Indexing Noncrashing Failures: A Dynamic Program Slicing-Based Approach *

Chao Liu
Department of Computer Science
University of Illinois-UC
Urbana, IL 61801 USA
chaoliu@cs.uiuc.edu

Xiangyu Zhang
Department of Computer Science
Purdue University
West Lafayette, IN 47907 USA
xyzhang@cs.purdue.edu

Jiawei Han
Dept of Computer Science
University of Illinois-UC
Urbana, IL 61801 USA
hanj@cs.uiuc.edu

Yu Zhang
Dept of Computer Science
Purdue University
West Lafayette, IN 47907 USA
zhangyu@cs.purdue.edu

Bharat K. Bhargava
Dept of Computer Science
Purdue University
West Lafayette, IN 47907 USA
bb@cs.purdue.edu

Abstract

Recent software systems usually feature an automated failure reporting component, with which a huge number of failures are collected from software end-users. With a proper support of failure indexing, which identifies failures due to the same fault, the collected failure data can help developers prioritize failure diagnosis, among other utilities of the failure data. Since crashing failures can be effectively indexed by program crashing venues, current practice has seen great success in prioritizing crashing failures.

A recent study of bug characteristics indicates that as excellent memory checking tools are widely adopted, semantic bugs and the resulting noncrashing failures have become dominant. Unfortunately, the problem of how to index noncrashing failures has not been seriously studied before. In previous study, two techniques have been proposed to index noncrashing failures, and they are T-PROXIMITY *and* R-PROXIMITY. *However, as* T-PROXIMITY *indexes failures by the profile of the entire execution, it is generally not effective because most information in the profile is fault-irrelevant. On the other hand, although* R-PROXIMITY *is more effective than* T-PROXIMITY*, it relies on a sufficient number of correct executions that may not be available in practice. In this paper, we propose a dynamic slicing-based approach, which does* not *require any correct executions, and is comparably effective as* R-PROXIMITY. *A detailed case study with* gzip *is reported, which clearly demonstrates the advantages of the proposed approach.*

*The work was supported in part by the U.S. National Science Foundation NSF ITR/CCR-0325603, IIS-05-13678, NSF BDI-05-15813, and IIS-02-42840. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.

1 Introduction

Software end-users are the most powerful testers: They keep revealing software faults (*i.e.*, bugs) in released software that has undergone rigorous in-house testing. In order to leverage end-users' testing power, failure reporting components have been widely adopted in deployed software, with Microsoft Dr. Watson System [2] and the Mozilla Quality Feedback Agent [3] being the two most typical examples. When a program fails, the failure reporting component automatically collects relevant information of the failure, and (with the user's permission) reports it to software vendors for failure diagnosis and patches. Recently, third-party libraries that implement such failure reporting functionalities have been released for both C++ and Java, so that any programs, disregarding their complexity, can have their own failure reporting channels. The authors have seen this in Google Toolbar and BitTorrent, just to name a few.

The automatically collected failures reflect how the software is exercised in practice, and what software faults really bother the users. Therefore, an appropriate analysis of such failure repository will provide invaluable guidance for software maintenance and development. However, most utilities of such reported failures rely on the resolution of a critical problem: *failure indexing*, which asks *how to identify all failures due to the same fault*. If failure indexing can be nicely performed, most utilities of the collected failure data will become routine work. For example, some typical and important utilities are

- **Failure Prioritization**: Reported failures have different levels of severity, and the most severe failure should be diagnosed and fixed first. Typically, the severity of a failure is determined by how many reported failures are due to the same fault as this particular one. With the support of failure indexing, failures due to the same fault can be easily identified, and consequently the diagnosis of failures can be prioritized.

- **Duplicate Failure Removal**: Because of the sheer number of reported failures, manual diagnosis of every failure is impractical. With the support of failure indexing, developers only need to diagnose one failure from each failure set that arises from the same fault.

- **Patch Suggestion**: When a new failure occurs, it can be easily checked whether this failure has been solved before through failure indices. If yes, the failure reporter can be automatically directed to the patch to resolve the problem.

Failure indexing can sometimes be straightforward, especially when apparently effective failure signature exists. A case in point is *crashing failures*, which manifest as program crashes. Usually, crashing failures are incurred by *memory bugs*, such as dereferences of NULL pointers and memory corruptions. For crashing failures, the crashing venue (*e.g.*, the call stack trace at program crashes) is a great failure signature because failures from the same fault *tend to* (but not always) exhibit the same crashing venue. By virtue of the nearly one-to-one mapping relationship between crashing venues and faults, indexing of crashing failures has been very successful in practice, as evidenced by the success of the Microsoft Dr. Watson System.

However, in the case of *noncrashing failures*, failure indexing becomes elusive because no unanimous signature like a crashing venue for crashing failure exists. The reason is that noncrashing failures are mostly incurred by *semantic bugs*, which usually cause program malfunctions (*e.g.*, incorrect outputs) without crashing the program. Since no apparently effective signature exists any more, how to index noncrashing failures becomes an interesting and challenging problem.

Previous studies propose two failure proximity measures, which can be used to index noncrashing failures. Podgurski et al. [19] propose the T-PROXIMITY, which assigns a small dissimilarity value to pairs of failures that exhibit similar execution traces. In consequence, under T-PROXIMITY, failures with similar behaviors (*e.g.*, similar branching actions) are indexed together. Because T-PROXIMITY does not rely on the crashing venue, it can be used to index noncrashing failures. But one shortcoming of T-PROXIMITY is that failures due to different faults can exhibit quite similar behaviors (especially before faults are triggered), which renders T-PROXIMITY ineffective in discriminating failures due to different faults. Based on this observation, Liu and Han propose R-PROXIMITY, which extracts *fault-relevant* information from program failures, and indexes failures accordingly [16]. Because only fault-relevant information is considered, R-PROXIMITY is shown to be more effective than T-PROXIMITY in distinguishing failures due to different faults.

However, the effectiveness of R-PROXIMITY does not come for free. The fault-relevant information is extracted from each failure by contrasting the failure against a set of passing executions. Unfortunately, the availability of such a set of passing executions cannot be freely assumed in practice. In the first place, non-trivial overhead will be imposed on user sides if passing executions, in addition to failures, are collected from end-users. More importantly, users are very sensitive to privacy which could be potentially infringed by the collection of correct executions. This explains why only program failures are collected in practice. In general, the availability of a non-trivial set of passing executions cannot be assumed. Therefore, in this paper, we investigate how to index noncrashing failures as effectively as R-PROXIMITY but *without* assuming any passing executions.

We propose a dynamic program slicing-based approach to indexing noncrashing failures. Specifically, we take the backward slices from the program failure point as the failure signature, and quantify whether two failures are due to the same fault according to the similarity between their corresponding backward slices. For noncrashing failures, the failure point is the source code that generates the first erroneous output. The advantages of this dynamic slicing-based approach are as follows.

- In comparison with T-PROXIMITY, we use dynamic slicing techniques to exclude fault-irrelevant information that is otherwise considered by T-PROXIMITY. For the same reason as R-PROXIMITY, exclusion of the fault-irrelevant information will improve the effectiveness in indexing noncrashing failures.

- In comparison with R-PROXIMITY, the dynamic slicing-based approach completely eliminates the need of any passing executions, and hence can be used in practice where only program failures are collected.

We will use a detailed case study with gzip to demonstrate the above claims.

Although current practice only reports crashing failures from user sites, indexing noncrashing failures is *not* an unrealistic problem. A recent study of bug characteristics [13] shows that semantic bugs have become dominant because of the wide adoption of excellent memory monitoring tools, such as Valgrind and Purify. Specifically, the authors find that semantic bugs account for 81.1-86.7% of the 364 bugs

they examined, and the ratio is projected to increase as software matures. These semantic bugs mainly manifest as wrong outputs, performance degradation, and incorrect functionality, which are all noncrashing failures. More importantly, the authors find that 71.9-83.9% of security bugs are also semantic bugs, and security break-ins always take place without crashing the program. Because of the increasing dominance of semantic bugs and the resulting noncrashing failures, we believe that the collection of noncrashing failures will be supported in the near future. Because no unanimous indexing techniques exist for indexing noncrashing failures, a systematic study of existing ones and investigation of new indexing techniques are in great need.

In summary, we make the following contributions in this paper.

- We pose the problem of indexing noncrashing failures, an increasingly critical problem due to the dominance of semantic bugs in the future.
- We propose a *distance metric-based framework*, which incorporates existing approaches and our proposed one. In order to foster future developments, a quantitative measure of indexing effectiveness is proposed within this framework, so that future techniques can be objectively evaluated.
- We propose a dynamic slicing-based approach to indexing noncrashing failures, which are advantageous over existing techniques. To the best of our knowledge, this is the first attempt of using dynamic slices in failure indexing.

The rest of this paper is organized as follows. Section 2 explains the distance metric-based framework for failure indexing, and Section 3 discusses our dynamic slicing-based approach with references to the framework. We report the experiment results in Section 4. Related work and threats to validity are discussed in Section 5, and Section 6 concludes this study.

2 A Distance Metric-based Framework for Failure Indexing

Intuitively, failure indexing tries to compute a failure signature (*i.e.*, the index) for each program failure, such that failures due to the same fault can be identified through the similarity between failure signatures. While this explanation suffices for intuitive understanding, a precise formulation facilitates unambiguous discussion and potentially fosters healthy development in the future. Therefore, in this section, we present a distance metric-based framework for failure indexing, which incorporates both existing approaches and our proposed one.

2.1 Failure Indexing in Formulation

Suppose a set of n failures $X = \{x_1, x_2, \cdots, x_n\}$ is collected from a program $\mathcal{P}$, and the n failures are due to m (unknown) faults $F = \{f_1, f_2, \cdots, f_m\}$. An oracle function Φ, which is also unknown, specifies the *due to* relationship between X and F, namely,

$$\Phi(x) = k \iff \text{the failure } x \text{ is due to fault } f_k,$$

and the fault f_k is the *root cause* of the failure x. For clarity, we only consider failures that are induced by one fault at runtime even though multiple faults may reside in the program.

The oracle function Φ partitions the set of failures X into m mutually exclusive and collectively exhaustive sets:

$$\underset{k=1,2,\cdots,m}{S_k} = \{x_i | \Phi(x_i) = k, \text{ for i} = 1, 2, \cdots, \text{n}\}.$$

For any failure x_i, $G(x_i)$ is the *failure group* that x_i belongs to, and $G(x_i)$ includes all the failures due to the same fault as x_i, namely,

$$G(x_i) = \{x_j | \Phi(x_j) = \Phi(x_i), \text{for } j = 1, 2, \cdots, n\},$$

and x_i is a *member* of $G(x_i)$. With the above definitions, we can formulate failure indexing within a distance metric-based framework as below.

A failure indexing technique is a function pair $(\phi, \mathcal{D})$, where the function ϕ is a *signature function*, and the function $\mathcal{D}$ is a *distance function* that is defined on a pair of signatures returned by ϕ. Specifically, function ϕ takes a program failure x as input, and returns a failure signature; the distance function $\mathcal{D}$ quantifies how failures are close to each other based on the similarity between their corresponding failure signatures. Usually, we require the distance function $\mathcal{D}$ be a metric, meaning that the following four properties are satisfied:

(1) $\mathcal{D}(\alpha, \beta) \geq 0$ (non-negativity),
(2) $\mathcal{D}(\alpha, \beta) = 0$ iff $\alpha = \beta$ (identity),
(3) $\mathcal{D}(\alpha, \beta) = \mathcal{D}(\beta, \alpha)$ (symmetry),
(4) $\mathcal{D}(\alpha, \gamma) \leq \mathcal{D}(\alpha, \beta) + \mathcal{D}(\beta, \gamma)$ (triangle inequality),
where α, β, and γ are three failure signatures.

Then a pair-wise distance matrix $M_{(\phi,\mathcal{D})}$, which is called the *proximity matrix*, can be calculated for the given set of n failures, where

$$M_{(\phi,\mathcal{D})}(i, j) = \mathcal{D}(\phi(x_i), \phi(x_j)).$$

A small value of $M_{(\phi,\mathcal{D})}(i, j)$ means that failures x_i and x_j are similar, and are likely to be indexed together by the indexing technique $(\phi, \mathcal{D})$. Each indexing technique defines a failure proximity, which is embodied by the proximity matrix.

Table 1. Different Indexing Techniques under the same Distance Metric-based Framework

	$\phi(x)$	$\mathcal{D}(\phi(x_i), \phi(x_j))$
The optimal index	$\Phi(x_i)$, *i.e.*, the root cause of x_i	1 if two root causes are different and 0 otherwise
T-PROXIMITY	Profile of the whole execution	Euclidean distance and city-block distance
R-PROXIMITY	A ranking of fault-relevant predicates	Weighted Kendall's tau distance
Index by dynamic slices	Dynamic slices from the program failure point	Set-based distance

Within this framework, the optimal indexing technique $(\phi, \mathcal{D})$ will minimize the intra-group distances,

$$min \sum_{\Phi(i)=\Phi(j)} M_{(\phi,\mathcal{D})}(i,j),$$

and meanwhile maximize the inter-group distances,

$$max \sum_{\Phi(i)\neq\Phi(j)} M_{(\phi,\mathcal{D})}(i,j).$$

Certainly, distances defined on different failure signatures must be first normalized before comparison. We will discuss a normalized measure in Section 2.2.

Previous studies, as well as the optimal indexing and our dynamic slicing-based approach, all fit into this distance metric-based framework, and Table 1 lists what functions are actually used in different indexing techniques. Especially, the first row of Table 1 indicates that if the oracle function Φ were known, the optimal indexing becomes a routine work. Because Φ can only be obtained through expensive manual work, our objective is to investigate automated indexing techniques that *approximate* the optimal one. In the next subsection, we propose an evaluation metric that quantifies the effectiveness of each indexing technique.

2.2 An Evaluation Metric

An evaluation metric should be independent of how indexing techniques are implemented, *i.e.*, it does not need to know what ϕ and $\mathcal{D}$ are; instead, the evaluation metric should only care about the proximity matrices that are generated by different indexing techniques. Besides the independence of indexing details, a good metric needs to consider the following two aspects:

- **Cohesion**: To what extent failures in the *same* group are close to each other;
- **Separation**: To what extent failures in *different* groups are separated from each other.

An excellent indexing technique will generate a proximity matrix that exhibits both high cohesion and high separation. In order to consider both cohesion and separation simultaneously, we propose the following metric, which borrows the idea of the Silhouette coefficient (SC) [20]. The Silhouette coefficient was originally proposed to evaluate the internal structure of data clustering results without knowing what data should be clustered together. Here, as we do know what failures should be indexed together, the Silhouette coefficient can be adapted to evaluate how effective an indexing technique is.

Specifically, the Silhouette coefficient (SC) of each failure x_i is defined as

$$SC(x_i) = \frac{b_i - a_i}{max\{a_i, b_i\}} \quad (1)$$

where

$$a_i = \frac{\sum_{x_j \in G(x_i)} M(i,j)}{|G(x_i)|}$$

and

$$b_i = \min_{k=1,2,\cdots,m,k\neq\Phi(x_i)} \frac{\sum_{x_j \in S_k} M(i,j)}{|S_k|}.$$

Intuitively, a_i is the average distance from x_i to all other failures in the same group. To compute b_i, we first calculate the average distances between x_i and failures in S_k for all $k \neq \Phi(x_i)$, and b_i is the *minimum* value among the $m-1$ average distances.

Apparently, $SC(x_i)$ varies between -1 and +1. A negative value is undesirable because it suggests x_i is closer to a group it does not belong to than to its own group. On the other hand, a positive value means x_i is close to other failures in the same group. After getting the Silhouette coefficients of each failure, the overall Silhouette coefficient, calculated from a proximity matrix M, is

$$SC(M) = \frac{\sum_{i=1}^{n} SC(x_i)}{n}. \quad (2)$$

Again $SC(M)$ ranges from -1 to 1, and a high value indicates that the indexing technique $(\phi, \mathcal{D})$ is effective in indexing *the* given n failures. It is easy to verify that $SC(M)$ is 1 for the optimal indexing technique.

3 Dynamic Slicing-based Failure Indexing

In this section, we discuss the dynamic slicing-based approach to noncrashing failure indexing. Specifically, Section 3.1 discusses dynamic slicing techniques that serve as the signature function ϕ, and Section 3.2 explains the distance function $\mathcal{D}$ defined on dynamic slices. Finally, in Section 3.3, we describe a technique that visualizes failure indexing result.

3.1 Dynamic Slices as Failure Signatures

Dynamic slicing, invented as a debugging aid [11], is able to identify a subset of program statements that are involved in producing a program failure. Dynamic slicing operates by observing the execution of the program on a given input and collecting the dependences between executed statements. These dependences are used to compute dynamic slices.

Because a statement s can be executed multiple times for a given input, we distinguish different execution of the same statement s by *execution instances*. Suppose s is executed n times, we use $s_1, s_2, \cdots, s_n$ to denote the n execution instances.

A dynamic slice is computed w.r.t. a specific execution instance s_i. In this paper, as we will use dynamic slicing techniques as the signature function ϕ, dynamic slices are computed w.r.t. program *failure points*. For noncrashing failures, the failure point is the statement instance that produces the first erroneous output. We now describe different types of dynamic slices that are used in this study.

Data Slice (DS). Statements that directly or indirectly influence computation of the faulty output through chains of *dynamic data dependences* are included in data slices. Formal definitions are as follows.

Definition 1 (Dynamic Data Dependence) *An execution instance s_i of the basic statement s has a data dependence on the execution instance t_j of the statement t, denoted as $s_i \xrightarrow{dd} t_j$, if and only if there exists a variable* `var` *whose value is defined at t_j and is then used at s_i.*

Definition 2 (Data Slice) *The data slice of an execution instance s_i, denoted as $DS(s_i)$, is*

$$DS(s_i) = \{s\} \cup \bigcup_{\forall t_j,\ s_i \xrightarrow{dd} t_j} DS(t_j).$$

Figure 1 (left) shows an example of DS. It presents an execution trace instead of the static source code even though the code is self-explicit from the trace. This is also the case in the rest of the paper unless otherwise specified. In this example, there are data dependences between 30 and 40, and between 10 and 30. Therefore, the data slice of the value z at 40 includes 10, 30, and 40.

10_1.	`x=...;`	10_1.	`x=...;`
	`...`		...
20_1.	`y=...;`	20_1.	`y=...;`
	`...`		...
30_1.	`z=...x...;`	30_1.	`if (y)`
	`...`	31_1.	`z=...x...`
40_1.	`print(z)`		...
		40_1.	`print(z)`
DS(40_1)={10, 30, 40}		FS(40_1)={10,20,30,31,40}	

Figure 1. Data Slice (left) and Full Slice (left)

Note that even though dependences are defined between statement *instances*, a slice contains unique statements instead of statement instances. In other words, a statement appears in the slice only once even when multiple instances of the statement are involved in computation of the faulty value.

Full Slice (FS). Statements that directly or indirectly influence the computation of faulty output value through chains of *dynamic data and/or control dependences* are included in full slices [11].

Definition 3 (Dynamic Control Dependence) *A statement execution instance s_i of statement s has a control dependence on the execution instance t_j of statement t, denoted as $s_i \xrightarrow{cd} t_j$, if and only if*

1. *statement t is a predicate statement, and*
2. *the execution of s_i is the result of the branch outcome of t_j.*

Definition 4 (Full Slice) *The full slice of an execution instance s_i, denoted as $FS(s_i)$, is*

$$FS(s_i) = \{s\} \cup \bigcup_{\forall t_j,\ s_i \xrightarrow{dd} t_j \text{ or } s_i \xrightarrow{cd} t_j} FS(t_j).$$

Figure 1 (right) shows an example of FS. The control dependence $31_1 \xrightarrow{cd} 30_1$ renders both statements 30 and then 20 included in the full slice.

3.2 Distances between Dynamic Slices

By taking dynamic slicing as the signature function ϕ, each failure is represented by a dynamic slice. Therefore, an appropriate distance function $\mathcal{D}$ that is defined on dynamic slices is needed to complete the dynamic slicing-based failure indexing. Given that a dynamic slice is essentially a set of statements, any distance metric defined on sets suffices. In this study, we choose the Jaccard distance, which was originally proposed by Levandowsky and Winter [12].

Definition 5 (Distance between Dynamic Slices) *For any two non-empty dynamic slices e_i and e_j of the same program $\mathcal{P}$, the distance between them is*

$$\mathcal{D}(e_i, e_j) = 1 - \frac{|e_i \cap e_j|}{|e_i \cup e_j|}.$$

This distance is a valid metric. Readers interested in the proof of the triangle inequality are referred to [12].

The distance $\mathcal{D}$ completes our dynamic slicing-based approach to failing indexing. Depending on what dynamic slices are chosen as failure signatures, we have a series of four indexing techniques: FS-PROXIMITY, DS-PROXIMITY, PFS-PROXIMITY and PDS-PROXIMITY, whose meanings are self-explained.

3.3 Failure Indexing in Visualization

The Silhouette coefficient discussed in Section 2.2 numerically summarizes the effectiveness of an indexing technique; consequently, different indexing techniques can be quantitatively compared. However, the ultimate goal of failure indexing is not to compare different techniques, but rather to help developers explore a (potentially huge) set of failures. A typical task of failure exploration is to identify the largest subset of failures that are likely due to the same fault for the purpose of failure prioritization. For this reason, we believe that a frontend that visualizes the indexing result of a set of failures will greatly assist users' failure exploration. In addition, the visualization also provides us with an intuitive approach to comparing different indexing techniques, *i.e.*, we can visually assess the cohesion and separation of a given indexing result.

For the same reason as the Silhouette coefficient, the visualization should only rely on the proximity matrix M. The dependence on neither original failure data nor failure signatures makes it compatible with any distance metric-based failure indexing techniques to be developed in the future. For this reason, we choose to use the multi-dimensional scaling (MDS) techniques [5], which visualize the proximity between the n failures given a proximity matrix M.

The obstacle that MDS techniques want to overcome is that the n objects whose pair-wise distances are specified by M could originally reside in a very high-dimensional space. For example, in our case, each failure is in a space of hundreds of dimensions because a typical slice contains hundreds of statements. Apparently, we cannot visualize the proximity between the n failures in the original space. Instead, what we can do is to re-arrange them in a specific way in a much lower (usually 2) dimensional space such that the pair-wise distances are *best* preserved. Readers interested in the technical details of MDS are referred to [5].

We call the visualization of an indexing result a *proximity graph*. Since the only objective of MDS techniques is to best preserve the original distances in a much lower dimensional space, the axes in a proximity graph are meaningless. A caveat that one should keep in mind while interpreting a proximity graph is that the proximity graph is *not* a projection of the original data into a low-dimensional subspace. Explicitly, a large distance between two objects in a proximity graph just indicates that the two objects are far from each other in the original space. No projection should be applied to proximity graphs.

4 Experiment Result

In this section, we report on a case study with gzip-1.2.3, which demonstrates the effectiveness of dynamic slicing-based indexing techniques. Before going into details about experiment result, let us first examine the experiment setup in Section 4.1.

4.1 Experiment Setup

The subject program gzip, together with the accompanying test suites, is obtained from the "Software-artifact Infrastructure Repository" (SIR) [10], which "is a repository of software-related artifact meant to support rigorous controlled experimentation." It has 6,184 lines of C code, excluding blanks and comments, as measured by the SLOCCount Tool[1], and the accompanying test suite contains 217 test cases.

Two "subclause-missing" bugs are seeded into the source code, as depicted in Figure 2. There are 82 failures when both faults are enabled. In particular, all these failures are noncrashing failures, *i.e.*, manifesting as incorrect outputs with no crashes.

Table 2. Failure Group Determination Table

Situation	Fails or Pass		Failure Group
	Fault 1	Fault 2	
1	Pass	Pass	$x \notin S_1$ and $x \notin S_2$
2	Fail	Pass	$x \in S_1$ and $x \notin S_2$
3	Pass	Fail	$x \notin S_1$ and $x \in S_2$
4	Fail	Fail	$x \in S_1$ and $x \in S_2$

For evaluation purpose, we need to determine the failure group for each program failure. Precisely, one needs to manually investigate each failure x, and decides whether $x \in S_1$ or $x \in S_2$, or even both for some extreme cases. However, manual examination of the 82 failures is not a big fun at all; plus, more extended experiments cannot rely on

[1] http://www.dwheeler.com/sloccount/

```
661 ulg deflate()
    {
        ...
675   while(lookahead != 0){
        ...
      #ifndef FAULT_1
686     if (hash_head != NIL && prev_length < max_lazy_match
      #else
686     if (hash_head != NIL
      #endif
687           && strstart - hash_head <= MAX_DIST ) {
            ...
692         match_length = longest_match (hash_head);
703     }
        ...
707     if (prev_length >= MIN_MATCH && match_length <= prev_length) {
          ...
711       flush = ct_tally(strstart-1-prev_match, prev_length - MIN_MATCH);
          ...
719       strstart++;
          ...
732     } else if (match_available) {
          ...
738       if (ct_tally (0, window[strstart-1])) {
739          FLUSH_BLOCK(0), block_start = strstart;
740       }
741       strstart++;
742       lookahead--;
743     } else {
          ...
750     }
        ...
759   } // end of while

762   return FLUSH_BLOCK(1);
763 }
```

Fault 1: A subclause missing error in deflate.c

```
580 local ulg deflate_fast()
    {
        ...
587     while(lookahead != 0){
          ...
      #ifndef FAULT_2
596       if (hash_head != NIL && strstart - hash_head <= MAX_DIST) {
      #else
596       if (hash_head != NIL) {
      #endif
601         match_length = longest_match (hash_head);
            ...
604       }
605       if (match_length >= MIN_MATCH) {
            ...
608         flush = ct_tally(strstart-match_start, match_length - MIN_MATCH);
610         lookahead -= match_length;

615         if (match_length <= max_insert_length) {
              ...
626           strstart++;
627       }else {
              ...
635       }
636       } else {
            ...
639         flush = ct_tally (0, window[strstart]);
            ...
642       }
          ...
652     } // end of while
653     return FLUSH_BLOCK(1); /* eof */
654 }
```

Fault 2: Another subclause missing error in deflate.c

Figure 2. Two Seeded Faults in Gzip-1.2.3

manual labeling. Therefore, we propose to determine the failure group for each failure through the following procedure, which we believe can accurately determine the true failure group membership for each failure.

For each subject program, we first activate both faults and run the faulty program through the whole test suite. This gives the set of failures that we want to index. Then we run through the test suite with one and only one fault enabled each time, and consult Table 2 to determine the failure group for each failure.

Table 2 presents the four situations that correspond to the four outcome (fail or pass) combinations when a failing test case is subject to each fault separately. In the case study with gzip, 65 failures fall into S_1, and the other 17 fall into S_2. This suggests that no failures fall into Situation 1 and 4. Basically, Situation 1 represents a small-probability event that only two faults together can fail a test case, but not by either one. In other words, the two faults need to collaborate to fail the test case. Situation 4 represents a reasonable scenario, but is nevertheless unobserved in our case study.

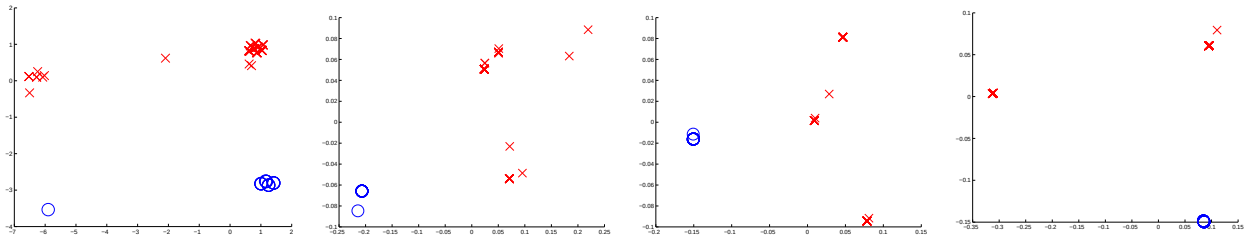
As one may have noticed, here we have not considered scenarios with more than two faults. We focus our discussion on the two-fault scenario because (1) the purpose of this study is to *compare* the dynamic slicing-based approach with existing techniques, and (2) we believe that no fundamental difference exists between two-fault and three-fault scenarios in order to study the indexing effectiveness. Therefore, we restrict our case studies to the two-fault scenarios in this paper, and leave more-fault scenarios to future work.

4.2 Comparison with T- and R-Proximity

We manually check the 82 failures from the two faults, and find all failures have the same failure point. This suggests that indexing by the failure point, which is the simplest slice, is not effective.

Figure 3 plots the proximity graphs for the four indexing techniques. Interestingly, we notice that the deviating blue circle in Figure 3(a) moves closer and closer to the blue cluster with R-PROXIMITY (Figure 3(b)) and FS-PROXIMITY (Figure 3(c)), and finally completely merges into the cluster with DS-PROXIMITY. This suggests that some failures that are not correctly identified by T-PROXIMITY can be correctly indexed by dynamic slicing-based approaches. In addition, DS-PROXIMITY has also done a great job in indexing failures in S_1: The red crosses clearly form two cohesive and dense clusters in Figure 3(d). This is a very nice property because a duplicate failure remover will have a high confidence in keeping just one representative failure from each dense cluster and throwing away the rest.

Although DS-PROXIMITY appears to achieve the best indexing result in Figure 3, its Silhouette coefficient is



(a) T-Proximity (0.352) (b) R-Proximity (0.807) (c) FS-Proximity (0.683) (d) DS-Proximity (0.508)

Figure 3. Comparison with T-Proximity and R-Proximity on Gzip-1.2.3

```
(trees.c)
958 int ct_tally (dist, lc)
    { ...
962     l_buf[last_lit++] = (uch)lc;
      ...

1011 local void compress_block(ltree, dtree)
    { ...
1026        lc = l_buf[lx++];
      ...
1032            code = length_code[lc];
1033            send_bits(code+LITERALS+1, ltree);
      ...
    }

(bits.c)
119 void send_bits(value, length)
    {  ...
132     if (bi_valid > (int)Buf_size - length) {
133         bi_buf |= (value << bi_valid);
134         put_short(bi_buf);  //Output point
      ...
    }
```

Figure 4. Data Slice of the Test Case 8

```
(trees.c)
958 int ct_tally (dist, lc)
    { ...
974         dyn_dtree[d_code(dist)].Freq++;
      }...
451 local void pqdownheap(tree, k)
    { ...
462         if (tree[n].Freq < tree[m].Freq ||...)
      ...
470                 heap[...] = ...;
      }...
483 local void gen_bitlen(desc)
    { ...
507         n = heap[h];
508         bits = tree[tree[n].Dad].Len + 1;
      ...
515         bl_count[bits]++;
      } ...
568 local void gen_codes (tree, max_code)
    { ...
581         next_code[...] = ... bl_count[...]) << 1;
      ...
594         tree[n].Code = bi_reverse(next_code[...]++, ...);
      ...
    }
740 local void send_tree (tree, max_code)
    { ...
773       send_bits(bl_tree[...].Code,...); ...
      ...
    }

(bits.c)
119 void send_bits(value, length)
    {  ...
134         put_short(bi_buf);  //Output point
      ...
    }
```

Figure 5. Data Slice of the Test Case 82

strangely low. Apparently, the low coefficient comes from the large distance between the two red clusters. Then, a natural wonder is that given that all red crosses represent failures in S_1, why are they separated into two clusters?

We manually investigate the two red clusters in Figure 3(d), and find that the two clusters correspond to two different failing mechanism although they are all due to Fault 1 (Figure 2). We select a representative from each cluster (test cases 8 and 82 respectively), and explain how they fail differently from the same fault. Because slices in the same cluster are nearly identical, it does not matter which particular failure is chosen.

Figure 4 presents the data slice of case 8. The wrong value is observed at statement 134 in function `send_bits()`, which is called and passed with a faulty parameter at line 1033. The faulty parameter is produced by the data dependence chain of $1033 \xrightarrow{dd} 1032 \xrightarrow{dd} 1026 \xrightarrow{dd} 962$.

A further study of Fault 1 in Figure 2 reveals that the faulty branch at statement 686 produces a faulty `match_length`, which makes the control flow select the wrong branch at 707. This in turn results in `ct_tally()` being called by mistake at line 738. Inside this call, the array `l_buf` is polluted. Finally, when the execution tries to print a compressed block that is affected by `l_buf`, an incorrect output is observed.

Figure 5 presents the data slice of case 82. In this failing case, the wrong output is observed at the same source code location (statement 134) as case 8. However, the failure follows a completely different dependence path. At the function level, the dependence chain is `send_bits` $\xrightarrow{dd}$ `send_tree` $\xrightarrow{dd}$ `gen_codes` $\xrightarrow{dd}$ `gen_bitlen` $\xrightarrow{dd}$ `pqdownheap` $\xrightarrow{dd}$ `ct_tally`.

The explanation is that `ct_tally()` is mistakenly called at line 707 due to Fault 1. The function

`ct_tally()` calculates the frequencies of different trees, which are used to encode bytes in gzip. Because of Fault 1, the faulty frequency calculated by `ct_tally()` results in wrong trees being constructed, which are eventually dumped to the output by the function `send_tree()`, as part of the entire output.

Therefore, the case study with gzip clearly indicates that the same fault can fail the program in totally different ways, and that DS-PROXIMITY explicitly indexes failures with different failing mechanism apart. While this is intuitively an advantage, DS-PROXIMITY is nevertheless penalized by the Silhouette coefficient for it. This raises our wonder about whether the optimal indexing should index all failures due to the same fault together, or should only index failures with similar failure mechanism together. For some applications, like failure prioritization, the former is preferred; but for some others, like assigning failures to the appropriate developers, the latter is better. Our current metric (Section 2.2) follows the former belief, and hence penalizes DS-PROXIMITY on gzip. The Silhouette coefficient metric can also follow the latter belief, but human beings need to specify what failures exhibit the same failure mechanism. In this study, we stick with the former belief for consistency.

5 Discussion

In this section, we review related work, and discuss potential threats to validity of the experiment.

5.1 Related Work

Failing indexing, although not yet formally studied, has been a widely supported functionality in bug tracking systems [18]. A bug tracking system supports bug diagnosis and software evolution by keeping records of reported failures. Some bug tracking systems, like Bugzilla [1], are designed for manual failure reporting. Software developers or technically savvy people manually type in critical attributes of encountered failures. Typical attributes include, but are not limited to, the platform, failure stack trace, and the submitter-perceived severity. By storing the reported information into databases, failure indexing on the provided attributes is automatically supported. For example, one can easily retrieve all failures that manifest on FreeBSD and have a severity level of 5. However, such borrowed indexing capability from databases does not support automated failure prioritization and duplicate removal because root causes are usually not reported, and automatically inferring the root cause from the reported *static* failure data is extremely hard. In comparison, this paper, as well as previous studies [16, 19], investigates how to index program failure by program *dynamic* data.

On the other hand, some bug tracking systems aim at automated collection of program failures from production runs [2,3,14], which save users' hassles in providing technical details. Given that current systems have done a great job in indexing crashing failures, this paper investigates how to index noncrashing failures that will prevail in the future.

In this paper, we compare our dynamic slicing-based approach to existing techniques T-PROXIMITY [19] and R-PROXIMITY [16]. T-PROXIMITY is inspired by the preceding studies that suggest program failures can be found from a set of mostly passing executions through clustering execution profiles [8, 9]. In comparison, our approach indexes program failures through dynamic slices, which are more fault-relevant than the execution profile used by T-PROXIMITY. In comparison with R-PROXIMITY, our approach eliminates the need of passing executions, and is shown to achieve comparable result as R-PROXIMITY. Interestingly, similar to R-PROXIMITY, the dynamic slicing-based approach also falls into the fault localization-based framework [16], because dynamic slicing is also a fault localization technique. Our approach is better than R-PROXIMITY because dynamic slicing does not need any passing executions while the SOBER [15] algorithm leveraged by R-PROXIMITY does.

Recently, the importance of failure indexing is also recognized by computer system researchers [7, 21, 22]. Cohen et al. suggest that as computer systems become increasingly complex, indices of system states are helpful for system maintenance and malfunction diagnosis [7]. Basically, system statistics, such as the average CPU and memory usage, is treated as the signature of system states during a time interval. If a state is known faulty or will eventually lead to a faulty state, it is put into the index together with patches. In the future, when a similar state is encountered, corresponding patches can be automatically retrieved from the index. This approach is shown particularly effective in diagnosing performance problems [6], which are essentially noncrashing failures. Similar work is also seen on Windows platform, where snapshots of Windows registry are treated as signatures of system states. Some tools, such as STRIDER [22] and PeerPressure [21] have been invented, which leverage the signature indices to troubleshoot misconfigurations, which are another form of noncrashing failures. In comparison, our dynamic slicing-based approach focuses on indexing program failures, rather than indexing failures in a computer system, but the dynamic slicing idea can be extended to indexing system problems because intensive dependences are also involved in system problems [6].

Finally, this study also relates to dynamic program slicing. Dynamic slicing [4, 11] is a debugging technique that captures the executed statements that are involved in computation of a wrong value. Dynamic dicing [17] leverages multiple dynamic slices to reduce the fault candidate set.

The idea of dynamic dicing is to take away the statements that appear in the dynamic slices of correct values from a dynamic slice of some incorrect value. The goal of these techniques is to locate the root cause of a failure more precisely. Therefore, data slices may not be a good starting point for dicing because they often miss the root cause. In contrast, the proposed technique uses multiple dynamic slices for the purpose of failure indexing, where the capability of discriminate failures from different groups is more important than the fault localization effectiveness. Finally, to the best of our knowledge, this is the first attempt to study the effectiveness of various types of dynamic slices in failure indexing.

5.2 Threats to Validity

A number of threats to validity need to be considered for the experiment results. First, although the two faults gzip mimic realistic semantic bugs, they are nevertheless manually seeded. For this reason, case studies with real-world faults are needed in the future. However, as this paper aims at a *comparative* study between different indexing techniques, seeded faults may suffice. Second, hand-crafted test inputs, rather than operational traces from the wild, are used in this study. In general, traces from the wild could be more complicated. But as dynamic slicing has been shown effective in extracting fault-relevant information from long executions [23], we expect similar observations about failure indexing will be made. Finally, the experiment in this paper is evaluated with the metric proposed in Section 2.2. Although every effort has been exercised to keep it objective and reasonable, the metric is by no means the ultimate measure. Ultimately, all indexing techniques need to be subjected to real-world noncrashing failures, and let the end-users, *i.e.*, the developers, to judge the effectiveness.

6 Conclusion

In this study, we proposed a dynamic slicing-based approach to indexing noncrashing failures, an increasingly critical problem due to the dominance of semantic bugs in the future. The case study with gzip clearly demonstrated the advantages of our proposed approach. Specifically, our proposed approach is more effective than T-PROXIMITY, and does not rely on correct execution as R-PROXIMITY does. During this study, a few interesting observations have been made, which merit further study in the future.

References

[1] Bugzilla, http://www.bugzilla.org/.

[2] Description of the Dr. Watson for Windows tool, http://support.microsoft.com/kb/308538.

[3] Mozilla quality feedback agent, http://www.mozilla.org/quality/qfa.html.

[4] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, 1990.

[5] I. Borg and P. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, first edition, 1996.

[6] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, 2004.

[7] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, 2005.

[8] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, 2001.

[9] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *ESEC/FSE*, 2001.

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

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

[12] M. Levandowsky and D. Winter. Distance between sets. *Nature*, 234:34–35, Nov. 1971.

[13] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Having things changed now?: An empirical study of bug characteristics in modern open source software. In *ASID*, 2006.

[14] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, 2005.

[15] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, 2006.

[16] C. Liu and J. Han. Failure Proximity: A fault localization-based approach. In *FSE*, 2006.

[17] J. Lyle and M. Weiser. Automatic program bug address by program slicing. In *ICCA*, 1987.

[18] L. McLaughlin. Automated bug tracking: the promise and the pitfalls. *IEEE Software*, 21:100 – 103, 2004.

[19] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE*, 2003.

[20] P. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2006.

[21] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, 2004.

[22] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA*, 2003.

[23] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *FSE*, 2006.