Parametric and Sliced Causality

Feng Chen and Grigore Roşu

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

Abstract. Happen-before causal partial order relations have been widely used in concurrent program verification and testing. In this paper, we present a parametric approach to happen-before causal partial orders. All existing variants of happen-before relations can be obtained as instances of the parametric framework for particular properties on the partial orders. A novel causal partial order, called sliced causality, is defined also as an instance of the parametric framework, which loosens the obvious but strict happens-before relation by considering static and dynamic dependence information about the program. Sliced causality has been implemented in a concurrent runtime verification tool for Java, named JPREDICTOR, and the evaluation results show that sliced causality can significantly improve the capability of concurrent verification and testing on multi-threaded Java programs.

1 Introduction

Concurrent systems are notoriously difficult to analyze, test and debug due to their inherent nondeterminism. The *happen-before* causality, first introduced in [15], provides an effective way to analyze the potential dynamic behaviors of concurrent systems and has been widely used in concurrent program verification and testing [24, 17, 20, 21, 8]. Intuitively, happen-before based approaches extract happen-before causal partial orders by analyzing thread communication in the observed execution; the extracted causal partial order can be regarded as an abstract model of the runtime behaviors of the program, which can further be investigated exhaustively against the desired property. This way, the user *does not need to re-execute* the program to detect potential errors. Happen-before based approaches are sound, meaning that no false alarms will be produced; and they can handle general purpose properties, e.g., temporal ones.

Several variants of happen-before causalities have been introduced for applications in different domains, e.g., distributed systems [15] and multi-threaded system [17, 21]. Although all these notions of happen-before causal partial orders are similar in principle, there is no adequate unifying framework for all of them, resulting in confusion and difficulty in understanding existing work. Morever, proofs of correctness have to be redone for every variant of happen-before causality, involving sophisticate details. This may slow future developments, in particular defining new, or domain-specific happen-before relations. The first contribution of this paper is a parametric framework for happen-before causal partial orders, which is purposely designed to facilitate defining and proving correctness of different happen-before causalities. The proof of correctness of a particular happen-before relation is reduced to proving a *closed* and *local* property of the causal partial order. All variants of happen-before relations that we are aware of can be obtained as instances of our framework.

The second contribution of this paper consists of defining a new happen-before relation, called *sliced causality*, within our parametric framework, which aims at improving coverage of the analysis 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.. Previous approaches based on happen-before (such as [17, 20, 21]) 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. 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 multi-threaded execution.

Let us consider a simple and common safety property for a shared resource, that any access should be authenticated. Figure 1 shows a buggy multi-threaded program using the shared resource. The main thread acquires the authentication and then the task thread uses the authenticated resource. They communicate via the `flag` variable. Synchronization is unnecessary, since only the main thread modifies `flag`. However, the developer makes a (rather common [6]) mistake by using `if` instead of `while` in the task thread. Suppose now that we observed a successful run of the program, as shown by the arrow. The traditional happen-before will not be able to find the bug because of the causality induced by write/read on `flag`. But since `resource.access()` is *not* controlled by `if`, our technique will be able to correctly predict the violation from the successful execution. When the bug is fixed by replacing `if` with `while`, `resource.access()` will be controlled by `while` because it is a non-terminating loop, then no violation will be reported by our technique.

```
Main Thread:                    Task Thread:

resource.authenticate()
flag.set = true;
                                if (! flag.set)
                                   flag.wait();
                                resource.access();
```

Fig. 1. Multi-threaded execution

We next explain our sliced causality 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 depends –this notion will be formally defined in Section 4.1) 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 multi-threaded system. If a permutation violates the property, then the system can do so in another

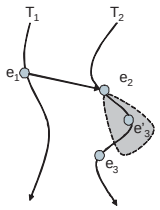

Fig. 2. Sliced causality

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 traditional happens-before approach would *not* work here.

One should not confuse the notion of sliced causality introduced in this paper with the existing notion of *computation slicing* [20]. 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.

2 Happen-Before Causalities

The first happen-before relation was introduced almost 3 decades ago by Lamport [15], to formally model and reason about concurrent behaviors of distributed systems. Since then, a plethora of variants of happen-before causal partial order relations have been introduced in various frameworks and for various purposes. The basic idea underlying happen-before relations is to observe the events generated by the execution of a distributed system and, based on their order, their type and a straightforward causal flow of information in the system (e.g., the receive event of a message follows its corresponding send event), to define a partial order relation, the happen-before causality. Two events related by the happen-before relation are causally linked in that order.

When using a particular happens-before relation for (concurrent) program analysis purposes, the crucial property of the happen-before relation that is needed is that, for an observed execution trace τ, other *sound permutations* of τ, also called *linearizations* or *linear extensions* or *consistent runs* or even *topological sortings* in the literature, are also possible computations of the distributed system. Consequently, if any of these linearizations violates or satisfies a property φ, then the system can indeed violate or satisfy the property, regardless of whether the particular observed execution that generated the happen-before relation violated or satisfied the property, respectively. For example, [5] defines for each trace property φ, complex formulae *Definitely*(φ) and *Possibly*(φ), which hold in all and, respectively, in at least one possible linearization of the happen-before causality.

In order to prove the main property of a happen-before causality defined in a particular concurrency setting, namely the feasibility of the other linearizations of the happen-before partial order extracted from observing one particular execution, one needs to define the actual computational model and to formally state what a concurrent com-

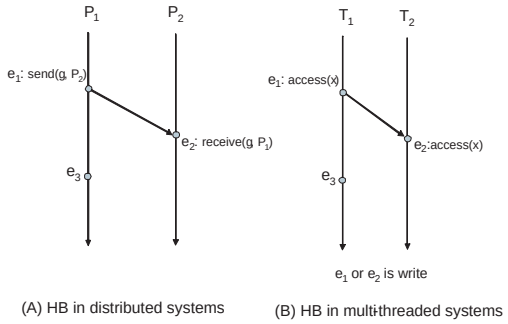

Fig. 3. Happen-before partial-order relations

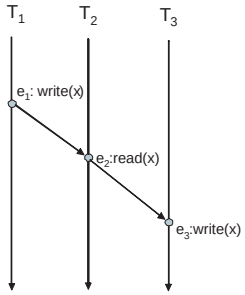

Fig. 4. Happen-before in multi-threaded systems

putation is. These definitions tend to be rather intricate and particular. For that reason, proofs of the property above need to be redone in different settings facing different "details", even though they follow conceptually the same idea. In the next section we present a simple and intuitive property on traces, called *feasibility*, which ensures the desired property of the happen-before causality and which appears to be easy to check in concrete situations.

To show how the various happen-before causalities fall as special cases of our parametric approach, we recall two important happen-before partial orders, one in the context of distributed systems where communication takes place exclusively via message passing, and another in the context of multi-threaded systems, where communication takes place via shared memory. In the next section we'll show that their correctness results [2, 21] follow as corollaries of our main theorem. In Section 4 we define another happen-before causality, called *sliced causality*, which uses static analysis information about the multi-threaded program in a rather non-trivial manner. The correctness of sliced causality will also follow as a corollary of our main theorem in the next section.

In the original setting of [15], a distributed system is formalized as a collection of processes communicating only by means of asynchronous message passing. A process is a sequence of events. An event can be a *send* of a message to another process, a *receive* of a message from another process, or an *internal* (local) event.

Definition 1. *Let τ be an execution trace of a distributed system consisting of a sequence of events as above. Let E be the set of all events appearing in τ, and let the* ***happen-before*** *partial order on E, $\rightarrow$, be defined as follows:*

1. *if e_1 appears before e_2 in some process, then $e_1 \rightarrow e_2$;*
2. *if e_1 is the send and e_2 is the receive of the same message, then $e_1 \rightarrow e_2$;*
3. *$e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ implies $e_1 \rightarrow e_3$.*

A space-time diagram to illustrate the above definition is shown in Figure 3 (A), in which e_1 is a send message, e_2 is the corresponding receive message; $e_1 \rightarrow e_2$ and $e_1 \rightarrow e_3$, but e_2 and e_3 are not related. It is easy to prove that $(E, \rightarrow)$ is a partial order.

The crucial property of this happen-before relation, namely that all the permutations of τ consistent with $\rightarrow$ are possible computations of the distributed system, was proved in [2] using a specific formalization of the global state of a distributed system, as a "cut"

in each of the total orders corresponding to processes. This property will also follow as an immediate corollary of our main theorem in the next section.

Happen-before causality relations have been devised in the context of multi-threaded systems for various purposes. For example, [16, 17] propose datarace detection techniques based on intuitive multi-threaded variants of happen-before causality; [21] proposes a happen-before relation to consider read/read accesses to shared memory as unrelated, and [22] one which even drops the write/write conflicts, but which relates each write with all its subsequent reads atomically.

A multi-threaded system is a collection of threads that communicate via synchronization operations and reads/writes on shared variables. Devising appropriate happen-before causalities for multi-threaded systems is a non-trivial task. The obvious approach would be to map the inter-thread communication in a multi-threaded system into send/receive events in some corresponding distributed system. For example, starting a new thread generates two events, namely a send event from the parent thread and a corresponding receive event from the new thread; releasing a lock is a send event and acquiring a lock is a receive event. A write on a shared variable is a send event while a read is a receive event. However, such a simplistic mapping suffers from several problems related to the semantics of shared variable accesses. First, every write on a shared variable can be followed by multiple reads whose order should not matter; in other words, some "send" events now can have multiple corresponding "receive" events. Second, consider the example in Figure 4. Since e_3 may write a different value into x other than e_1, the value read at where e_2 occurs may change if we observe e_3 after e_1 but before e_2 appears. Therefore, $e_1e_3e_2$ may not be a trace of some feasible execution since e_2 will not occurs any more. Hence, a causal order between e_2 and e_3 should be enforced too, which cannot be captured by the original happen-before relation in [15].

The various definitions of causal partial orders for multi-threaded systems in the works mentioned above address these problems (among others). However, each of them still needs to be proved correct: any sound permutation of its events results in a feasible execution of the multi-threaded system. If one does not prove such a property for one's desired happens-before causality, then one's techniques can lead to false alarms. We next recall one of the simplest happens-before relations for multi-threaded systems, namely the one in [21]:

Definition 2. *Let τ be an execution trace of a multi-threaded system, let E be the set of all events appearing in τ, and let the **happen-before** partial order on E, $\rightsquigarrow$:*

1. *if e_1 appears before e_2 in some thread, then $e_1 \rightsquigarrow e_2$;*
2. *if e_1 and e_2 are two accesses on the same shared variable such that e_1 appears before e_2 in the observed execution and one of them is a write, then $e_1 \rightsquigarrow e_2$;*
3. *$e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_3$ implies that $e_1 \rightsquigarrow e_3$.*

Figure 3 (B) illustrates this multi-threaded causality. e_1 and e_2 access x and one of them is a write. We have $e_1 \rightsquigarrow e_2$ and $e_1 \rightsquigarrow e_3$, but e_2 and e_3 are not comparable under $\rightsquigarrow$.

3 Parametric Framework for Causality

Different definitions of happen-before relations in different settings raise technical difficulties in proving the feasibility of linearizations of the extracted causal partial order,

which is crucial for the correctness of techniques using those causalities. In this section, we define a parametric framework that axiomatizes the notions of causalities over events and *feasibility* of traces in a system-independent manner, which helps the understanding of the essence of a correct causality relation. And more interestingly, we show that using this framework, proving the feasibility of the linearizations of a certain extracted causal partial order can be reduced to checking a "closure" local property of feasible traces, an easier task to achieve in most cases.

Definition 3. *Let* Events *be the set of all* events. $\mathsf{Traces} \subseteq \mathsf{Events}^*$ *is the set of* traces *with events in* Events*; each event appears* at most once *in a trace; if event e appears in trace τ, we write $e \in \tau$. If τ is a trace, let ξ_τ be the set of events in τ, i.e., $\xi_\tau = \{e \mid e \in \tau\}$ and let $<_\tau$ be the total order on ξ_τ, given by the order of appearance of events in τ.*

Therefore, trace τ can be identified with total orders $(\xi_\tau, <_\tau)$. Note that our notions of events and traces are rather abstract and generic: no concrete meaning has been associate to events and not all traces are necessarily feasible in a given system, that is, not all of them can be generated by actual executions; and a trace can be complete (generated by terminated executions) or incomplete (generated by non-terminated executions). We will refine them later according to specific contexts of different desired causal partial orders. For example, in our sliced causality, events are mapping of attributes to values (Definition 10) and a special *counter* attribute is used to distinguish different occurrences of events with the same attribute-values pairs, ensuring that every event appears at most once in the trace.

Definition 4. *A* ***causality operator*** *is a partial function C: $\mathsf{Traces} \rightarrow\!\!\!\!\!\circ\; \mathcal{PO}(\mathsf{Events})$ with the property that if $C(\tau)$ is defined and equal to $(\xi, <)$, then $\xi = \xi_\tau$ and $< \subseteq <_\tau$.*

$\mathcal{PO}(\mathsf{Events})$ is the set of partial order sets over Events and C maps feasible traces into partial order sets. Since not all traces of events are feasible, C is a partial function, as indicated by $\rightarrow\!\!\!\!\!\circ$. A feasible trace τ is therefore a linearization of $C(\tau)$, and there are many more linearizations that are not necessarily feasible traces. Based on C, we can define an equivalence relation on traces.

Definition 5. *Let $\equiv_C$ be the equivalence given by the "strong" kernel of C, that is, $\tau \equiv_C \tau'$ iff $C(\tau)$ and $C(\tau')$ are both defined and $C(\tau) = C(\tau')$.*

Note that $\tau \equiv_C \tau'$ implies that the two traces have the same events; in particular, they have the same size. Intuitively, τ and τ' are linearizations of the same causality.

Definition 6. *Given a causality operator C, a set of traces $\mathcal{F} \subseteq \mathsf{Traces}$ is* ***C-feasible*** *iff*

1. *C is defined on $\mathcal{F}$; and*
2. *If $\tau = \tau_1 e_1 e_2 \tau_2$ is some trace in $\mathcal{F}$ such that $C(\tau) = (\xi, <)$, and $e_1 \not< e_2$ then $\tau_1 e_2 e_1 \tau_2$ is also in $\mathcal{F}$.*

In the above definition, the feasibility of the set of traces is constructed on a local property, namely, if two successive events are not ordered under the desired causality, one should be able to exchange their occurrences in another execution of the system. We then prove the following theorem showing that this local property indeed enforces the feasibility of linearization of the extracted causality.

Theorem 1. *C-feasible sets of traces are closed under $\equiv_C$. In other words, if $\mathcal{F}$ is C-feasible, $\tau \in \mathcal{F}$ and $\tau \equiv_C \tau'$, then $\tau' \in \mathcal{F}$.*

Proof: Recall that,since $\tau \equiv_C \tau'$, τ and τ' have the same length. We prove this theorem by the well-founded induction on the length of the suffixes of τ and τ' starting with the first position, say $k+1$, on which τ and τ' are different (in other words, we assume that τ and τ' are identical on their first k events, where $k \geq 0$, but they have different events on position $k+1$).

Suppose that $\tau = e_1...e_k e_{k+1}\omega$ and $\tau' = e_1...e_k e\omega'$ for some $k \geq 0$, where $e \neq e_{k+1}$. Since $\tau \equiv_C \tau'$, the two traces have the same events. Therefore, ω must contain the event e, that is, must have the form $\omega_1 e \omega_2$ for some (empty or not) sequences of events ω_1 and ω_2. Let $(\xi, <)$ be the partial order $C(\tau)$. Since $C(\tau') = C(\tau)$, it follows that $< \subseteq <'_\tau$. In other words, $\{e' \in \xi \mid e' < e\} \subseteq \{e_1, ..., e_k\}$. Since $\tau = e_1...e_k e_{k+1}\omega_1 e \omega_2$, it follows that there is no event e' in $e_{k+1}\omega_1$ such that $e' < e$.

Using the definition of feasibility iteratively as many times as events in $e_{k+1}\omega_1$ and starting with τ, we can move e in front of $e_{k+1}\omega_1$; in other words, we get a new trace $\tau'' = e_1...e_k e e_{k+1}\omega_1\omega_2$ in $\mathcal{F}$ with $C(\tau'') = C(\tau) = C(\tau')$. The suffixes of τ'' and τ' starting with the first position on which τ'' and τ' have different events are now smaller, so we can use the induction hypothesis and conclude that $\tau' \in \mathcal{F}$. □

This theorem generalizes the classic theorem about permutations and transpositions, which states that any permutation is a product of adjacent transpositions [12], by extending the result to linearizations of partial orders. Assuming that $\mathcal{F}$ is the set of traces generated by feasible executions, this theorem states that every sound permutation of the extracted causal partial order is feasible in the system under analysis if the feasibility of $\mathcal{F}$ holds for the definition of the desired causality, because happen-before partial orders are extracted from observed executions of a system, which are feasible by definition.

3.1 Instantiations for Happen-Before Causalities

We next show that the two variants of happen-before discussed in Section 2 can be captured as instances of our parametric framework. For the happen-before relation defined in Definition 1, let Events_{hb} be the set of all the send, receive and internal events.

Definition 7. *Let C_{hb} be a partial function: $\mathit{Traces}_{hb} \twoheadrightarrow \mathcal{PO}_f(\mathit{Events}_{hb})$ such that, for $\tau \in \mathit{Traces}_{hb}$ and E_τ the set of events in τ, $C_{hb}(\tau) = (E_\tau, \rightarrow)$. Let $\mathcal{F}_{hb}$ be the set of traces that are computations of the distributed system as defined in [2].*

Lemma 1. *C_{hb} is causality operator.*

Proof: It follows by Definition 1 and Definition 4. □

Lemma 2. *$\mathcal{F}_{hb}$ is C_{hb}-feasible.*

Proof: Let $\tau = \tau_1 e_1 e_2 \tau_2$ be an observed computation of a distributed system such that e_1 and e_2 are not comparable under $\rightarrow$. That is, e_1 and e_2 are not in the same process and

e_1 is not a sending of a message that is received by e_2. It is easy to show by enumerating possible combinations of types of e_1 and e_2 that, for $\tau' = \tau_1 e_2 e_1 \tau_2$, using the definition of global states in [2], the global states right after τ_1 and right before τ_2 in τ' are identical to those in τ. As a consequence, the global state in τ' generated after e_2 is reachable from the initial state and can reach the final state of τ' which are identical to those of τ. So τ' is also a computation of the system. □

Proposition 1. *For an observed trace τ of a distributed system, any permutation of τ consistent with $\rightarrow$ is a possible computation of the system.*

Proof: It follows by Lemma 1, Lemma 2 and Theorem 1. □

For the happen-before relation defined in Definition 2, let $Events_{mhb}$ be the set of all the write and read events on shared variables as well as all internal events.

Definition 8. *Let C_{mhb} be a partial function: $Traces_{mhb} \twoheadrightarrow \mathcal{PO}_f(Events_{mhb})$ such that, for $\tau \in Traces_{mhb}$ and E_τ is the set of events in τ, $C_{mhb}(\tau) = (E_\tau, \rightsquigarrow)$. Let $\mathcal{F}_{mhb}$ be the set of traces that are generated by feasible executions of the multi-threaded system.*

Note that any execution of a multi-threaded system is the result of a given input and a certain thread scheduling of the system.

Lemma 3. *C_{mhb} is causality operator.*

Proof: It follows by Definition 2 and Definition 4. □

Lemma 4. *$\mathcal{F}_{mhb}$ is C_{mhb}-feasible.*

To prove this lemma, we first need to define the state of the system:

Definition 9. *In a multi-threaded system, the **thread state** is a tuple, $< M, Prog, P >$, where M is the state of the memory accessible by the thread, $Prog$ is the code executed by the thread, and P is the pointer to the instruction to execute next. The **global state** of the system is composed of the thread states in the system.*

Obviously, the action made by a thread at any moment is determined by its state and the action made by the system is determined by the global state.

Proof: Let $\tau = \tau_1 e_1 e_2 \tau_2$ be an observed trace of a multi-threaded system such that e_1 and e_2 are not comparable under $\rightsquigarrow$. That is, e_1 and e_2 are not in the same thread, and either e_1 and e_2 do not access the same shared variable or both of them are reads. Suppose e_1 is generated by thread T_1 and e_2 by T_2. Let the thread state of T_1 right after τ_1 is generated be S_{T_1}, the thread states of T_2 after τ_1 and e_1 be S_{T_2} and S'_{T_2} respectively, and the global state of the system after $\tau_1 e_1 e_2$ is S_g. We can re-execute the system using the same input as for the execution generating τ; by following the same thread scheduling, the same event trace are observed. Once the end of τ_1 is encountered, we change the thread scheduling to execute T_2 instead of T_1, where we have S_{T_1} and S_{T_2} as the thread states for T_1 and T_2 respectively. Then e_2 will be generated unless e_2 is a read of a memory location that has different values in S_{T_2} and S'_{T_2}, which is

impossible because, otherwise, e_1 is a write of that memory location and $e_1 \rightsquigarrow e_2$ in τ. After e_2 is observed, we pause T_2 and start T_1. Similar to the above argument, e_1 will be generated. Let the global state of the system after e_1 is generated in this run be S'_g. By enumerating the possible combinations of types of e_1 and e_2, we can show that to make S_g and S'_g different, e_1 and e_2 should both be writes of the same shared location, which contradicts the hypothesis. Hence, $S_g = S'_g$. Therefore we can continue from S'_g using the input and the thread scheduling of the execution generating τ to generate τ'. In other words, τ' is a feasible execution of the system. □

Proposition 2. *For an observed trace τ of a multi-threaded system, any permutation of τ consistent with $\rightsquigarrow$ is a feasible execution of the system.*

Proof: It follows by Lemma 3, Lemma 4 and Theorem 1. □

4 Sliced Causality

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 [22]. In what follows we show that one can construct a much less restrictive causal partial order, called *sliced causality*, by making use of dependence information obtained statically and dynamically. Briefly, instead of computing the causal partial order on all the observed events like in the traditional happen-before based approaches [21], our approach first slices τ according to the desired property and then computes the causal partial order on the achieved slice; the slice contains all the events relevant to the property, as well as all the events upon which the relevant events depend. This way, irrelevant causality on events is trimmed without breaking the soundness of the approach, allowing more permutations of relevant events to be analyzed and resulting in better coverage of the analysis.

We employ dependencies among events to assure correct slicing. The dependence discussed here somehow relates to *program slicing* [10, 25], but we focus on finer grained dynamic units, namely events, instead of statements. Our analysis keeps track of actual memory locations in every event, available at runtime, which avoids inter-procedural analysis. 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 ([3]).

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 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 dependence only relates events generated by the same thread, the data-flow dependence may relate 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'.

Events and Traces. Events play a crucial role in our approach, representing atomic steps in the execution of the program. In this paper, we focus on multi-threaded programs and consider the following types of events: writes/reads on variables, beginning/ending of function invocations, acquiring/releasing locks, and starts and exits of threads. 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 10. *An **event** is a mapping of **attributes** into corresponding **values** and a **trace** is a sequence of events. From now on in the paper, we assume an arbitrary but fixed trace τ, and let ξ denote ξ_τ (also called **concrete events**) for simplicity.*

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. Note that 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. Besides, 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*). This way, all events appearing in a trace can be assumed different.

4.1 Control Dependence on Events

Informally, if a change of $state(e)$ may affect the occurrence of e', then we say that e' has a *control dependence* on e, and write $e \sqsubset_{ctrl} e'$. For example, in Figure 5, 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. Control dependence occurs inside of a thread, so we first define the total order within one thread:

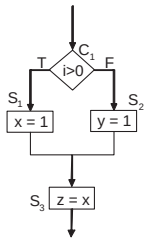

Fig. 5. Control dependence

Definition 11. *Let $<$ denote the union of the total orders on events of each thread, i.e., $e < e'$ iff $thread(e) = thread(e')$ and $e <_\tau e'$.*

The control dependence among events in sliced causality is parametric in a control dependence relation among statements. In particular, one can use off-the-shelf algorithms for classic [7] or for weak [18] control dependence. All we need 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 depend on C, for some appropriate notion of control dependence. We regard lock acquire statements as control statements that control all the following statements.

For simplicity, we assume that any control statement generates either a *read* event (the lock acquire is regarded as a read on the lock) or no event (the condition is a constant) when checking its condition. For control statements with complex conditions, e.g., involving function calls and side effects, we can always transform the program: one can compute the original condition before the control statement and use a fresh boolean variable to store the result and be the new condition of the control statement.

Definition 12. *$e \sqsubset_{ctrl} e'$ iff $e < e'$, $stmt(e') \in scope(stmt(e))$, and e is "largest" with this property, i.e., there is no e'' such that $e < e'' < e'$ and $stmt(e') \in scope(stmt(e''))$.*

Intuitively, e is control dependent on the *latest* event issued by some statement upon which $stmt(e)$ depends. For example, in Figure 5, a write of x at S_1 is control dependent on the most recent read of i at C_1 but not on previous reads of i at C_1. This way, we obtain a minimal yet precise control dependence relation on events.

The *soundness* (no false positives) of our runtime analysis technique is contingent to the *correctness* (no false negatives) of the employed static dependence 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 [18] when calculating the control scopes. If termination information of loops is available, termination-sensitive control dependence [4] can be utilized to provide correct and more precise results; this is the control dependence that we are currently using in jPredictor. One can also try to use the classic control dependence [7] instead, but one should be aware that false bugs may be reported (e.g., when the developer implements synchronization using "infinite" loops).

4.2 Data Dependence on Events

If a change of $state(e)$ may affect $state(e')$, we say e' has a *data dependence* on e:

Definition 13. *$e \sqsubset_{data} e'$ 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 $target(e')$.*

The first case encodes the common data dependence. For example, for an assignment $x := E$, the write of x data depends on all the reads generated by the evaluation of E. The second case captures the interference dependence [14] in multi-threaded programs: a read depends on *the most recent* write of the same memory location. For instance, in Figure 5, if the observed execution is $C_1 S_1 S_3$ then the read of x at S_3 is data dependent on the most recent write of x at S_1. We treat lock release as a write on the lock and lock acquire as a read. The third case is more intricate and related to what is called

"relevant dependence" in [9]. Assuming execution $C_1S_2S_3$ in Figure 5, no type *1* or *2* data dependence can be found. However, a change of the value of the read of *i* at C_1 can potentially change the value of the read of *x* at S_3: if the value of *i* changes then C_1 may choose to execute the branch of S_1, resulting in a new write of *x* that may change the value of the read of *x* at S_3. Therefore, we say that the read of *x* at S_3 is data dependent on the read of *i* at C_1, as defined in case *3*. Note that although the dependence is caused by the potential change of a control statement, it can *not* be caught by control dependence; for example, the read of *x* at S_3 is *not* control dependent on the read of *i* at C_1 since $S_3 \notin scope(C_1)$. Aliasing information of variables is needed to correctly compute the data dependence defined in case *3*, which one can obtain using any available techniques.

Note that there are no write-write, read-read, read-write data dependencies. Specifically, case *2* only considers the write-read data dependence, enforcing the read to depend upon only the latest write of the same variable. In other words, a write and the following reads of the same variable form an *atomic* block of events. This captures the work presented in [22], in the much more general setting of this paper.

4.3 Slicing Causality Using Relevance

When checking a trace τ against a property φ, not all the events in τ are relevant to φ; for example, to check dataraces on a variable *x*, accesses to other variables or function calls are not needed. Moreover, the *state* attributes of some relevant events may not be relevant; for example, the particular values written to or read from *x* for datarace detection. We next assume a generic *filtering function* that can be instantiated, usually automatically, to concrete filters depending upon the property φ under consideration:

Definition 14. *Let* α: $\mathsf{Events} \twoheadrightarrow \mathsf{Events}$ *be a partial function, called a* ***filtering function****. The image of* α*, that is* $\alpha(\mathsf{Events})$*, is written more compactly* Events_α*; its elements are called* ***abstract relevant events****, or simply just* ***relevant events****. All thread start and exit events are relevant. If* $\alpha(e)$ *is defined, then* $key(\alpha(e)) = key(e)$ *for any attribute* $key \neq state$*;* $state(\alpha(e))$ *is either undefined or equal to* $state(e)$*.*

We extend α to traces and sets of events:

Definition 15. *Let* $\alpha(\tau)$*, written more compactly* τ_α*, be the trace of relevant events obtained by applying* α *on each event in* τ *(Skip if undefined). Let* ξ_α *denote* ξ_{τ_α}*.*

This relevance-based abstraction plays a crucial role in increasing the predictive power of our analysis approach: in contrast to the concrete event set ξ, the corresponding abstract event set ξ_α allows more permutations of abstract events; instead of calculating permutations of ξ and then abstracting them into permutations of ξ_α like in traditional happen-before based approaches, e.g., [21], we will calculate valid permutations of a *slice* of $\xi \cup \xi_\alpha$ that contains only events (directly or indirectly) relevant to φ. This slice is defined using the dependence on concrete and abstract events.

Definition 16. *All dependence relations are extended to abstract relevant events: if* $e < / \sqsubset_{ctrl} / \sqsubset_{data} e'$ *then also* $\alpha(e) < / \sqsubset_{ctrl} / \sqsubset_{data} e'$ *and* $e < / \sqsubset_{ctrl} / \sqsubset_{data} \alpha(e')$ *and* $\alpha(e) < / \sqsubset_{ctrl} / \sqsubset_{data} \alpha(e')$*, whenever* $\alpha(e)$ *and/or* $\alpha(e')$ *is defined;* $\sqsubset_{data}$ *is extended only when* $state(\alpha(e'))$ *is defined.*

We next introduce a novel dependence relation on events, motivated by *potential* occurrences of relevant events. Consider the example in Figure 5 again. Suppose that the relevant events are only the writes of y and z. For the execution $C_1S_1S_3$, only one relevant event is observed, namely the write of z at S_3, which is neither control not data dependent on the read of i generated at C_1. Yet, the fact that the write of z event at S_3 is the only relevant event in this execution does *depend* on the read of i at C_1: if the other branch would have been chosen at C_1, then a write of y (relevant) event would have been generated as well. This subtle dependence has crucial implications on the correctness of happen-before causalities sliced using relevance. Indeed, if the read of i event at C_1 depends on some event generated by another thread, say e, then the write of z event at S_3 and e *cannot* be permuted, since we have no knowledge about the new relevant event which could occur. Since this dependence makes sense only in the context of an event filtering function α, we call it *relevant dependence*:

Definition 17. *For $e \in \xi, e' \in \xi_\alpha$, we write $e \sqsubset_{rlvn} e'$ iff $e < e'$, $stmt(e') \notin scope(stmt(e))$, and there is a statement $S \in scope(stmt(e))$ that may generate a relevant event.*

Intuitively, if $e \sqsubset_{rlvn} e'$ then a change of $state(e)$ may cause a new relevant event to occur before e'. This may invalidate some permutations of ξ_α.

Definition 18. *Let $\sqsubset$ be the relation $(\sqsubset_{data} \cup \sqsubset_{ctrl} \cup \sqsubset_{rlvn})^+$. If e and e' are concrete or relevant events with $e \sqsubset e'$, then we simply say that e' **depends upon** e.*

Now we are ready to define the *relevant slice* of events:

Definition 19. *Let $\overline{\xi_\alpha} \subseteq \xi \cup \xi_\alpha$ extend ξ_α with events $e \in \xi$ such that $e \sqsubset e'$ for some $e' \in \xi_\alpha$. Let $\overline{\tau_\alpha}$ be the **abstract trace** of τ, i.e., the total order on $\overline{\xi_\alpha}$ imposed by $<_\tau$.*

For example, in Figure 2, for the property "e_1 must happen before e_3", we have that $\overline{\xi_\alpha} = \xi_\alpha = \{e_1, e_3\}$, while for "$e_1$ must happen before e'_3", $\xi_\alpha = \{e_1, e'_3\}$ but $\overline{\xi_\alpha} = \{e_1, e_2, e'_3\}$ since $e_2 \sqsubset e'_3$. Our goal next is to define an appropriate notion of causal partial order on $\overline{\xi_\alpha}$ and then to prove that any permutation consistent with it is sound.

Definition 20. *Let $\prec \stackrel{def}{=} (< \cup \sqsubset)^+ \subseteq \overline{\xi_\alpha} \times \overline{\xi_\alpha}$ be the **sliced causality**.*

Note that the sliced causal partial order was defined on more events than those in ξ_α. We next show that sliced causality is an instance of the parametric framework in Section 3, so any permutation of relevant events consistent with sliced causality is sound.

Definition 21. *Let C_α: $Traces \rightarrow\!\!\!\!\circ\, \mathcal{PO}(Events)$ be the partial function defined as $C_\alpha(\tau) = (\xi_\tau, \prec)$ for each $\tau \in Traces$. Let $\mathcal{F}_\alpha \subseteq Traces$ be the set of all abstract traces that can be abstracted from executions of the program, i.e., for each $\tau_F \in F_\alpha$, there is a terminating execution generating a trace τ such that $\overline{\tau'_\alpha} = \tau_F$.*

Lemma 5. *C_α is a causality operator.*

Proof: It follows by Definition 4, Definition 20, and Definition 21. □

Definition 22. *An execution **generates** a trace that is composed of the events observed during the execution; if an event e appears in the trace generated by an execution π, we say that $e \in \pi$. Traces are denoted by τ, τ', and etc., if they are supposed to be generated by a terminated execution; otherwise they are denoted by ω, ω', and etc.. An execution π' **contains** a trace ω if and only if $\xi_\omega \subseteq \xi_{\omega'} \cup \xi_{\omega'_\alpha}$ and ω is consistent with $<'_\omega$ for the trace ω' generated by π'.*

Definition 23. *For a trace ω and an event e, we have a **cut of** e **in** ω, defined as $\delta_{e,\omega} = \{e' \mid e' \in \omega \text{ and } e' \prec e\}$. A trace ω is called a **feasible prefix (of π)** if and only if there exist an (incomplete) execution π' containing ω, $\omega'_\alpha = \omega_\alpha$ for the trace ω' generated by π, and for any $e \in \omega$, we have that $e' \in \xi_\omega$ for any event $e' \prec e$.*

Intuitively, ω is a feasible prefix if it can be observed during some actual execution and contains all the generated relevant events. Besides, the feasible prefix is closed under $\prec$. It is easy to prove that if $\omega\omega'$ is a feasible prefix then ω is a feasible prefix; in other words, a prefix of a feasible prefix is also a feasible prefix.

Lemma 6. *If ωe is a feasible prefix of π, ω' is a feasible prefix of π', $e \notin \pi'$, and $\delta_{e,\omega} = \delta_{e,\omega'}$ then $\omega' e$ is also a feasible prefix.*

Proof: Let us continue π' by keeping the thread *thread*(e) running until the statement *stmt*(e) is encountered. *stmt*(e) will executed in this run: if there is a control statement C with *stmt*(e) $\in$ *scope*(C) and the latest execution of C in π generates an event e', then $e' \prec e$; thus $e' \in \delta_{e,\omega} = \delta_{e,\omega'}$, that is to say, C will make the same decision in π' as in π, namely to execute the branch containing *stmt*(e). Let the new execution achieved by extending π' to execute *stmt*(e) be π''. When *stmt*(e) is executed in π'', the generated event e'' will have the same attributes as e: for the *state* attribute, all events used to compute the state of e in π are also preserved in π' according to data dependence, hence *state*(e'') = *state*(e); for the *counter* attribute, ω and ω' have the same occurrences of the events that have identical attributes as e except for *counter* because $<\subseteq\prec$, hence *counter*(e'') = *counter*(e); other attributes are decided by the positions (e.g., *thread* and *statement*) of the events and thus identical in both events. Therefore, π'' contains $\omega' e$.

We next prove that no new relevant events are generated before π''. If a new relevant event e_r is generated in π'', it can only occur after π' since all relevant events generated by π' are in ω'; similarly, e_r does not occur in π. The only possibility for this is that there is a control statement C, one of whose branch contains *stmt*(e_r), and π and π'' executes different branches. Let the last execution of C in π'' generates event e_C. $e_C \not\prec e$ because, otherwise, $e_C \in \delta_{e,\omega'} = \delta_{e,\omega}$, meaning that π executes the the same branch of C as π''. Hence, *stmt*(e) $\notin$ *scope*(C). we can always find one relevant event e'_r that occurs no before than e, considering that the exit of the thread is a relevant event. $e'_C \sqsubset_{rlvn} e'_r$ by Definition 17 and thus $e_C \prec e$, contradiction encountered. So no new relevant events can be generated in π'', that is, $\omega' e$ is a feasible prefix.

□

Lemma 7. *If $\omega\omega''$ is a feasible prefix, ω' is a feasible prefixes of π, and for any event $e \in \omega''$, $e \notin \pi$ and $\delta_{e,\omega} = \delta_{e,\omega'}$, then $\omega'\omega''$ is also a feasible prefix.*

Proof: We prove by induction on the length of ω''. If ω'' is empty, then the result follows immediately. Now suppose that $\omega'' = e\omega'''$. By Lemma 6, $\omega' e$ is a feasible prefix of π' that extends π to execute *stmt*(e). Now suppose that there is an event $e' \in \omega'''$ with $e' \in \pi'$. Obviously, *thread*(e') $\neq$ *thread*(e). That is to say, *thread*(e) is blocked by *thread*(e') before executing *stmt*(e). The only way to block a thread here is the lock acquire; in other words, there is a lock aquire operation before *stmt*(e) in π', which generates the event e_l. $e_l \sqsubset_{ctrl} e$ and thus $e_l \in \xi_{\omega'}$, meaning that *thread*(e) has acquired the lock in π. Therefore, *thread*(e) can not be blocked in π', that is, $e' \notin \pi'$ for any $e' \in \omega'''$. Moreover, for any event $e' \in \omega'''$, $\delta_{e',(\omega e)} = \delta_{e',(\omega' e)}$ since $\delta_{e',\omega} = \delta_{e',\omega'}$. Then by induction hypothesis, the result follows.

□

Lemma 8. *If $\omega_1\omega_2 \in \mathcal{F}_\alpha$, ω_3 is a feasible prefix, and $\xi_{\omega_1} = \xi_{\omega_3}$, then $\omega_3\omega_2 \in \mathcal{F}_\alpha$.*

Proof: Obviously, $\omega_1\omega_2$ is a feasible prefix. Then, by Lemma 7, $\omega_3\omega_2$ is a feasible prefix; obviously, $\xi_{\omega_1\omega_2} = \xi_{\omega_3\omega_2}$. Suppose that an execution π contains $\omega_3\omega_2$. Then π is a terminated execution because there is a corresponding thread exit for every thread beginning considering that $\omega_1\omega_2$ contains all relevant events of a terminated execution π''. Let the traces generated by π and π' be τ' and τ'' respectively. To prove that $\overline{\tau'_\alpha} = \omega_3\omega_1$, we only need to show that $\overline{\xi_{\tau'}} = \omega_3\omega_2$ because $\overline{\tau'_\alpha}$ and $\omega_3\omega_1$ are both consistent with $<_{\tau'}$. $\overline{\xi_{\tau'}} = \overline{\xi_{\tau''}}$, because $\tau'_\alpha = (\omega_3\omega_2)_\alpha$ and $(\omega_1\omega_2)_\alpha = \tau_\alpha$ considering that $\prec$ is fixed. Therefore, $\overline{\xi_{\tau'}} = \omega_3\omega_2$ because $\overline{\xi_{\tau'}} = \omega_1\omega_2$, that is, $\omega_3\omega_2 \in \mathcal{F}_\alpha$. □

Proposition 3. *$\mathcal{F}_\alpha$ is C_α-feasible.*

Proof: Suppose that $\omega_1 e_1 e_2 \omega_2 \in \mathcal{F}_\alpha$ with $e_1 \not\prec e_2$. Obviously, $\omega_1 e_1 e_2$ is a feasible prefix. By Lemma 7, $\omega_1 e_2 e_1$ is a feasible prefix. By Lemma 8, $\omega_1 e_2 e_1 \omega_2 \in F_\alpha$. □

Now we are ready to prove the theorem:

Theorem 2. *A permutation of ξ_α that is consistent with the sliced causality $\prec$ is sound.*

Proof: For a permutation of ξ_α, say β_α, which is consistent with $\prec$, we can always find a permutation of $\overline{\xi_\alpha}$, say $\overline{\beta_\alpha}$, which is consistent with $\prec$ and whose projection on ξ_α is β_α. By Proposition 3 and Theorem 1, $\overline{\beta_\alpha}$ is an abstract trace. Therefore, β_α is sound. □

We can therefore analyze the permutations of relevant events consistent with sliced causality to detect potential violations *without* re-executing the program.

5 Evaluation

We devised an efficient vector clock algorithm for computing the sliced causal partial order relation and implemented it as part of jPredictor, a prototype tool for concurrent runtime verification of Java programs, which is publicly available for download [13]. To measure the effectiveness of sliced causality in contrast with more conventional happens-before causalities, we also implemented the procedure in [23] for extracting a happens-before partial order from executions of multi-threaded systems. Interested

readers are referred to [3] for more details about the algoritm and implementation of jPredictor. jPredictor has been evaluated on several concurrent programs. Two measurements are used during the evaluation to compare the sliced causality with the traditional happen-before relation, namely the size of the partial order and the prediction capability to detect data races.

We next discuss some case studies, showing empirically that the use of sliced causality significantly increases the predictive capability of runtime analysis. We show that, on average, the sliced causality relation has about 50% direct inter-thread causal dependencies compared to the more conventional happens-before partial order. Since the number of linearizations of a partial order tends to be exponential with the size of the *complement* of the partial order, any linear reduction in size of the sliced causality compared to traditional happens-before relations is expected to *increase exponentially the coverage* of the corresponding runtime analysis, still avoiding any false alarms. Indeed, the use of sliced causality allowed us to detect concurrency errors that would be very little likely detected using the usual happens-before causality.

5.1 Benchmarks

Table 1 shows the benchmarks that we used, along with their size (LOC abbreviates "lines of code"), number of shared variables (S.V.), and number of threads created during their executions. Banking and Http-Server are two simple examples taken over from [27], showing relatively classical concurrent bug patterns that are discussed in detail in [6]. Elevator, sor, and tsp come from [28]. Elevator simulates the controls of multiple elevators. sor is a scientific computation program that uses barriers instead of locks for synchronization. tsp solves the traveling salesman problem (we run it on the data file tspfile5 coming with the program).

Program	LOC	S. V.	Threads
Banking	150	10	11
Http-Server	170	2	7
Elevator	530	20	4
sor	600	42	4
tsp	1.1k	15	3
Daisy	1.5K	312	3
Raytracer	1.8k	4	4
SystemLogHandler	320	3	3
WebappLoader	3k	10	3

Table 1. Benchmarks used in evaluation

Daisy [19] is a small highly concurrent file system proposed to challenge and evaluate software verification tools. It involves a large number of shared variables because every block of the disk holds a shared variable as a mutex lock. Raytracer is a program from the Java Grande benchmark suite [11]; it implements a multi-threaded ray tracing algorithm. SystemLogHandler and WebappLoader are two components of Tomcat [26], a popular open source Java application server. The version used in our experiments is 5.0.28, the latest of Tomcat 5.0.x.

The test cases used in experiments were manually generated using *fixed* inputs. Each test case was executed multiple (2 to 4) times to generate different execution traces. The detected bugs are all concurrency-related and no false alarms are reported. More bugs could be found if more effective test generation techniques were employed, but that was not our objective here.

5.2 Sliced Causality Increases Coverage Exponentially

As already mentioned, the coverage and therefore the predictive capability of runtime analysis based on causality depends directly upon the number of sound permutations of the causal partial order extracted from the observed execution. Therefore, an immediate measure of effectiveness would be to count all the sound permutations of both the sliced causality and the happens-before partial orders, and then to compare their numbers. Unfortunately, counting sound permutations, also called *linear extensions* or even *topological sortings* in the literature, is a very difficult problem; it is shown in [1] that it is a #P-complete problem, indicating that the counting question may be no easier than the generation question. Note that a fully constrained partial order, that is, a total order, admits only one linearization, while a fully unconstrained partial order, that is, a set, admits an exponential number of linearizations. Extrapolating, even though it should not be taken as an absolute measure in all situations, we can say that the larger the "degree of freedom" of a partial order the larger the number of sound permutations; moreover, we can also say that the number of linearizations is exponential in the number of unordered elements in the partial order.

The simplistic and admittedly informal reasoning above leads us to an important insight: any reduction in the number of causal dependencies may have a significant impact on coverage; in particular, a linear reduction of the number of causal dependencies can lead to an exponential increase in the coverage of the analysis. This suggests that measuring and comparing the "degrees of freedom" of the two partial orders, or complementarily their "degrees of rigidity" (i.e., how many causal dependencies they have), can give us a reasonable estimate of the improvement in coverage. Fortunately, the latter can be computed easily. Since the total orders on the events of each thread are enforced by both sliced causality and happen-before, we only measure the causal dependencies due to direct inter-thread communication. Therefore, the following dependencies are counted, their number declared the *size of the causality*, and then used as a measurement metric: $e_1 \sqsubset e_2$, $\mathit{thread}(e_1) \neq \mathit{thread}(e_2)$, and there is no e_3 such that $e_1 \sqsubset e_3$ and $e_3 \sqsubset e_2$. Table 2 gives the results of the comparison and shows that sliced causality is significantly smaller than the convectional happens-before:

Program	Ave. H.B. Size	Ave. S.C. Size	Ave. Red.
Banking	18	2	81%
Http-Server	22	2	74%
Elevator	240	2	90%
sor	21	8	61%
tsp	5	2	50%
Daisy	41	23	73%
Raytracer	7	3	44%
SystemLogHandler	2	1	50%
WebappLoader	9	5	42%

Table 2. Comparing happen-before relation with sliced causality

In these experiments, the properties to check are race conditions. The second column of Table 2 gives the average size of the happen-before causality for checking race conditions on one shared variable in each program; the third column is the average size of the corresponding sliced causality; and the last column is the average reduction of sliced causality over the unsliced one, for example, it shows that the sliced causality reduces 50% direct inter-thread causal dependencies of happen-before for tsp, and 73% for Daisy. As shown in the table, the sizes of the generated sliced causalities are significantly smaller than those of the corresponding happen-before partial orders; in most cases, the sliced causality is smaller than half the size of the corresponding happens-before. This means that, for each observed execution, more potential executions can be covered in the analysis; this is indeed well reflected by the next results.

5.3 Detecting Dataraces

We evaluated the effectiveness of sliced causality on datarace detection. Dataraces need no formal specification and their detection is highly desirable. The first column in the table is the number of races detected by jPredictor, while those races that are real bugs (the others being benign) are given in the second column. 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 detecting dataraces, 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 some programs, e.g., in tsp and Tomcat, benign or not, without enabling the sliced causality.

Program	Races	Bugs	HB
Banking	1	1	1
Http-Server	2	2	1
Elevator	0	0	0
sor	0	0	0
tsp	1	1	0
Daisy	1	1	0
Raytracer	1	1	1
SystemLogHandler	1	1	0
WebappLoader	3	1	0

Table 3. Race detection results

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 jPredictor shows that the bug still exists in the patched version. More detailed explanation about these bugs can be found in [3].

6 Conclusion

This paper presents a parametric approach to happen-before causal partial orders, which facilitates defining and proving correctness of happen-before relations. Existing vari-

ants of happen-before causalities can be obtained as instances of this parametric framework. A more relaxed happen-before relation, called sliced causality is also defined and proved within our parametric framework. Sliced causality employs static and dynamic analysis to filter out unnecessary dependencies on events in order to improve the coverage of analysis without losing soundness. Evaluation shows that sliced causality can significantly increase the predictive capability of the concurrent runtime verification.

References

1. G. Brightwell and P. Winkler. Counting linear extensions is #p-complete. In *Annual ACM symposium on Theory of computing (STOC)*, 1991.
2. K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
3. 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.
4. F. Chen and G. Roşu. Parametric and termination-sensitive control dependence - extended abstract. In *International Static Analysis Symposium (SAS)*, 2006.
5. R. Cooper and K. Marzullo. Consistent detection of global predicates. In *ACM/ONR workshop on Parallel and distributed debugging (PADD)*, 1991.
6. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.
7. 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.
8. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 2005.
9. T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 1999.
10. S. Horwitz and T. W. Reps. The use of program dependence graphs in software engineering. In *International Conference on Software Engineering (ICSE)*, 1992.
11. Java grande. http://www.javagrande.org/.
12. S. M. Johnson. Generation of permutations by adjacent transposition. *Mathematics of Computation*, 17(83):282–285, 1963.
13. jPredictor. http://fsl.cs.uiuc.edu/jPredictor/.
14. J. Krinke. Static slicing of threaded programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 1998.
15. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of ACM*, 21(7):558–565, 1978.
16. R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 1991.
17. 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.
18. 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.
19. S. Qadeer. research.microsoft.com/qadeer/cav-issta.htm.
20. 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.

21. K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 2003.
22. K. Sen, G. Roşu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 2005.
23. K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. *International Journal on Software Technology and Tools Transfer (STTT)*, 8(3):248–260, 2005. Previous version appeared in TACAS'04.
24. S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient detection of global properties in distributed systems using partial-order methods. In *International Conference on Computer Aided Verification (CAV)*, 2000.
25. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, CWI, 1994.
26. Apache group. Tomcat. http://jakarta.apache.org/tomcat/.
27. S. Ur. Website with examples of multi-threaded programs. http://qp.research.ibm.com/QuickPlace/concurrency_testing/Main.nsf.
28. C. von Praun and T. R. Gross. Object race detection. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.