Selective Mutation Testing for Concurrent Code

Milos Gligoric
University of Illinois
Urbana, IL 61801, USA
gliga@illinois.edu

Lingming Zhang
University of Texas
Austin, TX 78712, USA
zhanglm@utexas.edu

Cristiano Pereira and Gilles Pokam
Intel Corporation
Santa Clara, CA 95054, USA
{cristiano.l.pereira,gilles.a.pokam}@intel.com

ABSTRACT

Concurrent code is becoming increasingly important with the advent of multi-cores, but testing concurrent code is challenging. Researchers are developing new testing techniques and test suites for concurrent code, but evaluating these techniques and test suites often uses a small number of real or manually seeded bugs.

Mutation testing allows creating a large number of buggy programs to evaluate test suites. However, performing mutation testing is expensive even for sequential code, and the cost is higher for concurrent code where each test has to be executed for many (possibly all) thread schedules. The most widely used technique to speed up mutation testing is selective mutation, which reduces the number of mutants by applying only a subset of mutation operators such that test suites that kill all mutants generated by this subset also kill (almost) all mutants generated by all mutation operators. To date, selective mutation has been used only for sequential mutation operators.

This paper explores selective mutation for concurrent mutation operators. Our results identify several sets of concurrent mutation operators that can effectively reduce the number of mutants, show that operator-based selection is slightly better than random mutant selection, and show that sequential and concurrent mutation operators are independent, demonstrating the importance of studying concurrent mutation operators.

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

General Terms: Experimentation

Keywords: Selective mutation testing, concurrent code

1. INTRODUCTION

Concurrent code is becoming increasingly important with the wide-spread proliferation of multi-core processors in today's computers, ranging from mobile phones to tablets and laptops to desktops and data centers. While concurrent code unleashes the full potential of multi-core processors, developing concurrent code poses great challenges. Foremost, concurrent code is subject to unique concurrency bugs, including data races, atomicity violations, and deadlocks [31][1]. Finding these bugs warrants additional focus on developing and *evaluating* testing techniques for concurrent code.

Testing concurrent code is not only important but also *expensive*. A concurrent test executes two or more threads. Since the test result depends on the thread schedule, detecting concurrency bugs requires not just a single execution of the test for one schedule but *exploration* of the state-space reachable for multiple executions of various thread schedules. Researchers have proposed many techniques for state-space exploration, including random testing [9,15], unit testing [25], and systematic exploration [20, 38, 52]. However, these testing techniques and concurrent test suites have been mostly evaluated using programs with a *small number* of real or manually seeded concurrency bugs.

Mutation testing [1, 13, 22, 28] is an approach for evaluating test suites and testing techniques using a *large number* of systematically seeded program changes, which allows for statistical analysis of results [2]. Given a system under test, mutation testing first applies *mutation operators* to generate a set of mutants and then executes a test suite on each mutant to check if the mutant can be killed. The ratio of the number of killed mutants to the number of all (non-equivalent) *generated* mutants is called the *mutation score*.

Mutation testing has a high cost as the test suite is executed on all mutants. For sequential code, this cost is mostly due to the large number of mutants because executing one test on one mutant is relatively fast. For concurrent code, however, this cost is much higher because executing/exploring even one test on even one mutant is fairly expensive. Even a relatively small test (say, a few threads with a few dozens lines of code) can result in a large state space and large number of schedules (say, thousands or millions of states and schedules). Hence, running one concurrent test on one mutant takes time, and mutation testing requires running many tests on many mutants.

To reduce the cost of mutation testing, researchers have proposed [4, 37, 40, 41, 49, 55] two approaches for selecting a subset of the generated mutants: selective mutation (based on carefully selecting mutation operators) and random selection (based on randomly selecting generated mutants).

ISSTA '13, July 15-20, 2013, Lugano, Switzerland

[1]In this paper we focus on shared-memory concurrent code, the most widely used paradigm for concurrent programming, where threads exchange data and synchronize with one another through shared memory.

The previous studies used two metrics to compare different approaches: (1) cost in terms of the number of mutants, with *savings* defined as the ratio of the number of non-selected mutants to the total number of generated mutants, and (2) effectiveness in terms of *non-selective mutation score*, defined as the percentage of all generated non-equivalent mutants that are killed by the test suites that kill all *selected* non-equivalent mutants.

Selective mutation identifies a subset of operators that have high effectiveness and provide high savings. In other words, selective mutation generates fewer mutants such that test suites that have a high mutation score for these mutants also have a high mutation score for all mutants generated by all mutation operators. Selective mutation has been extensively studied for sequential mutation operators [4, 37, 40, 41, 49, 55]. However, selective mutation has not been studied for concurrent mutation operators, and researchers have no practical guidelines for choosing operators in mutation testing for concurrent code.

The major goal of this paper is to characterize the cost-effectiveness trade-offs that can be obtained for various subsets of concurrent mutation operators. To address this goal, we *repeat all previous studies* on selective mutation for sequential code [4,37,40,41,49,55] but perform them for concurrent code. In brief, these studies select the subset of operators based on the number of mutants they generate [37,41], on the categories of operators [4, 40], and on correlation of mutant killing [49]; additionally, a recent study [55] compares random selection and selective mutation. More details of these studies are presented in Section 2.2.

We additionally *perform three new evaluations* for concurrent code. First, we analyze cost-effectiveness exhaustively for all possible subsets of operators; the fact that a relatively small number (15) of concurrent operators generate mutants allows exhaustively trying all possible (2^{15}) subsets of operators to identify the subsets with the best cost-effectiveness trade-off. Second, we evaluate a new metric for comparing the cost of different selective approaches, measuring not only the number of mutants but also the exploration costs (i.e., time to execute schedules) for concurrent tests. Third, we evaluate whether sequential and concurrent mutation operators are independent or subsume one another.

Our study provides the following conclusions:

- There are important differences between concurrent and sequential mutation operators: selective mutation is applicable to concurrent code but provides lower savings than for sequential code, selecting operators based on the number of mutants or categories of operators does not provide good effectiveness, and random selection is not as good as selective mutation.
- While each previous study [4, 37, 40, 41, 49] provided *one set of sequential mutation operators* that researchers can use to speed up mutation for comparing test suites and testing techniques/tools, we provide *several sets of concurrent mutation operators* (Table 5) that researchers can use to trade-off the cost and effectiveness of mutation testing. One can choose savings for concurrent mutants as high as for sequential mutants but with reduced effectiveness, or one can choose high effectiveness but with low savings.
- We show evidence that the savings in terms of the number of mutants corresponds to the savings in terms of exploration cost for concurrent code.

Table 1: Mutation operators for concurrent code that have been previously proposed [6] and our three new operators: RTS, MTS, and WTIS

Category	Concurrency Mutation Operators
Modify Parameters of Concurrent Methods (MPCM)	MXT – Modify Method-X Time (wait(), sleep(), join(), and await() method calls)
	MSP – Modify Synchronized Block Parameter
	ESP – Exchange Synchronized Block Parameters
	MSF – Modify Semaphore Fairness
	MXC – Modify Permit Count in Semaphore and Modify Thread Count in Latches and Barriers
	MBR – Modify Barrier Runnable Parameter
Modify the Occurrence of Concurrency Method Calls (MOCMC)	RTXC – Remove Thread Method-X Call (wait(), join(), sleep(), yield(), notify(), notifyAll() Methods)
	RCXC – Remove Concurrency Mechanism Method-X Call (methods in Locks, Semaphores, Latches, Barriers, etc.)
	RNA – Replace NotifyAll with Notify
	RJS – Replace Join with Sleep
	ELPA – Exchange Lock/Permit Acquisition
	EAN – Exchange Atomic Call with Non-Atomic
	RTS – Remove Thread Start
	MTS – Move Thread Start
Modify Keyword (MK)	ASTK – Add Static Keyword to Method
	RSTK – Remove Static Keyword from Method
	RSK – Remove Synchronized Keyword from Method
	RSB – Remove Synchronized Block
	RVK – Remove Volatile Keyword
	RFU – Remove Finally Around Unlock
Switch Concurrent Objects (SCO)	RXO – Replace One Concurrency Mechanism-X with Another (Locks, Semaphores, etc.)
	EELO – Exchange Explicit Lock Objects
Modify Critical Region (MCR)	SHCR – Shift Critical Region
	SKCR – Shrink Critical Region
	EXCR – Expand Critical Region
	SPCR – Split Critical Region
	WTIS – Replace While With If Inside Synchronized

- We find that sequential and concurrent mutation operators are independent, i.e., none subsumes the other: concurrent test suites that achieve a high mutation score for sequential, respectively concurrent, mutants do not achieve high mutation score for concurrent, respectively sequential, mutants. This further demonstrates the importance of studying concurrent mutation operators.

2. BACKGROUND

2.1 Existing Mutation Operators

Each mutant is a small syntactic change to the system under test (SUT). Mutants are generated by applying mutation operators, which define how to modify the original code to generate a mutated version. Jia and Harman provide a survey of mutation testing [28], including mutation operators. For example, operators were proposed for procedural languages [12, 30], object-oriented languages [33], and other programming paradigms [5,26]. The most relevant for our study are operators for concurrent code.

Bradbury et al. [6] proposed mutation operators for concurrent Java. However, the cost-effectiveness of these operators has not been evaluated until this study. Table 1

includes their operators, grouped in 5 categories based on the programming constructs that the operators modify [6]. Each category (first column) and operator (second column) are identified by an acronym, e.g., "MK" for the "Modify Keyword" category and RSK for the "Remove Synchronized Keyword from Method" operator. Note that the operators are defined such that the mutated code is similar to common programming mistakes, e.g., RSK changes the method declaration by removing the `synchronized` keyword which can cause data races or atomicity violations. The details of these mutation operators and the example mutants can be found in the original paper [6]. We additionally introduce three new operators described in Section 3.1.

2.2 Selective Mutation

Selective mutation [4, 37, 40, 41, 49], initially called "constrained mutation" [37], reduces the number of mutants by selecting a subset of mutation operators—called *sufficient mutation operators*. The subset should be selected such that test suites that kill all non-equivalent mutants from the selected subset also kill almost all non-equivalent mutants generated by all mutation operators. In contrast to the optimization techniques for mutation testing (e.g., [19, 51, 57]), which are sound in that they do not affect the mutation score but only compute it faster, selective mutation is a heuristic that could change the mutation score. Therefore, selective mutation strongly relies on empirical studies.

Mathur [37] originally proposed that selective mutation removes (two) operators that generate the most mutants. The operators that generate the most mutants were termed *dominant operators*.

Offutt et al. [41] named the approach that removes n most dominant operators as *n-selective mutation* and empirically evaluated it for $n \in \{2, 4, 6\}$. Their study showed that 2, 4, and 6-selective mutation can achieve savings of over 20%, 40%, and 60% with effectiveness of 99.99%, 98.84%, and 88.71%, respectively. In the follow-up study, Offutt et al. [40] extended selective mutation to different categories of mutation operators; the categories are created based on the programming constructs that are mutated, e.g., expressions, operands, or statements. In the same study, Offutt et al. introduced the *"99% rule"*, which says that only a set of operators that achieves (non-selective) mutation score of 99% or more is *sufficient*. (In practice this rule may be relaxed, because a sufficient mutation score would depend on the quality requirements of the software project, as with other coverage metrics.) The experiments considered 22 operators (for the Fortran programming language), and the results showed that a category with 5 operators is sufficient.

Barbosa et al. [4] defined guidelines for selecting a sufficient set of operators *across* categories. The experiments considered 71 mutation operators (for C), and the results identified a set of 10 operators that gave the best results and provided savings higher than the previous studies.

Namin et al. [49] performed a statistical analysis on 108 mutation operators (for C) to identify a set of operators that can predict the mutation score accurately. The resulting set contains 28 operators.

Most recently, Zhang et al. [55] compared operator-based mutant selection techniques with two random mutant selection techniques: (1) *one-round*, which selects mutants uniformly from the set of all mutants, and (2) *two-round*, which selects mutants in two phases: first it uniformly selects a mutation operator and then selects a mutant generated by that mutation operator. They compared operator-based selection techniques [4,40,49] with random selection where the number of mutants selected randomly was 50%, 75%, and 100% of the mutants generated by operator-based selection techniques. The comparison was needed for all operator-based selection techniques, because they do not subsume one another and the original evaluations were done on different sets of programs and mutation operators. The results showed that random selection is even slightly better than all operator-based selection techniques.

All these previous studies on selective mutation focused on *sequential* mutation operators. In contrast, we investigate selective mutation for *concurrent* mutation operators.

3. MUTATION INFRASTRUCTURE

In this section, we first introduce three new concurrent mutation operators that we propose based on our experience with testing concurrent code [19,25,43,45]. We then present our tool for generating mutants for concurrent code. We finally describe mutant execution for concurrent code, which greatly differs from mutant execution for sequential code.

Note that we also describe some key design decisions that others can use when they build tools for concurrent mutation testing and describe some important constraints that arise when testing concurrent code, not only for mutation testing, but testing in general.

3.1 New Mutation Operators

We propose three new mutation operators (shown in bold in Table 1): RTS, MTS, and WTIS. These new mutation operators generate mutants that are similar to common programming mistakes [6, 17, 31]. RTS removes an invocation of the `start` method on a `Thread` object.

MTS moves an invocation of the `start` method on a `Thread` object. Specifically, MTS moves this invocation statement within the same block of statements to the destination statement that is the post-dominator of the original statement. The mutants generated by RTS and MTS can cause the program to deadlock if the affected thread is supposed to send a signal to the other thread(s) or can cause an incorrect result to be calculated if a thread has not been executed.

WTIS replaces `while` with `if` inside a `synchronized` block. A correct implementation of monitors in Java requires `while` due to spurious thread wake-ups. WTIS is a well known bug pattern that can cause an order violation [17].

Our intuition was that all three operators would create mutants that greatly differ from the mutants generated by the other operators and thus all three operators could end up in the sufficient sets of operators. Interestingly, our experiments show that only WTIS ends up in sufficient sets of operators, while RTS and MTS get subsumed by other mutation operators. This also shows the importance of empirically evaluating selective mutation because intuition about independence of the operators can be misleading.

3.2 Mutant Generation

To evaluate selective mutation for concurrent operators, it is necessary to have a tool that implements all mutation operators from Table 1. Also, it is important that the tool handles large codebases. Unfortunately, existing mutation tools for concurrent code [7, 19, 35] implement only a subset of the mutation operators and/or are not reported to scale.

We developed a tool, called *Comutation*[2], that implements all the mutation operators from Table 1. Also, Comutation can be applied to larger codebases (Section 4); in fact, some programs we mutated in our study are larger than any program used in any previous study on selective mutation.

We next discuss how Comutation inserts mutations in the SUT and why an alternative would not apply for concurrent code. There are two common approaches to inserting mutations. One approach is to create multiple copies of the SUT and insert one mutation in each copy, thereby generating as many copies as there are mutants; we follow this approach.

An alternative approach is to use *mutant schemata* [51] to create only one mutated version of the SUT for all mutants, which uses a global ID to specify which mutant (if any) should be enabled. For (intra-class) sequential mutation operators, both approaches are applicable, the trade-off laying essentially in the performance cost imposed by each approach. For concurrent mutation operators (and inter-class sequential mutation operators [42]), on the other hand, mutant schemata would require significant global changes in the SUT for some mutation operators.

For example, consider the RVK operator that removes the `volatile` keyword from a field declaration. (According to the Java memory model [36], this keyword specifies that each access to the field should introduce a happens-before edge.) Mutant schemata would require creating two fields (one `volatile`, one not) and instrumenting every field access to determine which of the two fields to access. Moreover, the particular instrumentation could change the semantics of the code (e.g., an access to the non-volatile field could introduce a happens-before edge [36] due to the instrumentation).

Although mutant schemata can significantly improve the performance of mutation testing [51], it *is orthogonal to selective mutation and does not directly impact any results presented in this paper*. Namely, we measure the savings of selective mutation in terms of the *number* of mutants that are *not generated* (and in terms of exploration cost), not in terms of the time saved for not generating those mutants. This is consistent with previous studies [4, 37, 40, 41, 49, 55].

3.3 Mutant Execution

A substantial difference between concurrent and sequential tests is that the result of concurrent tests depends on thread schedules: the same concurrent test code can produce one result for one schedule and another result for another schedule. This inherent non-determinism of concurrent tests raises important questions for mutation testing, including the theoretical questions of defining when a mutant is killed and practical questions of (quickly) determining if a mutant can be killed.

Defining Killed Mutants. For (deterministic) sequential code, a mutant is killed if, for some test, the observable *output* of the mutated code differs from the output of the original code. For concurrent code, a mutant is killed if, for some test, the set of observable *outputs* of the mutated code for *all* possible schedules differs from the set of outputs of the original code for *all* possible schedules [10, 19]. Hence, determining *precisely* if a concurrent mutant is killed is much harder than for a sequential mutant. Given a concurrent program, a test, and a mutant, it would be necessary to execute all possible schedules of both the original program and the mutant to determine if the mutant is killed. Doing so, however, is not practical, as the number of schedules to explore can be huge.

Approximating Killed Mutants. Several approaches are used in practice to reduce the number of explored schedules for a concurrent test. Some approaches pick a single schedule [25], a set of random schedules [15], or a subset of schedules with a limited number of thread preemptions [29, 38]. Exploring only a subset of schedules can lead to finding only a subset of all possible outputs. When the outputs are simply "pass" or "fail", these approaches can fail to find that a test can fail for some schedule (false negative); this is *not* specific to mutation testing.

What is specific to mutation testing is defining mutants to be killed based on different subsets of all possible outputs. As a consequence, it may fail to detect that a mutant is killed (false negative) if a schedule that would lead to a different output is never executed in the mutated code. Also, it may report that a mutant is killed when this should not be the case (false positive) if a different output is obtained on the mutated code than on the original code.

While it may seem that false positives can be avoided by running the same schedules on the mutated and original code, note that this *cannot* be done [10]. First, in theory, it may be impossible to enforce the execution of the same schedules in the original and mutated run because the changes in the mutant may enable or disable some schedules (e.g., grabbing some locks that prevent threads from executing or releasing some locks that allow other threads to execute). Second, in practice, it is very hard to tightly control the scheduler for larger programs where the execution can depend on real time, as is the case for some of our experiments (Section 4).

Collecting Killed Mutants. To enable the evaluation of selective mutation, Comutation builds a full *test-mutant matrix* to encode all tests that kill each mutant. In other words, Comutation does not stop executing tests on a mutant after it is killed but rather executes all tests on each mutant. (In the actual use of mutation testing, one would stop executing tests on a mutant after it is killed for the first time.)

More precisely, Comutation does not execute all tests on each mutant but uses a common mutation testing optimization [19, 30, 44, 47] that determines mutants that are *hit*. A mutant is hit if its mutated location is executed while running a test on the original code; Comutation does not execute mutants that are not hit because they cannot be killed for that test. While generating the mutants, Comutation instruments the original code to be able to detect which mutants are *hit* during the original run. After generating the mutants, Comutation executes all tests on the original code and collects mutants that are hit. (Note that the instrumentation has no effect on concurrent behavior as long as we control the schedule, which is done for most of our experiments, and in other cases we keep the instrumentation very lightweight, i.e., only collect mutant IDs.)

Note that Comutation potentially executes the original code only for a subset of schedules, so it may fail to find that some mutant could be hit (and thus could incorrectly label the mutant as not killed). However, even for toy programs

[2]Currently, the Comutation tool itself cannot be made publicly available because Intel did not authorize the release. However, the mutants generated by Comutation are publicly available at `http://mir.cs.illinois.edu/comutation/`. Note that the mutants are sufficient to reproduce the results of the study.

it may not be feasible to explore all the schedules. Also note that for some concurrent mutation operators it can be hard to track if a mutant is hit or not, even if all schedules would be explored. For example, this is the case for the RVK operator (for reasons similar as in mutant schemata). For mutants generated by such operators, Comutation runs all tests. Finally note that some concurrent mutation operators make several changes to the code (e.g., EELO can replace `synchronized (o1) { synchronized (o2) ...` with `synchronized (o2) { synchronized (o1) ...`). In such cases, Comutation considers that a mutant is hit if *any* change is executed, as similar to higher-order mutation of sequential code [23].

4. STUDY

This section describes the experiments that we carried out to evaluate selective mutation for concurrent code. The main goals of the evaluation were to (1) determine sets of mutation operators for concurrent code that researchers can use to trade-off the savings and the effectiveness of mutation testing, (2) explore relationship between the savings in terms of the number of mutants and in terms of exploration cost, and (3) evaluate whether sequential and concurrent mutation operators are independent.

Specifically, we address the following questions for concurrent operators, which subsume *all* questions from previous studies on *sequential* mutation operators (citation listed by each question) and include three new questions:

Q1: [41]; What are the most dominant mutation operators for concurrent code?

Q2: [40, 41]; Is n-selective mutation applicable in the context of mutation testing of concurrent code?

Q3: [40]; Which category of mutation operators can achieve the highest (non-selective) mutation score?

Q4: [4]; What is the set of mutation operators that achieves a high (non-selective) mutation score and significant savings (in terms of the number of mutants)?

Q5: [49]; What are the highest savings that can be achieved for different values of (non-selective) mutation scores?

Q6: Which sets of concurrent mutation operators provide good trade-offs for uses in future research studies?

Q7: [55]; How do operator-based and random mutant selection compare for concurrent mutation operators?

Q8: How do the savings in terms of number of mutants and the savings in terms of exploration cost relate?

Q9: Are concurrent and sequential mutation operators independent, i.e., none subsumes the other?

To answer these questions, we performed experiments on 14 Java programs that differ in size, number of tests, and concurrent constructs. The experimental setup closely followed the experiments done for mutation testing of sequential code.

We performed all the experiments on a machine with a 4-core Intel Core i7 2.70GHz processor and 4GB of main memory, running Linux version 3.2.0.

Table 2: Subject programs used in the experiments

Subject	#LOC	#Mutants	#Tests	#States	#Trans.
Account	52	6	22	503691	2343256
Accounts	43	6	8	1645	4679
Airline	38	5	18	88654	320434
Allocation	78	5	20	66998	141179
Barber	135	65	11	38978256	86820210
Bubble	37	2	12	81629	249985
Buffer	89	9	8	98887	223918
Guava$^{v.11.1}$	57395	336	569	na	na
LinkedList	179	4	32	2717344	11610491
Lucene$^{v.3.5}$	68218	738	334	na	na
MergeSort	200	16	2	1022530	1464879
Pool$^{v.1.5.2}$	4664	451	243	na	na
Shop	113	8	9	33187	65864
Tree	122	38	15	437695	670085
$\sum$	131363	1689	1303	44030516	103914980

4.1 Subject Programs

Table 2 lists the subject programs used in the study. For each subject, we tabulate its name, number of lines of code without comments or whitespace (reported by CLOC [11]), total number of concurrent mutants generated by our tool, number of tests, and number of explored states and transitions if applicable (more details later in the section).

The programs come from different sources [16, 19, 21, 32, 46]. Ten (small) programs contain classes that demonstrate various concurrent Java constructs and spawn most of the threads in tests (although small, these programs are implemented using appropriate concurrent constructs and are preferable when comparing testing techniques); all these programs were previously used in research on testing concurrent code [19,25]. Three (big) programs are from real-world open-source code: `Guava` [21] is a Google project that includes concurrent collections and concurrency libraries; `Lucene` [32] is an Apache project that implements a concurrent text search engine; and `Pool` [46] is an Apache project that provides a concurrent object-pooling API. We used the original code for this study, without any special modifications.

Note that `Guava`, `Lucene`, and `Pool` are *substantially larger than any program used in previous studies* on selective mutation [4, 37, 40, 41, 49, 55].

4.2 Generated Mutants

The number of mutants generated by a set of mutation operators varies based on the language constructs used in the SUT. Some studies report a high number of sequential mutants even for very simple code [49]. However, a (realistic) concurrent (Java) program uses concurrent constructs sparingly. Therefore, *the number of mutants generated by concurrent mutation operators is not as high as the number of mutants generated by sequential mutation operators* for code of similar size. As discussed in Section 1, selective mutation is still highly desirable for multithreaded code because execution/exploration of tests even on a single mutant is expensive, so reducing the cost is important.

Table 3 shows the number of mutants for each subject and mutation operator. The $\sum$M column and row show the total number of mutants for each subject and mutation operator, respectively. All the mutants are generated automatically

Table 3: Number of generated mutants with all mutation operators for all the subjects. ∑M is the total number of generated mutants, ∑H is the total number of hit mutants, and ∑K is the total number of killed mutants

	MPCM				MOCMC							MK						SCO	MCR					
Subject	MXT	MSP	ESP	MXC	RTXC	RCXC	RNA	RJS	ELPA	**RTS**	**MTS**	ASTK	RSTK	RSK	RSB	RVK	RFU	RXO	SHCR	SKCR	EXCR	SPCR	**WTIS**	∑M
Account	0	1	0	0	0	0	0	0	0	0	0	0	0	3	1	0	0	0	1	0	0	0	0	6
Accounts	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	0	0	0	1	0	1	0	0	6
Airline	0	0	0	0	1	0	0	1	0	1	0	0	0	0	2	0	0	0	0	0	0	0	0	5
Allocation	0	0	0	0	0	0	0	0	0	0	0	0	0	3	1	0	0	0	0	1	0	0	0	5
Barber	0	32	0	0	3	0	0	0	0	0	0	0	0	0	8	0	0	0	9	1	3	6	3	65
Bubble	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	2
Buffer	0	0	0	0	4	0	2	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	9
Guava	0	113	0	2	1	13	1	1	7	3	0	0	0	13	115	36	3	0	9	0	1	17	1	336
LinkedList	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	1	0	0	1	4
Lucene	12	114	1	2	23	11	19	2	5	3	1	1	1	248	83	45	5	0	65	28	28	35	6	738
MergeSort	0	0	0	0	2	0	0	3	0	3	2	0	3	3	0	0	0	0	0	0	0	0	0	16
Pool	0	109	0	0	15	0	7	0	0	0	0	0	0	106	89	8	0	0	49	13	22	32	1	451
Shop	0	0	0	0	0	0	0	0	0	0	0	1	0	4	2	0	0	0	0	0	0	1	0	8
Tree	0	0	0	6	1	9	0	1	8	1	0	0	0	2	1	0	0	9	0	0	0	0	0	38
∑M	12	370	1	10	50	33	29	8	20	11	3	2	4	384	310	89	8	9	134	44	55	91	12	1689
∑H	4	322	1	6	46	29	28	7	17	6	3	2	2	324	261	*0	7	9	126	42	51	88	10	
∑K	0	130	0	5	9	19	0	5	9	5	0	0	0	37	109	0	0	9	46	13	4	4	4	

∗ Comutation does not detect hit mutants generated by RVK mutation operator; we run all test cases for each RVK mutant (see Section 4.4)

by applying Comutation on the original code. As can be seen, Comutation can handle real-world examples. Mutation operators are grouped the same way as in Table 1. Values for some operators are not shown in Table 3 because they generate 0 mutants in the programs used in the study.

Mutation operators can be sorted based on how dominant they are [37, 41], i.e., based on the number of mutants that the operators generate (row ∑M in Table 3). Interestingly, the distribution of mutants generated by concurrent mutation operators is highly non-uniform, similar to the distribution for sequential operators [40, 41].

Answer 1: RSK is the most dominant mutation operator for concurrent code, closely followed by MSP and RSB.

4.3 Test Suites

All programs came with a number of manually written tests. For all programs but `Guava`, `Lucene`, and `Pool`, we inspected all the mutants not killed by the original tests to identify which mutants are equivalent to the original code, and then we created additional tests to kill these mutants. For the three big examples, it was not feasible to inspect non-equivalent mutants. Therefore, following previous studies [40, 41, 49, 55], we treat all non-killed mutants as equivalent. Table 2 shows the number of tests that kill all non-equivalent mutants. As pointed by others [49], equivalent mutants are of no interest to researchers who would use the mutants for comparison of test suites and testing techniques.

For `Guava` (which has a total of over 200,000 tests) and `Lucene` (which has a total of over 1,500 tests) we execute only a subset of tests to make the study feasible and focused on concurrent code. Specifically, we select all test methods from the test classes that reference `Thread` at least once.

As common in studies of mutation testing, we remove from the test suite *ineffective* tests that do not kill any mutant.

Our experiments use one test suite per program. All previous studies [37, 40, 41, 49, 55] on selective mutation testing tried to remove the potential bias of one test suite by generating a large number of test suites and averaging results across all of them. However, even the early experiments by Offutt et al. [40] showed that different test suites lead to very similar results, and the most recent study by Zhang et al. [55] shows almost negligible variance in (non-selective) mutation score across various test suites (standard deviation of under 0.25 for the score of over 99.00). Moreover, selecting multiple test suites is possible for sequential code because there are subjects available with large test pools (e.g., SIR programs [14]), sometimes generated using tools for automatic generation of tests. Unfortunately, there are no comparable subjects for concurrent code or test *generation* tools [39].

4.4 Mutant Execution

As described in Section 3.3, Comutation can use different exploration approaches to execute/explore tests for (mutated) concurrent code. Our experiments use two exploration approaches. (1) Stress testing simply runs the original or mutated SUT (with no further modifications) on a regular JVM. (2) Java PathFinder (JPF) [52] is an explicit-state model checker implemented as a special JVM that can explore all possible schedules; it also supports a limited number of thread preemptions [38].

In our study, we used the most thorough exploration possible for each subject: we used JPF to execute all schedules up to 6 preemptions if feasible and used stress testing when JPF runs out of memory, tests depend on real time, or have long execution time. Specifically, we executed all programs but `Guava`, `Lucene`, and `Pool` with JPF. We executed `Guava`, `Lucene`, and `Pool` using a single stress run because some tests depend on real time and some tests that spawn a number of threads incur long execution time.

4.5 Hit/Killed Mutants

Table 3 shows the total number of hit/killed mutants for each subject in the ∑H/∑K row. To determine these numbers, we followed the procedure from Section 3.3, using stress runs or JPF for each subject. Recall from Section 3.3 that the precise numbers of hit and killed mutants may differ because of false positives and false negatives. In total, there are 15 operators that create at least one mutant that is killed.

4.6 Collecting Cost-Effectiveness Data

Traditionally [40, 41], evaluations of selective mutation use two types of test suites: a test suite that kills all non-equivalent mutants generated by a selected set of mutation operators (*selective adequate test suite*) and a test suite that

Table 4: Average/Total values for {2, 4, 6, MK, MCR, MOCMC, MPCM, SCO}-selective mutation adequate tests. Detail results are shown only for 2-selective mutation because of the space limit

Subject	%Tests	#Live	Score[%]	#NSM	#SM	%Save
Account	83.33	3	50.00	6	2	66.67
Accounts	0.00	0	100.00	6	4	33.33
Airline	0.00	0	100.00	5	5	0.00
Allocation	10.00	2	50.00	5	2	60.00
Barber	0.00	0	100.00	65	33	49.23
Bubble	0.00	0	100.00	2	1	50.00
Buffer	0.00	0	100.00	9	9	0.00
Guava	0.00	0	100.00	336	210	37.50
LinkedList	0.00	0	100.00	4	4	0.00
Lucene	0.00	0	100.00	738	376	49.05
MergeSort	0.00	0	100.00	16	13	18.75
Pool	32.76	7	81.58	451	236	47.67
Shop	33.33	0	100.00	8	4	50.00
Tree	0.00	0	100.00	38	36	5.26

n-selective	Avg.			Total		Avg.
2-selective	11.38	0.85	**91.54**	1689	935	**33.39**
4-selective	47.28	9.00	63.23		491	63.24
6-selective	49.73	9.28	59.28		311	67.33

Category	Avg.			Total		Avg.
MK	22.65	1.00	**83.66**	1689	797	**51.53**
MCR	59.92	9.92	47.38		336	84.10
MOCMC	66.33	15.71	40.38		154	80.87
MPCM	59.53	3.21	44.78		393	85.17
SCO	92.85	26.92	7.14		9	98.30

kills all non-equivalent mutants generated by all mutation operators (*non-selective adequate test suite*). In our study, we generate each adequate test suite by taking *all* tests that kill at least one mutant (from selected mutants or all mutants, for respective test suites).

More precisely, each experiment follows these three steps: (1) select a subset of mutation operators; (2) find an adequate test suite for the mutants generated by the selected mutation operators; and (3) measure the non-selective mutation score achieved by the test suite with respect to all mutation operators. As in previous studies [40, 41, 49, 55], we measure non-selective mutation score relative to *non-equivalent* mutants (rather than relative to *all* generated mutants). The optional fourth step is to check if the "99% rule" is satisfied: a set of operators is *sufficient*, i.e., non-selective mutation score is above 99% (Section 2.2).

4.7 Results for Selective Mutation

n-Selective Mutation. Initially [37, 40, 41], selective mutation was proposed to remove n most dominant mutation operators (Section 2.2) and was evaluated for $n \in \{2, 4, 6\}$. Our first set of experiments evaluates n-selective mutation but in the context of concurrent mutation operators.

Table 4 shows the detailed per-program values only for 2-selective mutation because of space limit (the top of the table). "%Tests" is the percentage of tests that do *not* belong to the selective adequate test suite relative to the non-selective adequate test suite. "#Live" is the number of all (non-equivalent) mutants not killed by the selective adequate test suite. "Score[%]" is the non-selective mutation score achieved with a selective adequate test suite with respect to all (non-equivalent) mutants. "#NSM" is the number of mutants generated by non-selective mutation operators. "#SM" is the number of mutants generated by selective mutation operators. "%Save" is the percentage of mutants saved if selective mutation is used, i.e., `(#NSM - #SM) / #NSM`. In brief, "%Save" and "Score[%]" show the cost savings and effectiveness of selective mutation, respectively.

The middle of Table 4 shows the average/total values across all the programs, first for 2-selective mutation and then for several other sets of operators. To compare two sets of operators, we use the average effectiveness ("Score[%]") and cost savings ("%Save"), as in the previous studies [4, 37, 40, 41, 49, 55]. Using simple averages (rather than weighting by the number of mutants) avoids biasing the results by programs with more mutants [49].

The average "Score[%]" values for 2- (without {RSK, MPS}), 4- (without {RSK, MPS, RSB, SHCR}), and 6- (without {RSK, MPS, RSB, SHCR, SPCR, RVK}) selective mutation are all below 99% and thus not sufficient. Moreover, as n increases, the average "Score[%]" decreases more rapidly than for sequential mutation operators.

> *Answer 2: Different from selective mutation for sequential mutation operators, n-selective mutation testing ($n \in \{2, 4, 6\}$) is* not *applicable to mutation testing for concurrent mutation operators.*

Selective Mutation for Categories. As described in Section 2.1, mutation operators can be grouped in categories based on the constructs they modify. Our second set of experiments evaluates selective mutation based on categories [40] for concurrent mutation operators. The bottom of Table 4 shows the average values, across all subject programs, of mutation score for all 5 categories from Table 1.

> *Answer 3: The "Modify Keyword" (MK) category is the most effective. However, different from sequential mutation operators, no category achieves sufficient score.*

Trade-offs in Terms of Savings and Mutation Scores. Our next set of experiments evaluates *all* possible sets of mutation operators even from different categories, i.e., all sets of operators of size 1, 2, 3, ..., 15. The operators that generate no killed mutant are not considered. Note that the previous studies on selective mutation for sequential operators could not perform this exhaustive analysis because they had a larger number of operators that generate killed mutants. Overall, we evaluated 2^{15} sets of operators; for each set, we first (1) measure the non-selective mutation score for each program and (2) average the score over all programs. We then (3) identify the set that achieves maximum savings for each average score (Pareto optimality). We finally (4) plot the maximum savings and indicate the size of the set (1-9 operators) using an appropriate shape (Figure 1). (If there are multiple sets with the same values, we plot the smallest set size.) Only five sets are sufficient; the set that satisfies the "99% rule" and achieves the highest savings of 46.37% includes these operators: {MSP, RSK, RTXC, SHCR, SKCR, SPCR, WTIS}. We also identify the set that has the mutation score of 100% and achieves the highest savings (42.98%); the set includes: {RCXC, RJS, RSB, RSK, SHCR, WTIS}. It is worth mentioning that all sets that achieve savings over 99% require WTIS ("Replace While With If Inside Synchronized"), one of our new mutation operators (Section 3).

> *Answer 4: {MSP, RSK, RTXC, SHCR, SKCR, SPCR, WTIS} is the set that achieves the highest savings (46.37%) with the sufficient mutation score (99.67%).*

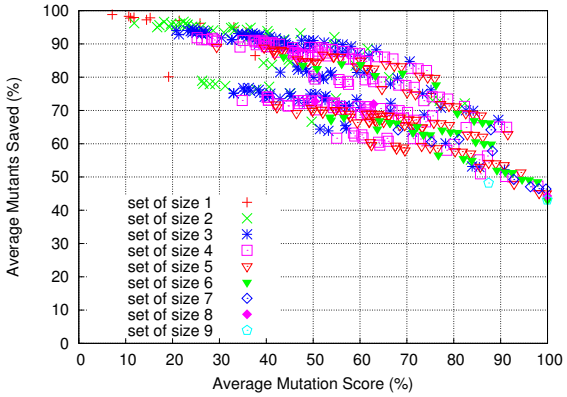

Figure 1: The best savings for mutation scores considering all sets of mutation operators (averaged across all programs)

Table 5: Sets of mutation operators that achieve the closest values to targeted mutation score and savings

Targeted [%]		Achieved [%]			Operators
%Save	Score[%]	%Save	Score[%]	%States	
MAX	100	42.98	100.00	40.58	{RCXC, RJS, RSB, RSK, SHCR, WTIS}
MAX	>99	46.37	99.67	44.95	{MSP, RSK, RTXC, SHCR, SKCR, SPCR, WTIS}
∼50	MAX	49.14	96.10	49.21	{MSP, RSK, RTXC, SHCR, SKCR, WTIS}
∼60	MAX	59.97	91.54	65.43	{ELPA, MXC, RCXC, RJS, RSB, SHCR, WTIS}
∼70	MAX	69.90	84.14	72.82	{RJS, RSB, WTIS}
∼80	MAX	79.77	75.96	84.31	{RTXC, SHCR, SKCR, SPCR, WTIS}
∼90	MAX	89.28	58.10	93.41	{RJS, SHCR, SPCR}

Answer 5: Figure 1 shows the maximum savings for each value of the mutation score. For the score over 95%, the savings ranges from 40% to 50%. For the savings over 90%, the score is about 60%.

Sets of Mutation Operators for Future Studies. While each previous study [4, 37, 40, 41, 49, 55] provided *one set of sequential mutation operators* that researchers can use to speed up mutation testing without sacrificing effectiveness, we provide *several sets of concurrent operators* that researchers can use to trade-off the cost and effectiveness of mutation testing. Table 5 shows the sets of mutation operators that achieve the values closest to the targeted mutation score and/or savings. (If multiple sets achieve the same savings and mutation score, we show only one because of the space limits.) If the goal is to maintain high effectiveness (over 99%) for concurrent mutation operators, one can only reduce the number of mutants by about half (savings of 46.37%); however, if the goal is to greatly reduce the number of mutants (say, 10X, for savings of about 90%), one can still do that but be prepared to have the expected effectiveness reduced (score of 58.10%).

Table 6: Operator-based vs. random selection. Random selection is done based on the set of operators that achieve mutation score of 99.67%

Subject	Score[%]					
	one-round random			two-round random		
	50%	75%	100%	50%	75%	100%
Account	75.33	90.00	100.00	65.00	83.33	100.00
Accounts	0.00	0.00	100.00	0.00	0.00	100.00
Airline	0.00	0.00	100.00	0.00	0.00	100.00
Allocation	40.50	68.00	89.00	44.50	64.50	82.50
Barber	100.00	100.00	100.00	100.00	100.00	100.00
Bubble	0.00	0.00	100.00	0.00	0.00	100.00
Buffer	100.00	100.00	100.00	100.00	100.00	100.00
Guava	76.53	85.07	90.44	75.02	83.11	89.75
LinkedList	61.00	61.00	85.00	55.00	55.00	91.00
Lucene	100.00	100.00	100.00	100.00	100.00	100.00
MergeSort	100.00	100.00	100.00	100.00	100.00	100.00
Pool	92.69	96.05	98.21	93.11	96.16	98.68
Shop	50.00	50.00	86.00	50.00	50.00	81.00
Tree	0.00	0.00	100.00	0.00	0.00	100.00
	Avg.					
	56.86	60.72	96.33	55.90	59.43	95.92

Answer 6: Table 5 is a key contribution of this paper and provides researchers with sets of operators to use in future studies on (mutation) testing of concurrent code.

Operator-based vs. Random Mutant Selection. Our next experiments compare operator-based and random mutant selection. Similar to the recent study for sequential mutation operators [55], we consider both one-round and two-round random selection strategies (Section 2.2). We compare them to the set of concurrent mutation operators that achieves the score of 99.67%. Our experiments follow the previous study [55] in randomly selecting 50%, 75%, and 100% of the number of mutants generated by the operator-based selection (46.37% of all mutants).

Table 6 shows the results of this experiment. Each number is averaged across 50 runs of random selection. The "Avg." numbers should be compared to 99.67%. We can see that the operator-based mutant selection is slightly better than random selection for the programs used in our experiments. This contrasts the results on sequential code [55] which showed that random selection performs slightly better than the operator-based selection in most of the cases. In agreement to the results on sequential code [55], our experiments show that one-round and two-round random selections are about the same.

Answer 7: Operator-based mutant selection performs slightly better than random selection for concurrent mutation operators.

Savings in Terms of the Number of States. All previous studies on selective mutation measured the cost of mutation testing using only the number of mutants as the metric. This appears reasonable for sequential code as one can expect that running tests on mutants takes *on average* similar time across different mutants. However, for concurrent code, tests are not simply executed but rather explored for various schedules. We could not tell a priori how reducing the number of mutants affects the cost of exploring the state space for concurrent tests. For example, it could have been the case that reducing the number of mutants in half reduces the exploration cost 10X (because the selected mutants are much cheaper to explore).

We performed experiments to measure the savings of selective mutation in terms of the exploration cost, specifically in terms of the number of states that JPF explores for the tests and selected mutants. Jagannath et al. [27] showed that the number of explored states relates to exploration time. Our experiments were performed for all the programs that are explored with JPF (all but the three largest programs). Table 5 shows the average results in the column "%States". Although the table does not include "%Save" values averaged across programs explored by JPF our findings are based on those values. We do not show the detailed results for the space limit. (Note that "%States" column should not be directly compared with "%Save" because the latter is computed over *all* subjects.) Our analysis of the two kinds of savings shows the following.

Answer 8: The savings in terms of the number of mutants correspond to the savings in terms of exploration cost for concurrent code.

Sequential vs. Concurrent Mutation Operators. Our final set of experiments was designed to check whether *concurrent* tests that achieve high mutation score for sequential, respectively concurrent, mutants also achieve high mutation score for concurrent, respectively sequential, mutants. First, we generate sequential mutants using the Javalanche mutation tool [47]. Javalanche uses selective mutation for sequential mutants and implements four sufficient operators: Negate Jump Condition (NJC), Omit Method Call (OMC), Replace Arithmetic Operator (RAO), and Replace Numerical Constant (RNC). Second, we collect the test-mutant matrix that encodes which tests kill which mutants; we run the same set of tests as for the concurrent mutants. The numbers of generated, hit, and killed mutants are shown in Table 7.[3] Third, we create for each program 20 test suites that kill all concurrent mutants, following the approach similar to Zhang et al. [55] to create each test suite: randomly select a test, include the test in the test suite if it kills at least one new mutant, and repeat the steps until all concurrent mutants are killed. We measure mutation score for sequential mutation operators achieved by the created test suites. On average, across all the programs, mutation score for sequential mutation operators is 57.78%. As Javalanche uses sufficient operators, the mutation score for all sequential operators should be similar. This finding demonstrates that sequential mutants are not subsumed by concurrent mutants. Fourth, we randomly select for each program 20 test suites that kill all sequential mutants, following the same approach to create test suites as in the previous experiment. We use these suites to measure mutation score for concurrent mutation operators. On average, across all the programs, mutation score for concurrent mutants is 84.73%. This finding demonstrates that concurrent mutants are not subsumed by sequential mutants.

[3]Unfortunately, Javalanche was unable to run tests for `Guava` and `Lucene` as code uses annotations on method parameters, which causes Javalanche/ASM instrumentation to crash. Also, tests in these projects do not properly shut down thread pools, which leads to runtime exceptions when executed with Javalanche.

Table 7: Number of generated sequential mutants with Javalanche [47]

Subject	NJC	OMC	RAO	RNC	∑M	∑H	∑K
Account	2	2	8	12	24	20	18
Accounts	8	27	2	27	64	57	49
Airline	7	2	3	19	31	31	18
Allocation	18	6	0	53	77	53	26
Barber	7	10	7	20	44	42	34
Bubble	5	3	4	16	28	28	21
Buffer	5	19	1	8	33	33	29
LinkedList	14	9	0	14	37	32	24
MergeSort	19	23	15	94	151	123	66
Pool	486	1351	126	1463	3426	2257	1712
Shop	6	12	3	40	61	50	43
Tree	16	25	6	50	97	93	77
∑M	593	1489	175	1816	4073		
∑H	458	1033	139	1189		2819	
∑K	413	808	102	794			2117

Table 8: The best savings, with sufficient mutation score, obtained in all studies on selective mutation

			Selective Mutation Study			
			[40]	[4]	[49]	[this]
Evaluation	Original	Language	Fortran	C	C	Java
		Code	sequential			conc.
		#Mutation Op.	22	71	108	27
		#Selected Op.	5	10	28	7
		%Save	77.66	65.01	92.60	46.37
	[55]	Language	C			na
		Code	sequential			
		#Mutation Op.	108			
		%Save	92.72	83.96	92.58	

Answer 9: Concurrent and sequential mutation operators are independent, i.e, none subsumes the other, demonstrating the importance of the study on concurrent mutation operators.

Summary of Savings in Studies on Selective Mutation. Table 8 shows the best savings (in terms of the number of mutants) reported in all studies on selective mutation testing, where each savings is achieved with a sufficient set of mutation operators. The best savings for concurrent code are substantially smaller (46.37%) than the best savings for sequential code (sometimes over 90%). One potential reason for smaller reduction is the smaller number of concurrent mutation operators and generated mutants. Another reason may be that concurrent mutation operators are more carefully selected, targeting generation of different types of bugs. We do believe that more concurrent mutation operators are required because of the spectrum of bugs that they have to cover [31], even if at the same time these operators generate a large number of mutants. Indeed, the goal of our study was not to achieve high savings but to evaluate selective mutation for concurrent code and to provide several sets of concurrent mutation operators that researchers can use in their future studies.

5. THREATS TO VALIDITY

Internal Validity. The main uncontrolled factors that may influence the results of our study are potential faults in our

tool for mutating code, collecting execution results, and analyzing the results. To reduce this threat, we reviewed many outputs of mutation testing tool and inspected the results of the execution of mutants on small programs.

External Validity. The conclusions of the study are based on the results obtained on 14 programs, which may not be representative for all programs and selective mutation. Firstly, some of the programs are small and do not come from production code. Secondly, some of the programs may be considered "old" in that they mostly use synchronized block/keyword rather than explicit locks. Lastly, no program uses new concurrency constructs (e.g., Java 7 includes Fork/Join parallel framework). To overcome the potential issues, the study relies on the programs that differ in size and constructs. All the programs have been used in many previous studies on different topics. Also, our study includes three (big) programs that are used in production. It remains as the future work to perform experiments for more mutation operators that focus on new parallel constructs.

Construct Validity. First, in our evaluation, we used only one pool of tests to select test suites for each program, which we extended for simple example to kill all non-equivalent mutants. The results may greatly differ if different test suites are used in the experiments. Unfortunately, we were unable to follow the approach taken in previous studies on sequential selective mutation to generate test cases automatically: the available tool [39] for test case generation for concurrent code is not fully automated and works with one class at a time. Our study relies on the previous empirical studies on sequential selective mutation that show that test suite does not have a significant impact on the results [40, 55]. Second, we mark all non-killed mutants as equivalent (for big examples). Having an actual set of equivalent mutants (which would require manual inspection) can potentially lead to different results. Our approach follows previous studies on selective mutation in deeming mutants equivalent if they are not killed by any test from the entire test pool. Third, we limit the number of preemptions (JPF) and runs (stress) to make the study feasible. These can lead to false positives and false negatives. To reduce this threat, we set a relatively high value for preemption bound.

6. RELATED WORK

There has been a lot of research on mutation testing in the last three decades, investigating different aspects of the technique. Some of the aspects include proposals of mutation operators for different programming paradigms [18,30], development of automated tools targeting different programming languages [12, 24, 34, 47], proposals for optimizing the technique, including the execution process and reduction in number of mutation operators [23, 37, 51, 56, 57], and evaluation of the optimizations. Jia and Harman describe many aspects of mutation testings in their recent survey [28]. In this section we relate our study to other optimization techniques, coverage metrics, and tools.

Our study is closely related to studies on sequential mutation operators [4, 37, 40, 41, 49, 55], discussed in Section 2.

Mutant schemata [51] merges all mutated versions of code together in, so called, metamutant. Schemata reduces compilation time, as metamutant has to be compiled only once, which can provide significant savings. Many existing tools for mutation testing implement this optimization. Currently our infrastructure for mutation testing of concurrent code creates one version of the code for each mutant. Schemata are orthogonal to selective mutation.

Higher-order mutation [23] optimizes mutation testing by replacing a number of mutants with one mutation, called first-order mutants (FOMs), with one mutant with a number of mutations, called a higher-order mutant (HOM). Subsuming HOM is harder to kill than FOMs and guarantees that FOMs are killed if HOM is killed. There has been some initial work on HOM for concurrent mutation operators [53].

MuTMuT [19] is an optimization that targets mutation testing of concurrent code. It optimizes the execution of mutants by reducing the state space that has to be explored for each mutant. Reduction up to 77% is reported. MuTMuT is an orthogonal optimization to selective mutation.

New coverage metrics were proposed specifically for concurrent code [3, 8, 48, 50, 54]. It would be valuable to investigate the correlation between mutation testing for concurrent code and these coverage metrics.

Following the development of mutation operators, a number of tools have been implemented for many programming languages including Java [24, 34, 47]. MuJava [34] implements both intra-class and inter-class mutation operators and was among the first tools for Java. Jumble [24] is an industrial tool for mutation testing that mutates bytecode and focuses on efficiency. ConMAn [7] implements a set of concurrent mutation operators. Javalanche [47] is a recent tool for mutation testing that uses selective mutation.

7. CONCLUSIONS

This paper described the first study on selective mutation for concurrent mutation operators. The results show important differences between concurrent mutation operators and sequential mutation operators: selective mutation is applicable to concurrent code but provides lower savings than for sequential code, selecting operators based on the number of mutants or categories of operators does not provide good effectiveness, and random selection is not as good as selective mutation. Second, we provide several sets of concurrent operators that researchers can use to trade-off the cost and effectiveness of mutation testing. Third, we show evidence that the savings in terms of the number of mutants correspond to the savings in terms of exploration cost for concurrent code. Fourth, we show that sequential and concurrent mutation operators are independent, i.e., concurrent tests suites that kill all sequential, respectively concurrent, mutants do not achieve high mutation score for concurrent, respectively sequential mutants, which demonstrates the need for the study on selective concurrent mutation operators.

In the future, we would like to investigate relation between concurrent mutation, recently proposed coverage criteria for concurrent code, and real bugs. We also believe that future studies on (mutation) testing of concurrent code could use the sets of operators we found (Table 5).

8. ACKNOWLEDGMENT

We thank Sarfraz Khurshid, Darko Marinov, and Aleksandar Milicevic for their feedback on this work, and Danny Dig, Hassan Eslami, Yu Lin, Qingzhou Luo, and Stas Negara for comments on a draft. This material is based upon work partially supported by Illinois-Intel Parallelism Center (I2PC), the National Science Foundation under Grant Nos. CCF-1012759 and CCF-0746856.

9. REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. 2008.

[2] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.

[3] Z. L. B. Krena and T. Vojnar. Coverage metrics for saturation-based and search-based testing of concurrent software. In *RV*, pages 53–62, 2011.

[4] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *STVR*, 11:113–136, 2001.

[5] L. Bottaci. Type sensitive application of mutation operators for dynamically typed programs. In *Mutation*, pages 126–131, 2010.

[6] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation operators for concurrent Java. In *Mutation*, pages 83–92, 2006.

[7] J. S. Bradbury, J. R. Cordy, and J. Dingel. Comparative assessment of testing and model checking using program mutation. In *Mutation*, pages 210–219, 2007.

[8] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *PPoPP*, pages 206–212, 2005.

[9] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, pages 167–178, 2010.

[10] R. H. Carver. Mutation-based testing of concurrent programs. In *ITC*, pages 845–853, 1993.

[11] Count Lines of Code home page. `http://cloc.sourceforge.net/`.

[12] M. E. Delamaro and J. C. Maldonado. Proteum — a tool for the assessment of test adequacy for C programs. In *CPCS*, pages 79–95, 1996.

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, pages 34–41, 1978.

[14] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, pages 405–435, 2005.

[15] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Syst. J.*, 41:111–125, 2002.

[16] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs. *CCPE*, 19:267–279, 2007.

[17] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS*, pages 286–293, 2003.

[18] S. Ghosh and A. P. Mathur. Interface mutation. *STVR*, 11:227–247, 2001.

[19] M. Gligoric, V. Jagannath, and D. Marinov. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In *ICST*, pages 55–64, 2010.

[20] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, pages 174–186, 1997.

[21] Guava home page. `http://code.google.com/p/guava-libraries/`.

[22] R. G. Hamlet. Testing programs with the aid of a compiler. *TSE*, 3:279–290, 1977.

[23] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *FSE*, pages 212–222, 2011.

[24] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, and M. Utting. Jumble Java byte code to measure the effectiveness of unit tests. In *Mutation*, pages 169–175, 2007.

[25] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *FSE*, pages 223–233, 2011.

[26] V. Jagannath, M. Gligoric, S. Lauterburg, D. Marinov, and G. Agha. Mutation operators for actor systems. In *Mutation*, pages 157–162, 2010.

[27] V. Jagannath, M. Kirn, Y. Lin, and D. Marinov. Evaluating machine-independent metrics for state-space exploration. In *ICST*, pages 320–329, 2012.

[28] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *TSE*, 37:649–678, 2011.

[29] JPF home page. `http://babelfish.arc.nasa.gov/trac/jpf/`.

[30] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *SPE*, 21:685–718, 1991.

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

[32] Lucene home page. `http://lucene.apache.org/core/`.

[33] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In *ISSRE*, pages 352–364, 2002.

[34] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava: a mutation system for Java. In *ICSE*, pages 827–830, 2006.

[35] P. Madiraju and A. Namin. ParaMu – a partial and higher-order mutation tool with concurrency operators. In *Mutation*, pages 351–356, 2011.

[36] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, pages 378–391, 2005.

[37] A. Mathur. Performance, effectiveness, and reliability issues in software testing. In *COMPSAC*, pages 604–605, 1991.

[38] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[39] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. BALLERINA: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *ICSE*, pages 727–737, 2012.

[40] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *TOSEM*, 5:99–118, 1996.

[41] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *ICSE*, pages 100–107, 1993.

[42] J. Offutt, Y.-S. Ma, and Y.-R. Kwon. The class-level mutant of MuJava. In *AST*, pages 78–84, 2006.

[43] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*, pages 2–11, 2010.

[44] PIT home page. `http://pitest.org/`.

[45] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. E. Gottschlich, J. Ha, and Y. Wu. CoreRacer: A practical memory race recorder for multicore x86 tso processors. In *MICRO*, pages 216–225, 2011.

[46] Apache Commons Pool home page. `http://commons.apache.org/pool/`.

[47] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA*, pages 69–80, 2009.

[48] E. Sherman, M. B. Dwyer, and S. Elbaum. Saturation-based testing of concurrent programs. In *FSE*, pages 53–62, 2009.

[49] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ICSE*, pages 351–360, 2008.

[50] R. Taylor, D. Levine, and C. Kelly. Structural testing of concurrent programs. *TSE*, 18:206–215, 1992.

[51] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *ISSTA*, pages 139–148, 1993.

[52] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Springer Autom. Softw. Eng.*, 10:203–232, 2003.

[53] L. Wu and G. Kaiser. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators. In *SEKE*, pages 244–249, 2011.

[54] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *ISSTA*, pages 153–162, 1998.

[55] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *ICSE*, pages 435–444, 2010.

[56] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *ISSTA*, 2013. to appear.

[57] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *ISSTA*, pages 331–341, 2012.