Experience Report: Mining Test Results for Reasons Other Than Functional Correctness

Jeff Anderson *
* Microsoft
North Dakota State University
jeffrey.r.anderson@ndsu.edu

Hyunsook Do †
† University of North Texas
hyunsook.do@unt.edu

Saeed Salem ‡
‡ North Dakota State University
saeed.salem@ndsu.edu

***Abstract*—Regression testing is an important part of software development projects, and it is used to ensure software quality. Traditionally, a regression test focuses primarily on functional correctness of a modified program and is examined only when it fails, meaning it found a fault that would have otherwise been undetected. For certain application domains, regression tests for non-functional quality aspects such as performance, security, and usability could be just as important. However, those regression tests are much more costly and difficult to create, and thus many applications lack adequate non-functional regression test coverage. This adds risk of regressions in these areas as changes are made over time. In this research, we propose using metrics from passing test cases to predict quality aspects of the software beyond the traditional focus of regression tests. Our industrial case study shows that metrics such as test response time from functional regression tests are good predictors of which product areas are likely to contain certain types of non-functional performance faults. Furthermore, we show that this prediction can be improved through environmental perturbation such as the use of synthetic volume datasets or data size variation.**

***Index Terms*—Data-mining software repositories, regression testing, performance, case study**

I. Introduction

Traditionally, one of the most important activities used in software development to ensure quality is regression testing [1]. Regression testing provides many benefits such as enabling refactoring of code with high confidence to catching product faults or functional regressions prior to product release. One of the most common forms of regression tests is a functional test, one that either passes or fails to determine if a functional fault exists. However, there are many other types of regression tests beyond functional tests. These tests for non-functional quality aspects include performance tests, security tests, and other aspects, which can be as important as functional regression tests for certain application domains.

A major issue encountered when writing non-functional quality tests is that those tests are often more costly to write and execute. For instance, performance tests may require highly controlled hardware environments in which they execute to avoid false positives due to processor or memory differences. Security tests may require additional data creation, setup, and orchestration between different security roles in the same test to accomplish a task. Examples such as these are one of the reasons that non-functional regression tests may have significantly less product coverage than functional regression tests. This does not mean that non-functional regression tests are less important than functional regression tests. For instance, a performance issue that stops a user from being able to complete a task is just as bad for that user as a functional bug that raises a failure during the task. Either way, the user is unable to complete their task and considers the software to be faulty.

One way regression test costs have been reduced for functional testing is through the areas of fault prediction, test case selection, and test case prioritization. These techniques use software metrics such as churn information, historical test failure information, bug information, and other metrics together with data mining techniques to predict which modules are likely to contain faults or which tests are likely to fail [2], [3], [4], [5]. More recently, advanced data mining techniques using software dependency graphs, historical defects and performance data have also been employed [6], [7], [8].

The majority of existing research in this area has focused on functional regression tests. In this research, however, we seek to determine if similar metrics and techniques can be successfully applied to non-functional testing, focusing on performance testing. In particular, we examine the use of response time metrics from *passing* functional regression tests to determine if they can improve fault prediction of one type of performance fault, a missing index fault.

This research is motivated by quality considerations in a large scale product currently being developed. As with many products, the first test artifacts produced are functional tests with adequate product coverage. However, the coverage of performance tests is smaller than desirable at this stage in the current release of the product. While the core processes have adequate performance tests, some of the less common features may not have performance tests prior to release due to engineering resource constraints. This poses a risk of undiscovered performance faults that will be costly to repair once the product has released. If these faults can be more accurately predicted, then engineering resources can more efficiently be focused on fixing them prior to release.

In our approach, we induce missing index faults in the product by manually removing indexes known to be important. Missing index faults were chosen as they are a common type of fault seen in large database products and are unable to be detected automatically by any known method. Manual induction of faults are used to allow for a known ground truth of missing index faults in the experiments. The response time for all regression tests is then tracked when run normally as well as in the presence of an artificially expanded volume dataset. Test response time is used as a metric, as the way missing index faults normally surface to the user is through excessively long response times. We then examine if the response times from test runs in these different environments can be used to accurately predict which functional regression tests also exercise queries that expose a missing index fault.

To evaluate our approach, we performed three experiments on an industrial product, *Microsoft Dynamics AX*. In these experiments, we tracked regression test response time and showed that ranking regression tests by net difference in response time between base and volume datasets provides the most effective prediction of missing index performance faults.

The rest of the paper is organized as follows. In Section II, we discuss the approach used in the research as well as describe in detail what a missing index fault is. In Section III, we detail the empirical study that we performed. Section IV provides the results of the study. Section V discusses the results and the implications of these results and Section VI presents related work. Finally, in Section VII, we provide conclusions and discuss future work.

II. Approach

In this section, we describe the approach used in performance fault prediction based on an industrial product release currently under development, *Microsoft Dynamics AX*[9]. [1]

A. Current Performance Testing Practices

Performance is known to be a very important quality aspect of the *Microsoft Dynamics AX* product because performance issues slow down the productivity of users of the product. The primary reason customers purchase an ERP system is to increase productivity, so performance is very important to users. Performance faults can lead to problems including increased hardware costs for customers who deploy the software, decreased productivity for users, and in some cases, performance faults can even stop a business process from being completed if the problem is bad enough. For these reasons, much focus is placed on performance analysis and testing within this product.

As with any major product, one problem with performance testing is a very large number of configuration options that can have significant impacts on the performance of processes. These thousands of options in different combinations can yield different performance characteristics making for a virtually infinite number of configurations that obviously cannot all be individually tested. For example, in the act of confirming a purchase order, one must consider whether taxes are being calculated, whether budgeting is being checked, what level of financial detail is being tracked, whether encumbrance for public sector entities needs to be considered, whether the purchase is for stocked or service items, and so on.

[1]At the time of writing this paper, the release from which the data has been derived has not yet shipped.

To deal with this untenable number of configurations, customer research is performed to determine which configurations and data volumes are most common in certain industry segments (e.g., retail, public sector, manufacturing, etc.). Then, targeted performance tests are built for these configurations in the business processes known to be important for that industry segment. This yields a reasonable level of confidence in the most important business processes.

As the limited performance testing resources are applied to targeted business processes and configurations, some of the less important processes and configurations may have less than desirable levels of performance testing coverage. Any performance faults in these less common areas will still be a major issue for customers if that process is core to their business. Our research focuses on detecting performance faults in these less common process and configurations that may otherwise not have extensive performance testing coverage.

B. Missing Index Faults

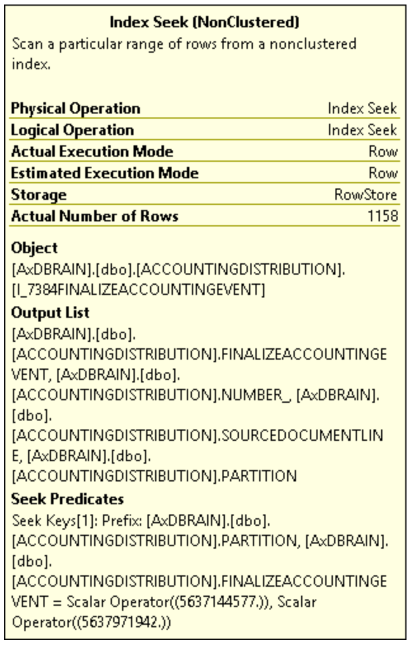
The type of fault being predicted in this study is a missing index on a table that results in what will be referred to as a "bad query plan". *Microsoft Dynamics AX* is a multi-tier enterprise resource planning product consisting of a client that connects to one or more servers that execute business logic. Those servers then connect to a back-end database that contains the data. As *Microsoft Dynamics AX* is often deployed in large companies, the number of rows in many of the database tables can be very large. It is not uncommon to have tables containing over 100,000,000 rows.

A single business operation often requires reading and writing multiple times from multiple of these tables. Scanning all the rows in the table would be exceedingly expensive, so databases rely on indexes to quickly find only the rows necessary to serve a given query. Occasionally, a table will not have an index that can support the query, and the database will need to revert to a scan of the entire table, which can be many orders of magnitude slower. This impacts not only the current running operation, but all other operations on that database as well, as it consumes CPU and IO capabilities that could otherwise be used by other queries.

There are also cases where an index *does* exist that can be used by a query, but it is not selective enough. Database indexes are an ordered set of columns for which all values must be specified in the query restriction. If a column is not specified in the query restriction, then all following index columns are ignored and the database must revert to a scan of the rest of all the records in that range. For example, suppose we have a table containing columns $(Company, PurchaseOrderId, Quantity, UnitPrice)$. An index of $(Company, Quantity)$ exists on the table. When the system executes the following query:
SELECT UnitPrice
FROM PurchaseOrders
WHERE Company = 'Contoso'

AND PurchaseOrderId = ‘1234’
Only the $Company$ column of the index can be used for selectivity. If there are many companies in the table, this may be highly selective. But if, in this installation, only a single company exists with millions of purchase orders, then the index is no more useful than a table scan.

Figure 1 shows an example of a real index seek from a missing index fault that was found in the product through manual inspection. In this case, the index seeks by *[Partition, FinanizeAccountingEvent]*, which may be selective in some cases. But, in some cases, a large number of *AccountingDistribution* records (in some configurations) finalized as part of the same *AccountingEvent*. So this index seek may still end up reading a large number of rows. When many records have the same value for both the *Partition* and *FinalizeAccountingEvent* fields, many records will be returned by this index seek. In an installation with millions of rows in the table, this would yield a significant performance issue for users who executed this query.



Index Seek (NonClustered)
Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	1158

Object
[AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].[I_7384FINALIZEACCOUNTINGEVENT]

Output List
[AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].FINALIZEACCOUNTINGEVENT, [AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].NUMBER_, [AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].SOURCEDOCUMENTLINE, [AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].PARTITION

Seek Predicates
Seek Keys[1]: Prefix: [AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].PARTITION, [AxDBRAIN].[dbo].[ACCOUNTINGDISTRIBUTION].FINALIZEACCOUNTINGEVENT = Scalar Operator((5637144577.)), Scalar Operator((5637971942.))

Fig. 1. Sample Index Seek from Missing Index Fault

As demonstrated by the example from Figure 1, the detection of bad query plans is largely a manual effort, requiring the knowledge of an expert developer who understands the different data distributions expected in installations of software as well as expected data volumes in each table and expected uses. One primary way of analyzing query plans is to take a trace of all the queries executed by some functionality and then manually analyze each query plan. This is relatively expensive as the cost of analyzing all queries executed by a single performance test may take 15 to 30 minutes for an expert developer.

C. Coverage Breadth Through Functional Tests

As previously mentioned, performance tests are often limited to the most highly-used portions of product functionality and lack product breadth. To gain that product breadth while minimizing engineering costs, we seek to use metrics from *passing* functional regression tests to predict functionality that is more likely to contain a missing index, which will yield a bad query plan.

The primary metric used in this study is the execution time of the functional regression test. This is under the assumption that bad query plans will cause functional tests to run slower than other tests without missing index faults. However, the relationship between test response time and bad query plans is not that simple. A bad query plan may make a query go from 5 milliseconds to 1500 milliseconds. But the test executing this query may take 5000 milliseconds total. A 5000 millisecond test with a 1500 bad query plan is not easily distinguishable from a 5500 millisecond test that does not invoke any bad query plans. Slow tests do not necessarily mean a bad query plan has been encountered. Further, the test dataset used by functional regression tests is very small in volume compared to a real customer deployment, as a real deployment may be many terabytes in size making unit testing unwieldy. To differentiate between tests that just take longer and tests that actually encounter bad query plans, we focus on the difference in execution time between base and volume datasets.

D. Experiment Approach

In this experiment, we will execute a large number of functional regression tests for the product, tracking the execution time of all the tests that pass. Only passing tests are evaluated, as a test failure leads to database cleanup processes and thus is less reliable as a timing. We then order the tests by descending execution time, expecting the tests with the higher execution time are more likely to have experience bad query plans due to a missing index.

If this ordering of tests proves reliable as a predictor of missing index faults, then an engineer could rank manual analysis tasks of the top tests for missing indexes. In this study, we analyze the effectiveness of the reordered rank through ROC analysis. The ROC curve is a graphical representation of the relative precision and recall as the number of selected items increases. The details about the experimental process will be discussed in Section III.

E. Ground Truth

In order to evaluate the results of the research, we must first know which functional regression tests actually encounter bad query plans. As previously mentioned, manual analysis is exceedingly expensive and would not yield a large dataset of results. To work around this, we utilized fault injection. Fault injection is a standard technique in this area of research. The technique was originally described by Clark and Pradhan[10] and is now in common use.

In this experiment, we injected faults by removing an existing index. Before doing that, we first determined which indexes are used by each query of each test. This can be done in an automated way by first capturing all the queries that were executed during the functional regression test, then querying the SQL Server dynamic management views (DMV's) to retrieve the query plan that was executed for each. DMV's are internal data structures that Microsoft SQL Server uses to track server aspects that can be queried by the database administrator. From this process, we then have a list of all queries executed by each test, as well as an XML description of the query plan that each test utilized.

Finally, an expert developer with deep product knowledge selected three very important indexes to remove. The three index removals represent three independent experiment datasets. More could be performed, but we limited the study to three based on time availability. Because similar results were seen in all three experiment datasets, we feel confident that the results are generalizable across this particular application. The indexes removed are on tables with high data volumes in normal installations, which provide high selectivity and are highly used by queries. The three indexes selected are:

1) CustTrans.TransIdIdx [Partition, DataAreaId, InventTransId, InvoiceId, InvoiceDate]
2) PurchLine.SourceDocumentLineIdx [SourceDocumentLine, Partition, DataAreaId]
3) TaxTrans.SourceTableIdSourceRecIdVoucherIdx [Partition, DataAreaId, SourceTableId, SourceRecId, Voucher]

Finally, the query and query plan data were queried to find all tests that contained at least one query that relies on that index to the level of attributes that provides good selectivity. For instance, any query plan using *CustTrans.TransIdIdx* would need to be constrained by at least three fields of the index as any fewer would not provide enough selectivity to make this a proper index usage. *DataAreaId* in these indexes corresponds to a legal entity in the installation, so a customer with a single legal entity but millions of inventory transactions would see no selectivity from this index unless *[Partition, DataAreaId, InventTransId]* were all provided. Once all three of these segments are provided, the remaining segments of *[InvoiceId, InvoiceDate]* provide little if any additional selectivity.

Following this pattern, *PurchLine* is useful with a single segment (since *SourceDocumentLine* is a unique *Int64* value for each row in the table), and *TaxTrans* is useful after four segments. In *TaxTrans*, *[Partition, DataAreaId]* filters to a single company, which is not selective. *SourceTableId* is also not selective. Some customers will have all of their tax transactions in a single table, for instance if they are a distribution company without direct sales. The *SourceTableId* value is only selective once *SourceRecId* is added along with the three prefix segments.

F. Synthetic Volume

Another important aspect to consider when analyzing performance of database applications is the amount of volume in the tables. As mentioned earlier, most functional tests are executed on a artificially small dataset. One tool used in performance testing is synthetic volume, where a large number of rows is added to "important" tables prior to test execution to simulate a more realistic volume in a real installation. The *Microsoft Dynamics AX* application contains many thousands of tables, and the relationships between these tables is extremely complex. So synthetic volume is dummy rows added only to a small number of important tables. Further, this data is not used by any functional tests. It just expands the number of rows in the tables so any bad query plans will show up as more degraded than they would have on base volume, hopefully making them easier to find.

III. Empirical Study

In this study, we seek to understand whether metrics from *passing* functional regression tests can be used for fault prediction of non-functional performance faults, specifically missing indexes that yield a bad query plan. To investigate this research problem, we address the following research questions:

1) RQ1: Does ordering regression tests by execution time effectively predict which tests encounter a bad query plan?
2) RQ2: Can environment perturbation by the introduction of synthetic volume improve the effectiveness of that prediction?
3) RQ3: Can the effectiveness of the prediction be further improved by analyzing the differences in results between baseline and synthetic volume datasets?

A. Object of Analysis

This study is based on a pre-release version of *Microsoft Dynamics AX*. The product is currently undergoing significant architectural change to enable new deployment topologies, and as such the vast majority of functional and performance tests need to be migrated. In many cases, the tests are being rewritten during the migration to make them faster, more stable and more targeted. This provides a unique opportunity to use a set of very reliable and very fast functional tests with good product breadth to predict performance issues on an otherwise relatively mature product. Specific metrics for product size cannot be provided as the product is still in development. However, we are able to provide high level ranges of metrics as discussed below.

Microsoft Dynamics AX is a major enterprise resource planning (ERP) product. It contains millions of lines of code and has tens of thousands of functional test cases, also spanning millions of lines of code. A full regression test suite run takes multiple hours (2-5) when execution is spanned out to multiple computers (up to 10).

In our previous work [11], [12], we have discussed regression tests on this product taking multiple minutes per test. With the new release, many of the tests have been entirely rewritten to be both more reliable and efficient. The average execution time per functional test is now around 1 second, with many of the tests executing in less than 100 milliseconds each.

The database schema for the product contains over 8,000 individual tables with over 19,000 indexes. This is an average of just over two indexes per table. Some tables contain more than 20 indexes. The indexes have been added over multiple versions, often as the result of detecting performance faults in either performance tests or occasionally even in real customer environments.

B. Variables and Measures

1) Independent Variables: This study manipulates one independent variable in each of the three independent experiment datasets. For each experiment dataset, one heuristic is used and compared against the control.

1) Ordering tests by descending response time on regular data (Heuristic for RQ1)
2) Ordering tests by descending response time on volume data (Heuristic for RQ2)
3) Ordering tests by descending net difference in response time between regular and volume data (Heuristic for RQ3)

The control technique used is random ordering of tests (control for all RQs), as the research question is attempting to determine if mining data from test metrics allows for faster detection of bad query plans. While this may appear overly naive, this is effectively the currently used approach as no ordering of tests is currently used when searching for missing index faults.

2) Dependent Variables: The dependent variable is the accuracy of the ordering of tests. An ideal ordering would rank all tests that expose a bad query plan first, while the worst possible ordering would rank them last. A standard technique of analyzing the accuracy of ranking is the use of an ROC curve. ROC stands for "receiver operating characteristic", and is a standard statistical measurement plotting the true positive rate against the false positive rate. In an experiment such as this evaluating ranking, the ROC curve represents the precision and recall of the number of bad query plans found as the number of tests analyzed is increased.

From the ROC curve, the AUC or "area under curve" can be computed. As the ranking improves, the AUC is also increased. The measurement of AUC provides a value between 0 and 1 (with 1 being as good as possible) that provides a numeric method of determining the benefit gained from an ordering. With an AUC of 1, this would indicate that all tests that exposed a bad query plan were ranked first. Similarly, an AUC of 0.5 would be generated by a random ordering of tests.

The use of ROC analysis also provides a standard method of computing the probability that the difference between curves is due to random chance. Thus, this study seeks to maximize AUC from the ROC curve using the heuristics specified, and to show the probability of random chance causing the difference is sufficiently low.

C. Experimental Process

There are three separate experiment datasets used in this study. This is done by manually removing one of three indexes, and then running all tests on both base and volume data as shown in Figure 2. These three experiment datasets are independent from each other. The resulting datasets are then ordered based on the control and heuristics methods explained in the previous section. For each ordering, an ROC curve is generated and the AUC is computed. To evaluate the effectiveness of the heuristics, the ROC curves are compared with each other to yield both a difference in AUC as well as a p-value.

Figure 2 illustrates the overall process used in this study. As shown in the top part of the figure, the query plan data was captured with all existing indexes intact. Then, as shown in the lower part, one important index (as decided by an expert developer) was removed and all tests were executed both on base data and on volume data, capturing the response times of each. The results of this response time were then compared against the ground truth of which tests encountered a missing index by querying the query and query plan data. The index was then re-enabled and another index was disabled so the same process could be repeated.

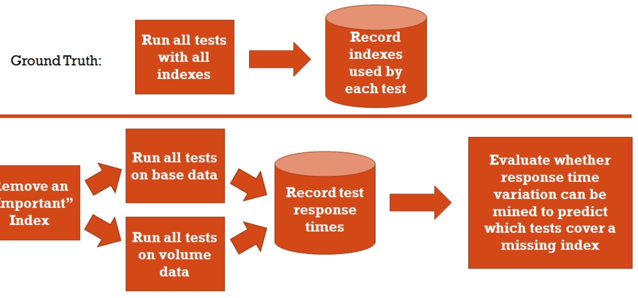
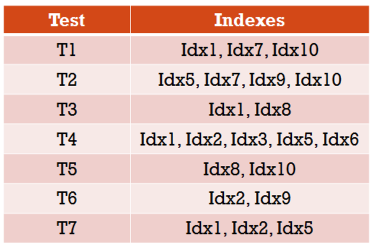

Ground Truth:
Run all tests with all indexes
Record indexes used by each test
Remove an "Important" Index
Run all tests on base data
Run all tests on volume data
Record test response times
Evaluate whether response time variation can be mined to predict which tests cover a missing index

Fig. 2. Experimental Process

The first step of the experiment process is determining which tests rely on which indexes. This step is done by executing the test with all indexes enabled and then querying the Microsoft SQL Server Dynamic Management Views to determine which indexes were used. A table is then created listing each test and the set of all indexes it used. An example is shown in Figure 3. For example, test $T1$ executes queries that rely on indexes 1, 7 and 10.



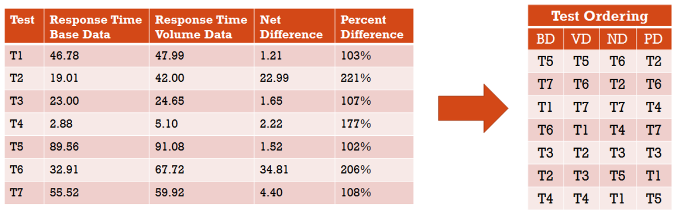
Test	Indexes
T1	Idx1, Idx7, Idx10
T2	Idx5, Idx7, Idx9, Idx10
T3	Idx1, Idx8
T4	Idx1, Idx2, Idx3, Idx5, Idx6
T5	Idx8, Idx10
T6	Idx2, Idx9
T7	Idx1, Idx2, Idx5

Fig. 3. Index Usage

The next step is to remove an index. In this example, consider a removal of index $Idx5$ as shown in Figure 3. This means that tests 4, 6 and 7 are missing an index when run, simulating a missing index fault.

Within each experiment dataset where an index has been removed, the next step is to generate an ordering of tests. Figure 4 illustrates how this is done. The test is run twice with the index removed: first on the base data and then again on a synthetic volume dataset. The response times for each are tracked and used to also generate the net difference in response time and percentage difference in response time. The net difference is $volumetime - basetime$ and the percentage difference is $volumetime/basetime$.

This gives four different values for each test. The tests are then ordered by each of the values individually producing four orderings, shown in Figure 4 as BD (base data), VD (volume data), ND (net difference), and PD (percent difference).

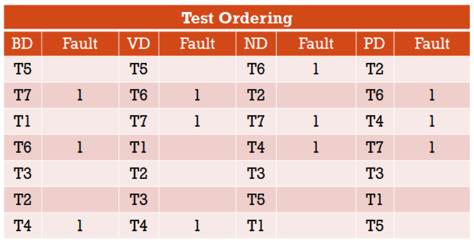


Test	Response Time Base Data	Response Time Volume Data	Net Difference	Percent Difference
T1	46.78	47.99	1.21	103%
T2	19.01	42.00	22.99	221%
T3	23.00	24.65	1.65	107%
T4	2.88	5.10	2.22	177%
T5	89.56	91.08	1.52	102%
T6	32.91	67.72	34.81	206%
T7	55.52	59.92	4.40	108%

Test Ordering			
BD	VD	ND	PD
T5	T5	T6	T2
T7	T6	T2	T6
T1	T7	T7	T4
T6	T1	T4	T7
T3	T2	T3	T3
T2	T3	T5	T1
T4	T4	T1	T5

Fig. 4. Test Ordering

Figure 5 illustrates how the ground truth information is applied to the test orderings from Figure 4. The ground truth was determined in the first step when all tests were run with all indexes applied. By tracing the indexes used, we know which indexes should exist for a given test. When an individual index is removed for an experiment dataset, $Idx5$ in this example, we can cross reference this with the tracing information to determine which tests are now missing an index in that experiment dataset. As shown in Figure 5, tests 4, 6 and 7 are known to have a missing index in this experiment dataset because these tests exercise index $Idx5$. These faults are then marked in the test orderings.



Test Ordering							
BD	Fault	VD	Fault	ND	Fault	PD	Fault
T5		T5		T6	1	T2	
T7	1	T6	1	T2		T6	1
T1		T7	1	T7	1	T4	1
T6	1	T1		T4	1	T7	1
T3		T2		T3		T3	
T2		T3		T5		T1	
T4	1	T4	1	T1		T5	

Fig. 5. Fault Detection By Test With Idx5 Removed

From the data shown in Figure 5, an ROC curve can be computed. Consider the net difference (ND) ordering with $Idx5$ removed. For the first ranked test, investigating this test leads to discovering a fault, one of the three faults that exist. Therefore, sensitivity (true positive rate) is "discovered-faults/all-faults", $1/3$, 0.33. Specificity (true negative rate) is $4/4$, 1 since all tests that do not have faults were labeled as not having faults, below the first rank. For the first ranking we get the [1,0.33] point as shown in the figure.

For the top two rankings, sensitivity is $1/3$, 0.33, since one true fault out of the three total was discovered in the top two ranked tests. Specificity for the second point is $3/4$, 0.75, since the second ranked test was a false positive. So the second point in the ROC figure is [0.75, 0.33]. For the top three rankings, sensitivity is $2/3$, 0.66, and the specificity remains 0.75. So we get point [0.75, 0.66] in the ROC curve. Graphing sensitivity (true positive rate) vs specificity for all the different rankings generates the ROC curve shown in Figure 6 with an AUC of 0.833.

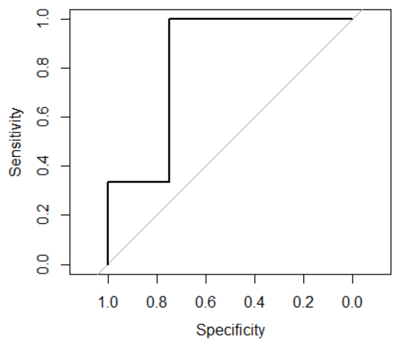


Fig. 6. Example ROC Curve

IV. Data and Analysis

In this section, we present the results of this study as described in the previous section.

A. Overall Results

TABLE I
AUC by Independent Variable for Each Experiment

	Control	Base Avg	Volume Avg	Net Diff
CustTrans	0.500	0.880	0.965	0.993
TaxTrans	0.500	0.882	0.900	0.906
PurchLine	0.500	0.948	0.967	0.988

Table I shows the AUC for the control and each heuristic for the three experiment datasets. Because the control technique is random selection, the AUC for each is 0.500. Higher numbers for AUC are better.

This data is based on three independent experiment datasets consisting of 950 tests for each. The 950 tests were the first 1000 executed by the test system with 50 removed that failed or showed extremely high variability between runs. The same tests were used in all three experiment sets, only the removed index was varied. For each heuristic in each experiment dataset, each test was executed a total of 6 times. The high and low response time dropped and the remaining 4 times averaged to yield the response time used. Standard deviation was calculated if the standard deviation was more than 25% of the average response time, that test was removed from the result set due to excessive variation.

For all experiment datasets the heuristics showed much higher AUC compared to the control. There were also slight improvements shown moving from base to volume data, and more slight improvements using the net difference between the two. This provides support for the goal of this study that was to show metrics obtained from passing test cases can be used in other quality activities. The remainder of this section will explain in more detail each of the research questions and the associated data.

B. RQ1 Analysis

RQ1 seeks to determine whether or not ordering tests by execution time can improve the prediction of tests that will invoke a bad query plan. The answer is yes: Ordering by execution time does effectively predict which tests encounter bad query plans. This can be seen in the first two columns of Table I. The control for this research question is always 0.500, having an AUC of 0.500. The three index removals used in this study with the first heuristic ranged in AUC from 0.880 to 0.948, showing a large improvement over the naive baseline.

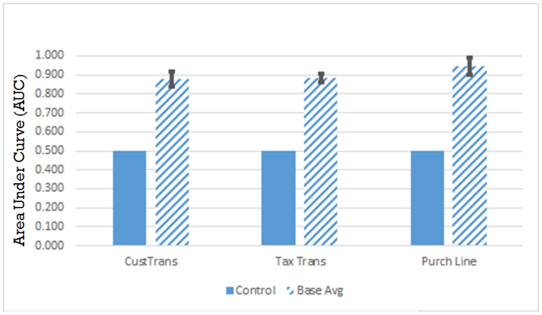


Fig. 7. Base Data AUC with 95% Confidence Interval

Figure 7 illustrates the difference in AUC between the control and the heuristic of ordering by descending execution time on base data. The 95% DeLong confidence interval is also shown by the vertical bars on top of the "Base Avg" results. From this result, we can say with high confidence that the heuristic does improve test ranking for detection of bad query plans.

C. RQ2 Analysis

After showing that the technique shown in RQ1 was effective, we now seek to improve that ranking by gathering additional metrics. The answer for RQ2 is also yes: Additional metrics beyond response time on base data do improve predictions. In this case, the heuristic is to order the test cases based on their descending execution time when executed on a synthetic volume dataset. Because RQ2 is trying to determine what improvement can be made over the heuristic from RQ1, we are no longer comparing against random ranking and so raw AUC differences are not sufficient to determine if the technique has improved predictions. Column three of Table I shows that AUC has been increased using this heuristic, but we must determine if the improvement is due to chance or because of an actual improvement.

Using a null hypothesis that the difference in AUC is zero between the base average ordering and the volume average ordering, we find p-values of 5.78 e-10, 0.007 and 0.068 for the *CustTrans*, *TaxTrans*, and *PurchLine* experiment datasets, respectively. While the p-value for the *PurchLine* test does not meet the traditional 95% level, the other two tests clearly show the null hypothesis can be rejected. The difference in p-value between the tests appears to be primarily due to the difference in the number of positive values in each dataset. *PurchLine* has only 8 tests that exercise a bad query plan, while *CustTrans* has 50. Comparisons between the ROC curves for order by base average and volume average times are shown in Figure 8.

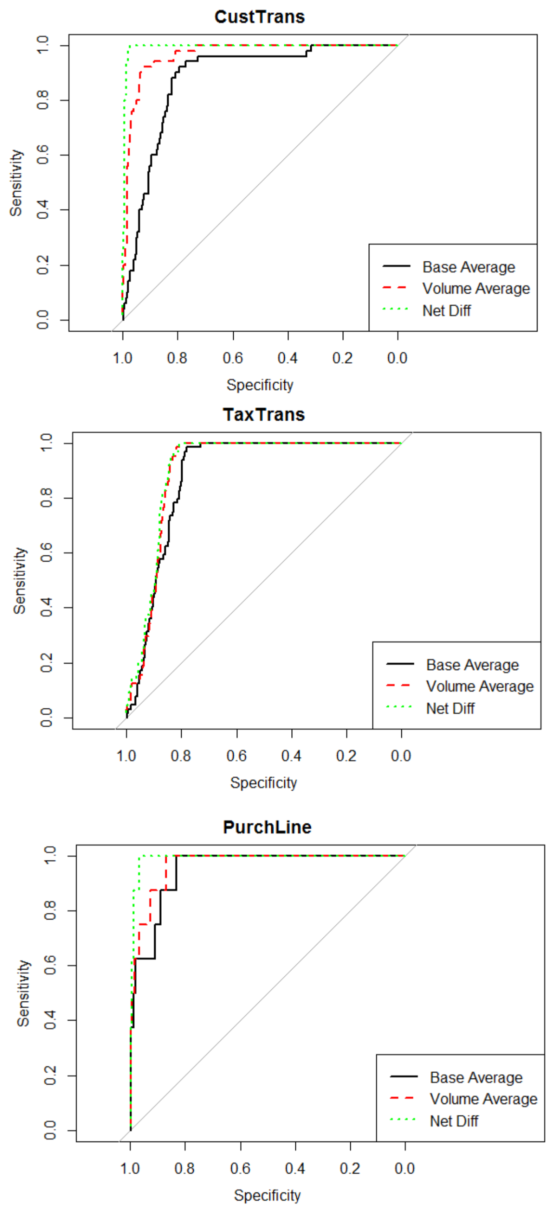


Fig. 8. ROC Curves for Three Experiments

D. RQ3 Analysis

In RQ3, we seek to determine if more complex metrics can further improve the ranking of test cases. The answer is yes: More complex metrics calculated based on multiple other metrics do further improve predictions. The heuristic used here is the net difference in response time between base and volume test runs. This data is the most complicated to capture, as switching between base and volume data requires a redeployment of the software. During redeployment, a large percentage of disk space is impacted, thus minor things like disk fragmentation, remaining disk free space and even various hard drive performance metrics can have a noticeable impact on test response time. This research question then seeks to determine if the extra data available in this more complex metric outweighs the implicit randomness introduced by the increased complexity.

Figure 8 shows the overall ROC curves for all three experiment datasets. Sensitivity is the true positive rate while specificity is the true negative rate. As shown in Table I, the AUC for all three has increased. Using the null hypothesis that the difference in AUC between ordering by volume average and net difference is zero, we find p-values of 0.0001, 0.018 and 0.145 for *CustTrans*, *TaxTrans* and *PurchLine*, respectively. We are unable to reject the null hypothesis for the *PurchLine* experiment dataset, but the *CustTrans* and *TaxTrans* experiment datasets both conclusively show that more advanced metrics can be mined to further improve the test case ranking.

The primary difference between these experiment datasets again appears to be due to the difference in the number of tests that exercise the bad query plans. With only 8 out of 942 positive tests in the *PurchLine* experiment dataset, there is not enough difference in the ranking to conclusively prove the results have been improved.

E. Other Metrics

The three metrics presented in this study so far of base response time, volume response time and net difference are obviously not the only metrics available. They were selected after manual analysis of numerous metrics that showed these three provided the best results. For example, it is also possible to calculate the percentage difference between the base average and volume average response times. However, using this metric significantly *decreases* the effectiveness of the ranking. This is shown in Figure 9. This and other less effective metrics thus show that the selection of proper metrics for use in ranking should be carefully considered, as more complex attributes are not necessarily better.

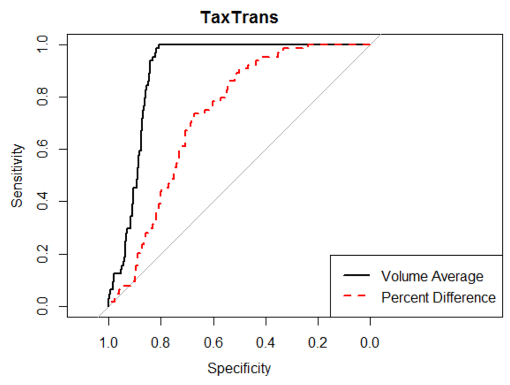


Fig. 9. ROC Curves for Percent Difference vs Volume Average

V. Discussion and Implications

In this section, we discuss the implications of the results presented. As previously mentioned, the motivation for our research was a lack of testing resources to provide adequate non-functional test coverage across the breadth of the application. We have shown through this study that functional test metrics *can* be mined to better prioritize limited resources for performance analysis and performance testing, compared to a naive baseline of random application.

A. Usefulness of Approach

With this study, we have demonstrated a new technique for locating non-functional faults in software. As with any quality process, the usefulness is a factor of the benefits from using the technique, offset by the costs of employing that technique. In all three research questions, we find that the the majority of tests exposing the missing index faults occur within the first 30% of the tests run when ordered by any of the heuristics. So given a project with missing indexes such as those used in our experiment, it would be a useful technique in detecting these faults.

There are a number of caveats to this approach as described below. In order to get these results, the missing indexes had to be important, had to occur in a table with artificial volume, and the tests had to be run in a stable environment between versions.

In practice, we have found this technique difficult to use thus far. We attempted to find new, previously unknown faults with this technique, but were largely unable to for one primary reason. In the product we used, the only tables that contain synthetic volume are the ones deemed most important, which are also the same ones that have had extensive manual index analysis performed, and therefore are unlikely to have missing index faults. This technique would be much more applicable to a database product earlier in its life cycle before this extensive index analysis had been done. At that time, when more severe missing index faults exist, we believe that the technique would be much more applicable.

We also believe this technique can be further improved by including many more factors in the prediction model beyond just response time. Adding metrics to fault prediction models in functional testing has been shown to be beneficial, so we believe it will be beneficial here as well. The remainder of this section discusses the specific considerations that must be made when applying this approach in a software development project.

B. Environmental Purity

One of the primary issues involved in response time measurement is ensuring the test environment is identical in any comparison. Very minor environmental issues such as a difference in processor speed, a difference in RAM available, or even a change in virus scan settings can have significant and appreciable impacts on response time. Thus any response time numbers either must be performed on a highly controlled environment to ensure repeatability, or must be averaged over a significant number of executions in order to ensure environmental aspects are averaged out.

For this experiment, we used a single HyperV instance on the same physical hardware for all experiment datasets. As the RAM, hard drive space, processors and other aspects are not changed between tests, this reduces variability. Further, each test was executed multiple times in a row and the fastest and slowest results are thrown out leaving only the stable runs in between in the result set. By doing this, most test response times had a standard deviation of less than 5% of the average response time for that test.

Unfortunately, in order to switch experiment datasets, the software needed to be reinstalled between each run. This involves the re-writing of many gigabytes of data on the hard drive, which does cause variation in hard drive seek time and other aspects for different records. For this reason, we do still see variation for the same test across two experiment datasets where bad query plans and backing data were not changed between the two runs. This is similar to the issues faced in an industrial product where nightly regression test runs will similarly exhibit random variation. Even with this variation, our results show that the prioritization based on response time is still effective.

C. SQL Server Specifics

One particular challenge faced when examining the response time of workloads based on SQL Server is the extremely complex data caching architecture used by SQL Server to optimize disk access. As data is used, SQL Server moves data pages into RAM memory in very efficient ways that reduce future physical disk access. Since disk access is orders of magnitude slower than RAM access, this means "warm" queries that are able to pull data from RAM are often times orders of magnitude faster than "cold" queries regardless of the goodness of the query plan. This means that the order in which tests are run can have a significant impact on the observed response times.

To protect against this variation, two solutions were used in these experiment datasets. First and most simplistically, each test was executed multiple times in a row and the slowest and fastest results were ignored. This ensures that SQL has a chance to cache any required data pages prior to the execution. Secondly, the test framework was updated to call DBCC DROPCLEANBUFFERS between each test execution. DROPCLEANBUFFERS is a special command that tells SQL Server to drop all clean buffers from the buffer pool. This command is provided to allow simulating test queries on a cold buffer cache without needing to restart SQL Server between test runs. While this does not guarantee equivalent execution time between queries, it greatly reduces any impact that cold vs warm buffers have on the results.

The other SQL aspect that has a significant impact on query performance is the available memory size. Many production servers for large enterprise resource planning products like *Microsoft Dynamics AX* would have tens to hundreds of gigabytes of RAM available. In a system like this, the entire contents of the synthetic dataset can easily be held in RAM. At the same time, a typical customer deployment for an instance like this would have multiple terabytes of data. This makes it difficult to properly simulate the required database paging necessary when an extremely large dataset does not fit in RAM and encounters a bad query plan with a table scan.

To work around this issue, we used a configuration setting in SQL Server that allows a maximum RAM size to be specified. For all experiment datasets, the maximum RAM size was set to 1 gigabyte of memory. This is far lower than a customer environment would ever have. But it is large enough to still be functional, while being small enough to exhibit slowness when bad query plans are encountered. In the synthetic volume dataset used for expansion in this experiment, the top 10 tables each contain approximately 1 gigabyte of data. This forces any bad query plans on these tables to page in and out data.

D. Relative Time

An important aspect of this study is the fact that relative execution time between tests in a given run is a fairly accurate measure to use for prioritization. Cross-run comparisons were shown to increase the effectiveness of the prioritization somewhat, but the majority of the benefits compared to a naive baseline were obtained without the need for cross-run data. This means the issues caused by different environments, different installations, and different configurations can largely be ignored as the results are still significant (beyond the 95% threshold) even when only looking at the relative differences in response time.

E. Test Data Volume by Table

When we originally started this study, the plan was to apply the different heuristics and then manually analyze all query plans for the top one hundred or so tests from each heuristic, looking for bad query plans. Unfortunately, after analyzing approximately 100 tests in this way, we had only found a total of two bad query plans. We attribute this to the fact that the only around 40 (out of over 8,000 total) tables have a large number of records in the synthetic volume dataset. Additionally, less than 10% of all the tables in the system have at least 1,000 records even in the base dataset. In fact, over half of the tables in the system have three or fewer records in both the base dataset and the synthetic volume dataset. Unfortunately, for extremely small row counts, the execution time of a bad query plan that scans the table is roughly equivalent to the execution time of a good query plan that hits an index. In fact, based on statistics SQL will sometimes (properly) choose a query plan with a table scan if internal statistics indicate the table has a very small number of records.

What this means is the only tables that have a sizable number of records are also the most important tables in the system. For these tables, we have already done extensive index optimization and thus very few missing index bugs are likely to exist. The goal of this research was to provide wide product coverage in less important areas for which extensive performance testing has not been done, and thus more performance risk exists.

From this study, we understand that synthetic volume is a valuable tool in increasing the detection of bad query plans. An interesting extension to the research would be to artificially expand volume in the 90% of tables that have little or no existing data in the test datasets. With that synthetic data created, it would then be interesting to do manual query analysis ordered by execution time to see if previously undiscovered bad query plans could be diagnosed.

F. Threats to Validity

The primary threat to validity in this study is the manual selection of indexes to disable. Only three indexes were chosen due to time constraints, as the repeated runs in multiple configurations made each experiment dataset take approximately one week. Since we desired to collect all experiment datasets on the same hardware, this limited the number of indexes we could disable. The indexes that were disabled were also chosen based on the expert developer knowledge that they were heavily used throughout the system, and thus were likely to be exposed by multiple tests. This was necessary as shown in the PurchLine results because if an index is only exercised by a handful of tests the P-value of the experiment becomes too low to be useful. Were indexes chosen randomly, or were we able to detect "naturally" missing indexes, the results may be different. Unfortunately it was not possible to construct the experiment in this way.

Another threat is the variation in timing even when run on the same machine. If the running time of the test is compared across two different runs, we see much larger deviation from the mean than we see in repeated runs within the same installation. This is discussed earlier in the paper, but it adds to the possibility that run variation has had an unintended impact on the results. The consistency of the results across multiple indexes makes this less likely, but it is still a concern.

There is also a concern with the number of tests executed. 942 tests were executed, and were selected by running the first 1,000 tests detected through code reflection to have a dependency on the test dataset. 58 of these tests failed on one or more of the environmental deployments and were thus dropped from all six results sets. More tests exist than just these 1,000, so it would be interesting to see if the same results hold if the entire regression test suite were run. Unfortunately, this was prohibitively expensive in these experiment datasets due to the excessive time involved in capturing all query plans from all queries performed by all tests.

VI. Related Work

Fault predication is one of the software engineering research areas that have been paid much attention by many researchers, much of this research was surveyed by Fenton and Neil [13]. Fault prediction research often focuses on collecting multiple metrics from various aspects of the software or environment, performing data mining techniques on them, and using the results to predict faulty modules or tests that are likely to fail.

Examples of this include Zimmerman et al.[14] who showed that source code repositories could be mined to make these types of predictions. Zimmerman together with Nagappan expanded this work to include dependency information [7], [15]. Other researchers such as Shihab [16], Guo et al.[17] and Nagappan and Ball[18] have further extended the research to include many different metrics to increase the quality of the predictions. A recently work by the authors [12] showed in related research that regression test run histories could be mined to perform similar test failure prediction.

Fault prediction is one way of reducing functional regression testing costs. In fault prediction, various metrics such as churn, historical fault information, bug information, and other metrics are mined to predict which files, functions, or modules are most likely to contain faults. Research on performance faults, on the other hand, focuses on direct detection of specific patterns. Recently, Nistor et al. [19] demonstrated a tool they call CARAMEL that detects specific patterns in loops that can be optimized. Moving beyond the detection of a specific pattern, performance research largely focuses on watching specific performance counters during execution. This can be automated as shown by Foo et al. [20] or a manual tracing process with techniques such as "subsuming methods" demonstrated by Maplesden et al. [21].

All of the existing techniques we are aware of rely on one of three patterns.

1) Detecting a performance fault by a known static pattern [19]
2) Identifying a performance regression from a previous version [20]
3) Identifying the most expensive method executions without regard to faults [21]

To our knowledge, our new approach is the first attempt at applying traditional data mining and fault prediction techniques to find faults without a known static signature and without the faults being a regression from a previous version.

VII. Conclusions and Future Work

In this research, we have shown that the techniques normally applied in fault prediction and regression testing have further applications beyond functional bugs and have presented a case study using an industrial application. Non-functional fault classes such as performance bugs can be more accurately predicted by mining software metrics and applying similar analysis and ranking techniques. We have discussed industrial situations in which test ranking is not only economically advantageous, but also necessary based on project constraints to ensure proper non-functional quality. The major contribution of this research is the demonstration of a new application for data mining in regression testing, as well as learning about the considerations and constraints that must be taken into account when applying test ranking for predicting non-functional faults.

In future research, we would like to expand on this novel application of traditional research techniques considering other classes of non-functional faults, as well as other metrics for use in predicting these faults.

Acknowledgment

This work was supported, in part, by NSF CAREER Award CCF-1149389 to University of North Texas.

References

[1] R. V. Binder, *Testing Object-Oriented Systems*. Upper Saddle River, NJ: Addison Wesley, 1999.

[2] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[3] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[4] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM*. IEEE, 2007, pp. 364–373.

[5] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4. ACM, 2004, pp. 86–96.

[6] M. Mayo and S. Spacey, "Predicting regression test failures using genetic algorithm-selected dynamic performance analysis metrics," in *Search Based Software Engineering*. Springer, 2013, pp. 158–171.

[7] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the International Conference on Software Engineering*, 2008, pp. 531–540.

[8] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev, "Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2011, pp. 357–366.

[9] Microsoft Corporation, "Overview of Microsoft Dynamics AX," https://www.microsoft.com/en-us/dynamics/erp-ax-overview.aspx, Jun. 2015.

[10] J. A. Clark and D. K. Pradhan, "Fault injection: A method for validating computer-system dependability," *Computer*, vol. 28, no. 6, pp. 47–56, 1995.

[11] J. Anderson, S. Salem, and H. Do, "Improving the effectiveness of test suite through mining historical data," in *Proceedings of the Working Conference on Mining Software Repositories*. ACM, 2014, pp. 142–151.

[12] ——, "Striving for failure: An industrial case study about test failure prediction," in *Proceedings of the 37th IEEE International Conference on Software Engineering*, 2015.

[13] N. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, Jul. 1999.

[14] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller, "Mining versions histories to guide software changes," in *Proceedings of the International Conference on Software Engineering*, May 2004, pp. 563–572.

[15] T. Zimmermann and N. Nagappan, "Predicting defects with program dependencies," in *ESEM*. IEEE Computer Society, 2009, pp. 435–438.

[16] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Predicting re-opened bugs: A case study on the eclipse project," in *2010 17th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2010, pp. 249–258.

[17] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Not my bug! and other reasons for software bug report reassignments," in *Proceedings of the ACM 2011 conference on Computer Supported Cooperative Work*. ACM, 2011, pp. 395–404.

[18] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," in *International Symposium on Empirical Software Engineering and Measurement*, 2007, pp. 364–373.

[19] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *Proceedings of the 37th IEEE International Conference on Software Engineering*, 2015.

[20] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "An industrial case study on the automated detection of performance regressions in heterogeneous environments," in *Proceedings of the 37th IEEE International Conference on Software Engineering*, 2015.

[21] D. Maplesden, K. von Randow, E. Tempero, J. Hosking, and J. Grundy, "Performance analysis using subsuming methods: An industrial case study," in *Proceedings of the 37th IEEE International Conference on Software Engineering*, 2015.