinks quadratically with the sampling rate. However, conjunctions remain important even with sparse sampling. With

Table 6: Results for `ccrypt` with only simple predicates

Score	True Successes	False Successes	Predicate	Function	File:Line
0.431678	0	342	xreadline == 0	`prompt`	traverse.c:122
0.385597	200	342	res == (char *)0	`xreadline`	xalloc.c:43

Table 7: Results for `ccrypt` with complex predicates

Score	True Successes	False Successes	Predicate	Function	File:Line
0.72814	0	342	xreadline == 0 *and* res == (char *)0	`prompt` `xreadline`	traverse.c:12 xalloc.c:43

		Type of Top Predictor		
Application	Variants	Simple	Conj.	Disj.
`print_tokens`	7	0	7	0
`print_tokens2`	10	0	10	0
`replace`	31	3	28	0
`schedule`	9	0	0	9
`schedule2`	9	1	8	0
`tcas`	41	1	40	0
`tot_info`	23	2	20	1
Overall	130	5%	86%	7%

Table 8: Number of buggy application variants in experiments, and number (percentage) having a complex predicate as the top-scoring predictor. Results are shown for complete data collection (sampling rate 1).

1/100 sampling, 84% of the Siemens applications excluding `tcas` still show a conjunction as the top-scoring predictor. `tcas` has no looping or recursion and every function is called at most twice. Thus, no simple predicate can be observed more than twice in a single `tcas` run, and two-predicate conjunctions are infrequently observed when sampling is sparse. Only 25% of the `tcas` experiments have a complex predicate as the top predictor.

6.2 Effectiveness of Pruning

Even when restricted to binary conjunction and disjunction, complex predicates could substantially increase the analysis workload if naïvely implemented. Section 4 suggests heuristics for pruning complex predicates that are unlikely to be useful or understandable to a programmer, while Section 3.4 describes how to compute an upper bound on a predicate's score. Figure 1 shows that these measures are highly effective in practice. On average, 53% of candidate complex predicates are discarded because the *effort* as defined in Definition 4 would require traversing more than 5% of the application code. Computing the exact scores of the remaining 47% of the complex predicates takes, on average, 9 minutes. Pruning complex predicates whose upper bound of the *Importance* scores, computed per Section 3.4, are lower than the scores of their constituent simple predicates eliminates a further 25% of the complex predicates. This pruning reduces the average time required per analysis to 6 minutes. Only 22% of complex predicates remain to have their exact scores computed, of which roughly a fourth (6% of the initial pool) are retained as potentially interesting bug predictors. Thus we find that the techniques proposed earlier significantly reduce the computational load required to identify a useful, high-scoring subset of complex predicates. Pruning based on upper bound estimation can be applied more aggressively by applying stricter thresholds. For example, during redundancy elimination, we are only interested in a complex predicate with score larger than the best simple predicate. Using this threshold eliminates, on average, 40% of the complex predicates, as opposed to the 25% pruned by the moderate threshold described earlier. This analysis takes just a minute, on average, to complete. Since these experiments have been conducted on nodes of a heterogeneous Condor cluster [15], the absolute timing measurements mentioned above are not precise. But the relative improvement is still valid because, for each application, the time taken for the different configurations (no pruning to aggressive pruning) are measured on the same Condor node.

6.3 Effect of Effort

Figure 2 shows how the number of interesting conjunction and disjunction predicates (Definition 3) varies at four different *effort* levels. As expected, as *effort* increases more predicates are evaluated and so more interesting predicates are found.

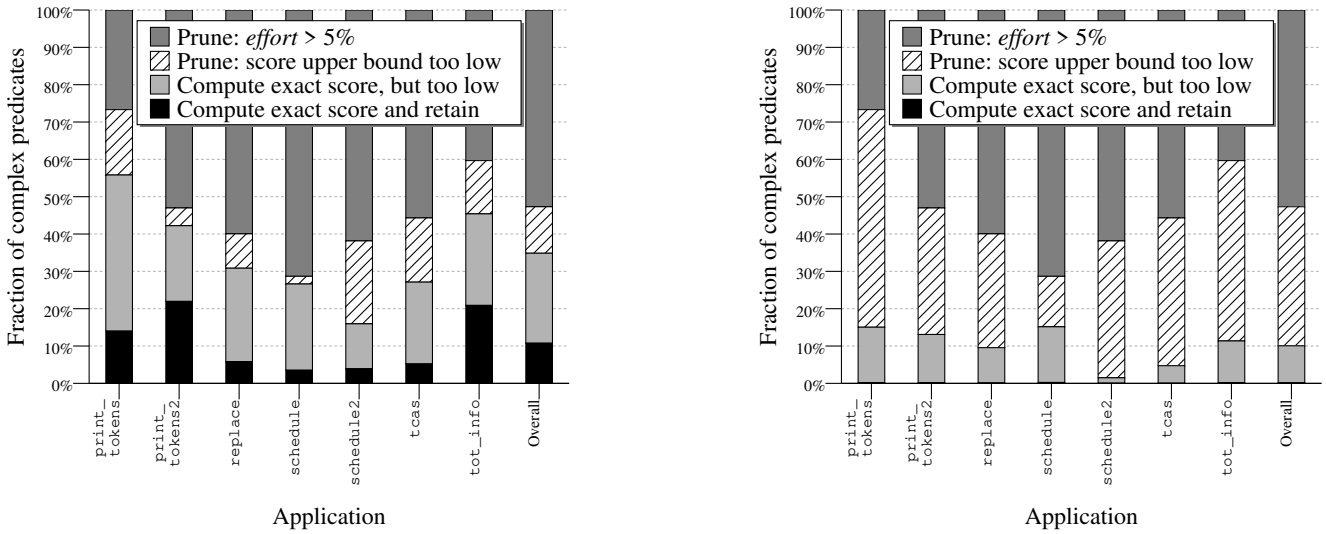
6.4 Effect of Sampling Rate

Real deployments of CBI use sparse random sampling of simple predicates to reduce performance overhead and protect user privacy. Prior work [12] has recommended sampling rates of 1/100 to 1/1,000 to balance data quality against performance and privacy concerns. However, a pair of independent features each observed in 1/100 runs have only a 1/10,000 chance of being observed together, raising doubts whether interesting complex predicates will be found in sparsely sampled data. (Note, however, that certain complex predicate values can be "observed" even if one simple component is not, per the rules in Table 1.)

Table 9 shows the average number of interesting simple and complex predicates found is each Siemens application for both complete data collection (1/1) and two realistic sampling rates. Figure 3 shows the same information at additional, denser rates for `print_tokens2`. Other Siemens applications behave similarly and are omitted for brevity.

The number of interesting disjunctions is always very low (order of tens) compared to interesting conjunctions. At sampling rates lower than 1/10, there is a sharp drop in the number of interesting conjunctions. This is likely due to the shrinking odds of observing both components of a conjunction within a single run. Despite the sharp drop, the number of interesting conjunctions is still comparable to the number of interesting simple predicates even at 1/1,000 sampling. This shows that interesting complex predicates can still be found at sparse but realistic sampling rates.

A puzzling trend in Figure 3 is that all three curves rise for a brief interval before dropping off. Other Siemens applications exhibit a similar bump. The bumps in the conjunction and disjunction curves could be attributed to the bump in the simple predicates curve. Any



(a) Conjunctions

(b) Disjunctions

Figure 1: Avoiding computing exact scores by pruning complex predicates. "Overall" summarizes the entire Siemens suite.

	Simple			Conjunctions			Disjunctions		
Application	1/1	1/100	1/1,000	1/1	1/100	1/1,000	1/1	1/100	1/1,000
print_tokens	1,305	837	399	231,658	44,279	8,514	2,518	8	-
print_tokens2	494	480	225	31,504	13,874	1,328	167	9	-
replace	699	431	128	23,381	4,259	196	398	2	-
schedule	239	236	10	1,486	65	-	61	-	-
schedule2	273	165	43	2,232	6	0	22	-	-
tcas	562	9	-	9,428	1	-	75	-	-
tot_info	404	429	172	24,104	29,652	3,378	177	27	0

Table 9: Sampling rate vs. number of interesting predicates, averaged across all variants of each Siemens application. "-" marks an average count of exactly zero, i.e., no interesting predicates in any variant.

increase in the number of interesting simple predicates is likely to produce a greater increase in the number of interesting complex predicates, especially because the additional simple predicates are likely to be redundant (as explained later).

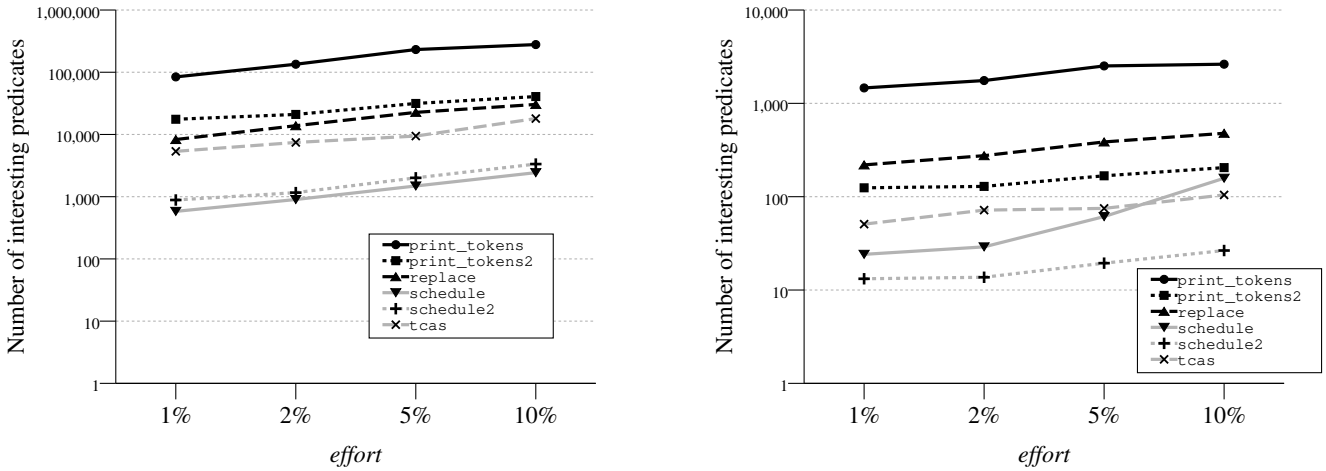
The transient increase in the number simple predicates at moderate sampling rates was previously undiscovered and initially seems counterintuitive. Closer inspection of experimental results reveals two scenarios where this can happen.

The first scenario happens due to an ad hoc but perfectly reasonable elimination of seemingly identical predicates. As an example, consider the predicates p: `a == b` and q: `a >= b`. If they have the same score, it is useful to just retain p as it is a more stringent condition than q. However, this does not mean that `a > b` was never true. `a > b` may be observed during some runs but does not affect the outcome of `a >= b` if `a == b` also happens to be true in those runs. However, at sampling rates lower than 1, only `a > b` may be observed in some runs and hence the number of runs in which p and q are observed true, and consequently their scores may be different. Thus, the ad hoc elimination heuristic performs less effectively at lower sampling rates, leading to an increase in the number of interesting simple predicates

To understand the second scenario, consider a predicate p for which $S(p \text{ obs}) = S(p)$ and $F(p \text{ obs}) = F(p)$. In other words, p is true at least once in every run in which it is observed. From Equation 1, $Increase(p) = 0$. In a run in which p was observed, it may also be false at least once. As we reduce the sampling rates, only the false occurrences may be recorded in some runs and hence the two equalities may no longer hold. As a result $Increase(p)$ may be nonzero and if it becomes positive, then a predicate that was not interesting at higher sampling rates becomes interesting at lower sampling rates.

7. RELATED WORK

Daikon [3] detects invariants in a program by observing multiple program runs. Invariants are predicates generated using operators like sum, max, etc. to combine program variables and collection (e.g., array) objects. Daikon is intended for many uses beyond bug isolation, and so it monitors a much larger set of predicates than CBI. This makes scalable complex predicate generation more difficult. However, Dodoo et al. [2] have successfully extended the work to generate implications from the simpler, measured predicates. Dodoo et al. alternate clustering and invariant detection to find invariant implications over a set of program runs. The initial clustering is performed using the k-means algorithm [8], with program runs represented as normalized vectors of scalar variable values. Since CBI represents run information as bit-vectors this



(a) Conjunctions (b) Disjunctions

Figure 2: Variation of number of interesting predicates with *effort*

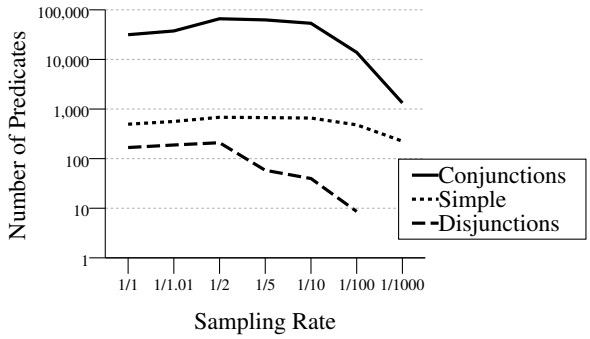


Figure 3: Sampling rate vs. number of interesting predicates, averaged across all variants of `print_tokens2`. Other Siemens applications are similar.

technique can be applied essentially unchanged.

Daikon's implication generation extends its vocabulary of possible invariants. CBI's focus is detection of bug predictors, which under sparse sampling conditions can rarely be identified as invariant. Additionally, the existence of an implication is of questionable value in this project; the implication revealed in Section 5.2 is an interesting and potentially useful side-effect of our analysis, but only because it involves identified bug predictors. The approach described in this paper is better suited to the goals and analysis techniques of CBI. There are no known attempts to use Daikon under sparse sampling conditions.

DIDUCE [5] detects invariant bits of program values during an initial training phase. During the checking phase, DIDUCE reports each invariant violation as it occurs, then relaxes the invariant to accept the new value. Unlike Daikon and CBI, DIDUCE tightly couples data collection and evaluation. Because of this coupling, neither our nor Daikon's offline style of predicate generation is readily combined with DIDUCE's framework.

SOBER [16] is a statistical debugging tool similar to CBI. Where CBI considers only whether a predicate was ever observed true during an execution, SOBER estimates the likelihood of it being true at any given evaluation. SOBER data is a probability vector, with each value representing the estimated chance of a simple predicate being true when observed. The similarity in collected data means that similar techniques for complex predicate generation are applicable. The three-valued logic described in Section 3.2 could be replaced with joint-probability when generating conjunctions; De Morgan's law can be applied to generate disjunctions. Our usability metrics can be used on the resulting data. There are no known experiments using SOBER under sparse sampling conditions. Complex predicate generation removes a key advantage of SOBER: predicate scores result directly from the number of actual predicate evaluations. Complex predicates generated by this technique are never truly evaluated, so their probability values would have little connection to actual program execution. Whether this would affect their usefulness is unknown.

Jones and Harrold [9] discuss a fault localization technique using statement coverage as predicates and weighted failure rate as the scoring metric. The ideas discussed in our paper, including the pruning techniques, can be applied directly to this technique. Jones et al. [10] also explore visualization of program-execution data, such as failure data. Compound predicates relate behavior at multiple program points, and therefore may be difficult to visualize. Presenting compound predicates in a way that programmers can readily understand remains an open problem.

Haran et al. [6] analyze data from deployed software to classify executions as *success* or *failure*. They use tree-based classifiers and association rules to model "failure signals." Tree-based classifiers can encode both conjunctions and disjunctions whereas association rules cannot encode disjunctions when limited to a constant size.

8. CONCLUSIONS AND FUTURE WORK

We have demonstrated that compound Boolean predicates are useful predictors of bugs. Our experiments show qualitative and quantitative evidence that statistical debugging techniques can be effectively applied to complex predicates, and that the resulting analysis provides improved results. We describe two methods of eliminating predicate combinations from consideration, making the task of computing complex predicates more feasible. The first employs three-valued logic to estimate set sizes and thereby estimate

the upper bound of the score of a complex predicate. The second uses distances in program dependence graphs to quantify the programmer effort involved in understanding complex predicates.

These techniques help the statistical debugging analysis scale up to handle the large number of candidate predicates we consider. However, using the analysis results in debugging can require sifting through a large number of less useful predicates that also pass automated inspection. Further shrinking this list while retaining useful predictors remains an important open problem. Most identified bug predictors redundantly describe the same small set of program failures. Thus the bi-clustering algorithm of Zheng et al. [19] may be promising as it was designed to handle multiple predictors for the same bug. Automated analyses which further process predictor lists, such as BTRACE [11], may also benefit from the richer diagnostic language offered by the work presented here.

Acknowledgments

We would like to thank Anne Mulhern for her insightful comments on an earlier draft of this paper.

9. REFERENCES

[1] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, 2005.

[2] N. Dodoo, A. Donovan, L. Lin, and M. D. Ernst. Selecting predicates for implications in program analysis, March 16, 2002. Draft. `http://pag.csail.mit.edu/~mernst/pubs/invariants-implications.ps`.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[4] G. R. H. Do, S. Elbaum. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[5] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM Press.

[6] M. Haran, A. Karr, M. Last, A. Orso, A. A. Porter, A. Sanil, and S. Fouché. Techniques for classifying executions of deployed software to support software engineering tasks. *IEEE Transactions on Software Engineering*, 33(5):287–304, 2007.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[8] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, Sept. 1999.

[9] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, New York, NY, USA, 2005. ACM Press.

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

[11] A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In P. Sestoft, editor, *15th European Symposium on Programming*, pages 246–263, Vienna, Austria, Mar. 2006. Springer.

[12] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154, New York, NY, USA, 2003. ACM Press.

[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, New York, NY, USA, 2005. ACM Press.

[14] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[15] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[16] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical model-based bug localization. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 286–295, New York, NY, USA, 2005. ACM Press.

[17] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software-artifact intrastructure repository. `http://sir.unl.edu/portal/`, Sept. 2006.

[18] E. W. Weisstein. Boolean function. *MathWorld–A Wolfram Web Resource*, Dec.20 2006. `http://mathworld.wolfram.com/BooleanFunction.html`.

[19] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112, New York, NY, USA, 2006. ACM Press.