Change Bursts as Defect Predictors

Nachiappan Nagappan* Andreas Zeller† Thomas Zimmermann‡ Kim Herzig§ Brendan Murphy¶

***Abstract*—In software development, every change induces a risk. What happens if code changes again and again in some period of time? In an empirical study on Windows Vista, we found that the features of such *change bursts* have the highest predictive power for defect-prone components. With precision and recall values well above 90%, change bursts significantly improve upon earlier predictors such as complexity metrics, code churn, or organizational structure. As they only rely on version history and a controlled change process, change bursts are straight-forward to detect and deploy.**

***Keywords*-Process metrics; product metrics; software quality assurance; version control; change history; defects; developers; software mining; empirical studies**

I. Introduction

Software development can be seen as a *sequence of changes*—a constant stream of activities that add new value to software, adapt it to a changing environment, delete features no longer required, or improve its structure for better maintenance. All of these activities are ultimately conducted by humans, and as humans make mistakes, it is unavoidable that some of these changes will induce defects.

The aim of quality assurance is to find and fix these defects before release—using testing, code reviews, and more. To make quality assurance effective, one must direct its efforts to those components that are the most likely to contain defects. Such predictions can draw on a variety of sources. A high number of defects found before release, a low test coverage, dependency on specific ("hard") components, code complexity, a large extent of changes ("code churn"), or organizational complexity (e.g., the number of engineers involved) have all been related to the defect-proneness of components. If one applies these predictors on the components of Windows Vista (Table I), their *precision* varies between 74% and 86% (i.e., 74%–86% of the components classified as defect-prone actually are defect-prone); their *recall* varies between 54% and 84% (i.e., 54%–84% of the defect-prone components are actually classified as such). The predictors also vary in their *requirements*. For instance, the organizational structure (the best predictor so far) requires employee data [1].

Table I
Comparing predictors for defect-prone Vista components [1]

Predictor	Precision	Recall
Pre-Release Defects	73.8%	62.9%
Test Coverage	83.8%	54.4%
Dependencies	74.4%	69.9%
Code Complexity	79.3%	66.0%
Code Churn	78.6%	79.9%
Organizational Structure	86.2%	84.0%
Change Bursts *(this paper)*	91.1%	92.0%

There is a *common denominator* to all these predictors, though. In earlier studies [2], Śliwerski et al. had observed that some changes apparently were *hard to get right*—that is, they led to further errors and subsequent fixes. The ECLIPSE `resolveClasspath()` method, for instance, was changed nine times, and all of these changes were fixes, including a "fallback fix" reverting the method to an earlier revision. Only after multiple attempts did the method reach a stable state—and still, it is very risky to change.

In this paper, our conjecture is that over the development time of a system, such multiple attempts would manifest themselves in consecutive code changes over a period of time. Such *change bursts* could be indicators for various problems, including those traditionally detected by earlier predictors:

- **Incomplete or changing requirements.** Requirements may only become stable after multiple implementation attempts—for instance, because of *conflicting organizations* involved.
- **Hairy bugs.** Defects may only be tentatively fixed without knowing the exact cause, making them re-occur again and again—that is, the code or task is overly *complex*.
- **Insufficient quality assurance.** Quality assurance may not detect all issues in the first place, thus requiring constant fixing of newly discovered defects—improving *test coverage* over time.

Problems like these would not be addressed by code fixes alone; instead, they are likely to persist until after the release date. This implies that a component with such problems will undergo a higher amount of maintenance, and be more defect-prone than others. We therefore investigate how

*Nachiappan Nagappan ⟨nachin@microsoft.com⟩ is with Microsoft Research, Redmond, Washington.

†Andreas Zeller ⟨zeller@cs.uni-saarland.de⟩ is with Saarland University, Saarbrücken, Germany. He was a visiting researcher with the Software Engineering Group at Microsoft Research in the Summer of 2009 when this work was carried out.

‡Thomas Zimmermann ⟨tzimmer@microsoft.com⟩ is with Microsoft Research, Redmond, Washington.

§Kim Herzig ⟨herzig@cs.uni-saarland.de⟩ is with Saarland University, Saarbrücken, Germany.

¶Brendan Murphy ⟨bmurphy@microsoft.com⟩ is with Microsoft Research, Cambridge, UK.

change bursts would fare as predictors for defect-prone components. To make a long story short, they fare among the best. Despite having little requirements (i.e., relying solely on the change history), change bursts yield very good predictive power—for Windows Server 2003, they rank among the best predictors; and for Windows Vista, they even bring both precision and recall above 90%, the highest such values ever measured. However, we also found that to distinguish bursts from regular activity, one needs a *controlled change process,* where changes are committed only when considered "ready", i.e., expected to keep the product stable. This becomes evident when applying the predictors to ECLIPSE, where anyone can commit any change at any time—and where change bursts are "only" as good as regular change activity. We assume, though, that most industrial software projects follow a controlled change process.

The remainder of this paper is organized as follows: In Section II, we introduce the concept of change bursts and how to extract them from a series of changes. Section III discusses the metrics we have applied on change bursts—metrics such as number of developers involved, the size of the burst, or the number of lines changed. In Section IV, we show how to leverage these metrics to predict defect-prone components. Section V discusses our experiments, where we evaluate and compare the predictive power of change bursts. After discussing the related work in Section VI, Section VII closes with contributions and future work.

II. Detecting Change Bursts

Let us start by describing how we identify change bursts. As stated in the introduction, a change burst is a sequence of consecutive changes. But what does "consecutive" mean? When do two changes follow each other, and when not?

We assume that we can split the history of a system into a series of events at which we would assume there could be some change or not. Such an event could be a *calendar day,* for instance—but then, we would find that weekends and holidays have a high chance of breaking a consecutive series. We therefore restrict ourselves to those days that some change is committed to the system code. For Windows, these are the *build days,* as every working day, a new build is derived from the version archive. We thus see a system S as a sequence of *builds* $S = \langle s_1, s_2, \dots \rangle$ where each build differs from the previous one $s_i \neq s_{i-1}$.

We assume that each build is created out of individual *components*—that is, classes, packages, modules or other constituents. (For Windows, the components are *binaries*—that is, `.exe`, `.sys`, or `.dll` files.) A component C of S also has a history across builds, and thus comes in a series of individual component versions c_i; we thus define $C = \langle c_1, c_2, \dots, c_{|S|} \rangle$. If $c_i \neq c_{i-1}$ holds, then the component C has *changed* in build s_i.

For each component, we now determine its change bursts as sequences of consecutive changes. These change bursts

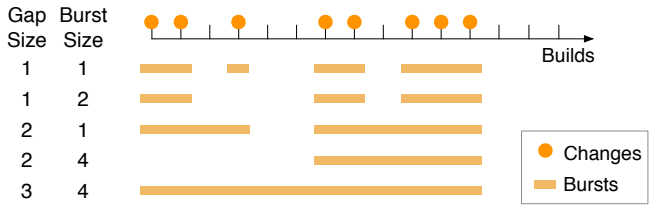


Figure 1. How gap size and burst size determine change burst detection from a sequence of changes.

are determined by two parameters:

- **Gap size.** We would like to permit short gaps in these sequences, such that a one-day distraction will not break the burst. We therefore introduce the *gap size* $\mathcal{G}$, which determines the *minimum distance* between two changes. If two changes have a distance that is smaller than $\mathcal{G}$, they will be part of the same burst.
- **Burst size.** On the other hand, we would like to consider only bursts of a certain significance—that is, a certain length. The *burst size* $\mathcal{B}$ determines the *minimum number of changes* in a burst. If the number of changes in a burst b is smaller than $\mathcal{B}$, it will not be considered.

As an example, consider Figure 1. The arrow at the top shows the sequence of builds; the builds in which a component c has changed are marked with a dot. The rectangles below show the change bursts. If we set the gap size and the burst size to 1, then all directly consecutive changes will be merged to bursts. Increasing the gap size yields longer bursts; increasing the burst size eliminates shorter bursts.

Formally, given a sequence $C = \langle c_1, c_2, \dots \rangle$, a *change burst* is a sequence $B = \langle c_{i_1}, c_{i_2}, \dots \rangle$ with indices $i_1, i_2, \dots, i_{|B|} \in \{1, \dots, |C|\}$ such that

- $c_{i_k} \neq c_{i_{k+1}}$ (each element has changed),
- $|B| \geq \mathcal{B}$ (the length is at least the burst size), and
- for all $1 \leq k < |B|$, we have $i_k - i_{k+1} \leq \mathcal{G}$ (there is no gap larger than the gap size).

The sequence $\mathit{bursts}(C) = \langle B_1, B_2, B_3, \dots \rangle$ of *all change bursts* for C consists of the longest non-overlapping change bursts; i.e., $\sum_{i=1}^{|\mathit{bursts}(C)|} |B_i|$ is maximal, and $b_i \cap b_j = \emptyset$ holds for all $b_i, b_j \in \mathit{bursts}(C)$ with $b_i \neq b_j$. The computation is deterministic; there is only one possible sequence $\mathit{bursts}(C)$ for a given C.

III. Change Burst Metrics

We want to characterize change bursts by a number of *features,* such that we can leverage these very features to predict defect densities. For each component C, we therefore determine change metrics, temporal metrics, people metrics, and churn metrics. In their formal definitions, we identify a component history as $C = \langle c_1, c_2, \dots \rangle$ and its bursts as $\mathit{bursts}(C) = \langle B_1, B_2, \dots \rangle$.

A. Change Metrics

We start with a few basic metrics for a component C concerning the size and extent of its change bursts.

- ***NumberOfChanges*.** This is the number of builds in which the component C has changed, i.e., $|\{k \mid c_k \neq c_{k+1}\}|$.
  Rationale: We assume that the more a component is changed, the more likely it is to have defects.
- ***NumberOfConsecutiveChanges*.** This is the number of consecutive builds for a given gap size $\mathcal{G}$—in other words, $|\mathit{bursts}(C)|$ with $\mathcal{B} = 0$.
  Rationale: This measure takes into account *all* consecutive changes, not just bursts exceeding a given size.
- ***NumberOfChangeBursts*.** This is the number of change bursts for a given gap size $\mathcal{G}$ and burst size $\mathcal{B}$, i.e., $|\mathit{bursts}(C)|$.
  Rationale: As stated in the introduction, we would assume that change bursts are risky; thus, the number of change bursts may be predictive for defect-prone components.
- ***TotalBurstSize*.** This is the number of changed builds in all change bursts, i.e., $\sum_{B \in \mathit{bursts}(C)} |B|$.
  Rationale: Assuming that change bursts indicate risky activities, a high number of changes during these bursts could be particularly risky.
- ***MaximumChangeBurst*.** This is the maximum number of changed builds in all change bursts, i.e., $\max\{|B| \mid B \in \mathit{bursts}(C)\}$.
  Rationale: As in *TotalBurstSize*, but looking for extremes.
- **Early and late metrics.** We also compute the above metrics for early periods (80% of the project's lifetime), resulting in *NumberOfChangesEarly*, *NumberOfChangeBurstsEarly*, and so on, as well as for the late periods (the last 20% before release), resulting in metrics such as *NumberOfChangesLate*, *NumberOfChangeBurstsLate* and likewise.
  Rationale: We assume that late activities right before release might be very different from the early activities, and thus deserve to be independently measured.

All these metrics are straight-forward to compute from a given version history, given the change burst definitions from Section II.

B. Temporal Metrics

As activities change during a project's time line, the time when change bursts occurred may be predictive for defects. We therefore introduce a set of *temporal metrics,* highlighting when change bursts occurred.

- ***TimeFirstBurst*.** We determine when the first burst occurred, normalized to the total number of builds, i.e., $\min\{i \mid C_i \in B_1\}/|C|$.
  Rationale: We assume that early change bursts may have the longest impact during the project.
- ***TimeLastBurst*.** As *TimeFirstBurst*, but looking at the *last* burst instead, i.e., $\min\{i \mid C_i \in B_{|\mathit{bursts}(C)|}\}/|C|$.
  Rationale: We assume that the last activities before release define the final shape of the component and thus may be particularly predictive.
- ***TimeMaxBurst*.** As *TimeFirstBurst*, but looking at the greatest burst instead. Let B_{max} be the burst with the most changes, i.e., $\forall B \in \mathit{bursts}(C) : |B_{\mathit{max}}| \geq |B|$ holds. Then, *TimeMaxBurst* is the time $\min\{i \mid C_i \in B_{\mathit{max}}\}/|C|$.
  Rationale: Again, we are looking for extremes, and check when the greatest change burst occurred—early or late.

To extract these metrics from a version history, all one needs is the timestamp of the individual changes.

C. People Metrics

In the following definitions, let $\mathit{people}(c_i)$ be the set of developers who committed the changes to the component revision c_i. By extension, let us also apply *people* to sets, as in $\mathit{people}(C) = \bigcup_{c_i \in C} \mathit{people}(c_i)$.

- ***PeopleTotal*.** The number of people who ever committed a change to the component C, i.e., $|\mathit{people}(C)|$.
  Rationale: "Too many cooks spoil the broth": The number of developers working on a component may be related to defects [3].
- ***TotalPeopleInBurst*.** Across all bursts, the number of people involved, i.e., $|\mathit{people}(\mathit{bursts}(C))|$.
  Rationale: We see change bursts as defining moments; hence, the number of people involved may be particularly predictive.
- ***MaxPeopleInBurst*.** Across all bursts, the maximum number of people involved, i.e., $\max\{|\mathit{people}(B)| \mid B \in \mathit{bursts}(C)\}$.
  Rationale: Again, we are looking for extremes, and check for the greatest change burst in terms of people.

To extract these metrics from a version history, all one needs is the name of the developer who committed the individual change. The only risk to be aware of is that some developers may change their user IDs over time, or may be working from multiple IDs [2]. This is not the case for Windows, though.

D. Churn Metrics

In the following definitions, let $\mathit{churn}(c_i)$ be the number of lines that were added, deleted, or modified during the changes to the component c_i. By extension, let us also apply *churn* to sets, as in $\mathit{churn}(C) = \sum_{c_i \in C} \mathit{churn}(c_i)$.

- ***ChurnTotal*.** The total churn over the lifetime of a component C, i.e., $\mathit{churn}(C)$.
  Rationale: We assume that the more has changed, the higher the likelihood defects will be introduced.

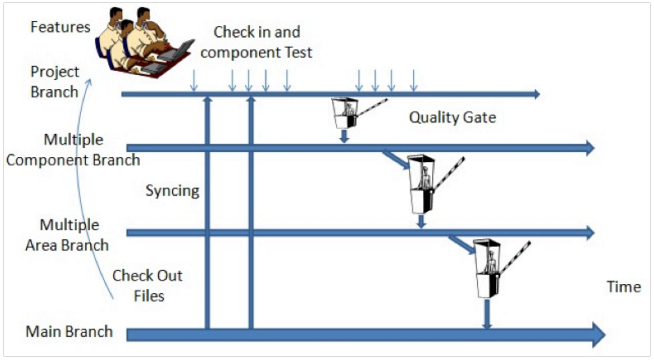


Figure 2. How the Windows development process works. Changes are first committed in project branches, and then subsequently merged and integrated into the Windows main branch.

- ***TotalChurnInBurst.*** The total churn in all change bursts, i.e., $churn(bursts(C))$.
  Rationale: Again, we see change bursts as defining moments; hence, the amount of change involved may be particularly predictive.
- ***MaxChurnInBurst.*** Across all bursts, this is the maximum churn, i.e., $\max\{|churn(B)| \mid B \in bursts(C)\}$.
  Rationale: Again, we are looking for extremes across change bursts.

All these metrics require is the set of changed lines—a standard feature for every version repository.

IV. Predicting Defect-Prone Components

In order to determine if the change burst metrics defined in Section III are effective indicators of defect-prone components, we use them as predictors in a logistic regression equation to classify components as defect-prone or not.

A. The Subject

We ran our initial experiments on the Windows Vista operating system, consisting of 3,404 Windows Vista binaries exceeding 50 million LOC. For Vista, a significant defect and change history was available, and it had been the subject of other defect prediction studies before (Table I); hence, we would be able to compare the predictive power against the state of the art.

As our metrics all assess the change process, let us illustrate how the Windows development process works. As shown in Figure 2, the Windows software development takes place in various *branches* of the version control system with tightly integrated schedules for code integration and comprehensive builds. There are several individual features of Windows each of which are developed in individual project branches (An example feature could be "sound" in the component "DirectX" in the area "Multimedia"). Each of these individual project branches assume the rest of the code base to be frozen, except for their evolving features. Engineers check-in their code to the project branches. To ensure that the newly developed code, in the project branch, maintains compatibility with the other changes committed to the main branches, the project branches continually synchronizes with the main branch.

After passing *quality gates* (ranging from code coverage to static analysis clean) the code in these branches moves to a component branch. Once the project code is checked into the main branch then it is automatically synced with the rest of the code base to ensure that other code being developed within the other project branches are compatible with these changes. This process ensures that there is stability in the main Windows branch, with a working version of Windows always available for system test and other purposes.

B. The Dataset

Our dataset for the initial experiments consisted of 3,404 Windows Vista binaries exceeding 50 million LOC where each binary has its change burst metrics and post-release defects mapped. Defect-proneness is the probability that a particular software component (such as a binary) will fail in operation in the field—and hence exhibit a defect. The higher the defect-proneness, the higher the probability of experiencing a post-release defect. To classify the binaries in Vista in two categories, not defect-prone and defect-prone, we define a statistical lower confidence bound (LCB) on all defects [1].

C. Stepwise Regression

An important question to address is whether all change burst metrics would be required in building a predictive model. For this purpose, we use *stepwise regression* [4]. The initial regression 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, we evaluate the model—and predictors that do not significantly contribute towards statistical significance in terms of the F-ratio are removed. Thus, in the end, the best set of predictors explaining the maximum possible variance is left.

We performed a stepwise regression using the change burst measures as the predictor variables and post-release defects as the dependent variable. The regression did not yield any reduction in the number of predictor variables (retaining all change burst measures)—indicating that *all metrics contributed* towards explaining the variance in accounting for the post-release defects.

D. Random Splits

We use the technique of *random data splitting* to measure the ability of the change burst metrics to predict defect-proneness. The data splitting technique is employed to get an independent assessment of how well the defect-proneness could be estimated from a population sample. We randomly

Table II
RESULT CLASSIFICATION

	Predicted as defect-prone	Predicted as not defect-prone
Actually defect-prone	*true positive*	*false negative*
Actually not defect-prone	*false positive*	*true negative*

select two thirds (2,268) of the Windows Vista binaries to build the prediction model (i.e., the *training set*) and use the remaining one third (1,136) to verify the prediction accuracy (the *testing set*). If the predicted defect-proneness probability is > 0.5, we predict the binary to be defect-prone, otherwise not defect-prone.

E. Precision and Recall

To evaluate the predictive power, we compared the *predicted* defect-proneness with the *actual* defect-proneness for all components in the testing set. We then group the components into four categories, shown in Table II. True positives and negatives are correctly classified, false positives and negatives are mismatches.

Of course, one would want to have as many true results and as little false results as possible. To capture the relationship between these numbers, and to evaluate the predictive power, we use the well-established measures of *precision* and *recall*.

- **Precision.** The *precision* P tells how many of the components predicted to be defect-prone actually are defect-prone:

$$P = \frac{\textit{true positives}}{\textit{false positives} + \textit{true positives}}$$

  A precision $P = 1.0$ implies *no false positives*: Every component predicted to be defect-prone is defect-prone.
- **Recall.** The *recall* R indicates how many of the defect-prone components actually are predicted as such:

$$R = \frac{\textit{true positives}}{\textit{false negatives} + \textit{true positives}}$$

  A recall $R = 1.0$ means *no false negatives*: Every defect-prone component is correctly predicted to be defect-prone.

A perfect prediction has $P = R = 1.0$, meaning neither false positives nor negatives. While it is easy to achieve either $P = 1.0$ or $R = 1.0$ (just predict *no* or *all* components to be defect-prone), the goal is to maximize both values.

To ensure the validity of our measures, we measured precision and recall in 50 random split experiments, as detailed in Section IV-D.

V. EXPERIMENTS

A. Can change bursts predict defect-prone components?

In our first experiment, we evaluated the precision and recall of change burst metrics as defect predictors as laid out in the previous section. We obtained a precision $P = 91.1$ and a recall $R = 92.0$.

With precision and recall well above 90%, change burst metrics make excellent defect predictors.

B. How do change bursts compare against other predictors?

Table I compares the results against precision and recall values observed in earlier studies on Windows Vista. Change burst metrics provide by far the highest precision and recall:

- Compared to *code churn metrics,* which use the same data source, they improve precision and recall by more than 10 percentage points. In practice, this means that the number of false positives or false negatives is reduced by 50%.
- Compared to *organizational metrics,* which showed the best predictive power so far, they improve by 5 percentage points, reducing the number of false positives and negatives by 30%.

One could assume that by combining code bursts with other metrics, one could further improve the predictive power. For instance, one could examine the complexity of the code being changed; or the number or distance of organizations involved. Keep in mind, though, that such metrics require additional data sources and that having too many metrics brings the risk of overfitting. Also, it is hard to argue why one still would want to improve on this predictive power.

On Vista, change burst metrics have the highest predictive power for defect-prone components ever observed.

C. How do gap size and burst size influence the results?

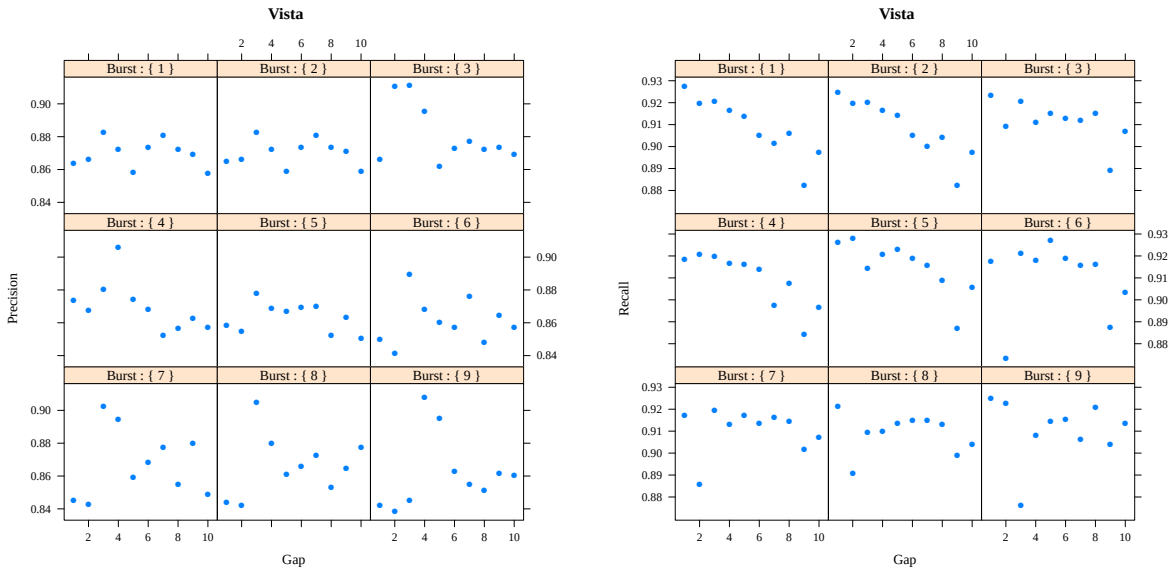
For our experiments, we used *exhaustive evaluation,* i.e., we tried out all possible gap and burst size combinations (with $\mathcal{G} \in \{1, \ldots, 10\}$ and $\mathcal{B} \in \{1, \ldots, 10\}$); the change burst results shown in Table I were obtained with a gap size $\mathcal{G} = 3$ and $\mathcal{B} = 3$.

In Figure 3(a), we see how the *precision* changes for different values of burst size $\mathcal{B}$ (from top left, $\mathcal{B} = 1$, to bottom right, $\mathcal{B} = 9$) and gap size $\mathcal{G}$ (on the X axis for each graph). We see that while indeed $\mathcal{G} = 3$ and $\mathcal{B} = 3$ bring a maximum, the precision never incurs large drops; it almost always remains better than the state of the art as shown in Table I. Similar effects are observed for *recall*, shown in Figure 3(b).

On Vista, change bursts retain their high predictive power even as gap and burst size vary.

D. How are change bursts distributed?

Interestingly, a gap size $\mathcal{G} = 3$ and a burst size $\mathcal{B} = 3$ not only yield the best precision and recall; they also are precisely those values where we could determine a *maximum number of change bursts.*



(a) Precision (b) Recall

Figure 3. Predictor precision and recall in Vista for varying gap sizes and burst sizes. The upper right panel shows precision (left) and recall (right) for a given burst size $\mathcal{B} = 3$ and a varying gap size $\mathcal{G}$; the values for $\mathcal{G} = 3$ are the values reported in Table I (precision 91.1, recall 92.0).

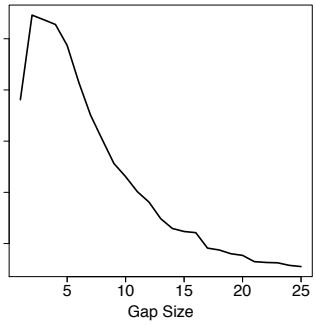


Figure 4. How gap size influences the number of change bursts in Vista. As the gap size $\mathcal{G}$ increases, changes first become consecutive, increasing the number of bursts. As $\mathcal{G} > 3$, change bursts are merged, decreasing their number again.

Figure 4 shows how the gap size influences the number of change bursts in Vista:

- As the gap size increases, more changes become *consecutive,* increasing the number of bursts. In Vista, this happens up to a gap size $\mathcal{G}$ of 2–3.
- As the gap size increases further, existing change bursts will be *merged,* decreasing the number of bursts. In Vista, the number of bursts thus constantly decreases as soon as $\mathcal{G} > 3$.

On Vista, a gap size $\mathcal{G} = 3$ and a burst size $\mathcal{B} = 3$ yields a maximum in change bursts and predictive power.

E. Are these results specific to Vista?

To get insight on how the change process influences the results, we have replicated this study on a different system. *Windows 2003 Server* has a similar staged change propagation mechanism as Vista, but is even stricter: Developers and teams would be allowed to commit their changes only on designated days, each followed by weeks of testing and quality assurance. In this setting, multiple change attempts would only manifest themselves in the developer's private history, but no longer in the project history. Despite these constraints, for *Windows 2003 Server,* our approach yields a precision of 74.4% and a recall of 88.0% ($\mathcal{G} = 10$ and $\mathcal{B} = 2$)—again, an excellent result that is on par with the best predictions made for this system.

Even with a strictly controlled change process, change bursts remain among the best defect predictors.

F. When does this not work?

The concept of change bursts assumes that changes are committed whenever the developer considers them ready for release. This is how the Windows change process works: A change can only be committed to the Windows kernel after an extensive series of tests and reviews. This is why change bursts are particular—changes had to be applied again and again *despite all these checks.*

What happens if such checks are not present—or not as much enforced? ECLIPSE is an open-source programming environment written in JAVA by paid IBM developers as well as contributors from all over the globe. Its code, version history, and defect database, are all open to the public; in particular, the ECLIPSE defect distribution, as mined from its defect database, is available as a dataset from the PROMISE repository [5].

Table III
COMPARING PREDICTORS FOR DEFECT-PRONE ECLIPSE 2.0 PACKAGES.

Predictor	Precision	Recall
Dependencies [6]	58.7%	79.6%
Code Complexity [5]	74.2%	73.5%
Change Metrics [7]	n/a[1]	81.0%
Change Bursts *(this paper)*	67.0%	51.0%

The ECLIPSE and Windows product differ in several features, including the programming language, the domain, and the degree at which the source code is available to the public. As it comes to change bursts, though, it is the difference in *process* that matters:

- In ECLIPSE, almost all changes are committed immediately to the main branch. There are few branches, if any; and there is no similarity to the Windows staged development and integration process as shown in Figure 2.
- The Windows process assumes the existence of a *daily build* where only tried-and-tested changes are committed. As laid out in Section II, it is this daily build that we use to identify change bursts.
- In ECLIPSE, there is no notion of a daily build or anything similar where we would assume that there is a specific commitment in terms of reliability or quality. There are *releases,* of course, where we can safely assume such a commitment, but these are not applicable to change bursts.

When measuring change bursts on ECLIPSE, we therefore assumed that *every day with any change committed* would be a build day, too. Otherwise, our experiment exactly replicated the study as previously conducted on Vista and Windows Server 2003.

Table III shows our results with $\mathcal{G} = 2$ and $\mathcal{B} = 1$, again comparing them against other predictors. While the precision of 67.0% is on par with previously reported predictors, the recall of 51.0% for change bursts is lower. This means that while change bursts in Eclipse indeed indicate a higher defect-proneness, not every defective component is characterized by change bursts—or at least not in the way we have modeled them in this paper.

The differences between the Windows and Eclipse development processes is best illustrated by again investigating how the number of bursts changes with varying gap size. Figure 5 shows the equivalent of Figure 4 for Eclipse: Again, with increasing gap size $\mathcal{G}$, we see an increase in the number of bursts as more changes become consecutive. We don't see any decrease due to *merging* of change bursts, though. Hence, in Eclipse, change bursts are much more isolated events than in Vista—and thus are more like small disturbances in a steady stream of changes, rather than intense activities focused on a specific task.

[1]Moser et al. [7] report the percentage of correctly predicted files PC = 82%, and a false positive rate FP = 11%.

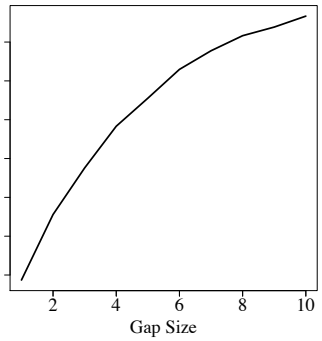


Figure 5. How gap size influences the number of change bursts in Eclipse. In contrast to Vista, we see no merges of bursts in Eclipse—the number of bursts simply increases. This indicates that bursts in Eclipse are much more isolated events than in Vista.

Such relative absence of change bursts can also be explained by the Eclipse architecture: Eclipse comes as a collection of loosely connected plug-ins which effectively limit the effects of changes. This is in sharp contrast to an operating system like Vista which shows a much tighter integration of modules—but also more widespread effects of changes.

These results confirm our reservations: Change bursts are good predictors of issues if each change initially came with the expectation of maintaining correctness and stability—and where bursts of such changes thus are clear indicators of trouble. In environments where arbitrary changes can be committed at arbitrary times, change bursts will fare just as good as regular change activity (which still is a very good predictor). While we assume that most industrial software projects follow a controlled change process, prospective users of change burst metrics are advised to first run an experimental study as set forth in this paper; the appendix contains all the necessary material for replicating this study.

Change bursts assume a process where changes are expected to maintain quality and stability.

G. Threats to Validity

As with any empirical study, the interpretation of our results is subject to several limitations.

- **Internal Validity.** In our study, internal validity issues primarily deal with the *causal issues* of our results. These concerns are addressed to some extent due to the fact that the engineers in Windows and ECLIPSE had no knowledge that this study was being performed for them to artificially modify their behavior/coding practices to affect our measurements. Furthermore, the second and fourth author are not Microsoft employees; hence, there is no internal motivation to show results either way to influence Windows.
  The experiment does suffer from experimenter bias, that is, bias towards a result expected by the human exper-

imenters. To minimize this bias, the authors split into two groups: The last two authors worked independently to collect and process the Windows and ECLIPSE data, respectively, and the first three authors independently analyzed the results.

- **Construct Validity.** Construct validity issues arise when there are errors in measurement. This is negated to an extent by the fact that the entire data collection process of change burst metrics and defects is automated. In the light of our results, we have individually implemented independent evaluation prototypes and came to the same result. These concerns are also alleviated to some extent by the large size and diversity of our datasets.
  While the Windows data maps all fixed problems to defects in the code, the PROMISE ECLIPSE data set only maps 70% of the fixed problems [8]. Hence, all research on the ECLIPSE data set only uses a subset of defects, which may not be representative for all fixed problems.
- **External Validity.** External validity issues may arise from the fact that all the data is from only two software systems (albeit with many different components) and that both systems are very large; other systems used for a similar analysis may not be of comparable size.
  In software engineering, any process depends to a large degree on a potentially large number of relevant context variables. The differences we observed between Windows and Eclipse may be due to such confounding factors.
  The large number of factors also means that we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted [9]. We do not claim that models learned from one project would be applicable to another; in all cases, training on existing defects will be required. On the other hand, existing defects also serve to evaluate the technique, as we did in this paper. We hope that our case study contributes to strengthening the existing empirical body of knowledge in the field.

VI. Related Work

In this section we summarize some of the related work regarding metrics and defects. Relevant studies on Microsoft systems are also presented providing context and for comparison to our current work. We organize our work based on the type of metrics that have been studied for defect prediction.

A. Code Churn

Graves et al. [10] predict fault incidences using software change history based on a weighted time damp model using the sum of contributions from all changes to a module, where large and/or recent changes contribute the most to fault potential [10].

Ostrand et al. [11] 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 successfully predict (high accuracy for faults found in both early and later stages of development) the number of faults in a multiple release software system.

Moser et al. [7] use *change metrics* such as the number of revisions or refactorings to predict defects in Eclipse classes. In their paper, they report that *cost-sensitive classification* yields $> 75\%$ of correctly classified files and a recall of $> 80\%$.[2] Hassan [12] measures the complexity of the *change process* by assessing how much modifications are scattered across space and time. The resulting *entropy metrics* are evaluated to be better predictors than prior faults.

B. Code Complexity

Khoshgoftaar et al. [13] studied two consecutive releases of a large legacy system (containing over 38,000 procedures in 171 modules) for telecommunications. 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% and 19.1%, respectively, and an overall misclassification rate of 21.0%.

The *CK metric suite* [14] consist of six metrics, designed primarily as object oriented design measures: weighted methods per class (WMC), coupling between objects (CBO), depth of inheritance (DIT), number of children (NOC), response for a class (RFC), and lack of cohesion among methods (LCOM). The CK metrics have also been investigated in the context of fault-proneness:

- Basili et al. [15] studied the fault-proneness in software programs using eight student projects. They observed that the WMC, CBO, DIT, NOC and RFC metrics were correlated with defects while the LCOM metric was not correlated with defects.
- Further, Briand et al. [16] performed an industrial case study and observed the CBO, RFC, and LCOM metrics to be associated with the fault-proneness of a class.
- Within five Microsoft projects, Nagappan et al. [17] identified complexity metrics that predict post-release failures and reported how to systematically build predictors for post-release failures from history.

C. Code Dependencies

Schröter et al. [6] showed that import dependencies can predict defects. They proposed an alternate way of predicting

[2]Note that these results were obtained from a flawed version of the Eclipse PROMISE data [5]; we are working on repeating their experiments on the revised data set.

defects for Java classes. Rather than looking at the complexity of a class, they looked exclusively at the components that a class uses. For Eclipse, the open source IDE they found that using compiler packages results in a significantly higher defect-proneness (71%) than using GUI packages (14%).

Prior work at Microsoft [18] on the Windows Server 2003 system illustrates that code dependencies can be used to successfully identify defect-prone binaries with precision and recall values of around 73% and 75% respectively.

D. Organizational Structure

Nagappan et al. [1] observed that organizational metrics (metrics extracted from how the software development team is organized hierarchically) had a significantly high prediction accuracy in a case study performed on Windows Vista with a precision of 86.2% and recall of 84.0%.

E. Combination of Metrics

Denaro et al. [19] calculated 38 different software metrics (lines of code, Halstead software metrics, nesting levels, cyclomatic complexity, knots, number of comparison operators, loops etc.) for the open source Apache 1.3 and Apache 2.0 projects. Using logistic regression models built using the data collected from the Apache 1.3 they verified the models against the Apache 2.0 project with high correctness/completeness.

Khoshgoftaar et al. [20] 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.

F. Summary

Across all these studies, precision and recall values are around 85% on average for the best of studies. We have not seen a study yet which has both precision and recall in the 90 percentile, as we have seen for Vista. Of course, the objective of our research is not to just keep increasing the prediction accuracy—but to find the reasons and factors that improve so. In that spirit, this paper presents two different product families (Windows and Eclipse) to illustrate how the difference in process can characterize the results obtained.

No other defect predictor has ever had both precision and recall above 90%.

VII. Conclusions and Future Work

In this paper, we have made the following contributions:

- We have introduced the concept of *change bursts*—consecutive changes over a period of time—and motivated why they would indicate problems during software development.
- We have presented *metrics based on change bursts* that can be used to predict defect-prone components.
- We have shown that these metrics yield *excellent predictive power* in projects with high-quality changes, with precision and recall exceeding 90% for Windows Vista—the highest predictive power ever observed.

Regarding future work, there are several ways to build on this study. One can further improve the predictive power (especially for less organized development processes like the one in ECLIPSE), consider alternate metrics, or build recommendation systems. To enable easy replication of this study, and to conduct alternate experiments, we have compiled datasets that contain all the necessary metrics and defect data for ECLIPSE; Appendix A gives the necessary instructions, including the R code fragments for building and applying regression models, running random split experiments, and computing precision and recall.

For practical purposes (at least within Microsoft), our results imply that defect prediction has reached a stage where it is, simply put, *good enough.* Our future work will thus concentrate on more *qualitative* topics: Why is it that change bursts are so important in predicting defects? What is so special about these activities? And what can we do to detect and avoid them? In all their generality, such questions are sure to keep us busy for several years.

Acknowledgments. We thank the Microsoft Windows product group for its cooperation in this study. Yana Mileva and the anonymous reviewers gave helpful comments on earlier revisions of this paper.

References

[1] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *ICSE '08*, 2008, pp. 521–530.

[2] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR '05*, 2005, pp. 1–5.

[3] 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 Softw. Engg.*, vol. 13, no. 5, pp. 539–559, 2008.

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

[5] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in *PROMISE '07*, 2007, p. 9.

[6] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *ISESE '06*, 2006, pp. 18–27.

[7] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *ICSE '08*, 2008, pp. 181–190.

[8] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *ESEC/FSE '09*, 2009, pp. 121–130.

[9] V. R. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments," *IEEE Trans. Softw. Eng.*, vol. 25, no. 4, pp. 456–473, 1999.

[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, pp. 653–661, 2000.

[11] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ISSTA '04*, 2004, pp. 86–96.

[12] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.

[13] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *ISSRE '96*, 1996, p. 364.

[14] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 476–493, June 1994.

[15] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, pp. 751–761, October 1996.

[16] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: an industrial case study," in *ICSE '99*, 1999, pp. 345–354.

[17] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *ICSE '06*.

[18] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM '07*.

[19] G. Denaro and M. Pezzè, "An empirical evaluation of fault-proneness models," in *ICSE '02*.

[20] T. M. Khoshgoftaar and R. M. Szabo, "Improving code churn predictions during the system test and maintenance phases," in *ICSM '94*, 1994, pp. 58–67.

APPENDIX

To facilitate reproduction of this study, Figure 6 documents the R script we have used to compute precision, recall, and accuracy as shown in this paper. The script takes a tabulator-separated table organized as follows:

Module	#Defects	#Changes	#Consecutive Changes	
A.dll	0	45	23	...
B.dll	2	33	21	...
⋮	⋮	⋮	⋮	⋱

In this table, `Module` is the name of the module in question; `NumberOfDefects` is the number of defects observed in that particular module. The columns `NumberOfChanges`, `NumberConsecutiveChanges`, etc., are the individual metrics as defined in Section III. Thus, to replicate this study:

1) Obtain the defect density for the modules in your study subject (possibly from a public source, such as [5]).
2) Implement the metrics as defined in this paper. For ECLIPSE, we have made a full dataset available for download [5].
3) Run the script.
4) Find results in `median_precision`, `median_recall`, and `median_accuracy`, respectively.

```
# Read the data
thedata ← read.table("...", header=T, sep="\t")

# Classify items with at least one defect as defect-prone
fp_threshold ← 0
thedata$HasDefect ← thedata$NumberOfDefects > fp_threshold

# Set a seed to make experiments deterministic.
set.seed(98052)

# Initialize result vectors with NA
precision ← rep(NA,N)
recall ← rep(NA,N)
accuracy ← rep(NA,N)

# Run 100 random experiments
N ← 100
for (i in 1:N) {
    # Split the data into training set (2/3) and testing set (1/3)
    # based on random sample idxs
    idxs ← sample(1:nrow(thedata), nrow(thedata)*2/3, F)
    train = thedata[idxs,]
    test = thedata[-idxs,]

    # Build a logistic regression model from the training data.
    train.lm ← glm(HasDefect ~ NumberOfChanges
      + NumberConsecutiveChanges + NumberCodeBursts
      + TotalBurstSize + MaximumCodeBurst
      + NumberOfChangesEarly
      + NumberConsecutiveChangesEarly
      + NumberCodeBurstsEarly + TotalBurstSizeEarly
      + MaximumCodeBurstEarly + NumberOfChangesLate
      + NumberConsecutiveChangesLate
      + NumberCodeBurstsLate
      + TotalBurstSizeLate + MaximumCodeBurstLate
      + TimeFirstBurst + TimeLastBurst + TimeMaxBurst
      + PeopleTotal + MaxPeopleInBurst + TotalPeopleInBurst
      + log(ChurnTotal+1) + log(MaximumChurnInBurst+1)
      + log(TotalChurnInBurst+1),
        data=train, family="binomial")

    # Apply the logistic regression model to the testing data
    # (cut off 0.50)
    test.prob ← predict(train.lm, test, type="response")
    test.pred ← test.prob>=0.50

    # Count true negatives, false negatives,
    # false positives, and true positives
    outcome ← table(factor(test$HasDefect, levels=c(F,T)),
                    factor(test.pred, levels=c(F,T)))
    TN = outcome[1,1]
    FN = outcome[2,1]
    FP = outcome[1,2]
    TP = outcome[2,2]

    # Compute precision, recall, and accuracy for experiment i
    precision[i] ←
      if (TP + FP == 0) { 1 } else { TP / (TP + FP) }
    recall[i] ← TP / (TP + FN)
    accuracy[i] ← (TP + TN) / (TN + FN + FP + TP)
}

# Compute median precision, recall, and accuracy
# for the 100 experiments
median_precision ← median(precision)
median_recall ← median(recall)
median_accuracy ← median(accuracy)
```

Figure 6. The R script used for conducting the experiments in this paper.