Change-Aware Preemption Prioritization

Vilas Jagannath, Qingzhou Luo, Darko Marinov
Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{vbangal2, qluo2, marinov}@illinois.edu

ABSTRACT

Successful software evolves as developers add more features, respond to requirements changes, and fix faults. Regression testing is widely used for ensuring the validity of evolving software. As regression test suites grow over time, it becomes expensive to execute them. The problem is exacerbated when test suites contain multithreaded tests. These tests are generally long running as they explore many different thread schedules searching for concurrency faults such as dataraces, atomicity violations, and deadlocks. While many techniques have been proposed for regression test prioritization, selection, and minimization for sequential tests, there is not much work for multithreaded code.

We present a novel technique, called Change-Aware Preemption Prioritization (CAPP), that uses information about the changes in software evolution to prioritize the exploration of schedules in a multithreaded regression test. We have implemented CAPP in two frameworks for systematic exploration of multithreaded Java code. We evaluated CAPP on the detection of 15 faults in multithreaded Java programs, including large open-source programs. The results show that CAPP can substantially reduce the exploration required to detect multithreaded regression faults.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Reliability, Experimentation

Keywords

Prioritization, Testing, Multithreaded

1. INTRODUCTION

The advent of multicore processors is ushering in a new era in computing. To exploit performance from the multiple cores, software developers now need to write parallel code. The currently dominant paradigm for parallel code is that of shared data, where multiple threads of computation communicate by reading and writing shared data objects. For example, Java provides support for threads in the language and libraries, with shared data residing on the heap. However, multithreaded code is notoriously hard to get right and is often afflicted by hard to detect faults like dataraces, atomicity violations, and deadlocks.

Ensuring the reliability of multithreaded code has been an active area of research with several promising recent results [8, 10, 15, 21, 25]. Most of these tools execute multithreaded tests to check for the presence of faults. Since multithreaded code can have different behavior for different thread schedules, these tools conceptually *explore* the code for a large number of schedules, and as a result they tend to be fairly time consuming. Moreover, most existing tools are *change-unaware*: they check only one version of code at a time, and do not exploit the fact that code evolves.

Regression testing is the most widely practiced method for ensuring the validity of evolving software. Regression testing involves re-executing the tests for a program when its code changes to ensure that the changes have not introduced a fault that causes test failures. As programs evolve and grow, their test suites also grow, and over time it becomes expensive to re-execute all the tests. The problem is exacerbated when test suites contain multithreaded tests that are generally long running. While many techniques have been proposed to alleviate this problem for sequential tests [39], there is much less work for multithreaded code [16, 38].

Yoo and Harman [39] present a detailed survey of regression testing techniques that minimize (e.g., [19]), select (e.g., [17, 37]), or prioritize (e.g., [14, 20, 40]) test suites. Test selection determines which tests to rerun after changing code, and test prioritization determines in what order to run tests to find faults faster. The techniques for sequential code showed good results in practice (e.g., [33]) but unfortunately cannot be applied directly for multithreaded code. Specifically, those techniques do not target exploration of schedules *within* one test.

There is some recent work on targeting program changes in systematic testing for multithreaded code [16, 38]. The proposed techniques reuse results from exploration of one program version to speed up exploration of the next program version (or a code mutant). These techniques in effect perform *selection*, pruning from exploration the schedules that are unaffected by the code changes, which is complementary to prioritization. In general, prioritization could be used in conjunction with selection to prioritize already selected parts of the exploration.

ISSTA '11, July 17-21, 2011, Toronto, ON, Canada

```
1  public void filterWrite(NextFilter nextFilter,
2                          IoSession session,
3                          WriteRequest writeRequest)
4                          throws Exception {
5    ...
6    flushWithoutFuture();
7    ...
8  }
9  public void flushWithoutFuture() {
10   Queue<Object> bufferQueue = getMessageQueue();
11   for (;;) {
12     Object encodedMessage = bufferQueue.poll();
13     if (encodedMessage == null) {
14       break;
15     }
16     // Flush only when the buffer has remaining.
17     if (!(encodedMessage instanceof IoBuffer) ||
18       ((IoBuffer) encodedMessage).hasRemaining()) {
19       SocketAddress destination = writeRequest.getDestination();
20       WriteRequest writeRequest = new EncodedWriteRequest(
               encodedMessage, null, destination);
21       nextFilter.filterWrite(session, writeRequest);
22     }
23   }
24 }
```

Figure 1: Code snippet from Mina revision 912148

This paper makes the following contributions:

Technique: We propose a new technique, called *Change-Aware Preemption Prioritization (CAPP)*, that uses information about the changes in software evolution to prioritize the exploration of schedules in a multithreaded regression test. The goal of CAPP is to find a fault faster if one exists. Our technique decides in what order to explore thread schedules based on how test exploration dynamically encounters changed code.

Heuristics: CAPP is a general technique that can be instantiated with different definitions of code changes and scheduling choices to prioritize. We present 14 heuristics that consider changes at the level of source-code lines/statements, methods, classes, or fields affected by the change, and consider prioritizing scheduling choices based on whether all or only some executing threads are executing changed code.

Implementation: We have implemented CAPP in two frameworks for systematic exploration of multithreaded Java code. Java PathFinder (JPF) [22, 36] is a widely used tool for checking Java code. JPF performs a stateful search, with checkpointing and restoration of program state to explore thread schedules, and with state comparison to prune the search. ReEx is a tool that we implemented following the ideas from CHESS [25]. ReEx performs a stateless search, with re-execution to establish a program state to explore thread schedules, and with no state comparison.

Evaluation: We evaluated CAPP and its heuristics on the detection of 15 faults in multithreaded Java code, including some large open-source programs. Our evaluation addresses the following research questions:

RQ1: How much reduction in exploration cost does CAPP provide over change-unaware techniques?

RQ2: How do the heuristics compare with each other?

RQ3: How do the results compare across stateful/stateless exploration?

RQ4: How do the results compare across default/randomized search order?

```
1  public void filterWrite(NextFilter nextFilter,
2                          IoSession session,
3                          WriteRequest writeRequest)
4                          throws Exception {
5    ...
6    Queue<Object> bufferQueue = getMessageQueue();
7    while (!bufferQueue.isEmpty()) {
8      Object encodedMessage = bufferQueue.poll();
9      // Flush only when the buffer has remaining.
10     if (!(encodedMessage instanceof IoBuffer) ||
11       ((IoBuffer) encodedMessage).hasRemaining()) {
12       future = new DefaultWriteFuture(session);
13       nextFilter.filterWrite(session, writeRequest);
14     }
15   }
16   ...
17 }
```

Figure 2: Code snippet from Mina revision 912149

```
1  class FilterWriteThread extends Thread {
2      ...
3      int result = 0;
4      ...
5      public void run() {
6          try {
7              pc.filterWrite(nextFilter, session, writeRequest);
8              ...
9          } catch (Exception e) {
10             e.printStackTrace();
11             result = 1;
12         }
13     }
14 }
15
16 @Test
17 public void testFilterWriteThreadSafety() {
18     ...
19     FilterWriteThread fwThread1 = new FilterWriteThread(...);
20     FilterWriteThread fwThread2 = new FilterWriteThread(...);
21     fwThread1.start(); fwThread2.start();
22     fwThread1.join(); fwThread2.join();
23     assertEquals(0, fwThread1.result);
24     assertEquals(0, fwThread2.result);
25 }
```

Figure 3: Multithreaded regression test for Mina

In short, the results show that CAPP can substantially reduce the exploration cost required to detect multithreaded regression faults, and there are interesting variations in cost among heuristics, stateful/stateless search, and default/randomized search order.

2. EXAMPLE

We illustrate how CAPP works through an example of a real code evolution of Apache Mina [1]. Figures 1 and 2 show code snippets from two consecutive revisions of Mina. In the newer revision, 912149, developers inlined the invocation of the method `flushWithoutFuture` into the method `filterWrite` and further changed the loop condition to use the predicate `!bufferQueue.isEmpty()`. While performing these changes, the developers also removed the `null` check for `encodedMessage` (lines 13-15 in Figure 1).

These changes in fact introduce a fault caused by an atomicity violation: if a preemption occurs after a thread has checked `!bufferQueue.isEmpty()` and before it calls `bufferQueue.poll()`, another thread could remove elements from

the `bufferQueue`, and `encodedMessage` could be assigned `null`, which will result in a `NullPointerException`. This issue was reported in Mina's issue tracking system (DIRMINA-803 [1]) and corrected since then.

Figure 3 shows a multithreaded test that exercises the changed code. This test creates two threads that call the faulty `filterWrite` method. Suppose that a Mina developer were to run this test after making the change to revision 912149. The sooner the test reveals the fault, the easier it is for the developer to debug [30].

However, this test reveals the fault in only a small number of its many possible schedules. Ideally, the test should be explored for all (non-equivalent) schedules to guarantee detection of the `NullPointerException`, but exploring all schedules can be very expensive. Even using advanced testing techniques to reduce the number of schedules, such as iterative context bounding from CHESS [25], requires executing many different schedules. Specifically, in this example, using the basic *change-unaware*, iterative context bounding exploration (with a bound of 2) *requires exploring 58 schedules* before the fault is revealed. In contrast, using *CAPP requires exploring as few as 5 schedules* before the fault is revealed, which is substantially faster.

CAPP can reduce the exploration required to reveal a fault by inferring the impact of code changes and prioritizing the exploration of schedules that perform preemptions at *Impacted Code Points (ICPs)*. CAPP uses different kinds of ICPs based on changed code lines/statements, methods, and classes, and the impact of these changes onto fields. Section 3.1 describes in detail how CAPP infers different kinds of ICPs. For example, in Mina revision 912149, CAPP marks as changed the highlighted lines in Figure 2. By analyzing these changed lines, CAPP infers that the method `filterWrite` and the class it belongs to are impacted by the changes. Moreover, while no fields are being directly accessed within the changed lines, CAPP analyzes the methods being invoked directly from the changed lines (in this case the methods `isEmpty` and `poll`), and infers that the fields `head` and `tail` in the `Queue` class are also impacted by the change. Using ICPs like these, CAPP prioritizes the exploration of the multithreaded test to focus on changes and thus reveal the fault with substantially lesser exploration.

CAPP can use various heuristics to identify and prioritize change-impacted preemptions based on the set of collected ICPs. Section 3 describes in detail the 14 different heuristics that we propose. These heuristics are expected but not guaranteed to reveal the faults faster than the change-unaware exploration. Section 4 presents our empirical evaluation. For example, the heuristic Line On-stack All, which prioritizes the exploration of schedules that encounter states where *all* enabled threads are executing changed-impacted *lines* (such as lines 6-15 in Figure 2), revealed the fault in just 5 schedules in this test. As another example, the heuristic Field Some, which prioritizes the exploration of schedules that encounter states where *some* enabled threads are accessing change-impacted *fields* (such as `head` and `tail`), revealed the fault in 19 schedules.

3. TECHNIQUE

Change-Aware Preemption Prioritization consists of two main parts: static collection of a set of Impacted Code Points (ICPs) and dynamic prioritization of the exploration of schedules that perform preemptions at the collected ICPs. We first present collection of ICPs and then present the algorithm for prioritization of schedules.

3.1 Collecting Impacted Code Points

Collecting ICPs is similar to change-impact analysis [26, 27, 33]. However, the goal of collecting ICPs is to identify points that are more likely to affect fault-revealing schedules and hence should be prioritized earlier. Note that we do *not* ensure that the collected ICPs capture the sound or complete impact of changes: CAPP can identify fewer points than really impacted (because CAPP performs prioritization and not selection/pruning, the unidentified points will still be explored, but later), and CAPP can identify more points than really impacted (because those points may be helpful in finding an appropriate schedule). Intuitively, our focus is on capturing the impact of changes on the communication among threads, i.e., the schedule-relevant points in the code. Since concurrency faults are related to synchronization orders and shared-memory accesses, CAPP collects not only directly changed code elements but also their impact on synchronized regions (blocks/methods) and fields (of shared objects).

An ICP is defined as a 4-tuple $\langle C, M, L, F\rangle$, where C is a class name, M is a method name, L is a line number, and F is a field name. An element of the tuple may be $\perp$ to denote a "don't care" value. For example, the ICP $\langle$`org.apache.mina...ProtocolCodecFilter`, `filterWrite()`, 325, $\perp\rangle$ denotes that line 325, which is in the method `filterWrite()` of the class `org.apache.mina...ProtocolCodecFilter`, is impacted by the changes. As another example, the ICP $\langle$`java.util.concurrent.ConcurrentLinkedQueue`, $\perp$, $\perp$, `head`$\rangle$ denotes that the field `head` of the `java.util.concurrent.ConcurrentLinkedQueue` class is impacted by the changes.

Our CAPP implementation utilizes a multi-step process to collect the set of ICPs. First, a diff utility (specifically, the Eclipse JDK structure diff [34]) is used to collect a set of lines that have been changed. This results in a set of ICPs where only the third element, i.e., the line, is specified. Then four analyses are performed on the abstract syntax tree (AST) of the changed code to fill in the missing elements of the partial ICPs and add additional ICPs.

First, any partial ICPs with changed lines that affect a synchronized region (e.g., adding/removing the `synchronized` keyword to/from methods, changing the scope of a `synchronized` block, etc.) are expanded to include the entire region (method/block). For example, if line 325 in `filterWrite()` were to belong to some `synchronized` block from lines 320 to 330, additional ICPs are added for all those lines.

Second, for each partial ICP, the method and class that contain the changed lines are identified and filled into the partial ICP. This is straightforward except for some special cases such as inner or anonymous classes.

Third, for any field accesses (reads or writes) within impacted lines, additional ICPs are added that specify change-impacted fields. For example, if the changed code has an access `o.f` for some object `o` of type `C`, an ICP $\langle$`C`, $\perp$, $\perp$, `f`$\rangle$ is added. Note that this ICP includes no (changed) lines. Indeed, it encodes that *any* access to the field is potentially relevant and not only the accesses within the changed code.

Fourth, additional change-impacted field ICPs are collected by determining the read- and write-sets [29] of all methods that are *directly* invoked from the impacted lines, and using fields from these sets. In case of dynamic dis-

patch, our implementation does not compute any precise call graph but simply approximates the set of callees using the statically declared type of the receiver objects.

3.2 Algorithm

CAPP uses the statically collected ICPs to dynamically prioritize the exploration of a multithreaded regression test. Figure 4 shows the pseudo-code of the algorithm used to perform the prioritized exploration. The algorithm takes as inputs the test to be explored and a set of ICPs. The algorithm also has two parameters—prioritization mode and ICP match mode—that identify which heuristic to use (or none if `BASIC`). The possible values for these modes are listed at the top and are explained later in this section.

The algorithm uses the `Tran` pair to represent a transition that consists of a `State` and a `Thread` that can be executed in that state. The main data structures of the algorithm are `toExplore` and `nextToExplore`, which are both sets of transitions, to be explored either in the current iteration or in the next iteration, respectively. Typically these structures would be stacks (for the depth-first search strategy), queues (for the breadth-first search strategy), or priority queues (for search strategies like iterative context bounding that use other prioritization in addition to CAPP). Our algorithm is orthogonal to the search strategy and does not presume any particular strategy. The algorithm also works for both stateful exploration (where `explored` tracks the explored states) and stateless exploration. For example, our ReEx tool applies the iterative context bounding prioritization strategy from CHESS [25].

The algorithm starts by initializing the data structures. The `toExplore` set is initialized with the enabled transitions of the initial state of the test, and the `nextToExplore` set is initialized to the empty set. The main exploration loop starts after the initialization and continues as long as `toExplore` is not empty. In each iteration of the main loop, a transition is selected and removed from the `toExplore` set. The state of the selected transition is reestablished (e.g., by restoring the state in JPF or re-executing the code in ReEx), and the thread of the transition is executed on the state to obtain the next state, `s'`. The algorithm then obtains the transitions that are enabled in `s'`. At this point, a selection criteria can be used to remove some enabled transitions from the enabled set.

The core part of the algorithm is the call to the function `partitionPrioritized` in line 33. This function partitions the enabled transitions into those that CAPP prioritizes for the current iteration and those it postpones for the next iteration, which it adds to `toExplore` and `nextToExplore`, respectively. The partitioning is configured by the prioritization mode and the ICP match mode, described in Section 3.2.1. At the end of the main exploration loop, the algorithm checks whether the `toExplore` set is empty; if so, the transitions from the `nextToExplore` set are moved into the `toExplore` set to begin the next iteration of the CAPP exploration. This is repeated until the entire state space has been explored. However, it would be also possible to stop the loop after one or more iterations, which would result in selection rather than prioritization of schedules.

3.2.1 Modes

The algorithm takes two parameters, for the prioritization mode and for the ICP match mode. The prioritization mode

```
1  // Parameters
2  enum PrioritizationMode { BASIC, ALL, SOME }
3  enum ICPMatchMode { CLASS, CLASS_ON_STACK,
4                      METHOD, METHOD_ON_STACK,
5                      LINE, LINE_ON_STACK, FIELD }
6  PrioritizationMode pMode;
7  ICPMatchMode mMode;
8  // Exploration state
9  class Tran { State state; Thread thread; }
10 Set⟨Tran⟩ toExplore; Set⟨Tran⟩ nextToExplore; Set⟨State⟩ explored;
11 // Inputs
12 Test test; Set⟨ICP⟩ impactedCodePoints;
13 PassOrFail explore() {
14   initializeExploration(test);
15   return performExploration();
16 }
17 void initializeExploration() {
18   State s_init = initial state for test;
19   toExplore = {Tran(s_init, t) | t ∈ enabledThreads(s_init)};
20   nextToExplore = explored = {};
21 }
22 PassOrFail performExploration() {
23   while (toExplore ≠ {}) {
24     Tran current = pickOne(toExplore);
25     toExplore = toExplore − { current };
26     restore current.state;
27     State s' = execute current.thread on current.state;
28     if (s' ∉ explored) {
29       if (s' is errorState) return FAIL;
30       Set⟨Tran⟩ enabled = {Tran(s', t') | t' ∈ enabledThreads(s')};
31       enabled = enabled − trans that satisfy some pruning condition
                 such as partial order reduction;
32       Set⟨Tran⟩ prioritized
33         = partitionPrioritized(enabled, current.thread);
34       toExplore = toExplore ∪ prioritized;
35       nextToExplore = nextToExplore ∪ ( enabled − prioritized );
36       if (STATEFUL) explored = explored ∪ { s' };
37     }
38     if (toExplore == {}) {
39       toExplore = nextToExplore;
40       nextToExplore = {};
41     }
42   }
43   return PASS;
44 }
45 Set⟨Tran⟩ partitionPrioritized(Set⟨Tran⟩ trans, Thread current) {
46   if (pMode == BASIC) // prioritization not performed
47     return trans;
48   if (∄ t ∈ trans : t.thread == current) // preemption not possible
49     return trans;
50   Set⟨Tran⟩ impacted = matchICPs(trans);
51   if (pMode == ALL)
52     if (impacted == trans) return trans;
53     else return { pickOne(trans − impacted) };
54   else // (pMode == SOME)
55     if (impacted ≠ {}) return trans;
56     else return { t ∈ trans | t.thread == current };
57 }
58 Set⟨Tran⟩ matchICPs(Set⟨Tran⟩ trans) {
59   Set⟨Tran⟩ matches = {};
60   for (tran ∈ trans) {
61     Instruction pc = tran.thread.pc;
62     StackTrace st = tran.thread.stackTrace;
63     for (icp ∈ impactedCodePoints) {
64       if ((mMode == CLASS ∧ pc.cls == icp.cls)
65       ∨ (mMode == CLASS_ON_STACK ∧ icp.cls ∈ st)
66       ∨ (mMode == METHOD ∧ pc.⟨cls, meth⟩ == icp.⟨cls, meth⟩)
67       ∨ (mMode == METHOD_ON_STACK ∧ pc.⟨cls, meth⟩ ∈ st)
68       ∨ (mMode == LINE ∧ pc.⟨cls, meth, ln⟩ == icp.⟨cls, meth, ln⟩)
69       ∨ (mMode == LINE_ON_STACK ∧ pc.⟨cls, meth, ln⟩ ∈ st)
70       ∨ (mMode == FIELD ∧ pc.instr is fieldInstr
71             ∧ pc.⟨cls, fld⟩ == icp.⟨cls, fld⟩))
72         matches = matches ∪ { tran };
73     }
74   }
75   return matches;
76 }
```

Figure 4: Exploration Prioritization Algorithm

can be `BASIC` (no prioritization), `ALL`, or `SOME`. It stipulates the conditions under which enabled transitions are kept for the current iteration (or postponed for the next iteration):

`ALL (A)` keeps all of the transitions if they are *all* executing a change-impacted point in the code (as determined by the ICPs). Otherwise, if one or more transitions are not executing a change-impacted point, only one of them is kept. The intuition behind this mode is to prioritize preemptions *only among* threads that are executing change-impacted code, and to ignore the threads that are not executing change-impacted code until they reach such code (or become disabled).

`SOME (S)` keeps all of the transitions if there *exists* at least one transition in the set that is executing a change-impacted point in the code. Otherwise, if no transition is executing a change-impacted point, only the transition of the currently executing thread is kept. The intuition behind this mode is to prioritize preemptions *between* threads that are executing change-impacted code and other threads that are not.

Note that both modes perform prioritization only if a preemption is possible, as shown in lines 48-49. If a preemption is not possible, all the enabled transitions are returned. Also note that the prioritization mode relies on the ICP match mode to decide which transitions/threads are executing change-impacted code.

There are seven ICP match modes that determine whether a transition is executing changed-impacted code, based on the `impactedCodePoints` set of collected ICPs. The match modes compare the program counter (i.e., the currently executing line/statement that belongs to some method in some class) and potentially stack trace (which has several program counters based on the call chain) of a transition/thread being executed with the collected ICPs:

`CLASS (C)` checks if the class of the program counter matches the class of an ICP.

`CLASS_ON_STACK (CO)` checks if the stack trace contains a class specified in an ICP.

`METHOD (M)` checks if the method of the program counter matches a method specified in an ICP.

`METHOD_ON_STACK (MO)` checks if the stack trace contains a method specified in an ICP.

`LINE (L)` checks if the line matches a line specified in an ICP.

`LINE_ON_STACK (LO)` checks if the stack trace contains a line specified in an ICP.

`FIELD (F)` checks if a field being accessed at a program counter (if any) matches a field specified in an ICP.

The combination of two (non-`BASIC`) prioritization modes and seven ICP match modes results in 14 different heuristics with which Change-Aware Preemption Prioritization can be instantiated. We refer to the heuristics using the respective ICP match mode and prioritization mode. For example, LS is the *Line Some* heuristic that uses the `LINE` match mode and the `SOME` prioritization mode, and COA is the *Class On-stack All* heuristic that uses the `CLASS_ON_STACK` match mode and the `ALL` prioritization mode.

4. EVALUATION

The motivation behind CAPP is to reduce the exploration required to detect multithreaded regression faults. We designed and performed experiments to determine whether CAPP heuristics can indeed reduce the exploration cost to detect such faults. We also compare the heuristics and analyze their effectiveness across stateful/stateless exploration and default/randomized search orders. The evaluation was conducted in the context of 15 multithreaded faults from a set of Java programs. We next present the implementations of CAPP that we use in the experiments, the artifacts on which we performed the experiments, the experimental setup, analysis of the results, and threats to validity.

4.1 Implementations

We implemented CAPP, along with all its heuristics, in two frameworks for systematic exploration of multithreaded Java programs, Java PathFinder (JPF) and ReEx. JPF is a widely used, stateful model checker for Java programs [36]. We implemented CAPP in JPF by customizing the existing `SimplePrioritySearch` to prioritize the search using CAPP. ReEx is a stateless exploration framework for Java programs that we developed using bytecode instrumentation. We developed a custom `SchedulingStrategy` in ReEx to control the search order using CAPP. The basic search strategy in JPF is (unbounded) depth first, while in ReEx it is the iterative context bounded from CHESS (with bound of 2) [25].

4.2 Artifacts

We conducted our experiments on 15 multithreaded faults in Java programs. Table 1 provides more information about each of the faults and the programs in which those faults were detected. For each of the faults, we collected two versions of the program, a *correct* version without the fault and a *buggy* version with the fault. For each of the faults, we also collected a multithreaded test that passed on the correct version and failed on the buggy version. For many of the programs such a test was included in the test suite. In cases where such a test was not available, we created the appropriate test based on the information gained from the corresponding bug report. Further, some of the tests provided with the programs could be configured with the number of threads to be used. In such instances, we used a small number of threads that revealed the fault. This is in line with standard practice; developers are usually encouraged to write simple unit tests that check a particular property. It is also the rationale behind techniques like CHESS [25] which detects faults with the smallest number of possible preemptions.

The statistics shown in Table 1 are for the buggy version of the programs on which the experiments were conducted. The correct versions were only used to obtain the initial diffs from which the ICPs for the buggy version were computed. The first 8 programs and their faults were obtained from the Software Infrastructure Repository (SIR) [11]. Each of these programs had one fault. We asked two graduate students to fix these faults to obtain the correct versions. The remaining 7 faults were obtained from various open-source Java projects. The correct and buggy versions for these faults were obtained from the respective project's source repository. The programs vary in size from 52 to 54,872 lines of code, and the size of the changes between the buggy and fixed versions ranges from 2 to 201 ICPs. The table

Table 1: Subject Regression Faults and Programs Statistics

	Source	Error	#Threads	#Classes	#Methods	SLOC	#ICP
Airline	[35]	Assertion violation	6	2	18	136	6
Allocation	[35]	Assertion violation	3	3	22	209	5
BoundedBuffer	[35]	Deadlock	9	5	10	110	2
BubbleSort	[35]	Assertion violation	4	3	15	89	4
Deadlock	[35]	Deadlock	3	4	4	52	3
ReadersWriters	[35]	Deadlock	5	6	19	154	2
ReplWorkers	[35]	Deadlock	3	14	45	432	2
RAXextended	[35]	Assertion violation	6	11	23	166	2
Groovy	[32]	Deadlock	3	607	6399	54872	60
Lang	[2]	Assertion violation	3	215	4422	48369	3
Mina	[1]	Assertion violation	3	341	3188	34804	36
Pool1	[3]	Assertion violation	3	51	688	10042	148
Pool2	[4]	Assertion violation	3	35	371	4473	201
Pool3	[5]	Deadlock	2	51	706	10802	29
Pool4	unreported fault	Deadlock	3	51	705	10783	47

also shows the type of error that was detected in each program. The assertion violations detected were all caused by dataraces or atomicity violations.

4.3 Setup

We conducted two sets of experiments with each implementation of CAPP. The first set of experiments measure the savings in terms of exploration cost that is achieved by using the CAPP heuristics compared to the basic exploration strategy of the respective exploration framework. Note that the basic exploration strategy naturally imposes a particular default search order on one exploration, e.g., in JPF depth-first strategy explores transitions enabled from a state in the order of the thread id of the transitions. However, a previous study has shown that exploration savings attributed to heuristics are often a function of the search order rather than the heuristic itself [13]. To evaluate the effect of the search order, we conducted a second set of experiments with randomized search orders. For these experiments, we chose 50 random seeds and, for each seed, performed a randomized depth-first search similar to [12] for basic and heuristic explorations. In total we performed 19,890 explorations which required about a month of computing time to complete.

Note that not all of the artifacts could be used for both frameworks. Specifically, Mina could not be explored using JPF since it uses networking libraries that are currently not modeled by JPF. Also, ReplWorkers, RAXextended, and ReadersWriters could not be explored using ReEx since they are reactive programs where a re-execution/schedule can potentially run infinitely. ReEx currently does not support exploring such programs.

4.4 Measures

Because the experiments were conducted on multiple computers with different configurations (some experiments were performed on compute clusters), we do not measure exploration cost in real time. This is consistent with previous studies on exploration costs [12,13,38]. Instead, we measure the costs of exploration in terms of the *number of transitions (new states + visited states - 1) for JPF* and the *number of schedules (re-executions) for ReEx.*

Note that CAPP prioritization does increase somewhat the per-transition or per-schedule real time cost compared to no prioritization, because CAPP checks for an ICP match (function `matchICPs` in Figure 4). However, for all modes without `ON_STACK`, this check can be rather cheap as it only compares the current program counter with a set of collected ICPs. In fact, the info can be statically pre-computed and each bytecode labeled with a boolean that indicates whether it is an ICP or not; the dynamic match then just checks the value of one boolean variable. For the modes with `ON_STACK`, the check is more expensive as it needs to maintain a stack of values and to compare the current program counter information with those values. Our current CAPP prototypes do not optimize these checks but follow the pseudo-code in Figure 4 fairly directly.

The additional cost of CAPP over no prioritization is the static analysis for collecting ICPs. Yet again, our prototype does not attempt to minimize this cost, but it can be made rather negligible compared to the cost of exploration of numerous schedules for many multithreaded tests.

4.5 Results for Default Search Order

We next present and analyze the results of the first set of experiments measuring the savings in exploration that can be achieved by using various CAPP heuristics with the default search order. Tables 2 and 3 show the results. The second columns of the tables show the number of transitions/schedules required to detect the fault using the basic, change-unaware exploration provided by JPF/ReEx. The following columns show the speedup (if greater than 1.0) or slowdown (if less than 1.0) obtained by the various CAPP heuristics compared to the basic exploration. The highest speedup achieved for the detection of each fault is highlighted. The last row in both tables shows the geometric mean of the speedups achieved by the various CAPP heuristics for all the faults. Recall from Section 3 that the heuristic acronyms are built using the ICP match mode and the prioritization mode that define the heuristic. For example the Line On-stack Some (LOS) heuristic uses the `LINE_ON_STACK` ICP match mode and the `SOME` prioritization mode. We use the results presented in these tables to address the first three research questions presented in Section 1.

4.5.1 RQ1: Exploration Cost

For stateful JPF exploration, the average reductions in exploration cost range from 1.0x for COS and MOS to 2.7x for MA, with the only average cost increases being 0.8x for LOS. For stateless ReEx exploration, the average reductions range from 1.5x for COS to 5.3x for LOA, with no average cost increases. Looking at individual faults, all the heuristics reduce cost in the detection of majority of the faults. For stateful JPF exploration, the greatest speedup of 37.8x was obtained by COA for detecting the RAXextended fault, and

Table 2: Default search order results for JPF

	Basic	CA	CS	COA	COS	MA	MS	MOA	MOS	LA	LS	LOA	LOS	FA	FS
Airline	1328	1.0	1.0	1.0	1.0	1.0	0.9	1.0	1.0	1.0	0.9	1.0	1.0	**1.3**	1.0
Allocation	1573	1.1	1.0	1.1	1.0	1.1	1.0	1.1	1.0	1.0	0.4	1.0	0.4	**4.1**	1.0
BoundedBuffer	1391	1.4	1.1	1.4	1.0	1.4	1.1	1.4	1.0	**1.8**	1.5	0.9	1.5	**1.8**	1.5
BubbleSort	3880	1.2	1.0	1.2	1.0	0.9	1.1	2.6	1.0	0.9	1.1	2.6	1.0	**3.2**	1.1
Deadlock	81	1.0	1.0	0.6	1.0	1.0	1.0	1.0	1.0	1.0	1.5	1.0	1.5	**3.5**	1.8
ReadersWriters	1545	1.1	0.7	1.0	1.0	**3.4**	0.3	3.0	0.3	1.3	0.4	1.3	0.4	3.3	0.2
ReplWorkers	5003	0.8	1.2	7.4	1.2	7.0	1.3	6.8	1.2	**7.9**	4.8	**7.9**	4.8	2.8	4.8
RAXextended	31987	0.6	2.9	**37.8**	0.8	0.6	2.9	0.6	0.8	1.1	1.9	1.1	0.5	1.7	5.9
Groovy	6721	**11.6**	1.0	**11.6**	1.0	**11.6**	1.0	**11.6**	1.0	**11.6**	1.0	**11.6**	1.0	**11.6**	1.0
Lang	1268	1.5	1.1	1.5	1.1	1.5	1.1	1.5	1.1	1.7	**1.8**	1.7	**1.8**	1.5	1.1
Pool1	2978	6.6	2.0	2.8	1.0	**7.2**	2.3	2.8	1.0	1.2	2.2	1.2	1.9	1.0	0.7
Pool2	5077	6.4	1.4	3.1	1.0	8.7	4.9	3.1	1.0	29.4	6.7	**35.5**	1.1	1.4	2.7
Pool3	507	2.6	1.2	1.6	1.0	1.8	1.2	1.8	1.0	2.6	0.2	2.6	0.02	**5.1**	0.3
Pool4	9327	4.8	1.8	2.9	1.0	**30.0**	8.3	15.0	2.8	0.8	1.5	1.0	1.4	0.8	9.8
Geom. Mean		1.9	1.2	2.4	1.0	**2.7**	1.4	2.4	1.0	2.0	1.2	2.2	0.8	2.4	1.4

Table 3: Default search order results for ReEx

	Basic	CA	CS	COA	COS	MA	MS	MOA	MOS	LA	LS	LOA	LOS	FA	FS
Airline	44312	1.0	1.0	1.0	1.0	13.4	1.0	**13.5**	1.0	13.4	1.0	**13.5**	1.0	2.2	1.0
Allocation	4101	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.4	0.6	0.4	0.6	**512.6**	1.0
BoundedBuffer	1329	22.2	4.5	22.2	4.5	22.2	4.5	22.2	4.5	**34.1**	4.5	**34.1**	4.5	13.3	4.5
BubbleSort	5179	**1294.8**	1.1	**1294.8**	1.1	**1294.8**	1.1	**1294.8**	1.1	**1294.8**	1.1	**1294.8**	1.1	**1294.8**	1.1
Deadlock	6	1.2	0.6	1.2	0.6	1.2	0.6	1.2	0.6	1.2	0.8	1.2	0.8	1.5	**2.0**
Groovy	19	**1.0**	0.4	**1.0**	0.4	**1.0**	0.4	**1.0**	0.4	**1.0**	0.4	**1.0**	0.4	0.6	0.7
Lang	9	2.3	**3.0**	2.3	**3.0**	2.3	**3.0**	2.3	**3.0**	2.3	**3.0**	2.3	**3.0**	2.3	**3.0**
Mina	58	0.6	3.1	0.9	2.9	0.7	2.6	0.6	3.9	0.7	2.6	**11.6**	8.3	0.6	3.1
Pool1	6463	4.4	8.7	1.2	8.0	30.6	22.6	1.2	8.0	**119.7**	113.4	10.6	22.0	1.3	8.0
Pool2	98	0.3	1.2	0.1	1.3	0.4	1.4	0.1	1.3	**16.3**	3.6	12.3	1.6	2.3	8.2
Pool3	2	**2.0**	1.0	1.0	1.0	**2.0**	1.0	1.0	1.0	0.5	1.0	0.5	1.0	0.5	1.0
Pool4	13593	6.3	15.7	2.0	1.1	17.8	31.9	121.4	16.1	1.0	15.3	1.0	0.9	1.1	**199.9**
Geom. Mean		3.1	1.9	2.3	1.5	5.1	2.1	3.8	1.9	5.1	2.4	**5.3**	1.7	4.3	3.1

the greatest slowdown of 0.02x was obtained by LOS for detecting the Pool3 fault. For stateless ReEx exploration, the greatest speedup of 1294.8x was obtained by the `ALL` prioritization mode based heuristics for detecting the BubbleSort fault, and the greatest slowdown of 0.1x was obtained by MOA for detecting the Pool2 fault.

> *The CAPP heuristics do reduce exploration costs on average, ranging from 1.0x to 5.3x. Only 1 instance (out of 28) increase costs on average with 0.8x.*

4.5.2 RQ2: Comparison of Heuristics

For JPF, grouping the heuristics by the ICP match mode, `METHOD` heuristics perform the best, followed by `FIELD` heuristics, and only the `METHOD_ON_STACK` and `LINE_ON_STACK` heuristics have average slowdowns. For ReEx, `LINE` heuristics perform the best, followed by `FIELD` heuristics, with the worst being `CLASS_ON_STACK` heuristics. Grouping the heuristics by prioritization mode, each `ALL` heuristic outperforms its corresponding `SOME` heuristic for both JPF and ReEx.

> *Heuristics based on the `FIELD` ICP match mode perform better than heuristics based on the other ICP match modes. Heuristics based on the `ALL` prioritization mode perform better than heuristics based on the `SOME` prioritization mode.*

4.5.3 RQ3: Stateful vs Stateless Exploration

The CAPP heuristics achieve speedups for both stateful and stateless explorations, but on average the speedups are greater for stateless exploration. There are also a few other differences between the performance of the heuristics across stateful and stateless exploration. While MA (with 2.7x speedup) is the best heuristic on average for stateful exploration, LOA (with 5.3x speedup) is the best heuristic on average for stateless exploration. While stateful exploration has two heuristics with average slowdowns (MOS and LOS), none of the stateless heuristics have average slowdown. Grouping the heuristics by prioritization mode, the `ALL` heuristics outperform the corresponding `SOME` heuristics for both stateful and stateless exploration. However, grouping heuristics by ICP match mode, `METHOD` is the best for stateful exploration, while `LINE` is the best for stateless exploration. `FIELD` performs consistently well for both stateful and stateless exploration.

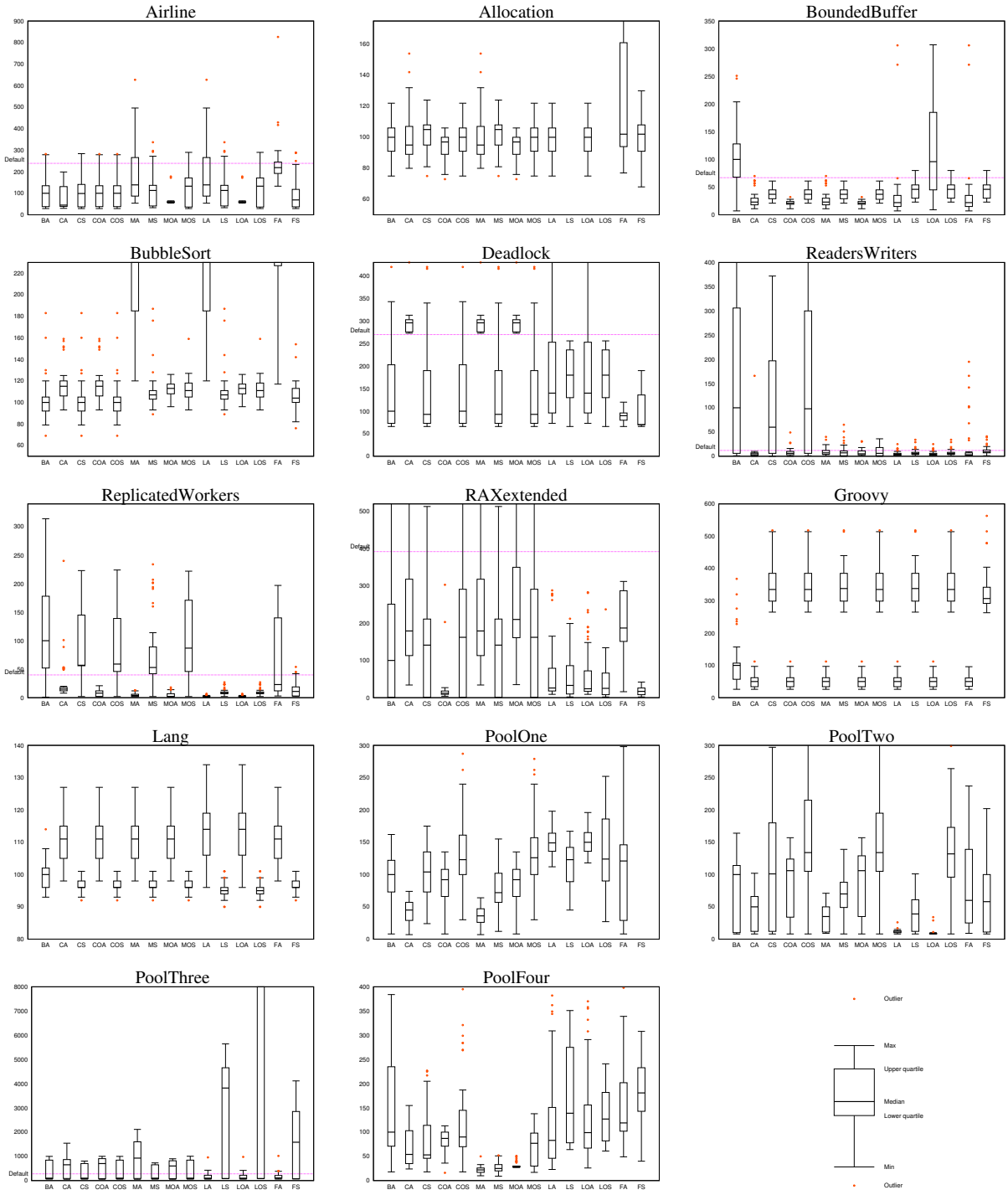
> *The CAPP heuristics achieve greater reduction in exploration costs for stateless exploration. `ALL` and `FIELD` heuristics perform well for both stateful and stateless exploration.*

While the default strategy in JPF is depth-first search (DFS), we also evaluated CAPP with the breadth-first search (BFS) strategy. The absolute numbers of transitions required to detect the faults were orders of magnitude larger for BFS than for DFS. However, BFS with the CAPP heuristics still performed better than Basic BFS and, in fact, achieved around twice as high average cost reduction than achieved for DFS with CAPP over Basic DFS.

4.6 Results for Randomized Search Order

Tables 4 and 5 show the results of the second set of experiments with 50 seeds that randomize the default search order [12]. For each fault, we show box plots for the randomized basic and randomized CAPP heuristic explorations, comparing the distributions of transitions/schedules that are explored to detect the fault. Each box plot shows the median, upper and lower quartile values, the max and min values not outside the 1.5 times inter-quartile range from their quartile values, and the outliers outside that range. Note that each y-axis is normalized such that the 100 mark is the median of the randomized basic exploration, and all

Table 4: Randomized search order results for JPF

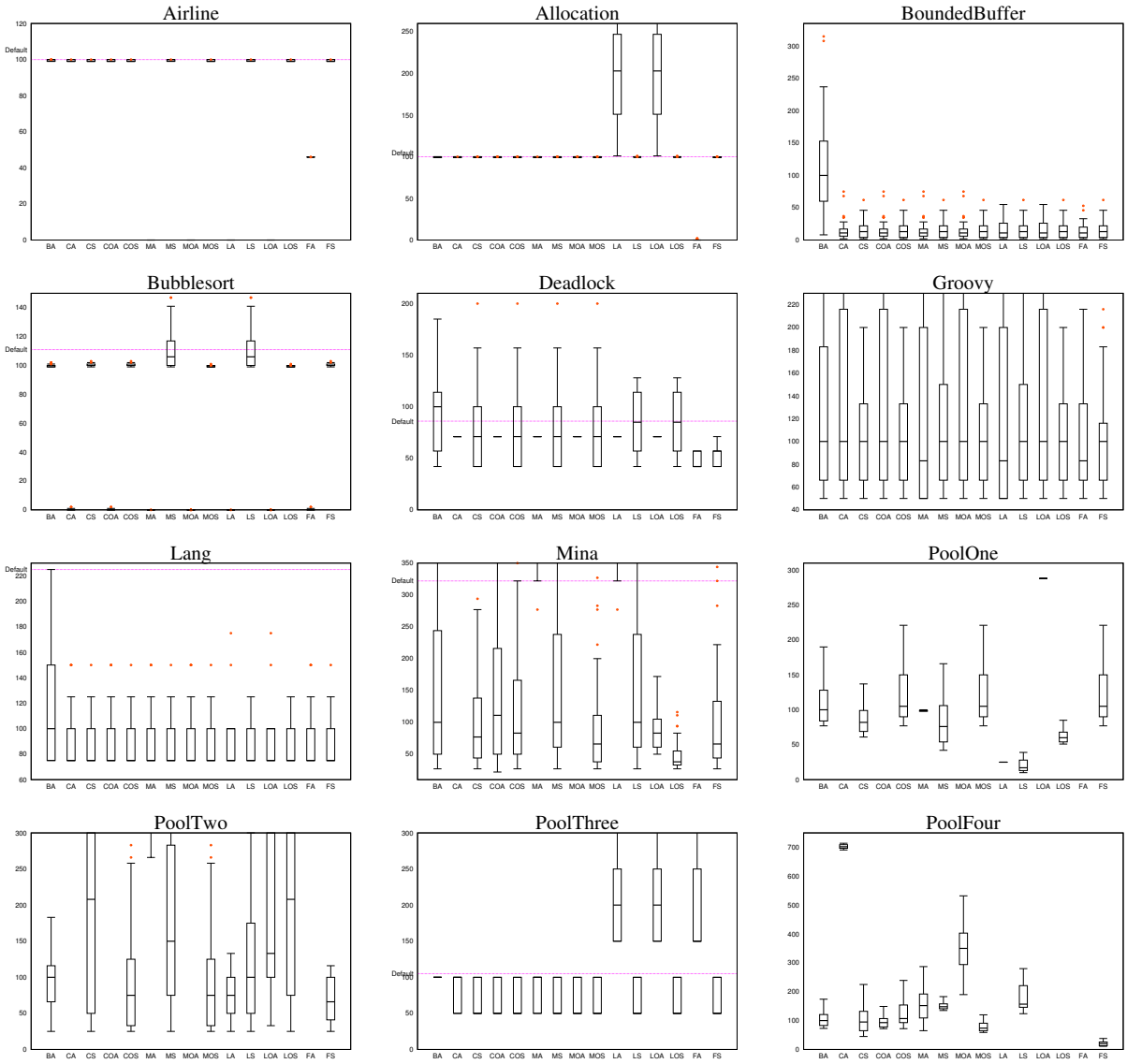


other values are divided by that median. We also include a reference line (`Default`) that shows the number of transitions/schedules that the basic exploration with the (non-randomized) default search order explored to detect the fault (i.e., the values from the second column of tables 2 and 3); when the line is missing, it is above the limit. Following the methodology recommended by Arcuri and Briand [6], we performed pairwise Mann-Whitney U tests and computed Vargha and Delaney's $\hat{A}_{12}$ non-parametric effect size measure to compare the random search order result distributions for each of the heuristics with basic. Tables 6 and 7 show the computed p-values and $\hat{A}_{12}$ values.

4.6.1 *RQ4: Default vs Random Search Order*

A heuristic can be considered to perform better than basic with high confidence if its p-value is less than 0.05 and its

Table 5: Randomized search order results for ReEx



$\hat{A}_{12}$ is greater than 0.5. As noticed with the default search order, majority of the heuristics for both stateful JPF and stateless ReEx exploration do perform better than basic. The best heuristics for JPF (MA) and ReEx (LOA) with the default search order, continue to be the best for random search order. Grouping heuristics by the ICP match mode, `METHOD_ON_STACK` and `LINE_ON_STACK` heuristics perform well for JPF, and `LINE` heuristics perform well for ReEx, which is contrary to the default search order results. Grouping heuristics by prioritization mode, `ALL` heuristics continue to do better than `SOME` heuristics..

The results for random search order confirm all the results for default search order (e.g., in terms of best heuristics and best prioritization modes), except that the ICP match modes that are the best for default search order are not the best for random search order.

4.7 Threats to Validity

Internal threats We conducted our experiments using the default settings of JPF (version 5.0pre2) and ReEx systematic exploration frameworks. During the course of our study, we found two bugs in JPF, which we have corrected on our local copy. To the best of our knowledge, there are no other bugs in the frameworks that would affect our results. However, changing the settings could affect the results.

External threats The artifacts that we perform our experiments on were collected from a variety of sources and are diverse in terms of the statistics that we show in Table 1. The artifacts that we collected from SIR [35] have all been used in previous experiments, and the other artifacts are from widely used open-source projects. However, we cannot guarantee that they form a representative sample of multi-threaded Java programs. To mitigate the limitation of using one particular exploration framework and search order, we evaluated CAPP with two different exploration frameworks,

Table 6: Summary of randomized search order results for JPF

	CA	CS	COA	COS	MA	MS	MOA	MOS	LA	LS	LOA	LOS	FA	FS
p-value	0.0661	0.0655	0.0283	0.0607	0.0001	0.2181	0.0001	0.0161	0.0001	0.0703	0.0001	0.0351	0.0046	0.0007
A_{12}	0.5306	0.5307	0.5365	0.5313	0.6411	0.5205	0.6082	0.5401	0.6299	0.5302	0.6109	0.5351	0.5471	0.5563

Table 7: Summary of randomized search order results for ReEx

	CA	CS	COA	COS	MA	MS	MOA	MOS	LA	LS	LOA	LOS	FA	FS
p-value	0.0249	0.5653	0.0267	0.1095	0.0054	0.0947	0.0025	0.6015	0.0012	0.0012	0.0001	0.0001	0.4769	0.1770
A_{12}	0.5346	0.4911	0.5342	0.4752	0.5430	0.5258	0.5467	0.4920	0.5499	0.4499	0.5594	0.4296	0.4890	0.4792

with both default and randomized search orders, and with both DFS and BFS for JPF.

Construct threats In our evaluation, we use the number of transitions (for JPF) and the number of schedules (for ReEx) as the measures for exploration cost instead of the real execution time. The reason for this was three fold. First, we performed our experiments across multiple computers with various hardware configurations, hence measuring real execution time across computers would not be a robust measure. Second, previous related studies [12, 13, 38] also use abstract, system-independent measures, such as the number of new states, which correlate with real time. Third, our CAPP prototypes do not optimize for speed as our goal was to evaluate the algorithms before focusing on the implementation.

Conclusion threats The number of random explorations (50) that we performed for each artifact and heuristic may not be sufficient to accurately characterize real distribution of the random explorations.

5. RELATED WORK

Many techniques have been developed for improving regression testing of sequential code. Test selection [17, 37] techniques choose to run only a subset of tests on the new program version. The key challenge is to perform safe selection [28], i.e., guarantee that tests that are not selected will not reveal faults. Test prioritization [14, 20, 33, 40] reorders (all or only selected) tests to reveal faults faster, thus reducing the time that a developer has to wait to find failing tests. Impact analysis [26, 27, 33] finds (statically or dynamically) which code changes could affect which tests, thus aiding test selection or debugging by pointing out which changes could (not) lead to failing tests. These techniques work well for selecting/prioritizing *among* sequential tests, which are typically short running. However, multithreaded tests are typically long running because they are explored for many different schedules. Hence, when a regression test suite contains multithreaded tests, selecting/prioritizing schedules *within* one test becomes an issue. While the exploration of multiple tests can be easily parallelized, efficiently parallelizing a single exploration is very challenging (e.g., witness many years of the PDMC workshop series). Our work prioritizes the exploration of schedules for a single multithreaded test to reduce the exploration required to detect a fault.

There is also a rich body of work on testing multithreaded code but mostly with *change-unaware* techniques [8, 10, 15, 21, 25]. Most of these techniques conceptually prioritize (or select) schedules to be explored such that faults are found faster (or exploration finishes faster if there are no faults), but the prioritization does not consider code changes. We believe that most of these techniques can be modified to be *change-aware* and that it would provide faster exploration when code evolves, but in this paper we focused on using CHESS [25] as an example stateless technique and JPF as an example stateful technique (with its underlying partial-order reductions). The original work on CHESS [25] showed that a large number of concurrent faults can be detected with a small number of preemptions, often up to two. The follow-up work on preemption sealing [7] employs a new scheduling strategy that only allows preemptions around certain program modules to enable modular testing, e.g., to speed up testing of applications that use reliable libraries.

We know of only a few *change-aware* (also called incremental) techniques for systematic testing. The initial work focused on control-intensive properties in model checking [9, 18,24,31], conceptually reusing results from one run to speed up the next run. Our ISSE technique [23] also reuses results from one run to another but for data-intensive properties of sequential, non-deterministic code. The most recent and most related projects are on regression model checking (RMC) [38] and improved mutation testing of multithreaded code [16]. Both projects reuse exploration of one program version to speed up the exploration of the next program version. However, these projects effectively focus on *selection* and attempt to only explore the states that behave differently after a change has been made. Our work differs in that (1) CAPP focuses on *prioritization*, (2) CAPP does not reuse the results from a previous exploration but only exploits changes and their impact, and (3) our evaluation includes real faults and not only mutants. It is likely that exploiting previous exploration can improve CAPP even further, and we plan to explore that in the future.

6. CONCLUSIONS AND FUTURE WORK

Multithreaded code is becoming mainstream, and testing it is gaining importance. Since multithreaded code can display different behaviors with different schedules, testing techniques have to explore many possible schedules to detect faults. Numerous techniques have been proposed for testing multithreaded code, but most of them focus on a single version of the program and are change-unaware. We presented Change-Aware Preemption Prioritization, the first technique that uses change information to prioritize the exploration of a multithreaded regression test to substantially reduce the exploration cost required to detect faults.

Encouraged by these results, we envision several lines of future work for change-aware testing of multithreaded code. CAPP itself can be improved by using an advanced impact analysis to collect ICPs more precisely. Moreover, CAPP performs *only prioritization* and *only across schedules within one test*. The future work should consider a *combination of prioritization and selection*, and it should consider techniques for *prioritization (and selection) across multiple multithreaded tests*. Also, our current experiments consider only

one fault per code. The future techniques should be evaluated in the presence of *multiple faults*.

Acknowledgements

We thank Milos Gligoric and Adrian Nistor for extensive discussions about this work, and Nicholas Chen, Ralph Johnson, and anonymous reviewers for comments on previous versions of this paper. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-1012759, CNS-0958199, CCF-0916893, and CCF-0746856, by Intel and Microsoft under the Universal Parallel Computing Research Center (UPCRC), and by Samsung under an award from SAIT.

7. REFERENCES

[1] Apache Software Foundation. DIRMINA-803. `https://issues.apache.org/jira/browse/DIRMINA-803`.

[2] Apache Software Foundation. LANG-481. `https://issues.apache.org/jira/browse/LANG-481`.

[3] Apache Software Foundation. POOL-107. `https://issues.apache.org/jira/browse/POOL-107`.

[4] Apache Software Foundation. POOL-120. `https://issues.apache.org/jira/browse/POOL-120`.

[5] Apache Software Foundation. POOL-146. `https://issues.apache.org/jira/browse/POOL-146`.

[6] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *To appear ICSE*, 2011.

[7] T. Ball, S. Burckhardt, K. Coons, M. Musuvathi, and S. Qadeer. Preemption sealing for efficient concurrency testing. In *TACAS*, 2010.

[8] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-Up: A complete and automatic linearizability checker. In *PLDI*, 2010.

[9] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *CAV*, 2005.

[10] K. Coons, S. Burckhardt, and M. Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *PPoPP*, 2010.

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

[12] M. B. Dwyer, S. G. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *ICSE*, 2007.

[13] M. B. Dwyer, S. Person, and S. G. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *FSE*, 2006.

[14] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE TSE*, 2002.

[15] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

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

[17] M. J. Harrold, J. A. Jones, T. Li, D. Liang, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA*, 2001.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, 2003.

[19] H.-Y. Hsu and A. Orso. MINTS: A general framework and tool for supporting test-suite minimization. In *ICSE*, 2009.

[20] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *ICSM*, 2001.

[21] P. Joshi, M. Naik, and K. Sen. An effective dynamic analysis for detecting generalized deadlocks. In *FSE*, 2010.

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

[23] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *ICSE*, 2008.

[24] J. Makowsky and E. Rawe. Incremental model checking for fixed point properties on decomposable structures. In *MFCS*, 1995.

[25] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[26] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE*, 2003.

[27] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *OOPSLA*, 2004.

[28] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 1997.

[29] A. Rountev. Precise identification of side-effect-free methods in Java. In *ICSM*, 2004.

[30] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, 2004.

[31] O. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *CAV*, 1994.

[32] SpringSource and Groovy Community. GROOVY-1890. `http://jira.codehaus.org/browse/GROOVY-1890`.

[33] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA*, 2002.

[34] The Eclipse Foundation. Eclipse JDT UI. `http://www.eclipse.org/jdt/ui/`.

[35] University of Nebraska Lincoln. Software-artifact Infrastructure Repository. `http://sir.unl.edu`.

[36] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *J-ASE*, 2003.

[37] G. H. Xu and A. Rountev. Regression test selection for AspectJ software. In *ICSE*, 2007.

[38] G. Yang, M. B. Dwyer, and G. Rothermel. Regression model checking. In *ICSM*, 2009.

[39] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 2010.

[40] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei. Time-aware test-case prioritization using integer linear programming. In *ISSTA*, 2009.