Noname manuscript No.
(will be inserted by the editor)

Location Pairs: A Test Coverage Metric for Shared-Memory Concurrent Programs

the date of receipt and acceptance should be inserted later

Abstract We present a coverage metric targeted at shared memory concurrent programs: the Location Pairs (LP) coverage metric. The goals of this metric are (i) to measure how thoroughly a program has been tested from a concurrency standpoint, i.e., whether enough qualitatively different thread interleavings have been explored, and (ii) to guide testing towards unexplored concurrency scenarios. This metric was inspired by an access pattern known to lead to high-level concurrency errors in industrial software and in the literature. We built a monitoring tool to measure LP coverage of test programs. We used the LP metric for interactive debugging, and compared LP coverage with other concurrency coverage metrics on Java benchmarks. We demonstrated that LP coverage corresponds better to concurrency errors, is a better measure of how well a program is exercised concurrency-wise by a test set, reaches saturation later than other coverage metrics, and is viable and useful as an interactive testing and debugging tool.

Keywords Concurrency, testing, coverage metrics, multi-threaded software, shared memory

1 Introduction

As concurrent software becomes more common, errors due to concurrency are becoming a central issue in software development. Concurrency errors can be very difficult to detect and may have serious consequences. While successful static analysis tools have been developed targeting particular concurrency errors such as race conditions, for most industrial-scale programs, testing and runtime verification of properties remain the prevalent methods for validation.

Since by testing and runtime verification it is possible to explore only a small subset of possible executions, it is important to guide tests towards interesting scenarios, and to provide a measure of test adequacy. Test coverage metrics address both of these issues. By defining a coverage metric, one identifies noteworthy features and behaviors of the program that tests aim to exercise. Coverage gaps then provide a means by

Address(es) of author(s) should be given

which to improve the quality of testing. Addressing coverage gaps guides test generation towards scenarios and behaviors that may have been overlooked during testing. The creation and examination of the additional scenarios increases the chance of finding bugs which are difficult to trigger [BFM+05]. A coverage metric also provides a test adequacy measure. When complete coverage is achieved, this is an indication that, by one measure, adequate testing has been performed. When the coverage obtained by a certain test generation method saturates, this is indication that the current approach to testing is unlikely to uncover further bugs that the kind of coverage metric corresponds to [SDE09]. Coverage measurement can also be used to pinpoint what important execution feature a test case examines and to reduce the cost of testing by identifying redundant test scenarios in regression tests [ZHM97]. In all of these ways, properly-chosen coverage metrics can provide the link between evaluating test results and test generation.

Coverage metrics have been serving these purposes well for sequential programs. Structural (code) coverage metrics such as statement (line) coverage [ZHM97], branch and path coverage [FW93] and metrics such as mutation adequacy which measures how well the testing program detects artificially inserted errors [ZHM97], have been useful guides and measures for testing. However, for concurrent programs, the exploration of suitable coverage metrics is in its early stages. We discuss some of the pioneering existing work on concurrent coverage metrics in Section 6. In this paper, we present and investigate a novel concurrent coverage metric: the location-pairs (LP) coverage metric. This metric is distinguished by the fact that it directly targets error-prone access patterns likely to lead to a particular kind of concurrency error.

In this work, we introduce a coverage metric which targets concurrency errors and serves as a measure of how well interesting distinct thread interleavings have been explored. The location-pairs (LP) coverage metric can be roughly described as follows. Let l_1 and l_2 be two control locations in a shared memory multithreaded program with the property that l_1 and l_2 (i) can access the same shared variable v, (ii) can be executed by two different threads, with no other access to v in between. Then, (l_1, l_2) is a location pair, and the LP coverage metric requires that all coverable location pairs be exercised by some program execution.

Intuitively, each different location pair points to a qualitatively distinct thread interleaving. Many high-level concurrency error patterns studied in the literature [ETQ05, KTE06, FNU03], such as atomicity violation patterns [WS06], are expressed in terms of pairs of shared variable accesses. Based on this intuition, the location pairs coverage metric aims to serve as a first-degree proxy for how well distinct thread interleavings have been explored. In this regard, the LP metric appears similar to the access-pair [SDE09] or concurrent definition-use pairs [YSP98] metrics. However, as explained in the motivation section (Section 2), illustrated on examples from the literature in Section 2.4, and demonstrated in our experiments, the LP metric targets not only qualitatively different interleavings, but ones that are likely to be unintended, error-prone but possible, and likely to lead to atomicity violations.

We implemented a dynamic monitoring tool by modifying the Java PathFinder (JPF) [VHB+03] model checker to measure the LP coverage of testing programs. The tool operates on Java bytecode and runs the Java program on a custom virtual machine written in Java. Before the dynamic monitoring begins, we run static analysis on the Java bytecode and build an approximate set of coverable pairs to serve as the testing coverage target. Then, we monitor program executions in our coverage tool and record covered location pairs. Our tool allows control of thread scheduling which allows us

to pick a particular schedule, or to randomize thread schedules in order to get more distinct interleavings on the same test inputs. We ran a number of experiments on concurrent Java benchmarks and demonstrated the following about the LP coverage metric:

1. The LP metric is a useful guide for devising interesting, untested scenarios that may lead to high-level concurrency errors.
2. The LP metric has good bug detection ability. It corresponds better to concurrency errors than the all definition-use pairs for concurrent programs, and the method pairs coverage metric for concurrent programs.
3. The LP metric can be used with large-scale programs. The coverage target is reasonably-sized, and its analysis is viable.

Section 2 gives an overview of correctness criteria for and testing of concurrent programs, and provides the motivation for our work. The LP metric is defined and the coverage monitoring algorithm is presented in Section 3. The coverage measurement tool is described in Section 4. Case studies and experimental results are presented in Section 5. Previous work on coverage metrics for concurrent programs is discussed in Section 6. Section 7 presents future research directions and concludes the paper.

2 Motivation

The goal of this section is to provide intuitive justification for the LP metric and its formulation. Section 2.1 describes the kind of concurrency errors and corresponding correctness criteria targeted by LP coverage. Section 2.2 presents a code example from an earlier version of the Java class libraries on which we illustrate the LP metric. In Section 2.3, we contrast the LP metric with other coverage metrics, including the concurrent all definitions-uses metric, on the code example, and motivate the design choices underlying the LP metric. In Section 2.4, on a set of benchmarks derived from the concurrency literature, we demonstrate how the LP metric directly corresponds to the atomicity and refinement violations that we target, and how other, similar coverage metrics fail to do so.

2.1 Concurrency Bugs and Correctness Criteria

Testing concurrent programs poses additional challenges compared to testing sequential programs. A sequential program produces the same results when given the same input. For concurrent programs, the non-determinism in the scheduling of threads may lead to different results for tests with the same inputs. Typically, the set of possible thread interleavings for concurrent programs is very large and testing all interleavings is often impossible [Tre99]. In most cases, running the same test suite many times may not help exercise different thread interleavings because most runs may follow the same thread schedule and rare but error-prone scenarios may be missed during testing [HSU03]. Several tools try to address this problem by randomizing, or exploring in a particular order the thread interleavings [MQB+08, BAEFU06, NBU07].

There are a number of types of bugs that often occur in concurrent programs. For instance, a *data race* occurs when a variable is accessed by more than one thread, with at least one of the accesses being a write, and there being no happens-before

```
public synchronized StringBuffer append(StringBuffer sb) {

1       int len = sb.length();
2       int newcount = count + len;
3       if (newcount > value.length)
4          expandCapacity(newCount);
5       sb.getChars(0, len, value, count);
6       count = newcount;
7       return this;
8     }
```

Fig. 1 StringBuffer append Method

relationship between the accesses [MPA05]. Data races can cause program crashes and data corruption. As a basic correctness criterion for concurrent programs, race-freedom ensures a total order on operations on data variables and sequential consistency. However, race-freedom is a low-level criterion, and is not sufficient in most of cases for the correct execution of the program. More frequently, race conditions are symptoms of higher-level, logical design errors.

A higher-level correctness criterion for concurrent programs is *atomicity*. A set of code blocks are said to be atomic if the resulting program state is such that the blocks appear to have been executed without interference from other concurrently-executing threads [FF04]. An *atomicity violation* is said to occur in an execution when a block that the programmer intended to be atomic experiences interference from other concurrent threads, and no longer appears to have executed sequentially. Atomicity violations frequently lead to crashes and data corruption. Deciding on the extents of atomic blocks is a key design concern for concurrent programs. In particular, erring in the opposite direction and making large code blocks atomic may lead to unacceptable performance degradation and can even prevent the program from functioning correctly [KTE06] as will be discussed in Section 5.4 on a case study.

For some programs, e.g., concurrently-accessed data structures, atomicity is too stringent a condition for expressing the desired correctness criterion. Frequently-used correctness criteria for concurrently-accessed data structures, *linearizability* and *refinement* require that the results of the operations on the data structure are such that the operations appear to execute as if they were performed in a linear order. These criteria allow finer-level concurrency and higher performance for concurrent data structures.

The LP coverage metric introduced in this paper mainly targets high-level concurrency errors such as atomicity and refinement violations, and aims the exploration of executions likely to contain such violations.

2.2 Motivating Example

In this section, we introduce an example on which we motivate the LP coverage metric. Figure 1 presents the `append` method of an older version of the `StringBuffer` class from the Java class library. In this implementation, the `append` method has a concurrency error, because, even though the method itself is synchronized, its argument, `sb`, the `StringBuffer` to be appended to `this` is not locked throughout the method [FF04].

During the execution of an `append` method by a thread t_1 the content of `sb` can be modified by another thread t_2. For example, after t_1 invokes `sb.length()` at line 1

```
   Thread t1                                  Thread t2
running o.append(StringBuffer sb)    running sb.setlength(0);
---------------------------------    ------------------------

int len = sb.length();
                                     // char array of sb's contents
                                     // sb.value = new char[0];
                                     sb.count = 0;
int newcount = count + len;
if (newcount > value.length)
  expandCapacity(newCount);
sb.getChars(0, len, value, count);
// sb.value gets indexed past out
// of bounds
count = newcount;
return this;
```

Fig. 2 Thread interleaving for the `StringBuffer` bug.

of the `append` method and saves the length value to local variable `len`, t_2 may call the `setLength(int newLength)` method of `sb` with a `newLength` value of 0, as shown in Fig. 2. When t_1 resumes execution and invokes `sb.getChars(0, len, value, count)`, an array out of bounds exception will be thrown because the value of `len` read in line 1 of `append` is no longer accurate. The array of characters storing the contents of `sb` has been replaced with a character array of size 0. This error is an atomicity violation. The operation of appending `sb` to `this` is not performed atomically by t_1 due to interference by t_2. Note that the array index out of bounds exception is merely one manifestation of the atomicity violation. The LP metric is not targeted at a particular kind of Java exception, but at atomicity and refinement errors. Such errors can have serious consequences that are hard to detect and to debug [ETQ05].

Line 1 of `append` performs a read access by thread t_1 to the field `sb.count`, while, within the `setLength()` method, a write access is performed by thread t_2 to `sb.count`. When these accesses are performed in this order with no other intervening access to `sb.count`, this atomicity violation occurs. The location pair consisting of this read-write pair exactly captures this atomicity violation. Write-after-read orderings are typically not targeted by coverage metrics, because they do not capture intended execution scenarios. Differently from existing metrics, the LP metric targets exactly these unintended, unlikely but possible and error-prone interleavings. In the next section, we will show that other metrics, even concurrency-coverage metrics such as the concurrent definition-use pair metric, fail to capture this error.

2.3 The LP metric vs Other Metrics

It is well known that different adequacy criteria are required for the testing of concurrent programs. Sequential code coverage metrics such as line coverage or branch coverage are, as is to be expected, not good measures of coverage of thread interleavings, as our experimental results demonstrate. Let us illustrate this point on the motivating example presented in the previous section. In this example if the `sb.setLength()` method is executed without interruption, followed by an execution of `append` by another thread, without interruption, 100 % line coverage of these methods is achieved. Similarly, 100

```
private static void CpToCache(byte[] buf, CacheEntry te, int lsn, Handle h) {

1    for(int i = 0; i < buf.Length; i++) {
2      te.data[i] = buf[i];
3    }
4    te.lsn = lsn;
5 }

public static void Flush(int lsn) {
...

1 lock(clean) {
2    BoxMain.alloc.Write(h, te.data, te.data.Length, 0, 0, WRITE_TYPE.RAW);
3 }
...
```

Fig. 3 Buggy code fragment from an earlier version of the Boxwood `Cache` module

```
1 public synchronized void addElement(Object obj) {
2    modCount++;
3    ensureCapacityHelper(elementCount + 1);
4    elementData[elementCount] = obj;
5    elementCount++;
6 }

1 public int lastIndexOf(Object elem) {
2    int count = Read(elementCount) - 1;
3    return lastIndexOf(elem, count);
4 }
```

Fig. 4 Code fragments illustrating the error in `java.util.Vector`

% branch coverage of these methods can be achieved if all branches in control flow of the program are followed still executing each method without interruption. Hovewer, since the interleaving that creates the bug explained in the motivating example does not occur when two methods are executed sequentially, these scenarios do not trigger the bug, and sequential coverage metrics are not appropriate for targeting concurrency errors.

More interestingly, the concurrent all definition-use pairs metric (discussed in detail in Section 3.2) does not capture this atomicity violation well. The interleaving of threads and methods described above accomplishes 100% concurrent DU coverage. The interleaving that leads to the error has the following ordering of accesses to `sb.count`:

$$\text{use by } t_1(\text{action } \alpha_1) \rightarrow \text{definition by } t_2(\text{action } \alpha_2) \rightarrow \text{use by } t_1(\text{action } \alpha_3)$$

If one runs first t_2's execution of `sb.setLength()` in its entirety, followed by an execution of `append(sb)`, the concurrent DU pair α_2, α_3 is still covered although the bug is not triggered in this case. This is because the DU metric when applied to the definition α_2 and the use α_3, does not refer to other use accesses between α_2 and α_3, as long as there is no other intervening definition. Furthermore, the DU metric stipulates nothing about the use α_1 followed by the definition α_2, which is exactly the ordering of actions that causes the atomicity violation.

2.4 Evidence for The LP Metric: Examples from the Literature

This section describes how the LP coverage metric captures concurrency errors from the literature and errors in industrial examples we studied in earlier work. It is noteworthy that each error scenario described directly corresponds to the coverage of a particular location pair. A test program first has to bring the program to a particular, perhaps unlikely state in order for it to be possible to exercise the LP. The LP metric succinctly expresses the concurrency error as a coverage goal, which leads the test writer to think of the program state that may make the LP possible.

The Cache Module of the Boxwood Storage Manager: The error, explained in more detail in [ETQ05], involves a cache block in a data structure that is flushed to the next level of the storage hierarchy while it is being overwritten by another thread. This corresponds to the location pair given by line 2 in the `CpToCache` method in Fig. 3 being executed right after line 2 in the `Flush` method, representing flush midway through the copying of buffer `buf` to the cache. In this example, exercising this single location pair guarantees that the bug will be triggered.

The "Scan" file system: The error in this system, as documented in [TBJ04], is very similar in spirit to the scenario in the Boxwood cache. While the file system cache is being written to the disk, after a block gets copied to disk and gets marked "clean", it gets overwritten by another file system thread. Exercising a particular location pair guarantees that this error will be triggered.

`java.util.Vector:` The code for this example is modified to contain one atomic action per line for illustration purposes (see Fig. 4). If line 3 in `lastIndexOf` is followed by line 5 in `addElement`, covering this location pair leads to an atomicity violation, which was previously discovered by [FQ03]. In particular, if `elem == obj`, this would have led to an incorrect return value for `lastIndexOf`, which is a refinement violation [ETQ05]. Thus, in this example, 100% LP coverage guarantees the detection of the concurrency bug.

The concurrency error categories in [FNU03] The LP metric can express as a coverage goal all error-prone scenarios that are described in this work. The errors in the category "Code Assumed to Be Protected" of [FNU03] are particularly relevant for atomicity and refinement violations. For all these bug categories, 100% LP coverage guarantees that the bugs will be triggered.

Our investigation of the LP metric was inspired by the fact that the LP metric guaranteed the detection of concurrency errors from a wide variety of sources in the literature.

3 The Location Pairs Coverage Metric

3.1 Concurrent Programs: Syntax and Semantics

We present a mathematical model of concurrent Java programs and their executions at the bytecode level sufficient to formalize our coverage metric.

Programs. Consider a Java source program $\mathcal{P}$, and a bytecode program $\mathbf{P}$ obtained by compiling $\mathcal{P}$. For coverage purposes, we are interested in the classes, methods and program locations of $\mathbf{P}$. We therefore formally represent a program as $\mathbf{P} = \langle C, M, L \rangle$.

- C represents the classes in $\mathbf{P}$. A class is represented by a tuple $c = (F_c, S_c, M_c)$ where F_c, and S_c are the sets of instance and static fields of the class c, and M_c is the set of c's methods.
- M represents the set of methods of all of the classes in $\mathbf{P}$.
- L denotes the set of program control locations of $\mathbf{P}$. Each *control location* (or, simply "location") of $\mathbf{P}$ corresponds one-to-one to a bytecode instruction in $\mathbf{P}$. Each location l is associated with a particular line of source code in $\mathcal{P}$, denoted by $sr(l)$.

Executions Executions of $\mathbf{P}$ are represented as triples consisting of the sequence of actions performed during the execution, and the set of threads and variables involved in the execution. Formally, $\psi = \langle \pi, T, V \rangle$, where π is a sequence of actions performed by ψ, T is the set of threads created during the execution of ψ, and V represents the variables created and accessed during the execution ψ. Let O_ψ be the set of objects instances of C created during execution ψ. *cls(o)* $= c$ if the object o is type of class c.

In order to define our coverage metric, we will need to refer to shared variables, i.e., variables accessed by more than one thread. Variables can be instance fields, static fields, or array elements. An instance field variable is a pair *(o,f)* where o is an object and f is a field of *cls(o)*. A static field variable, *(c,s)*, consists of a class c and a static field s of f. Array elements are referred to using the format *a[i]*, where a is an array and i is an index into array a. Observe that, the term "shared variable" refers to a particular address used during program execution, and not to a variable in the program text.

For the purposes of this paper, *actions* can be of three categories: (i) read accesses: *read* (v, t, l), (ii) write accesses: *write* (v, t, l), and (iii) other actions *other*. A `read` of a variable is a triple $\sigma =$ *read* (v, t, l) where v is the variable accessed by thread t when the bytecode instruction at location l is executed. A `write` action $\sigma =$ *write* (v, t, l) is defined similarly. Other actions are not interesting for coverage purposes, thus, we represent all of them with the single action *other*.

In the rest of the paper, we use the following shorthand. For a read or write action σ, σ.`accVar` denotes the variable accessed, σ.`accType` denotes the action type (either `read` or `write`), and σ.`accThr`,and σ.`accLoc` denote the executing thread and the control location of this action, respectively. We say that two actions are *dependent* if they access the same variable and at least one of them is a write action. The location-pairs coverage metric makes no reference to the values written and read, thus, our execution model does not include these.

When using the LP coverage metric, if a race condition is detected, we stop execution and declare that a concurrency error is caught as described in [EQT10]. As a result, in all executions, there is a happens-before relation between any two accesses to the same shared variable, where at least one of them is a write [MPA05]. The Java memory model guarantees that race-free executions are sequentially consistent. Therefore, for each such execution ψ, there is a total order on the actions of ψ that is consistent with the happens-before relation. We assume that the execution is described using this total order, i.e., that π is a total order that respects the happens-before relationship and $\pi(i)$ denotes the i-th action in this order. For purposes of defining and measuring the metric, any such total order π can be used.

3.2 Defining and Measuring the Location Pairs Coverage Metric

Our metric is built around the concept of a *location pair* (l_1, l_2), where $l_1 \in L$ and $l_2 \in L$. Roughly speaking, we require that all such location pairs that can perform conflicting accesses to the same shared variable (at least one of the accesses is a write) be exercised one after the other by different threads, with no other accesses to the shared variable in between the execution of l_1 and l_2.

In order to facilitate the definition of the metric, we first define the *happened just before* relation ($\xrightarrow{jhb}$) on a given execution ψ. Given $\psi = \langle \pi, T, V \rangle$, and two indices i and j, we say action $\pi(i)$ happened just before action $\pi(j)$ (denoted $\pi(i) \xrightarrow{jhb} \pi(j)$ iff the following hold:

- $i < j$,
- $\pi(i).\texttt{accVar} = \pi(j).\texttt{accVar} = v$, i.e., the actions $\pi(i)$ and $\pi(j)$ access the same (shared) variable v, and
- $\pi(i)$ and $\pi(j)$ are two consecutive accesses to v in ψ, i.e., there does not exist an index k such that $i < k < j$, and $\pi(k).\texttt{accVar} = v$.

A location pair (l_1, l_2) is said to be *exercised* or *covered* by an execution $\psi = \langle \pi, T, V \rangle$ if there exist indices i and j such that the following are satisfied.

- $\pi(i)$ happened just before $\pi(j)$ in ψ.
- $\pi(i).\texttt{accLoc} = l_1$, and $\pi(j).\texttt{accLoc} = l_2$.
- $\pi(i)$ and $\pi(j)$ are conflicting accesses, i.e., they access the same variable and at least one is a write,
- $\pi(i)$ and $\pi(j)$ are executed by different threads, i.e., $\pi(i).\texttt{accThr} \neq \pi(j).\texttt{accThr}$,

A location pair (l_1, l_2) is said to be *coverable in* $\mathbf{P}$ if there exists an execution of $\mathbf{P}$ in which (l_1, l_2) is covered. Our test adequacy criterion requires that all coverable pairs be covered by a test execution. Note that if $\mathbf{P}$ contains two locations l_1 and l_2 that may perform conflicting accesses to the same shared variable, then the metric requires that both of the pairs (l_1, l_2) and (l_2, l_1) be covered, if feasible.

The definition of the LP metric is similar to the concurrent all definitions-uses (DU) metric and the all access-pairs metric from the literature. A definition-use pair is a location pair (l_1, l_2) where l_1 is a write access and l_2 is a read access. Differently from covering a location pair, a definition-use pair is said to be covered by an execution if $\psi = \langle \pi, T, V \rangle$ if there exist indices i and j such that

- $\pi(i).\texttt{accLoc} = l_1$, and $\pi(j).\texttt{accLoc} = l_2$.
- $\pi(i)$ and $\pi(j)$ are executed by different threads, i.e., $\pi(i).\texttt{accThr} \neq \pi(j).\texttt{accThr}$,
- $\pi(i).\texttt{accVar} = \pi(j).\texttt{accVar} = v$, i.e., the actions $\pi(i)$ and $\pi(j)$ access the same (shared) variable v, and
- there is no other definition of v between $\pi(i)$ and $\pi(j)$. In other words, the read $pi(j)$ sees the write performed by $\pi(i)$.

The access-pair metric defined in [SDE09] requires the following to be covered:

- each read of a shared variable preceding each write of the shared variable
- each pair of write accesses to a shared variable with no intervening reads, and
- each write to a shared variable preceding a read of the shared variable with no intervening writes

The LP metric requires that use-definition and definition-definition pairs be covered, and has the more stringent happened just before requirement between the access pairs. The justification and consequences of this design choice are discussed in Sections 2.3 and 5.1, respectively.

Another concurrent coverage metric we compare the LP metric with is the method pairs (MP) metric. The MP metric considers a pair of methods (m_i, m_j) covered by an execution, if, during the execution, between the call and return actions of an instance of m_i by a thread t_1 at least one action of method m_j is executed by another thread, t_2.

We will now illustrate the location-pairs coverage metric on our motivating example. Let the read access to `sb.count` via the method call in line 1 of the `append` method correspond to control location l_1, while the read access to `sb.count` on line 5 of the `append` method correspond to control location l_2, and the write access to `sb.count` of `sb`'s `setLength(int newLength)` method correspond to control location l_3. Since l_1 and l_3 may access the same variable and l_3 executes a write on the variable, our metric requires that, in at least one execution, l_3 be executed after l_1 by a different thread, without any other locations accessing `sb` between. This sequence leads to the bug that was explained in Section 2.2.

The set of location pairs covered by an execution ψ is denoted by $Cov(\psi)$. When a set of test executions $\mathcal{T} = \{\psi_0, \psi_1, ..., \psi_n\}$ on a program $\mathbf{P}$ are obtained, the set of location pairs covered by the test set $\mathcal{T}$ is denoted by $Cov(\mathcal{T})$ and is the union of the sets of location pairs covered by each execution, i.e., $\cup_{0 \leq i \leq n} Cov(\psi_i)$. The set $UP = SP - Cov(\mathcal{T})$ is the set of uncovered location pairs, and constitutes coverage targets for further testing.

3.3 Constructing the Set of Coverable Location Pairs

For a coverage metric to be useful in practice, complete coverage must be a practically attainable target. Not only must complete coverage be precisely defined, but there must exist an efficient way to for the set of coverage targets or a reasonable overapproximation to be computed. The size of this set must also be reasonable, since every coverage gap will be the target of test generation effort. In this section, we first define the set of coverable location pairs, and then present our method for computing an overapproximation for this set. In Section 5.3, we present results on benchmarks that show that the overapproximate set is of tolerable size.

We denote by CP the set of coverable location pairs. We now illustrate how a location pair may not be coverable, despite the fact that the locations may perform conflicting accesses to a shared variable. Consider a Java class representing a bank account. Two synchronized methods, `deposit` and `withdraw` each read the value of the `balance` field into a thread-local variable, modify the thread-local variable, and write the updated value of the thread-local variable back to the `balance` field. Let the reads of the `balance` field in `deposit` and `withdraw` correspond to locations $l_{rd,dp}$ and $l_{rd,wd}$, and the writes to the `balance` field in `deposit` and `withdraw` correspond to locations $l_{wr,dp}$ and $l_{wr,wd}$. While it is possible for all of these locations to refer to the same shared variable (the `balance` field of the same `Account` object), it is not possible for the location pair $(l_{rd,wd}, l_{wr,dp})$ to be covered. This is because, once one of the synchronized methods starts execution and performs a read access, the very next

access to the `balance` field must be a write by the same method. Thus, $(l_{rd,wd}, l_{wr,dp})$ is not in the coverable pairs set CP.

Since the reachability of code locations is already an undecidable problem, computing the set CP exactly is undecidable as well. To approximate the set of coverable pairs, we use static analysis. We call the overapproximation to CP computed by this analysis the *statically-detected pairs set* SP, where $CP \subseteq SP$.

Clearly, a more effective static analysis builds a more precise SP set that is a better approximation of CP. The intended use of the SP set is as follows. The testing effort starts with SP as the coverage target. After a certain amount of testing, when uncovered pairs from SP are reported to the user, the user manually removes location pairs that he determines by inspection to be uncoverable. The user iteratively builds a set, denoted SP_{red}, which is a pruned version of SP and is still an overapproximation of CP. Thus, we have $CP \subseteq SP_{red} \subseteq SP$.

To compute SP, we adapt the static race detection tool, Chord [NAW06]. Chord analyzes Java bytecode to identify location pairs that may result in data races. The computation carried out by Chord consists of four phases: (i) reachable-pairs computation, (ii) aliasing-pairs computation, (iii) escaping-pairs computation, and (iv) unlocked-pairs computation [NAW06]. Since the focus of this paper is the usefulness of the location-pairs coverage metric for uncovering concurrency bugs, we only describe the computation carried out by Chord at a high level, and refer the interested reader to [NAW06] for more specifics.

The reachable-pairs computation phase identifies control locations containing data accesses which are reachable from a thread-spawning call site without switching threads afterwards. Any pair of control locations reachable in this way is included in the reachable-pairs set. The aliasing-pairs computation phase narrows this set by overapproximating the subset of reachable location pairs that may both refer to the same memory location (referred to as shared variables in our formalism). The escaping-pairs computation phase further narrows the aliasing-pairs set by performing an escape analysis and identifying thread-shared variables. The last phase of Chord, the unlocked-pairs analysis phase, discards location pairs which are guaranteed to be protected by the same lock.

To compute the SP set, we use phases (i)-(iii) of Chord. We do not use phase (iv) to eliminate lock-protected access pairs, since the location-pairs coverage metric aims to identify interesting interleavings and not data races. Given a location pair (l_1, l_2), even if all executions that cover this pair have both accesses protected by the same lock, for thread interleavings to be adequately explored, we still require test executions to exercise (l_1, l_2). For instance, if an object has synchronized setter and getter methods for a field f, corresponding to locations l_{set} and l_{get}, we require that the location pairs (l_{set}, l_{get}) and (l_{get}, l_{set}) be covered, since these point to different interesting interleavings.

While manually refining SP to arrive at SP_{red}, we use locked-pairs information to identify uncoverable pairs as illustrated by the `Account` example above, and will be illustrated further in Section 5.4 on a real bug from the Apache FTP server bug repository. It is possible to provide some automation for this process, at least for certain non-coverable LP pair patterns. However, we have not done so in this study, because our focus was to investigate whether the LP coverage corresponds well to concurrency errors. For the benchmarks on which we performed error detection experiments, the size of the SP set was reasonable and we were able to eliminate uncoverable pairs from SP with reasonable effort.

4 The Implementation

In this section, we explain how we implemented the LP coverage metric tool.

4.1 Static Analysis

The first step in the implementation is the determination and parsing in of the statically-computed coverable pairs set SP into the coverage tool. As explained in Section 3.3, on a given Java program, we run the first three phases of the Chord static race detection tool in order to obtain an approximation for the set of coverable pairs. The output of the third phase of Chord is a list of pairs of accesses that may access the same shared variable while being executed by two different threads. This output is parsed and read into the coverage tool in order to construct the set SP. For each such pair (ω_1, ω_2), the locations may access a shared variable (an instance or static field of a particular class or an array element) denoted $\omega_1.e = \omega_2.e$ and either $\omega_1.type$ or $\omega_2.type$ is a `write`. For each pair (ω_1, ω_2) parsed from the output of Chord, the location pairs $(\omega_1.l, \omega_2.l)$ and $(\omega_1.l, \omega_2.l)$ are included in the set SP.

4.2 Dynamic Monitoring

After constructing the SP set, we run the coverage measurement algorithm. We implemented this algorithm within the the dynamic analysis and model checking tool Java PathFinder (JPF, [VHB+03]) using the Java programming language. This implementation choice trades off coverage monitoring efficiency for more control over executions and thread interleavings. We chose this approach, because the goal of this study was to demonstrate that the location-pairs coverage metric corresponds well to concurrency errors and how well a program is exercised by a test set from a concurrency standpoint. It is possible to implement coverage measurement in much more efficient ways by, for instance, bytecode instrumentation.

Given a concurrent Java program and its inputs, JPF explores thread interleavings and continuously monitors the program state to signal specification violations. JPF is a model checker implemented as a Java virtual machine that is itself written in Java. The user has control over the thread scheduling policy. JPF has stateful and stateless modes. In the stateful mode, signatures of program states visited are cached in order to prevent repeated exploration of those states. In the stateless mode, only the sequence of states that has led from the program's initial state to the currently explored state are stored. We used JPF in stateless mode. For some programs, even storing a sequence of states required too much memory. In these cases, we modified the JPF code and turned off all state storage in JPF (including the path to the current state) and simply used it as an instrumented runtime with control over the scheduling policy. In this way, we were able to handle large programs with reasonable time and memory overhead.

We implemented the coverage measurement algorithm as a JPF Virtual Machine Listener, `VMListener`. The pseudocode is shown in Figure 5. VMListener is an interface in JPF which is used to observe and react to the actions taken by JVM. We implement the `instructionExecuted(JVM vm)` method in this interface. This method is invoked by JPF for each Virtual Machine Listener object after each executed bytecode instruction. Whenever the instruction just executed is a variable access, we create a new action

Coverage Measurement(ψ)

```
1  instructionExecuted(JVM vm){
2
3  instr: the instruction executed last by vm
4  ti: the thread that executed the instruction instr
5  l: the control location of executed
6
7  if instr is an access to memory
8     then
9          v = the variable that instr reads or modifies
10         type= type of instr (∈ {read, write})
11         σ = type(v, ti, l)
12         if Q_v does not exist
13            then
14                 create Q_v
15         Q_v.enqueue(σ_c)
16         if Q_v.size = 2
17            then
18                 σ_1 = Q_v.head
19                 σ_2 = Q_v.tail
20                 if (σ_1.type = write or σ_2.type = write) and
                      (σ_1.t ≠ σ_2.t)
21                    then
22                       Cov(ψ).add(σ_1.l,σ_2.l)
23 }
```

Fig. 5 Measuring Coverage

object σ. This object's fields contain the variable that is accessed, v, the type of the access (read or write), information about the thread performing the access, *ti*, and the control location involved in the access, l. All of this information is made available by the JPF runtime. In order to keep track of the happened just before relation between two actions, we use a FIFO queue of size two for each variable, i.e., instance field, static field or array element. For each such variable μ, the queue Q_μ contains the last two accesses to μ in the execution prefix examined so far. The last access to μ and the second-to-last access are thus Q_μ.head and Q_μ.tail, respectively. The queue becomes full when it contains two accesses. If a new access σ is enqueued into Q_μ while the queue is full, Q_μ.head is dequeued, Q_μ.tail becomes the new head, and the new action becomes the tail of the queue. Thus, if there are two elements in the queue, there is a just happened before relation between these two actions. If at least one of the actions is of type write and the actions in the FIFO have been executed by different threads, the location pair $(\sigma_1.l, \sigma_2.l)$ consisting of the control locations associated with Q_μ.head and Q_μ.tail is recorded as a covered pair.

We investigated two thread scheduling policies implemented in our modified JPF VM. The *normal scheduling* mode behaves the way an ordinary JVM would, and does not switch between threads unnecessarily. In the *random scheduling* mode, we force the scheduler to pick at random an enabled thread at each point where the JPF makes a scheduling decision. Intuitively, the random scheduling mode is more likely to exercise concurrency-related errors.

As referred to above, after a set of executions $\mathcal{T} = \{\psi_0, \psi_1, ..., \psi_n\}$ of a program **P** are explored, the set $UP = SP - \cup_{0 \le i \le n} Cov(\psi_i)$ is reported as the set of not-yet-

covered location pairs. Observe that different executions in $\mathcal{T}$ may have been obtained by running **P** with the same inputs, but using a different thread schedule.

5 Experimental Results

In this section we present our empirical evaluation of the LP coverage metric. This section is organized as follows:

- In Section 5.1, we explore how well the LP coverage metric corresponds to concurrency errors, especially atomicity and refinement violations. We compare the LP metric with the concurrent DU and MP metrics in this regard.
- In Section 5.2, following the work of [SDE09], we study the saturation behavior of the LP metric and compare it with the DU and line coverage metrics on a number of Java benchmarks from the literature. We study whether the LP metric reaches saturation later than other metrics, but still within a reasonable amount of time.
- In Section 5.3, we present results on a large set of Java benchmarks that indicate that both the coverage target and its overapproximation are of manageable size.
- In Section 5.4, on a code example extracted from an earlier version of the Apache FTP server, we illustrate how the LP coverage metric helps a programmer interactively detect and fix concurrency errors.

5.1 Concurrency Bug Detection Ability: Comparison with Other Metrics

In this section, we investigate how well the LP metric corresponds to concurrency errors. We compare the LP metric to the DU and MP metrics in this regard. For this purpose, we investigated buggy versions of two benchmarks: the `Multiset` benchmark [ETQ05], and the `Elevator` benchmark from the Parallel Java (PJ) benchmark suite. `Multiset` is a concurrent multiset data structure implementation whose operations mimic those of more complicated concurrent data structures such as concurrently-accessed storage and file systems. `Multiset` provides insert, delete, look up, and insert-pair operations (Figures 6, 8 and 7). The operations provided by `Multiset` are intended to implement a specification with atomic methods, therefore, we believe that `Multiset` is an appropriate benchmark on which to evaluate LP metric. `Elevator` is a concurrent Java program that simulates a multi-elevator control system.

We generated buggy versions of `Multiset` and `Elevator` in the following two ways:

1. **Mutation Operators:** We applied the concurrent program mutation operators for concurrent Java programs described in Bradbury et al. [BCD06]. The mutation operators applicable to the programs we studied were:
   - SHCR: Shrink Critical Region
   - EXCR: Expand Critical Region
   - SPCR: Split Critical Region
   - MSP: Modify Synchronized Block Parameter
   - RSB: Remove Synchronized Block
   - ESP: Exchange Synchronized Block Parameter

   We call a generated mutant "trivial" if it does not contain a concurrency error, or the error is very easily and quickly detected by random simulation.

Buggy benchmarks	Success of metric as adequacy measure (%)			% passes that cover a new MP/DU/LP that also detect bug			% passes that detect bug that cover a new MP/DU/LP			Avg. covg. when bug detected			Avg. # passes
	MP	DU	LP	MP	DU	LP	MP	DU	LP	MP	DU	LP	
MS Mutant 6	1	6	100	0	0.8	18.5	0	3	100	99.8	99.5	89.7	362
MS Mutant 9	2	7	100	0.2	0.3	17.6	1	1	99	99.1	98.5	83.1	223
MS Mutant 14	100	100	100	0	1.5	6.1	0	12	100	90.3	88.7	56.1	743
MS Mutant 15	100	100	100	0.4	24.7	14.7	3	100	100	84.0	70.6	59.7	237
MS + Seeded Error 1	31	29	100	26.6	27.0	60.0	30.3	26.4	100	98.2	97.7	85.1	5
MS + Seeded Error 2	7	100	100	0.6	22.3	14.1	5	100	100	99.0	98.4	78.1	223
MS + Seeded Error 3	2	22	100	0.5	1.4	13.9	2	6	100	99.4	95.4	74.7	208
MS + Seeded Error 4	100	100	100	0.3	1.2	12.9	2	6	100	83.8	70.1	40.4	566
EL Mutant 1	100	100	100	9.7	5.9	17.4	51	21	100	58.6	58.6	47.9	566

Table 1 Correlation between concurrency bugs and the MP, DU and LP metrics. MS: Multiset. EL: Elevator.

2. **Error Seeding:** We manually insert concurrency errors that will lead to atomicity violations. We select the errors so that the interleavings that lead to the atomicity violation are not easy to come up with after a few minutes of inspection or by a small amount of testing. For the `Multiset` example, the errors seeded consisted of moving reads of the `valid` or `element` fields outside of the critical regions. For the `Elevator` example, the errors seeded were the splitting of the critical regions of the `getUpPeople` and the `getDownPeople` of the `Controls` class, and a re-ordering of code lines that did not have data dependencies. In all cases, the read accesses ordered before the critical sections allowed multiple threads to enter the critical section one after the other, without validating the critical section entry condition within the critical section.

For `Multiset`, we generated 29 mutants using the mutation operators listed above. After a small amount of random testing, we found 25 of these mutants to be trivial, mostly due to easily-detected race conditions. We also generated four versions of the `Multiset` benchmark with manually-seeded concurrency errors. This resulted in a total of eight versions of `Multiset` with non-trivial concurrency errors. The results for these erroneous versions of `Multiset` are given in Table 1 and will be explained below. Similarly, for the `Elevator` program, we generated two versions with manually-seeded concurrency errors. It should be noted that generating versions of concurrent programs with non-trivial atomicity violations is a labor-intensive process.

For each different buggy version of Multiset, a small test program consisting of 4-5 threads, performing 1-2 operations each was manually constructed. The size of the array holding the contents of the multiset was kept small (one or two) in order to increase the likelihood of contention over array slots. Bugs in versions of Multiset manifest themselves as inconsistent end states for the multiset representation, or sets of method return values that would be impossible in the original, correct program. These are captured as assertions inserted into the program. The read accesses performed by assertions are not taken into account for coverage measurements.

We call the process of running a buggy version (using the multi-threaded test program written for it) once, using randomized thread scheduling, until the test program terminates, a *pass*. For each buggy version, we performed passes until the bug was detected. We call the process of performing passes of the same test program until the bug is detected one *iteration*. Throughout each iteration, LP, DU, MP and line coverage were measured. A metric was considered *not successful as a measure of test adequacy* for a given iteration if at the point 100% coverage for the metric had been reached when the bug had not yet been detected during that iteration. The metric was considered *successful as a measure of test adequacy* for the iteration otherwise, i.e., the metric did not indicate that adequate testing was performed until the bug was detected. For each benchmark, we ran 100 iterations, and computed the percentage of iterations for which a metric was successful as a measure of test adequacy. The comparison of the DU, MP and LP metrics can be found in the first group of three columns titled "Success as a measure of test adequacy" in Table 1. The results clearly indicate that the LP metric is superior to MP and DU as a measure of test adequacy for programs containing atomicity violations.

The second group of three columns in Table 1, titled "% of passes that cover a new MP/DU/LP that also detect the bug" presents experimental results intended to show the correlation between the MP, DU and LP metrics and the atomicity violations in the buggy programs studied. For instance, for LP, this percentage for each buggy

program was computed by computing the ratio of passes that cover a new LP while also detecting the bug to the total number of passes that cover a new LP. Roughly speaking, this is the likelihood that a newly-covered LP will uncover an atomicity violation bug. Experimental results indicate that this likelihood is significant for LP, and better than the likelihood for MP and DU for all buggy benchmarks.

The third group of columns in Table 1, titled "% of passes that detect the bug that also cover a new MP/DU/LP" is intended to provide another measure of correlation between coverage metrics and concurrency errors. Roughly speaking, this is the probability that bug detection is due to a new MP, DU or LP being covered. It is noteworthy that almost all passes that detect a bug also cover an LP that was previously not covered.

The last group of three columns presents the average coverage with respect to each metric when the bug was detected. These numbers also indicate that LP is a more demanding criterion that corresponds better to concurrency bugs.

```
43  public boolean InsertPair(int x, int y) {
44    int i = FindSlot(x);
45    if (i == -1) {
46       return false;}
47    int j = FindSlot(y);
48    if (j == -1) {
49       synchronized (A[i]) {
50          A[i].elt = -1;
51          return false;
52       }
53    }
54    synchronized (A[i]) {
55       synchronized (A[j]) {
56          A[i].valid = true;
57          A[j].valid = true;
58       }
59    }
60    return true;
61  }
```

Fig. 6 The `InsertPair()` method of the multiset benchmark.

5.2 Coverage Saturation Experiments

In this section, following the work of Sherman et al. [SDE09] on how coverage with respect to metrics saturates as testing progresses, we study the saturation behavior of the MP, DU and LP metrics on five benchmarks. As in this study, our experience indicates that metrics that prematurely saturate are poor measures of testing adequacy.

The five benchmarks studied in this section are SOR, Moldyn, Prim, Elevator and TSP from the Parallel Java (PJ) and the Java Grande Forum (JGF) benchmark suites. Figures 9, 10, 11, 12 and 13 present how coverage increases as a function of method calls performed in each benchmark. Random scheduling is used in all experiments. In each graph, the portion of the coverage curve where the three metrics behave most differently from each other are shown. For each metric, coverage is shown as a percentage of the

```
63 public boolean Delete(int e) {
64  for (int i = 0; i < capacity; ++i) {
65    synchronized (A[i]) {
66      if (A[i].elt == e) {
67        if(A[i].valid){
68          A[i].elt = -1;
69          A[i].valid = false;
70          return true;
71        }
72      }
73    }
74  }
75  return false;
76 }
```

Fig. 7 The `delete()` method of the multiset benchmark.

```
78 public boolean LookUp(int e) {
79  for (int i = 0; i < capacity; ++i) {
80    synchronized (A[i]) {
81      if (A[i].elt == e) {
82        if(A[i].valid){
83          return true;
84        }
85      }
86    }
87  }
88  return false;
89 }
```

Fig. 8 The `LookUp()` method of the multiset benchmark.

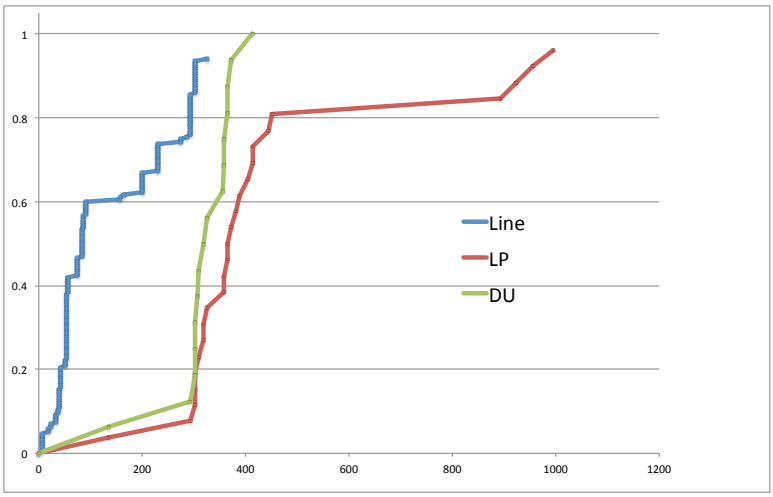

Fig. 9 Line, DU and LP coverage on SOR as a function of the number of method calls performed during testing.

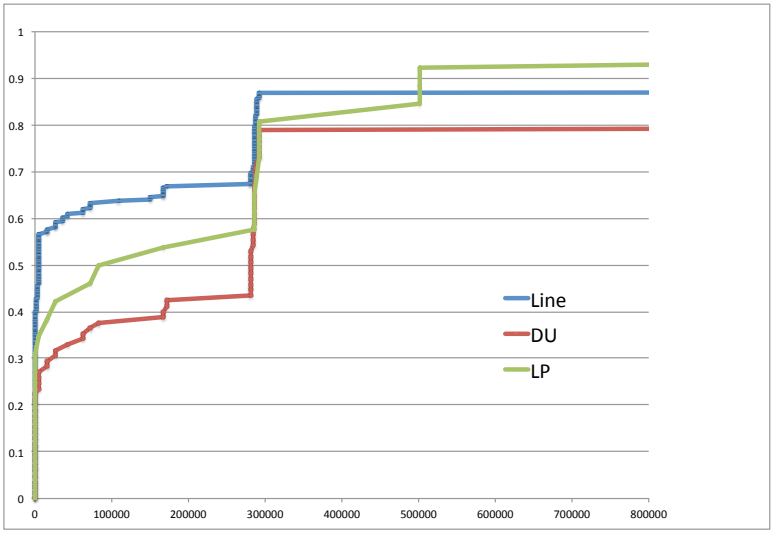

Fig. 10 Line, DU and LP coverage on Moldyn as a function of the number of method calls performed during testing.

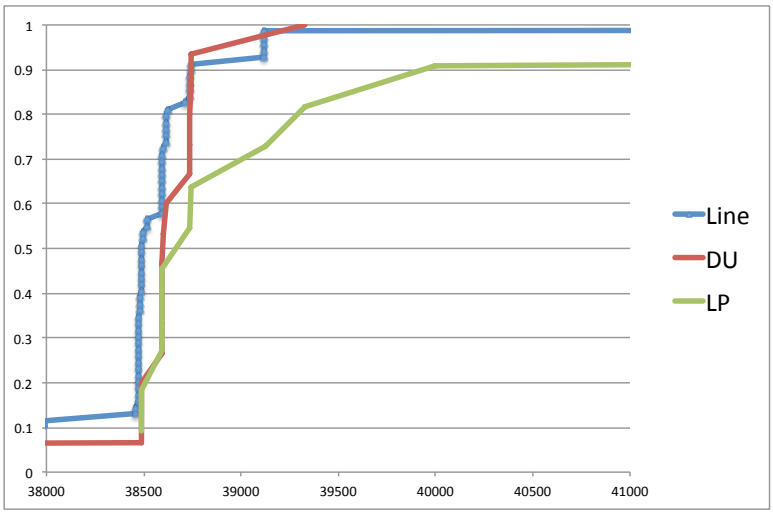

Fig. 11 Line, DU and LP coverage on Prim as a function of the number of method calls performed during testing.

maximum achieved for that metric for that experiment. Some graphs use the log scale to display the different coverage curves better.

We find that, in general, the LP metric saturates later than the DU metric, but not much later. Combined with the superior bug detection ability of the LP metric demonstrated in the previous section, this qualifies the LP metric as a practically usable metric that corresponds well to concurrency errors.

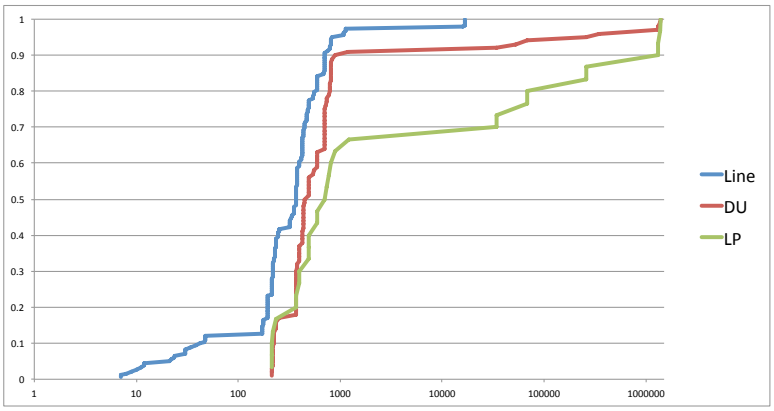

Fig. 12 Line, DU and LP coverage on Elevator as a function of the number of method calls performed during testing.

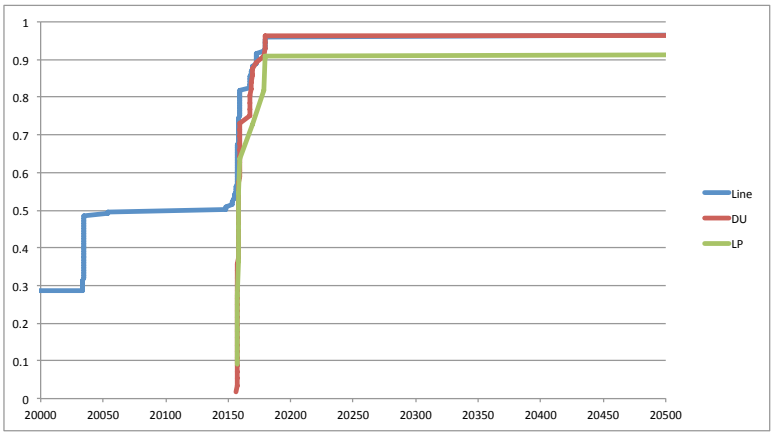

Fig. 13 Line, DU and LP coverage on TSP as a function of the number of method calls performed during testing.

For two of the benchmarks studied above, Moldyn and SOR, we continued the testing effort manually using the LP metric as a guide. Our experience is reported in the next two sections.

5.2.1 Testing Moldyn Using LP

For the moldyn benchmark, there are a total of 26 coverable pairs in the SP_{red} set. We ran the main method in the moldyn implementation by creating 2, 4, 8, 16 and 32 threads. When 100% line coverage is reached, only 34% of the pairs in the SP_{red} set had been covered for this benchmark.

For the moldyn benchmark, running the main method as is on longer tests, we reached 88% LP coverage when approximately 2.2 million methods calls had been made by 32 concurrent threads. Repeating the same tests several times did not help us reach 100% LP coverage. The size of the SP_{red} set was 26, of which 23 pairs had been

```
358    if(id == 0) {
359      for (j=0;j<3;j++) {
360        for (i=0;i<mdsize;i++) {
361           sh_force[j][i] = sh_force[j][i] * hsq2;
362        }
363      }
364    }
    ...
369     br.DoBarrier(id);
    ...
    ...
    ...
552    xvelocity = xvelocity + sh_force[0][part_id];
553    yvelocity = yvelocity + sh_force[1][part_id];
554    zvelocity = zvelocity + sh_force[2][part_id];
    ...
```

Fig. 14 A code sample from moldyn

covered. In order to cover the remaining three pairs, we examined the code closely, and picked inputs and a thread scheduling that exercised these rarely-encountered but possible interleavings.

The uncovered three pairs from SP_{red} were (l_1, l_2), (l_1, l_3) and (l_1, l_4) where l_1 is the write to `sh_force[j][i]` at line 361 and l_2, l_3 and l_4 are the reads of `sh_force[0][part_id]`, `sh_force[1][part_id]`, and `sh_force[2][part_id]` at lines 552, 553 and 554 on the code shown in Figure 14. In the code shown in Figure 14, the block between lines 358 and 364 is only accessed by a thread with id 0, because the if statement at line 358 checks the thread id and allows only thread 0 to enter this region. At line 369, there is a barrier point and all threads wait for each other to reach to that barrier point. For all runs of tests, the first thread that was allowed by the scheduler to pass this barrier point and execute the lines 552, 553, and 554 was thread 0. Since the lines were executed by the same thread, thread 0, we were not able to cover these three pairs. In order to cover these three location pairs, we paused thread 0 and let other threads to continue their runs. This hand-crafted, rare but possible thread schedule allowed us to achieve 100% LP coverage for the Moldyn benchmark.

5.2.2 Testing SOR Using LP

SOR is a benchmark from the Java Grande Forum Benchmark Suite [DHPW01]. It is a computationally-intensive benchmark that performs successive relaxation on a 2D grid. For this benchmark, the size and the elements of the SP_{red} set change with the number of threads and the size of the grid. In our experiments we used a 50×50 grid. For this grid size, when two threads are run on the benchmark, there are only 8 coverable pairs and with eight threads, there are 24 coverable pairs. The SP_{red} set constructed using Chord output has a size of 96. Thus, pruning the location pairs set using an understanding of the code for this benchmark was important, which we noticed by the small percentage of covered pairs we initially obtained. We manually constructed the SP_{red} sets for 2, 4, 8, 16 and 32 threads. The size of the SP_{red} is 24 for 4 and 32 threads, 25 for 8 threads, 16 for 16 threads and 4 for 2 threads. Maximum achievable line coverage is 97% for SOR. For all settings except the one with 8 threads, 100% LP

Program	Size of SP Set	Size of Program (Loc)
Knapsack	45	136
Eventdrivensimulation	246	2148
Prim	84	1450
Delaunayrefinement	1191	1317
Elevator	697	358
BarnesHut	467	473
PhylogenyParsBnbSmpFixed	871	6978
PhylogenyParsBnbSmp	871	6978
FindKeySmp3	384	5963
FloydSmpCol	106	5508
FloydSmpRow	106	5508
PiSmp3	114	7593
PiSmpFixed	114	7593
PiSmp	114	7593
MandelbrotSetSmp	103	5850
Tsp	587	450
JGFSORBenchSizeA	44	238
JGFSeriesBenchSizeA	14	225
JGFCryptBenchSizeA	316	358
JGFLUFactBenchSizeA	83	559
JGFSparseMatmultBenchSizeA	19	220
JGFMonteCarloBenchSizeA	55	1260
JGFRayTracerBenchSizeAFixed	549	851
JGFRayTracerBenchSizeA	555	851
HWQueue	29	127
Queue2	55	188
Queue	75	198
Spin09Set	101	236
Stack3	25	128
Stack2	21	121
Stack	34	132

Table 2 Size of the program and size of the SP set

coverage is achieved with test cases. With 8 threads, one pair remains uncovered and it can be covered by hand-crafting the thread scheduling.

From our experience with the SOR and Moldyn benchmarks, we conclude the following:

1. The LP coverage metric is helpful for improving the testing of concurrency better by indicating relevant coverage gaps. The LP metric provides a useful way to guide test generation and program inspection from a concurrency standpoint.
2. LP coverage provides a reasonably-sized coverage target even for large scientific benchmarks. For such benchmarks, exhaustive testing and model checking are not possible, and the LP metric provides a relevant measure of how adequately thread interleavings have been explored.

5.3 The Size of the Coverage Target

The purpose of the experiments in this section is to investigate whether the overapproximated coverage target expressed by SP is of reasonable size. Recall that this target is obtained by using a modified version of the static analysis tool Chord, as described

in Section 3.3. We worked on several benchmarks that were previously evaluated for research on concurrent programming tools. These benchmarks include programs from Java Grande Forum (JGF) benchmark suite, Parallel Java (PJ) Library and Lonestar benchmark suite [SBO01, Kam07, KBCP09]. Smaller benchmarks shown in the table were also used for testing tool performance before [FM08, FFY08].

The results show that the coverage target consists of at most several hundred location pairs. Moreover, the size of a program and its concurrent complexity measured in terms of the number of LP pair candidates, are not directly related. The location pairs that are not covered during random simulation require manual inspection. The set of location pairs that require manual inspection could be prioritized by considering location pairs from classes that are more prone to concurrency errors. Another possibility is the exploration of more precise static techniques that can rule out more uncoverable LP pairs. While our experience with benchmarks suggests immediate ways of improving static analysis precision, in this study, we have focused more on the bug-detection ability of the LP metric and chosen to inspect non-covered LP pairs manually.

In larger software projects consisting of hundreds of thousands of lines of code, such as projects that may be using benchmarks in Table 2 as libraries, most meaningful coverage metrics are likely to yield large coverage targets that need to be examined by prioritizing certain modules. Typically, one would concentrate on the "concurrency kernel" of such programs, and location pairs that come from these few core Java classes. The coverage targets corresponding to these concurrency kernels, as shown by the results in Table 2, are of manageable size.

5.4 A Case Study: The Apache FTP Server

In this section, we present a case study on the use of the LP coverage metric in the iterative testing and debugging of a real application.

5.4.1 Description of the Apache FTPServer and the Test Harness

The code fragments in Figures 15 and 16 are extracted from the version of the free, open source FTP server, Apache FTPServer used as a benchmark in the race-detection work of Naik et al. [NAW06]. This version of the FTPServer contains several concurrency errors, including data races and atomicity violations. To be able to run experiments on the FTPServer using our coverage tool in JPF, we created a test harness that mimics the behavior of the server. Because JPF does not handle I/O, we abstracted away the logging and networking-related parts of the server, and left only the FTP connection and request handling mechanisms of the server intact. In our model, connections can be created and they make requests to the server. The server replies to the requests using stub methods that send blank replies.

The Apache FTPServer works as follows: Upon a new connection being established between a client and the server, a thread is started in order to handle requests on this connection by using an instance of the `RequestHandler` class. Fragments from the corresponding `run()` method of this connection are shown in Figure 15. This method has a loop that listens to incoming FTP requests from the client. Each loop iteration processes one command from the client. In the body of the `run()` method, the fields `m_request`, `m_writer`, `m_reader` and `m_controlSocket` are accessed.

```
1  public void run() {
2
3   m_writer.send(220, null, null);
4   m_reader = new MReader();
5   do {
6       m_request.updateLastAccessTime();
7       String commandLine = m_reader.readLine();
8      ...
9     commandLine = commandLine.trim();
10    if(commandLine.equals("")) {
11        continue;
12    }
13     ...
14    m_request.parse(commandLine);
15    if(!hasPermission()) {
16       m_writer.send(530, "permission", null);
17       continue;
18    }
19    ...
20   service(m_request, m_writer);
21  }while(!m_isConnectionClosed);
22
23  if(!m_isConnectionClosed) {
24     conManager.closeConnection(this);
25  }
   }
```

Fig. 15 The run method of the FTPServer

```
1  public void close() {
2
3   synchronized(this) {
4       if(m_isConnectionClosed) {
5         return;
6       }
7       m_isConnectionClosed = true;
8   }
9   disconnect(m_request, m_writer);
10  ...
11  FtpRequestImpl request = m_request;
12  if(request != null) {
13      ...
14      m_request = null;
15  }
16  FtpWriter writer = m_writer;
17  if(writer != null) {
18      m_writer = null;
19  }
20  MReader reader = m_reader;
21  if(reader != null) {
22      m_reader = null;
23  }
   }
```

Fig. 16 The close method of the FTPServer

```
public void run() {
   conManager = new ConnectionManagerImpl();

   for(int i=0; i<2; i++){
        RequestHandler connection =
              new RequestHandler(conManager);
        conManager.newConnection(connection);
   }
}
```

Fig. 17 Creating two connections

The `close()` method is shown in Figure 16. The `run()` and `close()` methods are designed to be run concurrently by different threads. The `close()` method is executed by a thread different from the connection thread such as a time-out or shutdown thread. When `close()` is called, if the connection is still open, it sets the `m_isConnectionClosed` field of the current `RequestHandler` instance to `true` and sets the `m_request`, `m_writer`, `m_reader` and `m_controlSocket` fields to `null`.

In the initial test phase, we create two connections and run the server. In this setup, there are two `RequestHandler` objects which are created for each connection as shown in Figure 17. When `conManager.newConnection(connection)` is called, the `RequestHandler` object (connection) is added to the `conManager` and it starts running the `run()` method of this connection which is shown in Figure 15.

The bug scenarios reported in [NAW06] for this example exhibit themselves in the form of a `NullPointerException` being thrown by the thread executing the `run()` method. This exception occurs when the `close()` method is executed in its entirety by a time-out thread after the loop continuation condition `!m_isConnectionClosed` is checked, but before an access to the fields `m_request`, `m_writer`, `m_reader` or `m_controlSocket` by the `run()` thread takes place. The time-out thread sets these fields to null, and when the `run()` thread continues execution, a `NullPointerException` occurs.

5.4.2 Debugging the FTPServer

In this section, we will present how we iteratively debugged the Apache FTPServer by making use of the LP coverage metric. The debugging effort resulted in four versions of the program, to which we gave intuitive names below for easy reference: the *original* version, the *race-free* version, the *too coarsely-atomic* version, and the *final* version.

Arriving at the Race-Free Version Applying the static race-detection tool Chord on this example results in a number of race reports. As shown in Figure 15 and Figure 16, there are no locks protecting accesses to shared variables the in `run()` and `close()` methods and these unprotected accesses to shared fields may cause data races. As a first attempt at debugging the program, we eliminate these race conditions by wrapping every individual access to the fields `m_request`, `m_writer`, `m_reader` and `m_isConnectionClosed` in a separate code block, synchronized on `this`. This new version of the program is called the *race-free* version.

As explained in [KTE06], in this example, race conditions are symptoms of an atomicity violation and a higher-level error in the structure of the code. In this case, protecting individual accesses by the same object lock and removing the race conditions does not prevent the error scenarios described in the previous section. The thread interleavings leading to the `NullPointerException` remain possible.

Analyzing the Race-Free Version with the LP Metric: Let us now consider what information the LP coverage metric yields on this example. While analyzing the FTP server, in order to concentrate on the server code only, we assumed Java libraries to be safe from concurrency errors. While constructing the SP set, we only took into account accesses made by the FTP server. After removing the data races, we run the static analysis and the SP set is constructed for the FTPServer program. The SP set consists of pairs derived from read and write instructions on the fields `m_request`, `m_writer`, `m_reader` and `m_isConnectionClosed`. The number of pairs in the SP set is 22.

Let us name the following locations:

- *l1*: The read from the field `m_writer` in line 16 of the `run()` method (Figure 15)
- *l2*: The write to `m_writer` in line 18 of the `close()` method (Figure 16)

The location pair (*l1,l2*) is an element of SP. To cover this pair, there should be no access to the field `m_writer` between the execution of line 16 of the `run()` method by the `run()` thread and the execution of the line 18 of the `close()` method by the time-out thread. Observe that any execution that covers this pair would exercise the bug described, and result in a `NullPointerException`.

After running the tests described in the previous section 20 times using the random scheduling mode, we found that no location pairs are covered. When the number of connections in the test harness is increased to 4, 8 and 16 and the tests are repeated, still no location pairs were covered. Inspection of line coverage results show that the `close()` method is never called and only accesses in the `run()` method are exercised. In order to reason as a programmer unaware of the bug would, we inspected the code and discovered that the `close()` method is only called by a `timer` thread. The `timer` thread regularly checks the connections to determine if they timed out and closes the timed-out connections. The test harness did not exercise this time-out scenario. The only way to cover pairs of SP is to force the the time-out mechanism to close the connections. In order to accomplish this, we modified the code for the time-out thread so that the `close()` method could be called without waiting for a timer to expire. When we do this, the (*l1,l2*) pair, along with others leading to the `NullPointerException` are easily exercised.

The Fix Attempt: The Too-Coarsely-Atomic Version: The `NullPointerException` is a manifestation of a higher-level error. The processing of client commands by the `run()` method, and the closing of a connection by the `close()` method are not atomic, contrary to design intent. The first solution that comes to mind is to accomplish the desired atomicity by making the `run()` and `close()` methods `synchronized`. This fix does make the `run()` method and the `close()` method atomic. After applying this "fix", we repeat the same tests and measure the LP coverage of tests with 2, 4, 8 and 16 connections. We again face the same problem. After 20 tests for each setup, no location pairs are covered.

The problem is that making the `run()` and `close()` method `synchronized` makes it impossible for the time-out thread to run `close()` once the connection thread starts executing `run()` and acquires the lock for the `RequestHandler` object `this`! Since the thread executing `run()` never releases the lock, it is not possible for the time-out thread to set `m_isConnectionClosed` to true and interrupt the `run()` thread. In other words, the atomicity level of this version is too coarse. The programmer is made aware of this by the LP coverage report, since no location pairs have been covered. Without LP coverage, the program would have led to no race condition warnings and

```
   declare m_isConnectionClosed to be volatile
   ...
 1 public void run() {
 2  synchronized(this) {
 3      if(m_isConnectionClosed) return;
 4        else {
 5           m_writer.send(220, null, null);
 6           m_reader = new MReader();
 7        }
 8  }
 9  do {
10     synchronized(this) {
11      if(m_isConnectionClosed)
12         break;
13       else {
14         String commandLine = m_reader.readLine();
15         m_request.updateLastAccessTime();
16       }
17     }
18
19     synchronized(this) {
20        if(m_isConnectionClosed)
21          break;
22        else{
23           m_request.parse(commandLine);
24           if(!hasPermission()) {
25             m_writer.send(530,"permission",null);
26             continue;
27           }
28           ...
29           service(m_request, m_writer);
30        }
31     }
32  } while(!m_isConnectionClosed);
    ...
```

Fig. 18 The proposed fix for the bug of the FTPServer

no `NullPointerException`s, and this bug (the timeout thread can never execute) would never have been noticed! This highlights the use of the LP coverage metric in identifying important, unexercised concurrency scenarios.

5.4.3 Fixing the Bugs

In order to fix the concurrency bug while allowing the timeout thread enough concurrency to be able to close the connection thread, the programmer has to restructure the code and use atomic code blocks of the right scope. The goal is to accomplish a level of granularity finer than that provided by synchronized `run()` and `close()` methods, and coarser than protecting each individual access by a lock separately. An appropriate fix for the bug is presented in Figure 18. According to this solution, the entire body of `close()` is made synchronized since it is called only once and no other thread should be allowed to run it on the same `RequestHandler` object. But the `run()` method is implemented using more than one synchronized block, allowing the time-out thread to run in between the synchronized blocks.

After the fix is introduced, because new accesses to `m_isConnectionClosed` have been added to the code in the beginning of each `synchronized` block in `run()` as shown in Figure 18, the size of the SP set for this version of the program is larger than that for the original one. There are 28 elements in the SP set of this version of the program. By inspection, we obtain a set SP_{red} that consists of only 5 elements.

As an illustration of how the SP set is pruned, consider the following. Observe that the `close()` method is synchronized on `this` in this version. Let us label two locations in this version of the example:

- *l3*: The write to `m_reader` at line 22 of the `close()` method shown in Figure 16.
- *l4* The read to `m_reader` at line 14 of the `run()` method shown in Figure 18.

The location pair (*l3,l4*) is in the set SP, but it is not possible to cover this pair. As seen in Figure 18, the read of the `m_reader` field by `run()` is in a synchronized block, and if the `close()` method is executed before accessing this synchronized block, `m_isConnectionClosed` is set to `true`. So, when the connection thread enters in the synchronized block in the `run()` method, it will break at line 12 without executing the read to `m_reader` at line 14 of the `run()`.

Partial automation for the refinement of SP to SP_{red} could be provided, however, this is beyond the scope of this study. The purpose of this study is to demonstrate that location pairs form the basis of a useful coverage metric for concurrent programs.

Note that, for FTPServer example, it is impossible to cover all pairs in the SP_{red} set in only one run of the test. This is because one of the interfering methods is the `close()` method and we need to call this method to cover most of the pairs. We can only cover one pair on a single test because when `close()` method is called the program terminates. The five elements of the SP_{red} are covered by three different test configurations of the FTPServer.

1. The normal execution, by letting the timer to run the `close()` method at any time, covers three pairs.
2. The pair (*l5*, *l6*) $\in SP_{red}$ where *l5* is the write to `m_isConnectionClosed` at line 7 of the `close()` method (Figure 16) and *l6* is the read to `m_isConnectionClosed` at line 3 of the fixed `run()` method (Figure 18). In order to cover this pair, we need to call the `close()` method before the initialization of the connection is completed. We switch to normal scheduling mode and force the `close()` method to run before the connection thread reaches to line 2 of the fixed `run()` method. After the `close()` method is executed, the connection thread takes the execution control and executes *l6*. This execution covers the (*l5*, *l6*) pair.
3. The last uncovered pair in SP_{red} is (*l7*, *l8*) where *l7* is the write to `m_reader` at line 6 of the fixed `run()` method (Figure 18) and *l7* is the read to `m_reader` at line 20 of the `close()` method (Figure 16). This pair is covered as follows: The thread that runs the timer is set to be started and closes the connection immediately after initialization is complete. Then, we let the `run()` method to reach to line 8 of the `close()` method shown in Figure 18 and at that point we force a thread switch. The thread that closes the connection takes the control and we force it to call the `close()` method. After the `close()` method is executed (*l7*, *l8*) is covered, because the last access to `m_reader` was a write by the connection thread at line 6 of the `run()` method, and at line 20 of the `close()` method there is a read to the same variable by another thread.

At the end, we obtain 100 % LP coverage in the fixed version of the `run()` method. Observe that LP coverage has served as a proxy to obtain the interesting interleaving scenarios above. The use of the LP coverage metric on the FTPServer has illustrated that the LP coverage metric is useful for identifying interesting thread interleavings, determining when desired interleavings have not been covered by testing, and for arriving at a level of granularity for concurrency that corresponds to the designer's intent.

6 Related Work

Even though there is significant work on the adequacy of sequential testing, there is relatively little work on testing coverage for concurrent programs. Taylor et. al. proposed a coverage criterion for Ada programs in [TLK92]. Taylor's metric is composed of three different kinds of criteria: Concurrency state coverage, state transition coverage and synchronization coverage. The concurrent state coverage criterion is similar to exhaustive testing and requires traversal of the entire concurrency state space of the program. This makes this criterion an impractical measure for large-scale programs. State transition coverage has similar requirements to coverage metrics for sequential testing. It requires all states to be visited and all paths to be traversed at least once during the execution of a test program. The last criterion, synchronization coverage aims to measure the test of communication constructs. Synchronization coverage requires that, if a state is involved in a rendezvous communication, this state should be visited along any path.

Factor et. al. introduced another set of adequacy criteria for communication protocols [FFLM96]. The programs they investigate communicate through send and receive messages. The metric requires four criteria to be met: all messages should be sent and received at least once, every communication path should be traversed at least once, all corresponding send-receive pairs should be executed, and unrelated message transitions should be seen in both orders. This last criterion is similar to ours in spirit, but is not targeted at shared memory concurrent programs.

Bron et. al. introduced a similar metric for Java and C programs in [BFM$^+$05]. In their metric, called synchronization coverage, they define a code block to be blocking if a thread holds the lock for mutual exclusion for this section and prevents another thread from entering the locked section. The latter thread is called blocked. This metric requires that each coverage task be tested in blocked and blocking modes. Synchronization coverage focuses on communication of concurrency constructs. It can identify the problems associated with some patterns of concurrency errors which are caused by wrong synchronization of processes such as lost notifies [FNU03]. Synchronization coverage only refers to synchronization variables are checked whereas LP coverage involves all shared variable accesses. This way, LP coverage focuses on high-level data races and directs tests towards patterns associated with high-level concurrency errors.

Lu et al.[LJZ07] present a range of interleaving coverage criteria organized into a hierarchy. The LP metric is closest to the single-variable-interleavings (SVAR) criterion in this hierarchy, which the authors state that "is based on the observation that many concurrency bugs involve conflicting accesses to one shared variable." The SVAR criterion requires that *all* feasible interleavings of all shared accesses to any specific variable from any pair of threads are covered. The SVAR criterion is more demanding than the LP criterion, in that it requires (i) all interleavings of accesses rather than all ordered pairs of accesses, and (ii) all pairs of threads rather than any one pair of threads to

be covered. In fact, the SVAR criterion subsumes the LP criterion. The LP metric can be viewed as computationally cheaper, first-order approximation of the SVAR metric. The LP metric is also more widely applicable, since it does not require the set of all threads in a program to be known a priori. Lu et al.[LJZ07] also recognize the prohibitive computational cost of the SVAR metric, and propose a number of more computationally viable alternatives to it, based on partial interleavings. One alternative is the definition-use (DU) criterion widely studied in the literature and also studied for concurrent programs.

[SSaRY10] studies inter-thread interactions, particularly ones between layers, and dataflow techniques, such as inter-task and intra-task definition-use pairs. [YSSaR11] develops this approach further by improving observability of errors by means of "property-based oracles." The targeted errors are incorrect uses of synchronization primitives, concurrency and coordination errors between tasks. Yang et al.[YSP98] investigate the generalization of the well-known all definition-use pairs metric to concurrent programs. In this generalization, the set of threads created by the program and a parallel program flow graph describing the control and synchronization relationships of the program must be constructed a priori. This is often not possible for modern concurrent programs where threads are created dynamically, in a possibly input-dependent manner.

In concept, however, the definition-use (DU) metrics discussed above are closely related to the LP metric studied in this paper. The DU metric for concurrent programs is concerned with pairs of accesses, where the first one is a write to a shared variable x by a thread t_1 and the second one is a read of x by another thread t_2, with no other write to x in between. The DU metric allows other read accesses to x in between, however. The LP metric is more stringent than the DU metric in two regards:

- While the DU metric is concerned with the write-read (WR) access pattern, the LP metric also requires that write-write (WW) and read-write (RW) pairs to be covered. Thus, while the DU metric is more focused on more expected execution scenarios and intended interleavings, the LP metric deliberately focuses attention on the probably undesired and buggy, but possible interleavings captured by the write-write pairs and read-write pairs. As demonstrated in Section 2.3, this apparently small difference is key in uncovering atomicity violations.
- The LP metric requires that there be no intervening accesses to the shared variable other than the ones in the LP pair.

The DU metric, like the LP metric, requires static analysis for determining which DU pairs are coverable, or for overapproximating the set of DU pairs.

The idea of "value schedules" is explored by Chen and MacDonald in [CM07]. In this work as well, the smallest unit of concurrency coverage is concurrent definition-use (DU) pairs. A value schedule is an equivalence class of thread schedules that agree on the ordering of critical events, and exercise the same set of DU pairs so that the value consumed by each critical read event is the same write event in all thread schedules in the value schedule. In spirit, this work also is closely related to ours even though their target is the exploration by a model checker of fewer scenarios. Similarly to the LP vs. DU comparison, the key difference between this work and ours is that we target the unexpected but error-prone write-write and write-after-read pairs, and force the programmer to think about scenarios exercising these pairs. The idea and capability of exploring one thread schedule per equivalence class in [CM07] is very valuable and useful, and could be used to complement our approach. When we discover an unexplored

but apparently coverable LP, we can use this method to help the programmer generate the desired interleaving using model checking and JPF. Since we do not use JPF in model checking mode, however, our approach is not bound by the capacity limitations of model checkers.

Lai et al. [LCC08] define coverage metrics based on a concurrent control flow graph that represents control and synchronization relationships. This approach is well suited to embedded applications, but, in the applications we target, dynamic thread creation is very common, and the concurrent control flow graph is not known before the execution, since it is data and input-dependent. In addition to requiring that all definition-use pairs be covered, this work investigates a coverage metric which requires all inter-task edges (corresponding to points of possible context switches) to be covered. The comparison of LP with DU given above applies directly to this work. In terms of the control coverage required by this work, it targets the same idea of interesting, distinct thread interleavings that we target in our work. However, in their context, the set of thread interleaving points is, by design, severely restricted. Such an approach would not be viable for general-purpose shared memory concurrent Java programs.

Sherman et al. [SDE09], in their study of saturation-based testing using concurrency coverage metrics, point out an important gap in concurrent testing literature. While several techniques have been devised to randomize or provide controlled exploration of program thread interleavigns, "these approaches have yet to be studied from the perspective of whether they yield coverage of behaviors related to concurrency-specific faults." The work presented in our paper directly addresses that gap, and explores the connection between the LP metric, randomized exploration of thread interleavings, and a particular class of concurrency faults: atomicity and refinement violations.

Sherman et al. [SDE09] point out that a concurrent coverage criterion must try to avoid being (i) too weak, thus reaching saturation before targeted faults are detected, and (ii) too strong, thus making it difficult to compute the coverage target. They indicate that adequate testing of concurrent programs requires stronger coverage metrics, and present saturation as a method for addressing the difficulty of determining the coverage target. The idea of saturation can be used in conjunction with our work [SDE09]. While we informally use saturation in our experiments in Section 5.2, in this paper, we instead take the approach of using a powerful static analysis tool, a modified version of Chord [NAW06] to compute the coverage target. We demonstrated (see Section 5.3) that good approximations for the coverage target can be found even for strong coverage metrics, if sufficiently powerful static analysis tools are built. In this regard, the LP metric strikes a balance between strength of bug detection and feasibility of static analysis. As indicated in [SDE09], this has the additional benefit that coverage gaps can be used to guide further test generation.

The metric in [LJZ07] that is most closely related to the LP metric is the pair-interleave (PInv) metric. The PInv metric considers every consecutive access-pair for each thread, and requires that all feasible interleaving accesse to it by another thread are covered. The LP metric and the PInv metric are not comparable. The PInv metric is stated in terms of access triples (e_1^i, e_2^i, e^j) where e_1^i and e_2^i are a pair of consecutive accesses to the same variable by thred i, and e^j is an access to the same variable by another thread j. The triple is considered covered when the access e^j occurs between e_1^i and e_2^i. The two metrics are incomparable because (i) the PInv metric requires every pair of threads (i, j) to be exercised in this manner, while LP only requires any pair of threads to cover a pair of accesses, and (ii) the LP metric requires the accesses in a pair

to be the consecutive accesses to a memory location *with no intervening access* to the variable by another thread and location, while PInv does not have this requirement.

7 Conclusion and Future Work

This paper introduced a coverage metric for concurrent programs. We developed a coverage metric tool to measure how well the concurrency of the program is tested. The metric is inspired by a common pattern that leads to high-level concurrency errors. We showed that classical coverage metrics designed for sequential programs are not useful for improving testing of concurrent programs. We demonstrated the practical use of our metric and tool on large-scale programs.

While we empirically confirmed that the LP metric corresponds well to atomicity violations, we imagine that its use could be made difficult if the coverage target is too large, and it could be misled if almost all references to shared variables were made by the same few accessor methods. These are the key threats to the validity of our study. We address them further below.

7.1 Calculating the Coverage Target More Precisely

How well our initial coverage target, the statically computed location pairs set SP approximates the actual set of coverable pairs very closely depends on the static analysis used and the program analyzed. The SP sets our method constructed for the benchmarks we worked on were of reasonable size as coverage targets, and manual pruning of the SP set was a feasible approach. Even though it is an undecidable problem to construct the exact set of coverable pairs, there can be refinements on SP to decrease the number of uncoverable pairs.

7.2 Context-Sensitive Analysis of Location Pairs

The LP coverage metric is not context sensitive. Thus, executions of a particular control location via different call stacks are not distinguished from each other. In certain cases, this can be a weakness. For instance, if a shared field is only accesses by its `get()` method and modified only by its `set()` method, calling `get()` and `set()` methods anywhere in the code consecutively, in different orders is enough to cover the pairs that this field gives rise to. If there are several different calls to these methods in the code, and if only some of them lead to concurrency errors, they may be missed by the LP coverage metric. The LP coverage metric can be refined by taking into account the calling contexts.

References

[BAEFU06] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Producing scheduling that causes concurrent programs to fail. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, PADTAD '06, pages 37–40, New York, NY, USA, 2006. ACM.

[BCD06] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Mutation operators for concurrent java (j2se 5.0). In *Proceedings of the Second Workshop on Mutation Analysis*, MUTATION '06, pages 11–11, Washington, DC, USA, 2006. IEEE Computer Society.

[BFM+05] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 206–212, New York, NY, USA, 2005. ACM Press.

[CM07] Jun Chen and Steve MacDonald. Testing concurrent programs using value schedules. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 313–322, New York, NY, USA, 2007. ACM.

[DHPW01] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform independent dynamic java virtual machine analysis: the java grande forum benchmark suite. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 106–115, New York, NY, USA, 2001. ACM Press.

[EQT10] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race-aware java runtime. *Commun. ACM*, 53:85–92, November 2010.

[ETQ05] Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer. Vyrd: verifying concurrent programs by runtime refinement-violation detection. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 27–37, New York, NY, USA, 2005. ACM Press.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004.

[FFLM96] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka. Testing concurrent programs: a formal evaluation of coverage criteria. In *ICCSSE '96: Proceedings of the 7th Israeli Conference on Computer-Based Systems and Software Engineering*, page 119, Washington, DC, USA, 1996. IEEE Computer Society.

[FFY08] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 293–303, New York, NY, USA, 2008. ACM.

[FM08] Azadeh Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 52–65, Berlin, Heidelberg, 2008. Springer-Verlag.

[FNU03] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2, Washington, DC, USA, 2003. IEEE Computer Society.

[FQ03] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 1–12, New York, NY, USA, 2003. ACM.

[FW93] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, 1993.

[HSU03] Klaus Havelund, Scott D. Stoller, and Shmuel Ur. Benchmark and framework for encouraging research on multi-threaded testing tools. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.1, Washington, DC, USA, 2003. IEEE Computer Society.

[Kam07] Alan Kaminsky. Parallel java: A unified api for shared memory and cluster parallel programming in 100 *Parallel and Distributed Processing Symposium, International*, 0:231, 2007.

[KBCP09] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, pages 65–76, 2009.

[KTE06] M. Erkan Keremoglu, Serdar Tasiran, and Tayfun Elmas. A classification of concurrency bugs in java benchmarks by developer intent. In *PADTAD '06: Proceeding of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 23–26, New York, NY, USA, 2006. ACM Press.

[LCC08] Zhifeng Lai, S. C. Cheung, and W. K. Chan. Inter-context control-flow and data-flow test adequacy criteria for nesc applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 94–104, New York, NY, USA, 2008. ACM.

[LJZ07] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A study of interleaving coverage criteria. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 533–536, New York, NY, USA, 2007. ACM.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 2006. ACM Press.

[NBU07] Yarden Nir-Buchbinder and Shmuel Ur. Contest listeners: a concurrency-oriented infrastructure for java test and heal tools. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, SOQUA '07, pages 9–16, New York, NY, USA, 2007. ACM.

[SBO01] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 8–8, New York, NY, USA, 2001. ACM.

[SDE09] Elena Sherman, Matthew B. Dwyer, and Sebastian Elbaum. Saturation-based testing of concurrent programs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 53–62, New York, NY, USA, 2009. ACM.

[SSaRY10] Ahyoung Sung, Witawas Srisa-an, Gregg Rothermel, and Tingting Yu. Testing inter-layer and inter-task interactions in rtes applications. *Asia-Pacific Software Engineering Conference*, pages 260–269, 2010.

[TBJ04] Serdar Tasiran, Andrej Bogdanov, and Minwen Ji. Detecting concurrency errors in file systems by runtime refinement checking., 2004.

[TLK92] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 18(3):206–215, 1992.

[Tre99] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 46–65, London, UK, 1999. Springer-Verlag.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, 2003.

[WS06] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 137–146, New York, NY, USA, 2006. ACM.

[YSP98] Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. All-du-path coverage for parallel programs. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '98, pages 153–162, New York, NY, USA, 1998. ACM.

[YSSaR11] Tingting Yu, Ahyoung Sung, Witawas Srisa-an, and Gregg Rothermel. Using property-based oracles when testing embedded system applications. In *Proceedings of the Fourth IEEE International Conference on Software Testing*, 2011.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.