ConPredictor: Concurrency Defect Prediction in Real-World Applications

Tingting Yu, *Member, IEEE,* Wei Wen, Xue Han, Jane Huffman Hayes *Member, IEEE*

Abstract—Concurrent programs are difficult to test due to their inherent non-determinism. To address this problem, testing often requires the exploration of thread schedules of a program; this can be time-consuming when applied to real-world programs. Software defect prediction has been used to help developers find faults and prioritize their testing efforts. Prior studies have used machine learning to build such predicting models based on designed features that encode the characteristics of programs. However, research has focused on sequential programs; to date, no work has considered defect prediction for concurrent programs, with program characteristics distinguished from sequential programs. In this paper, we present ConPredictor, an approach to predict defects specific to concurrent programs by combining both static and dynamic program metrics. Specifically, we propose a set of novel static code metrics based on the unique properties of concurrent programs. We also leverage additional guidance from dynamic metrics constructed based on mutation analysis. Our evaluation on four large open source projects shows that ConPredictor improved both within-project defect prediction and cross-project defect prediction compared to traditional features.

Index Terms—Concurrency, defect prediction, software quality, software metrics

1 Introduction

Software quality assurance is an expensive activity: it requires time and resources to be performed properly, and it delays a product's delivery to market. This high-cost issue is more challenging in many of today's concurrent software systems due to their complicated behaviors. For example, assuring the quality of concurrent programs is difficult primarily because testing faces this challenge: concurrency faults are sensitive to execution interleavings that are imposed by various concurrency constructs (e.g., synchronization operations). Testing usually requires exploring as many interleavings as possible to amplify the chance of exposing faults. Recent work [21] reports that testing concurrent programs can introduce a 10x-100x slowdown for each test run. Such overhead increases as test suite size increases. Therefore, it is desirable to determine which code regions are more likely to contain concurrency faults as this can guide developers to focus the testing efforts on the identified code, thus reducing the time and resources required for testing and leading to reduced quality assurance costs.

For this reason, defect prediction has been an active research area in software engineering [48], [57], [58]. Defect prediction techniques build models from software data and use the models to predict whether new instances of code regions, e.g., files, changes, and methods, contain defects. These techniques first design features or combinations of features and then use machine learning algorithms to build prediction models. Based on the prediction results, developers can allocate limited testing efforts more effectively to focus on the defect-prone modules. In particular, source code metrics have been used widely and shown effectiveness in prediction. There has been much research on software defect prediction by combining static code metrics to identify defect-prone source code artifacts [101]. A variety of statistical and machine learning techniques have been used to build defect prediction models [14].

However, all existing defect prediction research has focused on sequential software. To date, no work has considered concurrent software systems for which adapting existing prediction models may not be effective. Unlike defect prediction for sequential programs, that often relies on a set of well-defined and traditional code metrics (e.g., lines of code, cyclomatic complexity), predicting concurrency defects in concurrent programs must consider the unique concurrency properties in its fault models: threads, shared variable accesses between threads, and synchronization operations.

Another challenge is that most existing research has focused on designing features from static code analysis. However, the performance of defect prediction can also rely on the quality of the test suite [10]. This is because the performance of a defect prediction model should be measured as its ability to predict faults that ultimately lead to test failures. Although recent work [10] has proposed using mutation analysis to guide defect detection, it focuses on sequential programs. None of the existing research has considered using either static or dynamic metrics to predict defects for concurrent programs.

In this paper, we propose ConPredictor, a defect prediction framework for predicting functions that are likely to contain concurrency defects in real-world applications. Specifically, we propose six novel code metrics specific to concurrent programs by taking unique features related to concurrency properties into account. We adapt the concurrency control flow graph (CCFG) to generate code metrics involving: (i) concurrent cyclomatic complexity, (ii) number of shared variables, (iii) number of conditional basic blocks that contain concurrency constructs, (iv) number of communication edges in CCFG, (v) number of synchronization operations, and (vi) access distance between shared variables in a local thread. We then define 18 mutation metrics computed by applying a variety of mutation operators specific to concurrent pro-

- *All authors are with the Department of Computer Science, University of Kentucky, Lexington, Kentucky, USA, 40506.*
  E-mail: tyu@cs.uky.edu, wei.wen0@uky.edu, xha225@g.uky.edu, hayes@cs.uky.edu

Manuscript received June 27, 2017; revised November 16, 2017.

grams. These metrics include six static metrics (e.g., the number of times a mutation operator is applied) and 12 dynamic metrics from dynamic mutation analysis. The ConPredictor prediction model is built upon all 24 static and dynamic concurrency metrics. Then, we empirically compare the performance of ConPredictor to those of prediction models built using traditional metrics that have been widely used in previous fault prediction work [57]. We also investigate whether the combined use of mutation metrics and source code metrics improves the accuracy of the resulting prediction model. Moreover, we examine the extent to which different machine learning techniques benefit from the metrics in ConPredictor. We also determine the best combination of metrics for predicting testability.

To evaluate our approach, we apply it to four large real-word systems. Our primary finding is that ConPredictor can significantly improve the prediction performance, with large effect sizes, when comparing to the traditional metrics for sequential programs. The dynamic concurrency metrics are even more effective than the static concurrency metrics. Our paper makes the following contributions:

1) The first approach to effectively predict concurrency faults,
2) A set of novel source code metrics specific to concurrent programs,
3) The introduction of dynamic concurrency metrics for fault prediction, and
4) An empirical study showing the effectiveness of our approach.

This article draws on our previously published conference paper [98], in which we propose a set of novel static code metrics to predict testability of concurrent programs. The testability is calculated based on mutation analysis. We have extended this work substantially by introducing a new static metric and a set of dynamic concurrency metrics. We then use both static and dynamic metrics to predict concurrency faults. All concurrency faults considered in this work are real. We have performed an empirical study that is different from our previous work in terms of research questions, study design, results, and analysis.

The remainder of the paper is organized as follows. Section 2 presents background and definitions. Our approach and concurrency code metrics are introduced in Section 3. Section 4 discusses the empirical study. Results are presented in Section 5 followed by discussion in Section 6. Prior work is presented in Section 7 and Section 8 concludes.

2 Background and Definitions

In this section, we provide background information on defection prediction and mutation analysis. Related work is discussed further in Section 7.

2.1 Software Defect Prediction

A software defect prediction model generally exploits historical data to classify software modules as either faulty or non-faulty. A prediction model infers a single aspect of the data (i.e., dependent variable) from a combination of other aspects of the data (e.g., independent variables). In the software fault prediction context, the dependent variable is the label indicating whether a software module contains a fault or not while the independent variables can

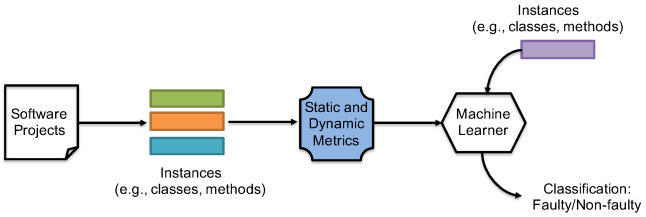


Fig. 1: Process of Predicting Software Faults

be related to different aspects of the software such as source code metrics. Figure 1 illustrates the process of defect prediction.

The performance of the fault prediction model depends both on the modeling technique and the independent variables (i.e., metrics) used. Classification techniques (e.g., Decision Trees, Logistic Regression, and Naive Bayes) have been widely used to build fault predication models [35], [56], [69]. However, according to recent systematic literature reviews [32], [79], the choice of a modeling technique seems to have less impact on the classification accuracy of a model than the choice of a metrics set.

Feature selection has been used to select a set of most relevant independent variables contained in the original dataset to eliminate variables that do not contribute to the performance of prediction, and can thus improve learning efficiency and increase prediction accuracy [34]. Feature selection is usually performed by leveraging a machine learning algorithm that can evaluate the usefulness of the feature set (i.e., wrappers [53]). This can also be done by ranking methods (i.e., filters) that evaluate the features according to heuristics based on general characteristics of the data.

The performance of a classification model is typically evaluated based on the confusion matrix. The matrix contains four instances: True Positive (TP) – faulty components correctly classified as faulty; False Negative (FN) – faulty components incorrectly classified as non-faulty; False Positive (FP) – non-faulty components incorrectly classified as faulty; and True Negative (TN) – non-faulty components correctly classified as non-faulty. The confusion matrix values are used to calculate a set of evaluation measures, including precision (measuring the proportion of the components classified as faulty which are actually faulty), recall (measuring the proportion of faulty components classified as faulty), and F-Measure (which is the harmonic mean of precision and recall).

2.2 Mutation Analysis

Mutation testing is an approach for evaluating test suites and testing techniques using a large number of systematically seeded program changes, allowing for statistical analysis of results [3], [41]. Mutation testing typically involves three stages. (1) Mutant generation – in this stage, a predefined set of mutation operators are used to generate mutants from program source code or bytecode. A mutation operator is a rule that is applied to a program to create mutants, such as arithmetic operator replacement (AOR) [92]. (2) Mutant execution – in this stage, the goal is execution of test cases against both the original program and the mutants. (3) Result analysis – in this stage, the goal is to check the mutation score obtained by the test suite, where mutation score is defined as the ratio of the number of killed mutants to the number of all (non-equivalent) generated mutants.

TABLE 1: List of Sequential Mutation Operators

index	*Operator*	*Description*
1	ssdl	statement deletion
2	swdd	while replacement by do-while
3	oasn	arithmetic operator by shift operator
4	oeba	plain assignment by bitwise assignment
5	olng	logical negation
6	orrn	relational operator mutation

TABLE 2: List of Concurrency Mutation Operators

index	*Operator*	*Description*
1	rmlock	Remove call to lock/unlock
2	rmwait	Remove call to cond_wait/cond_timedwait
3	rmsig	Remove call to cond_signal/cond_broadcast
4	rmjoinyld	Remove call to join/yield
5	shfecs	Shift critical section
6	spltecs	Split critical section

Mutants that contain a single fault are called first-order mutants. First-order mutants have been widely used for mutation analysis of sequential programs. Table 1 lists six commonly used mutation operators for sequential programs [1], including operator indexes, operator names, and descriptions. These operators are used in our study.

There has been some work to generate first-order mutants for concurrent programs [11], [25]. For example, Ghosh generates concurrency-related mutants by removing single synchronization keywords [25]. Bradbury et al. proposed a set of first-order mutation operators for Java [11]. However, more recent work has shown that first-order mutants are not sufficient to simulate subtle concurrency faults due to the complexity of thread synchronizations [29], [46], [54]. Therefore, some research has investigated higher-order mutants [33], [40] for concurrent mutation operators [46] by inserting two or more faults. Higher-order mutants subsume first-order mutants, as killing the former is a sufficient but not necessary condition for killing the latter.

To generate higher order mutants for concurrent programs, Kaiser et al. [46] propose a set of mutation operators for multi-threaded Java programs based on concurrency bug patterns that include subtle concurrency faults (e.g., data races). Kusano et al. [54] implemented *CCmutator* based on the Clang/LLVM compiler framework to inject concurrency faults for multithreaded C/C++ applications. Their work considers both first-order and higher-order mutants. In this work, we consider various concurrency-related mutation operators from *CCmutator*. Table 2 summarizes the mutation operators used in this paper.

These operators include mutex locks, condition variables, atomic objects, semaphores, thread creation, and thread join. For example, removing a lock-unlock pair can create potential data races and atomicity violations, removing a conditional wait can create order violations, shifting and splitting critical sections can introduce potential data races and order violations as some variables are no longer synchronized. Replacing a call to join with a call to sleep can cause nondeterministic behavior.

We use the term location to indicate a place in a program where a fault can occur. Although the techniques we propose can be used at different granularities (e.g., statements), this paper concentrates on locations that correspond to single program instructions.

3 Concurrency-related Code Metrics

Figure 1 shows the process of fault prediction in this work. First, we define instances as units of programs, these can be files, classes, or functions. The instances that we consider are at the function level. We label an instance as faulty if it has any concurrency faults, or non-faulty otherwise. The next step is to compute static and dynamic metrics. The static metrics consist of static code metrics and static mutation metrics, and the dynamic metrics consist of coverage metrics and dynamic mutation metrics. To obtain static code metrics, we perform static analysis on the concurrency control flow graph (CCFG). We obtain the static mutation metrics by recording the number of times each mutation operator is applied. We then execute test cases on the instances to compute metrics based on code coverage. Next, we apply concurrency mutation operators to the instances and execute test cases to collect dynamic mutation metrics. Finally, we train prediction models using machine learning algorithms implemented in Weka [30]. The trained prediction models classify instances as faulty or non-faulty.

In the following sections, we describe the approach to computing the static and dynamic metrics for concurrency fault prediction.

3.1 Concurrency Control Flow Graph

A concurrent program P consists of threads that communicate with each other through shared variables and synchronization operations. Given the program source code, we can construct a concurrent control flow graph (CCFG) for a procedure $p \in P$ based on a flow and context-sensitive pointer analysis, where p can be accessed by multiple threads. The idea of building CCFGs is not new and there has been research on using CCFGs to achieve different objectives [24], [45]. For example, Kahlon et al. [45] build a context-sensitive CCFG to perform staged data race detection. Our CCFG is similar to those used in existing work but is implemented to satisfy our goal of predicting concurrency faults. First, p is constructed into a control flow graph (CFG), denoted as $(N(p),\ E(p))$. A node $N(p)$ is an instruction I and an edge $I_i \rightarrow I_j \in E(p)$ describes the control flow and data flow between nodes in this CFG. In the CCFG, we add additional edges to represent communications between procedures potentially running on two different threads. Figure 2 illustrates an example CCFG, where the solid lines reflect the local edges and the dotted lines reflect the cross-threads' edges, including `fork`, `join`, and `communication` edges. The `Main` function creates two threads, on which functions `foo` and `bar` are running, respectively. The variables marked as bold are shared between threads. For readability purposes, we use statements rather than instructions to represent each node.

Specifically, a `fork` edge is added from the program location where `thread_create` instruction is called to the entry node of the procedure to be executed. In Figure 2, edges $< 1, foo >$ and $< 1, bar >$ form two fork edges. If the thread on which the procedure is to be executed is specified as in the thread pool model, the procedure is duplicated on the other thread. A `join` edge is added from the return of a procedure that is executed by the fork to the node representing `thread_join` instruction. In Figure 2, edges $< 14, 3 >$ and $< 24, 3 >$ are considered to be join edges. A `communicate` edge is added from a write of one shared variable (SV) on one thread to the read of the same SV on the other

thread. For example, Figure 2 contains two communication edges involving two shared variable pairs $< 11, 17 >$ and $< 18, 7 >$.

3.2 Static Metrics

We introduce five code metrics and ten mutation metrics specific to concurrent programs.

3.2.1 Static Code Metrics

Static code metrics are generated from the CCFG.

Synchronization point count (SPC). We define the *synchronization point count*, $SPC(f)$, as the number of nodes involving synchronization operations (SOs) in a function f. The use of the SPC metric is based on the intuition that the number of SOs contributes to the complexity of concurrent programs. As the number of SOs increase, the program is more likely to contain more faults related to synchronization usage, such as deadlock and atomicity violations. The SPC metric for a function is defined as:

$$SPC(f) = num\_of\_syncs(f)$$

Here, we consider mutex, semaphore, conditional variables, and barriers as synchronization operations. In the Figure 2 example, $SPC(main)$ = 2, $SPC(foo)$ = 2, and $SPC(bar)$ = 2, because each function contains two synchronization operations (e.g., `mutex_lock`).

Shared variable count (SVC). In this metric, we count the nodes in the CCFG involving shared variable (SV) read/write in the procedure p. The shared variables can affect data communication between threads. The increasing complexity of SV usage is likely to cause incorrect data state to propagate across threads. As such, we define SVC for p as:

$$SVC(f) = num\_of\_SVs(f)$$

In Figure 2, $SVC(foo)$ = 2 and $SVC(bar)$ = 2. The variables marked with bold are SVs. Note that we do not count SVs passed as lock objects. $SVC(main)$ = 0, because there are no global variables in the `main` function.

Conditional synchronization count (CSC). We define *conditional synchronization count*, $CSC(f)$, as the number of conditional basic blocks (e.g., branches) within function f that contain at least one shared variable or synchronization operation. The CSC metric takes into account the local control flow of a procedure in one thread that can potentially complicate the communication with procedures running on other threads. The intuition is that the conditional block containing concurrency elements increases the complexity of a function in terms of multithreading communication, affecting the sensitivity of inputs that reach such synchronization points. The CSC metric for f is defined as:

$$CSC(f) = num\_of\_cond\_syncs(f)$$

As Figure 2 shows, $CSC(foo)$ = 2, because there are two conditional blocks in `foo` that contain SVs (i.e., `while`, `if`). The shaded areas are irrelevant basic blocks that CSC does not count. Note that $CSC(bar)$ = 0 and $CSC(main)$ = 0, because the two functions contain no conditional basic blocks containing SVs or synchronization points (the two shaded conditional basic blocks in `bar` are irrelevant basic blocks).

Communication edge count (CEC). The *communication edge count* metric, denoted by $CEC(f)$, counts the number of communication edges involved across all shared variables in a function f. Since communication edges indicate how different threads can interleave, CEC reflects the complexity of interleaving space. In this case, the CEC value may be proportional to the likelihood of exposing concurrency faults.

$$CEC(f) = num\_of\_comm\_edges(f)$$

In Figure 2, $CEC(main)$ = 4, $CEC(foo)$ = 4, and $CEC(bar)$ = 4. Each of the four functions contains four communication edges.

Concurrency cyclomatic complexity (CCC). We extend the traditional McCabe's cyclomatic complexity [65] to measure complexity of concurrent programs, denoted as *concurrency cyclomatic complexity* (CCC). To compute CCC, we first prune the CCFG to transform it into CCFG', which involves two steps: 1) remove each basic block b that is irrelevant to concurrency properties (i.e., SVs and synchronizations) computed by the CSC metric, as well as remove their incoming and outgoing edges; and 2) add an auxiliary edge from each predecessor of b to each successor of b. Thus, the $CCC(p)$ of a function is defined with reference to its CCFG':

$$CCC(f) = E' - N' + 2$$

Here, E' is the number of edges and N' is the number of nodes in the f of CCFG, where E' includes both ingoing and outgoing edges for f. In Figure 2, the shaded nodes are irrelevant and thus removed. Auxiliary edges are added from node 8 to node 13, and from node 19 to node 24. Thus, $CCC(main)$ = 8 - 5 + 2 = 5, $CCC(foo)$ = 15 - 10 + 2 = 7, and $CCC(bar)$ = 10 - 7 + 2 = 5. Note that the sequential Cyclomatic Complexity of the three functions are 5 - 4 + 1 = 2, 12 - 11 + 2 = 3, and 11 - 11 + 2 = 2.

Shared variable access distance (SVD). The distance between two shared variable (SV) accesses in one thread is also an important factor for concurrency fault exposure [60]. For example, if `x` is written at l_1 and later read at l_2 by the same thread $T1$, and there exists a different thread $T2$ that updates the value of `x`, the distance between l_1 and l_2 can impact the chances of $T2$ interleaving between them. We consider access distance as the instruction gap between two SV accesses (reads or writes) in the same procedure p. To compute $SVD(f)$, we first identify all SV pairs (SVP) in p. For each SV pair $< sv1, sv2 >$, we calculate all instruction gaps by traversing all path segments[1] between $sv1$ and $sv2$. Note that one SVP can associate with multiple distance values due to possible control flow edges. To consider all path segements, we calculate the mean distance over all path segments. Specifically, we average all distance values if they are normally distributed, otherwise we use their trimmed mean. Thus, $SVD(f)$ is defined as:

$$SVD(f) = \overline{\sum_{i=1}^{N} \sum_{i=j}^{M} Dis(SVP_{i,j})}$$

Here, N is the number of shared variable pairs and M is the number of path segments for an SVP. In the function `foo` of Figure 2, there are two path segments: (7,8,10,11) and (11,12, 13, 6,7) for SVPs <7, 11> and <11, 7>, respectively. Suppose each node counts for 2 instructions, then $SVD(foo)$ = (8 + 10) / 2 = 9.

1. A path segment is a path slice for which every node is visited at most once.

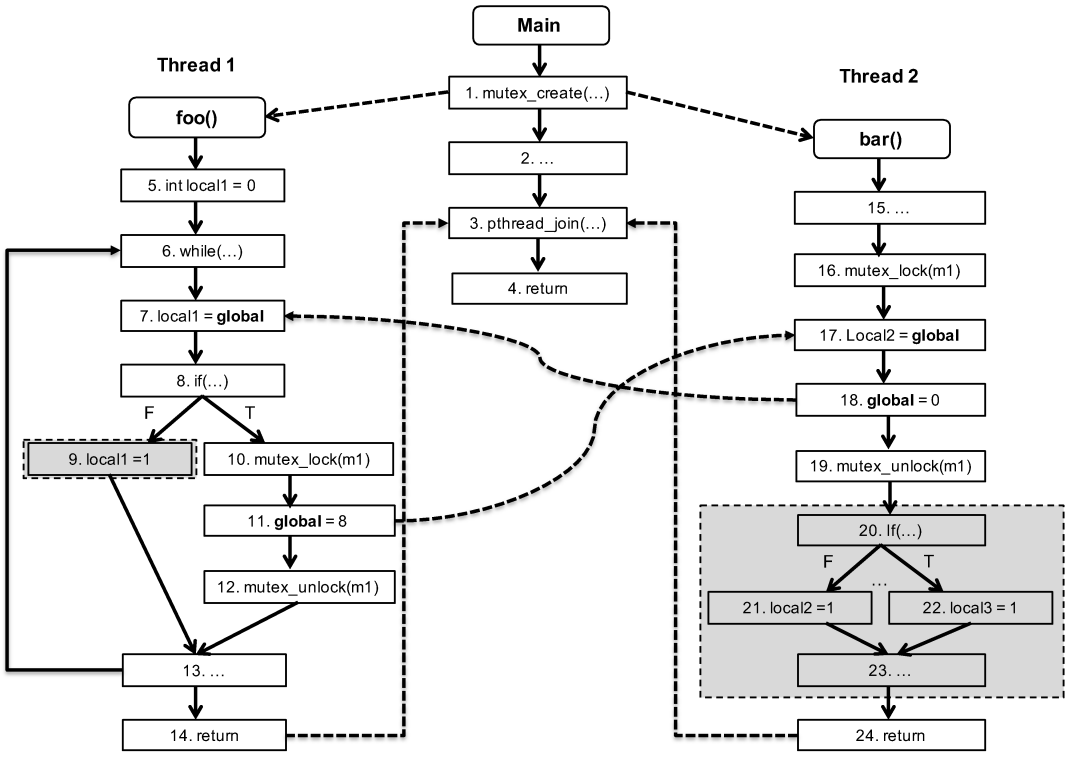


Fig. 2: Concurrency Control Flow Graph

3.3 Mutation Metrics

Most previous fault prediction studies have used static code metrics for this prediction task. Dynamic information resulting from test executions can also be leveraged for guiding fault prediction. In this paper, we use mutation testing to produce both static and dynamic metrics. We choose mutation testing for two reasons. First, the quality of the test suite is a considerable factor for finding faults: mutation testing has been used to assess the quality of a test suite [20], [52]. Second, empirical study [4], [44] has shown that test cases that are good at killing mutants are good at detecting real faults. Therefore, it has been validated that mutants can be a legitimate substitute for real faults in testing.

We consider three metric suites: one static metric suite and two dynamic metric suites.

3.3.1 Static Mutation Metrics

The concurrency static mutation metric suite consists of a list of mutation metrics that simply record the number of times a particular mutation operator can be applied. They are static metrics because the metric data can be obtained statically without executing the code. Therefore, these metrics are independent of any test suites. Each mutation operator is associated with a metric. For instance, the six concurrency mutation metrics listed in Table 2 yield six static mutation metrics.

3.3.2 Dynamic Mutation Metrics

The dynamic mutation metrics take into account the test suite as well as the potential faults that can be seeded. For each type of concurrency mutation operator defined in Section 2.2, we compute two corresponding dynamic metrics: the percentage of mutants (with respect to a mutation operator) that are executed (MuDuE), and the percentage of mutants (with respect to a mutation operator) that are killed (MuDuK). This results in 12 (2×6) dynamic metrics, each of which contributes to an independent variable of the prediction model. Each metric can also assess the quality of the test suite regarding whether or not the test suite can exercise a particular simulated fault (covered mutant), and whether the test suite can detect this simulated fault (killed mutant).

3.4 Implementation

Our metrics are implemented using the popular Clang/LLVM compiler platform [55] using the LLVM opt pass [59] to collect program information. Clang's CFG provides a directed graph for each function, where the nodes are the basic blocks and the directed edges represent how the control flows. As noted above, our CCFG extends the basic CFG by adding edges describing inter-thread communication. We apply shared variable analysis to identify variables shared by two threads, such as heap objects and data objects that are passed to a function (e.g., thread starter function) called by another thread. Since our metrics implemented by LLVM's intermediate representation (IR) are based on single static assignment (SSA) form, we can potentially leverage compiler front-ends to handle other languages. We leverage CCMutator [54], a concurrency mutation tool, to generate mutants of concurrent programs.

4 Empirical Study

In this section, our goal is to evaluate the effectiveness of ConPredictor. We use four metric suites consisting of 48 metrics in this evaluation, shown in Table 3. The CStaMc and CDyMc are static and dynamic metric sets used in ConPredictor. In addition to ConPredictor, we use SPM, a baseline defect predictor built using 24 metrics, where SStaMc indicates the static metric set and

TABLE 3: List of Static Code Metrics

Predictor	Metric Suite		
ConPredictor	CStaMc	*Concurrency Code Metrics (CCM)*	*Description*
		ConcurrencyComplexity(CCC)	Concurrent program complexity
		CountSharedVaraible (CSV)	# of shared variables
		CountConditionalBasicBlock (CBB)	# of conditional basic blocks
		CountSynchronizations(CSO)	# of synchronization operations
		CountComEdges(CCE)	# of communication edges
		CountDistance(CCD)	average distance between shared variables
		Con. Stat. Mut. Met. (CSMM)	*Description*
		MuS$_{rmlock}$	# of generated mutants on $rmlock$
		MuS$_{rmwait}$	# of generated mutants on $rmwait$
		MuS$_{rmsig}$	# of generated mutants on $rmsig$
		MuS$_{rmjoinyld}$	# of generated mutants on $rmjoinyld$
		MuS$_{shfecs}$	# of generated mutants on $shfecs$
		MuS$_{spltecs}$	# of generated mutants on $spltecs$
	CDyMc	*Con. Dyn. Mut. Met. (CDMM)*	*Description*
		MuDuE$_{rmlock}$	% of mutants executed on $rmlock$
		MuDuE$_{rmwait}$	% of mutants executed on $rmwait$
		MuDuE$_{rmsig}$	% of mutants executed on $rmsig$
		MuDuE$_{rmjoinyld}$	% of mutants executed on $rmjoinyld$
		MuDuE$_{shfecs}$	% of mutants executed on $shfecs$
		MuDuE$_{spltecs}$	% of mutants executed on $spltecs$
		MuDuK$_{rmlock}$	% of mutants killed on $rmlock$
		MuDuK$_{rmwait}$	% of mutants killed on $rmwait$
		MuDuK$_{rmsig}$	% of mutants killed on $rmsig$
		MuDuK$_{rmjoinyld}$	% of mutants killed on $rmjoinyld$
		MuDuK$_{shfecs}$	% of mutants killed on $shfecs$
		MuDuK$_{spltecs}$	% of mutants killed on $spltecs$
SPM	SStaMc	*Sequential Code Metrics (SCM)*	*Description*
		CountFunctionIn (CN)	# of functions that call a given function
		CountFunctionCall(CM)	# of functions called by a given function
		CountLocalVar (CL)	# of local variables in the body of a method
		CountParameters (CPA)	# of parameters for a function
		ComToCo (CTC)	the ratio of comments to source code
		CountPath (CP)	the ratio of possible paths in the body of a function
		CyclomaticComplexity (CC)	Mcabe's cyclomatic complexity
		ExecStmt (ES)	# of executable source code statements
		MaxNesting (MN)	maximum nested depth of all control structures
		Seq. Stat. Mut. Met. (SSMM)	*Description*
		MuS$_{ssdl}$	# of generated mutants on $ssdl$
		MuS$_{swdd}$	# of generated mutants on $swdd$
		MuS$_{oasn}$	# of generated mutants on $oasn$
		MuS$_{oeba}$	# of generated mutants on $oeba$
		MuS$_{olng}$	# of generated mutants on $olng$
		MuS$_{orrn}$	# of generated mutants on $orrn$
	SDyMc	*Seq. Dyn. Mut. Met. (SDMM)*	*Description*
		MuS$_{ssdl}$	# of mutants executed on $ssdl$
		MuS$_{swdd}$	# of mutants executed on $swdd$
		MuS$_{oasn}$	# of mutants executed on $oasn$
		MuS$_{oeba}$	# of mutants executed on $oeba$
		MuS$_{olng}$	# of mutants executed on $olng$
		MuS$_{orrn}$	# of mutants executed on $orrn$
		MuDuK$_{ssdl}$	% of mutants killed $ssdl$
		MuDuK$_{swdd}$	% of mutants killed $swdd$
		MuDuK$_{oasn}$	% of mutants killed $oasn$
		MuDuK$_{oeba}$	% of mutants killed $oeba$
		MuDuK$_{olng}$	% of mutants killed $olng$
		MuDuK$_{orrn}$	% of mutants killed $orrn$

SDyMc indicates the dynamic metric set. We next consider three research questions.

RQ1: Can ConPredictor improve defect prediction performance compared to the metrics used for sequential programs?

RQ2: What metrics are particularly effective contributors to concurrency defect prediction improvement?

RQ3: Can a combination of concurrent program metrics be used to predict concurrency faults of new instances (i.e., functions) in a new project?

RQ1 lets us evaluate the defect prediction performance of ConPredictor, compared to the baseline approach in which code and mutation metrics for sequential programs are used. We also split this overall question into three sub questions, which ask about the effects of static metrics, dynamic metrics, and machine learning on defect prediction performance.

RQ1.1: Can applying feature selection improve the performance of defect prediction?

RQ1.2: How effective are the static metrics at predicting concurrency faults?

RQ1.3: How effective are the dynamic metrics at predicting concurrency faults?

RQ1.4: How can different machine learners affect the perfor-

TABLE 4: Object Program Characteristics

Program	*NLOC*	*versions*	*Total inst.*	*Selected insts.*	*Faulty insts.*	*Increase*	*Faults.*	*mutants*	*tests*	$mutants_e$	$mutants_k$
APACHE	128K - 201K	10 (v2.0 - v2.2)	35,761	2,164	70	3000%	51	2,644	2,972	62%	39%
MYSQL	199K - 438K	10 (v5.0 - v5.6)	71,665	2,478	230	1000%	184	2,267	1,813	71%	33%
MOZILLA	1,120K - 1,268k	10 (v4 - v34)	144,382	4,988	142	3500%	103	4,908	2,972	55%	31%
OPENOFFICE	3,033K - 4,138K	10 (v1.0 - v4.1)	110,509	6,835	116	5800%	82	5,097	3,846	62%	32%

mance of ConPredictor?

RQ2 investigates the contribution of different metrics for fault prediction because choosing a different machine learner can produce different performance results. RQ3 lets us evaluate whether the proposed technique is effective when applied to a new project.

4.1 Objects of Analysis

We study four large concurrent software projects: Apache, MySQL, Mozilla, and OpenOffice. We selected these subjects because with millions of lines of publicly accessible code and well maintained bug repositories, they have been widely used by existing bug characteristic studies [43], [96], [99] and concurrency fault detection and testing techniques [21], [60]. In addition, comparing to medium and small projects, they contain a number of concurrency bugs that are more appropriate for training datasets. All four projects started in the early 2000's and each has over ten years of bug reports. The subject programs cover various application spectrums - the world's most used HTTP server, the world's most popular database engine, a leading web browser suite, and a popular office suite. Server applications mostly use concurrency to handle concurrent client requests. They can have hundreds or thousands of threads running at the same time. Client and office applications mostly use concurrency to synchronize multiple GUI sessions and background working threads. Table 4 lists our object versions along with some of their characteristics. Column 2 lists the number of lines of non-comment code (NLOC). Other columns are described later.

There are not enough concurrency bugs in a single application version to build classification models (usually 2-5 concurrency bugs per version). Therefore, we collected data from multiple versions of each application.

We randomly selected 10 versions for each application released between 2000 and 2014. Column 3 of Table 4 lists the number of versions and release period of each subject. Column 4 lists the number of all function instances in all 10 versions of each application. Since ConPredictor targets concurrent programs, we selected function instances that can be executed by multiple threads. We then removed redundant instances across multiple versions. As a result, a total of 16,465 functions (Column 5 in Table 4) were identified for use.

We labeled a function as buggy if it contains at least one concurrency bug that was reported in bug reports or release notes (that contain bug IDs). We searched bug reports for the studied application versions using a set of concurrency-related keywords (e.g., "race(s)," "deadlock," "atomic," "concurrency," "synchronization(s)," "mutex(es)"). These keywords have been used by existing concurrency bug detection techniques [61], [97]. We filtered out unconfirmed reports. We then manually identified functions containing at least one reported bug.

Column 6 in Table 4 lists the number of faulty function instances. Column 7 lists the number of concurrency bugs. While having a larger number of bugs may yield better evaluation, the cost of the manual process is quite high: the understanding and preparation of the object used in the study and the conduct of the study required between 150 and 180 hours of researcher time. Other columns are described later.

The imbalanced datasets may affect the accuracy of defect prediction [87]. Table 4 shows that only 3.4% of the instances are buggy and thus the data is imbalanced. To address this problem and improve defect prediction models, we perform the re-sampling technique used in existing work [87], i.e., SMOTE [15], on our training data for both concurrency and sequential code metric sets. Column 7 of Table 4 shows the percentage of increase of the minority class by the SMOTE filter.

4.2 Data collection

To compute mutation scores, we required mutants of our object programs. To seed sequential faults, we use Clang [55] to implement a mutation generation tool applying the mutation operators described in Figure 1. For concurrency mutants, we extended CCMUTATOR [54] to create concurrency mutants of the classes described in Section 2. This process left us with the numbers of mutants reported in Column 9 of Table 4. A total of 36 mutation metrics, including both static and dynamic metrics, are collected as shown in Table 3.

Test oracles are needed when evaluating whether a mutant is killed. These programs are released with existing test suites and with built-in oracles provided, and we used those. We also checked program outputs, including messages printed on the console and files generated and written by the programs.

We executed our test cases on all of the mutants of each object program. Column 10 of Table 4 lists the number of test cases. To control for variance due to randomization of thread interleavings, we ran each mutant 100 times. A mutant is marked as being executed or propagated if it does this at least once. We used a Linux cluster to perform the executions, distributing each job on a distinct node. The mutation score was computed by following the process described in Section 2.2. Columns 11-12 of Table 4 report the percentage of mutants executed and killed.

To gather static code metric data, for each function we first computed six concurrency-related code metrics. The method of computing concurrency metrics is described in Section 3.

We next computed sequential code metrics (SCM) used as the baseline approach. There are two traditional suites of code metrics: The Chidember-Kemmerer (CK) metrics [16] and metrics that are directly calculated at the method level [7], [86]. The CK metrics measure the size and complexity of various aspects of object-oriented source code and are calculated at the class level. CK metrics have been successfully applied for bug prediction in prior work [27]. The method level metrics are not limited to object-oriented source code, but include measures such as lines of code. When applying these metrics to source files, they are typically averaged or summed up over all methods that belong to a file [57], [76], [102]. Since our goal is to build fault prediction models for C/C++ programs at the function level, we do not use the CK metrics because they are not directly applicable to functions, e.g., depth of the inheritance tree. We choose instead the nine metrics used previoiusly [27], a method-level fault prediction

technique (SCM). The nine sequential metrics include the number of functions that call a given function (funIN), the number of functions called by a given function (funOut), the number of local variables in the body of a method (localVar), the number of parameters in the declaration (parameters), the ratio of comments to source code (comToCo), the number of possible paths in the body of a function (countPath), McCabe Cyclomatic complexity of a function (complexity), the number of executable source code statements (execStmt), and the maximum nested depth of all control structures (maxNesting).

Table 5 summarizes all concurrency metrics and sequential metrics. The static metric set of ConPredictor (i.e., CStaMc) consists of a static code metric set, denoted by CCM, and a static mutation metric set, denoted by CSMM. The dynamic metric set of ConPredictor (i.e., CDyMc) is equal to the dynamic mutation set, denoted by CDMM. On SPM, the static metric set (i.e., SStaMc) consists of a code metric set (SCM) and a mutation metric set (SSMM) specific to sequential programs. SPM's dynamic metric set uses a set of dynamic mutation metrics for sequential programs, denoted by SDMM. The sequential code metric set SCM is proposed in prior work [27] and used as a baseline approach in our study.

4.3 Techniques for Comparison

The ConPredictor basically combines all proposed static and dynamic concurrency metrics, i.e., CCM + CSMM + CDMM. To answer RQ1, we compare ConPredictor to SPM (i.e., SCM + SSMM + SDMM). The subquestion RQ1.1 isolates CStaMC from ConPredictor and compares it to ConPredictor, SStaMC, and CDyMC. The subquestion RQ1.2 compares CDyMc to ConPredictor, and SDyMC. To answer the subquestion RQ1.3, we compare the results of prediction models using four different classification algorithms widely adopted in defect prediction studies, including Decision Tree, Logistic Regression, Naive Bayesian, and Random Forest. To answer RQ2, we calculate the importance of each individual metric used in ConPredictor and examine their contribution in concurrency defect prediction. To answer RQ3, we apply the model learned from each dataset to predict concurrency faults in each of the other three datasets. We then evaluate the prediction performance of both ConPredictor and SPM on all nine pairs of comparison.

4.4 Prediction Models

We first performed feature selection to select effective metrics for use in constructing prediction models. To do this, we used the WEKA Wrapper Subset Selection Filter [31], [53] (`WrapperSubsetEval`), which performs a best first search algorithm to identify the subset of attributes that generalize best on the training set. The SS algorithm can resolve the multicollinearity problem between correlated features [2] and thereby avoid the model construction overfitting problem. The `WrapperSubsetEval` evaluates the power of a subset of metrics by considering the individual predictive ability of each metric along with the degree of redundancy between them. Metrics that are highly correlated with the faulty class while having a low inter-correlation are preferred.

Next, a classification algorithm was required to build the prediction model for each subject. In ConPredictor, we consider four classification techniques: Bayesian Network, J48 Decision Tree, Logistic Regression, and Random Forest in Weka [30]. We chose them because they are popular and have been shown to be effective at predicting defects in a recent study [26]. Naive Bayes (NB) [91] is a statistical technique which uses the combined probabilities of the different attributes to predict faultiness. Logistic Regression (LR) [18] is a regression technique which identifies the best set of weights for each attribute to predict the faulty or non-faulty class. J48 is a Java implementation of the C4.5 [78] decision tree algorithm which uses entropy information to determine which attribute to use as decision nodes. Random Forest (RF) [12] is an ensemble technique which aggregates the predictions made by a collection of decision trees (each with a subset of the original set of attributes). To examine our research questions, we applied the four models to different metric sets. The random forest algorithm was primarily used in our experiments because its performance was good, as noted in Section 5.1.4.

TABLE 5: Metrics selected in each metric suite for all subjects

Concurrency Metric Suite	*Selected metrics*
ConPredictor	CSV, CCC, CCD, CCE, CSO, MuS$_{rmlock}$, MuDuE$_{spltecs}$, MuDuE$_{rmsig}$, MuDuE$_{swptw}$, MuDuE$_{mwait}$, MuDuK$_{swptw}$, MuDuK$_{rmsig}$, MuDuk$_{rmlock}$
CStaMc	CSM, CCC, CCD, MuS$_{rmlock}$, MuS$_{swptw}$, MuS$_{rmsig}$.
CDyMc	MuDuE$_{rmsig}$, MuDuE$_{rmlock}$, MuDuE$_{swptw}$, MuDuE$_{mwait}$, MuDuE$_{spltecs}$, MuDuK$_{swptw}$, MuDuK$_{rmsig}$
Sequential Metric Suite	*Selected metrics*
SPM	CI, CP, CC, MuS$_{ssdl}$, MuS$_{oasn}$, MuS$_{orrn}$, MuDuE$_{swdd}$, MuDuE$_{oasn}$, MuDuK$_{ssdl}$, MuDuK$_{oeba}$
SStaMc	CI, CP, CC, CV, MuS$_{ssdl}$, MuS$_{olng}$, MuS$_{swdd}$
SDyMc	MuDuE$_{swdd}$, MuDuE$_{oasn}$, MuDuE$_{ssdl}$, MuDuK$_{ssdl}$, MuDuK$_{oeba}$, MuDuK$_{olng}$

To evaluate our prediction models, we again used 10-fold cross validation, widely used to evaluate prediction models [56], [70].

In 10-fold cross validation we randomly divide the dataset into ten folds. Of these ten folds, nine folds are used to train the classifier, while the remaining one fold is used to evaluate the performance (Section 4.5). The feature selection is performed on the training set. Specifically, the `WrapperSubsetEval` selected the best features (metrics) in a fold. Next, only the metrics that were nominated were adopted in the model construction. This selection and model construction process was iterated for each of the ten folds. Table 5 shows the selected metrics for each metrics suite in all subjects.

Since 10-fold cross validation randomly samples instances and puts them in ten folds [2], we repeated this process 100 times for each prediction model to avoid sampling bias [56]. Note that we use Weka's SMOTE filter to increase the instances of the minority class. The filter is applied only to the training folds of the cross validation instead of the whole dataset in advance. This is because the latter approach is likely to provide over optimistic results [82].

4.5 Performance Metrics

We chose performance metrics allowing us to answer each of our three research questions. Specifically, we employ precision, recall, F1-measure, area under the curve (AUC), and cost-effectiveness metric. An instance can be classified as: buggy when it is truly buggy (true positive, TP); it can be classified as buggy when it is actually not (false positive, FP); it can be classified as non-buggy when it is actually buggy (false negative, FN); or it can be correctly classified as non-buggy (true negative, TN).

Precision, Recall, and F1-measure.

- **Precision:** the number of instances correctly classified as buggy over the number of all instances classified as buggy.

$$P = \frac{TP}{TP+FP}$$

- **Recall:** the number of instances correctly classified as buggy over the total number of buggy instances.

$$R = \frac{TP}{TP+FN}$$

- **F-measure:** a composite measure of precision and recall for buggy instances.

$$F(b) = \frac{2*P*R}{P+R}$$

AUC (area under the curve). We use the AUC of the receiver operating characteristics (ROC) [57] as an additional measure to evaluate the performance of the prediction models. The range of AUC is [0, 1]. A larger AUC score indicates better prediction performance. A prediction model achieving AUC above 0.5 is considered more effective than the random predictor and 0.7 is reasonably good [38]. As a scalar value, AUC is well suited to compare the performance of different classifiers, and is often used for that purpose [66].

Cost Effectiveness. The cost effectiveness metric, which evaluates prediction performance given a cost limit, has been used widely in existing defect prediction techniques [47], [57], [94], [95]. In our context, the cost is the amount of function instances to inspect, and the benefit is the number of bugs that can be discovered. If developers inspect all predicted buggy instances, the percentage of bugs that can be detected is equivalent to the recall. In some circumstances (e.g., meeting a deadline), developers can only inspect certain amount of functions. Therefore, it is useful to maximize the bugs to be detected while minimizing the number of instances to inspect. In this case, the cost effectiveness metric is appropriate. We use the cost effectiveness metric, PofB20, used by Jiang et al. [42]. They measure the percentage of bugs that a developer can identify by inspecting the top 20 percent of lines of code. In our context, we measure the percentage of concurrency faults that a developer can identify by inspecting the top 20 percent of function instances.

To compute PofB20, we sort instances by their probability (provided by WEKA) of buggy [80] We then simulate a developer that inspects these potentially buggy instances one at a time. As the instances are inspected one at a time, we count the number of lines of code that have been inspected and the number of bugs that have been identified. We stop the inspection process when 20 percent of the lines of code have been inspected and compute the percentage of bugs that are identified. This percentage is the PofB20 cost effectiveness score. A higher cost effectiveness score represents that a developer can detect more bugs when inspecting a limited number of lines of code.

PofB20 metric uses 20 percent of all effort as the cut-off value. However, a different cut-off value might lead to different results. As an additional metric, we use P_{opt} [66] to evaluate the prediction performance of models. P_{opt} is defined as the area Δ_{opt} between the optimal model and the prediction model. In the optimal model, all instances are ordered by decreasing fault density, and in the predicted model, all instances are ordered by decreasing predicted value (i.e., probability of being buggy). The equation of computing P_{opt} is shown below, where a larger P_{opt} value means a smaller difference between the optimal and predicted model:

$$P_{opt} = 1 - \Delta_{opt}$$

The range of P_{opt} is [0, 1] and any predictor achieving the P_{opt} above 0.5 is more effective than the random predictor.

Statistical significance analysis. To assess whether prediction performance of different metric sets were statistically significant, we applied the Wilcoxon test [90] to the data sets, comparing each pair of metric sets within each model. We did not use t-test because F-measure outcomes from cross validation did not follow a Normal distribution. We checked if the mean of F-measure values of one predictor P_i was greater than the mean of F-measures of another predictor P_j at the 95% confidence level ($p - value < 0.05$).

Specifically, the null and alternative hypotheses for the t-test are:

- **H0:** F-measure mean of P_i is equal to the F-measure mean of P_j.
- **H1:** F-measure mean of P_i is greater to the F-measure mean of P_j. (i.e., P_i has better performance if the mean value is higher).

We rejected the null hypothesis H0 and accepted the alternative hypothesis H1 if the p-value was smaller than 0.05 (at the 95% confidence level).

Effect size. The effect size uses Cliff's delta [17] that quantifies the amount of difference between two non-parametric variables beyond the p-value interpretation. The Cliff's delta is computed by $d = 2W/mn - 1$, where W is the statistic of the Wilcoxon rank-sum test, and m and n are the sizes of two compared distributions. Here $W = R - n(n+1)/2$, where R is the sum of the rank in the sample and n is the sample size. The magnitude of effect size is usually assessed using the thresholds [83], where $|d| < 0.147$ is negligible, $0.147 \geq |d| < 0.33$ is small, $0.33 \geq |d| < 0.474$ is medium, and otherwise large. For example, suppose the effect size between two metric suites A and B is -0.75. The sign is negative because the mean of A is greater than the mean of B and the magnitude of the effect size is regarded as large.

4.6 Importance of Features

To study the most influential metrics, we compute Breiman's variable importance score [12] for each feature. The larger the score, the greater the influence of the metric on our models. We use the option `-attribute-importance` provided by Weka to compute the variable importance scores. For each run of the 10-fold cross validation we obtain an importance score for each feature. In order to determine the features that are most influential for the whole dataset, we apply the Scott-Knott test [39] on the values from all 10 runs. The Scott-Knott test will cluster the metrics according to statistically significant differences in their mean variable importance scores (p - value = 0.05). We use the implementation of the Scott-Knott test provided by the ScottKnott R package. The Scott-Knott test ranks each metric exactly once, however several metrics may appear within one rank.

Next, to assess how each factor is related to buggy instances, we compare the values of each feature between buggy instances and non-buggy instances. We first analyze the statistical significance of the difference between the two classes (buggy and non-buggy) by applying the Mann-Whitney U test at p - value = 0.01.

To show the effect size of the difference between the two features in two groups, we calculate Cliff's Delta.

4.7 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our programs, mutants, coverage criteria, and test cases. Other systems may exhibit different behaviors, as may other forms of test cases. However, the programs we investigated are popular open source programs. Furthermore, the test cases are those provided with the programs: they are representative of test cases that could be used in practice to test these programs. Most of the test subjects we used had relatively good test suites (i.e., of the covered mutants, the mutation scores were above 80%). Mutants can be influenced by external factors such as mutation operators. We used only concurrency mutation operators. However, concurrency faults can also be introduced by sequential glitches. In addition, other interleaving criteria (e.g., synchronization coverage) may lead to different coverage results. We controlled for these threats by using well studied concurrency mutation operators and popular interleaving criteria.

The data collected for defect prediction may contain noise, such as false positives (identifying non-buggy changes/files as buggy), random sampling, and imbalanced data [8], [50]. For example, Bird et al. [8] discovered that data collected via automated mining software repositories (MSR) often contain noise. To mitigate this threat, we manually and carefully selected high quality datasets. The bugs we selected are confirmed and fixed in the subsequent versions. While it is possible that some functions contain concurrency bugs but were mislabeled as clean, we selected program versions between 2000 and 2014, so the functions labeled as clean are unlikely to contain concurrency bugs because no such bugs have been reported since 2014. We used SMOTE to handle the class imbalance problem.

The primary threats to internal validity for this study are possible faults in the implementation of our approach and in the tools that we used to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against smaller programs for which we could manually determine the correct results. We also chose to use popular and established tools (e.g., LLVM, Weka) for implementing the various stages of our approach.

Where construct validity is concerned, our measurements involve using metrics extracted from source code and mutation analysis to predict defects in concurrent programs. Other static metrics and dynamic metrics (e.g., test suite metrics) are also of interest. Furthermore, other machine learning performance measures can be used to measure effectiveness and accuracy. To control for this threat, we chose commonly used F-measures. Other metrics, such as the Matthews Correlation Coefficient [85] (MCC), can be used to handle unbalanced data.

Conclusion validity concerns the statistical significance of the result. We applied 10-fold cross validation, and did so 100 times, as is common in experiments of this type. We also undertook statistical analysis to test our hypotheses. To further reduce threats to conclusion validity, we were careful to check the assumptions of the statistical tests that were used.

5 Results and Analysis

In this section, we present results related to the three research questions [2].

5.1 RQ1: Effectiveness of ConPredictor vs. SPM vs. SCM

To examine RQ1, we compare the performance of ConPredictor to that of SPM and SCM. Columns 3-5 and 7-9 of Table 6 show the performance of each set of metrics in different subjects in terms of precision, recall, F-measure, AUC, PofB20, and P_{opt} from 100 times ten-fold cross validations. The numbers marked in boldface indicate that the performance of ConPredictor is significantly different from that of SPM and SCM. Columns 6 reports the F-measure values of the techniques without applying feature selection.

Although the performance values varied, there was a clear trend in which ConPredictor outperformed SPM and SCM for every subject. For example, comparing to SPM, the improvement of F-measure ranged from 29.7% to 34%. Comparing to SCM, the improvement of F-measure ranged from 37.7% to 50%. The most improvement occurred on Apache while the least improvement was seen with OpenOffice. In other words, metrics considering concurrency characteristics improved the prediction performance.

Table 7 shows the effect size when comparing different metric suites in terms of their F-measure values. The numbers marked as bold indicate that the Wilcoxon rank-sum test rejected the null hypothesis of RQ1 (p-value < 0.05). For example, on Apache, the effect size between ConPredictor and SPM is -0.675 and they are *statistically* significantly different. Comparing ConPredictor to SPM, the largest effect size occurred with OpenOffice and smallest effect size occurred for Mozilla.

Among all 558 buggy instances, 368 and 270 are predicted as buggy by ConPredictor and SPM, respectively. All 270 faults predicted by SPM are also predicted by ConPredictor. SPM has a lower precision because the missing buggy instances typically have lower sequential metric scores but higher concurrency metric scores. For example, a function that does not contain any branches (CyclomaticComplexity is 2) but has several unprotected shared variables involves an atomicity violation bug but is falsely classified as non-buggy by SPM.

In all four subjects, the AUC values of ConPredictor are above 0.7 and significantly better than SPM and SCM. This result confirms the impact of our proposed concurrency metrics on concurrency fault prediction.

PofB20 represents the number of bugs that can be discovered by examining the top 20% LOC. For example, ConPredictor can help the developers identify 24 bugs for Apache by inspecting 20% LOC. SCM can help identify 12 bugs by inspecting 20% LOC, which is 12 less bugs than those of ConPredictor. Overall, ConPredictor improved PofB20 over SPM by amounts raninge from 5.8% to 23.4% and over SCM by amounts ranging from 30.8% to 53.8%.

Overall, these results suggest that *ConPredictor outperforms the traditional sequential metrics*.

5.1.1 RQ1.1: Feature Selection

As Column 6 of Table 6 shows, feature selection improves performance of the defect prediction in terms of F-measure in all

2. All data we used in our experiments are publicly available at http://cs.uky.edu/~tyu/research/ConPredictor

TABLE 6: Results of evaluated metrics

Project	Technique	P	R	F1	$F1_{nf}$	AUC	PofB20	P_{opt}
Apache	ConPredictor	**0.68**	**0.64**	**0.66**	0.62↓	**0.72**	**0.52**	**0.67**
	CStaMc	0.65	0.51	0.57	0.51↓	0.68	0.51	0.61
	CDyMc	0.64	0.56	0.60	0.55↓	0.52	0.54	0.50
	SPM	0.52	0.41	0.46	0.42↓	0.48	0.49	0.44
	SStaMc	0.43	0.31	0.33	0.31	0.44	0.38	0.42
	SDyMc	0.46	0.32	0.35	0.29↓	0.47	0.34	0.44
	SCM	0.43	0.34	0.33	0.31	0.45	0.36	0.41
MySQL	ConPredictor	**0.65**	**0.63**	**0.64**	0.55↓	**0.74**	**0.42**	**0.69**
	CStaMc	0.59	0.57	0.58	0.52↓	0.64	0.40	0.58
	CDyMc	0.61	0.63	0.62	0.56↓	0.66	0.42	0.60
	SPM	0.52	0.40	0.45	0.41	0.51	0.35	0.51
	SStaMc	0.35	0.38	0.41	0.35↓	0.48	0.31	0.50
	SDyMc	0.38	0.33	0.39	0.32↓	0.48	0.28	0.45
	SCM	0.32	0.32	0.36	0.32↓	0.47	0.25	0.42
Mozilla	ConPredictor	**0.60**	**0.54**	**0.57**	0.52↓	**0.75**	**0.47**	**0.70**
	CStaMc	0.55	0.48	0.51	0.45↓	0.55	0.39	0.52
	CDyMc	0.57	0.48	0.52	0.47↓	0.58	0.42	0.55
	SPM	0.54	0.29	0.38	0.35	0.44	0.36	0.54
	SStaMc	0.41	0.34	0.37	0.35	0.40	0.36	0.38
	SDyMc	0.42	0.30	0.35	0.32	0.44	0.34	0.40
	SCM	0.41	0.31	0.35	0.33↓	0.42	0.32	0.42
OpenOffice	ConPredictor	**0.59**	**0.48**	**0.53**	0.50↓	**0.77**	**0.52**	**0.68**
	CStaMc	0.57	0.41	0.48	0.46↓	0.56	0.50	0.52
	CDyMc	0.60	0.47	0.53	0.51↓	0.58	0.51	0.53
	SPM	0.48	0.28	0.35	0.33↓	0.46	0.40	0.42
	SStaMc	0.35	0.25	0.29	0.28	0.44	0.36	0.42
	SDyMc	0.39	0.27	0.32	0.29↓	0.46	0.24	0.41
	SCM	0.37	0.30	0.33	0.31	0.42	0.24	0.39

TABLE 7: Effect Sizes for each metric set in different subjects.

Prog.	CONFPREDICTOR						CSTAMC					
	CStaMc	CDyMc	SPM	SStaMc	SDyMc	SCM	CStaMc	CDyMc	SPM	SStaMc	SDyMc	SCM
Apache	**-0.223**	**-0.142**	**-0.675**	**-0.734**	**-0.539**	**-0.777**	-	0.133	**-0.424**	**-0.498**	**-0.432**	**-0.501**
MySQL	**-0.374**	**-0.258**	**-0.842**	**-0.577**	**-0.699**	**-0.850**	-	**0.241**	**-0.692**	**-0.701**	**-0.522**	**-0.725**
Mozilla	**-0.331**	**-0.251**	**-0.603**	**-0.658**	**-0.694**	**-0.709**	-	0.231	**-0.498**	**-0.572**	**-0.451**	**-0.533**
OpenOffice	**-0.402**	**-0.333**	**-0.862**	**-0.801**	**-0.792**	**-0.877**	-	**0.312**	**-0.398**	**-0.402**	**-0.257**	**-0.429**

Prog.	CDYMC					
	CStaMc	CDyMc	SPM	SStaMc	SDyMc	SCM
Apache	-	-	**-0.563**	**-0.852**	**-0.771**	**-0.642**
MySQL	-	-	**-0.701**	**-0.721**	**-0.814**	**-0.759**
Mozilla	-	-	**-0.423**	**-0.499**	**-0.367**	**-0.598**
OpenOffice	-	-	**-0.598**	**-0.744**	**-0.725**	**-0.415**

TABLE 8: Performance comparison of prediction models by different machine learners.

Learner	B.N	D.T.	L.R.	R.F.
ConPredictor	0.57	0.61	0.53	0.65
CStaMc	0.48	0.47	0.47	0.57
CDyMc	0.53	0.55	0.49	0.62
SPM	0.39	0.40	0.37	0.45
SStaMc	0.34	0.33	0.32	0.38
SDyMc	0.37	0.36	0.34	0.40
SCM	0.32	0. 31	0.32	0.35

28 metric sets across all four subjects. The ↓ symbol indicates that the improvement is statistically significant. The significant improvement occurs to 20 out of 28 metric sets across all four subjects by amounts ranging from 5.7% to 25.7%. The lowest level of improvement occurred on SPM for OpenOffice, while the highest levels of improvement occurred on ConPredictor for MySQL. Overall, these results suggest that *the use of feature selection improves the performance of the concurrency fault prediction*.

5.1.2 RQ1.2: Effectiveness of Static Metrics

As Table 6 shows, comparing the static concurrency metric set (CStaMC) to the static sequential metric set (SStaMC), when averaging the F-measures, CStaMC improved over SStaMC ranging from 23.8% to 64.2% on all four subjects. The best improvement is for Apache and the worst improvement is for OpenOffice. Table 7 (columns under the SStaMC) shows the improvements were statistically significant. The results indicate that *when choosing static metrics for predicting concurrency faults, CStaMC is better than SStaMC*.

When comparing CStaMC to ConPredictor, ConPredictor was more effective for all four subjects. Their F-measures are statistically significant. In other words, adding dynamic metrics to CStaMC improved the prediction performance of static metrics. The effectiveness improvement achieved by ConPredictor with

TABLE 9: Effect sizes for each machine learner in different subjects.

Learner	ConfPredictor			CStaMc			CDyMc		
	B.N.	D.T.	L.R.	B.N.	D.T.	L.R.	B.N.	D.T.	L.R.
B.N.	-	0.122	**-0.358**	-	-0.025	-0.152	0.231	-0.101	-0. 132
D.T.	**-0.544**	-	**0.205**	-0.332	-	0.108	0.098	-	0.133
L.R.	**0.198**	0.330	-	0.182	**0.258**	-	**0.209**	**0.298**	-
R.F.	**-0.499**	0.223	**-0.532**	**-0.392**	**-0.254**	**-0.338**	**-0.292**	**-1.163**	**-0.083**

TABLE 10: Scott-Knott test results.

Group	Metric	Rank_{mean}	$\text{Rank}_{highest}$	Rank_{lowest}
G1	CCC	1	1	1
	$\text{MuDuE}_{spltecs}$	1.6	1	2
	MuDuS_{rmlock}	1.8	1	3
G2	MuDuK_{rmlock}	3.1	2	5
	CSV	4.9	3	6
G3	MuDuK_{swptw}	5.8	4	7
	MuDuE_{mwait}	6.2	5	7
	MuDuE_{swptw}	7.5	6	9
G4	CCE	9.1	8	10
	MuDuK_{rmsig}	10.6	9	11
G5	CCD	10.9	10	12
G6	CSO	11.5	10	13
G7	MuDuE_{rmsig}	12.2	11	13

TABLE 11: Effects of different metrics

Metric set	Metric	Rel.	d-value
CStaMc	CCC	+	**0.341 (medium)**
	CSV	+	0.225 (small)
	MuDuS_{rmlock}	+	0.204 (small)
CDyMc	MuDuE_{rmlock}	+	**0.325 (medium)**
	$\text{MuDuE}_{spltecs}$	+	0.212 (small)
	MuDuk_{rmlock}	+	**0.308 (medium)**
	MuDuK_{swptw}	+	0.187 (small)

Rel.= relationship. "+" indicates faulty instances have significantly higher value on this metric.

respect to CStaMC ranged from 10.8% (MySQL) to 16.2% (Apache). These results indicate that *the combination of static and dynamic metric sets is more effective than the static metric set alone*. The results also implies that *the use of the dynamic metric sets amplifies the effectiveness of the defect prediction*.

When comparing CStaMC to CDyMC, CDyMC performs better than CStaMC, ranging from 2% (Mozilla) to 7.2% (OpenOffice). However, such differences were statistically significant on only MySQL and OpenOffice. The results imply that *dynamic metrics have the potential to improve the performance of prediction over static metrics on certain programs*.

5.1.3 RQ1.3: Effectiveness of Dynamic Metrics

As shown in Table 6 and Table 7 (columns under CDyMC), ConPredictor was more effective than CDyMC on all four subjects. The improvements, ranging from 3.9% (MySQL) to 9.9% (Apache), were statistically significant. These results indicate that *the combination of static metrics and dynamic metrics has better performance than the dynamic metrics alone,* and *the use of the static metrics validates the prediction performance of ConPredictor*.

When comparing CDyMC and SDyMC, CDyMC could consistently improve the performance of SDyMC on all subjects ranging from 14.3% (Mozilla) to 66.7% (Apache). The improvements were still statistically significant. These results imply that *when choosing dynamic metrics for defect prediction on concurrent programs, CDyMC is better than SStaMC*.

5.1.4 RQ1.4: Different Machine Learners

Table 8 shows the performance of each set of metrics in different machine learners in terms of the F-measure distributions from 100 times 10-fold cross validations. As the table shows, the F-measure distributions from different machine learners did vary. However, the trend indicates that ConPredictor performs better than the other metric sets over all learners. Table 9 shows the comparisons among different learners across ConPredictor and the two metric sets used in ConPredictor. The numbers are the effect sizes and those marked as bold indicate that the differences are statistically significant different.

Among the mean values, Random Forest was significantly better than two other learners over all the metric sets. Only on ConPredictor was Random Forest not significantly different from Decision Tree. Therefore, we employed Random Forest in the other experiments mentioned in Section 4.4. In contrast, Logistic Regression was the worst choice for model construction in our experiment.

If these results generalize to other real subjects, we then conclude that *when constructing machine learning models to predict concurrency faults, Random Forest is the best choice, whereas Logistic Regression is inferior.*

5.2 RQ2: Metrics Effectiveness Analysis

Table 10 shows the Scott-Knott test results. The importance values of metrics in one group are statistically significantly different from those of metrics in other groups. The results show that CCC (concurrency code complexity), $\text{MuDuE}_{spltecs}$, and MuDuS_{rmlock} are the top three most important features that influence our random forest model. The best predictor was CCC. For code quality prediction, this metric resembles McCabe complexity for sequential programs. A previous study [69] has shown that McCabe is effective at predicting defects in sequential programs. Moreover, lock operations are commonly used in concurrent programming to synchronize shared resource accesses. These results suggest subjects in our dataset that have higher concurrency code complexity, with more locks are more likely to discriminate faulty instances from non-faulty instances.

We next use Wilcoxon rank-sum and Cliff's delta to evaluate the effects of feature importance. Table 11 shows the factors that have p - value < 0.01 and d > 0.147 (i.e., statistically significant difference with at least a small effect size). We find that the faulty and non-faulty instances have statistically significant differences with at least a small effect size in 7 out of the 17 selected features. The effect sizes are small for most of the 17 factors, except for CCC, $\text{MuDuE}_{spltecs}$, and MuDuS_{rmlock}. The results are consistent with feature ranking in Table 10.

TABLE 12: F-measures on across project prediction using ConPredictor and SPM.

Prog.	CONFPREDICTOR				SPM			
	Apache	MySQL	Mozilla	OpenOffice	Apache	MySQL	Mozilla	OpenOffice
Apache	-	**0.546**	**0.566**	**0.496**	-	0.421	0.443	0.402
MySQL	0.311	-	**0.399**	**0.475**	0.294	-	0.315	0.358
Mozilla	**0.502**	**0.387**	-	0.375	0.394	0.302	-	0.331
OpenOffice	**0.451**	0.325	**0.364**	-	0.425	0.308	0.341	-

For the static metric set, three out of six metrics can differentiate faulty instances from non-faulty instances. Typically, when a program has more complex code, shared variables, and locks, it becomes more difficult to maintain the program and thus is more likely to cause field failures. For the dynamic metric set, four out of seven metrics can differentiate faulty instances from non-faulty instances. These metrics are all related to lock operations, which again suggest that faulty instances are more likely to misuse locks.

5.3 RQ3: Across Projects Prediction

RQ3 investigates whether a predictor for one application group (dataset) can be used for other applications. We applied classification model built by ConPredictor from each dataset to the instances of each of the other four datasets. We then checked how accurate the prediction is by assessing the performance of each model. The prediction results are shown in columns 2-5 of Table 12. For example, columns 3-5 show the performance values of using models learned from MYSQL, MOZZILA and OPENOFFICE to predict APACHE.

As the results show, the F-measure values for three out of 12 models were greater than 0.5, and six out of 12 models were greater than 0.4. Two out three models that are greater than 0.5 happen between MOZZILA and APACHE. The two projects have certain similarities because they involve web applications (i.e., in the case when Mozilla is used to predict Apache and in the case when Apache is used to predict Mozilla).

We next compare the across project prediction performance of ConPredictor to that of SPM. Columns 6-9 of Table 12 show the F-measures using SPM. The results indicate that ConPredictor is more effective than SPM on all 12 program pairs. The numbers marked as bold indicate that the differences are statistically significant.

Overall, these results suggest that *ConPredictor is moderately effective at predicting concurrency faults across projects, and is more effective when predicting across similar projects. When comparing ConPredictor to SPM, ConPredictor is more effective at predicting concurrency faults across projects*. For future work, we intend to improve ConPredictor on cross-project defect prediction by applying Transfer Component Analysis+ (TCA+) analysis [74].

6 SUMMARY AND DISCUSSION

As presented in the previous section, we were able to demonstrate that ConPredictor is useful for predicting defects of concurrent programs. Specifically, we showed (subject to stated threats to validity) that 1) the metrics used in ConPredictor are more effective at predicting concurrency faults than sequential metrics, 2) the combination of static and dynamic metrics has better performance than either the static metric set or the dynamic metric set, 3) ConPredictor's dynamic concurrency metrics are more effective contributors than static metrics based on the information ratios, 4) the concurrency code complexity metric (CCC) is the most effective code metric, and 5) the models built on three datasets can be used to predict concurrency faults for the fourth dataset with good effectiveness.

One of our primary findings is that mutation metrics can significantly improve predictive performance and with large effect sizes. This is the first time any kind of concurrency mutation has been used to support fault prediction. Naturally, subsequent studies can further investigate/exploit this predictive improvement, perhaps in combination with other sets of metrics (e.g., process metrics). In this first study we present the empirical evidence that concurrency mutation metrics' effect size on prediction outcomes can be large, thereby motivating and opening up this avenue of research.

Our results have implications for practitioners and researchers, discussed below.

6.1 Implications for Practitioners

Results indicate that concurrency-related program metrics can be effective and are more effective than predictors learned from sequential program attributes. Practitioners can apply this finding by building a CCFG, obtaining the concurrency metrics (we plan to provide a tool in the future to simplify this), and substituting their metrics into our learned model in order to predict defect-prone functions. In addition, our results showed that several static code metrics (e.g., CCC, CSV) were good predictors. For example, industry practitioners can use the CCFG to calculate CCC and examine its distribution for their project. Functions that have higher values of CCC should be examined and possibly subjected to additional code review and/or unit testing to lower the risk of concurrency faults. Also, functions that share communication edges with defect-prone functions warrant additional attention during software assurance activities.

6.2 Implications for Researchers

The use of CCFG. Our study shows that code and mutation metrics can be used to predict faults specific to concurrent programs. Researchers should consider adding the CCFG to their arsenal of program representations. The ConPredictor metrics may have other applications, such as for predicting testability of concurrent programs, predicting change prone components in concurrent programs, and predicting the number of tests that are needed to achieve coverage of concurrent programs.

Lowering the recall. We inspected a few cases where an instance (i.e., function) is mislabeled as non-faulty. This is because a concurrency fault may involve more than one function because the conflicting shared variables may exist in different functions. A function labeled as non-faulty may have conflicting accesses with other functions that are labeled as faulty. To address this problem, we could employ the following heuristic: if one function f is labeled as faulty, other functions that can be reached through a communication edge from f should also be flagged as *possible*

faulty. Identification and test of functions that can reach defect-prone functions may hold promise for improving the recall of concurrency fault prediction and could be the focus of future work.

Incorporating test suite metrics. Our results suggest that dynamic metrics are more important than static metrics in concurrency fault detection. This implies that the quality of the test suite is also a considerable factor in building effective predictors. In this work, the dynamic metric set includes only mutation metrics. More dynamic metrics are worth being studied to improve the performance of ConPredictor. One direction is to develop test suite metrics by which their executions provide various observable attributes. For example, coverage metrics are commonly used as they directly measure the relationship between the test suite and source code. Thus, predictors can be built by incorporating the coverage metrics of the program under test.

Additionally, the finding on communication edges implies that researchers should examine concurrent edge coverage more carefully as an important test criterion. The *interleaving coverage criteria* has been widely used to measure test suite quality for concurrent programs [13], [37], [60]. An interleaving criterion is a pattern of inter-thread dependencies through SV accesses that helps select representative interleavings to effectively expose concurrency faults. An interleaving criterion is satisfied if all feasible interleavings of SV defined in the criteria are covered. As part of the future work, we can employ a Def-Use criterion, which is satisfied if and only if a write `w` in one thread happens before a read `r` in another thread and there is no other write to the variable read by `r` between them. In fact, the Def-Use criterion is equivalent to communication edge coverage in the context of CCFG.

7 Related Work

There has been much research on developing various software metrics and prediction algorithms to assess software quality [7], [14], [23], [49], [56], [57], [58], [68], [69], [70]. For example, Lee et al. [56] proposed a set of micro-interaction metrics (MIMs) that leverage developers' interaction information combined with source code metrics to predict defects. Meneely [67] et al. examine structure of developer collaboration and use developer network derived from code information to predict defects. Basili et al. [7] used Chidamber and Kemerer metrics, and Ohlsson et al. [77] used McCabe's cyclomatic complexity for defect prediction. Koru and Liu utilized static software measures along with defect data at the class level to predict bugs using machine learning. Menzies et al. [68] conclude that static code metrics are useful in predicting defects under specific learning methods. These techniques, however, focus on sequential programs while ignoring code attributes and testability for concurrent programs.

Besides code metrics, other metrics can be obtained from different software artifacts. For example, Ohlsson et al. [77] study metrics derived from design documents that are used to predict fault-prone modules. Metrics collecting from version control systems (e.g., number of commits) have also been used to predict faults [36], [51], [71]. Nagappan et al. [72] build fault prediction models by considering the organizational metrics. However, none of these techniques consider metrics or faults in concurrent programs. In addition, they do not consider dynamic metrics related to test executions.

The only related work found is the work by Mathews et al. [64] based on Ada programs. This work considers only the number of synchronizations and conditional branches that contain synchronization points without utilizing them to perform defect prediction. However, this work does not aim to predict concurrency faults. In addition, this work does not consider dynamic metrics.

There have been many approaches to improving performance of cross-project defect prediction [62], [73], [74], [81], [88], [89], [100]. For example, Nam et al. [74] adapted a state-of-the-art transfer learning technique called Transfer Component Analysis (TCA) and proposed TCA+. Turhan et al. proposed the nearest-neighbor (NN) filter to improve the performance of cross-project defect prediction [88]. The basic idea of the NN filter is that prediction models are built by source instances that are nearest-neighbors of target instances. In the future, we intend to leverage these techniques to improve the performance of ConPredictor in cross-project concurrency fault prediction.

Various data quality issues can arise when constructing defect prediction datasets. a variety of methods have been proposed for dealing with class imbalance problems. Chawla et al. [15] proposed an over-sampling approach in which the minority class is over-sampled by creating "synthetic" instances rather than by over-sampling with replacement. Estabrooks et al. [22] and Barandela et al. [6] both suggested that a combination of over-sampling and under-sampling might be more effective to solve the class imbalance problem. ConPredictor employs an over-sampling technique [87], i.e., SMOTE.

Studies have also shown that the presence of systematic data noise and bias in several open source data sets affect the performance of defect prediction models [5], [8], [75]. Therefore, research on improving mappings between bug reports and code has been proposed. For example, Bird et al. [9] develop a technique to manually annotate bug reports and code changes to reduce the overhead of manual data point inspection. Wu et al. [93] propose an automatic link recovery algorithm to recover missing links between bug reports and code changes. Kim et al. [50] propose guidelines about acceptable noise levels and propose a noise detection and elimination algorithm. In the future, we will leverage the above techniques to speed up the construction of datasets.

There has been a great deal of research on mutation testing for sequential programs [19], [63], [84]. Jia and Harman [41] provide a recent survey. In this work, we focus on techniques that share similarities with ours. There has also been some work on mutation testing for concurrent programs [11], [29], [46], which has been discussed in Section 2. Other tools such as MuTMuT [28] have been used to optimize the execution of mutants by reducing interleaving space that has to be explored. None of the techniques, however, attempt to predict defects using software metrics.

Recent work by Bowes et al. [10] propose a mutation-aware fault prediction technique, which leverages guidance from mutation analysis to construct dynamic metrics. Both their work and ConPredictor use mutants and the test cases that cover and detect them for building additional metrics. However, their technique focuses on sequential programs. As shown in our results, either static or dynamic sequential metrics are significantly less effective than ConPredictor's metrics.

8 Conclusions

This paper presents an approach to predict defects of concurrent programs at the function level. We proposed six novel static code metrics, six static mutation metrics, 12 dynamic mutation metrics,

and combined them with a dynamic test suite metric to learn four prediction models. We applied the models to four large-scale real-world programs. We found that ConPredictor can outperform traditional defect prediction using sequential metrics. In all cases, both dynamic and static metrics feature prominently in the top 10 most influential metrics across all subjects, providing consistent evidence that they are beneficial to the performance of prediction models. In addition, ConPredictor showed good effectiveness when applied to different software projects. Our study extended existing knowledge in the field of software quality metrics by proposing novel metrics specific to concurrent programs. In the future, we will consider other code metrics and test suite attributes and investigate their effectiveness as discussed in Section 6.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1464032, CCF-1652149, and CCF-1511117.

REFERENCES

[1] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of mutant operators for the c programming language. Technical report, Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[2] E. Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2004.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering*, pages 402–411, 2005.

[5] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106, 2010.

[6] R. Barandela, R. Valdovinos, J. Sanchez, and F. Ferri. The imbalanced training sample problem: Under or over sampling? *Structural, syntactic, and statistical pattern recognition*, pages 806–814, 2004.

[7] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761.

[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 *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130, 2009.

[9] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein. Linkster: enabling efficient manual inspection and annotation of mined data. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 369–370, 2010.

[10] D. Bowes, T. Hall, M. Harman, Y. Jia, F. Sarro, and F. Wu. Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 330–341, 2016.

[11] J. Bradbury, J. Cordy, and J. Dingel. Mutation operators for concurrent java (j2se 5.0). In *International Workshop on Mutation Analysis*, pages 11–11, 2006.

[12] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[13] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–212, 2005.

[14] V. U. B. Challagulla, F. B. Bastani, I.-L. Yen, and R. A. Paul. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools*, 17(02):389–400.

[15] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[16] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[17] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494, 1993.

[18] D. R. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 215–242, 1958.

[19] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247.

[20] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practical programmer. 11 (4): 34–41, 1978.

[21] D. Deng, W. Zhang, and S. Lu. Efficient concurrency-bug detection across inputs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 785–802, 2013.

[22] A. Estabrooks, T. Jo, and N. Japkowicz. A multiple resampling method for learning from imbalanced data sets. *Computational intelligence*, 20(1):18–36, 2004.

[23] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. CRC Press, 1998.

[24] M. K. Ganai and C. Wang. Interval analysis for concurrent trace programs using transaction sequence graphs. In *Proceedings of the International Conference on Runtime Verification*, pages 253–269, 2010.

[25] S. Ghosh. Towards measurement of testability of concurrent object-oriented programs using fault insertion: A preliminary investigation. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, pages 17–, 2002.

[26] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the International Conference on Software Engineering - Volume 1*, pages 789–800, 2015.

[27] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 171–180, 2012.

[28] M. Gligoric, V. Jagannath, and D. Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *International Conference on Software Testing, Verification and Validation*, pages 55–64, 2010.

[29] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 224–234, 2013.

[30] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *Special Interest Group on Knowledge Discovery and Data Mining Explorations Newsletter*, 11(1):10–18, Nov. 2009.

[31] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[32] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[33] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*, pages 212–222, 2011.

[34] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo. Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 397–408, 2014.

[35] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering*, 27(3):248–263, March 2001.

[36] K. Herzig, S. Just, A. Rau, and A. Zeller. Predicting defects using change genealogies. In *IEEE 24th International Symposium on Software Reliability Engineering*, pages 118–127, 2013.

[37] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 210–220, 2012.

[38] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[39] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman. Scottknott: a package for performing the scott-knott clustering algorithm in r. *TEMA (São Carlos)*, 15(1):3–17, 2014.

[40] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology Journal*, 51(10):1379–1393.

[41] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[42] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 279–289, 2013.

[43] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[44] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[45] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 13–22, 2009.

[46] L. W. G. Kaiser. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators. In *International Conference on Software Engineering and Knowledge Engineering*, pages 244–249, 2011.

[47] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[48] T. M. Khoshgoftaar, A. S. Pandya, and D. L. Lanning. Application of neural networks for predicting program faults. *Annals of Software Engineering*, 1(1):141–154, 1995.

[49] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196.

[50] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 481–490, 2011.

[51] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498, 2007.

[52] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[53] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1):273–324, 1997.

[54] M. Kusano and C. Wang. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 722–725, 2013.

[55] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization.*, pages 75–86, 2004.

[56] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*, pages 311–321, 2011.

[57] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[58] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson. Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. In *Proceedings of the 28th international conference on Software engineering*, pages 413–422, 2006.

[59] LLVM Pass. Writing an LLVM Pass. Web page. http://llvm.org/docs/WritingAnLLVMPass.html.

[60] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *ACM SIGOPS Operating Systems Review*, pages 103–116, 2007.

[61] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. pages 329–339, 2008.

[62] Y. Ma, G. Luo, X. Zeng, and A. Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, 2012.

[63] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. Mujava: A mutation system for java. In *Proceedings of the International Conference on Software Engineering*, pages 827–830, 2006.

[64] M. E. Mathews and S. Tu. Metrics measuring control flow complexity in concurrent programs. In *In Proceedings of the 13th Pacific Northwest Software Quality Conference*, page 5, 1995.

[65] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

[66] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, page 7, 2009.

[67] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 13–23, 2008.

[68] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13.

[69] T. Menzies, O. Jalali, J. Hihn, D. Baker, and K. Lum. Stable rankings for different effort models. *Automated Software Engineering*, 17(4):409–437, Dec. 2010.

[70] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ACM/IEEE International Conference on Software Engineering.*, pages 181–190, 2008.

[71] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190, 2008.

[72] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality. In *ACM/IEEE 30th International Conference on Software Engineering*, pages 521–530, 2008.

[73] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 2017.

[74] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 382–391, 2013.

[75] T. H. Nguyen, B. Adams, and A. E. Hassan. A case study of bias in bug-fix datasets. In *17th Working Conference on Reverse Engineering*, pages 259–268, 2010.

[76] T. H. Nguyen, B. Adams, and A. E. Hassan. Studying the impact of dependency network measures on software quality. In *IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[77] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.

[78] J. R. Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[79] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[80] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441, 2013.

[81] F. Rahman, D. Posnett, and P. Devanbu. Recalling the imprecision of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 61, 2012.

[82] D. Rodriguez, I. Herraiz, R. Harrison, J. Dolado, and J. C. Riquelme. Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 43, 2014.

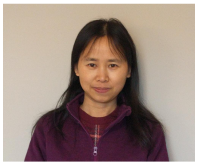
[83] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

[84] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 69–80, 2009.

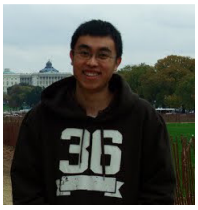
[85] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[86] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering*, 29(4):297–310, 2003.
[87] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering*, pages 99–108, 2015.
[88] B. Turhan. On the dataset shift problem in software engineering prediction models. *Empirical Software Engineering*, 17(1-2):62–74, 2012.
[89] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.
[90] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye. *Probability and statistics for engineers and scientists*, volume 5. Macmillan New York, 1993.
[91] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.
[92] W. E. Wong, editor. *Mutation Testing for the New Century*. Advances in Database Systems. Springer, 2001.
[93] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25, 2011.
[94] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering*, 42(10):977–998, 2016.
[95] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 157–168, 2016.
[96] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, pages 159–172, 2011.
[97] T. Yu, W. Srisa-an, and G. Rothermel. SimRacer: An automated framework to support testing for process-level races. In *ISSTA*, pages 167–177, 2013.
[98] T. Yu, W. Wen, X. Han, and J. H. Hayes. Predicting testability of concurrent programs. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 168–179, 2016.
[99] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *MSR*, pages 199–208, 2012.
[100] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 182–191, 2014.
[101] H. Zhang, X. Zhang, and M. Gu. Predicting defective software components from code complexity measures. In *Pacific Rim International Symposium on Dependable Computing*, pages 93–96, 2007.
[102] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *International Conference on Software Engineering*, pages 531–540, 2008.

Tingting Yu Tingting Yu is an Assistant Professor of Computer Science at University of Kentucky. She received her M.S. and Ph.D degree from University of Nebraska-Lincoln in 2014, and B.E. degree in Software Engineering from Sichuan University in 2008. Her research is in software engineering, with focus on developing methods and tools for improving reliability and security of complex software systems; testing for sequential and concurrent software; regression testing; and performance testing. She is a member of the IEEE Computer Society.



Wei Wen Ms. Wei Wen received her M.S. degrees in Electrical & Computer Engineering and Computer Science from University of Kentucky in 2009 and 2016. She is currently working as a Mobile Software Engineer developing apps for both Android and iOS platforms.



Xue Han Mr. Xue Han is in his 4th year of Ph.D. study in the Computer Science Department, University of Kentucky. He is passionate about doing research in Software Testing, Search Based Software Testing, Performance Modeling, Program Analysis, Machine Learning, Natural Language Processing, and Data Mining. He has published several research papers in premier Software Engineering conferences. Before his Ph.D. study, he was working as a .NET Engineer.

Jane Huffman Hayes Jane Huffman Hayes received the PhD degree in information technology from George Mason University. She is a professor in the Department of Computer Science at the University of Kentucky. Her research interests include requirements, software verification and validation, traceability, maintainability, and reliability. She is a member of the IEEE Computer Society.