University of Illinois at Urbana-Champaign
Department of Computer Science Technical Report UIUCDCS-R-2006-2965

Predicting Concurrency Errors at Runtime using Sliced Causality

Feng Chen
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
fengchen@cs.uiuc.edu

Grigore Roşu
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
grosu@cs.uiuc.edu

ABSTRACT

A runtime analysis technique is presented, which can predict concurrency errors in multithreaded systems by observing potentially non-erroneous executions. It builds upon a novel causal partial order, *sliced causality*, that weakens the classic but strict "happens-before" by using both static information about the program, such as control- and data-flow dependence, and dynamic synchronization information, such as lock-sets. A vector clock algorithm is introduced to automatically extract a sliced causality from any execution. A memory-efficient procedure then checks all causally consistent *potential runs* against properties given as *monitors*. If any of these runs violates a property, it is returned as a "predicted" counter-example. This runtime analysis technique is *sound* (no false alarms) but *not complete* (says nothing about code that was not covered). A prototype called jPredictor has been implemented and evaluated on several Java applications with promising results.

1. INTRODUCTION

Concurrent systems in general and multi-threaded systems in particular may exhibit different behaviors when executed at different times. This inherent nondeterminism makes multi-threaded programs difficult to analyze, test and debug. This paper introduces a technique to correctly detect concurrency errors from observing execution traces of multithreaded programs. The program is automatically instrumented to emit "more than the obvious" information to an external observer, by means of runtime events. The particular execution that is observed needs *not* hit the error; yet, errors in other executions can be predicted *without false alarms*. The observer, which can potentially run on a different machine, will never need to see the code which generated those events but still be able to correctly predict errors that may appear in other executions, and report them to the user as counter-example executions.

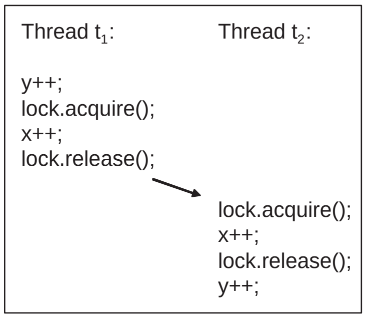
There are several other approaches also aiming at detecting potential concurrency errors by examining particular execution traces. Some of these approaches aim at verifying general purpose behavioral properties [18, 20], including temporal ones, and are inspired from debugging of distributed systems based on Lamport's *happens-before* causality [11]. Other approaches are based on *lock-sets* [17, 6] and work with particular properties, such as data-races and/or atomicity. These previous efforts focus on either soundness or coverage: those based on happens-before are sound but have limited coverage over interleavings, thus resulting in more false negatives (missing errors); lock-set based approaches produce fewer false negatives but suffer from false positives (false alarms). There are works combining happens-before and lock-set techniques, e.g., [14], but currently these support only particular properties (data-races) and do not use static information to increase coverage.

Our runtime analysis technique aims at improving coverage 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., and combines a general novel happens-before relation, called *sliced causality*, with lock-sets. Based on an apriori static analysis, sliced causality drastically cuts the usual happen-before causality on runtime events by removing unnecessary dependencies; this way, a significantly larger number of consistent runs can be inferred and thus analyzed by the observer of the multithreaded execution. Even though we present sliced causality in the context of our predictive runtime analysis application, it can actually be used as an improved causal partial order in other works based on happens-before as well, e.g., those on concurrent test generation [13, 19].

One should not confuse the notion of sliced causality introduced in this paper with the existing notion of *computation slicing* [18]. The two slicing techniques are quite opposed in scope: the objective of computation slicing is to safely *reduce* the size of the computation lattice extracted from a run of a distributed system, in order to reduce the complexity of debugging, while our goal is to *increase* the size of the computation lattice extracted from a run, in order to strengthen the predictive power of our analysis by covering more consistent runs. Computation slicing and sliced causality do not exclude each other. Sliced causality can be used as a front end to increase the coverage of the analysis, while computation slicing can then remove redundant consistent runs from the computation lattice, thus reducing the complexity of analysis. At this moment we do not use computation slicing in our implementation, but it will be addressed soon to improve the performance of our prototype.

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 causal partial order extracted from the running program can be regarded as an abstract model of the program, which can further be investigated exhaustively by the observer. Previous approaches based on happens-before (such as [14, 18, 20]) extract causal partial orders from analyzing *exclusively* the dynamic thread communication in program executions. Since these approaches consider *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. By considering information about the static structure of the multithreaded program in the computation of the causal partial order, we can filter out irrelevant thread interactions and thus obtain a more *relaxed* causality, allowing more consistent runs. Furthermore, we also consider synchronization: events protected by locks can only be permuted *in blocks*. This way, our approach borrows comprehensiveness from lock-set approaches without giving up soundness. Moreover, it is fully generic: the possible linearizations that are consistent with the observed causal partial orders can be checked against *any* monitorable property on execution traces.

Figure 1 shows a canonical example reflecting the limitation of the classic happens-before [17]. When the execution proceeds as indicated by the arrow, the datarace on *y* is masked by the protected accesses to *x*. Our sliced causality technique detects that the updates on *x* and the synchronization operations are *irrelevant* for *y*, so it cuts the happens-before causality accordingly; consequently, our runtime analysis technique correctly *predicts* the datarace on *y* by observing this apparently non-erroneous execution. As mentioned, our technique works for any monitorable property, not only for dataraces, and reports no false alarms.



```
Thread t1:              Thread t2:

y++;
lock.acquire();
x++;
lock.release();
                        lock.acquire();
                        x++;
                        lock.release();
                        y++;
```

Figure 1: Limitation of happens-before

We next explain our runtime analysis technique on an abstract example. Assume the threads and events in Figure 2, where e_1 causally precedes e_2 (e.g., e_1 writes a shared variable and e_2 reads it right afterwards), and the statement generating e'_3 is in the *control scope* (i.e., it control-flow termination-sensitive depends – this notion will be formally defined in Section 2) of the statement generating e_2, while the statement of e_3 is not in the control scope of that of e_2. Then we say that e'_3 *depends on* e_1, but e_3 does *not* depend on e_1, despite the fact that e_1 obviously happened before e_3. The intuition here is that e_3 *would happen anyway*, with or without e_1 happening. Note that this is a dependence partial order *on events*, not on statements. Any permutation of *relevant* events consistent with the intra-thread total order and this dependence corresponds to a valid execution of the multithreaded system. If a permutation violates the property, then the system can do so in another execution. In particular, without any other dependencies but those in Figure 2, the property "e_1 must happen before e_3" (statements in the control scope of e_2 are not relevant for this property) can be violated by the program generating the execution in Figure 2, even though the particular observed run does not! Indeed, there is no evidence in the observed run that e_1 should precede e_3, because e_3 would happen anyway. Note that a purely dynamic "happens-before" approach would *not* work here.

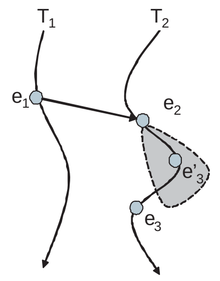


Figure 2: Sliced causality

This paper makes three novel contributions. First, we define *sliced causality*; this can be used as a more informative causality relation in any application based on Lamport's happens-before. Second, we propose a *predictive runtime analysis* technique that can detect property violations in multithreaded systems from successful executions. Third, we discuss a prototype supporting this runtime analysis technique, jPredictor, that has been evaluated on several non-trivial applications with promising results: we were able not only to find errors in large systems, but also to reveal a wrong patch in the latest version of the Tomcat webserver.

2. SLICED CAUSALITY

Sliced causality is a generalized happens-before relation that makes use of dependence information obtained statically and dynamically.

2.1 Control Dependence and Control Scopes

Sliced causality is parametric in a control dependence relation. In particular, one can use off-the-shelf algorithms for classic [5] or for weak [15] control dependence. All we need in order to define our sliced causality is a function returning the *control scope* of any statement C, say *scope* (C): the set of statements whose reachability depends upon the choice made at C, that is, the statements that control-flow depend on C, for some appropriate notion of control dependence. For example, in Figure 3 (A), C_1 decides which of S_1 or S_2 is executed, but does not decide the execution of S_3; so *scope* $(C_1) = \{S_1, S_2\}$ and *scope* (S_1) = *scope* $(S_2) = \emptyset$. Classic control dependence optimistically assumes that all loops terminate, so *scope* $(C_1) = \{S_1\}$ in Figure 3 (B). On the other hand, weak control dependence assumes that all loops may potentially be infinite, so *scope* $(C_1) = \{S_1, S_2\}$ in (B). Since dependence among events can also be indirect, we tacitly consider the transitive closures of control dependencies from here on.

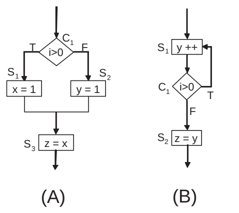


Figure 3: Control dependence

The *soundness* (no false positives) of our runtime analysis technique is contingent to the *correctness* (no false negatives) of the employed static analysis: our runtime analysis produces no false alarms when the *scope* function returns for each statement *at least* all the statements that control-depend on it. An extreme solution is to include all the statements in the program in each scope, in which case sliced causality becomes precisely the classic happens-before relation. As already pointed out in Section 1 and empirically shown in Section 5, such a choice significantly reduces the predictive capability of our runtime analysis technique. A better solution, still over-conservative, is to use weak dependence when calculating the control scopes. One can try to use the classic control dependence instead, but one should be aware that false bugs may be reported (and they typically will: e.g., when analyzing Daisy –see Section 5– which implements synchronization using "infinite" loops).

Ideally, we would want a minimal correct control-dependence; unfortunately, this is impossible, because it would need to statically know which loops terminate and which do not. In [2] we introduced *termination-sensitive control dependence*, a variant of control dependence that is sensitive to the termination information of loops, which can be given as annotations (or conservatively assumed using heuristics, as we do in jPredictor), together with an $O(n^2)$ algorithm to calculate all control scopes for structured languages like Java. If all loops are annotated as terminating then the termination-sensitive control dependence becomes the classic control dependence, while if all loops are annotated as non-terminating then it becomes the weak control dependence. If some loops are annotated as terminating while others not, then the termination-sensitive dependence strictly includes the classic control dependence and is strictly included in the weak one. Thus, one can regard it as a "knob" allowing one to tune the precision anywhere in between the two most widely accepted, but rather extreme control dependence relations. The reader is referred to [2] for technical definitions and details; from here on in this paper we assume a *scope* function that returns (correct) control scopes of statements.

2.2 Events and Traces

Events play a crucial role in our approach, representing atomic steps in the execution of the program. An event can be a write/read on a location, the beginning/ending of a function invocation, acquiring a lock, etc. A statement in the program may produce multiple events. Events need to store enough information about the program state in order for the observer to perform its analysis.

Definition 1. *An **event** is a mapping of **attributes** into corresponding **values**. Let Events be the set of all events. A **trace** is a finite sequence of events. We assume an arbitrary but fixed trace τ, let ξ denote the set of events in τ (also called **concrete events**), and let $<_\tau$ be the total order on ξ: $e <_\tau e'$ iff e occurs before e' in τ.*

For example, one event can be $e_1 : (counter = 8, thread = t_1, stmt = L_{11}, type = write, target = a, state = 1)$, which is a write on location a with value 1, produced at statement L_{11} by thread t_1. One can easily 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 among 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*).

When the trace τ is checked against a property φ, most likely not all the attributes of the events in ξ are needed; some events may not even be needed at all. For example, to check data races on a variable x, the *states*, i.e., the values of x, of the events of type *write* and *read* on x are not important; also, updates of other variables or function call events are not needed at all. We next assume a generic *filtering function* that can be instantiated, usually automatically, to concrete filters depending upon the property φ under consideration:

Definition 2. *Let $\alpha_\varphi : \xi \to Events$ be a partial function, called a **filtering function**. The image of α_φ, that is $\alpha_\varphi(\xi)$, is written more compactly ξ_φ; its elements are called **abstract relevant events**, or simply just **relevant events**.*

Abstraction plays a crucial role in increasing the predictive power of our analysis approach: in contrast to ξ, the abstract ξ_φ allows more permutations of abstract events; instead of calculating permutations of ξ and then abstracting them into permutations of ξ_φ like in [20], we will calculate *directly* valid permutations of ξ_φ. One major goal is to compute the precise causality on abstract events in ξ_φ by analyzing the dependence among concrete events in ξ.

2.3 Hybrid Dependence

Without additional information about the structure of the program that generated the event trace τ, the least restrictive causal partial order that an observer can extract from τ is the one which is total on the events generated by each thread and in which each write event of a shared variable precedes all the corresponding subsequent read events. This is investigated and discussed in detail in [21]. In this section we show that one can do much better than that if one uses control- and data-flow dependence information that can be obtained via static and dynamic analysis of the original program.

The dependence discussed below somehow relates to *program slicing* [8, 22], but we focus on finer grained units here, namely events, instead of statements. Our analysis keeps track of actual memory locations in every event, available at runtime, which avoids inter-procedural analysis and eases the computation of the dependence relation. Also, we need *not* maintain the entire dependence relation, since we only need to compute the causal partial order among events that are relevant to the property to check. This leads to an effective vector clock (*VC*) algorithm (Section 3.1).

Intuitively, event e' *depends upon* event e in τ, written $e \sqsubset e'$, iff a change of e may change or eliminate e'. This tells the observer that *e should occur before e' in any consistent permutation of τ*. There are two kinds of dependence: (1) *control-flow 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'. While the control-flow dependence relates events generated by the same thread of the multi-threaded program, the data-flow dependence relates events generated by different threads: e may write some shared variable in a thread t, whose new value is used for the computation of the value of e' in another thread t'.

2.3.1 *Control-flow Dependence.*

Informally, if a change of $state(e)$ may affect the occurrence of e', then we say that e' has a *control-flow dependence* on e, and write $e \sqsubset_{ctrl} e'$. Control-flow dependence occurs inside of a thread, so we first define the total order within one thread:

Definition 1. *Let $<$ denote the union of the total orders on events of each thread, i.e., $e < e'$ iff $thread(e) = thread(e')$ and $e <_\tau e'$.*

This relation is extended by convention to abstract relevant events (when these are defined): if $e < e'$ then we also write $\alpha_\varphi(e) < e'$ and $e < \alpha_\varphi(e')$ and $\alpha_\varphi(e) < \alpha_\varphi(e')$. Then, with the help of the assumed *scope* function, we can define the control-flow dependence on events as follows:

Definition 2. *We write $e \sqsubset_{ctrl} e'$ iff $e < e'$ and $stmt(e') \in scope(stmt(e))$, and e is largest with this property, i.e., there is no e'' such that $e < e'' < e'$ and $stmt(e') \in scope(stmt(e''))$.*

In other words, if e and e' are events occurring within the same thread in an execution trace τ of some multi-threaded system, we say that e' has a *control-flow dependence* on e, written $e \sqsubset_{ctrl} e'$, iff e is the *latest* event occurring before e' with the statement that generated e' in the control scope of the statement that generated e. The purpose of control-flow dependence, say $e \sqsubset_{ctrl} e'$, is to show that the existence of an event e' is determined by the existence of all the events e. Obviously, the events generated in the branches of some conditional statement or in the body of a loop have the control-flow dependence on the events determining the choice of the condition statement. Consider the two example programs in Figure 3. In (A), the write on x at S_1 and the write on y at S_2 have a control-flow dependence on the read on i at C_1, while the write on z at S_3 does not have such control-flow dependence; in (B), the write on y at S_1 control-flow depends on the read on i at C_1. But for the write on z at S_2, the situation is more complicated. Briefly, if the loop always terminates, events produced outside of the loop do not control-flow depend on the condition of the loop; otherwise, the loop condition control-flow precedes all the subsequent events.

2.3.2 *Data-flow Dependence.*

If a change of $state(e)$ may affect the $state(e')$ then we say e' has a *data-flow dependence* on e and write $e \sqsubset_{data} e'$.

Definition 3. *For two events e and e', $e \sqsubset_{data} e'$ iff $e <_\tau e'$ and one of the following situations happens:*

1. *$e < e'$, type(e) = read and stmt(e') uses target(e) to compute state(e');*

2. *type(e) = write, type(e') = read, target(e) = target(e'), and there is no other e'' with $e <_\tau e'' <_\tau e'$, type(e'') = write, and target(e'') = target(e');*

3. *$e < e'$, type(e') = read, stmt(e') $\notin$ scope (stmt(e)), and there exists a statement S in scope (stmt(e)) s.t. S can change the value of target(e').*

One can see in the definition that, in most cases, the data-flow dependence is straightforward: for an assignment statement, the write on the left hand side has the data-flow dependence on the reads on the right hand side; and a read data-flow depends on the most recent write on the same memory location. Note that the second case in this definition resembles the interference dependence in [10], but our definition is based on runtime events instead of statements. For example, in Figure 3 (A), if an execution is $C_1S_1S_3$, then the read on x at S_3 has data-flow dependence on the write on x at S_1. However, some cases are a little more intricate. Assuming another execution of Figure 3 (A), say $C_1S_2S_3$, one will not see a direct data-flow dependence. If the value of i changes then S_1 could be executed instead of S_2, so the value of the write at S_3 would be different. Therefore, there is a data-flow dependence from the write at S_3 to the read at C_1. Similarly, in Figure 3 (B), the write on z at S_2 data-flow depends on the read at C_1. Therefore, event e' data-flow depends on e if e is an affecting event at a choice statement S and the value of e' can be changed by some statement in the control scope of S. By affecting, we mean that the value of the event may change the choice of the statement. To correctly determine such data-flow dependence, aliasing information among variables is required, which one can achieve using any available techniques.

Note that there are no write-write, read-read, read-write data dependencies. Case (2) above only considers the write-read data dependence, enforcing the read to depend upon only the latest write of the same variable. This way, a write and the following reads of the same shared variable form an *atomic* block of events. This captures the work presented in [21], in the much more general setting of this paper.

Similarly to the control-flow dependence, the data-flow dependence also extends by convention to abstract relevant events (when defined) as expected: if $e \sqsubset_{data} e'$ then $e \sqsubset_{data} \alpha_\varphi(e')$, $\alpha_\varphi(e) \sqsubset_{data} e'$, and $\alpha_\varphi(e) \sqsubset_{data} \alpha_\varphi(e')$. Note that if the abstract event e does not contain a state attribute, then the data-flow dependence is not taken into account.

2.3.3 *Hybrid Dependence.*

Now we can define the notion of hybrid dependence on events by merging the control-flow and the data-flow dependences:

Definition 3. *Event e'* ***depends upon*** *e if and only if $e \sqsubset e'$, where $\sqsubset$ is the relation $(\sqsubset_{data} \cup \sqsubset_{ctrl})^+$.*

As indicated by the discussion above, to compute this dependence relation on events some static structural information about the program is required. The most important piece of information that we collect statically is the *control scope* of every conditional statement, which is formally discussed in [2]. Besides, termination information of loops and aliasing relationship among variables are also needed. Termination and aliasing analyses are difficult problems by themselves and out of the scope of this paper. We are trying to make use of off-the-shelf analysis tools in our approach to accumulate static information about the program, which is further *conservatively* used by the subsequent dynamic analysis components, guaranteeing the soundness of our approach. Some heuristic assumptions may be adopted in implementations of the technique to improve performance, but these may introduce false alarms (see Section 4).

Thanks to the dynamic/static combined flavor of our approach, we only need to carry out intra-procedural static analysis. Method invocations will be expanded at runtime and the dependence relation can be propagated along method invocations easily: if a method call control-flow depends on an event e, then all events produced by the method call control-flow depend on e. Moreover, since our actual analysis takes place at runtime, we keep track of actual memory locations appearing in events, so the inter-procedural data-flow dependence can be computed similarly to the intra-procedural one using memory locations instead of variable names.

It is worth noting that the above discussion about dependence is independent from the particular definition of an event. The hybrid dependence can be computed on either the concrete events generated by the execution, or on the abstract relevant events. The latter usually results in more relaxed (less constrained) dependence relationships. For example, if some abstract event does not contain/need information about the state of an event (e.g., for data-race analysis we only care that there is a write of z at S_3 in Figure 3 (A), but not the concrete value written to z), then only the control-flow dependence is considered and the data-flow dependence can be ignored.

2.4 Sound Permutations and Sliced Causality

One can show that any linearization of events that is consistent with, or preserves, the hybrid dependence partial-order guarantees the occurrence of relevant events and also preserves their state. Our goal is to *generate and analyze permutations of relevant events* that correspond to possible executions of the system.

Definition 4. *A permutation of ξ_φ is* ***sound*** *iff there is some execution whose trace can be abstracted to this permutation.*

The most appealing aspect of predictive runtime analysis is that *one does not need to re-execute the program* to generate sound traces; instead, we define an appropriate notion of causal partial order and then prove that any permutation consistent with it is sound. Intuitively, a sound permutation preserves relevant events as well as events upon which relevant events depend.

Definition 5. *Let $\overline{\xi_\varphi} \subseteq \xi \cup \xi_\varphi$ be the set extending ξ_φ with events $e \in \xi$ such that $e \sqsubset e'$ for some $e' \in \xi_\varphi$. We then let $\prec \subseteq \overline{\xi_\varphi} \times \overline{\xi_\varphi}$ be the* ***sliced causal partial order*** *relation, or the* ***sliced causality****, defined as $(< \cup \sqsubset)^+$.*

For example, in Figure 2, for the property "e_1 must happen before e_3", we have that $\overline{\xi_\varphi} = \xi_\varphi = \{e_1, e_3\}$, while for the property "e_1 must happen before e'_3", $\xi_\varphi = \{e_1, e'_3\}$ and $\overline{\xi_\varphi} = \{e_1, e_2, e'_3\}$ because $e_2 \sqsubset e'_3$. Unless otherwise specified, from now on by "causal partial order" we mean the sliced one. Therefore, the sliced causality is nothing but the hybrid dependence relation extended with the total order on the events generated by each thread; or, in other words, it can be regarded as the slice of the traditional causal partial order based on the dependence relation extracted statically. The causal partial order was defined on more events than those in ξ_φ, but in order to generate sound permutations of relevant events we only need its projection onto the relevant events:

Theorem 1. *A permutation of ξ_φ is a sound abstract trace whenever it is consistent with the sliced causality $\prec$.*

Proof: First, let us define what it means for an observed event to occur in another execution. Event e is said to *occur* in a trace γ iff there is an event e' in γ, such that for any attribute *key*, either $key(e) = key(e')$ or both are undefined. Event e is said to *occur regardless of attribute key* in γ iff there is some e' in γ, such that for any *key' ≠ key*, either *key'*(e) = *key'*(e') or both are undefined.

Now we can show that the control-flow dependence determines the occurrence of an event. Suppose an incomplete execution of the program that generated partial trace β and a relevant event e' that has not occurred yet but has *counter(e')* $- 1$ occurrences regardless of state in β. Also, suppose that for any event e with $e \sqsubset_{ctrl} e'$, e has already occurred in β. Then e' will occur regardless of its *state* when the execution continues, independently of thread scheduling: the choice made at the statement generating e keeps unchanged as the state of e is unchanged, therefore the statement generating e' will be executed and e' is to occur regardless of its state.

One can also show that an event e' is uniquely determined by all the events e with $e \sqsubset_{data} e'$. Suppose an incomplete execution of the program that generated partial trace β and a relevant event e' that has not occurred yet but which will occur regardless of its *state* attribute, which also has the property for any event e with $e \sqsubset_{data} e'$, e has already occurred in β. Then e' (including the value of its *state*) will also occur when this execution continues, independently of thread scheduling, because all the values used to compute the state of e' keep unchanged.

Let $e_1 e_2 ...$ be a permutation of the events in ξ_φ that is consistent with $\prec$, or in other words a *linearization of* $\prec$, and let $\Sigma_i = \{e_1, ..., e_i\}$ denote the set of the first i events of this abstract trace. Then one can easily show by induction on i that if $e \prec e_i$ for some event e, then $e \in \Sigma_i$. Such sets Σ_i are also called *consistent cuts* and will be further discussed in Section 3.2. Then we can construct an execution of the program for this permutation by induction (it is also employed to generate a counter-example for detected violations):

1. For e_1, we simply start the thread $thread(e_1)$ and pause it after e_1 is generated;

2. For e_i, by the induction hypothesis we have constructed an execution of the program which produces $e_1 ... e_{i-1}$. Since all the events upon which e_i depends are already preserved in the execution, according to the above discussion, e_i is about to occur regardless of its state (because for all e, $e \sqsubset_{ctrl} e_i$, e is preserved in the permutation by hypothesis), and if e_i contains the attribute *state*, e_i is about to occur (because all e, $e \sqsubset_{data} e_i$, e is preserved in the permutation). In other words, we can safely start the thread $thread(e_i)$ to produce e_i and pause it.

□

We can therefore simulate an execution of the system that generates the original permutation of relevant events as an abstract trace.

3. GENERATING SOUND PERMUTATIONS

We next describe the generation of *sound* event permutations, that is, ones that are consistent with the sliced causality relation discussed above. First, a vector clock (*VC*) based algorithm that encodes the sliced causality is introduced. Then we show that the causal order can be further relaxed by considering the particular atomicity semantics of synchronization objects (locks), rather than a generic read/write semantics. Finally, an algorithm is given which generates all the sound permutations of relevant events *in parallel*, following a level-by-level (in terms of the associated computation lattice) or breadth-first strategy.

3.1 Computing Causal Partial Order

A *VC*-based algorithm was presented in [20] to encode a "happen-before" causal partial ordering on events that was extracted entirely dynamically, ignoring any static information about the program that generated the execution trace. We next non-trivially extend that algorithm to consider static information, transforming it into a *VC* algorithm encoding the slicing causality relation.

Definition 6. *A vector clock (VC) is a function from threads to integers,* $VC : T \rightarrow Int$*, where* T *is the set of threads.* $VC \leq VC'$ *iff* $\forall t \in T, VC(t) \leq VC'(t)$*. And we have the max function for VCs:* $max(VC_1, ..., VC_n)(t) = max(VC_1(t), ..., VC_n(t))$.

Every thread t has a VC_t, which keeps the order within the thread as well as the information about other threads that it knows from their communication (read/write events on shared variables). Every variable x has a VC_x that shows how the value of the variable is computed. Every shared variable x has a VC^r_x that accumulates the information about variable accesses. When a concrete event e is encountered, it will be abstracted using the filter function and then associated with a VC_e, which encodes the causal partial order. We next show how to update these *VCs* when an event e is encountered during the analysis (the third case can overlap the first two cases):

1. $type(e) = write, target(e) = x, thread(e) = t$ (the variable x is written in thread t). In this case, the VC_x is updated using VC^r_x, VC_t, and VCs of those events upon which e depends: $VC_x = max(VC^r_x, VC_t, VC_{e1}, ..., VC_{en})$ where $e_1, ..., e_n \sqsubset e$. Then $VC_e = VC^r_x = VC_x$.

2. $type(e) = read, target(e) = x, thread(e) = t$ (the variable x is read in t), and x is a shared variable. The information of the thread is accumulated into VC^r_x: $VC^r_x = max(VC^r_x, VC_t)$, and $VC_e = VC^r_x$.

3. e is a relevant event w.r.t. the desired property, $thread(e) = t$. For this case, VC_t needs to be increased in order to keep the total order within the thread, and the corresponding relevant event will be issued to the observer with an up-to-date VC.

However, it is not straightforward to determine the relevant event if one tries to calculate the vector clocks *online*, when only the information up to some execution point is available (no look-up into the future). Figure 4 (A) and (B) illustrate two cases that require backtracking in the calculation, in which $e, e', e'' \in \xi_\varphi$ and $e_1, e_2 \in \xi - \xi_\varphi$. Basically, this is caused by some "delayed" dependence among events. For the case in (A), when e' is processed, e_1 seems unimportant to verify φ and is not taken into account by the algorithm. But when e'' is encountered, e_1 becomes an important event and the algorithm has to re-compute $VC_{e'}$. (B) is similar but a little more complex: e_1 and e_2 are considered unimportant until e'' is hit. To recognize such cases, we can notice that, if e_1 is not taken into account, the thread's vector clock is not updated using VC_{e1}. Therefore, we have $VC_{e1} \nleq VC_t$. And for the same reason, if e' has been processed and $e_1 < e'$, $VC_{e1} \nleq VC_{e'}$. This way, we are able to go back and refine the *VCs* of previous events.

Because of backtracking, in the worst case, the complexity of the online algorithm is square in the number of events. For *offline* analysis (carried out after the execution finishes), one can first scan the execution trace backwards to figure out all important events and then compute *VCs* from the beginning of the trace, reducing the worst case complexity to linear. The online version of the algorithm is adopted in our prototype jPredictor, although it presently works in the offline mode, because the experiments show that in practice backtracking appear very infrequently (Section 5).

We can show that the vector clocks encode the sliced causality:

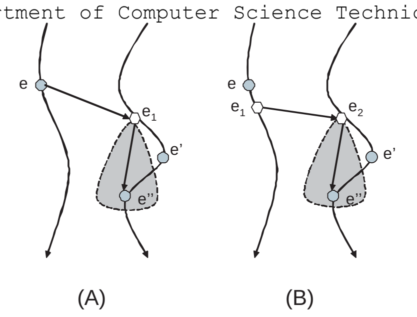


Figure 4: Backtracking cases for VC Generation (solid nodes are relevant events and blank nodes are irrelevant ones.)

Theorem 2. $e \prec e' \Rightarrow VC_e \leq VC_{e'}$

The proof of the important theorem above can be (non-trivially) derived from the one in [20]. The extension here is that the dependence is taken into account when computing the *VCs* of variables and relevant events. Note that in our case the partial order $\leq$ among *VCs* is stronger than the sliced causality $\prec$ among events. This is because when *VCs* are computed, the read-after-write order is also taken into account (the second case above), which the $\prec$ order does not need to encode. Theorem 1 yields the following immediately:

Proposition 1. *Any permutation on events that is consistent with $\leq$ among events' VCs is sound w.r.t. the sliced causality $\prec$.*

3.2 Lock-Atomicity of Events

One may further loosen the causal partial order. We next discuss how to incorporate the lock mechanism into our approach to construct more sound traces. One may notice that the dependence caused by lock operations discussed below is related to the synchronization dependence defined in [7], but, again, our notion is based on runtime events instead of statements; moreover, in our approach, the dependence is used to preserve the atomic block of synchronized events.

In most causal order based approaches, locks are treated as shared variables, and acquiring and releasing locks are viewed as reads and writes of the associated lock objects. This way, blocks protected by the same lock are naturally ordered and kept exclusive to one another. However, this ordering is stronger than the actual lock semantics, which only imposes the mutual exclusion among blocks protected by the same lock. To better support lock semantics, we next extend our sliced causality with *lock related atomicity*. Using this concept, two sets of events that are *atomic* w.r.t. the same lock cannot be interleaved, but can be permuted if there are no other causal constraints on them.

```
Thread t1:

e11(type = read, target = y…)
e12(type = write, target = y…)
e13(type = acquire, target = lock…)
e14(type = read, target = x…)
e15(type = write, target = x…)
e16(type = acquire, target = lock…)
```

Figure 5: Event trace containing lock operations

Two new types of events are introduced for lock operations, *acquire* and *release*. The target of these events is the lock to be accessed. If there are embedded lock operations on the same lock (a thread can acquire the same lock multiple times), only the outmost acquire-release pair generates events. For example, the thread t_1 in Figure 1 may produce the event trace in Figure 5. The control-flow dependence is extended correspondingly:

Definition 7. *$e, e' \in \xi$, $type(e) = acquire$ and $type(e') = release$ of the same lock l. Then $e \sqsubset_{ctrl} e''$ for all $e < e'' < e'$.*

That is to say, an event e protected by an acquiring of l has the control-flow dependence on the acquiring event. For example, in Figure 5, $e_{13} \sqsubset_{ctrl} e_{14}$ since $e_{13} < e_{14} < e_{16}$; and $e_{13} \sqsubset_{ctrl} e_{15}$. Two events protected by the same lock are *atomic w.r.t. the lock*:

Definition 8. *Two events e and e' are l-atomic, written $e \Updownarrow_l e'$, iff $\exists e'' \in \xi, type(e'') = acquire, target(e'') = l$, $e'' \sqsubset_{ctrl} e$ and $e'' \sqsubset_{ctrl} e'$. $\Updownarrow_l$ is an equivalence relation on ξ_φ. Let $[e]_l$ denote the corresponding equivalence class of an event $e \in \xi_\varphi$.*

For example, in Figure 5, $e_{14} \Updownarrow_{lock} e_{15}$, meaning that they are atomic w.r.t. *lock*. To capture the lock-atomicity among events, we associate a counter $counter_l$ with every lock l. Let LS_t denote the set of locks held by the thread t. A new attribute, LS, is also added into the event, whose value is a mapping on locks to corresponding counters. When an event e is processed, the lock information is updated as follows:

1. if $type(e) = acquire, thread(e) = t, target(e) = l$, then $counter_l = counter_l + 1$, $LS_t = LS_t \cup \{l\}$.
2. if $type(e) = release, thread(e) = t, target(e) = l$, then $LS_t = LS_t - \{l\}$.
3. if $\alpha(e)$ defined, then let $LS(e)(l) = count_l$ for any l in $LS_{thread(e)}$, and $LS(e)(l) = -1$ for any other l.

The following theorem states the correctness of this algorithm:

Theorem 3. $e \Updownarrow_l e'$ *iff* $LS(e)(l) = LS(e')(l) \neq -1$

Proof: Similar to the proof for Theorem 1, the definition of the consistent run actually gives the way to construct an execution of the program which can be represented by the permutation. □

3.3 Consistent Runs and Cuts

Every sound permutation of relevant events can be viewed as an abstract run of the program. A run is called *consistent* from now on if it preserves not only the sliced causality, but also the lock-atomicity relation above. Let us first define the concept of *consistent cuts*:

Definition 9. *A **cut** Σ is a set of events. Σ is **consistent** if and only if for all $e, e' \in \Sigma$,*
(a) if $e' \in \Sigma$ and $e \prec e'$, then $e \in \Sigma$ and
(b) if $e' \notin [e]_l$ for some lock l, then either $[e']_l \subseteq \Sigma$ or $[e]_l \subseteq \Sigma$.

The first property 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 l-atomic set in Σ. Otherwise, the l-atomicity is broken. Essentially, Σ contains the events in the prefix of a consistent run. When an event e can be added to Σ without breaking the consistency, e is called *enabled* for Σ.

Definition 10. *An event e' is **enabled** for a consistent cut Σ iff*
(a) for any event $e \in \xi$, 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 $[e]_l \subseteq \Sigma$.

This definition is equivalent to the following one:

Definition 11. *e is **enabled** for a consistent cut Σ if and only if $\Sigma \cup \{e\}$ is also consistent.*

Now we can define a consistent run:

Definition 12. *A **consistent multi-threaded run** $e_1 e_2 ... e_{|\xi_\varphi|}$ is one which generates a sequence of consistent cuts $\Sigma_0 \Sigma_1 ... \Sigma_{|\xi_\varphi|}$: for all $1 \leq r \leq |\xi_\varphi|$, Σ_{r-1} is a consistent cut, e_r is enabled for Σ_{r-1}, and $\Sigma_r = \Sigma_{r-1} \cup \{e_r\}$.*

Theorem 4. *Any consistent run of ξ_φ is sound.*

3.4 Generation of Consistent Runs.

```
procedure $main()$
1.  while $(\xi_\varphi \neq \emptyset)$
2.      $verifyNextLevel()$
3.  endwhile
endprocedure
procedure $verifyNextLevel()$
1.  for all $m \in \xi_\varphi$ and $\Sigma \in CurrentLevel$ do
2.      if $enabled(\Sigma, m)$
3.      then
4.          $NextLevel \leftarrow NextLevel \cup createCut(\Sigma, m, \xi_\varphi)$
5.      endif
6.  endfor
7.  $\xi_\varphi \leftarrow removeUselessMessages(CurrentLevel, \xi_\varphi)$
8.  $CurrentLevel \leftarrow NextLevel$
9.  $NextLevel \leftarrow \emptyset$
endprocedure
procedure $enabled(\Sigma, m)$
1.  $i \leftarrow thread(m)$
2.  if not $(\forall j \neq i : VC(\Sigma)[j] \geq VC(m)[j])$
            and $(VC(\Sigma)[i] + 1 = VC(m)[i])$
3.  then return $false$ endif
4.  if $(\exists lock\ l,\ LS_l(m) > -1,\ LS_l(\Sigma) > -1,$
            and $LS_l(\Sigma) \neq LS_l(m))$
5.  then return $false$ endif
6.  return $true$
endprocedure
procedure $createCut(\Sigma, m, \xi_\varphi)$
1.  if not $monitor(\Sigma, m)$
2.  then $reportViolation(\Sigma, m)$ endif
3.  $\Sigma' \leftarrow$ new $copy\ of\ \Sigma$
4.  $i \leftarrow thread(m)$
5.  $VC(\Sigma')[i] \leftarrow VC(\Sigma)[i] + 1$
6.  if $type(m) = acquire$ and $target(m) = l$
7.  then $LS_l(\Sigma) \leftarrow LS_l(m)$
8.  else
9.      if $type(m) = release$ and $target(m) = l$
10.     then $LS_l(\Sigma) \leftarrow -1$ endif
11. endif
12. return $\Sigma'$
endprocedure
```

Figure 6: Consistent runs generation algorithm

Figure 6 gives a breadth-first traversal algorithm to generate and verify, on a level-by-level basis, consistent runs based on the causal partial order and the lock-atomicity. In this algorithm, ξ_φ is the set of relevant events, while *CurrentLevel* and *NextLevel* are sets of cuts. We do not store all the events for the cut Σ in the algorithm; instead, Σ is encoded using the following information: the *VCs* of threads and shared variables, lock sets held by threads, and the current state of the property monitor for this run. The property monitor is a program which verifies the run against the desired property. In our current prototype, the monitor is automatically generated from the specification of the property (see Section 4).

Figure 7 shows a simple example for generating consistent runs. Figure 7 (A) is an observed execution of a two-thread program. The solid arrow lines are threads, the dotted arrows are dependencies among events and the dotted boxes show the scopes of synchronized blocks. Both synchronized blocks are protected by the same lock, and all events marked here are relevant. Figure 7 (B) then illustrates a run of the algorithm in Figure 6, 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, LS_{00}, at the top level. 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 the consistent run: it first compares the VCs between a candidate event and a cut, and then checks for the compatibility of their lock-atomicity. For example, only e_{11} is enabled for the initial cut, LS_{00}; and e_{12} is enabled for LS_{10} on the second level, but e_{21} is not because of the lock-atomicity. On the third level, after e_{11} and e_{12} have been consumed, e_{21} and e_{13} are both enabled for the cut LS_{20}. If an event e is enabled for a cut Σ, it will be added to Σ to create a new cut Σ', as shown by the transitions in Figure 7. But first, as shown in the *createCut* procedure, it is sent to the monitor along with Σ to verify against the desired property. Violations are reported as soon as detected. Otherwise, the vector clocks and lock set information of Σ' will be computed and Σ' is returned. After the next level is generated, redundant events, i.e., already processed in all runs, e.g., e_{11} after the second level is generated, will be removed from ξ_φ.

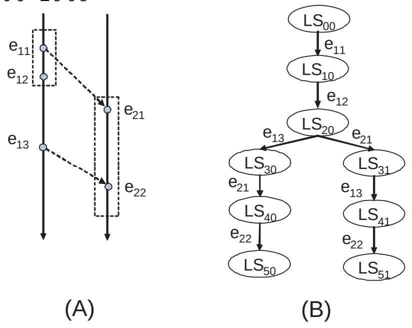


Figure 7: Example for consistent run generation

4. JPREDICTOR

To evaluate the effectiveness of the proposed technique, we implemented a prototype predictive runtime analysis tool for multi-threaded JAVA programs, called JPREDICTOR. Despite its yet unfriendly user interface and room for improving its performance, JPREDICTOR was able to detect several concurrency bugs, some of them unknown yet, in non-trivial applications (Section5), including Tomcat 5 [23]. We are continuously improving this prototype and applying it on new examples, and intend to transform it in a real, easy to use tool. It is fair to mention here that, despite the theoretical soundness of our sliced causality technique, its implementation in JPREDICTOR is *not* sound anymore, i.e., false alarms may be reported. Our decision to break its theoretical soundness was due to purely pragmatic reasons, explained in the sequel. However, in our experiments with JPREDICTOR all the violations it reported were real violations.

Figure 8 shows the architecture of JPREDICTOR. The system contains three major components: a *static analyzer*, a *trace analyzer* and a *monitor synthesizer*. The static analyzer *instruments* the program to issue events when executed; it also extracts static structural information from the program to be used later by the trace analyzer. The monitor synthesizer generates monitors from requirements specifications, which will be further used by the trace analyzer to verify the various permutations of relevant events against desired properties. For efficiency and modularity reasons, we distinguish two kinds of monitors in jPredictor: (1) specialized monitors that check well-defined, particular but important properties, such as dataraces for different variables; and (2) general purpose property monitors, automatically generated from formal specifications using the publicly available *logic-plugins* of the JavaMOP system [1]. By analyzing statically the property specification, the monitor synthesizer also provides the definition of relevant events. We do not discuss the monitor synthesizer here.

Once the program is instrumented and the monitors are generated, the user of jPredictor needs to run the instrumented program to gather execution traces, which are fed into the trace analyzer for the actual predictive analysis. The trace analyzer extracts the relevant events form the concrete trace(s), computes their *VCs*, and then constructs consistent runs by permuting relevant events, at the same time checking them against the corresponding monitors.

4.1 Static Analyzer

The static analyzer takes the original program as input and produces an instrumented program as output, together with static information needed for the trace analyzer. Figure 9 shows the three main components of the static analyzer: a *program instrumentor*, a *control flow analyzer* and an *alias analyzer*. All the outputs are stored in ASCII text files.

The program instrumentor is the core component of the static analyzer. It works at the byte code level. We are currently using the jTrek [3] package. The original program is instrumented with bytecode instructions that issue events at runtime, such as reads/writes on memory locations and begins/ends of function calls. The generated events are first placed in a global synchronized buffer and then flushed into a log file.

The soundness of our sliced causality technique is based on the assumption that *all* the code is instrumented. However, in practice, complete code instrumentation can cause an unacceptable runtime overhead; moreover, sometimes it is even impossible to achieve it, e.g., due to native methods. To keep its analysis practical and effective, jPredictor allows its users to specify which parts of the program to instrument. This way, the user can control the granularity and performance of the analysis by choosing different sets of classes to instrument according to the property of interest. There may be therefore uninstrumented methods invoked from the instrumented program. To avoid losing dependencies on variable updates, the untracked methods can be annotated with *purity* information: pure methods do not change the receiver object and will be regarded as reads on the object, while non-pure uninstrumented methods are regarded as writes on the receiver. Also, arguments that can be changed by the method can be annotated as *out* arguments. These method annotations can be reused and may be obtained by static analysis on the source code (if available), or even contained in interface specifications of classes, e.g., JML [12] specifications. By default, jPredictor is conservative and assumes all the un-tracked methods impure and all their arguments of reference types vulnerable. User annotations can only improve the predictive capability of jPredictor, but not affect its soundness.

As mentioned in Section 2, the termination information of loops may also be taken into account to relax the control dependence relation. jPredictor allows the user to introduce annotations regarding termination in the code; one can produce these annotations either manually, or otherwise automatically by using some off-the-shelf static analysis tool. To relive the user from producing termination annotations, a heuristic assumption for loops is implemented in jPredictor: when the condition of the loop involves no shared variables, the loop is assumed to terminate. This assumption brings unsoundness into the tool, but turned out to be so effective in our experiments (we did not need to further annotate any loops) that we decided to allow it anyway.

The control flow analyzer computes control scopes of statements using a simple algorithm discussed in [2]. The trace analyzer uses these control scopes to determine a refined control-flow dependence on events, as briefly explained in Section 2.3 and elaborated in depth in [2]. The alias analyzer implements a naive intra-procedural conservative alias analysis in our current implementation of jPredictor. By conservative, we here mean that all variables not known to be unaliased are assumed aliased. This way, the lack of precision of the alias analyzer only affects the predictive power of our tool, not its soundness. The soundness of jPredictor is only affected by our heuristic regarding the termination of loops, which was not a source of false alarms in our experiments.

4.2 Trace Analyzer

The trace analyzer implements our runtime analysis technique based on sliced causality, and therefore has the capability of predicting potential bugs from concrete executions of the program that may not hit the bug. Its input includes the execution trace generated by the instrumented program together with the static information produced by the static analyzer, along with the monitor to check the desired property. Currently, for simplicity, jPredictor works in the *offline* mode; that means that the analyzer is not invoked at runtime, but after the execution, analyzing the generated trace log. However, the main *VC* generation algorithm is designed to also work in the online mode. For efficiency reasons, the analysis process is divided into three phases, as depicted in Figure 10 (these will need to be changed in online mode).

In the first phase, the pre-processor traverses the input execution trace and collects information about the usage of objects. Specifically, life cycle information about objects is collected, based on which the *VC* generator can minimize the usage of memory by discarding information about objects when they are dead. Besides, if jPredictor is requested to detect dataraces, the pre-processor will also detect the shared variables. The complexity of the pre-processor is linear in the length of the trace.

The *VC* generator extracts the relevant events from the execution trace and computes the corresponding *VCs* and the lock-atomicity using the algorithms in Section 3.1 and Section 3.2. Since the trace contains detailed runtime information, the analysis can be very fine-grained, e.g., every element in an array can be processed individually if desired. However, in many cases such a fine-grained analysis is not necessary. Because of the back-tracing step, the cost of *VC* generation is, in the worst case, square in the length of the trace. In the offline mode, one can perform a backwards analysis of the trace first to compute the dependence, and then use another forward pass to compute the *VCs*. This way, the algorithm would be linear in the length of the trace, but it would only work in the offline mode. However, our experiments show that backtracking is needed very rarely: the forwards *VC* generation algorithm behaved linearly in the length of the trace in all tested cases.

The computed relevant event set, along with the implicit partial-order relationship encoded by vector clocks and lock-atomic sets, is then passed to the trace checker to verify it against the desired property. The trace checker generates all the consistent runs *in parallel*, on a level-by-level basis using the algorithm in Figure 6, invoking at the same time the property monitor to verify these runs. In the worst case when there is no partial-ordering among the relevant events (corresponding to no thread-interaction), case in which our technique explores the same state-space as a model-checker, the complexity of our trace analyzer would be exponential in the number of relevant events. Yet, as shown in the next section, since

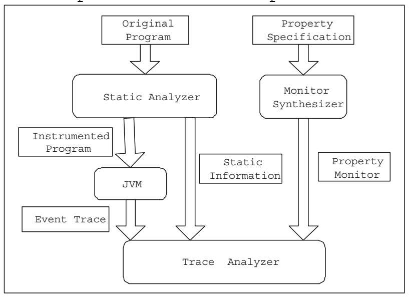


Figure 8: Architecture of jPredictor

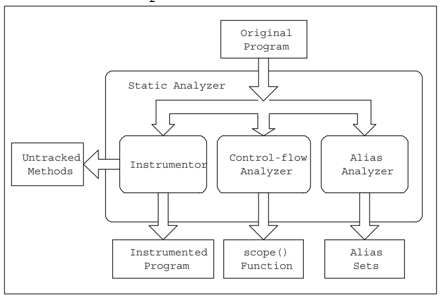


Figure 9: Architecture of static analyzer

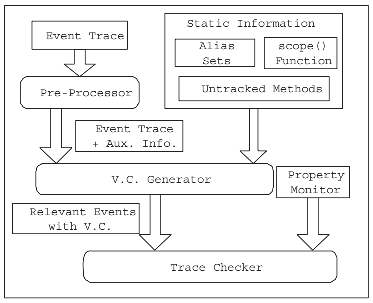


Figure 10: Architecture of trace analyzer

the number of relevant events is usually small, the complexity of the trace analyzer is quite reasonable. Dually to model-checking, where the goal is to reduce the state-space, in predictive runtime analysis we want to investigate as many potential runs as possible that are consistent with the observed execution. However, if the number of such runs is too large, one can select only those runs that are most likely to appear using the idea of *causality cone*, as we did in [21] for the purely dynamic happen-before relation considered there. Moreover, for some simple properties one does not even need to generate all the runs. For example, for data-race detection on a shared variable *x*, all one needs to check is that there are two causally unrelated access events on *x*, at least one of them is a write, and the two events have disjoint lock-sets; in this case, all what jPredictor needs to do is to compare the *VCs* and the lock-sets of events instead of generating the expensive permutations.

5. EVALUATION AND DISCUSSION

We evaluated jPredictor on two kinds of properties: dataraces and formal safety. Datarace properties are self-explanatory; formal safety properties were specified using regular expressions and temporal formalisms, and were synthesized into actual monitors using the logic plugin modules of JavaMOP [1]. We next 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. It is fair to say that improving (non-asymptotically) the performance of jPredictor was not and is still not among our immediate priorities. While raw efficiency is nevertheless desirable, we believe that in this particular project there are several other technically and intellectually more challenging aspects to be addressed, such as increasing the predictive capability by strengthening the static analysis part, exploring and perhaps automating the duality between sliced causality and computation slicing to early cut permutations that need not be verified, investigating test case generation and random testing techniques to generate causally orthogonal executions, among others. We leave performance optimizations of jPredictor to future work, mentioning that it was actually relatively efficient in most case studies. To evaluate the effectiveness of sliced causality, we also implemented the (unsliced) happens-before algorithm in [21]. All experiments were performed on a 2.0GHz Pentium4 machine with 1GB memory.

5.1 Benchmarks

Table 1 shows the benchmarks that we used, along with their size (lines of code), slowdown ratios after instrumentation, number of shared variables (S.V.) detected and number of threads created during their executions. Banking and Http-Server are two simple examples taken over from [24], showing relatively classical concurrent bug patterns that are discussed in detail in [4].

Program	LOC	Slowdown	S. V.	Threads
Banking	150	x3	10	11
Http-Server	170	x3	2	7
Daisy	1.3K	x10	312	3
Daisy-2	1.5K	x20	572	3
Raytracer	1.8k	x2	4	4
(Part of) Tomcat 5	10K	x10	20	3 - 4

Table 1: Benchmarks used in evaluation

Daisy [16] is a small file system proposed to challenge and evaluate software verification tools. It is highly concurrent with fine-grained locking. Specifically, it uses a RandomAccessFile object to simulate the hard disk, and user-defined spin-wait locks to protect logic blocks and directories. Since RandomAccessFile is a native class in Java, jPredictor cannot instrument it. This results in non-informative warnings: it only points out that there are dataraces on the *disk* variable, which is an object of the RandomAccessFile, but cannot give more specific reasons. We also implemented a revised version of Daisy, named Daisy-2, which replaces RandomAccessFile by a PseudoFile class, based on a byte array. The tool now successfully reports fine-grained race conditions. Both Daisy and Daisy-2 involve a large number of shared variables because every block of the disk holds a shared variable as a mutex lock, imposing a heavy load on the tool. Daisy-2 has more shared variables because of the shared byte array for disk simulation.

Raytracer is a program from the Java Grande benchmark suite [9]; it implements a multithreaded ray tracing algorithm. Tomcat [23] is a popular open source Java application server. The version used in our experiments is 5.0.28, the latest of Tomcat 5.0.x. 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; because of performance considerations, we only instrumented the tested components.

The test cases used in experiments were manually generated, using fixed inputs. Each test case was executed multiple times, to generate different execution traces. The detected bugs are all related to “unfortunate” thread scheduling. More bugs could be found if more effective test generation techniques were employed.

5.2 Detecting Dataraces

Program	T_{pre}	T_{vc}	T_{φ}	Races	Bugs	HB
Banking	1s	2s	5s	1	1	1
Http-Server	0.2s	0.3s	0.3s	2	2	1
Daisy	5s	30s	30s	1	1	0
Daisy-2	29s	30s	30s	2	2	0
Raytracer	1s	2s	2s	1	1	1
Tomcat	10s	20s	10s	4	2	0

Table 2: Race detection results

We evaluated our predictive runtime analysis technique first on datarace detection, since dataraces need no formal specifications and since their detection is highly desirable. jPredictor needs to repeat the *VC* generation and property checking for every shared variable. The times shown in Table 2 refer to checking races on just one shared variable: T_{pre} is the time for pre-processing, T_{vc} for *VC* generation, and T_{φ} for the actual datarace detection. Even though the worst-case cost of *VC* generation is $O(|\xi|^2)$, experiments strengthen our conjecture that backtracing is rare or inexistent in practice, so *VC* generation is expected to be linear. The performance of property verification, datarace in this case, is also reasonable. Last column shows the races detected with the standard, unsliced happen-before causality using the *same* execution traces. As expected, sliced causality is more effective in predicting bugs, since it covers more potential runs. Even though, in theory, the standard happens-before technique may also be able to detect, through many executions of the system, the errors detected from one run using sliced causality, we were not able to find any of the races in Tomcat, benign or not, without enabling the sliced causality.

In these experiments, jPredictor did *not* produce any false alarms and, except for Tomcat, it found all the previously known dataraces. For Tomcat, it found four dataraces: two of them are benign (do not cause real errors in the system) and the other two are real bugs. Indeed, 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. Interestingly, one bug was claimed to be fixed, but when we tried the patched version, the bug was still there. Let us take a closer look at these bugs.

```
public static void startCapture() {
  ...
  if (!reuse.isEmpty()) {
    log = (CaptureLog)reuse.pop();
  } else {
    log = new CaptureLog();
  }
  ...
}
```

Figure 11: **SystemLogHandler bug**

The first one is found in the *startCapture* method of the org.apache.tomcat. util.log.SystemLogHandler class. The buggy code is shown in Figure 11. *reuse* is a static member of the class, shared by different threads. There is an obvious datarace between *reuse.isEmpty* and *reuse.pop*, that causes an *EmptyStackException*. The difficulty in detecting this bug resides in the complicated thread interaction in the Tomcat. There are many unprotected shared variables that do not cause dataraces; therefore, plain lock-set algorithms would very likely produce many false alarms, overwhelming the true bug. On the other hand, standard happens-before techniques need real "luck" in such complex concurrent systems to generate executions in which there are no other unrelated inter-thread interactions between those conflicting memory accesses; otherwise, the race will not be revealed. It would be interesting to emprically compare jPredictor with the hybrid datarace detection technique in [14]; however, their tool appears not to be available for public download.

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

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

Figure 12: Buggy code in WebappClassLoader

```
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 13: Patched code in WebappClassLoader

The other bug is more subtle; it 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 12. 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 a function *definePackage(packageName, entry.manifest, entry.codeBase)*[1] to be null, which is illegal. We located this tricky error by a subsequent predictive analysis, this time checking a safety property about the usage of this method (Section 5.3); the safety property was derived from trials to evaluate the impact of the detected races. The error scene is not straightforward and would take quite some time to infer it directly from the datarace. It seems that a Tomcat developer tried to fix this bug by putting a lock around the conditional statement, as shown in Figure 13. However, jPredictorshows that the error still exists in the patched code, which is now a part of the latest verion of Tomcat 5, indicating that the problem has not been solved.

5.3 Verifying Safety Properties

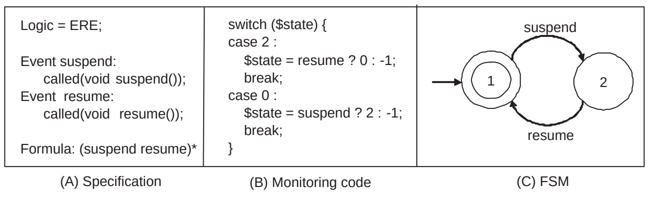
Although datarace detection has a central role in debugging concurrent systems, dataraces are not always corelated with actual bugs: many dataraces are benign, such as those in Tomcat, while concurrency errors may occur even when there are no dataraces, such as atomicity violations [6]. There are, therefore, arguments favoring detection of violations of desired property *directly*, as opposed to first detecting dataraces and then guessing possible violations caused by them. Our predictive runtime analysis technique makes no difference between dataraces and other properties, as far as they can be monitored against execution traces. For example, jPredictor can detect a datarace in Http-Server on the client object, but the actual error is the violation of the interface specification of the *Thread* class, stating that the calls to *suspend* and *resume* should alternate. This property can be written as a regular pattern: *(suspend resume)** ([1] gives more details).

We evaluated jPredictor also on a few temporal properties; the results are shown in Table 3, where the *Prop* column shows the number of properties checked and the *HB* column shows the violations detected using the standard, unsliced "happens-before" tech-

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

Program	T_{pre}	T_{vc}	T_{φ}	Prop	Violations	HB
Http-Server	0.2s	0.3s	0.3s	1	1	1
Daisy	3s	20s	10s	1	1	0
Tomcat	10s	20s	13s	1	1	0

Table 3: Safety property checking results



```
Logic = ERE;

Event suspend:
    called(void suspend());
Event  resume:
    called(void  resume());

Formula: (suspend resume)*
```

```
switch ($state) {
case 2 :
    $state = resume ? 0 : -1;
    break;
case 0 :
    $state = suspend ? 2 : -1;
    break;
}
```

Figure 14: Suspend/resume spec and generated monitor

nique. The properties are given as formal JavaMOP specifications [1], from which monitors can be automatically generated. For example, Figure 14 (A) gives the specification that we checked for Http-Server, and Figure 14 (B) and (C) show the corresponding generated monitor.

It may sometimes be more convenient to write monitors for desired properties manually than to first devise formal specifications for those properties and then generate monitors from them. For example, we have hand-written a (simple) monitor for atomicity in jPredictor, which checks whether all the events between the start and the end of the desired atomic block are generated by the same thread. If some valid permutation of events is found by jPredictor to satisfy this property, then the atomicity is *not* violated, otherwise an atomicity violation is reported. It would be interesting to explore in more depth the relationship between our predictive runtime analysis (instantiated to detect atomicity violations) based on consistent permutations of events and the left/right movers used in [6]. We believe that atomicity can be specified as a temporal property in some appropriate temporal logic and then a monitor for it can be generated automatically.

6. CONCLUSION

A runtime analysis technique was presented that can predict potential concurrency errors in multi-threaded programs by observing successful executions. The technique builds upon a novel causal partial order on events, called sliced causality, that relaxes the classic happens-before by taking into account control-flow dependency information, including termination of loops. A prototype tool implementing the proposed technique in the context of Java, called jPredictor has been implemented and evaluated. The experiments illustrate that jPredictor is able to detect concurrent bugs at a reasonable cost. A possibly unknown bug of Tomcat 5 was found during our evaluation.

7. REFERENCES

[1] F. Chen, M. d'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *International Conference on Formal Engineering Methods (ICFEM)*, LNCS, pages 357–372. Springer, 2004.

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

[3] S. Cohen. jTrek. Developed by Compaq.

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

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

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

[7] 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*, 1999.

[8] S. Horwitz and T. W. Reps. The use of program dependence graphs in software engineering. In *ICSE*, 1992.

[9] Java grande. http://www.javagrande.org/.

[10] J. Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, 1998.

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

[12] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA*, 2000.

[13] Y. Lei and R. Carver. A new algorithm for reachability testing of concurrent programs. In *International Symposium on Software Reliability Engineering (ISSRE'05)*, 2005.

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

[15] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.

[16] S. Qadeer. research.microsoft.com/qadeer/cav-issta.htm.

[17] 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.

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

[19] K. Sen and G. Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical Report UIUCDCS-R-2006-2676, University of Illinois at Urbana Champaign, 2006.

[20] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *FSE*, 2003.

[21] K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. *International Journal on Software Technology and Tools Transfer (STTT) (To Appear)*, 2005. Previous version appeared in TACAS'04.

[22] F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, CWI, 1994.

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

[24] S. Ur. Website with examples of multi-threaded programs. http://qp.research.ibm.com/QuickPlace/concurrency_testing/Main.nsf.