Using Branch Frequency Spectra to Evaluate Operational Coverage

Marc Brünink
National University of Singapore
School of Computing
marc@u.nus.edu

David S. Rosenblum
National University of Singapore
School of Computing
david@comp.nus.edu.sg

***Abstract*—Coverage metrics try to quantify how well a software artifact is tested. High coverage numbers instill confidence in the software and might even be necessary to obtain certification. Unfortunately, achieving high coverage numbers does not imply high quality of the test suite. One shortcoming is that coverage metrics do not measure how well test suites cover systems in production.**

We look at coverage from an operational perspective. We evaluate test suite quality by comparing runs executed during testing with runs executed in production. Branch frequency spectra are employed to capture the behavior during runtime. Differences in the branch frequency spectra between field executions and testing runs indicate test suite deficiencies. This post-release test suite quality assurance mechanism can be used to (1) build confidence by pooling coverage information from many execution sites and (2) guide test suite augmentation in order to prepare the test suite for the next release cycle.

I. Introduction

Testing has at least two main objectives; (1) detecting failures before release often with an implicit goal of (2) instilling some confidence that the tested software artifact will expose only a limited number of additional failures after release.

Confidence is achieved through some software process often involving some form of coverage metric. For example, if the test suite has 100% statement coverage, we might be confident about the functional correctness of our program. Unfortunately, coverage metrics often make strong assumptions that are not always justified. For example, statement coverage assumes that no dependencies between basic blocks exist. Once all statements are covered, we do not care about how they got covered. Furthermore, coverage numbers are hard to interpret, especially in the common case that full coverage is not achieved. If we achieve 99.9% coverage, we simply do not know whether the residual 0.1% is critical to executions in deployment.

For instance, the test suite of `coreutils 8.25` achieves 87.9% line coverage for `sort`. This is the fourth highest coverage observed for 14 tools that we investigated (cf. Section IV, Table I). Thus, we might perceive some level of confidence in the testing process. However, when used in production, 42% of runs execute lines that were not covered during testing at all. This illustrates that confidence instilled through high coverage numbers is not always justified [12], [13].

During testing the software developer tries to instill confidence that the probability of failure for a piece of software in the field is low. One approach to testing (especially to performance testing) is to test common workload scenarios. The software is tested with a workload that is supposed to resemble either corner cases or a realistic workload expected at a common user site. Unfortunately, capturing a representative test workload is challenging [10]. Additionally, it is often not possible to test a complete workload due to cost and possibly complexity [23]; instead an approximation is used. Finally, workloads evolve, making it necessary to re-collect and update test workloads constantly.

As a result, workloads utilized during testing often enough do not resemble real workload used in deployment. This enables problems to go unnoticed and escape to deployment. For example, in one study the authors found that 35% of all performance bugs escaped detection due to wrong workload assumptions [18].

In summary, getting testing right is hard. A software developer wants to ensure that real executions in production are covered during testing. It would be beneficial to be able to verify that the operational distribution is close to the testing distribution. This would increase the overall confidence that can be put in the testing process and the software artifact.

A. Contributions

We introduce a new approach that uses operational data to detect executions that behave differently from testing. We make the following contributions:

- We present a coverage metric that captures how well behaviour of field executions correlates with runs observed during testing.
- Our evaluation illustrates that the technique can be used to detect risky executions in the `coreutils 8.25` tool set.
- Most of these risky behaviours can be detected even if the data is subjected to low sampling rates.
- Our technique includes a debugging approach that identifies branch sites that contribute most to deviant behaviour.
- All tools and supplemental data are available at the accompanying website.[1]

[1] http://www.comp.nus.edu.sg/~a0091945/apsec2017/

The paper is structured as follows. In Section II we introduce branch frequency spectra and discuss their detection capabilities. Next, we present how branch frequency spectra can be used to evaluate operational coverage (Section III). We evaluate the introduced technique in Section IV. Section V discusses related work and we conclude in Section VI.

II. Branch Frequency Spectra

Stakeholders are interested in how well a test suite prepared a software artifact for deployment. They want to be confident that executions post-release are covered by testing. We tackle this challenge by observing executions in the field and correlating their operational profiles with profiles collected during testing. Field executions that behave differently from anything seen during testing indicate potential deficiencies in the test suite.

We capture the behavior of a software artifact in a branch frequencies spectrum.

Definition 1: A branch frequency spectrum is a n-dimensional vector

$$s = (f_1, f_2, ..., f_n)$$

where n denotes the number of conditional branches in the software and each element f_i can be computed as:

$$f_i = \frac{\#\ branch\ i\ taken}{\#\ branch\ i\ reached}$$

In case a branch is never reached, the branch frequency is defined as 0.

Branch frequencies have been used before, for example as transition probabilities in Markov models [6]. Branch frequency spectra are similar to branch counts spectra. Branch count spectra have been shown to be a good indicator for functional failures [15]. In contrast, branch traversal profiles might be not be very effective for well tested applications [7].

In the following, we illustrate how differences in a single conditional branch b_i can be detected. Let us assume branch b_i is taken regularly in deployment runs with frequency $f_i >> 0$. However, it is never taken during testing, i.e. the frequency during testing is 0. Furthermore, all the frequencies for all other branches are the same. Then the resulting frequency vectors are $p = (f_1, ..., f_i, ..., f_n)$ for executions observed in production and $t = (f_1, ..., f_{i-1}, 0, ..., f_n)$ for testing.

We use the Manhattan distance as a distance metric since it performs better than Euclidean distance in high dimensional spaces [1]. The Manhattan distance is the sum of the element wise difference of two vectors p and t, i.e. $d(p,t) = \|p-t\|_1 = \sum_{i=1}^{n} |p_i - t_i|$.

Since all frequencies except for branch i are the same, the distance between the production and the testing vector is $\|p-t\|_1 = \sum_{j=1}^{n} |p_j - t_j| = |p_i - t_i| = p_i = f_i$.

It follows that the more frequent branch b_i is taken in deployment (f_i increases), the larger is the distance between the deployment run p and the testing run t.

Listing 1 shows an example that illustrates the potential of branch frequency spectra collected at production sites. It shows

```
1  static void
2  sequential_sort(struct line *restrict lines,
3                  size_t nlines,
4                  struct line *restrict temp,
5                  bool to_temp) {
6    if (nlines == 2) {
7      int swap = (0 < compare (&lines[-1],
8                               &lines[-2]));
9      if (to_temp) {...}
10     else if (swap) {
11       temp[-1] = lines[-1];
12       lines[-1] = lines[-2];
13       lines[-2] = temp[-1];
14     }
15   } else {...}
16 }
```

Listing 1. Excerpt from `sort.c` from `coreutils 8.25`.

an excerpt from the `sort` tool from `coreutils 8.25`. The tool arranges input lines according to sorting criterions chosen by the user. The function `sequential_sort` implements sorting by divide and conquer. It splits the input into two subproblems and calls itself recursively until the number of input lines is two (line 6). For brevity, the implementation of the splitting and the recursive call is not shown (line 15). Once the number of input lines is two, `sequential_sort` swaps the remaining two lines if necessary. If sorting is performed in place (`to_temp` is false), swapping is performed at lines 11 to 13, otherwise at line 9.

The frequency of the branch at line 10 depends strongly on the input to the algorithm. We could reach full coverage for many coverage metrics with a low quality test suite. For example, let us assume we test `sequential_sort` only with input that is already sorted except for a single position. With this input, we can easily achieve full statement and branch coverage. Achieving these high coverage number might give us a false sense of security. Confidence in the quality of the test suite is only justified if real world workload satisfies the assumptions made during testing; specifically, the assumption that the input is already sorted except for a single position.

A. Completeness of Branch Frequency Spectra

Branch frequency spectra contain frequencies and exclude absolute execution counts. However, the execution counts are indirectly included in a branch frequency spectrum. In order to execute a branch multiple times, we need to have a cycle in the call graph. Except for infinite cycles, i.e., loops that never terminate, all cycles have an exit edge that is guarded by a conditional branch. For instance, if the iteration count of the loop in Figure 1 increases, the fraction of how often the exit edge is taken upon reaching b_3 decreases. This fraction of b_3 is part of the branch frequency spectrum. Compared to branch count spectra, the information in branch frequency spectra is more localized. Loop iterations are only reflected at the branch causing the cycle and they do not permeate to all branches along the path.

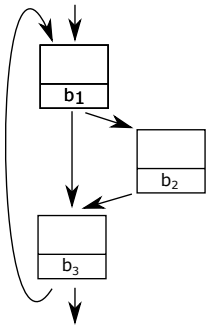

Fig. 1. Branch frequency spectra capture the information how often statements are executed in the conditional branch that terminates the cycle, in this example b_3. b_2 is an unconditional branch and thus is not part of the branch frequency spectrum.

Branch frequency spectra do not contain any ordering information. Thus, if we just look at a branch frequency vector, we cannot determine in which order a specific set of branches was executed. Furthermore, the values are aggregated across the complete executions. Since branch-count spectra make the same assumptions and are sufficient to detect most failures [15], we expect that the loss of information is non-critical. We evaluate how well branch frequency spectra can detect suspicious executions in Section IV.

B. Sampling Without Time-Bias

We can minimize overhead by reducing the amount of data collected. For example, the Dapper system collects function calls for common libraries only; however, to achieve acceptable levels of overhead, it also uses sampling [30].

One main benefit of sampling is that the overhead can be adjusted easily by manipulating the sampling rate. However, sampling also suffers from some severe problems.

Sampling introduced uncertainty in the data. Since we only sample the data, the accuracy of the data strongly depends on the number of samples. We need to gather a large enough data set to compensate for noise and ensure that the collected samples resembles the actual data. Furthermore, detecting brief transient events with sampling is hard; sampling might just miss them.

Another important problem of sampling is rooted in the challenges of achieving truly independent sampling with constant probability for all sampling points. Statistical models are used to analyse collected data. Statistical models often depend on independently drawn samples. Unfortunately, sampling every sample point independently and with the same probability is hard.

Sampling can be triggered either based on counters or based on wall-clock time [2]. Even though counter based sampling has many benefits, implementing it efficiently is hard (especially if the counters are implemented in software). It requires minimal instrumentation that is always-on. Even though this instrumentation does not collect any data, it is reported to lead to an average overhead of 6% [2].

In contrast, time-based sampling does not require additional instrumentation. In fact, if it is triggered during existing interrupts (e.g. scheduling), the overhead can be minimal [32]. However, time-based sampling will oversample some branches due to the-constant and non-uniform execution times of basic blocks. This time-bias can affect the accuracy if we collect branch count spectra. In contrast, branch frequency spectra are resilient towards time-bias.

III. Grouping Branch Frequency Spectra To Avoid Test Suite Bloat

Each execution yields a branch frequency vector, capturing how the software behaved. We use branch frequency vectors to calculate coverage. Coverage is the percentage of observed in-field executions that are covered by a test run. We call this coverage *operational coverage*. An in-field run is covered if it is similar to a test execution. A key challenge is to define what *similar* means exactly.

It is always possible to define an accurate similarity metric; two executions are similar if, and only if, the branch frequency vectors for both executions are exactly the same. Using this metric implies that the test suite has to contain at least as many test cases as unique branch frequency vectors observable in deployment to achieve 100% coverage. In such a scenario we can be confident that testing is comprehensive and complete.

Unfortunately, in general, the input space and with it the branch frequency space contains a very large (if not infinite) number of elements. Achieving high coverage would lead to an explosion in the number of required test cases.

To solve this problem, we group executions. Each group is covered if it contains at least one test run. We group executions using hierarchical agglomerative clustering [17]. Initially, hierarchical agglomerative clustering puts each observation in a dedicated cluster. Next, it merges clusters iteratively until only a single cluster remains.

An important parameter of hierarchical agglomerative clustering is the method that is used to decide which clusters should be merged. We use the single linkage criterion. In each step the single linkage method merges the two clusters with the smallest distance to each other.

Definition 2: The distance between two clusters C_1 and C_2, denoted as $d(C_1, C_2)$, is the minimal distance between any two members of the two clusters [17]. It is defined as follows:

$$d(C_1, C_2) = \min_{c_1 \in C_1, c_2 \in C_2} (d(c_1, c_2)) = \min_{c_1 \in C_1, c_2 \in C_2} (\|c_1, c_2\|_1) \quad (1)$$

Figure 2 illustrates hierarchical agglomerative clustering using single linkage. In the first step single linkage merges the data points O_1 and O_2. Next, it progresses by iteratively merging the points T_1, O_3, and O_4. In a last step point O_5 is merged into the group. Once O_5 is merged, only one cluster remains containing all elements; the hierarchical agglomerative clustering algorithm terminates. Note how O_1 and O_4 are merged into a cluster before O_5, even though the distance between O_1 and O_5 is significantly smaller than the distance O_1 to O_4.

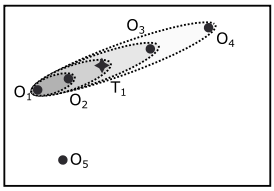

Fig. 2. A 2-dimensional space with 5 operational observations, O_1 to O_5, and one data point observed during testing, T_1. Single linkage merges O_1 and O_2 first and then slowly progresses towards O_5. Single linkage tends to form elongated clusters, chaining data points together.

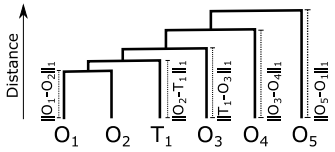

Fig. 3. The resulting hierarchy plotted as a dendrogram after the data points in Figure 2 are merged using single linkage. The annotations reflect the distance that needs to be bridged to merge two clusters. E.g. to merge O_5 in the last step we need to span the distance $\|O_5 - O_1\|_1$ between O_5 and its closest neighbor, O_1.

Compared to other merging methods, single linkage has a tendency to chain data points together. In our context, these chains are desirable. Let us assume T_1 is an observation resulting from a test case. Since O_2 is close to T_1, O_2 is covered. But if O_2 is covered and since the distance from O_2 to O_1 is small, then O_1 is also covered. We define coverage as *absolute* and *transitive*. We reflect this by using the single linkage merging criterion.

An alternative to hierarchical agglomerative clustering would be to simply pick a threshold value θ. Then, all executions with a pairwise distance smaller than θ form a group. Unlike hierarchical agglomerative clustering, this method does not chain executions together. Furthermore, deciding on a suitable threshold value is hard. Whether a value is a good pick depends strongly on the collected data. Making manual decisions is laborious, error prone and subject to bias. In contrast, hierarchical agglomerative clustering does not require any fixed threshold value.

A. No Distance Thresholds

In order to decide which runs are covered, we cluster all executions, tests and deployment runs, in one step. The clustering will provide us with a hierarchy of data points. For example, if we cluster the data points shown in Figure 2 using single linkage, we obtain the hierarchy presented in Figure 3.

We denote the set of clusters that exist at distance d with $\mathcal{C}_d$. For example, the hierarchy in Figure 3 changes the grouping of data points at 5 different distances. The resulting clusters are:

$$\begin{aligned}
\mathcal{C}_0 &= \{\{O_1\},\{O_2\},\{T_1\},\{O_3\},\{O_4\},\{O_5\}\} \\
\mathcal{C}_{\|O_1-O_2\|_1} &= \{\{O_1,O_2\},\{T_1\},\{O_3\},\{O_4\},\{O_5\}\} \\
\mathcal{C}_{\|O_2-T_1\|_1} &= \{\{O_1,O_2,T_1\},\{O_3\},\{O_4\},\{O_5\}\} \\
&\vdots \quad \vdots \\
\mathcal{C}_{\|O_5-O_1\|_1} &= \{\{O_1,O_2,T_1,O_3,O_4,O_5\}\}
\end{aligned}$$

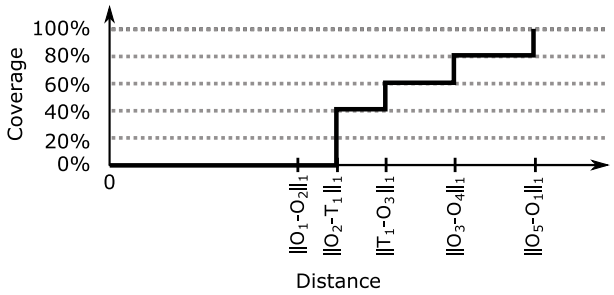

Fig. 4. A coverage graph for the data points from Figure 2. Coverage stays 0 until $\|O_2 - T_1\|_1$ because until then no cluster contains a test run. At $\|O_2 - T_1\|_1$ T_1 is merged into $\{O_1, O_2\}$ leading to a coverage of $\frac{2}{5} = 40\%$. Coverage continues to increase each time a new operational observation is merged into the cluster containing T_1.

Definition 3: Let T be the set of observations made during testing. The set of covered clusters at distance d is defined as:

$$covered(\mathcal{C}_d) = \{C \in \mathcal{C}_d : C \cap T \neq \emptyset\}$$

Definition 4: Let O be the set of observations made during operation. Then coverage at distance d is defined as:

$$Coverage_d = \sum_{C \in covered(\mathcal{C}_d)} \frac{|O \cap C|}{|O|} \tag{2}$$

Thus, coverage at a distance d is the fraction of operational observations that are members of a covered cluster. Note that if $T \neq \emptyset$ (there is at least one test run), coverage always reaches 100%, because ultimately all data points are merged into a single cluster, i.e. $|\mathcal{C}_\infty| = 1$.

Using Equation 2 we can calculate the coverage for any distance. A stakeholder just needs to select an appropriate distance threshold. However, picking a single threshold value is hard and subject to bias. Instead, of fixing a single threshold distance and providing one coverage number, we strongly recommend to use coverage graphs. These graphs plot the achieved coverage for all distances. For, example Figure 4 shows the coverage graph for our running example of the data points initially shown in Figure 2.

B. Identifying Root Causes For Large Distances

A software developer might not only be interested in pure coverage numbers. Instead, she might want to understand why a specific set of executions is so different from all test runs.

We support the developer in this task. We rank edges according to how strongly they influence the resulting coverage graph. First, the software developer selects the cluster C of observations she is interested in. Probably the developer will be most interested in clusters that require a large distance to be covered by a test run. Next, she uses the function RANKEDGES, described in Algorithm 1, to sort edges according to importance. She provides the parameters E (the set of edges), C (the cluster she is interested in), O (the set of operational observations), and T (the set of observations made during testing).

Algorithm 1 Ranking edges by Effect on Test Case Distance.

```
1:  function RankEdges(E, C, O, T)
2:      rank ← []
3:      for e ∈ E do
4:          O_tmp ← RemoveEdge(e, C ∩ O)
5:          T_tmp ← RemoveEdge(e, T)
6:          d_min ← ∞
7:          for o ∈ O_tmp, t ∈ T_tmp do
8:              if ‖o − t‖_1 < d_min then
9:                  d_min = ‖o − t‖_1
10:         rank.append((e, d_min))
11:     return ReverseSortByDistance(rank)
```

RankEdges iterates over all edges one by one. In each step it removes the current edge from $C \cap O$ and T (lines 4 and 5). Next, it finds the operational observation that has a minimal distance to any test case (lines 6 to 9). Finally, the algorithm reverse sorts the edges by minimal test case distance at line 11. This effectively ranks the edges. The edge that, once it is removed, results in the minimal distance to any test case is assigned the highest rank.

Note that we find the minimal distance to a test case in lines 6 to 9. In general, after removing the edges in lines 4 and 5, we would have to re-calculate the complete clustering. Then we could measure the precise effect on cluster distance. However, clustering is a relatively expensive calculation. We approximate it by using the minimal distance to any test case. This approximation is faster than clustering and resulted in very good insights in our evaluation.

IV. Evaluation

In our evaluation, we will tackle three main questions.

- **RQ 1**: Is the technique able to detect behavioural differences? Are branch frequencies sufficient to discover behaviours that are different from everything seen during testing?
- **RQ 2**: How does sampling affect the model quality and in particular, the distances to the test cases?
- **RQ 3**: How do the detection capabilities of our technique compare to other approaches?

To evaluate our approach, we analyzed `coreutils 8.25`. We instrumented all `coreutils` tools using LLVM. At each conditional branch, we log the branch outcome, i.e to which target the branch jumped. In a post-processing step we calculate the branch frequencies and use them in our analysis. We run all experiments on a machine equipped with a Intel Core i7-2600 processor with 3.4 GHz and 8 GB of RAM running Linux kernel 3.13.0.

We gathered two sets of data. First, we collected data for all test cases. We ran the test suite of `coreutils` in the default settings. Second, we used the instrumented tools over a period of multiple weeks to achieve our daily work.

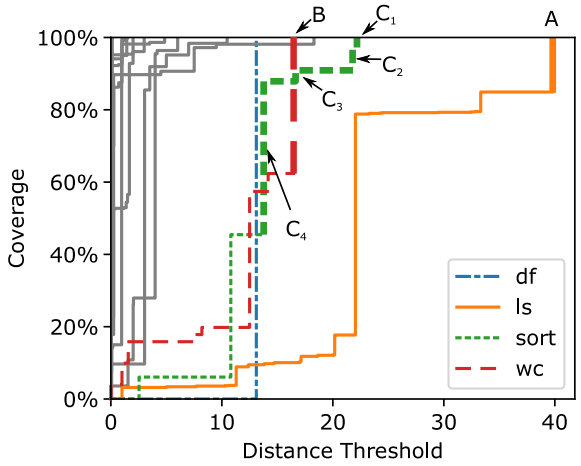

Fig. 5. Coverage graph for `coreutils 8.25`. We excluded all tools that have less than 20 executions during testing (`basename`, `chown`, `date`, `dircolors`, `dirname`, `groups`, `nice`, `readlink`, `sleep`, `tee`, `uname`) or less than 20 executions during our data collection period (`chmod`, `ln`, `mktemp`, `uniq`). We label only `df`, `ls`, `sort`, `wc` for clarity.

A. (RQ 1) *Can we use branch frequencies to discover behaviors that are different from testing?*

Figure 5 shows the distances needed to merge at least a single test case into an existing cluster.

For example, for `ls` only 20% of the executions have a test case in their cluster if the threshold distance is 20. As soon as the threshold distance is slightly larger than 20, a large portion of the executions are covered. This is caused by a large cluster containing about 60% of the executions; this cluster contains a test case only if the threshold for the distance cutoff is larger than 20.

In the following, we investigate the four `coreutils` tools `ls`, `wc`, `sort`, and `df` one by one. These four tools show a large distance between field and testing runs (Figure 5).

During the data collection period, we gathered 1035 executions of **`ls`**. The `ls` tool lists the content of a directory. Figure 5 shows that we need a fairly large distance to ensure that each of our executions of `ls` is a member of a cluster that includes at least one test case.

To evaluate why the distance between real world runs and test cases is so large, we focus on the executions that are members of the cluster with the largest distance, i.e. the cluster marked A in Figure 5, containing approximately 15% of all executions. We apply Algorithm 1 to cluster A.

Algorithm 1 returns a list of ranked edges. In the list, 24 edges share rank #1. Removal of any one of these 24 edges results in the same reduction in distance to a test case, making them all good candidates for manual inspection.

Looking at the source code, it is apparent that many of these edges are related to coloring the output. In fact, the source code lines of 10 out of the 24 edges match the term ”color”.

In total the compilation unit has 678 edges. 92 of these have the term ”color” in their source code line. The probability of selecting 24 edges randomly from 678 edges and having 10

or more edges matching the term "color" is $p = 5.1 * 10^{-4}$. Thus, we can be confident that the high occurrence of the term "color" is not caused by random chance.

Observation 1: Many of our executions of `ls` are fairly distant from any test case in the test suite. This is surprising and might impact the confidence we put in the test suite. The main cause for the large distance is the prevalent usage of colorization in our deployment. During execution of the test suite we log 8005 runs of `ls`. Out of these, 60 print colors in the output. The underrepresentation of colorization in the test suite might indicate missed testing opportunities.

Next, we look at the tool **`wc`**. `wc` counts the number of lines, words, and bytes in an input file. We collected 101 real world executions of `wc` during our data collection period. Our executions of `wc` show a large distance to the runs executed during testing. For example, about 80% of our executions need a distance threshold larger than 10 to be in a cluster that contains a test run (cf. Figure 5). To investigate why this large distance exists, we follow the same approach we took during the analysis of `ls`. We look at the cluster that is merged last with a test run (B in Figure 5). Next, we invoke Algorithm 1 to find the edges that have the largest impact on the distance to a test case.

13 edges are ranked #1 and have the largest influence on the distance. Removing any of these 13 edges reduces the distance threshold needed to merge a test run into the cluster by the same amount. We inspected the edges manually. 7 of the 13 edges are related to multibyte encoding, i.e. encodings that use multiple bytes to encode a single character. Thus, multibyte encoding seems to be an important factor. In fact, 95 of our 101 field runs use multibyte encoding. In contrast, only 48 of 1812 executions during testing use multibyte encoding.

Further investigation revealed that the test run that causes cluster B to be covered is using multibyte encoding as well. However, the origin of this test run is not a test case for `wc`; instead, it is a test case for `ls`.[2] This test case of `ls` is the only place in the test suite that executes `wc` with multibyte encoding.

Observation 2: Most of our executions of `wc` use multibyte encoding. There is no dedicated test case for `wc` that uses multibyte encoding. As a result the distance between our executions and the test runs is large. Even though `wc` relies on gnulib to handle key functionality of multibyte encoding some test cases that explicitly verify that `wc` works as expected with multibyte encoding seems warranted.

During our data collection period we collected 33 executions of **`sort`**. This tool sorts input files by line using different sorting criteria. Our executions show a fairly large distance towards the test cases (Figure 5). The cluster with the largest distance (C_1 in Figure 5) is merged last with a test run. This cluster contains only a single run. Further analysis revealed that this run was interrupted by a signal before termination. We suspect a Ctrl+C manual abort. Since it was interrupted, its operational profile is very different from the test runs causing the large distance. The second cluster C_2 contains 2 runs. These 2 runs executed exactly the same paths. Looking at the edges that influence the distances most, we isolate 7 edges. A brief inspection of these 7 edges was inconclusive at first. However, we noticed that one of the 7 edges is a branch that checks whether 2 lines of input need to be swapped during sorting. The branch is never executed in the 2 runs. This implies that the input for these 2 runs was already sorted. Deeper investigation revealed that this was indeed the case, explaining the distance to the test cases. The next cluster C_3 is similar to C_1 insofar as the single execution in the cluster was terminated prematurely with a signal. Since cluster C_1, C_2, and C_3 are abnormal executions, we also had a look at cluster C_4.

For cluster C_4 Algorithm 1 returns 5 edges with rank one. 3 of the 5 edges are directly related to locales. In fact, by extending our selection cutoff value for the edges only slightly, we find 2 more edges that reduce the distance by virtually the same amount.[3] One of the 2 is related to our environment variables and is of no further help. The second one is the branch at `sort.c:2725`; this branch is also related to locale, to be precise, it is related to collation; it determines whether an expensive call to `xmemcoll0` is made. This call is made in all executions in cluster C_4. It is never made in any of the runs during testing.

Observation 3: Our tool correctly isolates four runs of `sort` that behave abnormal. Two were terminated prematurely by a signal. The other two executed on sorted input. Additionally, for cluster C_4 we find that our executions are different due to locale and we detect a branch that is executed always in our executions but never during testing.

As a last example, we look at the tool **`df`**. This tool reports available space and other useful information for all file systems of a machine. Our 58 executions of `df` show a high distance to the runs of the test cases. This was unexpected. We expected our executions of `df` to be closer to at least one of the 100 runs executed during testing. The large distance is caused by two factors.

One reason is the similarity of all our 58 runs. This is not too surprising, considering our limited usage of `df`. We use it to check our free disk space occasionally. We always invoked `df -h` and we did not use any other runtime options. Since all runs are very similar, the observed frequency vectors are not diverse and quite homogeneous. As a result, intra-cluster distance is small and our hierarchical agglomerative clustering cannot cover larger areas by chaining data points together (cf. Section III).

Observation 4: The main reason for the large distance of `df` is our usage of the `-h` runtime option. This leads to the activation of `HUMAN_MODE`. No test case activates this mode. As a result, every single of our observed executions executes statements and conditions that are not covered during testing. Thus, the large distance reported for `df` is justified

[2]The test case `abmon_align.sh` uses `wc -L` to verify the output of `ls`.

[3]The reduction is $1 - 10^{15}$ of maximum reduction.

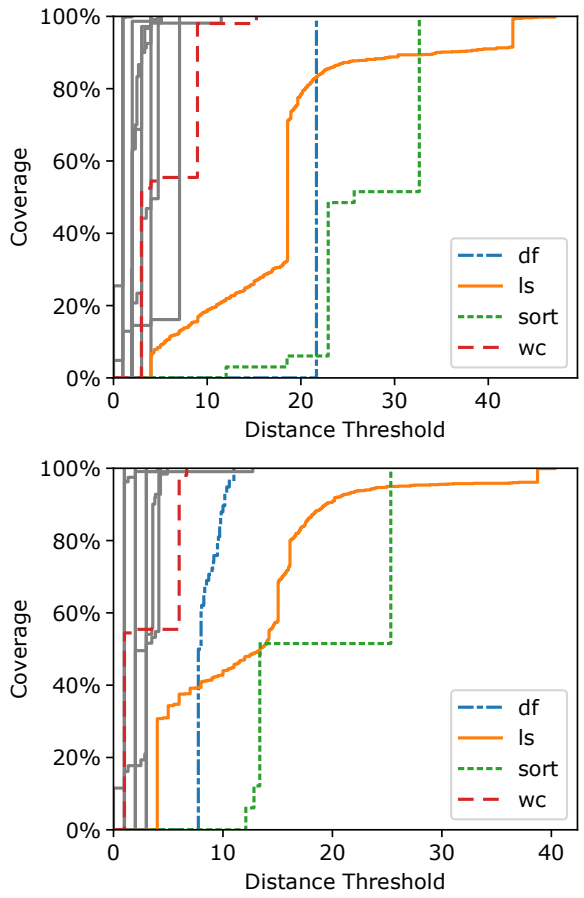

Fig. 6. Coverage graph of covered executions with a sampling rate of 5% (top) and 1% (bottom).

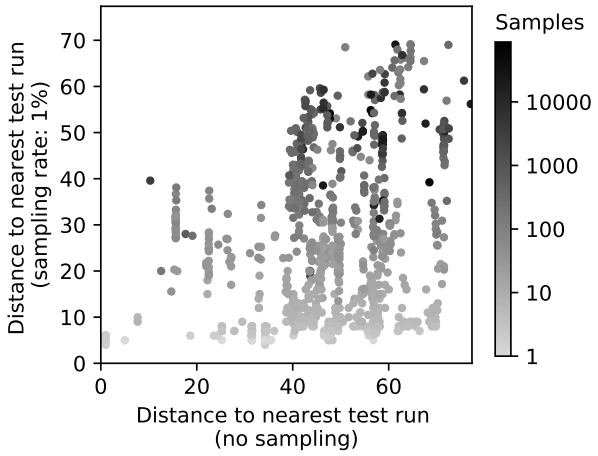

Fig. 7. Distance-distance graph for `ls`. Each data point is an execution of `ls`. We plot the distance to the nearest test run with no sampling (x-axis) versus the distance to the nearest test run with a sampling rate of 1% (y-axis). If the distance would not be affected by sampling, we would expect a straight line. The color indicates how many samples are drawn during sampling with a rate of 1%.

and correctly reflects the potential risk involved in executing uncovered statements in our deployment.

B. (RQ 2) *How does sampling affect the distance measures?*

We sample our data to investigate the effect it has on the model quality. We pick a sampling rate r. To allow for randomness in the sampling process, we capture our first sample at a randomly chosen offset picked from the closed interval $[0, \frac{1}{r}]$. Starting from this offset, we take our next sample at distance d. The sampling distance d between one sample and the next follows a normal distribution with mean $\frac{1}{r}$ and standard deviation $\sigma = 0.1 * \frac{1}{r}$.

Not surprisingly, sampling changes the coverage graphs (Figure 6). Sampling reduces the information content in the collected data. Especially with low sampling rates, a majority of the executed edges are not sampled.

With a 5% sampling rate, we are able to detect that `df`, `ls`, and `sort` behave very different from all our test runs (Figure 6, top). However, the sampled data for `wc` does not allow such a conclusion.

If we decrease the sampling rate to 1%, the data degenerates more (Figure 6, bottom). Due to decreased sampling fewer samples are drawn. The median number of executed edges for `df` is only 2647 (mean: 2692.6). Thus, with a sampling rate of 1%, we expect less than 27 samples in half of the executions.

Observation 5: With decreasing sampling rate, it becomes harder to conclude that an execution behaves different from test runs due to the small number of samples. Short-lived executions, e.g. `df`, would benefit from sampling rates larger than 1%.

To evaluate the effect sampling has on the distance to the closest test run, we explore in depth how `ls` behaves under a sampling rate of 1%. Figure 7 shows a distance-distance graph. Each data point represents one of our 1035 field executions of `ls`. The x- and the y-axis both represent the distance from a single execution to the closest test case. For the x-axis, we do not sample and use the complete path information to create the branch frequency profile. For the y-axis, we sample the path before we create the profile.

If sampling would not affect the distance, the distance would not change and we would see a straight line. It is clear that sampling has a profound effect on the distance to the closest test run. Across the whole range of distances observed without sampling (x-axis), we can find some executions that decline to distances below 10 once sampling is used (i.e. they have low y-values).

To investigate why this degradation occurs, we examine the number of samples we took for each data point. Executions with a low number of drawn samples are marked grey in Figure 7. These executions did not execute many edges in the first place (and probably ran for a short period of time only). Without sampling, their distances span the whole range (x-axis). However, once we use sampling, their distance consistently decreases to very low levels (y-axis).

The observation that short-lived executions degrade once they are sampled is not unexpected. For any sampling technique, the sampled vectors always approach 0 with a decreas-

TABLE I
COMPARISON TO LINE AND RESIDUAL TEST COVERAGE.

Name	gcov Line Coverage	No. of Exec.	Residual Test Cov.	Mean Dist.	Median Dist.	Max. Dist.
df	84.7%	58	3 (1)	13.1	13.1	13.1
ls	79.4%	1035	41 (12)	23.8	22.0	39.8
sort	87.9%	33	1 (0)	12.7	13.8	22.2
wc	85.4%	101	6 (1)	12.1	12.5	16.5
cat	70.5%	220	7 (3)	0.2	0	4.8
cut	93.7%	111	1 (0)	3.4	4.0	6.0
du	78.9%	62	0 (0)	3.1	3.0	10.5
expr	73.3%	30	0 (0)	0.1	0.1	0.1
head	76.0%	120	0 (0)	0.8	0.2	1.3
mkdir	75.5%	564	1 (0)	0.1	0	3.0
mv	91.6%	656	0 (0)	0.9	0.3	2.0
rm	91.2%	373	1 (0)	0.1	0	1.5
tail	66.4%	107	6 (2)	1.1	0.3	18.3
tr	84.8%	80	0 (0)	0.2	0	1.4

ing number of samples. In the extreme case that no samples are drawn at all, all sampled vectors are exactly 0.

If a sampled vector approaches 0 with decreasing number of samples, then it follows that its distance approaches the distance between 0 and the test run closest to 0. We can observe this behavior in Figure 7. For data points with a very small number of samples, the distance bottoms out at about 5.

Executions with a larger number of samples behave more stable when confronted with sampling. They tend to be much closer to the ideal straight line.

Observation 6: The distances for runs that execute short degrade when confronted with sampling due to the minimal number of samples. In contrast, executions that run longer are resilient when confronted with sampling. Large distances tend to stay large even when we sample only 1% of the original data.

C. (RQ 3) *How do the detection capabilities of our technique compare to other approaches?*

Coverage metrics are used to instill confidence in the software artifact. The assumption is that high coverage numbers correlate with a low number of failures in deployment. In contrast, we build confidence by comparing runs in deployment with runs observed during testing. Executions that behave vastly different from anything we saw during testing indicate missed testing opportunities.

We compare our approach with line coverage and Residual Test Coverage Monitoring [27] (Table I). We report numbers for all `coreutils` tools except tools that executed less than 20 times during testing or during our data collection period. `gcov` instrumentation relies on `gcc`. All other measurement make use of LLVM and `clang`. We disabled all compiler optimizations.

We report the standard `gcov` line coverage achieved during testing in the first column. The second column shows the number of executions observed during data collection for reference. The third column contains the total number of unique basic blocks that were not covered during testing, but executed in at least one of our deployment runs. Care should be taken to not put too much importance in the absolute values. Observing three uncovered basic blocks is not necessarily worse than observing only one. The small difference might be caused by unoptimized code; without optimization, LLVM produces many basic blocks, some of them chained together with unconditional branches. If one block is executed, all blocks in the chain are executed. To put the numbers into perspective, we also report the number of uncovered basic blocks that terminate with a conditional branch (in parentheses). The last three columns report the mean, median, and maximum distance to a test case observed in all deployment runs.

First, we notice that the two tools with the lowest line coverage (`tail` and `cat`) also have a large number of uncovered, but executed basic blocks (6 and 7, respectively). However, out of the three tools that achieve a line coverage of more than 90%, two also execute an uncovered block during production.

Observation 7: There is no clear correlation between line coverage and number of uncovered blocks executed in production.

Next, we notice that `ls` has 41 uncovered basic blocks executed during deployment. This is in line with our analysis in Section IV-A in which we flagged `ls` as a potentially problematic tool. `df` and `wc` also show an increased count of uncovered basic blocks. Potentially, we could use Residual Test Coverage Monitoring to flag `ls`, `wc`, and possibly `df`.

Considering the fact that 9 out of 14 tools execute at least one uncovered basic block, the numbers for `sort` are less prominent. An extended Residual Test Coverage Monitoring could struggle to report `sort` without triggering too many alarms. However, our approach reports `sort` because in our deployment it behaves very different from the test cases due to our locale setting.

Finally, `cat` and `tail` have counts that are relatively large. Our approach did not report them because the distance to test cases is low.

Observation 8: Our approach and Residual Test Coverage Monitoring have different strengths. We believe they would augment each other well.

V. RELATED WORK

Execution spectra are often used with a focus on functional failures, either to detect failed executions [8], [9], [14] or to locate bugs, e.g. [7], [20], [24].

Path profiling has been used for bug isolation and performance analysis [3], [4], [7]. While path profiling contains a wealth of information, it has been shown that the more compact branch count spectra are sufficient to detect many functional failures [15]. Our approach uses the related branch frequency spectra to assess the quality of a test suite by calculating a coverage metric that compares production executions with runs observed during testing. We use hierarchical agglomerative clustering to group executions.

Besides hierarchical agglomerative clustering many other clustering techniques exist and can be used in the context of software engineering. For example, KMeans clustering has been applied to reduce dimensionality [21] or to group similar behavior [16]. Clustering can also be used to draw stratified random samples from a large population of executions. These samples can then be used to estimate reliability of the software [28], [29].

Cluster filtering partitions executions spectra collected on test cases. These partitions can be sampled to select a subpopulation for further inspection. [8], [31]. Our technique creates clusters during the calculation of the coverage. These clusters can be used in a similar way to (1) filter test cases during regression testing while ensuring that all operational executions are still covered, and (2) to automatically augment a test suite by filtering generated test case guided by the goal of reducing the distance between field executions and test cases. It should be noted that high coverage numbers do not imply good failure finding capabilities [12]. Instead of maximizing bug finding abilities, the goal of the operational coverage metric is to instill confidence by ensuring that at least a subset of test cases resemble the majority of real world executions.

Our approach relies on execution data collected in the field. Existing infrastructures, e.g. [22], [26], can be adapted to manage the collection process for large populations. Software tomography collects branch coverage information and splits the data collection tasks across an array of software instances to distribute the load and minimize overhead incurred at each site [5]. Branch frequency spectra contain more information than branch coverage spectra. Data collection for branch frequency spectra can be separated across multiple machines if all executions are relatively homogeneous, e.g. multiple nodes in a data center running the same application with comparable workload.

Branch frequencies have been used as transition probabilities in Markov models to classify software behavior [6]. The resulting classifier detect new unknown behavior. This detection capability can be used on test cases generated by a test case generator. Then, the rate at which new unknown behavior is detected can serve as a proxy measure for test suite quality [6]. In contrast, we compare test runs with operational runs. We calculate a coverage metric that reflects how many of the observed executions in deployment are covered by a test case. In case the coverage is unsatisfactory, the collected data can be used to guide the creation of new test cases that reflect real world usage better [11].

Our technique helps a software developer to understand why a specific set of executions behave very different from test cases by ranking edges by effect on test execution distance. It would be worthwhile to combine this approach with existing visualization techniques, e.g. [19], [25].

VI. Conclusion

This paper introduces *operational coverage* as a post-release test quality assurance mechanism. The technique compares branch frequency spectra from field runs with spectra gathered during testing. Using these spectra, the approach is able to detect behavioral difference between executions. Coverage graphs give intuitive insight into the achieved coverage, helping developers to focus their attention on the field executions with the largest distance to test runs.

Coverage results obtained from the installation base can be communicated to potential new users to increase their confidence in the software artifact before installation. Additionally, low operational coverage indicates deficiencies in the test suite. Using this feedback to guide test suite augmentation to ensure field executions are covered by test cases would prepare a test suite for the next release cycle.

References

[1] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional spaces. In *Proceedings of the 8th International Conference on Database Theory*, ICDT '01, pages 420–434, London, UK, UK, 2001. Springer-Verlag.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[4] M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 130–140, Washington, DC, USA, 2005. IEEE Computer Society.

[5] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proc, of the 2002 workshop on Program analysis for software tools and engineering*, PASTE '02, pages 2–9, New York, NY, USA, 2002. ACM.

[6] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 195–205, New York, NY, USA, 2004. ACM.

[7] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, pages 34–44, 2009.

[8] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 339–348, May 2001.

[9] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: The distribution of program failures in a profile space. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 246–255, New York, NY, USA, 2001. ACM.

[10] A. B. Downey and D. G. Feitelson. The elusive goal of workload characterization. *SIGMETRICS Perform. Eval. Rev.*, 26(4):14–29, Mar. 1999.

[11] S. Elbaum and M. Diep. Profiling deployed software: assessing strategies and testing opportunities. *Software Engineering, IEEE Transactions on*, 31(4):312–327, April 2005.

[12] G. Gay, M. Staats, M. Whalen, and M. P. E. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, Aug 2015.

[13] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, Dec 1990.

[14] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely-collected program execution data. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 146–155, New York, NY, USA, 2005. ACM.

[15] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '98, pages 83–90, New York, NY, USA, 1998. ACM.

[16] K. A. Huck and A. D. Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 41–41, Nov 2005.

[17] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[18] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *SIGPLAN Not.*, 47(6):77–88, June 2012.

[19] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[20] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, 2005.

[21] H. Malik, H. Hemmati, and A. E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1012–1021, Piscataway, NJ, USA, 2013. IEEE Press.

[22] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: distributed continuous quality assurance. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 459–468, 2004.

[23] I. Molyneaux. *The Art of Application Performance Testing*. O'Reilly Media, 1 edition, Jan. 2009.

[24] P. A. Nainar and B. Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 255–264, 2010.

[25] A. Orso, J. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 67–ff, New York, NY, USA, 2003. ACM.

[26] A. Orso, D. Liang, and M. J. Harrold. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 65–69, Rome,Italy, July 2002.

[27] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 277–284, New York, NY, USA, 1999. ACM.

[28] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang. Estimation of software reliability by stratified sampling. *ACM Trans. Softw. Eng. Methodol.*, 8(3):263–283, July 1999.

[29] A. Podgurski and C. Yang. Partition testing, stratified sampling, and cluster analysis. In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '93, pages 169–181, New York, NY, USA, 1993. ACM.

[30] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[31] S. Yan, Z. Chen, Z. Zhao, C. Zhang, and Y. Zhou. A dynamic test cluster sampling strategy by leveraging execution spectra information. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 147–154, April 2010.

[32] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 15–26, New York, NY, USA, 1997. ACM.