University of Illinois at Urbana-Champaign
Department of Computer Science Technical Report UIUCDCS-R-2007-2905

Effective Predictive Runtime Analysis Using Sliced Causality and Atomicity

Feng Chen and Traian Florin Şerbănuţă and Grigore Roşu

Department of Computer Science, University of Illinois at Urbana-Champaign
{fengchen,tserban2,grosu}@cs.uiuc.edu

Abstract. Predictive runtime analysis has been proposed to improve the effectiveness of concurrent program analysis and testing. By observing one execution trace of the running system, predictive runtime analysis extracts a causality relation among runtime events, which is then used as an abstract model of the program and checked against desired properties. This way, one can predict concurrency errors *without* actually hitting them and *without* re-executing the program. The quality of the extracted causality relation directly determines the effectiveness of predictive runtime analysis. This paper presents an efficient and sound approach to compute *sliced causality* and *sliced atomicity*. These significantly improve upon existing causalities: irrelevant causal relationships are removed using an apriori static analysis process based on control and data dependence, and on *property relevance* and *atomicity* analysis. The algorithms presented in this paper have been implemented and extensively evaluated. The results show that the proposed technique is effective and sound: we found the previously known concurrency bugs as well as some unknown errors in popular systems, like the Tomcat webserver and the Apache FTP server, *without* any false alarms.

1 Introduction

Concurrent programs are notoriously difficult to test, analyze and debug, due to their inherent non-deterministic nature. Predictive runtime analysis [19, 20, 7] has been proposed to improve the coverage and effectiveness of concurrent program testing and analysis without breaking the soundness of the analysis (i.e., no false alarms). By observing executions of the analyzed concurrent program, a predictive runtime analysis tool can infer and analyze feasible executions of the program that may have never occurred during testing. This way, concurrency bugs can be detected *without* having to generate test cases that make the program hit the bugs and even *without* the need to re-execute the program at all.

A good causality relation extracted from the observed execution trace can play a critical role in the effectiveness of predictive runtime analysis: the more relaxed (i.e., less constrained) that causality is, the more potential runs we can infer from the observed trace, and thus the larger the coverage and the better the predictive capability of the analysis is. The "proof-of-concept" predictive runtime analysis technique presented in [19] was based on the traditional happen-before

causality [14] and had comparatively little coverage. It was then improved in [20] by relaxing the traditional happens-before with write-read atomicity. Recently, [7] introduced the sliced causality, which drastically yet soundly improves the happen-before causality by removing irrelevant causal orders using dependence information from the program: it increases the analysis coverage exponentially.

However, the definition of sliced causality in [7] is built upon a dependence relation which imposes a strict ordering among events, thus being still over restrictive in many cases. Not all relations between events impose a causal order; the atomicity relation between a write and its subsequent reads, formally investigated in [20], is such an example. This paper presents a novel sliced causality which starts with a causality with atomicity, as in [20], and computes from it a sliced causality which is more relaxed than the one introduced in [7]. The key to achieving that is the introduction of the *relevance* among events, which points out relevant events without imposing an order between them. We show that atomicity can also be "sliced" using the property, without losing any feasible linearizations of the property-related events. The obtained sliced causality and atomicity can be efficiently computed from the observed trace. The presented algorithm has been implemented in jPredictor, our predictive runtime analysis tool for Java, and a series of experiments has been carried out.

There is much effort put in testing and dynamically analyzing concurrent programs. Some approaches [8, 18] aim at checking general purpose properties, while others [17, 22, 16, 11, 23] are specialized on particular properties, e.g., data-races and/or block atomicity. The existing approaches tend to focus on either soundness or coverage: for example, happen-before based techniques [8] try to avoid false alarms at the expense of a limited coverage, while lock-set based approaches [17, 16, 11] aim at providing better coverage at the expense of more false alarms. Predictive runtime analysis is generic to the property to check, and the sliced causality technique is proposed to improve the coverage of analysis without breaking its soundness. Indeed, our evaluation results show that our approach is viable and effective in practice: we were able to find all the previously known errors as well as some unknown ones in a comprehensive benchmark suite containing highly non-trivial Java applications.

The reminder of the paper is organized as follows. Section 2 revisits the basic concepts of events, dependence, and atomicity introduced in [20, 7], providing finer grained definitions of dependence. Section 3 introduces the *relevance* relation. Section 4 is the theoretical core of the paper, introducing and proving the soundness of the *sliced causality with atomicity*. Section 5 shows how our technique can be used to verify trace-related properties. Algorithms to extract sliced causality and atomicity are given in Section 6. Section 7 briefly presents the evaluation results and Section 8 concludes the paper.

2 Preliminaries: Events, Causality, and Atomicity

This section recalls definitions and results from [20, 7]. Events represent atomic steps observed in the execution of the program. In this paper, we focus on multi-

threaded programs and consider the following types of events (other types can be easily added): write/read of variables, beginning/ending of function invocations, acquiring/releasing of locks, and start/exit of threads. A statement in the program may produce multiple events. Events need to store enough information about the program state to allow the observer to analyze the trace.

Definition 1. *An **event** is a mapping of **attributes** into corresponding **values**.*

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. The attribute *state* contains the value associated to the event; specifically, for the write/read on a variable, $state(e)$ is the value written to/read from the variable; for ending of a function call, $state(e)$ is the return value if there is one; for the lock operation, $state(e)$ is the lock object; for other events, $state(e)$ is undefined. 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*).

Definition 2. *A trace τ is a finite ordered set of (distinct) events $\{e_1 < e_2 < \cdots < e_n\}$. Let $\mathcal{E}_\tau = \{e_1, e_2, \ldots, e_n\}$ be called the alphabet of τ and let $<_\tau$ be the total order induced by τ on $\mathcal{E}_\tau$. The **thread ordering** $<$ is given by $e < e'$ if $e <_\tau e'$ and $thread(e) = thread(e')$.*

Control dependence. Intuitively, a statement S *control-depends* on a choice statement C, iff the choice made at C determines whether S is executed or not. For example, in Figure 1, the write on x at S_1 and the write on y at S_2 have a control dependence on the read on i at C_1, while the write on z at S_3 does not have such control dependence.

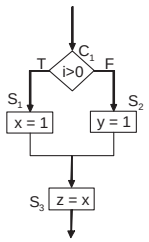


Fig. 1. Control dependence

The control dependence among events is parametric in a control dependence relation among statements. All we need to define it is a function returning the *control scope* of any statement C, say $scope(C)$, which is the set of statements whose reachability depends upon the choice made at C, that is, the statements that control depend on C, for some appropriate or preferred notion of control dependence. For control statements with a complex condition, e.g., involving function calls and side effects, we assume that one can transform the program to simplify its condition to a simple check of a boolean variable.

Formally, $e \sqsubset_{ctrl} e'$ iff $e < e'$, $stmt(e') \in scope(stmt(e))$, and there is no e'' such that $e < e'' < e'$ and $stmt(e') \in scope(stmt(e''))$. The last condition says that an event e is control dependent on the *latest* event issued by some statement upon which $stmt(e)$ depends. For example, in Figure 1, a write of x at S_1 is control dependent only on the most recent read of i at C_1.

The *soundness* of the analysis based on sliced causality is contingent to the *correctness* (no false negatives) of the employed control dependence: the analysis produces no false alarms when the *scope* function returns for each statement *at least* all the statements that control-depend on it. In our implementation of jPredictor[4], we chose to use the termination-sensitive control dependence (TSCD) introduced in [6].

Next three dependence relations (write-read, data and reference) were introduced in [7] as a single dependence, namely *data-flow dependence*. We here chose to introduce and name them separately to simplify the our presentation.

Write-Read dependence. The write-read dependence was originally introduced in [20]. We say e' *write-read depends on* x e, written $e \sqsubset^x_{wr} e'$, iff e is a write of x and e' is a read of x such that the latest write on x that happens-before e' is e. In other words, we say that $e \sqsubset^x_{wr} e'$ if and only if the value of x read by event e' is the value written by the event e.

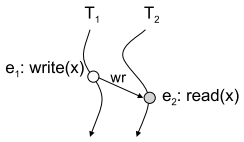


Fig. 2. WR dependence

Data dependence. Since the inter-thread data dependence is given by the write-read dependence, this data dependence is precisely the common intra-thread data dependence. For example, for an assignment $x = E$, the write of x has data dependence on the reads generated by the evaluation of E.

Formally, we say that e' *data-depends* on e, written $e \sqsubset_{data} e'$, if $e < e'$, $type(e) = read$ and $stmt(e')$ uses $target(e)$ to compute $state(e')$.

Reference dependence. Accessing the element of an array or a field/method of an object, for example, require that the location of the access if computed, typically using values of other locations. This induces a dependence between the target of an event and the reads events that are used to compute it. Consider, for example the statement `a[x] = 3`. Here, the location at which the value 3 will be written, depends on the values read from both a and x.

Formally, we say that e' *reference-depends* on e, written $e' \sqsubset_{ref} e$, if $state(e)$ is used to compute $target(e')$.

Causal dependence. Let the *causal dependence* $\sqsubset$ be the transitive closure of control-dependence, data-dependence, target-dependence and write-read dependence, that is, $\sqsubset = (\sqsubset_{ctrl} \cup \sqsubset_{data} \cup \sqsubset_{ref} \cup \sqsubset_{wr})^+$.

Given a causally closed set of events R, a linearization $\overline{\tau}$ of the events from R is consistent with a dependence ordering $\sqsubset$ if $\sqsubset{\restriction_R} \subseteq <_{\overline{\tau}}$. That is, dependence between events is preserved by the linearization.

All other events an event e causally depends on must occur in a linearization if we want to ensure that e will also occur. However, the occurrence of all these events in a linearization might not be enough to ensure the occurrence of e. That is because there might be other events which, although e does not directly depend on them, prove to be *relevant for* e under a different interleaving.

Write-Read atomicity[20]. Given a write event, say e, the set $\{e\} \cup \{e' \mid e' \in E \wedge e \sqsubset^x_{wr} e'\}$ should be regarded as atomic with respect to any other event outside the set writes x. Such a set is called an *atomic set* induced by e on target

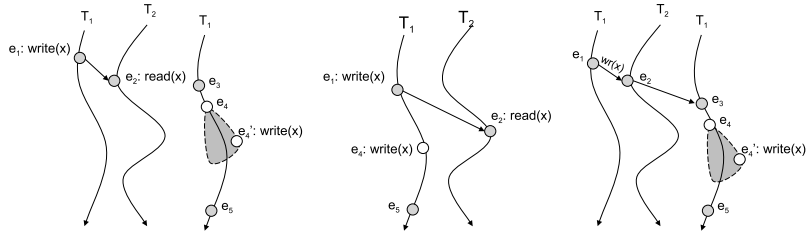
x and is denoted by $[e]_x$. A linearization of the events in $\mathcal{E}_\tau$ is consistent with the write-read atomicity if for any event $e' \in [e]_x$, any other write event e'' of x, such that $e'' \notin [e]_x$, appears either before or after both e and e' in the linearization.

As pointed out in [20], lock atomicity can be handled as a special type of read-write atomicity by regarding lock acquire as a write event and lock release as a read event. Therefore, we will use in the sequel "atomicity" to refer to both lock atomicity and write-read atomicity.

In [20] is proved that any linearization of $\mathcal{E}_\tau$ which is consistent with the thread ordering, the write-read dependence, and atomicity is a feasible trace on the multi-threaded system. This result takes into consideration *all* accesses to shared variables, using them to restrict feasible linearizations. However, this might prove too restrictive, since usually one only tries to verify one property at a time, and that property usually only depends on a very small number of shared variables. In this paper, we will prove a similar result but focusing only on the "events of interest" from the observed trace, and thus, maximizing the number of linearizations against which the property can be checked.

3 Relevance

[7] introduced a special kind of dependence, called relevance dependence, to point out that some otherwise not causally dependent events are relevant in the sense that they should not change in any consistent linearization. We here relax this notion in the sense that it no longer needs to be a dependence relation, keeping from it only the information that an should not be removed when slicing the trace. This proves to further enrich our property checking capabilities, since fewer dependencies yield more linearizations.



(a) Definition (b) Instance (c) Counter-example

Fig. 3. Relevance

Let R be a set of events closed under the causal dependence. To get a better intuition about relevance, let us begin with a visual, though abstract, example. Events in R are depicted by full discs and causality between them is pictured by arrows. In Figure 3(a), assume e_3 is the "latest" element in R before e_4 in T_3 and e_5 is the "first" element in R in T_3, not control-dependent on e_4. If $e_2 \prec_R e_3$, as in Figure 3(c), then e_2 will always occur before e_4 in any sound linearization, thus e_4 bears no relevance to e_2. Similarly, if $e_5 \prec_R e_1$, then again e_4 is not relevant for e_2, because even if e'_4 is reached and writes some value on x, e_1 will be generated after it, and before e_2, setting the right value of x for e_2 to read.

However, if neither of the above holds, then a permutation of the events in R in the order e_1, e_3, e_5, e_2 might be unsound, as e_4' might be generated. Therefore, in that case, e_4 must become relevant to ensure that e_4' is *not* generated.

Definition 3. *If e_1 and e_2 are from R, $e_1 \sqsubset_{wr}^{x} e_2$, and e' contains in its scope a statement which could generate a write on x, and if the scope of e' can't be proved to be either before e_1 or after e_2 using the causality on R and the thread ordering, then e'* is relevant for *e_2.*

An interesting instance of the above definition, not captured using the above intuition, occurs when in the above abstract example e_4 and e_4' coincide, as in Figure 3(b). In this setting, the definition above is read as follows: a write event which could occur as the last write before a relevant read on the same target, although it didn't in the observed execution, must become relevant; note, however, that by making it relevant, due to the write-read atomicity condition, e_4 won't appear between e_1 and e_2 in any permutation of the relevant events consistent with the write-read atomicity.

A similar kind of relevance arises in the case of lock acquire and release operations. Indeed, if an acquire operation occurred before another on the same lock in the observed trace, than the fact that the second acquire occurred depends on the fact that the release corresponding to the first acquire has occurred before. However, since we want to enable the blocks depending on the same lock to be permuted, we chose to say that the release is relevant (if it cannot be proved before the second acquire). To have a uniform treatment for relevant events, we chose to treat release events also as writes of their target when computing the relevant events and let the relevance as defined above handle them. Therefore, when computing relevant events a release event will be treated as a succession of a read event, depending on the write of the lock acquire and a write event (with no event depending on it) to ensure release is made relevant if required.

Definition 4. *We say that a set of events $R \subseteq \mathcal{E}_\tau$ is **relevance closed** if (1) if $e \sqsubset e'$ and $e' \in R$ then $e \in R$; and (2) if e is relevant for $e' \in R$, then $e \in R$.*

As we will shortly prove, a relevance closed set of events has the (good) property that all its linearizations consistent with the causality and atomicity are feasible traces. An obvious example for a relevance closed set of events would be the set of all events in a given trace, but this set is obviously not we are looking for. Instead, our goal is to obtain a "small" relevance closed set of events generated by the property-related events. We say "small" instead of "the smallest" because indeed there might not exists such a "smallest" set. This is primary due to the induced relevant events – their definition makes it possible that, when building the closure, one would have to choose between two events which one should be made relevant, and choosing any one of them might create enough dependencies to prevent the other from being chosen.

4 Sliced Causality with Atomicity

Definition 5. *Given a set of events R, let $\prec_R$ denote the* ***sliced causality relation*** *induced by R, defined by $\prec_R = (\sqsubset\restriction_R \cup <\restriction_R)^+$.*

In [20], every write of a shared variable is grouped with all its subsequent reads, forming an atomic block of variable accesses. We extend this *atomicity* to event sets in our approach as follows.

Definition 6. *Let R be a relevance closed set of events. For each write event e on target x, the* ***atomic block of*** *e induced by $\prec_R$ is the set*

$$[e]_x^R = \{e\} \cup \{e' \mid \exists e'', e \prec_R e' \prec_R e'' \text{ and } e \sqsubset_{wr}^x e''\}.$$

The *R-sliced-causality-with-atomicity*, weakens the sliced-causality introduced in [7] through the addition of relevance and atomicity.

Definition 7. *Let R be a relevance closed set of events. The R-sliced-causality-with-atomicity, or simply, the* ***R-sliced-causality*** *is the triple $(R, \prec_R, ([e]_x)_{e,x})$, where $\prec_R$ is the sliced causality relation induced by R and $([e]_x)_{e,x})$ are the atomic blocks induced by $\prec_R$.*

Given $E \subseteq R$, the E-sliced-causality induced by R, or, simply, the ***E-sliced-causality****, is the triple $(E, \prec_E, ([e]_x^E)_{e,x})$, where $\prec_E = \prec_R \cap E \times E$, and $[e]_x^E = [e]_x^R \cap E$ for each atomic block $[e]_x^R$.*

A permutation l of elements from E is a ***linearization of the E-sliced causality*** *if $\prec_E \subseteq <_l$, and for any $e_1 <_l e_2 <_l e_3$, such that there exists $e, e' \in E$, with $e_1, e_3 \in [e]_x^E$ and $e_2 \in [e']_x^E$, then it must be that $e = e'$, that is, two atomic blocks sharing the same location cannot not be interleaved.*

Notice that, if one takes E to be the entire set R in the above definition, it obtains the R-sliced causality. The following Theorem is the main result of this paper. It basically ensures that, given a relevance closed set of events R, all the linearizations of the R-sliced-causality are modeled by real executions of the program.

Theorem 1. *Consider a relevance closed set of events $R \subseteq \mathcal{E}_\tau$. For any linearization l of the R-sliced-causality there exists a trace τ generated by the system such that $<_l \subseteq <_\tau$. Call any such trace τ, a model of l.*

Proof. Let R be a relevance closed set of events. A prefix of a linearization of the event in R is termed *R-relevant trace*. We say that a (partial) trace σ models an R-relevant trace $\overline{\sigma}$ if $\mathcal{E}_{\overline{\sigma}} \subseteq \mathcal{E}_\sigma$ and $<_{\overline{\sigma}} \subseteq <_\sigma$. σ is a minimal model for $\overline{\sigma}$ if, additionally, the maximal elements of $< \sigma$ are in $\mathcal{E}_{\overline{\sigma}}$. Note that if σ is a minimal model for $\overline{\sigma}e$, then $\sigma = \sigma' e$ and σ is a model for $\overline{\sigma}$.

We will prove the following inductive property. Let $\overline{\sigma}e$ be a prefix of l, and suppose $\overline{\sigma}$ has a minimal model σ. Then there exists a minimal model for $\overline{\sigma}e$ containing σ as a prefix. The way this model is obtained is by simply continuing the execution of σ on the thread of e.

Let us first show that *stmt*(e) is reached. Since all events e control-depends on maintained their original values in the current execution, we infer that each time a choice point which could have chosen a path not leading to *stmt*(e) was reached, the execution took the same path as the original one.

Having proved that *stmt*(e) will be reached, let us show that it actually generates e. First let us show that *target*(e) is read/written when *stmt*(e) is executed. This is ensured by our target-relevance, since if *target*(e) is computed by using a read (e.g., *target*(e) references a field/method of an object), then that read was relevant, and thus it must have already occurred in σ. But this implies that *target*(e) was computed in the same manner it originally had been.

Now, let's prove that the value of *target*(e) after the execution of *stmt*(e) will be *state*(e). If e is a write event, then, since all read events e data-depends on have occurred, then the value computed for *state*(e) has to be the same, since it is precisely computed from those reads. If e is a read event, then, by write-read dependence, we know that the write event e' whose state e read in the original execution has already occurred in σ. However, in order to complete our proof we need to show that no other write event e'' has occurred in σ on *target*(e), after e'. We will show that the existence of such a'' would lead to contradiction. If e'' appeared in the original execution, then, it would have been relevant (from the definition of the relevance closure), which leads to a contradiction since the ordering e', e'', e would violate the atomicity consistency on the relevant events. e'' being an event which did not occurred in the original execution is also impossible since the events which control-prevented e'' from happening in the original execution, are relevant (by the definition of relevance), thus they must also occur in the current execution, preventing e'' from occurring. □

One can use the above theorem as-is to check a trace property: select the property-related events, compute their relevance closure, compute the sliced causality, generate its linearizations and check them against the property. However, the thus obtained linearizations, though sound, will yield too many traces with the same ordering between property events.

A naive solution to the above problem is to restrict the causality to just the property-related events. It might seem that we can safely do that since the relevant events have already played their role in building the causality between the property-related events. However, this only solves half of the problem, since our linearizations still depend on the atomicity, and it might be the case that there exists atomicity introduced by the relevant, yet not property-related events.

Atomic Relevance. This new kind of relevance is motivated by the fact that a linearization of the property-related-events-sliced-causality may still break the actual atomicity if there exist atomic blocks containing no property-related events. Let us consider the example in Figure 4. Suppose that all the events are relevant events and they are observed in the order indicated in the figure. $e_1, e_2, e_3, e_4, e_5, e_6$ are property-related events with no causal dependence among them except for the intra-thread ordering. According to the atomic block

definition, e_3, e_4, e_5, and e_6 are not in any atomic block of x. Therefore, $e_1, e_3, e_4, e_5, e_6, e_2$ is linearization of the property-related-events-sliced-causality, which breaks the atomicity for the block $[e'_1]_x$. In order to fix that, the write event e'_3 is relevant for generating consistent traces of property events. We term this new type of relevance *atomic relevance*. But even if we consider e'_3 together with the property events, and thus the atomic block is no longer empty, the atomicity is not yet guaranteed to hold. In the above example, although e'_3 is now relevant, one still can build the following linearization: $e_3, e'_3, e_4, e_1, e_2, e_5, e_6$, which breaks the atomicity for $[e'_3]_x$, therefore e'_5 is also relevant.

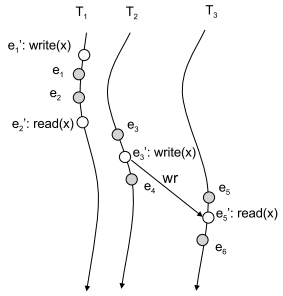


Fig. 4. Empty atomic block

Definition 8. *Let R be a relevance-closed set of events and let $E \subseteq R$. $A = \{e' \in [e]^R_x \mid thread(e) = t\}$ is an **atomically-relevant set** for E if $A \neq \emptyset$ and $E \cap A = \emptyset$. E is **atomically closed** if it has no atomically relevant sets.*

A very simple way to compute the atomic closure of a set E of events is to randomly select an event from each atomically relevant set and add it to E. However, one may employ strategies to add a minimal number of events, by adding events which belong to multiple atomically relevant sets.

Theorem 2 shows that if we are interested just in linearizations of a particular subset of a relevance closed set of events, then we can soundly "slice" the sliced causality even more, without losing any existing linearization of the events of interest, provided that we chose an atomically closed set of events to slice on.

Theorem 2. *Let R be a relevance closed set of events. Let L_R be the set of linearizations of the R-sliced-causality. Let $E \subseteq R$ be a atomically closed set of events. Let L_E be the set of linearizations of the E-sliced-causality.*

Then, (1) for any linearization $l_R \in L_R$, there exists a linearization $l_E \in L_E$ such that $<_{l_E} \subseteq <_{l_R}$; and (2) for any linearization $l_E \in L_E$, there exists a linearization $l_R \in L_R$ such that $<_{l_E} \subseteq <_{l_R}$.

Proof. (1) Trivial: since $\prec_E \subseteq \prec_R$ and $[e]^E_x \subseteq [e]^R_x$, any linearization of $\prec_R$ consistent with $([e]^R_x)_{e,x}$ will also be a linearization of $\prec_E$, consistent with $([e]^E_x)_{e,x}$.

(2) The idea is to refine the causality $\prec_R$ and the atomic blocks $([e]^R_x)_{e,x}$ using the order $<_{l_E}$ and the atomic blocks induced by it on $([e]^E_x)_{e,x}$ such that any consistent linearization of the new obtained causality and atomicity will be a linearization of $\prec_R$ containing $<_{l_E}$ and consistent with the atomicity. The proof then proceeds by showing that the newly obtained causality admits linearizations consistent with the atomicity. E being atomically closed plays a crucial role in this part because, guarantees that no atomic block is broken in consistent permutations of $\prec_E$. □

5 Checking Properties

Our goal is to verify trace-related properties over possible runs of a program, obtainable through different interleavings of the original execution, yet preserv-

ing unchanged the events related to the input property. Therefore, we aim to filter the causal dependencies between events by using information about the property to be checked.

Property Events. Each logical property φ conceptually defines a set A_φ of abstract events, that is, events which are needed for either validating or invalidating the formula φ. We call those events abstract because they only specify parts of the attributes held by a regular event. For example, if φ is a formula specifying a datarace on a target x, then the abstract events of this formula only need to specify the thread, the target (which should be x) and the type of event (whether a read or a write occurred on x).

Given an observed trace τ, one can identify within it the concrete events related to φ, that is, which specialize abstract events from A_φ. However, these concrete events might contain more information than one needs for testing the validity of φ (for example, the value read from or written to the target is not relevant when testing dataraces). We therefore choose to enrich the trace by inserting into τ a set of artificial events E_φ, termed *property events* at places where instances of the abstract events are found.

Property events are themselves instances of abstract events, carrying the information of the abstract events they represent together with attributes (such as counter and thread) making them unique among the other events from the trace. Property events are inserted in the trace right before their corresponding concrete events, in order to guarantee that every event which causally depends on their associated concrete events, will also depend on them.

We extend the causal dependence on property events by letting them depend on the same events their associated concrete events depend on, but only if the dependence does not involve their undefined attributes (e.g., for dataraces we need not consider data dependencies for property events, since they have no state attribute). However, we allow no other event to directly depend on them. Intuitively, this is due to the fact that we will slice the trace to insure that property events are generated, therefore we only need in the sliced trace events on which property events depend on. Nevertheless, if the events depending on a property event e are relevant for another property event, then they will appear in the sliced trace anyway, and their dependence to e will be preserved through the corresponding concrete event, by the thread ordering. That is why we have chosen property events to occur before their corresponding concrete event.

Property Closure. An event e is relevant for the absence of a property event, termed *relevant for* φ, if a change in the state of e might lead to the apparition of a new instance of an abstract event. An example of such a relevant event is read event guarding the execution of a block in which an instance of a abstract property event might be produced.

Definition 9. *A relevance closed set of events $R \in \mathcal{E}_\tau$ is* ***property closed*** *if (1) $E_\varphi \subseteq R$; and (2) if e is relevant for φ, then $e \in R$.*

The first condition assures that all property events in τ will be taken into account by R. The second condition insures that abstract property events which

don't occur in τ will not appear in any execution modeling a linearization of elements in R – requiring that all control points which could lead to generating new abstract property events will not change.

In this setting, one can safely use Theorem 1 to generate linearizations of the property closed set of events and check them against the property.

Corollary 1. *For any linearizations of a property-closed set of events, consistent with the sliced causality and the atomicity, there exists a real execution modeling it, without introducing new instances of abstract property events.*

Combining Theorem 2 with Corollary 1, we obtain that one can soundly and without any loss of generality use the linearizations of any atomically closed set of events which includes the properties events to check the property.

Corollary 2. *Given R a property-closed set of events and $E \subseteq R$ an atomically closed set including E_φ, any linearization of the E-sliced-causality is modeled by a real execution, without introducing new instances of abstract property events.*

6 Extracting Sliced Causality and Atomicity

We next present algorithms to extract the sliced causality and atomicity from traces. First, a property closure is obtained using a slicing algorithm. Then we adapt the vector clock based algorithm in [20] to compute the causal dependence on the sliced trace. A new atomicity algorithm is devised to associate property events with the sliced atomicity based on the computed causal dependence. Finally, one can generate and verify permutations of property events that are consistent with the sliced causality using a depth-first algorithm.

Slicing Traces Our goal here is to take a trace τ and a property φ, and to generate a *sliced trace* τ_φ which is the total order induced by τ reduced to a property closed set of events of φ. We next briefly explain an algorithm to compute ξ_φ using a variant of *dynamic program slicing* [1] and give the corresponding correctness result.

First one has to keep any event e with $e \sqsubset^+ e'$ for some property event (or event preventing a property event from being generated) 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 [6]. One backwards traversal of the trace does not suffice to correctly calculate all the relevant events since an event may turn out to be relevant only when another event occurring before it becomes relevant, e.g., e_4 in the instance of the induced relevance shown in Fig. 3. Therefore, the backward traversal should be carried out until the computed sliced trace stabilizes. The following holds:

Proposition 1. *The computed sliced trace is φ property closed.*

Computing causal dependence using vector clocks. [20] introduces an algorithm to compute a "weak" causal partial order using vector clocks. That algorithm can be easily adapted to capture the causal dependence from the sliced trace. The major difference here is that we have user-defined property events while the algorithm in [20] only handles writes/reads of shared variables. What follows illustrates the adapted algorithm.

A vector clock (VC) is a function from threads to integers, $VC : T \rightarrow Int$. We say that $VC \leq VC'$ iff $\forall t \in T$, $VC(t) \leq VC'(t)$. The max function on VCs is defined as: $\max(VC_1, ..., VC_n)(t) = \max(VC_1(t), ..., VC_n(t))$. Intuitively, vector clocks are used to track and transmit the causal partial ordering information in a concurrent computation. 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 a vector clock to enforce the order among writes of x. Every property event e found in the analysis is associated a VC attribute, which represents the computed causal partial order. When an event e is encountered during the analysis, the VCs are updated as follows:

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 variable vector clock VC_x is updated to reflect the newly obtained information: $VC_x \leftarrow \max(VC_x, VC_t)$.
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 information carried by x: $VC_t \leftarrow \max(VC_x, VC_t)$.
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: $VC_t(t) \leftarrow VC_t(t) + 1$.

VCs associated with property events precisely capture the causal dependence:

Theorem 3. *$e \sqsubset e'$ iff $VC(e) \leq VC(e')$.*

The proof of Theorem 3 can be easily derived from the one in [20].

Computing sliced atomicity. The atomicity algorithm in [20] can be applied directly on the sliced causality to extract the write-read atomicity. But it can be over-restrictive when compared with the sliced atomicity, which can be captured using the causal dependence computed above as follows:

A counter, called atomicity identifier, is associated with every shared variable, to keep track of its atomic set. Let AI_x denote the atomicity identifier associated with x. An extra VC, denoted VC^{wr}, is associated to every write of a shared variable, containing the information about all the reads following this write, i.e., $VC^{wr}(e) = max(VC(e_1), ..., VC(e_n))$ with $e_1, ..., e_n$ are all the reads following the write event e. VC^{wr} can be computed via a backward traversal of the sliced trace after the causal dependence is computed. Also, we associate two VCs, VC^w_x and VC^r_x, with every shared variable, which are the VC and VC^{wr} of the latest

```
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
         SV(e)(x) = SV(Σ)(x) when both defined, for all shared variables x
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 SV(Σ')(x) ← SV(e)(x)
  if type(e) = release and target(e) = l then SV(Σ')(x) ← undefined
  MS(Σ') ← runMonitor(MS(Σ), e)
  if MS(Σ') = "error" then reportViolation(Σ, e)
  return Σ'
endprocedure
```

Fig. 5. Consistent runs generation algorithm

write of x respectively. Property events are enriched with a new attribute, SV, which is a partial mapping from shared variable into corresponding counters. And the variable atomicity information is updated as follows:

1. $type(e) = write$, $target(e) = x$, and x is a shared variable. Then we need to increase the atomicity identifier of x: $AI_x = AI_x + 1$. Also, the variable's VCs are updated: $VC_x^w = VC(e)$ and $VC_x^r = VC^{wr}(e)$.
2. e is a property event. In this case, $SV(e)(x) = AI_x$ for all x such that $VC_x^w \leq VC(e) \leq VC_x^r$ and $SV(e)(x)$ is undefined for any other x.

Definition 6 and Theorem 3 yield the following corollary, stating that the VC algorithm precisely captures the atomicity:

Corollary 3. *Two events e_1 and e_2 are both in the atomic block $[e]_x$ iff $SV(e_1)(x) = SV(e_2)(x)$.*

Generating consistent runs. We next show 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 5 gives a high-level pseudocode to generate and verify, on a level-by-level basis, potential runs consistent with the sliced causality with 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 $SV(\Sigma)$ which keeps for each x its current atomicity identifier (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 [5].

The pseudocode in Figure 5 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 *VC*s are found to be smaller than or equal to the *VC*s of all the cuts in the current level. In other words, to process an event, a good implementation of the algorithm in Figure 5 would take time $O(|\mathit{Threads}|)$.

7 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 [5] logic-plugins, but we do not discuss those here; the interested reader is referred to [4]. 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 [13].

7.1 Benchmarks

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

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

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

(S.V.) detected, and slowdown ratios after instrumentation [2]. Banking is a simple example taken over from [10], showing relatively classical concurrent bug patterns. Elevator, tsp, sor and hedc come from [22]. 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 [12]. IBM web crawler is a component of the IBM Websphere tested in [9]. [3] 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 [21] 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 in-

[2] Not applicable for some programs, e.g., Elevator.
[3] No source code is available for this program.

voke 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.

7.2 Race Detection

Program	Var to	Trace Size		Running Time (seconds) per S.V.				Races		
	Check	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

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: 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 [15] 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 [15], 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 6. 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)*[4] 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 7. 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.

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

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

Fig. 6. 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);
}
```

Fig. 7. Patched code in WebappClassLoader

7.3 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

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

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

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 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 **??**, 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., [23] and [11]. Moreover, jPredictor found harmful atomicity violations in tsp and hedc that were missed by [23] and [11] 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

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

Fig. 8. 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);
  }
}
```

Fig. 9. Atomic iteration on the map

shown in Figure 8, 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 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 9 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 9, 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.

8 Conclusions

Sliced causality with atomicity was defined in this paper, a novel causality which generalizes the sliced causality from [7] with *relevance* and which incorporates the write/read atomicity from [20]. The obtained causality, which is more than a partial order, allows more sound permutations, or linearizations, of property events than previous causalities. Thus, when used in the context of predictive runtime analysis, it achieves an increased predictive capability (or coverage). We showed that atomicity can also be soundly sliced w.r.t. the property to check,

improving the efficiency of the prediction algorithm. Algorithms to efficiently extract the sliced causality and atomicity were presented and implemented. Evaluation results showed that our technique is viable and effective in practice: many concurrent bugs were revealed in popular Java systems *without* false alarms.

References

1. H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.
2. Apache Commons project. http://commons.apache.org/.
3. Apache FTP server project. incubator.apache.org/ftpserver/.
4. F. Chen and G. Roşu. Predicting concurrency errors at runtime using sliced causality. Technical Report UIUCDCS-R-2005-2660, Department of Computer Science at UIUC, 2005.
5. F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, 2007. to appear.
6. F. Chen and G. Rosu. Parametric and termination-sensitive control dependence. In *SAS*, volume 4134 of *LNCS*, pages 387–404. Springer, 2006.
7. F. Chen and G. Rosu. Parametric and sliced causality. In *CAV*, volume 4590 of *LNCS*, pages 240–253. Springer, 2007.
8. M. Christiaens and K. D. Bosschere. Trade: Data race detection for java. In *International Conference on Computational Science (2)*, volume 2074 of *LNCS*, pages 761–770. Springer, 2001.
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)*, 2003.
11. C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.
12. Java. http://java.sun.com.
13. jPredictor. http://fsl.cs.uiuc.edu/jPredictor/.
14. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of ACM*, 21(7):558–565, 1978.
15. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, pages 308–319, 2006.
16. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, pages 167–178, 2003.
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, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ESEC / SIGSOFT FSE*, pages 337–346, 2003.
20. K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *FMOODS*, volume 3535 of *LNCS*, pages 211–226. Springer, 2005.

21. Apache group. Tomcat. http://jakarta.apache.org/tomcat/.
22. C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.
23. L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPOPP*, pages 137–146, 2006.