Does measuring code change improve fault prediction?

Conference Paper · September 2011

DOI: 10.1145/2020390.2020392 · Source: DBLP

3 authors, including:

Thomas J. Ostrand
Mälardalen University
70 PUBLICATIONS **4,593** CITATIONS
SEE PROFILE

Elaine J. Weyuker
University of Central Florida
194 PUBLICATIONS **10,626** CITATIONS
SEE PROFILE

Does Measuring Code Change Improve Fault Prediction?

Robert M. Bell, Thomas J. Ostrand, Elaine J. Weyuker
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
(rbell,ostrand,weyuker)@research.att.com

ABSTRACT

Background: Several studies have examined code churn as a variable for predicting faults in large software systems. High churn is usually associated with more faults appearing in code that has been changed frequently.

Aims: We investigate the extent to which faults can be predicted by the degree of churn alone, whether other code characteristics occur together with churn, and which combinations of churn and other characteristics provide the best predictions. We also investigate different types of churn, including both additions to and deletions from code, as well as overall amount of change to code.

Method: We have mined the version control database of a large software system to collect churn and other software measures from 18 successive releases of the system. We examine the frequency of faults plotted against various code characteristics, and evaluate a diverse set of prediction models based on many different combinations of independent variables, including both absolute and relative churn.

Results: Churn measures based on counts of lines added, deleted, and modified are very effective for fault prediction. Individually, counts of adds and modifications outperform counts of deletes, while the sum of all three counts was most effective. However, these counts did not improve prediction accuracy relative to a model that included a simple count of the number of times that a file had been changed in the prior release.

Conclusions: Including a measure of change in the prior release is an essential component of our fault prediction method. Various measures seem to work roughly equivalently.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging – *Debugging aids*

General Terms: Experimentation

Keywords: software faults, fault prediction, code churn, fault-percentile average, empirical study

1. INTRODUCTION

Several researchers have noted that the amount of code change in some part of the code of a software system can be an effective indicator of the fault-proneness of that part, either in the modified release or in succeeding releases. This change is often referred to as *churn*, and we will use that term to refer to any sort of change in the remainder of the paper.

For example, we have found that the number of times that a file has been changed in the previous two releases is a powerful indicator of fault-proneness in the negative binomial prediction models that we have investigated [11, 12]. In the present paper, we examine finer-grained measurements of churn, including the counts of numbers of lines within a file that have been added, deleted, or modified in previous releases.

The studies described in this paper are performed on 18 quarterly releases spanning approximately five years of a provisioning system. In previous work, we have referred to this system as *System 7* [10]. It includes files written in six different programming languages including C, C++, Java, SQL, a proprietary variant of C, a proprietary variant of C++, as well as .h files. About 60% of the files were written in Java with the remaining 40% of the files roughly equally distributed among the other six file types. Table 1 provides basic data on the size of the system's releases, the occurrence of faults, and the percent of each release that was changed. The first 16 rows of the table are updated from the data that appeared in [10]. The present table includes one file type (`.h files`) that was not included in the previous paper and two additional releases, as well as updates to the fault and change counts due to the ongoing development and maintenance of the system.

2. MEASURING CHANGE

Changes to a file can be measured in many ways in addition to simply determining whether or not the file was changed during a release. In this paper we consider ways of quantifying the degree of change, in order to see whether that added information is helpful when predicting fault-proneness. In our previous work [11, 12, 13], we measured the number of changes within a release, that is, the number of times that the file had gone through a check-out/check-in cycle.

The number of changes in Releases N-1 and N-2 are two attributes that are part of our Standard Model to predict fault-proneness of files in Release N. We refer to these in the sequel as Prior Changes and Prior-Prior Changes re-

Release	Files	KLOC	% New files	% Changed files	% Faulty files	Faults	Faults/file	Faults/KLOC
1	2470	885.3	100.0	.0	5.6	284	.11	.32
2	2487	912.3	.7	10.2	3.7	168	.07	.18
3	2507	924.8	.8	7.5	5.1	404	.16	.44
4	2569	943.8	2.4	8.4	4.9	243	.09	.26
5	2610	966.0	1.6	15.1	4.4	192	.07	.20
6	2707	989.3	3.6	10.5	5.1	288	.11	.29
7	2872	1034.8	5.8	8.1	6.8	580	.20	.56
8	3005	1082.0	4.4	14.6	5.7	355	.12	.33
9	3159	1122.6	4.9	13.2	7.0	453	.14	.40
10	3506	1229.9	9.9	18.9	9.1	689	.20	.56
11	3552	1288.2	1.3	22.9	6.7	576	.16	.45
12	3517	1293.8	.1	11.2	2.5	161	.05	.12
13	3713	1364.0	5.3	2.9	6.1	715	.19	.52
14	3769	1408.1	1.5	17.9	4.8	454	.12	.32
15	3797	1441.9	.7	10.5	4.5	389	.10	.27
16	3829	1466.4	.8	8.5	4.0	352	.09	.24
17	3891	1489.0	1.6	6.8	3.4	275	.07	.18
18	3920	1520.1	.7	8.2	4.0	522	.13	.34
Average			6.8	11.0	5.2	7100 (total)	.12	.33

Table 1: Size, Faults, and Changes for System 7 (Provisioning System)

spectively, and note here that all model variables, with the exception of size, relate to prior releases and are named accordingly. The other attributes in the Standard Model are the count of faults in Release N-1, the size of the files in 1000's of lines of code (KLOC), the age of the file in terms of the number of previous releases the file was part of the system, and the file type.

In this paper, we use finer-grained change information to measure the volume of the editing changes that occur each time a user checks out the latest version of the file, edits it, and then checks in the edited version. Specifically, we consider three types of changes that can occur within an editing session: *lines added* to the file, *lines deleted*, and *lines modified*. The definitions we use are as follows.

Changes are always measured for contiguous sets of lines in the file. If a user edits two groups of lines that are separated by at least one unchanged line, that is considered two separate actions, and the following definitions must be applied separately to both of the actions.

If an editing action involves only adding lines or only deleting lines, the result is simply the count of the number of lines added or deleted. If an action involves modifying individual lines without inserting or removing additional lines, the result is again simply the number of lines that are modified. If an action involves replacing k contiguous lines with $k + i$ lines (k and i both positive integers), then the result is k lines modified and i lines added. If an action involves replacing k contiguous lines with $k - i$ lines (k and i both positive integers, and $k > i$), then the result is $k - i$ lines modified and i lines deleted.

It is possible to have alternative views of the changes that are made in going from the checked-out version to the checked-in version. In particular, one could define editing changes only in terms of adds and deletes. For example, the replacement of k lines with $k + i$ lines could be defined as deleting k and adding $k + i$. However, we believe it is more meaningful to distinguish this presumably single editing action from the situation where a developer performs two potentially unrelated edits, deleting k lines and adding $k + i$ in widely-separated sections of code.

Change	Lines Added	Lines Deleted	Lines Modified
[7] `x+1` $\rightarrow$ [7] `x+2`	0	0	1
[21:30] $\rightarrow$ 1 line	0	9	1
[41:45] $\rightarrow$ 8 lines	3	0	5
[99:100] $\rightarrow$ [99] 12 lines [100]	12	0	0
Total	15	9	7

Table 2: Examples of changes to a file

A single editing session can involve multiple adds, deletes, and modifications. For example, suppose a user checks out a file with 100 lines, and makes the following changes before checking the file back in. The user changes the expression `x+1` in line 7 to `x+2`, replaces lines 21-30 with a single line, replaces the original lines 41-45 with a new set of 8 lines, and inserts 12 new lines just in front of the last line of the file, resulting in a new file with 106 lines. Table 2 shows the way these changes are counted. The numbers in square brackets represent the line numbers in the original checked-out file.

A file may undergo multiple editing sessions in a single release. The churn counts for the file in the release are the sums of the respective adds, deletes, and modifications done in all those sessions.

3. PRELIMINARIES

We start by looking at simple properties of file size and fault occurrence, according to the change status of the files. For each release, we categorize each file's status as being one of: *new* to the system, *changed* from the prior release, or *unchanged*. Figure 1 shows the mean file size, by release, for each change status. At Release 1, we treat all files as new. Consequently, the majority of instances of new files occur at Release 1.

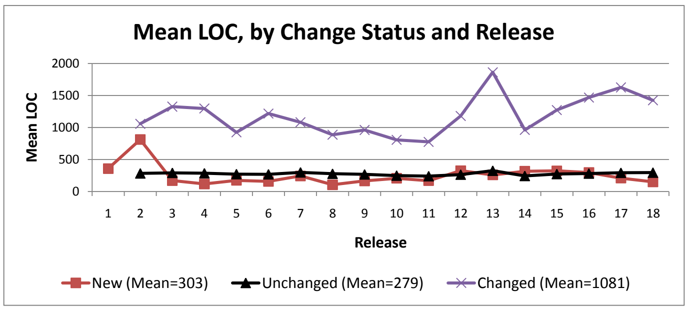

Figure 1: Sizes of files, by change status, Releases 1-18

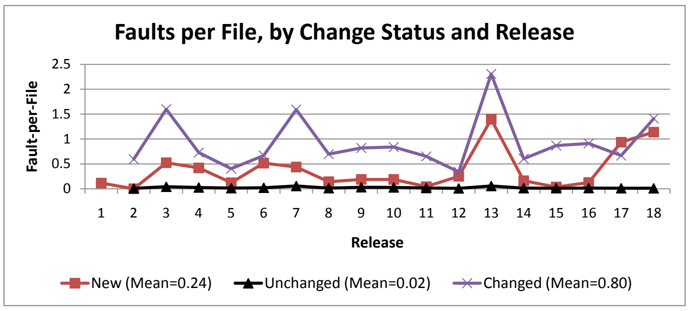

Figure 2: Faults per file, by change status, Releases 1-18

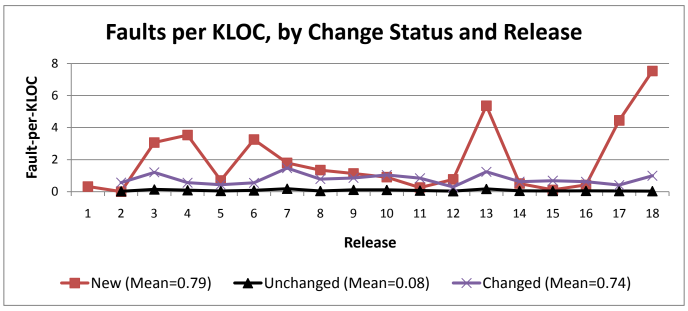

Figure 3: Faults per LOC, by change status, Releases 1-18

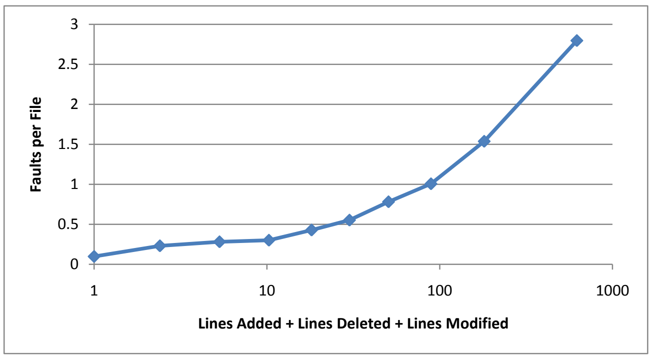

Figure 4: Faults per size of change

Changed files are notably larger than the other types, implying that large files are more likely to be changed. Note that new files introduced after the first two releases were much smaller on average than those created in Releases 1 and 2.

Figure 2 shows the average number of faults per file, by release and change status. Unchanged files always have extremely low fault counts, never exceeding 0.06 faults per file. Fault rates are much higher for the other two types of file, and changed files have higher fault rates than new files in every release from 2 through 18 except Release 17. This is consistent with the data from our previous work, where we noted that files that have been changed in the previous release are much more likely to have faults than unchanged files.

Because the average size of changed files is so much larger than that of new files, the fault rate per KLOC is usually lower for changed files, as seen in Figure 3.

Table 3 summarizes the results in the last three figures by aggregating over release. In addition, the last two columns of the table show the aggregate counts and percentages of faults by change status. Despite being only 11 percent of files overall (Table 1), changed files contain 72 percent of all faults and 75 percent of faults after Release 1.

Going beyond merely the change status of files, Table 4 shows fault rates for changed files, broken down by the types of changes that occurred. As the number of types of changes grows from one to all three, both the average number of lines changed and the subsequent fault rates increase. Performing more than one type of change is more fault-prone than doing a single one, and performing all three types during a single release results in more than 4 times as many faults as doing a single type of change. When only one type of change occurs in a file, Adds are the most fault-prone, Modifications second, and Deletes yield the fewest faults.

Figure 4 shows faults per file, for files changed in the prior release, as a function of the total number of lines changed. Files are sorted into ten bins based on the number of lines changed, with each plotting symbol representing about 600 to 700 points. Not surprisingly, the number of faults occurring in changed files rises with the total number of lines changed. Obviously, the more lines that are changed, the greater the opportunity for mistakes to occur.

4. PREDICTION MODELS AND PREDICTION RESULTS

To determine the effectiveness of different variables for fault prediction, we constructed models containing various combinations of the original variables from the Standard Model, together with subsets of the churn variables. Models to predict fault counts for Release N are constructed using negative binomial regression, with values from Releases 1 through $N-1$ as training data. See [11] for details on the negative binomial regression model for fault prediction as well as for example results.

Models were constructed, applied, and evaluated following our standard procedure, as follows: The dependent output variable is the number of faults for each file in a release. The non-churn independent variables are chosen from a set

File status	Number of files	Average LOC	Faults per file	Faults per KLOC	Number of faults	Percent of faults
New (Release 1)	2470	358	.11	.32	284	4.0
New (Releases 2-18)	1493	210	.44	2.11	661	9.3
Changed	6389	1081	.80	.74	5121	72.1
Unchanged	47528	279	.02	.08	1034	14.6

Table 3: Faults Rates, by File Status

File status	Number of files	Percentage of files	Average LOC	Average Lines changed	Faults per file	Faults per KLOC
Adds only	597	9.4	769	21	.30	.39
Deletes only	296	4.6	513	5	.04	.07
Modifications only	683	10.7	605	4	.19	.32
Adds & Deletes	126	2.0	702	21	.50	.71
Adds & Mods	1894	29.6	940	37	.55	.59
Deletes & Mods	168	2.6	521	23	.36	.69
Adds, Deletes, & Mods	2625	41.1	1495	210	1.38	.92

Table 4: Faults Rates for Changed Files, by Type(s) of Change

that includes *code attributes,* which are obtainable from the release to be predicted, and *history attributes*, which are obtainable from previous releases. The code attributes include LOC, file status, file age, and file type. History attributes include prior changes and prior faults. All models in this paper include release number, implemented as a series of dummy variables for Releases 1 through $N-2$.

4.1 Evaluation

After a model has been constructed, it is applied to generate predictions for Release N using code attributes of Release N and history attributes of the previous two Releases ($N-1$ and $N-2$). The files of Release N are sorted in descending order of the resulting predictions of fault counts. To evaluate a model, we sum the actual number of faults that occur in the files at the top X% of the sorted list, and determine the percentage of all faults in Release N that is included in the top X%. This percentage is the *yield* of the model, relative to the chosen value of X. While X can be any suitable value, we have typically presented results with X=20, as repeated studies on a wide variety of systems have found that 80% or more of the faults are contained within 20% of the system's files. For the systems we have studied, we have frequently found even higher fault concentration. For example, in each of the 18 releases of System 7, all the faults were contained in 9.1% or fewer of the system's files, as shown in Table 1.

In our previous work, the top 20% of the files identified by negative binomial regression models for 6 large systems contained 83% to 95% of the faults, and 76% of the faults in one other system.

To dispel the potential concern that using any particular value of X to evaluate a model may seem arbitrary, we defined the notion of *fault-percentile average*, which essentially averages the top X% figure over all values of X. A full description of the fault-percentile average is in Reference [13]. The mean fault-percentile averages for the systems studied in that paper ranged from 88.1 to 92.8 when the predictions were generated by negative binomial regression using the Standard Model.

While we find the top X% metric to be the most useful for interpreting effectiveness of the prediction models, we prefer fault-percentile average for comparison of alternative models. The top X% metric can be sensitive to whether a few faulty files just make or miss the threshold. In contrast, we have found the fault-percentile average to be more stable.

4.2 Prediction Accuracy for Simple Prediction Models

Figure 5 shows fault-percentile average (FPA) values, by release, for four selected predictor variables, representing the four main components of our prediction models. *LOC* is the total length of the file, including comment lines. To reduce skewness, the value used as a predictor variable in models is *log(KLOC)*. *Language* is a categorical variable representing the file type. *Age* represents the number of prior releases containing the file. *Prior Changes* is a count of the number of times the file was changed in the prior release.

The two most effective predictors are easily log(KLOC) and Prior Changes, which produced very similar FPAs except at Release 13.

Table 5 displays mean FPAs across Releases 3 to 16 for models that use the four predictor variables shown in Figure 5, as well as models based on a number of other measures of churn in the prior release. We note that whether or not a predictor variable was transformed does not matter for this analysis because the FPA metric depends only on the ranking of predictions, not on the actual values. Among the various measures of changes in the prior release, Prior Changes and the total number of lines changed performed best on the FPA metric. A count of Prior Developers is not far behind. Among types of changes, lines Added and Modified are most predictive; lines Deleted performs substantially worse.

Prior Changed is a simple binary indicator for whether any changes were applied to the file in the prior release. It is basically a simplification of Prior Changes, lumping together all files for which Prior Changes > 0. While clearly some signal is lost, the model using Prior Changed performs nearly as well as the Prior Changes model.

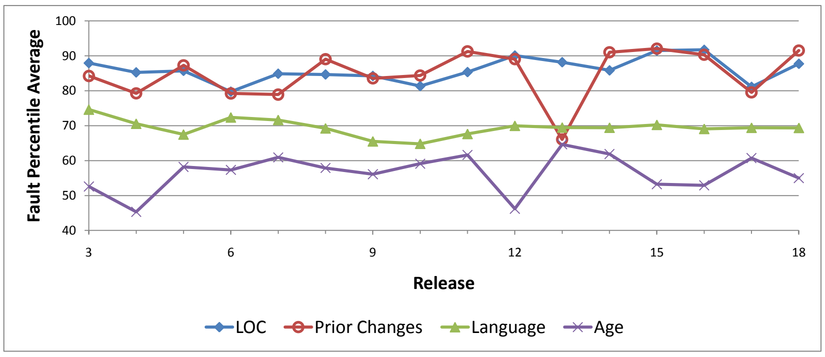

Figure 5: Fault percentile average, for four simple predictor variables

Predictor	Mean FPA
log(KLOC)	85.9
Prior Changes	84.8
Prior Adds+Deletes+Mods	84.7
Prior Developers	84.1
Prior Lines Added	83.5
Prior Lines Modified	82.9
Prior Changed	82.3
Prior Faults	75.9
Prior Lines Deleted	75.6
Language	69.4
Age	56.5

Table 5: Mean FPA for selected simple predictor variables

Results for Prior Faults are subtantially inferior to all other prior change measures except for Lines Deleted.

4.3 Prediction Accuracy for Multivariate Prediction Models

In this section, we investigate the performance of various churn measures in a multivariate model that controls for other code and history variables. We begin with a simplification of our Standard Model that includes only log(LOC), programming language and file age (the number of previous releases that a file was in the system). We will call this the *Base Model* and it is the Standard Model without any churn-related predictor variables.

As mentioned earlier, this system includes seven different file types including C, C++, Java, SQL, a proprietary variant of C, a proprietary variant of C++, and .h files. Because analysis of previous systems indicated that fault rates declined quickly over the first few releases of a file before flattening out, we treat age (the number of prior releases) as a categorical variable with four values: 0, 1, 2-4, and greater than 4.

The first line of Table 6 shows that this Base Model produces a mean fault percentile average of 90.93 when averaged across Releases 3 to 18. When we consider log(KLOC) alone, the FPA is 85.9 when averaged across these releases. This is shown in Table 5, so for this system, adding Language and Age yields about a 5.0 percentage point increment in the mean FPA over using size alone.

Subsequent lines of Table 6 display the additional improvement associated with a selected list of churn measures. In order to have measures of the density of changes in a file, we also evaluated relative churn, ratios of the counts of Adds, Deletes, and Modifications to the LOC for the file. Each ratio was truncated at one, if the line count exceeded the LOC. These fault densities are shown in column five of Table 3 and the last column of Table 4.

Because the various counts and line count ratios can be very skewed, we evaluated three versions of each change measure (except the binary-valued PriorChanged indicator): the raw variable, the square root, and the fourth root. We only show the most effective version of each churn measure in the table.

In Table 6 we observe that the two best churn measures were Prior Changes and Prior Adds+Deletes+Mods—with increments of about 2.4 percentage points each. This is consistent with what we observed in Table 5,

For Prior Changes, the best form was the square root, although the fourth root was close. In contrast, the fourth root worked best for Prior Adds+Deletes+Mods, which is more skewed. In general, the untransformed measures performed worse than their transformed counterparts.

Next to each increment in the mean FPA, we show an estimated standard error for that increment. These standard errors are estimated based on release to release variability in the increments associated with each change measure. The resulting t-statistics and P-values are the results of paired two-sample t tests. We note that all these measures produce statistically significant improvements in mean FPA relative to the Base Model without any churn measures.

Other churn measures ranked similarly to those in Table 5. Increments in mean FPA for the relative churn ratios were uniformly smaller than for the absolute churn measures themselves (both after transformations).

Table 7 shows corresponding results starting with an initial model that includes Prior Changes. The addition of Prior Changes to the base model dramatically changes the marginal effectiveness of the other churn measures, of which only three retain statistical significance at the 0.05 level. Prior Cumulative Developers rises from the middle of the pack to the top of the list. The effectiveness of this predictor was previously reported for this system [10] as well as for three other systems [12].

The other statistically significant variable at this point was Prior-Prior Changes (i.e., changes at Release $N-2$), the least effective measure in Table 6. What sets these two variables apart from the others is that each one incorporates information about churn going back beyond the immediately

Predictor Variables	**Mean FPA**	**Increase vs. Base**	**Standard error**	**t-value**	**P-value**
Base: log(KLOC), Language, Age	90.93	NA	NA	NA	NA
$(\text{Prior Changes})^{1/2}$	93.35	2.42	0.24	9.90	0.0000
$(\text{Prior Add+Deletes+Mods})^{1/4}$	93.28	2.35	0.26	9.08	0.0000
$(\text{Prior Add+Deletes+Mods/LOC})^{1/4}$	93.19	2.25	0.28	8.15	0.0000
$(\text{Prior Developers})^{1/2}$	93.17	2.24	0.23	9.68	0.0000
$(\text{Prior Lines Added})^{1/4}$	93.15	2.21	0.26	8.41	0.0000
$(\text{Prior Lines Added/LOC})^{1/4}$	93.03	2.10	0.29	7.25	0.0000
Prior Changed	92.95	2.01	0.23	8.77	0.0000
$(\text{Prior Cum Developers})^{1/2}$	92.93	2.00	0.17	11.65	0.0000
$(\text{Prior Lines Modified})^{1/4}$	92.91	1.98	0.20	9.85	0.0000
$(\text{Prior Lines Modified/LOC})^{1/4}$	92.81	1.87	0.20	9.48	0.0000
$(\text{Prior Faults})^{1/4}$	92.21	1.28	0.16	8.03	0.0000
$(\text{Prior New Developers})^{1/2}$	92.06	1.13	0.25	4.56	0.0004
$(\text{Prior Lines Deleted})^{1/4}$	92.06	1.13	0.16	6.93	0.0000
$(\text{Prior Lines Deleted/LOC})^{1/4}$	92.00	1.06	0.18	5.88	0.0000
$(\text{Prior-Prior Changes})^{1/4}$	91.96	1.03	0.14	7.21	0.0000

Table 6: FPA Improvements for selected churn measures, relative to model without any churn variables

Predictor Variables	**Mean FPA**	**Increase vs. Base**	**Standard error**	**t-value**	**P-value**
Base: log(KLOC), Language, Age, $(\text{Prior Changes})^{1/2}$	93.35	NA	NA	NA	NA
$(\text{Prior Cum Developers})^{1/2}$	93.67	0.32	0.07	4.32	0.0006
$(\text{Prior-Prior Changes})^{1/4}$	93.50	0.14	0.05	3.06	0.0080
$(\text{Prior Add+Deletes+Mods})^{1/4}$	93.38	0.03	0.03	1.08	0.2959
$(\text{Prior Lines Added})^{1/4}$	93.37	0.02	0.03	0.84	0.4132
$(\text{Prior Faults})^{1/4}$	93.36	0.01	0.02	0.47	0.6435
$(\text{Prior Lines Modified})^{1/4}$	93.35	-0.00	0.01	-0.21	0.8387
Prior New Developers	93.35	-0.00	0.00	-1.45	0.1678
Prior Developers	93.35	-0.00	0.00	-0.98	0.3440
$(\text{Prior Lines Deleted})^{1/4}$	93.34	-0.01	0.01	-0.95	0.3549
$(\text{Prior Add+Deletes+Mods/LOC})^{1/2}$	93.34	-0.01	0.04	-0.27	0.7910
$(\text{Prior Lines Modified/LOC})^{1/2}$	93.34	-0.01	0.01	-1.02	0.3237
$(\text{Prior Lines Added/LOC})^{1/2}$	93.34	-0.01	0.05	-0.30	0.7669
$(\text{Prior Lines Deleted/LOC})^{1/4}$	93.33	-0.02	0.01	-2.43	0.0279
Prior Changed	93.30	-0.06	0.05	-1.17	0.2610

Table 7: FPA Improvements for selected churn measures, relative to model with Prior Changes

prior release. For each of the other churn measures, there is apparently sufficient correlation with the Prior Changes measure to blunt any statistically significant improvement.

Adding Prior Cumulative Developers to the initial model produced a mean FPA of 93.67. Beyond that point, the largest observed increment in the Mean FPA is only 0.03 percentage points (for Prior-Prior Changes), and no increment was statistically significant (not shown). For comparison, we note that our Standard Model achieves a mean FPA value of 93.44 for the subject system.

5. THREATS TO VALIDITY

Our various measures of churn tend to be highly correlated. Consequently, it is difficult to determine with much confidence that a particular measure is more effective than all others for fault prediction. Instead, we can mainly assess the marginal value of certain measures in the presence of others.

This study has been carried out on 18 releases of one large system, and similar results may or may not be found for other systems. Many aspects of software development and maintenance can vary from one system to another, including design processes, development approaches, programming languages, and testing strategies. Any of these may introduce significant differences in the frequency or locations of faults, and render the prediction models less successful. In particular, development processes that emphasize minimal rewriting of code, such as cleanroom development, may have a large impact on the number and type of code modifications. In such environments, the relations presented here may not hold. We intend to repeat the investigation on several of the other large systems that we have access to, to see whether we observe similar patterns.

6. RELATED WORK

Fault prediction researchers have investigated a wide variety of predictor variables. Many authors have utilized code mass, various complexity metrics, design information and variations on the history variables that make up our basic model. Work of this type has been reported by Graves et al. [1], Khoshgoftaar et al. [2], Menzies et al. [3, 4], Mockus and Weiss [5], Nagappan and Ball [6, 7], Ohlsson and Alberg [9], Ostrand et al. [11], and Zimmermann and Nagappan [14], among others.

Only a few authors have incorporated churn into prediction models. Mockus and Weiss [5] developed a model to predict the probability that a given change to the software system would result in a software failure. A *given change* is defined as all the modifications to the system that result from a single Maintenance Request, and may consist of multiple individual modifications to multiple code units. Their full prediction model included among its predictor variables the number of files changed, total LOC in the changed files, lines added, lines deleted, and the total number of deltas (check-in/check-out cycles) to a file. Stepwise regression yielded a reduced model that included number of deltas and lines added. This methodology has been incorporated into a "risk assessment tool", but the authors do not state whether it uses the full or the reduced model.

Nagappan and Ball [6] constructed models to predict expected defect density (defects/KLOC) using either absolute or relative churn measures for predictor variables. The absolute measures include added + changed lines, deleted lines, and total number of changes made (apparently equivalent to the deltas of Mockus and Weiss). Relative measures normalize the corresponding absolute measure in terms of total LOC or total file count. The Nagappan-Ball measures are computed for binaries that are the result of compilation of many source files. Since files are the largest code unit in our software, there are no measures that are comparable to their measures that are based on the files within a binary. However, the following two relative measures of code churn are based only on the lines of code in a file, and are comparable to measures that can be computed for the software in the provisioning system:

- M1: $(addedlines + changedlines)/LOC$
- M2: $(deletedlines)/LOC$

	M1	M2	Faults/KLOC
M1	1.000	0.731	**0.258**
M2		1.000	**0.219**
Faults/KLOC			1.000

Table 8: Rank Correlations between Relative Churn Measures and Fault Density

Nagappan and Ball found rank correlations of 0.8 and above between M1 or M2, and fault density. We observed considerably lower correlations, although the relative values of the correlations were similar. Table 8 shows rank correlations between fault density for the provisioning system we analyzed, and the measures M1, M2, as well as the inter-measure correlations.

Nagappan and Ball's relative measure models produced R^2 values of .8 or higher on the Windows code, significantly better than the absolute measure models, leading to their conclusion that relative churn can be an efficient and effective predictor of defect density. In contrast to the results for Windows 2003, we found virtually no difference between the effectiveness of absolute and relative churn measures. In fact, the relative measures proved uniformly slightly less effective than the absolute.

In [7], Nagappan and Ball examine models that combine churn and system dependency information to predict post-release failures of system binaries that are constructed from a set of source files. They use 3 churn-related metrics: overall count of lines added, deleted, or modified, number of files in the binary that were changed, and total number of changes made to the files in the binary. Comparison of the post-release failures predicted by these models against actual failures showed R^2 values better than .6, with $P<.0005$, leading them to conclude that, at least for the subject system (Windows Server 2003), churn and system dependency information are reliable early indicators of failures.

Nagappan et al. [8] investigate the ability of *change bursts*, sequences of consecutive changes to a software system, to predict the system's fault-prone components. They define churn as the total of lines added, deleted, and modified during changes made to a component, and use the total churn over a component's lifetime, the churn within bursts, and the maximum churn occurring in any burst as three of the predictor variables in their model. When measured for Windows Vista, bursts turn out to be highly effective predictors of fault-prone components.

It is worth noting that each of these studies assessed their models by data splitting the information from a single release of the software. The churn data is based on changes made between the release's baseline version up to the final version that produced the production binaries.

7. CONCLUSIONS

Consistent with our earlier observations and results, we found that most faults in the system analyzed here occurred in files that had been changed in the prior release. The importance of changes is so high that even a simple changed/not-changed variable is capable of providing respectable predictions of fault-prone files.

Confirming other research studies, we have seen that low-level measurements of code changes can be very effective for fault prediction. Counts of additions, deletions and changes to a code base can be derived from the version control history of a system, and be used as input to a fault prediction model.

Of the three specific types of file changes, lines added delivered the most accurate predictions, followed by lines modified. Lines deleted had substantially less predictive value than the other two. The sum of all three counts, essentially a count of the total lines changed in a file, proved the most effective way to use the individual file churn data and was as good as any other variable that we tried.

However, the line counts did not improve prediction accuracy for this system relative to our Standard Model, which already included an alternative measure of churn in the prior release. It appears that either our old measure, Prior Changes, or a sum of Adds+Deletes+Mods can be equally effective.

8. REFERENCES

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

[2] T.M. Khoshgoftaar, E.B. Allen, J. Deng. Using Regression Trees to Classify Fault-Prone Software Modules. *IEEE Trans. on Reliability*, Vol 51, No. 4, Dec 2002, pp. 455-462.

[3] T. Menzies, J. Greenwald and A. Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Trans. Software Eng.*, 33(1), pp. 2-13, 2007.

[4] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic and Y. Jiang. Implications of Ceiling Effects in Defect Predictors. *Proc. 4th Int. Workshop on Predictor Models in Software Engineering (PROMISE08)*, pp. 47-54, 2008.

[5] A. Mockus and D.M. Weiss. Predicting Risk of Software Changes. *Bell Labs Technical Journal*, April-June 2000, pp. 169-180.

[6] N. Nagappan and T. Ball. Use of Relative Code Churn Measures to Predict System Defect Density. *Proc. 27th Int. Conference on Software Engineering (ICSE05)*, 2005.

[7] N. Nagappan and T. Ball. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. *Proc. Empirical Software Engineering and Measurement Conference (ESEM)*, Madrid, Spain, 2007.

[8] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change Bursts as Defect Predictors. *Proc. 21st IEEE Int. Symposium on Software Reliability Engineering (ISSRE2010).*

[9] N. Ohlsson and H. Alberg. Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Trans. on Software Engineering*, Vol 22, No 12, December 1996, pp. 886-894.

[10] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Programmer-based Fault Prediction. *Proc. Int. Conference on Predictive Models (PROMISE10)*, Timisoara, Romania, September 2010.

[11] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Trans. on Software Engineering*, Vol 31, No 4, April 2005.

[12] E.J. Weyuker, T.J. Ostrand, and R.M. Bell. Do Too Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models. *Empirical Software Eng.*, Vol 13, No. 5, October 2008.

[13] E.J. Weyuker, T.J. Ostrand, and R.M. Bell. Comparing the Effectiveness of Several Modeling Methods for Fault Prediction. *Empirical Software Eng.* Vol 15, No. 3, June 2010.

[14] T. Zimmermann and N. Nagappan. Predicting Defects Using Network Analysis on Dependency Graphs. *Proc. 13th Int. Conference on Software Engineering (ICSE08)*, p.531-540, 2008.