# PENELOPE: Weaving Threads to Expose Atomicity Violations [*]

Francesco Sorrentino
University of Illinois,
Urbana-Champaign
sorrent1@illinois.edu

Azadeh Farzan
University of Toronto
azadeh@cs.toronto.edu

P. Madhusudan
University of Illinois,
Urbana-Champaign
madhu@illinois.edu

## ABSTRACT

Testing concurrent programs is challenged by the *interleaving explosion problem*— the problem of exploring the large number of interleavings a program exhibits, even under a single test input. Rather than try all interleavings, we propose to test wisely: to exercise only those schedules that lead to interleavings that are typical error patterns. In particular, in this paper we select schedules that exercise patterns of interaction that correspond to atomicity violations. Given an execution of a program under a test harness, our technique is to *algorithmically* mine from the execution a small set of alternate schedules that cause atomicity violations. The program is then re-executed under these predicted atomicity-violating schedules, and verified by the test harness. The salient feature of our tool is the efficient algorithmic prediction and synthesis of alternate schedules that cover all possible atomicity violations at program locations. We implement the tool PENELOPE that realizes this testing framework and show that the monitoring, prediction, and rescheduling (with precise repro) are efficient and effective in finding bugs related to atomicity violations.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.4 [**Software Engineering**]: Software/Program Verification

## General Terms

Algorithms, Reliability, Verification

## Keywords

atomicity violation, concurrency, dynamic analysis, predictive analysis, schedule selection, testing

## 1. INTRODUCTION

Concurrency errors are notoriously hard to find, and are characterized by subtle interleaving patterns that tend to manifest in the field, while passing extensive randomized testing in development. Furthermore, they are hard to reproduce, record, and repair, primarily because the programmer faces the difficulty of considering all the possible interleavings exercised by a nondeterministic scheduler. Testing all possible interleavings, even under a single test input, is infeasible, and this *interleaving explosion* problem is one of the most important challenges that needs to be overcome to achieve effective testing of concurrent programs. With the advent of multicore hardware, concurrent and parallel software feature prominently in the future, making this an important problem to solve.

The current technology of testing concurrent programs on a test input is *stress testing*— to test the program under random schedules by creating a large number of threads and strewing the code with `sleep` commands for random time intervals, and letting the system run and re-run for days together. This methodology is highly unsystematic and does not seem to exploit the fact that most concurrency errors can be explained using a simple schedule that uses a very few threads but an intricate interleaving of them.

The focus of research in this area has hence moved to testing systematically a small subset of schedules. For instance, the CHESS tool from Microsoft expects programs to use a small number of threads, and tests all schedules that use a bounded number preemptions (unforced context-switches).

**Atomicity-violating schedules.**

The thesis of this paper is that one can do interleaving selection more effectively by only exercising those interleavings that are symptomatic of common error patterns of interactions among threads. In this paper, we focus on the common error pattern of *atomicity violations* and explore schedule selection algorithms that systematically choose and schedule those that violate atomicity.

A programmer writing a procedure (say a method of a class) often desires *uninterfered* access to certain shared data that will enable him/her to reason about the procedure locally. The programmer puts together a concurrency control mechanism to ensure this atomicity, often achieved by taking locks associated to the data accessed, or implicitly taking locks, e.g. using `synchronized` blocks in Java. This is however extremely error-prone: not acquiring all required locks for the data leads to errors, non-uniform ordering of locking can cause deadlocks, and naive ways of granular locking can inhibit concurrency, which force programmers to invent intricate ways to achieve concurrency and correctness at the same time. When the procedure interacts with another concurrent thread, inconsistent state configurations could occur leading to unexpected behaviors and errors. Such errors due to violation

[*] This work was funded partly by the Universal Parallel Computing Research Center (UPCRC) at the University of Illinois at Urbana-Champaign, sponsored by Intel Corp. and Microsoft Corp, by NSF award #0747041, and by an NSERC Discovery Grant.

*FSE-18,* November 7–11, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-60558-791-2/10/11 ...$10.00.

of atomicity are in fact, as far as we know, the most prevalent of concurrency errors: a recent study of classifying concurrency errors [12] shows that a majority of errors (69%) are atomicity violations (about 30% more are caused by *ordering violations* and together with atomicity capture almost all concurrency errors). This motivates our choice in selecting executions that violate atomicity as the criterion for choosing interleavings to execute.

Our philosophy of schedule selection is to exercise only those schedules that lead the execution to violate atomicity. Atomicity violations are defined with respect to specially marked sequential blocks of code in the program, which we expect are reasoned sequentially and locally by the programmer. In our framework, we choose these blocks automatically to be obvious syntactically delineated blocks of code, such as the methods of a class, bodies of loop iteration, etc. While our framework does allow this markup to be refined by the tester, manual markup is not necessary (and is not done in our experiments).

**The PENELOPE framework.**

The PENELOPE framework works by taking a concurrent program $P$ under a test harness $T$. The test harness $T$ calls $P$, feeding it inputs as and when required, and verifies that the output (or reactive behavior) of the program is correct. PENELOPE first executes and observes an arbitrary concurrent execution $r$ of $P$ on $T$. This execution $r$ is then *abstracted* to an abstract execution $r'$ that records only the reads and writes to shared variables, synchronization events such as the acquisition and release of locks, thread creations, and barriers, and the begin and end of the sequential blocks of code (local computation, conditional checks, etc. are suppressed). The abstracted execution $r'$ is then subject to an *algorithmic analysis* that checks whether there are alternate ways to schedule the events in $r'$ to obtain atomicity-violating schedules, where the violation involves two threads $T_1$ and $T_2$, where a block of $T_1$ gets interrupted non-trivially by a statement of $T_2$ due to an interaction involving a single variable. This algorithmic analysis is *sound and complete* in predicting atomicity violating schedules (at this level of abstraction). It also generates a set $S$ of schedules that cover all possible atomicity violations involving two program locations in every thread $T_1$ that can be interrupted non-trivially by a program statement in $T_2$.

The predicted schedules, though feasible at the abstract level of observation (involving locks, barriers, and thread creation), may not be actually feasible in the program $P$, and even if feasible, may not lead to actual errors the test harness attests to. PENELOPE hence *reschedules* $P$ under the predicted atomicity-violating schedules in $S$, and checks whether the test harness verifies these executions to be correct. PENELOPE hence has no false positives— it executes predicted schedules and reports errors only when it finds one that violates the test harness.

**Algorithms for predicting schedules that violate atomicity.**

The algorithms to select atomicity-violating schedules and to predict with high-accuracy only *feasible schedules* that respect the concurrency control mechanism in the program is non-trivial to realize efficiently (in comparison, selecting all schedules with only $k$ preemptions, as in CHESS, is algorithmically trivial). In this paper we exhibit carefully crafted algorithmic techniques that can examine a single (arbitrary) execution of a concurrent program on a test and *predict* alternate ways of scheduling the same events to cause atomicity violations. Our technique generates schedules that violate atomicity with respect to any two threads and any single variable according to the marked boundaries. The restriction to two threads and one variable is deliberate and pragmatically motivated— most atomicity errors occur due to two threads and one variable and our algorithms scale well in this case. The predicted schedules, however, may involve the scheduling of all threads, as these may be necessary to enable the violation in two threads.

The prediction phase of our technique examines the monitored run and algorithmically checks if they can be rescheduled to violate atomicity. Intuitively, an atomicity violation is characterized by three events $e_1$, $f$, and $e_2$, occurring in that order, where $e_1$ and $e_2$ are in the same block of one thread, and $f$ is in a different thread, and $e_1$ and $e_2$ are both in *conflict* with $f$. The first part of this phase consists in identifying *cut-points*, which are pairs of events $e$ and $f$, such that $e$ belongs to the same thread as $e_1$ and $e_2$, occurs *between* $e_1$ *and* $e_2$, and can be scheduled concurrently with $f$. Hence any schedule that executes $e$ and $f$ concurrently will expose the atomicity violation. Finding atomicity violation using cut-points runs in linear time using a static analysis of the execution by examining the *locksets* and *acquisition histories* [5, 10] at every point, and is adapted from our earlier work [5].

After finding the cut-points, we develop new techniques to generate alternate schedules that reach these cut-points concurrently, hence exposing atomicity violations. This is perhaps the most important technical contribution of this paper. Schedules that reach the cut-points concurrently are theoretically possible (indeed the *proof* of the cut-point generation relies on the existence of such a schedule [5, 10]). However, our algorithm synthesizes a schedule that also (heuristically) ensures maximum conformance to the original observed execution. Building schedules that adhere as much as possible to the causal order of the original observed execution is crucial to building feasible schedules— the program under test may after all have many causal orderings such as communication, barriers, and even creation of threads that need to be respected to ensure feasibility.

We also propose several heuristics to *reduce* the number of context-switches in the predicted schedules, using a combination of escape analysis and preventing context-switches between a contiguous set of read-blocks of a thread. This reduction helps controlling the overhead in the rescheduling phase, as context-switching is expensive to control (our predicted schedules can have hundreds of thousands of context-switches).

Notice that our prediction algorithms are designed on purpose *not* to completely adhere to the *happens-before* causal relation. In the predictive world, we do not wish to generate schedules that have the same partial order as the original run (atomicity is a property of the partial order, not the exact linearization; hence adhering to the exact causal order will not lead to an atomicity violating schedule).

**Implementation and evaluation.**

We have implemented the above algorithms and heuristics in a framework, called PENELOPE, that tests Java programs. PENELOPE observes an execution, and weaves the events in them in different ways to exercise atomicity violating patterns, subjecting the resulting executions against a given test harness. PENELOPE achieves the monitoring and scheduling phases using Java bytecode transformations that inserts code to monitor and orchestrate the scheduling of threads. We show that with a standard automatic marking of atomic blocks, PENELOPE accurately predicts schedules that violate atomicity, is effective in reducing the number of alternate schedules to be tested to a small fraction of all possible interleavings, and is effective in not only finding bugs but actually accurately scheduling them (with accurate playback) to expose errors detectable by the test harness.

While there have been several tools recently that aim to find errors using atomicity violations, most of them (including VELODROME [7] and our own earlier framework reported in [4]) only *monitor one observed execution* for atomicity violations. In contrast, our tool observes an execution, predicts a small set of alternate executions that violate atomicity, and runs them against the test harness to check for actual errors.

**Related Work.**

There are two main streams of work that use predictive analysis for concurrent programs that are relevant. In two papers [21, 20], Wang and Stoller study the prediction of runs that violate serializability from a single run, similar to our prediction algorithm. In recent work by us [5], we also propose a similar prediction algorithm for detecting serializability violations. The predictive algorithm of [5] is better as it works with little memory overhead, while that of Wang and Stoller keeps track of a large graph, which doesn't scale as the size of executions increases. The prediction algorithm in PENELOPE is adapted from the algorithm described in Farzan et al [5]. Note that while both these tools predict possible atomicity violations, they do not actually synthesize alternate schedules nor try to reschedule the program to expose these errors, and have a lot of false-positives. In [19], the prediction algorithm of [5] in combination with SMT solvers was used to remove false positives.

A recent work related to ours is the tool CTRIGGER [16], that has similar motivation as ours in finding and scheduling atomicity violations. CTRIGGER works by examining a few executions, finding points where atomicity violations could occur, prunes away many schedules that are infeasible due to mutually excluded blocks or ordering constraints, and tries to schedule a selected subset of the rest by scheduling those that are less likely to manifest. The algorithms for pruning are however entirely based on heuristics; in contrast, the algorithms for pruning in PENELOPE, and in particular algorithms it uses for synthesizing schedules, are accurate with respect to the nested locking present in the program. An experimental comparison was not possible, first because CTRIGGER works for C while PENELOPE works for Java, and also because CTRIGGER tool was not available for comparison.

There has also been recent work on *active randomized testing for atomicity* in the tool ATOMFUZZER [15], that uses randomization and guidance techniques that executes and "hold" threads at strategic points to try to manifest atomicity errors. We found the holding of threads at strategic points dynamically and randomly are a bit unpredictable in our experience with the tool This technique also does not have the capability of handling nested locks, and interrupts threads at wrong positions which lead it to not find errors.

There has also been work on finding atomicity violations by using a *generalized* dynamic analysis of an execution. SIDETRACK is a new tool [23] that finds atomicity violations by a generalized analysis of the observed run. Note that this technique does not examine runs that are causally different (as PENELOPE does), and hence does not do any rescheduling. Moreover, this technique only detects and reports atomicity violations for a programmer to examine, and cannot produce error traces that violate the test harness A more liberal notion of generalized dynamic analysis of a single run have also been studied in a series of papers by Chen et al [3], who use static analysis of the program to build a causal map to analyze for atomicity.

The run-time *monitoring* algorithms and tools for atomicity violations that check violations in just the observed run are well understood, and the tool by Farzan and Madhusudan reported in [4] provides the most space-efficient algorithm known for this problem, as the space overhead is bounded when the number of threads and variables are fixed (the algorithm used in VELODROME[7] also can perform sound and complete serializability detection, but the space-overhead is not bounded). These techniques work for *any number of threads and variables*. Monitoring, of course, is a much simpler problem than prediction.

AVIO [17] defines *access interleaving invariants*— certain patterns of access interactions on variables— and learns the intended access interleaving using tests, and monitors runs to find errors. A variant of dynamic two-phase locking algorithm [14] for detection of serializability violations is used in the atomicity monitoring tool developed in [22].

Apart from the related work discussed above, *atomicity violations based on serializability* have been suggested to be effective in finding concurrency bugs in many works [8, 6, 21, 20, 22]. Lipton transactions have been used to find atomicity violations in programs [11, 8, 6, 9].

CHESS[13], from Microsoft, and CONTEST from IBM, are two mature testing tools. CONTEST uses random injection of sleep and wait statements in programs to find errors. CHESS systematically exercises all executions that involve only a few context-switches, guided by the intuition that many errors manifest with a few context-switches/preemptions. However, CHESS is constrained by the number of executions it must explore, which even with two context-switches grows quadratically in the length of the executions, making it infeasible for long tests. Some of the benchmarks handled by PENELOPE in this paper will overwhelm CHESS, as there are too may interleavings it must exercise. On the other hand, CHESS has one distinct advantage over execution-based analysis techniques (including PENELOPE and CTRIGGER) in that it explores *all* executions with two context-switches and is not constrained by the events that occur in a particular observed schedule. We believe that PENELOPE and CHESS are complementary in the their coverage of the interleaving space.

There is a subtlety in our definition of serializability: only accesses to data (and *not* synchronization events like acquisition and release of locks) are considered in determining if a block in an execution is serially executed. For example, if a thread reads the field $f$ of an object twice in an atomically marked section, each time acquiring an releasing the lock on the object, and another thread accesses a different field $g$ of the same object in between these two accesses under the same object-lock, we will declare the execution as a serializable one (this is the definition we follow in our earlier papers as well [5, 4]). However, several papers, including the ones on SIDETRACK and ATOMFUZZER will declare this execution as non-serializable, because they take into account the edges caused by the lock acquisitions and releases in the two threads. Our definition is in fact the more classical definition (followed in database theory), and we believe is the more accurate and useful definition for checking for errors in concurrent programs.

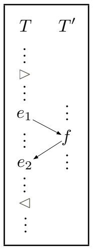
## 2. OVERVIEW

Assume that we are given a concurrent program $P$, a test input $I$, and a corresponding *set* of expected (correct) outputs $\mathcal{O}$ (we consider a set of outputs as concurrent programs can be nondeterministic). The input-output pair $(I, \mathcal{O})$ constitutes a test harness. We run the program $P$ on the input $I$, and monitor (observe) an execution $R$. In most cases, we will get an expected output $O \in \mathcal{O}$ at the end of the execution $R$. We would like to know if there is an alternative schedule that leads to an execution $R'$ of the program which manifest some concurrency bug in $P$. In other words, we would like to find an alternate execution $R'$ that generates an output $O'$ that does not belong to the set of expected outputs $\mathcal{O}$, signalling the existence of a bug. The interleaving explosion problem prevents us from trying all possible runs (as there are too many of them) to find a bad one. Here, we show how we come up with likely candidates for such runs, and how we schedule these candidate runs against the harness to check if it causes a real bug.

```
public synchronized boolean addAll(Collection c) {
    modCount++;
    int numNew = c.size();
    // ....... possible interference ....
    ensureCapacityHelper(elementCount + numNew);
    Iterator e = c.iterator();
    for (int i=0; i<numNew; i++)
        elementData[elementCount++] = e.next();
    return numNew != 0;
}
```

**Figure 1: Method `addAll` of concurrent Java class `Vector`.**

We use the following example to illustrate how PENELOPE comes up with alternative runs that violate atomicity, which can find concurrency bugs in a program. Consider the implementation of the method `addAll` from the built-in Java library class `Vector` in Figure 1. The purpose of this method is to add all elements of the parameter collection (say another vector) to the end of the current vector. The method `size`, which returns the number of elements of the source, is a `synchronized` method. The problem in this program is that after safely retrieving the number elements that are to be copied, there can be a concurrent thread that modifies the source vector before the method finishes (for example, a concurrent thread could remove all the elements from the source vector). Therefore, when the execution of the `addAll` method happens, the information about the number of the items that are being copied is stale, and an exception will be raised when it tries to access elements that are not there anymore.



Testing this program is unlikely to reveal these kinds of bugs in small methods such as `addAll`, as scenarios where the second thread gets interleaved in between the events of the first thread is unlikely. One has to actively make it happen, and that is exactly what we do.

This problematic scenario can be generalized using the following pattern of shared variable accesses:

1. Thread (1) reads the value of a shared variable (`c.size()` in the above example).
2. Thread (2) writes (modifies) the value of that same shared variable.
3. Thread (1) reads the value of the shared variable again (in `c.iterator()` in the above example).

We refer to such a pattern as a RWR pattern (Read-Write-Read). There are four more patterns that can capture other forms of undesired interference: WRW (Write-Read-Write), RWW (Read-Write-Write), WWR (Write-Write-Read), and WWW (Write-Write-Write). PENELOPE focuses on executions that contain one of these patterns as good candidates for finding hidden concurrency bugs.

PENELOPE works in three phases.

- **Phase I: Monitoring.** In this phase we execute the program on the input from the test harness and observe a run $R$. PENELOPE focusses its attention to read and write accesses to the (potentially) shared variables and acquisitions and releases of locks, thread creation, and barriers, and ignores all the other events.
- **Phase II: Prediction.** In phase II, PENELOPE constructs several runs $R'$, based on run $R$, by carefully reordering the events in run $R$ so that each of the runs $R'$ contain at least one of the suspect access patterns given above.
- **Phase III: Rescheduling.** In the last phase, PENELOPE forces the program to execute $R'$ by weaving the threads using a single processor: if this is successful, then the generated output $O'$ is checked to be in the set of expected outputs, $\mathcal{O}$; if this fails, a bug is reported. If PENELOPE fails to schedule a run $R'$, i.e. if $R'$ is not a feasible schedule of the program, PENELOPE discards it and moves to another predicted execution.

We describe in detail how the main algorithms in Phase II work in Section 4,

**Interleaving selection and comparison with CHESS.**

Our approach differs from that of the CHESS [13] tool in the rationale that is used for selecting candidate runs. CHESS limits the number of explored runs by bounding the number of context switches (actually pre-emptions) allowed in a single run. It is assumed that the number of runs with up to a few context switches is small enough that one can check all of them. This is not quite true. In a program with $n$ threads where each thread executes $k$ steps out of which at most $b$ are potentially blocking, there can be up to $\binom{nk}{(nb+c)!}$ executions when the number of context switches is limited to $c$ [13]. This can amount to a huge number of executions for very simple programs. The number of executions explored by CHESS is in fact *higher* because of the way they define *pre-emptions*, where context-switching caused by a thread encountering a *yield* statement is *not counted as a pre-emption* even though it may be possible to continue executing the thread. The number of pre-emptions is hence not polynomially bounded in $n$ and $k$, even for a fixed $c$.

Consider, for example, a program that has two threads $T_1$ and $T_2$ that manipulate a $10 \times 10$ matrix, where $T_1$ updates the first 5 rows and $T_2$ updates the latter 5 rows. Each thread sequentially iterates over the cells in its portion, and for each cell $c$ adds up the value of all the 8 neighboring cells and the cell $c$ itself, and writes the result into the cell $c$; this example is typical of particle computations and parallel algorithms in graphics. The computations of the two threads overlap at the two middle rows. Assume that there is a lock associated to each cell that is acquired before a thread accesses that cell. Since there are about 400 accesses to lock-protected data by each thread, CHESS, even with a preemption bound of 2, sets out to explore approximately $400^2$ of the interleavings, and failed to finish even after 4 hours. PENELOPE, on the other hand, explores a much smaller set of interleavings (at most one atomicity violating schedule for each of the 20 variables accessed by both threads), and finishes in under 3 minutes.

# 3. PRELIMINARIES

A concurrent shared memory program, during its computation, does local computation, may read and write to shared entities (memory locations), dynamically create threads, and use synchronization primitives (such as locks). The framework we build will observe only an abstraction of this computation that ignores local computation, ignores precise values read or written, but keeps track precisely of the read and write operations to shared memory locations as well as synchronization primitive usage. This abstract view of a computation, which we call an *execution*, will be the one that is algorithmically analyzed in order to predict and schedule alternate executions.

Recall that we assume that the program has been already automatically annotated with *atomicity transaction boundaries* to indicate rough logical sequential units of computation. This is done by PENELOPE automatically by marking obvious units such as the code for every method, etc. Note that these annotations have no semantic value– in particular, they are certainly not respected by the compiler in any way. These transactional boundaries are also part of the execution that we observe.

Hence, given an execution, each thread executes in it a series of *transactions*. A transaction is a sequence of actions; each action can be a read or a write to a (global) variable, or a synchronization action.

Let us now define the notation to talk about executions (they must satisfy the property that they contain transactions and that they respect the semantics of locks).

We assume an infinite set of thread identifiers $\mathcal{T} = \{T_1, T_2, \ldots, \}$. We also assume an infinite set of shared entity names (or just entities) $\mathcal{X} = \{x_1, x_2, \ldots\}$ that the threads can access. Let us also fix a set of global locks $\mathcal{L}$.

The set of actions that a thread $T$ can perform on a set of entities $X \subseteq \mathcal{X}$ is defined as $\Sigma_{T,X} = \{T{:}\triangleright, T{:}\triangleleft\} \cup \{T{:}\text{read}(x), T{:}\text{write}(x) \mid x \in X\} \cup \{T{:}\mathit{acquire}(l), T{:}\mathit{release}(l) \mid l \in L\}$. Actions $T{:}\text{read}(x)$ and $T{:}\text{write}(x)$ correspond to thread $T$ reading and writing to entity $x$, actions $T{:}\triangleright$ and $T{:}\triangleleft$ correspond to the beginning and the end of transaction blocks in thread $T$, and actions $T{:}\mathit{acquire}(l)$ and $T{:}\mathit{release}(l)$ correspond to acquiring and releasing the lock $l$.

Define $\Sigma_X = \bigcup_{T \in \mathcal{T}} \Sigma_{T,X}$ (actions on entities $X$ by all threads), $\Sigma_T = \bigcup_{X \subseteq \mathcal{X}} \Sigma_{T,X}$ (actions by thread $T$ on all entities), and $\Sigma = \bigcup_{X \subseteq \mathcal{X}, T \in \mathcal{T}} \Sigma_{T,X}$ (all actions).

For a word $\sigma \subseteq \Sigma^\star$, let $\sigma|_T$ be a shorthand notation for $\sigma|_{\Sigma_T}$, which includes only the actions of thread $T$ from $\sigma$.

A word $\sigma \in \Sigma^\star$ is *lock-valid* if it respects the semantics of the locking mechanism. Formally, let $\Sigma_l = \{T{:}\mathit{acquire}(l), T{:}\mathit{release}(l) \mid T \in \mathcal{T}\}$ denote the set of locking actions on a lock $l$. Then $\sigma$ is lock-valid if for every $l \in \mathcal{L}$, $\sigma|_{\Sigma_l}$ is a prefix of $\left[\bigcup_{T \in \mathcal{T}} (T{:}\mathit{acquire}(l)\ T{:}\mathit{release}(l))\right]^\star$.

Let $\mathit{Tran}_{T,\mathcal{X}} = (T{:}\triangleright) \cdot \{T{:}\text{read}(x), T{:}\text{write}(x) \mid x \in \mathcal{X}\}^\star \cdot (T{:}\triangleleft)$. A *transaction tr* of a thread $T$ is a word in $\mathit{Tran}_{T,\mathcal{X}}$. Let $\mathit{Tran}_T = (\mathit{Tran}_{T,\mathcal{X}})^\star$ denote the set of all possible sequences of transactions for a thread $T$, and let $\mathit{Tran}$ denote the set of all possible transaction sequences.

DEFINITION 3.1. *An* execution*, over a set of threads $\mathcal{T}$, entities $\mathcal{X}$, and locks $\mathcal{L}$, is a word $\sigma \in (\Sigma_{\mathcal{T},\mathcal{X}})^\star$ such that for each $T \in \mathcal{T}$, $\sigma|_T$ belongs to $\mathit{Tran}_T$, and $\sigma$ is lock-valid. Let $\mathit{Exec}_{\mathcal{T},\mathcal{X}}$ denote the set of all executions over threads $\mathcal{T}$ and entities $\mathcal{X}$.*

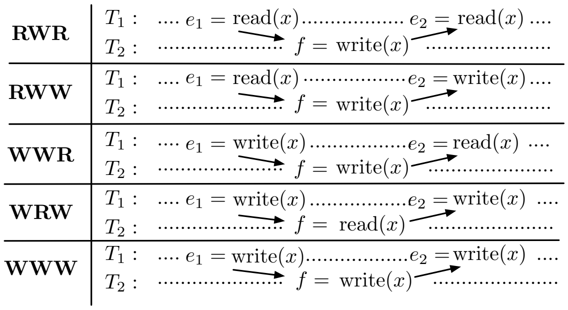
In other words, a execution is a lock-valid sequence of actions such that its projection to any thread $T$ is a word divided into a sequence of transactions, where each transaction begins with $T{:}\triangleright$, is followed by a set of reads and writes, and ends with $T{:}\triangleleft$.

When we refer to two particular actions $\sigma[i]$ and $\sigma[j]$ in $\sigma$, we say they *belong to the same transaction* if they are actions of the same thread $T$, and they are in the same transaction block in $\sigma|_T$: i.e. if there is some $T$ such that $\sigma[i], \sigma[j] \in \mathcal{A}_T$, and there is no $i'$, $i < i' < j$ such that $\sigma[i'] = T{:}\triangleleft$.

Throughout this paper, we will assume that acquisitions and releases of locks are *nested*. In other words, locks are released in the reverse order of how they were acquired. This is true in most of our examples, and in fact, using *synchronized* commands in Java naturally give nested acquisitions and releases for the most part. In any case, even if lock acquisitions and releases are not nested, our algorithms and tool will work; it is only the accuracy of predicting feasible schedules that will get mildly affected.

## 3.1 Access Patterns for Serializability Violations

There are *five* access patterns that correspond to simple atomicity (serializability) violations of *two threads* and *one variable*. Each access pattern corresponds to the existence and relative ordering of three events in an execution. A pattern consists of two events $e_1$ and $e_2$ which belong to a thread $T_1$, and a third event $f$ which belongs to thread $T_2$. These events should appear in an execution $\sigma$ such that $\sigma = \ldots e_1 \ldots f \ldots e_2 \ldots$, in other words, $f$ occurs after $e_1$, and $e_2$ occurs after $f$ in $\sigma$. Moreover, $e_1$, $f$, and $e_2$ should be all accesses to the same shared variable $x$, and should be of one formats given below. Note that these exhaust all patterns where $e_1$ and $f$ are dependent as well as $f$ and $e_2$ are dependent (two events are dependent if they correspond to two accesses to a single location, where one of them is a write).



| | | |
|---|---|---|
| **RWR** | $T_1$ : | .... $e_1 = \text{read}(x)$................. $e_2 = \text{read}(x)$ .... |
| | $T_2$ : | ..................... $f = \text{write}(x)$ ..................... |
| **RWW** | $T_1$ : | .... $e_1 = \text{read}(x)$.................$e_2 = \text{write}(x)$.... |
| | $T_2$ : | ..................... $f = \text{write}(x)$ ..................... |
| **WWR** | $T_1$ : | .... $e_1 = \text{write}(x)$................$e_2 = \text{read}(x)$ .... |
| | $T_2$ : | ..................... $f = \text{write}(x)$ ..................... |
| **WRW** | $T_1$ : | .... $e_1 = \text{write}(x)$................$e_2 = \text{write}(x)$ .... |
| | $T_2$ : | ..................... $f = \text{read}(x)$ ..................... |
| **WWW** | $T_1$ : | .... $e_1 = \text{write}(x)$................$e_2 = \text{write}(x)$ .... |
| | $T_2$ : | ..................... $f = \text{write}(x)$ ..................... |

## 3.2 Locksets and acquisition histories

Let $\sigma$ be an execution and let $\{\sigma_T\}_{T \in \mathcal{T}}$ be its set of local executions. Consider $\sigma_T$ (for any $T$). The *lockset held after* $\sigma_T$ is the set of all locks $T$ holds: $\mathit{LockSet}(\sigma_T) = \{l \in \mathcal{L} \mid \exists i.\sigma_T[i] = T{:}\mathit{acquire}(l)$ *and there is no* $j > i$ *and* $\sigma_T[j] = T{:}\mathit{release}(l)\}$.

The *acquisition history* of $\sigma_T$ records, for each lock $l$ held by $T$ at the end of $\sigma_T$, the set of locks that $T$ acquired (and possibly released) by $T$ after the last acquisition of the lock $l$. Formally, the acquisition history of $\sigma_T$, $AH(\sigma_T) : \mathit{LockSet}(\sigma_T) \rightarrow 2^{\mathcal{L}}$, where $AH(l)$ is the set of all locks $l' \in \mathcal{L}$ such that $\exists i.\sigma_T[i] = T{:}\mathit{acquire}(l)$ and there is no $j > i$ such that $\sigma_T[j] = T{:}\mathit{release}(l)$ and $\exists k > i.\sigma_T[k] = T{:}\mathit{acquire}(l')$.

Consider the program structure on the right. We have

$$\begin{aligned} \mathit{LockSet}(A) &= \{l_1\}, \\ \mathit{LockSet}(B) &= \{l_1, l_2\} \\ \mathit{LockSet}(C) &= \{l_1, l_3\}. \end{aligned}$$

Also, we have $AH(A) = \{(l_1, \{\})\}$, $AH(B) = \{(l_1, \{l_2\}), (l_2, \{\})\}$, and $AH(C) = \{(l_1, \{l_2, l_3\}), (l_3, \{\})\}$.

```
synchronize (l1) {
    // point A
    synchronize (l2) {
        // point B
    }
    synchronize (l3) {
        // point C
    }
}
```

Two acquisition histories $AH_1$ and $AH_2$ are said to be *compatible* if there are no two locks $l$ and $l'$ such that $l' \in AH_1(l)$ and $l \in AH_2(l')$.

At at the level of abstraction we observe executions (namely observing reads, writes, and lock acquisitions and releases), it turns out that we can precisely capture when two sequences of events of threads $T_1$ and $T_2$ can be combined to a lock-valid execution using just locksets and acquisition histories. More precisely, let $\sigma$ be an execution and let $w_1$ be a prefix of $\sigma|_{T_1}$ and $w_2$ be a prefix of $\sigma|_{T_2}$. Then there is an execution $\sigma'$ such that $\sigma'|_{T_1} = w_1$ and $\sigma'_{T_2} = w_2$ *if and only if* the locksets of $T_1$ and $T_2$ after $w_1$ and $w_2$ are disjoint and the acquisition histories at the end of $w_1$ and $w_2$ are compatible (this technical result is due to Kahlon et al [10]). This result will underpin our algorithms for predicting and scheduling, as it accurately captures those schedules that are feasible in a program at our level of abstraction.

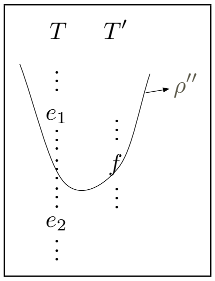
## 4. PREDICTION OF ATOMICITY-VIOLATING SCHEDULES

Phase II of our framework, which is the predictive phase, forms the crux of our algorithms and we describe it in detail below. This phase is divided into two sub-phases: Phase IIa deals with the problem of identifying cut-points of the form $(e, f)$ such that a schedule that reaches the two events in each cut-point would trigger an atomicity violation. This is followed by Phase IIb, the schedule generation phase, that actually synthesizes schedules based on the cut-points.

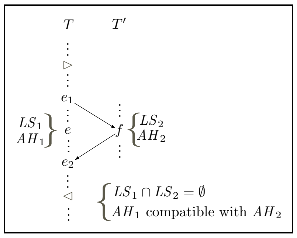
### 4.1 Phase IIa: Cut-point Generation

The first part of Phase II of PENELOPE is one that generates a set of *cut-points* for the observed execution. This algorithm essentially determines *whether* it is possible to reschedule the original execution to one that violates atomicity, and further provides witnesses in terms of a set of cut-points.

This algorithm is adapted from an algorithm from our earlier work [5] that determines cut-points accurately and extremely efficiently in linear time in the length of the execution. We define the notion of cut-points and recap the algorithm here as it is involves concepts that will help in understanding our scheduling algorithm.



Consider a typical execution that we are looking for, one where $e_1$ occurs first, then $f$, and then $e_2$, where $e_1$ and $e_2$ occur in one transaction block while $f$ occurs in another thread, and the three events form one of the patterns violating atomicity. The basic argument in our earlier work [5] is that such an abstract lock-valid execution is possible if and only if there is an intermediate event $e$ between $e_1$ and $e_2$ ($e$ can be $e_1$ but not $e_2$) such that there is an execution that reaches exactly up to executing $e$ and $f$ in the two threads. To see why, notice that we can roll back $f$ and play it last (as it is an access to a variable), and hence we can execute $e_1$ followed by $f$, and let the run proceed to execute $e_2$ eventually.



It now turns out that checking whether there is a lock-respecting run that reaches simultaneously $e$ and $f$ is easily solvable, simply by examining the locksets and acquisition histories at these two points, due to a result by Kahlon et al [10], which gives a necessary and sufficient condition for it: the locksets at $e$ and $f$ must be disjoint and the acquisition histories at $e$ and $f$ must be compatible (see Section 3.2 for definitions of locksets, acquisition histories, and compatibility).

More precisely, a cut-point is a pair of events in the execution, $(e, f)$, such that there is an alternate schedule that can reach exactly up to $e$ and $f$ simultaneously, and furthermore any schedule that does that gives an atomicity violation. The above arguments give a characterization of all cut-points that lead to atomicity violations.

The algorithm to detect and compute cut-points works as follows. We iterate over the events of each thread, gathering four sets $R$, $W$, $WW$, and $AA$, which are events along with locksets and acquisition histories. The sets $R$ and $W$ for a thread $T$ record, for every shared variable $x$, one witness pair of lockset and acquisition history among all reads and writes to $x$, respectively. The sets $WW$ and $AA$ are more complex. $WW$ records a witness for every pair of lockset and acquisition-history, an event *in between two writes* to $x$. Similarly, $AA$ records events between any two accesses (reads or writes) to $x$. Finally the sets $WW$ and $R$ are examined together to check for a compatible pair, and so are the sets $AA$ and $W$— these capture all the five possible atomicity patterns. For a fixed set of shared memory locations and locks, the time taken by the algorithm grows linearly in the length of the execution, quadratic in the number of threads (as every pair of threads must be examined), and at most quadratic in the size of the program (as all possible program locations involved in atomicity violations are identified). Furthermore, the algorithm works very well in practice, scaling to work in minutes on executions of hundreds of thousands of events.

### 4.2 Phase IIb: Schedule Generation

The scheduling of alternate executions that violate atomicity is technically and practically the hardest aspect of this paper, and we will try to explain all the ideas behind it. We first describe a *theoretically motivated algorithm* that builds a simple execution from scratch from a cut-point $(e, f)$ that causes an atomicity violation, using locksets and acquisition histories. This algorithm assumes that the program is made of $n$ concurrent threads that do not use any mode of communication, but just interact using nested locking; the algorithm is accurate in predicting *under this assumption*.

Despite its promised accuracy, it cannot itself be used in practice because the assumption is false— programs do indeed have causal ordering of events and communication (one thread could wait for another for a signal to proceed), and threads do get created and destroyed, which also gives a causal ordering to events (if $T_1$ creates $T_2$, we can't commute events of $T_2$ before the event in $T_1$ that created it). Our main idea is to modify the algorithm to make the predicted run *adhere* to the causal ordering of events in the originally observed run as much as possible. The effectiveness of this adherence is not captured in a theoretical assurance, but its proof is in the pudding, as we show that this strategy reduces infeasible executions from being predicted to a large degree in our experiments.

Finally, because of the adherence to the original execution, the predicted executions share the complexity of the original one. In particular, the predicted schedules that PENELOPE synthesizes can have *hundreds of thousands of context-switches*. The number of context-switches affects the time required to schedule the predicted execution, as context-switches take relatively more time than local executions. We provide two heuristic but technically sound transformations of the schedule that reduce the number of context-switches (the transformations do not break any causal links that may be there in the program).

We now outline the theoretical algorithm, the adherence algorithm, and the two heuristics to reduce the number of context-switches.

#### 4.2.1 The Theoretical Scheduling Algorithm

Let us, in this subsection, assume that the program consists of a static set of $n$ threads all already active (view them as $n$ sequential programs interacting with each other), and further assume that the threads do not communicate with each other explicitly using data. Let us assume the threads are $T_1, \ldots, T_n$ and further that $e$ and $f$ occur in threads $T_1$ and $T_2$, and the locksets at $e$ and $f$ are disjoint and the acquisition histories at $e$ and $f$ are compatible. Our goal is to find a locking-respecting execution that precisely executes up until $e$ and $f$ in $T_1$ and $T_2$.

First, note that, since we assumed there is no communication, executing events from $T_3, \ldots T_n$ can *never help* in the execution of

$T_1$ and $T_2$ (i.e. never enable events in $T_1$ and $T_2$); in fact, they can only hinder finding an execution as they could acquire locks that $T_1$ and $T_2$ may require. Consequently, we can completely focus only on scheduling events in threads $T_1$ and $T_2$.

The idea is that we are rescheduling up until a cut-point $(e, f)$ in threads $T_1$ and $T_2$, where $e$ falls in between $e_1$ and $e_2$ in a single thread, and where the sequence $e_1$ followed by $f$ followed by $e_2$ realizes the atomicity violation. In doing this alternate schedule, assume that a lock $l$ is held by $T_2$ at $f$, and assume that the same lock $l$ is acquired and released by thread $T_1$, before $e$. Then we must schedule this block in $T_1$ *before the last acquire of lock $l$ by $T_2$ before $f$*, as $T_1$ will have no chance to acquire it once $T_2$ has made this acquisition.

Computing the above efficiently and in linear time is non-trivial, and this is where the acquisition history helps, as it exactly captures the above information. Let $x_1$ and $x_2$ be the *last* events in $T_1$ and $T_2$, respectively, that are before $e$ and $f$, respectively, with locksets empty. Then, we can easily schedule (without violating locking) first $T_1$ up until $x_1$ and then $T_2$ up until $x_2$. Note that the first events after $x_1$ (and $x_2$) either must be $e$ ($f$) or must be acquisitions of locks that are never released till $e$ ($f$) is reached. Hence the crux of the scheduling is to schedule from $x_1$ and $x_2$ till $e$ and $f$.

The algorithm works by building a graph of *causal edges* between events. For every lock $l$ in the lockset of $f$, if $l$ occurs in the acquisition history of $e$ with respect to some lock $l'$, then we know that after the last acquisition of $l'$ by $T_1$, there was an acquisition (followed by a release) of the lock $l$. Hence we know that we must schedule the last release of lock $l'$ in $T_1$ (say event $u$) *before* the last acquisition of $l$ in $T_2$ (say $v$). We capture this by adding a causal edge from $u$ to $v$. Symmetrically, we examine the lockset of $e$ and the acquisition history of $f$ and throw in causal edges. It turns out that since the acquisition histories are compatible, this graph will by *acyclic*, and hence there is a schedule that respects these orderings. The algorithm simply takes a linearization of partial order to obtain a schedule.

The above argument is adapted from the *proof that the prediction algorithm works correctly*, which appears in Farzan et al [5], and in turn depends on a proof of a theorem by Kahlon et al [10]. Our contribution here is using it in the scheduling algorithm. We have also augmented the above construction by using *vector clocks* to rule out cut-points that are clearly infeasible. We maintain vector-clocks for events that get updated during *thread-creation* and at *barriers* only, and eliminate cut-points $(e, f)$ where $e$ and $f$ are not concurrent with respect to the vector clocks. This greatly reduces the number of infeasible interleavings PENELOPE generates.

### 4.2.2 Algorithms to adhere to original execution

Note that the theoretically synthesized schedule described above blindly executes $T_1$ till $x_1$ and $T_2$ till $x_2$ (i.e. to the last point where thread have empty locksets), and further ignores all other threads. This actually causes the scheduling algorithm to break in practice; when we implemented this, most predicted schedules were not feasible in the program. The reason is that the theoretical assumption that there is a set of $n$ threads from the beginning that do not communicate is not real in practice.

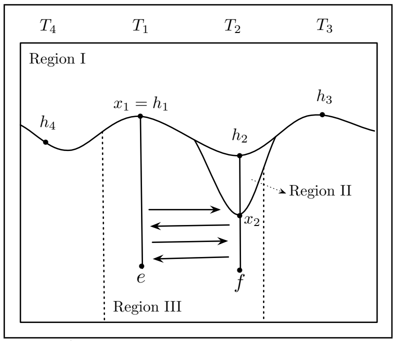
PENELOPE gets around this problem by scheduling a large *prefix* of the run accurately according to the original observed schedule. Adhering to the observed schedule lets us keep the causal conditions that exist in the actual program and yet allows us to get past it without modeling it or analyzing in our prediction model.

The scheduling algorithm implemented in PENELOPE works as follows. Consider a cut-point $(e, f)$ in threads $T_1$ and $T_2$ with disjoint locksets and compatible acquisition histories, and let $x_1$ and $x_2$ be the last events in threads $T_1$ and $T_2$, before $e$ and $f$, respectively, with empty locksets. Without loss of generality, assume $x_1$ occurs before $x_2$ in the original schedule. Now, for every thread $T_i$ (including $T_1$ and $T_2$), let $h_i$ be the *last event in $T_i$ that occurs before $x_1$ in the original schedule with its lockset empty*. Note that $h_1 = x_1$, but $h_2$ need not be equal to $x_2$. The idea now is to schedule all events of each thread $T_i$ up until each $h_i$, and moreover in the exact order as they occurred in the original schedule.

More precisely, for an observed schedule $\sigma$, we say event $j$ ($1 \leq j \leq |\sigma|$) is in the *prefix set $P$* iff $\sigma[j] = T_i{:}a$ and this event occurs before or is the same as the event $h_i$. The prefix predicted schedule is then simply the concatenation of all $\sigma[j]$, in increasing order of $j$, where $j \in P$. In other words, we run through the original schedule and pick an event $u$ to occur provided it is an event of thread $T_i$ (for some $i$) and occurs before $h_i$.

The rationale of why this prefix has the desired properties goes as follows.

First, notice that if an event $u$ of thread $T_i$ occurs in the prefix schedule, then so does every event of $T_i$ before $u$; hence the events we choose to execute certainly have their predecessor events in the same thread scheduled. Second, we argue that the schedule is lock-respecting. Assume that there is an acquisition of a lock $l$ at an event $u$ of $T_i$ in the prefix. Then the only way this lock is not feasible to acquire is that another thread $T_j$ acquired it but did not release it. But since the original observed execution was feasible, there must have been a release of this lock by $T_j$ in the original execution, before $u$. This release must be included in the predicted prefix execution as well, because we have chosen each $h_k$ to be an event at which the lockset is empty. Consequently the release must have happened before $u$ in the predicted prefix schedule as well. Finally, note that since this schedule adheres to the original thread to a large extent it is likely to be feasible in the real program as well. This prefix is represented as Region I in the figure above.



The next middle phase of the predicted execution is to execute the program from $h_2$ to $x_2$ in $T_2$ (marked as Region II in the figure). Since *no other thread* holds any lock, this is at least lock-feasible.

Finally, we come to the last phase of the predicted execution (Region III), which schedules events in between $x_1$ and $e$ and between $x_2$ and $f$. This part is done according to the theoretical schedule synthesized in the previous section, that uses locksets and acquisition histories to go back and forth between the two threads $T_1$ and $T_2$ only to reach $e$ and $f$. Once we reach $e$ and $f$, we let all threads go free to execute (this will, hopefully, make $T_1$ execute $e_2$ eventually which will cause the atomicity violation).

In practice, events in the prefix, Region I, constitute the majority of events compared to Regions II and III. Hence adhering to the original observed execution accurately in this region greatly increases the chance for the schedule being feasible, and as we show in the experiments, leads to most predicted executions being schedulable.

### 4.2.3 Heuristics for reducing context-switches

While the above described algorithms focus on generating executions that are likely to be *feasible*, they ignore the overhead complexity of actually scheduling the runs. The executions we observe

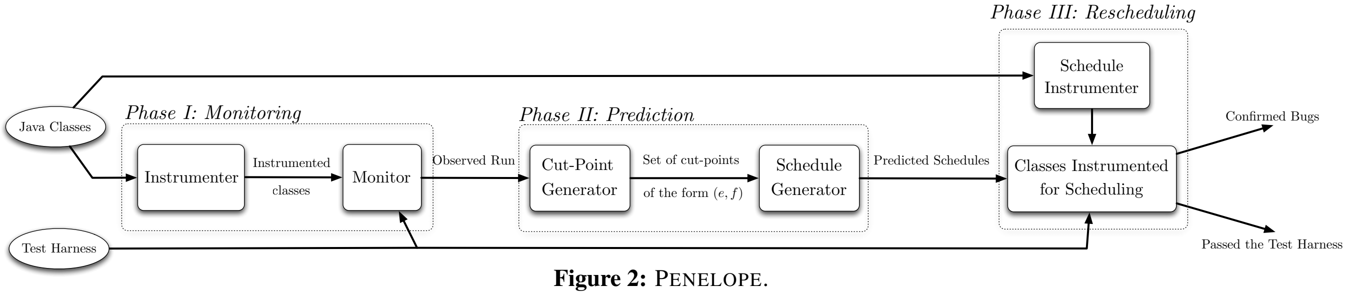


**Figure 2:** PENELOPE.

have a large number of context-switches (hundreds of thousands in some of our examples), and since our predicted executions try to adhere to the observed runs for feasibility, they also have a large number of context switches. We hence employ two heuristics to bring down the number of context-switches, while at the same time preserving feasibility. Both heuristics rely on the idea that if a sequence of actions is provably non-interfering among the threads, then we can execute the sequence in any order, and in particular, execute them with the least number of context-switches.

**Escape analysis to reduce context-switches:**
The first idea is to observe which of the (non-local) accesses to memory locations are actually shared in the run; in actual runs, several shared variables may be touched by only one thread. Let us call these memory locations *unshared* in the observed run. Unshared variables are not known in advance, and we do observe them, and they do play a role when we try to reschedule executions. However, accesses to unshared variables by a thread cannot affect another thread, and hence we can shuffle these to obtain fewer context-switches, without affecting feasibility.

**Identifying read-blocks to reduce context-switches:**
The second idea is that even if there are a large number of contiguous *reads* to shared variables, these accesses do not cause real interference between the threads. Hence the reads in this contiguous block can be shuffled in any way without affecting feasibility of the schedule. Again, we schedule these reads in a way that minimizes the number of context-switches, and reduce it to at most the number of active threads in the block.

## 5. IMPLEMENTATION

In this section, we explain the key implementation details of PENELOPE. Figure 2 illustrates the structure of PENELOPE. For more information on PENELOPE and the experiments presented in Section 6, see `http://www.cs.uiuc.edu/∼sorrent1/penelope`.

**Phase I: Monitoring:** We implemented our monitoring instrumenter using the Bytecode Engineering Library (BCEL) [2]. Every class file in bytecode is (automatically) transformed so that a *call* to a global monitor is made after each *relevant* action is performed. These relevant actions include field and static field reads/writes, entry/exits to synchronized blocks and methods, array reads/writes, etc., but excludes actions such as accesses to local variables. The *global monitor* communicates with all threads. Each thread informs the monitor when it is performing an action that needs to be observed.

**Phase II(a): Cut-point generation:** In this phase, the algorithm in Section 4.1 is used by the *cut-point generator* to identifies all possible *cut-points* off-line: i.e. pairs of events $(e, f)$, such that a schedule that reaches these points concurrently will violate atomicity. Note that our algorithm computes only one representative violation for each pair of threads, each entity, each program location, each pair of events with a compatible set of locksets and acquisition histories, and each pattern of violation (W-R-W and A-W-A). Since *recurrent* locks (multiple acquisitions of the same lock by the same thread) are typical in Java, the tool is tuned to handle them by suppressing from the analysis the subsequent acquisitions of the same lock by the same thread. A simple automatic *escape analysis* unit (written as a Perl script) excludes from the execution all accesses to thread-local entities by replacing them with *skips* (nops), which enables faster prediction.

**Phase II(b): Schedule generation:** The *schedule generator* synthesizes a schedule for each (predicted) *cut-point* using the algorithms described in Sections 4.2. The idea is that schedules are first synthesized up to the points where locksets are empty using the original observed schedule, and then the accurate theoretical scheduling algorithm is used to execute the events of $T_1$ and $T_2$ to cause an atomicity pattern violation. Also, we implemented the heuristics to reduce the number of context-switches by rearranging events that are not truly shared (using the *escape analysis*) as well as blocks of reads, to reduce the number of context-switches.

**Phase III: Rescheduling predicted schedules:** Our scheduler is also implemented using BCEL [2]; we instrument the scheduling algorithm into the Java classes using bytecode transformations, so that the same events that were monitored, now interact with a global scheduler. The scheduler, at each point, looks at the predicted schedule, and directs the appropriate thread to perform a sequence of $n$ steps. The threads stop at the first point with a relevant access, and wait for a signal from the scheduler to proceed, and only then, they execute the number of (observable) events they were asked to execute. After this, the threads communicate back to the scheduler, relinquishing the processor, and await further instructions.

The scheduler uses two vectors $S$ and $N$ to communicate with program threads. For each thread $T_i$, $N[i]$ will contain the number of steps that $T_i$ should take next, and $S[i]$ contains a Boolean value that allows $T_i$ progress when true. Only $S$ is used to communicate with threads, and is protected by global locks (one for each cell). Thread $T_i$ can safely access $N[i]$ without synchronization (as the synchronization in $S$ is used to provide the synchronization for $N$ as well). The thread $T[i]$ hence does a `wait()` on $S[i]$ (to become true), and the scheduler wakes up the thread when it is time for the thread to proceed. The scheduler then waits on $S[0]$ to become true. When the thread completes its designated number of steps (from $T[i]$), it sets $S[0]$ to true, notifies the scheduler, and then relinquishes control and waits on $S[i]$ to become true again. Once the execution reaches the point that the $e$ and $f$ event from the violation pattern are executed, the scheduler releases all threads to execute as they please. There is also a *timeout* mechanism that detects when the scheduler is trying to schedule an infeasible run. Once the scheduler reaches $e$ and $f$ and lets all threads go, if a timeout occurs, we signal a deadlock; note that at this point, if all threads are stuck, then this is a real deadlock in the concurrent program and

|  |  |  | Monitoring |  |  |  |  |  | Prediction |  | Scheduling |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Application (LOC) | Input | Execution Time | Threads | Entities | Locks | Execution Length | Execution Time | Context Switches | Number of Predicted Schedules | Execution Time | Patterns |  |  | Context Switches | Average Time per feasible schedule | Total Time rescheduling | Errors |
|  |  |  |  |  |  |  |  |  |  |  | A | N | F |  |  |  |  |
| Vector (1.3K) | VectorTest | 0.22s | 4 | 12 | 2 | 142 | 0.29s | 5 | 2 | 0.01s | 0 | 0 | 2 | 5 | 0.19s | 1.10s | 1 |
|  | VectorTest1 | 0.20s | 4 | 12 | 2 | 231 | 0.20s | 5 | 4 | 0.01s | 0 | 1 | 3 | 5 | 0.33s | 1.61s | 1 |
|  | VectorTest2 | 0.18s | 4 | 12 | 2 | 248 | 0.20s | 5 | 4 | 0.01s | 0 | 0 | 4 | 5 | 0.23s | 1.57s | 1 |
|  | VectorTest3 | 0.27s | 4 | 12 | 2 | 231 | 0.27s | 5 | 3 | 0.01s | 0 | 0 | 3 | 5 | 0.21s | 1.27s | 1 |
|  | VectorTest4 | 0.33s | 4 | 12 | 2 | 159 | 0.35s | 3 | 1 | 0.01s | 0 | 0 | 1 | 3 | 0.35s | 1.27s | 1 |
| Stack (1.4K) | StackTest | 0.21s | 4 | 12 | 2 | 136 | 0.24s | 6 | 2 | 0.01s | 0 | 0 | 2 | 5 | 0.32s | 1.06s | 1 |
|  | StackTest1 | 0.48s | 4 | 12 | 2 | 209 | 0.58s | 6 | 1 | 0.01s | 0 | 0 | 1 | 6 | 0.56s | 1.35s | 1 |
|  | StackTest2 | 0.29s | 4 | 12 | 2 | 231 | 0.28s | 5 | 3 | 0.01s | 0 | 1 | 2 | 5 | 0.35s | 1.77s | 1 |
|  | StackTest3 | 0.17s | 4 | 12 | 2 | 265 | 0.27s | 6 | 3 | 0.01s | 0 | 0 | 3 | 6 | 0.18s | 1.20s | 1 |
|  | StackTest4 | 0.18s | 4 | 12 | 2 | 127 | 0.18s | 4 | 2 | 0.01s | 0 | 0 | 2 | 4 | 0.19s | 1.04s | 1 |
| elevator (566) | data | 16.37s | 3 | 65 | 8 | 26K | 18.94s | 899 | 167 | 5.24s | 0 | 164 | 3 | 595 | 16.78s | 30m42s | 0 |
|  | data2 | 16.77s | 5 | 113 | 8 | 54K | 16.26s | 1428 | 63 | 3.87s | 0 | 59 | 4 | 1262 | 17.12s | 11m44s | 0 |
|  | data3 | 16.37s | 5 | 457 | 50 | 329K | 16.93s | 50K | 699 | 6m48s | 42 | 651 | 6 | 39K | 17.13s | 2h52m | 0 |
| tsp (794) | map14 | 0.19s | 2 | 588 | 2 | 32M | 3m41s | 72K | 83 | 1m36s | 29 | 45 | 9 | 60K | 15.85s | 50m01s | 0 |
|  | map14 | 0.15s | 4 | 652 | 2 | 14M | 1m58s | 22K | 168 | 29.98s | 87 | 70 | 11 | 18K | 16.16s | 23m41s | 0 |
| raytracer (1.5K) | SizeA | 3.38s | 10 | 80 | 10 | 560 | 3.57s | 54 | 90 | 0.01s | 0 | 0 | 90 | 48 | 3.57s | 5m52s | 1 |
|  | SizeA | 3.94s | 20 | 150 | 20 | 1.5K | 4.03s | 106 | 380 | 0.03s | 0 | 0 | 380 | 93 | 4.14s | 26m20s | 1 |
|  | SizeA | 6.52s | 30 | 220 | 30 | 2.9K | 6.08s | 150 | 870 | 0.09s | 0 | 0 | 870 | 99 | 6.75s | 1h38m | 1 |
|  | SizeB | 36.50s | 10 | 80 | 10 | 560 | 36.62s | 59 | 90 | 0.01s | 0 | 0 | 90 | 44 | 38.46s | 58m16s | 1 |
| colt (29K) | dgemm 50X50 | 0.27s | 3 | 12K | 0 | 286K | 0.42s | 358 | 0 | 1.7s | 0 | 0 | 0 | 297 | - | - | 0 |
| Pool 1.2 (5.8K) | PoolTest | 0.19s | 4 | 28 | 1 | 98 | 0.19s | 11 | 1 | 0.01s | 0 | 0 | 1 | 4 | 0.17s | 0.18s | 1 |
|  | PoolTest1 | 0.20s | 4 | 29 | 1 | 267 | 0.24s | 25 | 27 | 0.01s | 0 | 3 | 24 | 19 | 0.24s | 8.20s | 1 |
|  | PoolTest2 | 0.12s | 4 | 15 | 1 | 104 | 0.17s | 5 | 7 | 0.01s | 0 | 0 | 7 | 5 | 0.20s | 1.48s | 1 |
|  | PoolTest3 | 0.12s | 4 | 14 | 1 | 251 | 0.18s | 20 | 145 | 0.01s | 50 | 46 | 36 | 49 | 0.20s | 20.76s | 1 |
| Pool 1.3 (7K) | PoolTest | 0.25s | 4 | 30 | 1 | 100 | 0.21s | 3 | 0 | 0.01s | 0 | 0 | 0 | 5 | - | - | 0 |
|  | PoolTest1 | 0.18s | 4 | 31 | 1 | 265 | 0.25s | 14 | 19 | 0.02s | 6 | 13 | 0 | 14 | - | 5.78s | 0 |
|  | PoolTest2 | 0.12s | 4 | 15 | 1 | 112 | 0.17s | 5 | 6 | 0.01s | 0 | 6 | 0 | 5 | - | 1.06s | 0 |
|  | PoolTest3 | 0.12s | 4 | 18 | 1 | 250 | 0.88s | 24 | 99 | 0.01s | 52 | 47 | 0 | 22 | - | 11.31s | 0 |
| Apache FtpServer (22K) | lgn_script | 1m02s | 5 | 67 | 4 | 412 | 1m02s | 9 | 109 | 0.02s | 5 | 83 | 31 | 9 | 1m03s | 2h16m | 5 |

**Table 1: Experimental Results. A, N and F (of Patterns) indicate number of schedules that are, respectively, "Already appeared in observed execution", "Not feasible" and "Feasible".**

hence corresponds to a real bug (though not an atomicity-related bug). PENELOPE detects these kinds of deadlocks as well.

# 6. EVALUATION

We ran PENELOPE on a benchmark suite of 9 programs, against several test harnesses and input parameters, and we evaluated it under several criteria: whether it is successful in predicting alternative schedules with violation patterns, whether it is able to execute the predicted schedules, the time it takes to predict and schedule executions, and whether it is able to discover errors by checking the result of the execution of schedules with violations on test harnesses.

**Benchmarks.** The benchmarks used are all concurrent Java programs that use `synchronized` blocks and methods as means of synchronization (using synchronized blocks automatically ensures nested locking for the most part). They include `raytracer` from the Java Grande multithreaded benchmarks [1], `elevator` and `tsp` from [18], `Vector` and `Stack` from Java libraries, `Pool` (two different releases) from Apache Commons, `Colt`, and `Apache FtpServer`.

The concurrent program `elevator` simulates multiple lifts in a building; `tsp` is a parallel program that solves the traveling salesman problem for a given input map; `raytracer` renders a frame of an arrangement of spheres from a given view point; `Pool` is an object pooling API in the Apache Commons suite; `Colt` is an open source library for high performance computing; `Apache FtpServer` is a ftp server and by Apache; and `Vector` and `Stack` are Java libraries that respectively implement the concurrent vector and the concurrent stack data structures.

**Test Suites.** In Table 1, we provide all the relevant information about the conditions under which the tests were run, such as input files and parameters. For `elevator`, and `tsp`, the input files were included in the benchmarks, and the table indicates which input file was used for the results. For `elevator`, the number of threads was also specified in the input files, and there were no additional parameters to be provided by the user. The test harness for `tsp` includes an input file, a given number of threads, and a script would compare the minimum tour computed by the program against the minimum tour computed by a single thread execution.

For `Vector` (respectively `Stack`), we wrote test harnesses with two threads and two small vectors (respectively stacks), where each thread executes exactly one method from class `Vector` (respectively `Stack`). All the scheduled runs which manifested real errors raised exceptions during the execution. The test cases for `Pool` were designed similarly, that is for each test case two different methods were run concurrently on the same object pool. There was no particular test harness check, and the error in `Pool 1.2` manifested as an exception. The same bug was fixed in `Pool 1.3`.

For `colt`, the `dgemm` command was used which invokes a matrix multiplication routine on matrices of size 50 × 50. The test harness was designed to check the result from a multi-threaded execution against the result from a single threaded execution. For `raytracer`, all the data is already incorporated in the benchmark which comes in two sizes A, and B. The only parameter that the user specifies is the number of threads. `raytracer` has a built-in validation test. In the case of `FtpServer`, we wrote a test harness with a client and a server where the client logs in and requests a connection; the errors manifested as exceptions.

**Experimental Evaluation.** Table 1 demonstrates the result of the evaluation of Penelope on the benchmarks. The table provides information about all three phases: monitoring, prediction, and scheduling. For the monitoring phase, the number of threads, entities (variables), locks, and the length of execution is reported, as well as how long the execution takes, and how many context switches exist in the observed run. For the prediction phase, we report how many violations were discovered (total over all 5 patterns), and how long the prediction phase takes. In the scheduling part, we report the number of violation patters (out of the total reported in the prediction column) existed in the original run observed (A), the number of violations for which the schedules were not feasible (N), and the number of violations which appeared in a successful alternative schedule generated by PENELOPE (F). We also report how many context switches (on average) there are in the alternative feasible schedules, what is the average time per feasible schedule (a reasonable indication of overhead), and finally and most importantly, how many *real* errors were found.

PENELOPE finds several bugs in the benchmarks. The bug in `raytracer` is caused by an atomicity violation involving the field `JGFRayTracerBench.checksum1` due to wrong synchronization. The error in `Pool 1.2` (which was fixed in `Pool 1.3`) is caused by an atomicity violation on the variable `_factory` in methods `borrowObject`, `returnObject`, ran in parallel with method `close`, and in methods `addObject` and `borrowObject` ran in parallel with method `setFactory`. Note that these are 4 different errors, and they all manifested as exceptions during alternative feasible schedules exercised by PENELOPE.

All bugs in `Vector` and `Stack` are the result of an atomicity violation that causes the size of a parameter collection to go stale in the middle of an operation (such as `addAll`) that is using the parameter collection as a source of information, while only the destination vector/stack is synchronized properly. There were 5 discovered errors for `FtpSever` (each giving a different exception), which correspond to variables `m_currConnections`, `m_writer`, `m_name`, `m_currLogins`, `m_request` all accessed in method `RequestHandler.run` of the server while the timer thread interrupts by closing the connection as a result of a timeout.

PENELOPE predicts several atomicity violating schedules in `tsp` and `elevator`, but they all pass the test harness, and in fact are not errors (the violation of atomicity was intended and correct).

**Observations:** Here are some observations we gather from these experiments on the effectiveness and performance of PENELOPE:

- *The number of predicted schedules is small*; in fact, a tiny fraction of all possible executions. This is true even compared to the number of all runs limited to just two context switches (or preemptions), as CHESS would do. For instance, in `elevator` (data3), there are close to 50,000 points (releases of locks) in 4 threads where preemptions can happen, giving around 15 billion possible schedules involving just 2 preemptions!
- PENELOPE *is effective in finding bugs*. We ran the programs under the test harness several times, and did not find (almost) any of the reported bugs in any of these benchmarks by merely running tests randomly. It is clear that a more focused approach is absolutely necessary in finding errors on these benchmarks. Despite its small selection of schedules to test, PENELOPE was able to identify bugs in these programs.
- *Reasonable time overhead*. The runtime overhead in precisely scheduling the alternate executions is not prohibitively high, and is in fact very minimal in most examples. This is despite the large number of context-switches that are being exercised (284K context-switches in elevator-data3).
- PENELOPE *finds bugs under complex scenarios*. Note that the number of context-switches scheduled in the predicted executions are very high. We believe that this allows PENELOPE to dig deep into the search space of the runs. Tools like CHESS execute all runs with a few context-switches and offer a complementary search strategy [13].
- *Zero false positives*. If a bug is reported by PENELOPE, it is a real bug (i.e. an execution that violates the test harness). A significant amount of violations found did not correspond to real bugs, and are not reported as bugs. This is in contrast to similar tools based on atomicity checking by Wang and Stoller [20], Farzan et al [5], and the tools SIDETRACK [23], ATOMFUZZER [15] and VELODROME [7]. VELODROME in fact reports atomicity violations for benchmarks `elevator`, `tsp`, and `colt`, though they do *not* correspond to bugs.

# 7. REFERENCES

[1] http://http://www.javagrande.org/.

[2] http://jakarta.apache.org/bcel/.

[3] F. Chen, T.F. Serbanuta, and G. Rosu. jpredictor: a predictive runtime analysis tool for java. In *ICSE*, pages 221–230, 2008.

[4] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *CAV*, pages 52–65, 2008.

[5] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *CAV*, pages 248–262, 2009.

[6] C. Flanagan and S. N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.

[7] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, pages 293–303, 2008.

[8] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349, 2003.

[9] J. Hatcliff, Robby, and M. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model checking. In *VMCAI*, pages 175–190, 2004.

[10] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, pages 505–518, 2005.

[11] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

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

[13] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455. ACM, 2007.

[14] C. Papadimitriou. *The theory of database concurrency control.* Computer Science Press., 1986.

[15] C-S Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT '08/FSE-16*, pages 135–145, New York, NY, USA, 2008. ACM.

[16] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.

[17] S., J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, pages 37–48, 2006.

[18] C. von Praun and T. R. Gross. Object race detection. *SIGPLAN Not.*, 36(11):70–82, 2001.

[19] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, 2010.

[20] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, pages 137–146, 2006.

[21] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, 2006.

[22] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005.

[23] J. Yi, C. Sadowski, and C. Flanagan. Sidetrack: generalizing dynamic atomicity analysis. In *PADTAD*, pages 1–10, 2009.