jPredictor: A Predictive Runtime Analysis Tool for Java

Feng Chen
CS Department
University of Illinois, Urbana
fengchen@cs.uiuc.edu

Traian Florin Şerbănuţă
CS Department
University of Illinois, Urbana
tserban2@cs.uiuc.edu

Grigore Roşu
CS Department
University of Illinois, Urbana
grosu@cs.uiuc.edu

ABSTRACT

jPredictor is a tool for detecting concurrency errors in Java programs. The Java program is instrumented to emit property-relevant events at runtime and then executed. The resulting execution trace is collected and analyzed by jPredictor, which extracts a causality relation sliced using static analysis and refined with lock-atomicity information. The resulting abstract model, a hybrid of a partial order and atomic blocks, is then exhaustively analyzed against the property and errors with counter-examples are reported to the user. Thus, jPredictor can "predict" errors that did not happen in the observed execution, but which could have happened under a different thread scheduling. The analysis technique employed in jPredictor is fully automatic, generic (works for any trace property), sound (produces no false alarms) but it is incomplete (may miss errors). Two common types of errors are investigated in this paper: dataraces and atomicity violations. Experiments show that jPredictor is precise (in its predictions), effective and efficient. After the code producing them was executed only once, jPredictor found all the errors reported by other tools. It also found errors missed by other tools, including static race detectors, as well as unknown errors in popular systems like Tomcat and the Apache FTP server.

Categories and Subject Descriptors

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

General Terms

Verification, Algorithms

Keywords

predictive runtime analysis, sliced causality, runtime verification

1. INTRODUCTION

Concurrent systems in general and multithreaded systems in particular may exhibit different behaviors when executed at different times. This inherent nondeterminism makes multithreaded programs difficult to analyze, test and debug. This paper presents a novel runtime predictive analysis technique to correctly detect concurrency errors from observing execution traces of multithreaded programs. By "correct" or "sound" prediction of errors, in this paper we mean *no false alarms*. The program is automatically instrumented to emit runtime events to an external observer. The particular execution that is observed needs *not* hit the error; yet, errors in other executions can be correctly predicted together with counter-examples leading to them. This technique has been implemented in jPredictor and evaluated on several non-trivial Java applications.

There are several other approaches also aiming at detecting potential concurrency errors by examining particular execution traces. Some of these approaches aim at verifying general purpose properties [21, 22], including temporal ones, and are inspired from debugging distributed systems based on Lamport's *happens-before* causality [17]. Other approaches work with particular properties, such as data-races and/or atomicity. [20] introduces a first lock-set based algorithm to detect data-races dynamically, followed by many variants aiming at improving its accuracy. For example, an ownership model was used in [27] to achieve a more precise race detection at the object level. [19] combines the lock-set and the happen-before techniques. The lock-set technique has also been used to detect atomicity violations at runtime, e.g., the reduction based algorithms in [12] and [28]. [28] also proposes a block-based algorithm for dynamic checking of atomicity built on a simplified happen-before relation, as well as a graph-based algorithm to improve the efficiency and precision of runtime atomicity analysis.

Previous efforts tend to focus on either soundness or coverage: those based on happens-before try to be sound, but have limited coverage over interleavings, resulting in more false negatives; lock-set based approaches have better coverage but suffer from false alarms. Our runtime analysis technique proposed in this paper aims at covering more interleavings from one execution without giving up soundness or genericity of properties (errors may still be missed, e.g., when the code causing errors is not executed in one execution). It combines *sliced causality* [7], a happen-before causality drastically but soundly sliced by removing irrelevant causalities using semantic information about the program obtained with an apriori static analysis, with *lock-atomicity*. Our predictive runtime analysis technique can be understood as a hybrid of testing and model checking. Testing because one runs the system and observes its runtime behavior in order to detect errors, and model checking because the special causality with lock-atomicity extracted from the running program can be regarded as an abstract model of the program, which can further be investigated exhaustively by the observer in order to detect potential errors.

This paper makes two important contributions:

1. It proposes a sound and generic (with respect to the trace property to check) approach to extend causal partial orders with lock-atomicity semantics to improve the coverage of predictive analysis, together with a technique to efficiently compute the sliced causality with lock-atomicity using vector clocks and lock sets; and

ICSE'08, May 10–18, 2008, Leipzig, Germany.

2. It presents and empirically evaluates with encouraging results jPredictor, a predictive runtime analysis tool for Java that implements the above technique, together with specialized algorithms to efficiently check data races and atomicity based on sliced causality with lock-atomicity.

Many concurrency errors, including several previously unknown ones caused by data races or atomicity violations, were discovered using jPredictor in popular open source programs such as the Apache Commons package and the Tomcat webserver. After the code producing them was executed *only once*, jPredictor found all the data-race or atomicity errors reported by other tools. It also found errors missed by other tools, including static race detectors supposed to have high code coverage. Even though for performance reasons jPredictor slightly deviates in its implementation from the soundness of the sliced causality with lock-atomicity technique, *no* false alarms were ever reported in our experiments.

This paper is organized as follows. Section 2 contains the main theoretical developments: it first gives a brief overview of sliced causality, then proposes a sound but incomplete vector clock algorithm to capture it, then proposes our general technique to extend causalities with lock-atomicity, and finally describes an algorithm to generate sound linearizations of events consistent with both the sliced causality and the lock-atomicity. Section 3 is dedicated to the implementation of jPredictor. The evaluation results are discussed in Section 4 and Section 5 concludes the paper.

2. PREDICTIVE RUNTIME ANALYSIS

2.1 Sliced causality

Previous approaches to detect concurrency bugs based on happen-before techniques, e.g., [19, 21, 22], extract causal partial orders from analyzing *exclusively* the dynamic thread communication in executions. But as discussed in [23], without additional information about the structure of the program that generated the event trace, the least restrictive causal partial order that an observer can extract is the one in which each write event of a shared variable precedes all the corresponding subsequent read events and which is a total order on the events generated by each thread. Since this causality considers *all* interactions among threads, e.g., all reads/writes of shared variables, the obtained causal partial orders are rather *restrictive*, or *rigid*, in the sense of allowing a reduced number of linearizations and thus of errors that can be detected; in general, the larger the causality (as a binary relation) the fewer linearizations it has, i.e., the more restrictive it is.

In [7] we defined *sliced causality*, a causal partial order relation significantly reducing the size of the computed causality without giving up soundness or genericity of properties to check: it works with any *monitorable* (safety) properties, including regular patterns, temporal assertions, data-races, atomicity, etc. In this subsection we recall sliced causality and intuitively explain how and why sliced-causality-based predictive runtime analysis techniques have an increased coverage still avoiding any false alarms.

Consider a simple and common safety property for a shared resource, that any access should be authenticated. Figure 1 shows a buggy program using the shared resource. The main thread authenticates and then the task thread uses the authenticated resource. They communicate via the `flag` variable. Synchronization is unnecessary, since only the main thread modifies `flag`. However, the developer makes a (rather common [10]) mistake, using `if` instead of `while` in the task thread. Suppose now that we observed a successful run of the program, as shown by the arrow. Techniques based on traditional happen-before will not be able to find this bug, due to the causality induced by the write/read on `flag`. But since `resource.access()` is *not* controlled by `if` [1], sliced-causality techniques will correctly predict this error from the successful execution. When the bug is fixed replacing `if` with `while`, `resource.access()` is controlled by `while` (since it is a potentially non-terminating loop), so no violation is reported.

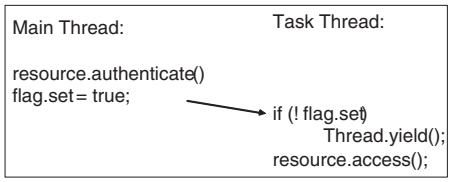

Figure 1: Multi-threaded execution

Based on an apriori static analysis, sliced causality drastically cuts the usual happen-before causality by removing unnecessary dependencies; this way, a significantly larger number of consistent runs can be inferred and thus analyzed. Experiments show that, on average, the sliced causality relation has 50% or less direct inter-thread causal dependencies compared to happen-before [7]. Since the number of linearizations of a partial order tends to be exponential with the size of the *complement* of the partial order (as a binary relation), any linear reduction in size of the sliced causality compared to traditional happen-before, is expected to *increase exponentially the coverage* of the analysis. Indeed, the use of sliced causality allowed us to detect concurrency errors that are unlikely be detected using conventional happen-before causalities.

The sliced causality is constructed by making use of dependence information obtained both statically and dynamically. Briefly, instead of computing the causal partial order on all the observed events like in the traditional happen-before based approaches, our approach first slices the trace according to the desired property and then computes the causal partial order on the achieved slice; the slice contains all the *property events*, i.e., events relevant to the property, as well as all the *relevant events*, i.e., events upon which the property events depend, directly or indirectly. This way, irrelevant causality on events is trimmed without breaking the soundness of the approach, allowing more permutations of relevant events to be analyzed and resulting in better coverage of the analysis.

Unlike in (static) program slicing [14] which is based on dependencies between statements, to assure correct causal slicing we employ dependencies between finer grained units here, namely between *events*. Our dependency analysis keeps track of actual memory locations in every event, available at runtime, avoiding inter-procedural analysis. Intuitively, event e' *depends upon* event e, written $e \sqsubset e'$, iff a change of e may change or eliminate e'; in other words, *e should occur before e' in any consistent permutation of events*. We distinguish: (1) *control dependence*, written $e \sqsubset_{ctrl} e'$, when a change of the state of e may eliminate e'; and (2) *data-flow dependence*, written $e \sqsubset_{data} e'$, when a change of the state of e may lead to a change in the state of e'. Control dependence only relates events generated by the same thread. Data-flow dependence may relate events generated by different threads (e.g., e writes shared variable x in thread t, then e' reads x in thread t').

An additional dependence relation, called the *relevance dependence* in [7], is also needed for soundness. In Figure 2, threads T_1 and T_2 are executed as shown by the solid arrows (A), yielding the event sequence "e_1, e_2, e_3, e_4, e_5, e_6" (B). Suppose the property to check refers only to y; the property events are then e_1, e_5, and e_6. Events e_2 and e_3 are immediately marked as relevant, since $e_2 \sqsubset_{data} e_3 \sqsubset_{ctrl} e_5$. If only closure under control- and data-dependence was used to compute the relevant events, like in dynamic program slicing [1], then e_4 would appear to be irrelevant, so one may conclude that "e_2, e_6, e_1, e_3, e_5" is a sound permutation;

[1] It is more complicated to decide the control dependence when more control flow statements, e.g., exception throwing, are considered. Interested readers can refer to [8] for details.

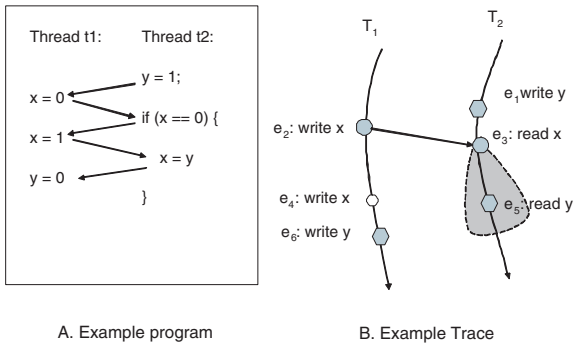

Figure 2: Example for relevance dependence

there is, obviously, no execution that can produce that trace, so one reported a false alarm if that trace violated the original property on y. Consequently, e_4 is also a relevant event and $e_3 \sqsubset_{rlvn} e_4$.

Sliced causality is defined as the transitively closed union of the three dependencies above (together with the total intra-thread orders). Its sound use in predictive runtime analysis techniques is given by the following important result (see [7] for formal details).

THEOREM 1. *If "$\prec$" is the sliced causality, then any permutation of property events that is consistent with $\prec$ corresponds to some possible execution of the multi-threaded system.*

Therefore, one can run the system once, extract a sliced causality, and then analyze sliced-causality-consistent permutations of property events. This way, one can detect potential violations of properties *without* re-executing the program.

2.2 Extracting Sliced Causality

We here describe a technique to extract from an execution trace of a multithreaded system the sliced causality relation corresponding to some property of interest φ. Our technique is *offline*, in the sense that it takes as input an already generated execution trace; that is because it needs to traverse the trace backwards. Our technique consists of two steps: ***(1)*** all the irrelevant events (those which are neither property events nor relevant events) are removed from the original trace, obtaining the *(φ)-sliced trace*; and ***(2)*** a *vector clock (VC)* based algorithm is applied on the sliced trace to capture the sliced causality partial order.

2.2.1 Slicing Traces

Our goal here is to take a trace ξ and a property φ, and to generate a trace ξ_φ obtained from ξ filtering out all its events which are irrelevant for φ. When slicing the execution trace, one must nevertheless keep all the property events. Moreover, one must also keep any event e with $e\ (\sqsubset_{ctrl} \cup \sqsubset_{data})^+\ e'$ for some property event e'. This can be easily achieved by traversing the original trace backwards, starting with ξ_φ empty and accumulating in ξ_φ events that either are property events or have events depending on them already in ξ_φ. One can employ any off-the-shelf analysis tool for data- and control- dependence; e.g., our predictive analysis tool, JPREDICTOR, uses termination-sensitive control dependence [8].

The algorithm informally described above is a variant of *dynamic program slicing* [1], where the slicing criterion is not the conventional reachability of a particular program statement, but, more generally, a set of event patterns determined by the desired property (e.g., reads/writes of a shared location for dataraces, etc.). Unfortunately, one backwards traversal of the trace does not suffice to correctly calculate all the relevant events. Let us re-consider the example in Figure 2. When the backward traversal first reaches e_4, it is unclear whether e_4 is relevant or not, because we have not seen e_3 and e_2 yet. Thus a second scan of the trace is needed to include e_4. Once e_4 is included in ξ_φ, it may induce other relevance dependencies, requiring more traversals of the trace to include them. This process ceases only when no new relevant events are detected and thus resulting sliced trace stabilizes.

As the discussion preceding Theorem 1 shows, if one misses relevant events like e_4 then one may "slice the trace too much" and, consequently, one may produce false alarms. Because at each trace traversal some event is added to ξ_φ, the worse-case complexity of the sound trace slicing procedure is square in the number of events. Since execution traces can be huge, in the order of billions of events, any trace slicing algorithms that is worse than linear may easily become prohibitive. For that reason, JPREDICTOR slices the trace only once, thus achieving an approximation of the complete slice that can, in theory, lead to false alarms. However, our experiments show that this approximation is actually very precise in practice: all the programs that we have evaluated follow our approximation (Section 4).

2.2.2 Capturing Sliced Causality with Vector Clocks

Vector clocks [17] are routinely used to capture causal partial orders in distributed and concurrent systems. A *VC*-based algorithm was presented in [22] to encode a conventional multithreaded-system "happen-before" causal partial order on the unsliced trace. We next adapt that algorithm to work on our sliced trace and thus to capture the sliced causality. Recall from [22] that a vector clock (*VC*) is a function from threads to integers, $VC : T \to Int$. We say that $VC \leq VC'$ iff $\forall t \in T, VC(t) \leq VC'(t)$. The max function on *VCs* is defined as: $\max(VC_1, ..., VC_n)(t) = \max(VC_1(t), ..., VC_n(t))$.

Before we explain our *VC* algorithm, let us introduce our event and trace notation. An *event* is a mapping of *attributes* into corresponding *values*. One event can be, e.g., $e_1 : (counter = 8, thread = t_1, stmt = L_{11}, type = write, target = a, state = 1)$, which is a write on location a with value 1, produced at statement L_{11} by thread t_1. One can include more information into an event by adding new attribute-value pairs. We use $key(e)$ to refer to the value of attribute *key* of event e. To distinguish different occurrences of events with the same attribute values, we add a designated attribute to every event, *counter*, collecting the number of previous events with the same attribute-value pairs (other than the *counter*). A *trace* is a finite sequence of events. From here on, our default trace is the φ-sliced trace ξ_φ obtained in Section 2.2.1.

Intuitively, vector clocks are used to track and transmit the causal partial ordering information in a concurrent computation, and are typically associated with elements participating in such computations, such as threads, processes, shared variables, messages, signals, etc. If *VC* and *VC'* are vector clocks such that $VC(t) \leq VC'(t)$ for some thread t, then we can say that *VC'* has newer information about t than *VC*. In our *VC* technique, every thread t keeps a vector clock, VC_t, maintaining information about all the threads obtained both locally and from thread communications (reads/writes of shared variables). Every shared variable is associated with two vector clocks, one for writes (VC^w_x) used to enforce the order among writes of x, and one for all accesses (VC^a_x) used to accumulate information about all accesses of x. They are then used together to keep the order between writes and reads of x. Every property event e found in the analysis is associated a *VC* attribute, which represents the computed causal partial order. We next show how to update these VCs when an event e is encountered (the third case can overlap the first two; if so, the third case will be handled first):

1. $type(e) = write$, $target(e) = x$, $thread(e) = t$ (the variable x is written in thread t) and x is a shared variable. In this case, the write vector clock VC^w_x is updated to reflect the newly obtained information; since a write is also an access, the access *VC* of x is also updated; we also want to capture that t committed a causally irreversible action, by updating its *VC* as well: $VC_t \leftarrow VC^a_x \leftarrow VC^w_x \leftarrow \max(VC^a_x, VC_t)$.

2. $type(e) = read$, $target(e) = x$, $thread(e) = t$ (the variable x is read in t), and x is a shared variable. Then the thread updates its information with the write information of x (we do not want to causally order reads of shared variables!), and x updates its access information with that of the thread: $VC_t \leftarrow \max(VC^w_x, VC_t)$ and then $VC^a_x \leftarrow \max(VC^x_a, VC_t)$.
3. e is a property event and $thread(e) = t$. In this case, let $VC(e) := VC_t$. Then $VC_t(t)$ is increased to capture the intra-thread total ordering: $VC_t(t) \leftarrow VC_t(t) + 1$.

The vector clocks associated with property events as above soundly, but incompletely, capture the sliced causality:

THEOREM 2. *$e \prec e'$ implies $VC(e) \leq VC(e')$.*

The proof of Theorem 2 can be (non-trivially) derived from the one in [22]. The extension here is that the dependence is taken into account when computing the sliced trace. Note that, unlike in [22], the partial order $\leq$ among *VCs* is *stronger* than the causality. This is because when *VCs* are computed, the write-after-read order is also taken into account (the first case above), which the sliced causality $\prec$ does not need to encode. We do not know how to faithfully capture the sliced causality using *VCs* yet. Nevertheless, soundness is not affected because Theorems 1 and 2 yield the following:

COROLLARY 1. *Any permutation of property events consistent with $\leq$ (on events' VCs) is sound w.r.t. the sliced causality $\prec$.*

2.3 Causality with Lock-Atomicity

Happen-before techniques for detecting/predicting concurrency bugs typically rely on checking, directly or indirectly, linearizations of events consistent with a causal partial order. Two happen-before techniques that we are aware of generalize the concept of causality beyond a partial-order. One example is [19], which proposes a hybrid approach combining the happen-before causality with lock-set techniques for detecting data-races. The algorithm in [19] is, unfortunately, unsound and specialized for detecting data-races, so it cannot be used to generate sound linearizations of events to check against arbitrary properties. Another example is [23], which groups a write atomicity with all its subsequent reads; the resulting causality is extended with atomicity information and new linearizations of events are allowed, namely those that permute groups of events in the same atomic block. In this subsection we present another generalization of causality beyond a partial-order, one borrowing the idea of atomic blocks from the technique in [23], but whose atomicity is given by the semantics of locks rather than by writes and subsequent reads of shared variables. Unlike the technique in [19], our novel causality is general purpose, in the sense that it still allows for sound linearizations of events, so it can be used in combination with any trace property, data-races being only a special case. Even though in this paper we use this improved causality as a generalization of our sliced causality, the idea is general and can be used in combination with any other happen-before causality.

Our causality with lock-atomicity below is reminiscent of the notion of synchronization dependence defined in [13] and used for static slicing there. However, our causality is based on runtime events instead of statements in the control-flow graph, and is used to assure the (dynamic) causal atomicity of lock-protected blocks.

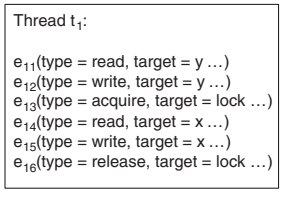

Figure 3: Event trace containing lock operations

A simple sound approach to partially incorporating lock semantics into causality, followed for example in [22], is to regard locks as shared variables and their acquire and release as reads and writes. This way, blocks protected by the same lock are ordered and kept separate. However, this ordering is stronger than the actual lock semantics (which only requires mutual exclusion). We extend our sliced causality to take into account the actual semantics of lock-atomicity: the set of events generated within a lock-protected block are considered lock-atomic. Two lock-atomic event sets w.r.t. the same lock cannot be interleaved, but *can be permuted* if there are no other causal constraints on them. Consider two more types of events for lock operations, *acquire* and *release*, whose target is the accessed lock. If there are embedded same-lock lock operations (a thread can acquire the same lock multiple times), only the outermost acquire-release event pair is considered. Figure 3 shows a trace containing lock events.

From here on, let τ be any execution trace. Let "$<$" be the union of the total intra-thread orders induced by τ, that is, $e < e'$ iff $thread(e) = thread(e')$ and e appears before e' in τ. Let $\prec$ be any partial order on the events in τ. The lock-atomicity technique described below can be therefore used in combination with any (causal) partial order. We are, however, going to apply the subsequent results for a particular causality, namely the sliced causality restricted to relevant and property events (see Corollary 2). When doing that, we also discard all those irrelevant acquire/release events (i.e., those synchronizing only irrelevant events).

DEFINITION 1. *Events e_1 and e_2 are l-*atomic*, written $e_1 \Updownarrow_l e_2$, if and only if there is some event e such that $type(e) = acquire$, $target(e) = l$, $e < e_1$, $e < e_2$, and there is no e' with $type(e') = release$, $target(e) = l$, and $e < e' < e_1$ or $e < e' < e_2$. For each lock l, we let $[e]_l$ denote the l-*atomic equivalence class of e.

In Figure 3, $e_{14} \Updownarrow_{lock} e_{15}$. We capture the lock-atomicity as follows. A counter c_l is associated with every lock l. Each thread stores the set of locks that it holds in LS_t. Events are enriched with a new attribute, LS, which is a partial mapping from locks into corresponding counters. When event e is processed, the lock information is updated as follows:

1. if $type(e) = acquire$, $thread(e) = t$, and $target(e) = l$, then $c_l = c_l + 1$, $LS_t = LS_t \cup \{l\}$;
2. if $type(e) = release$, $thread(e) = t$, and $target(e) = l$, then $LS_t = LS_t - \{l\}$.
3. $LS(e)(l) = c_l$ for all $l \in LS_{thread(e)}$ and $LS(e)(l)$ undefined for any other l;

If we write $LS(e)(l) = LS(e')(l)$ then we mean that the two are *defined and equal*. The following important result holds:

THEOREM 3. *$e \Updownarrow_l e'$ iff $LS(e)(l) = LS(e')(l)$.*

Intuitively, a permutation of events is consistent, (or sound, or realizable, or possible during an execution) if and only if it preserves *both* the sliced causality and the lock-atomicity relation above.

DEFINITION 2. *A **cut** Σ is a set of events in τ. Σ is **consistent** if and only if for all $e, e' \in \tau$,*
(a) if $e' \in \Sigma$ and $e \prec e'$ then $e \in \Sigma$; and
(b) if $e, e' \in \Sigma$ and $e' \notin [e]_l$ for a lock l, then $[e']_l \subseteq \Sigma$ or $[e]_l \subseteq \Sigma$.

The first item says that for any event in Σ, all the events upon which it depends should also be in Σ. The second property states that there is *at most one* incomplete lock-atomic set for every particular lock l in Σ. Otherwise, the lock-atomicity is broken. Essentially, Σ contains the events in the prefix of a consistent permutation. When an event e can be added to Σ without breaking the consistency, e is called *enabled* for Σ.

DEFINITION 3. *Event $e' \in \tau - \Sigma$ is **enabled** for consistent cut Σ iff*
(a) for any event $e \in \tau$, if $e \prec e'$ then $e \in \Sigma$; and
(b) for any $e \in \Sigma$ and any lock l, either $e' \in [e]_l$ or $[e]_l \subseteq \Sigma$.

Hence, e is enabled for consistent cut Σ iff $\Sigma \cup \{e\}$ is also consistent.

```
globals ξ_φ ← φ-sliced trace, CurrentLevel ← {Σ_0...0}
procedure main()
  while (ξ_φ ≠ ∅) do verifyNextLevel()
endprocedure
procedure verifyNextLevel()
  local NextLevel ← ∅
  for all e ∈ ξ_φ and Σ ∈ CurrentLevel do
    if enabled(Σ, e) then NextLevel ← NextLevel ∪ createCut(Σ, e)
  CurrentLevel ← NextLevel
  ξ_φ ← removeRedundantEvents()
endprocedure
procedure enabled(Σ, e)
  return VC(e)(thread(e)) = VC(Σ)(thread(e)) + 1   and
         VC(e)(t) ≤ VC(Σ)(t) for all t ≠ thread(e)   and
         LS(e)(l) = LS(Σ)(l) when both defined, for all locks l
endprocedure
procedure createCut(Σ, e)
  Σ' ← new copy of Σ
  VC(Σ')(thread(e)) ← VC(Σ)(thread(e)) + 1
  if type(e) = acquire and target(e) = l then LS(Σ')(l) ← LS(e)(l)
  if type(e) = release and target(e) = l then LS(Σ')(l) ← undefined
  MS(Σ') ← runMonitor(MS(Σ), e)
  if MS(Σ') = "error" then reportViolation(Σ, e)
  return Σ'
endprocedure
```

Figure 4: Consistent runs generation algorithm

DEFINITION 4. *A **consistent permutation** $e_1e_2...e_{|\tau|}$ of τ is one that generates a sequence of consistent cuts $\Sigma_0\Sigma_1...\Sigma_{|\tau|}$: for all $1 \leq r \leq |\tau|$, Σ_{r-1} is consistent, e_r is enabled for Σ_{r-1}, and $\Sigma_r = \Sigma_{r-1} \cup \{e_r\}$.*

The following result holds and shows the soundness of lock-atomicity:

THEOREM 4. *Any consistent permutation of τ corresponds to some possible execution of the multithreaded system.*

The proof is non-trivial and can be done by extending the proof of Theorem 1 (using the parametric framework for causality like in [7]) to incorporate lock-atomicity.

COROLLARY 2. *Consider now our original trace ξ together with its φ-sliced trace ξ_φ. Then any permutation of property events that is consistent with the sliced causality and the lock-atomicity corresponds to some possible execution of the multi-threaded system.*

2.4 Generating Potential Runs

We here discuss an algorithm to check all the consistent permutations of events against the desired property φ. The actual permutations of events are *not* generated, because that would be prohibitive. Instead, a monitor is assumed for the property φ which is run synchronously with the generation of the next level in the computation lattice, following a breadth-first strategy. Figure 4 gives a high-level pseudocode to generate and verify, on a level-by-level basis, potential runs consistent with the sliced causality with lock-atomicity. ξ_φ is the set of relevant events. *CurrentLevel* and *NextLevel* are sets of cuts. We encode cuts Σ as: a $VC(\Sigma)$ which is the max of the *VCs* of all its threads (updated as shown in procedure *createCut*); a partial mapping $LS(\Sigma)$ which keeps for each lock l its current counter c_l (updated as also shown in *createCut*); and the current state of the property monitor for this run, *MS*. The property monitor can be any program, in particular those generated automatically from specifications, like in MOP [6].

Figure 5 shows a simple example for generating consistent permutations. Figure 5 (A) is an observed execution of a two-thread program. The solid arrow lines are threads, the dotted arrows are dependencies, and the dotted boxes show the scopes of synchronized blocks. Both synchronized blocks are protected by the same

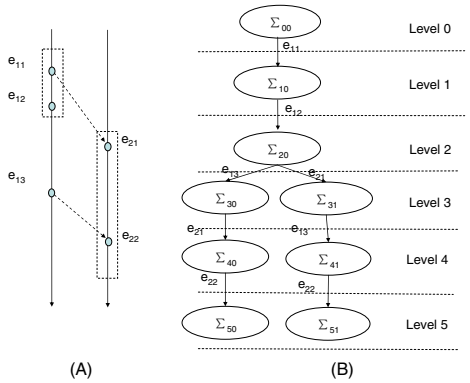

Figure 5: Example for consistent run generation

lock, and all events marked here are relevant. Figure 5 (B) illustrates a run of the algorithm in Figure 4, where each level corresponds to a set of cuts generated by the algorithm. The labels of transitions between cuts give the added events. Initially, there is only one cut, Σ_{00}, on Level 0. The algorithm first checks every event in ξ_φ and every cut in the current level to generate cuts of the next level by appending enabled events to current cuts. The *enabled* procedure implements the definition of a consistent permutation (compares the *VCs* between a candidate event and a cut, then checks for the compatibility of their lock-atomicity). For example, only e_{11} is enabled for the initial cut, Σ_{00}; e_{12} is enabled for Σ_{10} on Level 1, but e_{21} is not because of the lock-atomicity. On Level 2, after e_{11} and e_{12} have been consumed, e_{21} and e_{13} are both enabled for the cut Σ_{20}. If an event e is enabled for a cut Σ, e is added to Σ to create a new cut Σ', as depicted by the transitions in Figure 5. The vector clocks and lock set information of Σ' will be computed according to e. After the next level is generated, redundant events, e.g., e_{11} after Level 1, will be removed from ξ_φ. Also, the property monitor in Σ will be run (see the *createCut* procedure) and its new state stored in Σ'; violations are reported as soon as detected.

The pseudocode in Figure 4 glossed over many implementation details that make it efficient. For example, ξ_φ can be stored as a set of lists, each corresponding to a thread. Then the *VC* of a cut Σ can be seen as a set of pointers into each of these lists. The potential event e for the loop in *verifyNextLevel* can only be among the next events in these lists. The function *removeRedundantEvents()* eliminates events at the beginning of these lists when their *VCs* are found to be smaller than or equal to the *VCs* of all the cuts in the current level. In other words, to process an event, a good implementation of the algorithm in Figure 4 would take time $O(|Threads|)$.

3. JPREDICTOR

JPREDICTOR is a runtime analysis tool to detect concurrent bugs in Java programs using sliced causality with lock-atomicity. In addition to an efficient implementation of the vector-clock-based algorithm discussed above, JPREDICTOR also provides an optimal instrumentation framework to log and replay program execution, as well as specialized property checkers for data races and atomicity. Interested readers can find more information on JPREDICTOR at its website [16], where it is also available for download.

3.1 Architecture

JPREDICTOR is composed of two major components: the program instrumentor and the trace predictor (Figure 6). The program instrumentor instruments the program under testing with instructions that log the execution. To reduce the runtime overhead caused by monitoring, only partial information is logged during execution. The trace predictor analyzes the logged execution trace to predict potential bugs using sliced causality. If a possible bug is

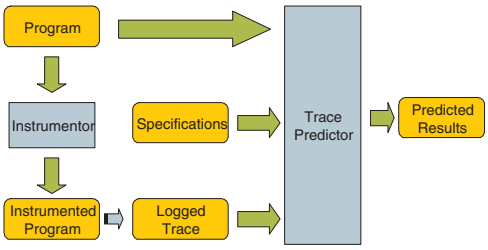

Figure 6: Working Architecture of jPredictor

detected, jPredictor generates an abstract execution trace leading to it, which explains how the bug can be hit in a real execution.

As shown in Figure 7, the trace predictor consists of four stages: the pre-processor, the trace slicer, the *VC* calculator, and the property checker. The role of the pre-processor is two-fold. First, it constructs a more informative trace from the partially logged trace using static analysis on the original program, providing a foundation for the subsequent analysis. Second, it identifies all the shared locations in the observed execution, which are critical for a precise predictive analysis. The slicer scans the re-constructed trace, producing a trace slice for every property to check. The generated slices are fed into the VC calculator, which computes the sliced causality as discussed in Section 2.2. In the last stage, the property checker verifies the execution against the desired property using the computed sliced causal with lock-atomicity. All these stages communicate using plain ASCII text, making the tool easy to extend. For example, we also implemented a conventional happen-before slicer to generate trace slices containing all the shared variable accesses. This way, we were able to compute the traditional happen-before causality without changing any other components of jPredictor; it has been used in [7] for comparison purposes. We next give more details about each component of jPredictor.

3.2 Partial Monitoring

Program monitoring plays a fundamental role in predictive runtime analysis. For sliced causality, *complete* monitoring, i.e., observing every instruction of the program, is desired for an accurate data-flow analysis. However, such monitoring imposes huge runtime overhead that we want to avoid in practice. An important observation here is that, by using static analysis, one can replay the execution *with much less observation* of the program. The more complicated the static analysis, the fewer observation points and the less monitoring overhead one can achieve. For example, one could symbolically execute the program and only need runtime information when the symbolic execution cannot decide how to proceed. How to achieve minimum but sufficient information by runtime monitoring in order to replay an execution is an interesting question by itself, but out of the scope of this paper. In what follows, we briefly discuss an effective solution adopted by jPredictor, which aims at reducing the monitoring overhead with relatively simple static analysis.

Two components of jPredictor are involved in obtaining a complete trace via partial monitoring, namely the program instrumentor and the pre-processor of the trace predictor (Figures 6 and 7). The program instrumentor, built on top of Soot [24], a Java bytecode engineering package, is used to insert logging instructions into the original program. Three kinds of program points are observed: beginnings of methods, targets of conditionals, and accesses to objects/arrays (i.e., field/element access or method invocations). To faithfully replay an execution, we need to know which method implementation was actually executed when a method invocation is encountered. Because of the polymorphism and the virtual method mechanism of Java, it can be difficult to identify the actual target

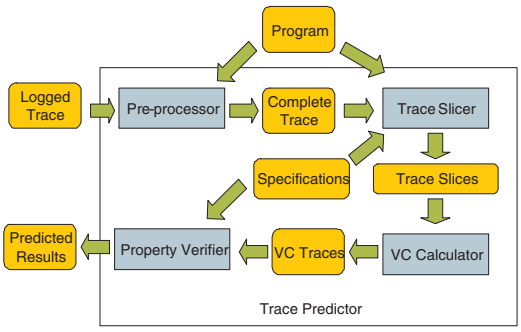

Figure 7: Staged Architecture of Trace Predictor

method implementation at a specific program point by static analysis, while logging the entry of the method at runtime is a simple and precise solution. Similarly, static conditional analysis is often difficult and imprecise but can be totally avoided using runtime information. For object accesses, static aliasing analysis is usually expensive and often imprecise in the absence of runtime information. Admittedly, monitoring every object/array access is fairly heavy and can be improved by advanced aliasing analysis, but we leave that to future research; the current solution yielded reasonable runtime overhead in our experiments.

The information logged for every event is as follows. Each event should carry along the id of the thread which issued it. In addition, the begin method event records a full signature of the method, so that we can locate the actual method implementation later; the branch target event needs to store the line number of the target; the object/array access event contains the id of the accessed object/array and, for the array access, the index of the element. Instead of analyzing this partial trace directly to compute the sliced causality, jPredictor first re-constructs a complete trace out of it using the pre-processor. The constructed complete trace can be reused by different trace slicers to verify different properties. Also, the trace slicer becomes simpler and independent of the instrumentation strategy. The pre-processor also works at the Java bytecode level. It goes through the instructions of the program following the control flow information recorded in the logged trace, supplementing the trace with useful information ignored by the monitoring. More specifically, the following information is added into the trace: return points of method executions, origins of jump statements, field names of object accesses, starts of threads, and lock/unlock events. The first two are needed because the slicing is performed in a backward manner; field names are used to achieved fine grained data analysis; thread start events and lock-related events are needed to compute correct VCs and lock-atomicity.

It is impossible or undesired to instrument all the methods used in a program, e.g., the (native) Java library functions. jPredictor by default does not instrument any Java library function unless requested by the user. However, without further knowledge, the data-flow dependence analysis of jPredictor may lose information and produce imprecise results. For example, when the *ArrayList.addAll(Collection c)* is called, the target *ArrayList* object will be changed according to the input argument *c*. A conservative solution is to assume all the instrumented methods *impure*, i.e., they may change the target object and any of the input arguments. This assumption is often too restrictive, because many functions simply return the state of the object. Hence, jPredictor allows the user to input purity information about functions that are used but not instrumented in the program. If no information is provided for a certain function, the function will be considered impure. The purity information can be reused for different programs and we have pre-defined the purity of many Java library functions; that was sufficient to do our experiments entirely automatically.

3.3 Trace Slicing and VC Calculator

The trace slicer implements the dynamic slicing algorithm described in Section 2.2.2 to extract property-specific trace slices from the completed trace. jPredictor handles all thread-related, e.g., *start* and *join*, and all synchronization events specially, according to the Java semantics. Also, for efficiency purposes, only one pass of the backward slicing is performed in the present implementation. This may result in a trace slice that does not contain all the relevant events and thus lose the soundness of the predictive analysis. In other words, the current jPredictor prototype may produce false alarms due to incomplete trace slices. However, this deliberately unsound implementation proved to be sufficiently effective in our evaluation: no false alarm has been reported in our experiments.

The trace slice is then used by the *VC* calculator to compute *VC*s based on the algorithm described in Section 2.2.2. Similarly to the slicer, Java-specific language features are specially handled during the *VC* computation to ensure the correctness and accuracy of the results, for example, the beginning of a thread execution should depend on the corresponding thread creation event. The output of the *VC* calculator is a sequence of property events associated with *VC*s and lock-atomicity information. This output is then verified against the desired property as explained below.

3.4 Verifying Properties

The property checker of jPredictor implements the algorithm in Figure 5 to generate consistent permutations of property events, providing a generic way to verify temporal properties against the observed execution using the sliced causality. In other words, jPredictor is not bound to any particular type of property: one can hook up any property monitor to jPredictor to predict possible violations of the desired property which can be specified using any trace specification language, e.g., regular expressions, temporal properties, or context-free languages. This way, combining it with the automatic monitor generation framework provided by JavaMOP [6], jPredictor gives an automated platform to formally specify and dynamically verify trace properties against concurrent Java programs.

Unfortunately, generating all the consistent permutations of a partial order is a #P-complete problem [4] and can be unnecessarily expensive for those properties for which we can have more efficient solutions. jPredictorprovides two specialized checkers to detect data races and atomicity violations efficiently using sliced causality and lock-atomicity. We next discuss both of them briefly.

First, we need to determine the property events for these two types of properties: the property events for detecting races of a specific memory location (i.e., the same object field or array element) are all the writes/reads of the memory location; the property events for analyzing the atomicity of atomic blocks are all the accesses of shared locations used within those blocks. Let $\prec^x_{race}$ be the sliced causality for detecting the data race of the shared location x. We then formally define a data race as follows:

Definition 5. *For two events, e_1 and e_2, if they access the same memory location x and at least one of them is a write, then we say that they cause a* data race *on x iff e_1 and e_2 are not comparable under $\prec^x_{race}$ and they are not protected by the same lock.*

Therefore, the data race can be detected by comparing events generated in different threads. In our implementation, the property events, writes/reads of the shared variable in this case, are processed following the order in the logged trace (i.e., the order in the original execution). When a new property event is processed, it is checked against those events processed in other threads using the above race condition. The complexity of this algorithm is square to the number of property events.

Thanks to the genericity of jPredictor with regards to properties, one is allowed to use existing algorithms, e.g., the reduction based algorithm in [12] or the causality based algorithm in [11], to check the atomicity on the consistent permutations generated by jPredictor. The specialized efficient algorithm implemented in jPredictor to identify atomicity violations is based on the problematic scenarios proposed in [26]. In short, jPredictor first constructs all the atomic blocks from the execution trace; then each pair of blocks generated in different threads is examined to see if any of the 11 violation patterns in [26] can be matched under the sliced causality and lock-atomicity constraints. Since those patterns involve at most two different variables, the complexity of checking each pair is $O(m^2)$, where m is the number of events in both blocks. The worse case complexity of this atomicity algorithm is therefore $O(n^4)$, where n is the number of all the property events.

When both data races and atomicity are checked, the analysis cost can be further reduced by reusing the trace slices generated for race detection in the atomicity analysis. More specifically, if an atomic block B contains accesses of shared variables $x_1, ...x_i$, the trace slice for checking the atomicity of B is the union of the slices for checking races on $x_1, ..., x_i$. The formal proof of the correctness is left out of this paper. Intuitively, for two sets of events, E_1 and E_2, if trace slices ξ_1 and ξ_2 contain all the events affecting E_1 and E_2, respectively, then $\xi_1 \cup \xi_2$ should also contain all the events affecting $E_1 \cup E_2$. This way, one does not need to re-slice the trace for atomicity analysis if race detection has been carried out. As our experiments show, merging trace slices is much cheaper than generating slices (Section 4.3). So the analysis performance can be significantly improved by reusing existing slices.

Program	LOC	Threads	S.V.	Slowdown
Banking	150	3	10	0.34
Elevator	530	4	123	N/A
tsp	706	4	648	7.05
sor	17.7k	4	102	0.47
hedc	39.9k	10	119	0.56
StringBuffer	1.4k	3	7	0.61
Vector	12.1k	18	49	0.79
IBM Web Crawler	unknown	7	76	0.01
StaticBucketMap	748	6	381	35.6
Pool 1.2	5.8k	2	119	0.29
Pool 1.3	7.0k	2	95	0.32
Apache Ftp Server	22.0k	12	281	N/A
Tomcat Components	4.0k	3	13	0.1

Table 1: Benchmarks

4. EVALUATION

Here we present evaluation results of jPredictor on two types of common and well-understood concurrency properties, which need no formal specifications to be given by the user and whose violation detection is highly desirable: dataraces and atomicity. jPredictor has also been tried on properties specified formally and monitors generated using the MOP [6] logic-plugins, but we do not discuss those here; the interested reader is referred to [5]. We discuss some case studies, showing empirically that the proposed predictive runtime verification technique is viable and that the use of sliced causality significantly increases the predictive capabilities of the technique. All experiments were performed on a 2.6GHz X2 AMD machine with 2GB memory. Interested readers can find detailed result reports on jPredictor's website at [16].

4.1 Benchmarks

Table 1 shows the benchmarks that we used, along with their size (lines of code),[2] number of threads created during their execution,

[2]Different papers give different numbers of lines of code for the

Program	Var to Check	Trace Size		Running Time (seconds) per S.V.				Races		
		Logged	Complete	Preprocess	Slice	VC	Verify	Harmful	Benign	False
Banking	10	244	320	0.01	0.04	0.01	0.01	1	0	0
Elevator	48	62314	71269	1.0	8.1	1.2	0.15	0	0	0
tsp	47	141239	237801	2.2	26.1	2.3	0.23	1	0	0
sor	17	10968	12654	0.3	1.9	0.2	0.01	0	0	0
hedc	43	128289	183317	2.1	17.9	0.16	0.01	4	0	0
StringBuffer	4	738	871	0.06	0.28	0.05	0.01	0	0	0
Vector	47	876	1086	0.08	0.3	0.06	0.01	0	1	0
IBM Web Crawler	59	3128	3472	0.18	0.6	0.16	0.01	1	3	0
StaticBucketMap	39	319482	366743	7.6	131.6	12.2	0.03	1	0	0
Pool 1.2	54	20541	24072	0.26	1.42	0.34	0.01	35	0	0
Pool 1.3	45	1426	1669	0.16	0.76	0.23	0.01	1	0	0
Apache FTP Sever	71	19765	20047	0.69	3.87	0.34	0.02	11	5	0
Tomcat Components	13	3240	3698	0.21	0.62	0.2	0.01	2	2	0

Table 2: Race detection results

number of shared variables (S.V.) detected, and slowdown ratios after instrumentation [3]. Banking is a simple example taken over from [10], showing relatively classical concurrent bug patterns. Elevator, tsp, sor and hedc come from [27]. Elevator is a discrete event simulator of an elevator system. tsp is a parallelized solution to the traveling salesman problem. sor is a scientific computation application synchronized by barriers instead of locks. hedc is an application developed at ETH that implements a meta-crawler for searching multiple Internet achieves concurrently.

StringBuffer and Vector are standard library classes of Java 1.4.2 [15]. IBM web crawler is a component of the IBM Websphere tested in [9]. [4] StaticBucketMap, Pool 1.2 and 1.3 are part of the Apache Commons project [2]: StaticBucketMap is a thread-safe implementation of the Java Map interface; Pool 1.2 and 1.3 are two versions of the Apache Commons object pooling components. Apache FTP server [3] is a pure Java FTP server designed to be a complete and portable FTP server engine solution. Tomcat [25] is a popular open source Java application server. The version used in our experiments is 5.0.28. Tomcat is so large, concurrent, and has so many components, that it provides a base for almost unlimited experimentation all by itself. We only tested a few components of Tomcat, including the class loaders and logging handlers.

For most programs, we used the test cases contained in the original packages. The Apache Commons benchmarks, i.e., StaticBucketMap and Pool 1.2/1.3, provide no concurrent test drivers, but only sequential unit tests. We manually translated some of these into concurrent tests by executing the tests concurrently and modifying the initialization part of each unit test method to use a shared global instance. For StringBuffer and Vector, some simple test drivers were implemented, which simply start several threads at the same time to invoke different methods on a shared global object. The present implementation of jPredictor tracks accesses of array elements, leading to the large numbers of shared variables and significant runtime overhead in tsp and StaticBucketMap. For other programs, the runtime overhead is quite acceptable.

Each test was executed *precisely once* and the resulting trace has been analyzed. While multiple runs of the system, and especially combinations of test case generation and random testing with predictive runtime analysis would almost certainly increase the coverage of predictive runtime analysis and is worth exploring in depth, our explicit purpose in this paper is to present and evaluate predictive runtime analysis based on sliced causality *in isolation*. Careful inspection of the evaluation results revealed that the known bugs that were missed by jPredictor were missed simply because of limited test inputs: their corresponding program points were not touched during the execution. Any dynamic analysis technique suffers from this problem. Our empirical evaluation of jPredictor indicates that the use of sliced causality in predictive runtime analysis makes it less important to generate "bad" thread interleavings in order to find concurrent bugs, but more important to generate test inputs with better code coverage.

4.2 Race Detection

The results of race detection are shown in Table 2. The second column gives the number of shared variables checked in the analysis, which is in some cases smaller than the number of shared variables in Table 1 for the following reasons. Some shared variables were introduced by the test drivers and therefore not needed to check. Also, as already mentioned, many shared variables are just different elements of the same array and it is usually redundant to check all of them. jPredictor provides options to turn on an automatic filter that removes undesired shared variables (using class names) or randomly picks only one element in each array to check. This filter was kept on during our experiments, resulting in fewer shared variables to check. The third and the fourth columns report the size of the trace (i.e., the number of events) logged at runtime and the size of the trace constructed after preprocessing, respectively. The difference between these shows that, with the help of static analysis, the number of events to log at runtime is indeed reduced, implying a reduction of runtime overhead.

Columns 5 to 8 show the times used in different stages of the race detection. Because jPredictor needs to repeat the trace slicing, the *VC* calculation, and the property checking for every shared variable, the times shown in Table 2 for these three stages are the average times for one shared variable. Considering the analysis process is entirely automatic, the performance is quite reasonable. Among all the four stages, the trace slicing is the slowest, because it is performed on the complete trace. In spite of its highest algorithmic complexity, the actual race detection is the fastest part of the process. This is not unexpected though, since it works on the sliced trace containing only the property events, which is much shorter than the complete one.

The last section of Table 2 reports the number of races detected in our experiments. The races are categorized into three classes:

same program due to different settings. In our experiments, we counted those files that were instrumented during the testing, which can be more than the program itself. For example, the kernel of hedc contains around 2k lines of code; but some other classes used in the program were also instrumented and checked, e.g., a computing library developed at ETH. This gave us a much larger benchmark than the original hedc.

[3]Not applicable for some programs, e.g., Elevator.

[4]No source code is available for this program.

```
if ((entry == null) || (entry.binaryContent == null)
   && (entry.loadedClass == null))
     throw new ClassNotFoundException(name);

Class clazz = entry.loadedClass;
if (clazz != null)  return clazz;
```

Figure 8: Buggy code in WebappClassLoader

harmful, benign (do not cause real errors in the system) and false (not real races). JPREDICTORreported *no false alarms* and, for all the examples used in other works except for the FTP server, e.g., hedc and Pool 1.2, it *found all the previously known dataraces*. Note that we only count the races on the same field once, so our numbers in Table 2 may appear to be smaller than those in other approaches that use the number of unsafe access pairs. Some races in the FTP server reported in [18] were missed by JPREDICTOR because the provided test driver is comparatively simple and preformed limited testing of the server, avoiding the execution of the buggy code.

Surprisingly, JPREDICTOR found some races in Pool 1.2 that were missed by the static race detector in [18], which is expected to have a very comprehensive coverage of the code (at the expense of false alarms). JPREDICTOR also reported some unknown harmful races in StaticBucketMap, Pool 1.3 and Tomcat. The race in StaticBucketMap is caused by unprotected accesses to the internal nodes of the map via the *Map.Entry* interface. It leads to a harmful atomicity violation, explained in more detail in the next subsection. Although Pool 1.3 fixed all the races found in Pool 1.2, JPREDICTOR still detected a race when an object pool is closed: in GenericObjectPool, a concrete subclasses of the abstract BaseObjectPool class, the close process first invokes the close function in the super class *without* proper synchronization. Hence, other methods can interfere with the close function, leading to unexpected exceptions.

For Tomcat, JPREDICTOR found four dataraces: two of them are benign and the other two are real bugs. Our investigation showed that they have been previously submitted to the bug database of Tomcat by other users. Both bugs are hard to reproduce and only rarely occur, under very heavy workloads; JPREDICTOR was able to catch them using only a few working threads. More interestingly, one bug was claimed to be fixed, but when we tried the patched version, the bug was still there. Let us take a close look at this bug.

This bug resides in *findClassInternal* of org.apache.catalina.loader.WebappClassLoader. This bug was first reported by JPREDICTOR as dataraces on variables *entry.binaryContent* and *entry.loadedClass* at the first conditional statement in Figure 8. The race on *entry.loadedClass* does not lead to any errors, and the one on *entry.binaryContent* does no harm by itself, but *together* they may cause some arguments of a later call to *definePackage(packageName, entry.manifest, entry.codeBase)*[5] to be null, which is illegal. It seems that a Tomcat developer tried to fix this bug by putting a lock around the conditional statement, as shown in Figure 9. However, JPREDICTOR showed that the error still exists in the patched code, which was a part of the latest version of Tomcat 5 when we carried out our experiments. We reported the bug with a fix and it has been accepted by the Tomcat developers.

4.3 Atomicity Violation Detection

The results of evaluating JPREDICTOR on atomicity analysis are shown in Table 3. Although JPREDICTOR allows the user to define different kinds of atomic blocks, we only checked for the atomicity of methods in these experiments. Not all benchmarks were checked: we do not have enough knowledge of the IBM Web Crawler to judge atomicity (its source code is not public), while method

[5]There is another *definePackage* function with eight arguments that allows null arguments.

```
if (entry == null)
    throw new ClassNotFoundException(name);
Class clazz = entry.loadedClass;
if (clazz != null)  return clazz;
synchronized (this) {
if (entry.binaryContent == null && entry.loadedClass == null)
   throw new ClassNotFoundException(name);
}
```

Figure 9: Patched code in WebappClassLoader

Program	Running Time (seconds)			Violations	
	Slice	VC	Verify	Actual	False
Banking	0.01	0.01	0.01	1	0
Elevator	0.4	3.2	0.6	0	0
tsp	0.5	2.5	0.6	1	0
sor	0.1	0.6	0.46	0	0
hedc	0.02	0.18	0.02	1	0
StringBuffer	0.01	0.05	0.01	1	0
Vector	0.06	0.15	0.06	4	0
StaticBucketMap	8	14	0.03	1	0
Pool 1.2	0.23	1.87	3.4	10	0
Pool 1.3	0.19	0.61	0.03	0	0

Table 3: Atomicity analysis results

atomicity is not significant for FTP and Tomcat, since their methods are complex and usually not atomic (finer grained atomic blocks are more desirable there, but this is beyond our purpose in this paper).

We do not need to repeat the pre-processing stage for atomicity analysis. Hence, only the times for slicing, *VC* calculation and atomicity checking are shown in columns 2 to 4 in Table 3. As discussed in Section 3.4, our evaluation of atomicity analysis reused the trace slices generated for race detection to reduce the slicing cost, which turned out to be effective according to the results. The other two stages took more time in atomicity analysis than in race detection because the analyzed trace slice was larger. The last part of Table 3 shows the number of detected atomicity violations, which are divided into two categories: actual violations and false alarms. No benign violations were found in our evaluation, probably because the definition of atomicity that we adopted is based on problematic patterns of event sequences.

JPREDICTOR *did not report any atomicity false alarm* in its analysis. It also *found all the previously known harmful atomicity violations* in the examples also analyzed by other approaches, e.g., [28] and [12]. Moreover, JPREDICTOR found harmful atomicity violations in tsp and hedc that were missed by [28] and [12] using the same test drivers. This indicates that JPREDICTOR, through its combination of static dependence analysis and sliced causality, provides a better capability of predicting atomicity violations. Some unknown violations in StaticBucketMap and Pool 1.2 were also detected. We next briefly explain the violation in StaticBucketMap.

In StaticBucketMap, fine grained internal locks are used to provide thread-safe map operations. Specifically, every bucket in the map is protected by a designated lock. A data race was still detected by JPREDICTOR in this well synchronized implementation, caused by the usage of the *Map.Entry* interface. As shown in Figure 10, one can obtain a map entry, which represents a key-value pair, via an iterator of the map and use the *setValue* method to change the entry. JPREDICTOR showed that the *setValue* method is not correctly synchronized and causes a data race. This data race is benign in most cases, because no new entry can be added or removed through the *Map.Entry* interface and also because the bulk operations of the

```
StaticBucketMap map;
...
Map.Entry entry = (Map.Entry)map.entrySet().iterator().next();
entry.setValue(null);
```

Figure 10: Unprotected modification of the map entry

```
class MapPrinter implements Runnable{
  public void run(){
    Iterator it = map.entrySet().iterator();
    while (it.hasNext()) {
       Map.Entry entry = (Map.Entry)it.next();
       if (entry.getValue() != null)
         System.out.println(entry.getValue().toString());
    }
  }
  public void atomicPrint(){
    map.atomic(this);
  }
}
```

Figure 11: Atomic iteration on the map

map, e.g., iteration, are not guaranteed to be atomic. However, StaticBucketMap provides an *atomic(Runnable r)* method to support atomic bulk operations. This method accepts a *Runnable* object and executes the *run()* method of the object atomically with regards to the map. Figure 11 shows an example of using this method to print out all the values in the map atomically. However, this atomicity guarantee can be violated when another thread accesses the map's elements using the unsafe *setValue* method, like the code in Figure 11, which can cause an unexpected null pointer exception. jPredictor detected this violation (without directly hitting it during the execution) in our experiments, generating a warning message that clearly points out the cause of the violation.

5. CONCLUSION

A novel predictive runtime analysis technique was presented, which employs sliced causality and lock-atomicity to improve the coverage of concurrent program testing. This technique has been implemented in the jPredictor tool, and was evaluated using jPredictor on several non-trivial Java applications for race detection and atomicity analysis. The results of our experiments show that our predictive runtime analysis technique implemented in jPredictor is very precise and effective. No false alarms were reported in any of our experiments, and several unknown bugs were found. It is also surprising to see that jPredictor achieved results that are as comprehensive as the static analysis based approaches in some cases, even though the coverage of the runtime analysis is restricted by the limited coverage of the test input. The runtime overhead of trace logging and the analysis time of jPredictor were reasonable in our experiments. The runtime analysis process is fully automated and its setting requires very little human interference. There are several interesting future directions to pursue, such as increasing the predictive capability by strengthening the static analysis part, or investigating test case generation and random testing techniques to generate causally orthogonal executions.

6. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*, 1990.

[2] Apache Commons project. http://commons.apache.org/.

[3] Apache FTP server project. incubator.apache.org/ftpserver/.

[4] G. Brightwell and P. Winkler. Counting linear extensions is #p-complete. In *ACM symposium on Theory of computing (STOC'91)*, 1991.

[5] F. Chen and G. Roşu. Predicting concurrency errors at runtime using sliced causality. Technical Report UIUCDCS-R-2005-2660, Department of CS at UIUC, 2005.

[6] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, 2007.

[7] F. Chen and G. Roşu. Parametric and Sliced Causality. In *Computer Aided Verification (CAV'07)*, 2007.

[8] F. Chen and G. Roşu. Parametric and termination-sensitive control dependence - extended abstract. In *Static Analysis Symposium (SAS'06)*, 2006.

[9] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[10] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[11] A. Farzan and M. Parthasarathy. Causal atomicity. In *Computer Aided Verification (CAV'06)*, 2006.

[12] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Principles of Programming Languages (POPL'04)*, 2004.

[13] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Static Analysis Symposium (SAS'99)*, 1999.

[14] S. Horwitz and T. W. Reps. The use of program dependence graphs in software engineering. In *International Conference on Software Engineering (ICSE'92)*, 1992.

[15] Java. http://java.sun.com.

[16] jPredictor. http://fsl.cs.uiuc.edu/jPredictor/.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of ACM*, 21(7):558–565, 1978.

[18] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. *ACM SIGPLAN conference on Programming language design and implementation (PLDI'06)*, 2006.

[19] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, 2003.

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transaction of Computer System*, 15(4):391–411, 1997.

[21] A. Sen and V. K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *International Conference on Principles of Distributed Systems (OPODIS'03)*, 2003.

[22] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'03)*, 2003.

[23] K. Sen, G. Roşu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, 2005.

[24] Soot website. http://www.sable.mcgill.ca/soot/.

[25] Apache group. Tomcat. http://jakarta.apache.org/tomcat/.

[26] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Principles of Programming Languages (POPL'06)*, 2006.

[27] C. von Praun and T. R. Gross. Object race detection. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, 2001.

[28] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, 2006.