Use of Relative Code Churn Measures to Predict System Defect Density

Nachiappan Nagappan*
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
nnagapp@ncsu.edu

Thomas Ball
Microsoft Research
Redmond, WA 98052
tball@microsoft.com

ABSTRACT
Software systems evolve over time due to changes in requirements, optimization of code, fixes for security and reliability bugs etc. Code churn, which measures the changes made to a component over a period of time, quantifies the extent of this change. We present a technique for early prediction of system defect density using a set of relative code churn measures that relate the amount of churn to other variables such as component size and the temporal extent of churn.
Using statistical regression models, we show that while absolute measures of code churn are poor predictors of defect density, our set of relative measures of code churn is highly predictive of defect density. A case study performed on Windows Server 2003 indicates the validity of the relative code churn measures as early indicators of system defect density. Furthermore, our code churn metric suite is able to discriminate between fault and not fault-prone binaries with an accuracy of 89.0 percent.

Categories and Subject Descriptors
D.2.7 **[Software Engineering]:** Distribution, Maintenance, and Enhancement – *Version control.* D.2.8 **[Software Engineering]:** Metrics - *Performance measures*, *Process metrics*, *Product metrics.*

General Terms
Measurement, Design, Reliability.

Keywords
Relative code churn, defect density, fault-proneness, multiple regression, principal component analysis.

1. INTRODUCTION
A "reliability chasm" often separates the quality of a software product observed in its pre-release testing in a software development shop and its post-release use in the field. That is, true field reliability, as measured by the number of failures found by customers over a period of time, cannot be measured before a product has been completed and delivered to a customer. Because true reliability information is available late in the process, corrective actions tend to be expensive [3]. Clearly, software organizations can benefit in many ways from an early warning system concerning potential post-release defects in their product to guide corrective actions to the quality of the software.

We use *code churn* to predict the defect density in software systems. Code churn is a measure of the amount of code change taking place within a software unit over time. It is easily extracted from a system's change history, as recorded automatically by a version control system. Most version control systems use a file comparison utility (such as diff) to automatically estimate how many lines were added, deleted and changed by a programmer to create a new version of a file from an old version. These differences are the basis of churn measures.

We create and validate a set of *relative* code churn measures as early indicators of system defect density. Relative churn measures are normalized values of the various measures obtained during the churn process. Some of the normalization parameters are total lines of code, file churn, file count etc. Munson et al. [17] use a similar relative approach towards establishing a baseline while studying code churn. Studies have shown that absolute measures like LOC are poor predictors of pre- and post release faults [7] in industrial software systems. In general, process measures based on change history have been found be better indicators of fault rates than product metrics of code [9]. In an evolving system it is highly beneficial to use a relative approach to quantify the change in a system. As we show, these relative measures can be devised to cross check each other so that the metrics do not provide conflicting information.

Our basic hypothesis is that code that changes many times pre-release will likely have more post-release defects than code that changes less over the same period of time. More precisely, we address the hypotheses shown in Table 1.

Our experiments on Windows Server 2003 (W2k3) support these four hypotheses with high statistical significance. We analyzed the code churn between the release of W2k3 and the release of the W2k3 Service Pack 1 (W2k3-SP1) to predict the defect density in W2k3-SP1. The relative code churn measures are statistically better predictors of defect density than the absolute measures.

ICSE'05, May 15–21, 2005, St. Louis, MO, USA.

* Nachiappan Nagappan was an intern with the Testing, Verification and Measurement Group, Microsoft Research in the Summer of 2004 when this work was carried out.

They also they are indicative of increase in system defect density and can accurately predict the system defect density with a high degree of sensitivity. Our metric suite is able to discriminate between fault and not fault-prone binaries in W2k3-SP1 with an accuracy of 89.0 percent.

Table 1. Research Hypotheses

	Hypothesis
H_1	Increase in relative code churn measures is accompanied by an increase in system defect density
H_2	Using relative values of code churn predictors is better than using direct (absolute) values to explain the system defect density
H_3	Relative code churn measures can be used as efficient predictors of system defect density.
H_4	Relative code churn measures can be used to discriminate between fault and not fault-prone binaries.

The organization of this paper is as follows. Section 2 describes the related work. Section 3 explains data collection and section 4 the relative code churn measures. Section 5 presents the case study and the observed results. Section 6 discusses our conclusions and future work.

2. RELATED WORK

Prior analyses on predicting defect density used code churn measures as part of a larger set of metrics. Code churn measures have not been studied in isolation as predictors of software defect density. The background work presented below is from studies that involved industrial software systems. The source code base of W2k3 is two orders of magnitude larger than the largest example considered below.

Munson et al. [17] observe that as a system is developed, the relative complexity of each program module that has been altered (or churned) also will change. The rate of change in relative complexity serves as a good index of the rate of fault injection. They studied a 300 KLOC (thousand lines of code) embedded real time system with 3700 modules programmed in C. Code churn metrics were found to be among the most highly correlated with problem reports [17].

Khoshgoftaar et al.[13] define debug churn as the number of lines of code added or changed for bug fixes. Their objective was to identify modules where debug code churn exceeds a threshold, in order to classify the modules as fault-prone. They studied two consecutive releases of a large legacy system for telecommunications. The system contained over 38,000 procedures in 171 modules. Discriminant analysis identified fault-prone modules based on 16 static software product metrics. Their model when used on the second release showed a type I and II misclassification rate of 21.7%, 19.1% respectively and an overall misclassification rate of 21.0%.

Ohlsson et al. [19] identify fault-prone modules by analyzing legacy software through successive releases. They use a total of 28 measures, twelve of which are based on size and change measures. These measures were used to identify 25 percent of the most fault-prone components successfully.

Karunanithi [12] uses a neural network approach for software reliability growth modeling in the presence of continuous code churn, which he shows improves over the traditional time-domain based models. Similarly Khoshgoftaar et al. [15] use code churn as a measure of software quality in a program of 225,000 lines of assembly language. Using eight complexity measures, including code churn, they found neural networks and multiple regression to be an efficient predictor of software quality, as measured by gross change in the code. They suggest that using neural networks may not work in all environments and the results obtained are environment specific. Neural networks can be used for improving software maintenance [15].

Ostrand et al. [20] use information of file status such as new, changed, unchanged files along with other explanatory variables such as lines of code, age, prior faults etc. as predictors in a negative binomial regression equation to predict the number of faults in a multiple release software system. The predictions made using binomial regression model were of a high accuracy for faults found in both early and later stages of development. [20]

Closely related to our study is the work performed by Graves et al. [9] on predicting fault incidences using software change history. Several statistical models were built based on a weighted time damp model using the sum of contributions from all changes to a module in its history. The most successful model computes the fault potential by summing contributions from changes to the module, where large and/or recent changes contribute the most to fault potential [9]. This is similar to our approach of using relative measures to predict fault potential.

Drawing general conclusions from empirical studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted [2]. Researchers become more confident in a theory when similar findings emerge in different contexts [2]. Towards this end we intend that our case study contributes towards strengthening the existing empirical body of knowledge in this field [7, 9, 13, 15, 17, 19, 20].

3. DATA COLLECTION

The baseline used for measuring the code churn and other measures described below is Windows Server 2003 (W2k3). We measured churn between this baseline and Windows Server 2003 Service Pack 1 (W2k3-SP1). We sometimes refer to W2k3-SP1 as the "new version" of the code. Service packs are a means by which product updates are distributed[1]. Service packs contain updates for system reliability, program compatibility, security, etc. that are conveniently bundled for easy downloading.

The size of the code base analyzed is 44.97 *million* LOC (44,970 KLOC). This consisted of 2465 binaries which were compiled from 96,189 files. Some files contribute to more than one binary. As defects for W2k3-SP1 are reported at the binary level, we relate churn to defects at the level of binaries.

[1] http://support.microsoft.com/

The absolute measures and methods of data collection are described below:

- **Total LOC** is the number of lines of non-commented executable lines in the files comprising the new version of a binary. Internal Microsoft tools were used to compute this measure.
- **Churned LOC** is the sum of the added and changed lines of code between a baseline version and a new version of the files comprising a binary.
- **Deleted LOC** is the number of lines of code deleted between the baseline version and the new version of a binary. The churned LOC and the deleted LOC are computed by the version control systems using a file comparison utility like diff.
- **File count** is the number of files compiled to create a binary.
- **Weeks of churn** is the cumulative time that a file was opened for editing from the version control system.
- **Churn count** is the number of changes made to the files comprising a binary between the two versions (W2k3 and W2k3-SP1).
- **Files churned** is the number of files within the binary that churned.

4. RELATIVE CODE CHURN MEASURES

In this section we describe our relative code churn measures. The churn measures are denoted by the elements M1-M8. The elements and their relationship to defect density are explained below (these relationships are verified in section 5.1):

- ***M1: Churned LOC / Total LOC.*** We expect the larger the proportion of churned (added + changed) code to the LOC of the new binary, the larger the magnitude of the defect density for that binary will be.
- ***M2: Deleted LOC / Total LOC.*** We expect the larger the proportion of deleted code to the LOC of the new binary, the larger the magnitude of the defect density for that binary will be.
- ***M3: Files churned / File count.*** We expect the greater the proportion of files in a binary that get churned, the greater the probability of these files introducing defects. For e.g. suppose binaries A and B contain twenty files each. If binary A has five churned files and binary B has two churned files, we expect binary A to have a higher defect density.
- ***M4: Churn count / Files churned.*** Suppose binaries A and B have twenty files each and also have five churned files each. If the five files in binary A are churned twenty times and the five files in binary B are churned ten times, then we expect binary A to have a higher defect density. M4 acts as a cross check on M3.
- ***M5: Weeks of churn / File count.*** M5 is used to account for the temporal extent of churn. A higher value of M5 indicates that it took a longer time to fix a smaller number of files. This may indicate that the binary contains complex files that may be hard to modify correctly. Thus, we expect that an increase in M5 would be accompanied by an increase in the defect density of the related binary.
- ***M6: Lines worked on / Weeks of churn:*** The measure "Lines worked on" is the sum of the churned LOC and the deleted LOC. M6 measures the extent of code churn over time in order to cross check on M5. Weeks of churn does not necessarily indicate the amount of churn. M6 reflects our expectation that the more lines are worked on, the longer the weeks of churn should be. A high value of M6 cross checks on M5 and should predict a higher defect density.
- ***M7: Churned LOC / Deleted LOC.*** M7 is used in order to quantify new development. All churn is not due to bug fixes. In feature development the lines churned is much greater than the lines deleted, so a high value of M7 indicates new feature development. M7 acts as a cross check on M1 and M2, neither of which accurately predicts new feature development.
- ***M8: Lines worked on / Churn count:*** We expect that the larger a change (lines worked on) relative to the number of changes (churn count), the greater the defect density will be. M8 acts as a cross check on M3 and M4, as well as M5 and M6. With respect to M3 and M4, M8 measures the amount of actual change that took place. M8 cross checks to account for the fact that files are not getting churned repeatedly for small fixes. M8 also cross checks on M5 and M6 to account for the fact that the higher the value of M8 (more lines per churn), the higher is the time (M5) and lines worked on per week (M6).). If this is not so then a large amount of churn might have been performed in a small amount of time, which can cause an increased defect density.

Figure 1 illustrates the cross check relationships of these relative code churn measures. As discussed above M1, M2 and M7 cross check on each other and M8 cross checks on the set of M3, M4 and M5, M6. All these measures triangulate on their respective dependent measures with the goal of providing the best possible estimate of defect density with a minimum inflation in the estimation.

5. CASE STUDY

We now describe the case study performed at Microsoft. Section 5.1 presents the correlation analysis between the relative code churn measures and system defect density. Section 5.2 details the model building activities and Section 5.3 the predictive ability of the models. Section 5.4 discusses the discriminative power of the relative code churn measures and Section 5.5 the limitations of the study.

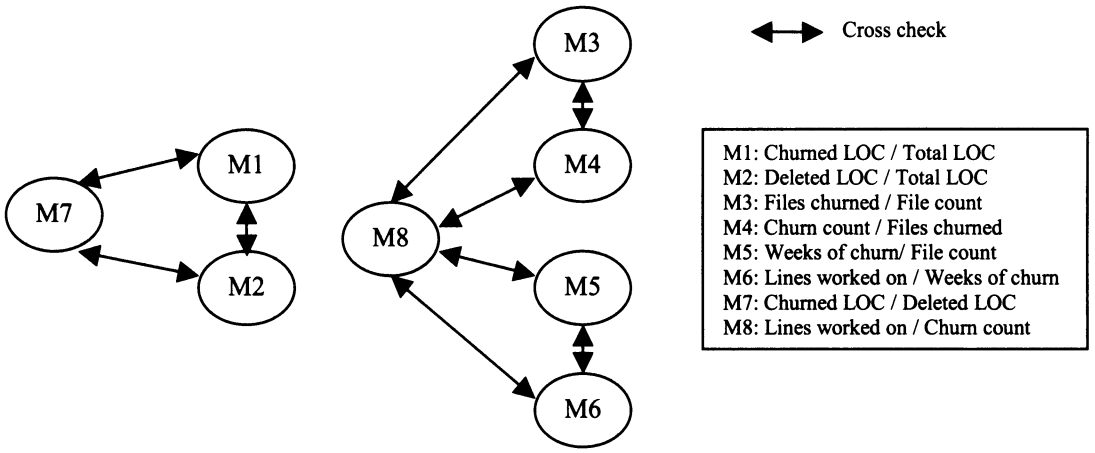

Figure 1. Relative Churn Measure Cross Check Relationships

Table 2. Cross Correlations. All correlations are significant at the 0.01 (99%) level (2-tailed).

		M1	M2	M3	M4	M5	M6	M7	M8	Defects /KLOC
M1	ρ	1.000	.834	.795	.413	.707	.651	.466	.588	**.883**
M2	ρ		1.000	.645	.553	.747	.446	.219	.492	**.798**
M3	ρ			1.000	.186	.749	.434	.445	.269	**.868**
M4	ρ				1.000	.531	.429	.210	.631	**.288**
M5	ρ					1.000	.263	.201	.390	**.729**
M6	ρ						1.000	.701	.843	**.374**
M7	ρ							1.000	.507	**.288**
M8	ρ								1.000	**.262**
Defects/ KLOC	ρ									1.000

As mentioned before, the system defect density for W2k3-SP1 was collected at the level of binaries. That is, for each binary we have a count of the number of defects assigned to that binary.

Throughout the rest of the paper we assume a statistical significance at 99% confidence (level of significance ($\alpha = 0.01$)).

5.1 Correlation Analysis

Our goal is to verify that with an increase in the code churn measures (M1-M8) there is a statistically significant increase in the defects/KLOC. Table 2 shows the Spearman rank correlation (ρ) among the defects/KLOC and the relative code churn measures. Spearman rank correlation is a commonly-used robust correlation technique [8] because it can be applied even when the association between elements is non-linear.

Table 2 shows that there exists a statistically significant (at 99% confidence) positive relationship between the measures and the defects/KLOC (shown in bold). Thus, with an increase in the relative churn measures there is a corresponding positive increase in the defects/KLOC. This is indicated by the statistically significant positive Spearman rank correlation coefficient . From the above observations we conclude that *an increase in relative code churn measures is accompanied by an increase in system defect density (H_1).*

In order to illustrate the cross checks better consider the measures M1, M2 and M7 in Figure 2 with their Spearman rank correlation coefficients from Table 2.

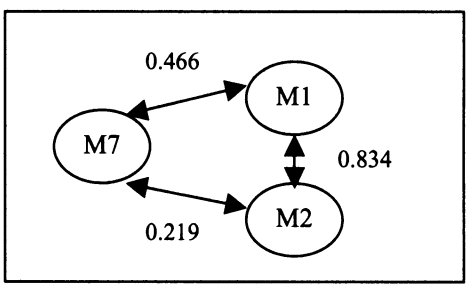

Figure 2: Cross Correlation Relationships

The Spearman correlation coefficient of 0.834 between M1 and M2 indicates that there is a very strong correlation between the two measures. But this might not be the case when there is a higher proportion of churned code compared to deleted code (as measured by M7 for new feature development). Since this cannot be measured by M1 or M2, M7 acts as a cross check on them. The correlation between M1 and M7 (0.466) indicates when there is a

new feature addition there is a corresponding increase in the churned code. For M2 and M7 this correlation is not as strong (but is statistically significant) because there were relatively fewer new feature additions compared to other changes in the W2k3-SP1 source base.

5.2 Model Fitting

We now compare predictive models built using absolute measures against those built using the relative churn measures. For the absolute model, defects/KLOC is the dependent variable and the predictors are the absolute measures described in Section 3. For the relative model, defects/KLOC is the dependent variable and the predictors are the relative measures described in Section 4.

R^2 is a measure of variance in the dependent variable that is accounted for by the model built using the predictors [4]. R^2 is a measure of the fit for the given data set. (It cannot be interpreted as the quality of the dataset to make future predictions). The adjusted R^2 measure also can be used to evaluate how well a model will fit a given data set [5]. Adjusted R^2 explains for any bias in the R^2 measure by taking into account the degrees of freedom of the predictor variables and the sample population. The adjusted R^2 tends to remain constant as the R^2 measure for large population samples.

The multiple regression model fit for absolute measures using all the predictors has an R^2 value of 0.052 ($F=16.922$, $p<0.0005$). (The F-ratio is used to test the hypothesis that all regression coefficients are zero). This is a poor fit of the data and irrespective of other transformations (like for e.g. log) we cannot get a marked improvement in R^2. The adjusted R^2 value for the absolute measures is 0.49. Throughout the rest of this paper we present the adjusted R^2 values in addition to the R^2 measures in order to eliminate any bias in model building. But with respect to the large sample size (2465 binaries) the adjusted R^2 and R^2 value show only minor variation, not sufficient enough to drop the R^2 value and employ the adjusted R^2 value.

There are different ways in which regression models [16] can be built. Three common regression methods [16] are forward, backward and step-wise regression. In forward regression, one adds a single predictor at a time to the model based on the strength of its correlation with the dependent variable. The effect of adding each predictor is evaluated based on the results of an F-ratio test [16]. Variables that do not significantly add to the success of the model are excluded. In backward regression, a model is built using all the predictors. The weakest predictor variable is removed and the strength of the overall built model is assessed similar to the forward regression procedure. If this significantly weakens the model then the predictor is put back (and otherwise removed). Step-wise regression [16] is the more robust technique of these methods. The initial model consists of the predictor having the single largest correlation with the dependent variable. Subsequently, new predictors are selected for addition into the model based on their partial correlation with the predictors already in the model. With each new set of predictors, the model is evaluated and predictors that do not significantly contribute towards statistical significance in terms of the F-ratio are removed so that, in the end, the best set of predictors explaining the maximum possible variance is left.

A step-wise regression analysis using the absolute set of predictors does not lead to any significant change in the R^2 values (=0.051) (adjusted $R^2 = 0.050$). Only the LOC and the number of times a file is churned are kept as predictors. This further confirms the fact that using the absolute measures is not an appropriate method for assessing the system defect density.

Several empirical studies use Principal Component Analysis (PCA) [10] to build regression models [6]. In PCA a smaller number of uncorrelated linear combinations of metrics, which account for as much sample variance as possible, are selected for use in regression. PCA is not a possible solution when using absolute measures because the correlation matrix is not positive definite. We still use the two principal components generated to build a multiple regression equation. The multiple regression equation constructed has an even lower value of R^2=0.026, ($F=33.279$, $p<0.0005$).

Based on the three results discussed above (multiple regression using all the predictors, step-wise regression and PCA) we conclude that the absolute measures are not good predictors of system defect density.

As outlined in Section 3 we calculate the relative code churn measures (M1-M8) and build regression models using all the measures, step-wise regression and PCA. Table 3 shows the R^2 value of the regression equation built using all the measures. We also present the adjusted R^2 value and the root MSE (Mean Squared Error).

Table 3. Regression Fit Using All Measures

Model	R^2	Adjusted R^2	Root MSE
All Measures	.811	.811	1.301215

Table 4 shows how the R^2 value changes in step-wise regression for all the models built during that process. In the step-wise regression model the measure M7 is dropped. The best R^2 value in Table 4 (without M7) is the same as that of Table 3 (.811) but there is a change in the third decimal place of the standard error of the estimate. M7 probably was dropped because there were relatively fewer new feature additions compared to other changes in the W2k3-SP1 source base. The adjusted R^2 values are also shown but are not significantly different from the R^2 values due to the large sample size used to build the models.

Table 4. Step-wise Regression Models

Model	R-Square	Adjusted R-Square	Root MSE
(a)	.592	.592	1.908727
(b)	.685	.685	1.677762
(c)	.769	.769	1.437246
(d)	.802	.801	1.331717
(e)	.808	.807	1.312777
(f)	.810	.809	1.305817
(g)	.811	.811	1.300985

a Predictors: (Constant), M2
b Predictors: (Constant), M2, M3
c Predictors: (Constant), M2, M3, M8
d Predictors: (Constant), M2, M3, M8, M1
e Predictors: (Constant), M2, M3, M8, M1, M6
f Predictors: (Constant), M2, M3, M8, M1, M6, M5
g Predictors: (Constant), M2, M3, M8, M1, M6, M5, M4.

The PCA of the eight relative code churn measures yields three principal components. PCA can account for the multicollinearity among the measures, which can lead to inflated variance in the estimation of the defect density.

But for PCA to be applicable the KMO (Kaiser-Meyer-Olkin) measure[11] of sampling adequacy should be greater than 0.6 [4]. The KMO measure of sampling adequacy is a test of the amount of variance within the data that can be explained by the measures. The KMO measure of the eight relative code churn measures is 0.594 which indicates that PCA might not be an appropriate method to apply.

We still perform the analysis to investigate and present those results as well on a comparative basis. The results for all three models are summarized in Table 5.

Table 5. Relative Measures Model Fits

Model	R^2	Adjusted R^2	F-Test sig.
All measures	0.811	0.811	1318.44, ($p<0.0005$)
Step-wise regression	0.811	0.811	1507.31, ($p<0.0005$)
PCA	0.749	0.748	2450.89, ($p<0.0005$)

From the above results we can see that *using relative values of code churn predictors is better than using absolute values to explain the system defect density (H_2).*

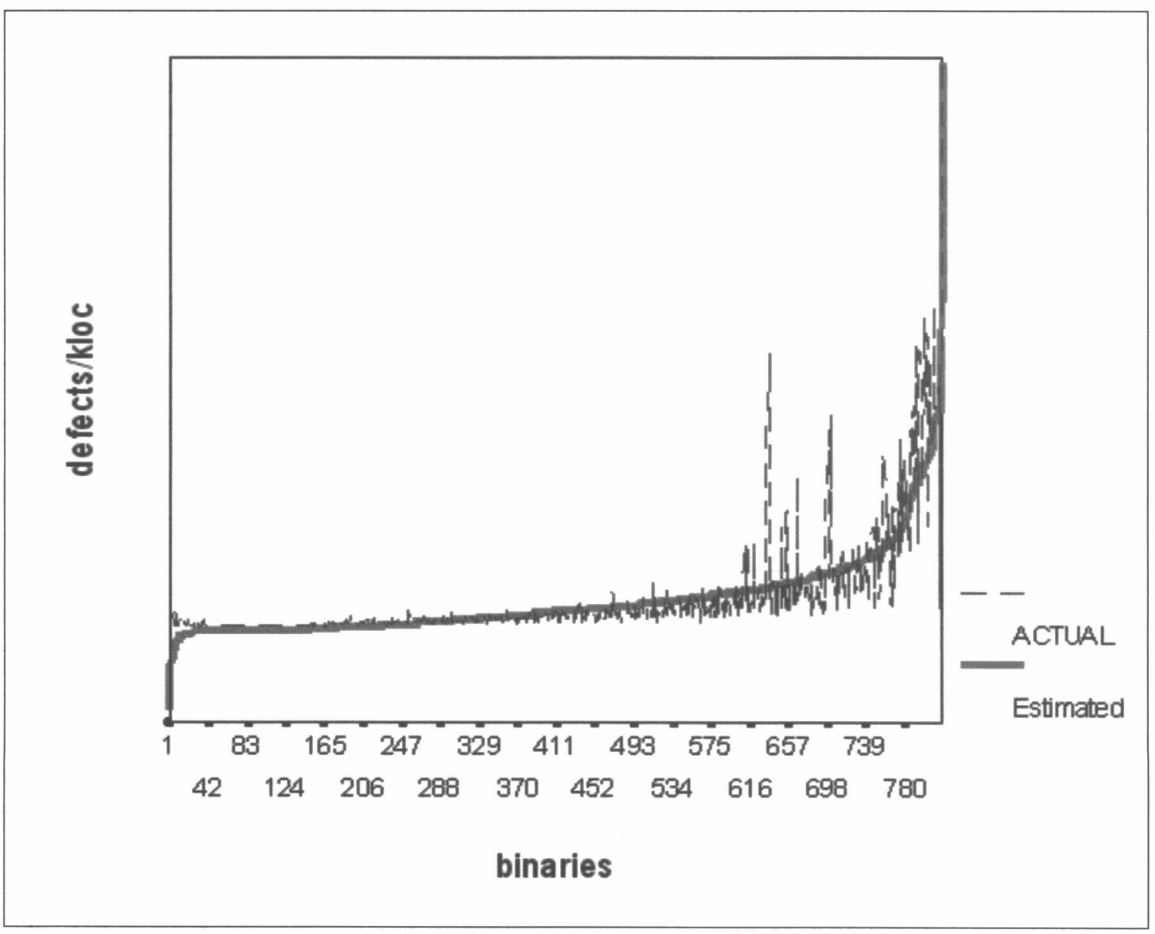

Figure 3: Actual vs. Estimated System Defect Density

5.3 Defect Density Prediction

We use the technique of data splitting [18] to measure the ability of the relative code churn measures to predict system defect density. The data splitting technique was employed to get an independent assessment of how well the defect density can be estimated from a population sample. We randomly select two thirds of the binaries (1645) to build the prediction model and use the remaining one third (820) to verify the prediction accuracy. We constructed models using all the measures, step-wise regression and PCA (for purpose of completeness). Table 6 shows the results for these models.

Table 6. Regression Data Fit

Model	R^2	Adjusted R^2	F-Test sig.
All measures	0.821	0.820	938.304, (p<0.0005)
Step-wise regression (M7 dropped)	0.821	0.820	1072.975, (p<0.0005)
PCA	0.762	0.761	1749.113, (p<0.0005)

Using the fitted regression equation we estimate the system defect density for the remaining 820 binaries. Figure 3 shows the estimated and actual defect density using the regression equation constructed using all the measures (sorted by estimated defect density). The estimated defect density is shown by the thicker continuous line. From the graph we can see that the estimated defect density is similar to the actual defect density. The axes on the graphs are removed in order to protect proprietary data

To quantify the sensitivity of prediction, we run a correlation analysis between the estimated and actual values. A high positive correlation coefficient indicates that with an increase in the actual defect density there is a corresponding positive increase in the estimated defect density. We perform Pearson and Spearman correlations to indicate their sensitivity. The Pearson correlation indicates a linear relationship. The Spearman correlation is a more robust correlation technique.

Table 7 shows that the correlations are all positive and statistically significant. The magnitude of the correlations indicates the sensitivity of the predictions (the stronger the correlations the more sensitive are the predictions). The models built using all the measures and the step-wise method have the same sensitivity and are better than the model built using PCA.

Table 7. Correlation Results

Model	Pearson (sig.)	Spearman (sig.)
All measures	0.889 (p<0.0005)	0.929 (p<0.0005)
Step-wise regression	0.889 (p<0.0005)	0.929 (p<0.0005)
PCA	0.849 (p<0.0005)	0.826 (p<0.0005)

Analyses that are based on a single dataset that use the same data to both estimate the model and to assess its performance can lead to unreasonably negative biased estimates of sampling variability. In order to address this we repeat the random sampling with 3 different random samples to verify if the above results are repeatable. For each sample the model is fit with 1645 binaries to build the model. Table 8 shows the fit of the various models built for each sample.

Table 8. Random Splits Data Fit

Model	R^2	Adjusted R^2	F-Test (Sig.)
Random 1: All	0.836	0.835	1045.07, (p<0.0005)
Random 1: Stepwise (drop none)	0.836	0.835	1045.07, (p<0.0005)
Random 1: PCA	0.757	0.756	1701.98, (p<0.0005)
Random 2: All	0.822	0.821	941.86, (p<0.0005)
Random 2: Stepwise (drop M4)	0.821	0.820	1074.05, (p<0.0005)
Random 2: PCA	0.765	0.764	1776.87, (p<0.0005)
Random 3: All	0.799	0.798	813.12, (p<0.0005)
Random 3: Stepwise (drop M7)	0.799	0.798	927.54, (p<0.0005)
Random 3: PCA	0.737	0.736	1529.25, (p<0.0005)

Using each of the above predictive models we calculate the estimated defect density for the remaining 820 binaries. Table 9 shows the correlation between the estimated and the actual defect density.

Table 9. Correlation Between Actual and Estimated Defects/KLOC

Model	Pearson Correlation (sig.)	Spearman Correlation (sig.)
Random 1: All	0.873 (p<0.0005)	0.931 (p<0.0005)
Random 1: Stepwise	0.873 (p<0.0005)	0.931 (p<0.0005)
Random 1: PCA	0.858 (p<0.0005)	0.836 (p<0.0005)
Random 2: All	0.878 (p<0.0005)	0.917 (p<0.0005)
Random 2: Stepwise	0.876 (p<0.0005)	0.906 (p<0.0005)
Random 2: PCA	0.847 (p<0.0005)	0.825 (p<0.0005)
Random 3: All	0.899 (p<0.0005)	0.892 (p<0.0005)
Random 3: Stepwise	0.901 (p<0.0005)	0.893 (p<0.0005)
Random 3: PCA	0.880 (p<0.0005)	0.818 (p<0.0005)

Based on the consistent positive and statistically significant correlations, indicating the sensitivity of predictions obtained in Table 9 we can say *that relative code churn measures can be used as efficient predictors of system defect density (H_3).*

Our results demonstrate it is effective to use all eight measures rather than dropping any of them from the predictive equation. Each of these measures cross check on each other and any

abnormal behavior in one of the measures (for e.g. like a file getting churned too many times) would be immediately highlighted.

By interchanging the measures in a model equation we can get estimated values for all the relative measures independently. For example, in order to determine the maximum allowable code churn with respect to the file size (i.e. M1), say for a particular software model we fix the maximum allowable system defect density. We then can build a regression model with M2-M8 and defect density as predictors and M1 as the dependent variable.

5.4 Discriminant Analysis

Discriminant analysis, is a statistical technique used to categorize programs into groups based on the metric values. It has been used as a tool for the detection of fault-prone programs [13, 14, 18]. The ANSI-IEEE Std. [1] defines a *fault* as an accidental condition that causes a functional unit to fail to perform its required function. We use discriminant analysis to identify binaries as fault-prone or not fault-prone. To classify if a binary is fault-prone or not we use the system defect density in a normal confidence interval calculation as shown in equation 1.

$$LB = \mu_x - z_{\alpha/2} * \frac{\text{Standard deviation of defect density}}{\sqrt{n}} \quad \ldots (1)$$

where

- LB is the lower bound on system defect density;
- μ_x is the mean of defect density;
- $Z_{\alpha/2}$ is the upper $\alpha/2$ quantile of the standard normal distribution;
- n is the number of observations.

We conservatively classify all binaries that have a defect density less than LB as not fault-prone and the remaining as fault-prone. Table 10 shows the eigenvalue and overall classification ability of using the eight measures and the three principal components. The eigenvalue is a measure of the discriminative ability of the discriminant function. The higher the eigenvalue the better is the discriminative ability. For all measures, the function correctly classifies nearly nine out of every ten binaries.

Table 10. Overall Discriminant Function Fit

Model	Eigenvalue	Classification ability
All Measures	1.025	2188/2465 (88.8%)
PCA	0.624	2195/2465 (89.0%)

As before, we split the data set into 1645 programs to build the discriminant function and the remaining 820 binaries to verify the classification ability of the discriminant function. We perform this analysis using all the measures and the principal components. The results of this fit and classification are shown below in table 11.

Table 11. Discriminant Analysis

	For Model Fit (for 1645 binaries to build the model)		For Test Data (820 binaries)
Model	**Eigen value**	**Classification ability**	**Classification ability**
All Measures	1.063	1464/1645 (90.0%)	735/820 (89.6%)
PCA	0.601	1461/1645 (88.8%)	739/820 (90.1%)

Table 11 shows that the relative code churn measures have effective discriminant ability (comparable to prior studies done on industrial software [13]). We conclude that *relative code churn measures can be used to discriminate between fault and not fault-prone binaries (H_4).*

5.5 Limitations of Study

Internal validity. Internal validity issues arise when there are errors in measurement. This is negated to an extent by the fact that the entire data collection process is automated via the version control systems. However, the version control systems only records data upon developer check-out or check-in of files. If a developer made many overlapping edits to a file in a single check-out/check-in period then a certain amount of churn will not be visible. A developer also might have a file checked out for a very long period of time during which few changes were made, inflating the "weeks of churn" measure.

These concerns are alleviated to some extent by the cross check among the measures to identify abnormal values for any of the measures and the huge size and diversity of our dataset.

In our case study we provide evidence for using all the relative churn measures rather than a subset of values or principal components. This is case study specific and should be refined based on further results.

External validity. External validity issues may arise from the fact that all the data is from one software system (albeit one with many different components) and that the software is very large (some 44 million lines of code) as other software systems used for a similar analysis might not be of comparable size.

6. CONCLUSIONS AND FUTURE WORK

We have shown how relative code churn metrics are excellent predictors of defect density in a large industrial software system. Our case study provides strong support for the following conclusions:

- Increase in relative code churn measures is accompanied by an increase in system defect density;
- Using relative values of code churn predictors is better than using absolute values to explain the system defect density;
- Relative code churn measures can be used as efficient predictors of system defect density; and
- Relative code churn measures can be used to discriminate between fault and not fault-prone binaries.

We plan to validate our approach on other products developed inside Microsoft like SQL Server and Office. We also plan to develop standards for all the measures to provide guidance to the developers on the maximum allowable change. We also plan to investigate how testing can more effectively be directed towards churned code.

ACKNOWLEDGEMENTS

We would like to express our appreciation to Brendan Murphy of Microsoft Research for providing the Windows Server 2003 SP1 data set. We would like to thank Madan Musuvathi of Microsoft Research, for critical feedback on the relative churn measures. We would like to thank Jim Larus of Microsoft Research, Laurie Williams, Jason Osborne of North Carolina State University for

reviewing initial drafts of this paper and the anonymous referees for their thoughtful comments on an earlier draft of this paper.

REFERENCES

[1] ANSI/IEEE, "IEEE Standard Glossary of Software Engineering Terminology, Standard 729," 1983.

[2] Basili, V., Shull, F.,Lanubile, F., "Building Knowledge through Families of Experiments," *IEEE Transactions on Software Engineering*, Vol. Vol. 25, No.4, No., 1999.

[3] Boehm, B. W., *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[4] Brace, N., Kemp, R., Snelgar, R., *SPSS for Psychologists*: Palgrave Macmillan, 2003.

[5] Brito e Abreu, F., Melo, W., "Evaluating the Impact of Object-Oriented Design on Software Quality," Proceedings of Third International Software Metrics Symposium, 1996, pp. 90-99.

[6] Denaro, G., Pezze, M., "An Empirical Evaluation of Fault-Proneness Models," Proceedings of International Conference on Software Engineering, 2002, pp. 241 - 251.

[7] Fenton, N. E., Ohlsson, N., "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, Vol. 26, No. 8, pp. 797-814, 2000.

[8] Fenton, N. E., Pfleeger, S.L., *Software Metrics*. Boston, MA: International Thompson Publishing, 1997.

[9] Graves, T. L., Karr, A.F., Marron, J.S., Siy, H., "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, Vol. 26, No. 7, pp. 653-661, 2000.

[10] Jackson, E. J., *A User's Guide to Principal Components*: John Wiley & Sons, Inc., 1991.

[11] Kaiser, H. F., "An Index of Factorial Simplicity," *Psychometrika*, Vol. 39, No., pp. 31-36, 1974.

[12] Karunanithi, N., "A Neural Network approach for Software Reliability Growth Modeling in the Presence of Code Churn," Proceedings of International Symposium on Software Reliability Engineering, 1993, pp. 310-317.

[13] Khoshgoftaar, T. M., Allen, E.B., Goel, N., Nandi, A., McMullan, J., "Detection of Software Modules with high Debug Code Churn in a very large Legacy System," Proceedings of International Symposium on Software Reliability Engineering, 1996, pp. 364-371.

[14] Khoshgoftaar, T. M., Allen, E.B., Kalaichelvan, K.S., Goel, N., Hudepohl, J.P., Mayrand, J., "Detection of fault-prone program modules in a very large telecommunications system," Proceedings of International Symposium Software Reliability Engineering, 1995, pp. 24-33.

[15] Khoshgoftaar, T. M., Szabo, R.M., "Improving Code Churn Predictions During the System Test and Maintenance Phases," Proceedings of IEEE International Conference on Software Maintainence, 1994, pp. 58-67.

[16] Kleinbaum, D. G., Kupper, L.L., Muller, K.E., *Applied Regression Analysis and Other Multivariable Methods*. Boston: PWS-KENT Publishing Company, 1987.

[17] Munson, J. C., Elbaum, S., "Code Churn: A Measure for Estimating the Impact of Code Change," Proceedings of IEEE International Conference on Software Maintenance, 1998, pp. 24-31.

[18] Munson, J. C., Khoshgoftaar, T.M., "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, Vol. 18, No. 5, pp. 423-433, 1992.

[19] Ohlsson, M. C., von Mayrhauser, A., McGuire, B., Wohlin, C., "Code Decay Analysis of Legacy Software through Successive Releases," Proceedings of IEEE Aerospace Conference, 1999, pp. 69-81.

[20] Ostrand, T. J., Weyuker, E.J, Bell, R.M., "Where the Bugs Are," Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2004, pp. 86-96.