2013 IEEE 33rd International Conference on Distributed Computing Systems

Towards an Efficient Online Causal-Event-Pattern-Matching Framework

Sukanta Pramanik David Taylor Bernard Wong
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Canada
Email: {spramanik, dtaylor, bernard}@uwaterloo.ca

***Abstract*—Event monitoring and logging, that is, recording the communication events between processes, is a critical component in many highly reliable distributed systems. The event logs enable the identification of certain safety-condition violations, such as race conditions and mutual-exclusion violations, as safety is generally contingent on processes communicating in a specific causally ordered pattern. Previous efforts at finding such patterns have often focused on offline techniques, which are unable to identify operational problems as they occur. Online monitoring tools exist but they are often restricted to identifying a specific violation condition, such as a deadlock or a race condition, using dedicated data structures. We address the more general problem of detecting causally related event patterns that can be used to identify various undesired behaviours in the system.**

The main challenge for online pattern matching is the need to store the partial matches to the pattern, as they may combine with future events to form a complete match. Unlike pattern matching in most other domains, causally ordered patterns can span a potentially unbounded number of events and efficiently searching through this large collection poses a significant challenge.

In this paper, we introduce OCEP, an efficient online causal-event-pattern-matching framework that bounds the number of partial matches it stores by reporting only a representative subset of pattern matches. We define a subset of matches as representative if it has at least *one occurrence of each event in the pattern on each process*, which is applicable for a large class of distributed applications. With this definition, OCEP introduces a backtracking algorithm to efficiently find a representative subset from the history of events. An evaluation of the framework shows that OCEP is capable of handling several frequently occurring violation patterns at the event rates of some representative distributed applications.

***Index Terms*—Causal Ordering, Event-Based System, Distributed System, Distributed System Monitoring.**

I. Introduction

Distributed application development has, in the past few years, seen a significant resurgence due to the popularity of large-scale Web services that rely on distributed application backends. These distributed backends are designed to scale horizontally to utilize the vast quantity of resources available in modern datacenters. Supporting such large-scale deployments, however, introduces additional uncertainty and complexity to these distributed applications, which already have complicated communication patterns to support sophisticated and demanding Web services. These factors make it incredibly difficult to reason about the correctness of a modern distributed application.

One common approach to help with understanding the runtime behaviour of a distributed application is to track its execution over time in order to capture its runtime-state information. The collected information is then used to detect whether a property is satisfied or violated in the global state. This approach of *global-predicate* detection is a well studied problem and is based on building a lattice of global states [12], which is known to be NP-complete [29].

The semantics intended by the programmer for an application is also closely related to a *linearized history* of the system [18]. A *linearized history* is a finite sequence of events that consists of either invocation of operations or responses to operations. Software behaviour can then be analyzed by monitoring for a pattern of events that represent some undesired behaviour, such as bugs, misuse, or intrusions. For example, in a traffic-light system, a correctness condition is that lights in only one direction may be green in the global state. Alternatively, this problem can be modeled as a sequence of events between the lights. An event-matching-based approach monitors the events e_i that denote light i has turned green and then searches for a pattern that represents two events e_i and e_j happening concurrently. A match to this pattern signifies that the system is in an unsafe state. Similar patterns can also be used to identify mutual-exclusion violations in a distributed application even if the actual global state observed by the system is correct [36].

In this paper, we introduce OCEP, an efficient online framework for matching causal event-patterns. We define an event as a state transition in the system, often a result of receiving or sending a message, and a causal event-pattern as a distributed sequence of causally ordered events representing a complex interaction in the system. Matching a causal event-pattern allows us to reason about the correctness of many distributed systems without requiring the global state.

Although the use of causal event-pattern matching is significantly less expensive than tracking the global state, it still requires storing a substantial amount of intermediate information, representing partial pattern matches, to match arbitrary causal relationships. This affects the runtime performance in two different ways. Firstly, when monitoring a set of arbitrarily long-running processes, the amount of memory required to store the partial matches may also become arbitrarily large. Secondly, the monitoring algorithm needs to traverse this large collection of partial matches on each event in order to extend them to a total match. For these reasons, previous

1063-6927/13 $26.00 © 2013 IEEE
DOI 10.1109/ICDCS.2013.66

attempts [31, 41] at causal event-pattern matching are targeted at post-mortem analysis. Although post-mortem analysis is a valuable tool in identifying system problems, it does not help service providers resolve operational problems as they occur. The ability to diagnose problems in a distributed application in realtime can significantly help in reducing the impact of problems on the service providers.

One possible approach to providing online causal event-matching is to maintain a time-based sliding window and discard the partial matches that lie outside the window [3, 15]. While this is a simple and effective solution in some cases, there are applications, such as intrusion and anomaly detection, that require event-patterns that span a long interval.

An alternative approach is to restrict the pattern-matching queries to limit the search space. For example, there are existing tools that focus on Select-Project-Join (SPJ) queries, also known as conjunctive queries with arithmetic comparisons [5], which are sufficient for applications that only require reporting the existence of matches. These queries are not sufficiently expressive to determine the participating processes in the query results. Therefore, for applications, such as intrusion-detection systems, that require such information, an additional post-mortem tool is necessary to determine the participating processes.

In contrast, OCEP does not restrict the matching time interval or event-pattern, but instead provides a *representative* subset of matches for a given causal event-pattern. A representative subset includes each event in the pattern that has occurred anywhere in the system, if such an event exists, and is part of an event-pattern match. By limiting the number of matches, OCEP can perform online event-pattern matching while retaining only a bounded number of partial matches. More importantly, finding a representative subset is sufficient for a large class of applications. We illustrate this using some of the key concurrency-bug patterns in distributed systems such as deadlock, race condition, atomicity violation, and ordering bug. We show through simulations that OCEP can efficiently perform causal event-pattern matching on these sample patterns; OCEP requires less than 1 ms to detect a safety violation in most of the test cases.

The rest of this paper is organized as follows. We begin in Section II by reviewing related work. Section III provides our framework for causality in partial-order event data in distributed systems. Section IV details our causal-pattern-matching algorithm. Section V reports our experimental results and we conclude in Section VI.

II. Related Work

A distributed application can be modeled as a set of unbounded data streams that are constantly generating new events. While a typical database query searches for a set of correlated data that have equivalence between values in one or more columns, in event-based pattern search the correlation is based on precedence relationships between events or sets of events. The conventional wisdom for stream query processing has been to restrict the pattern-matching queries so as to handle only conjunctive queries with arithmetic comparisons [5]. Another approach is to constrain the search space by using a sliding window and report only the matches that fall within it [3]. We address the problem of finding the constituent events that match a pattern limiting only the number of matches that are reported.

Dynamic analysis of program behaviour through instrumentation is also a well studied field. Often it is focused on identifying specific concurrency errors, e.g., to detect deadlocks [2], data races [35], message races [32], or atomicity violations [40]. We address the problem of detecting causally related event patterns that are more generic in nature and can be used to match various undesired behaviours in a system.

There is a large collection of literature that focuses on post-mortem analysis by parsing the logs of instrumented events [7, 31, 34, 41] or passively monitored communication [4]. We see our work as a complementary tool that can be used alongside post-mortem analysis tools like log-based or replay-based analysis. A user may identify a runtime safety violation using our tool and then restrict offline analysis, for in-depth checking, to particular traces that are involved.

There also exist replay-based tools for checking runtime properties by replaying the instances of a distributed system [17, 24]. The benefit of these approaches is their ability to deterministically replay the previous execution and reproduce a bug. Unfortunately, saving and replaying an entire execution of a large system is prohibitively expensive. Also replay-based debuggers are not constrained to react in a timely manner, as an online monitor needs to.

D^3S [25] and P2 [37] allow developers to specify predicates on distributed properties of a deployed system. Both of them use temporal causality which may sometimes indicate a potential ordering relationship when none is present. Our work is built using vector clocks that accurately encode potential causality between events [10].

III. Event-Based Monitoring

We model our distributed system as a finite set of n sequential processes $P_1, P_2, \ldots, P_n$ communicating only by message passing. The processes do not share memory and furthermore there is no global clock or perfectly synchronized local clocks. Each process executes a local algorithm that controls its behaviour by changing its local state and sending messages to the other processes. The occurrences of these actions performed by the local algorithm are called *events*. The coordinated execution of all these local algorithms running concurrently is what we call a *distributed computation*.

An online monitoring system is a set of processes that dynamically collects information about an application and analyzes it in some meaningful way as that application executes. For a proper understanding of the application, it is essential to determine the causal relationship between the events that occur during its execution.

Monitoring a distributed computation often involves checking whether a certain property holds at a certain instant. This requires detecting the *global state* of the system which is

defined as a collection of the local states of all processes at that particular instant. As each process P_i is sequential, the events that occur on a single process are totally ordered. Since there is no shared clock, we can only observe a partial order between events that occur on different processes [22]. Distributed systems thus have an inherent nondeterminism, because a set of concurrent or causally independent events may occur in any order, possibly yielding different results in each case [36]. Detecting a global predicate thus involves maintaining an n-dimensional state lattice [12] or progressively tracking for consistent global checkpoints [6].

In contrast, we focus on the events which represent transition of states. If the correct order in which the events should occur within a system can be specified by a predicate, then detecting the violation of this relative causal order would also signify that the resulting system state is incorrect. Our work focuses on developing an online search facility that, given a causal pattern of events, returns information to the user when an interaction is found that matches the specified pattern.

A. Specifying an Event

An *event* is an activity of interest in the target application. It is the smallest building block and as such is also called a *primitive event*. We specify a class of events in our pattern language as a 3-tuple:

$$class\text{–}id := [process,\ type,\ text]$$

Classes are assigned *id*s which are later used to form *compound events*. The attributes can be specified by providing the *process* on which the event occurs, the *type* of the event, and a *text* field. These attributes can be specified for an exact match, left empty as a wild-card or used as a variable to enforce equality comparison in an operator.

Our algorithm is built on top of an existing tool, POET (Section V-A), which monitors instrumented events from a target system and can send them to a client as a linearization of the partial order. POET stores the events grouped by trace, where a trace is equivalent to any relevant entity with sequential behaviour, such as a process or a thread, but may include passive entities such as an object or a communication channel.

In order to determine the causal relationship between two events we use vector timestamps [14, 28]. Given two events a (on P_i) and b (on P_j) and their timestamps V_a and V_b, we can check whether a happens before b or b happens before a with at most two integer comparisons.

$$a \rightarrow b \iff V_a[i] < V_b[i]$$

If neither $a \rightarrow b$ nor $b \rightarrow a$ holds, two more integer comparisons between process numbers and event numbers are needed to distinguish between equality and concurrency.

B. Specifying a Pattern

Event classes are used with *operator*s and *connector*s to build a *pattern* representing a *compound event*, which is a non-empty set of causally related primitive events. In the rest of the paper we have used uppercase letters for classes of events specified in the pattern and lowercase letters for specific occurrences of matches to an event class. A compound event is represented by uppercase letters in boldface. For example, a compound event $\mathbf{M}$ can be written as a pattern $A \rightarrow B$ and can be used to find pairs a, b where a matches the specification of event class A, b matches the specification of event class B, and $a \rightarrow b$. Figure 1 lists all the operators in our pattern language. In the figure, a and a' match event class A and b matches event class B.

Operator	Meaning
$A \rightarrow B$	Event a happens before event b
$A \| B$	Event a is concurrent with event b
$A.B$	a and b are partner events in a point-to-point communication
$A \xrightarrow{lim} B$	a happens before b and $\nexists a' : a \rightarrow a' \wedge a' \rightarrow b$

Fig. 1: Causality Operators for Patterns.

The causality relationship between compound events is defined by the causal relations between their constituent primitive events. This leads to the definition of *strong* and *weak* precedence of compound events, first defined by Lamport [23]. For any compound events $\mathbf{A}$ and $\mathbf{B}$, strong precedence is defined as,

$$\mathbf{A} \twoheadrightarrow \mathbf{B} \iff \forall a \in \mathbf{A}, \forall b \in \mathbf{B} : a \rightarrow b$$

A causality framework with strong precedence and concurrency leaves out a large number of pairs of compound events which have some primitive events in one preceding some primitive events in the other [8]. This observation inspires another definition of causality among compound events. For any compound events $\mathbf{A}$ and $\mathbf{B}$, weak precedence is defined as,

$$\mathbf{A} \rightarrow \mathbf{B} \iff \exists a \in \mathbf{A}, \exists b \in \mathbf{B} : a \rightarrow b$$

Weak precedence allows the natural definition of concurrency: two compound events are concurrent if and only if they are unrelated by precedence. Unfortunately, weak precedence contradicts the partial-order properties because it is possible that, when it is used, a compound event happens simultaneously before and after another primitive or compound event. Nichols argued that the causality framework needs to be extended to fully classify all possible pairs of compound events [31]. For any compound events $\mathbf{A}$ and $\mathbf{B}$,

$$\mathbf{A}\ \textit{overlaps}\ \mathbf{B} \iff \mathbf{A} \cap \mathbf{B} \neq \phi$$

$$\mathbf{A}\ \textit{is disjoint from}\ \mathbf{B} \iff \mathbf{A} \cap \mathbf{B} = \phi$$

$$\mathbf{A}\ \textit{crosses}\ \mathbf{B} \iff (\exists a_0, a_1 \in \mathbf{A}, \exists b_0, b_1 \in \mathbf{B} : a_0 \rightarrow b_0 \wedge b_1 \rightarrow a_1) \wedge (\mathbf{A}\ \textit{is disjoint from}\ \mathbf{B})$$

These three definitions can be used to define a new operator ($\leftrightarrow$) to recognize *entanglement* of two compound events and to modify the definitions of precedence and concurrence.

$$\mathbf{A} \leftrightarrow \mathbf{B} \iff \mathbf{A} \textit{ crosses } \mathbf{B} \vee \mathbf{A} \textit{ overlaps } \mathbf{B} \quad (1)$$

$$\mathbf{A} \rightarrow \mathbf{B} \iff (\exists a \in \mathbf{A}, \exists b \in \mathbf{B} : a \rightarrow b) \wedge \mathbf{A} \nleftrightarrow \mathbf{B} \quad (2)$$

$$\mathbf{A} \| \mathbf{B} \iff \forall a \in \mathbf{A}, \forall b \in \mathbf{B} : a \| b \quad (3)$$

With the inclusion of *entanglement* ($\leftrightarrow$), given any two event sets, **A** and **B**, their relationships can be described by exactly one of the four relationships: $\mathbf{A} \rightarrow \mathbf{B}$, $\mathbf{B} \rightarrow \mathbf{A}$, $\mathbf{A} \| \mathbf{B}$ or $\mathbf{A} \leftrightarrow \mathbf{B}$. We use the framework proposed by Nichols and in this work precedence and concurrence are defined by (2) and (3) above.

C. Variable Binding

If we build a pattern such as $A \rightarrow B \wedge A \rightarrow C$, the pattern does not specify that the two occurrences of event-class A need to match the same event a. Thus it can happen that the matcher returns two events a_1 and a_2 that separately match the two constraints. We can use *variables* in the pattern to specify that once a matched event is *bound* to a variable, the same matched event must match at all the occurrences of that variable in the pattern.

```
class-A $A;
pattern := $A → B ∧ $A → C;
```

For the above pattern, $\$A$ defines a variable of *class–A* and once bound to a matching event a, it remains the same for the rest of the pattern.

D. Motivating example

Ordering bug refers to the situation where the desired ordering between groups of events is violated. This violation of order is known to be the one common concurrency bug that is not addressed by existing debuggers [27]. One example of this type of bug is bug#962 [1] of ZooKeeper [19]. Zookeeper is a coordination service for distributed processes that achieves high availability through replication. It uses an active-replication technique where a follower sends synchronization requests to the leader for a snapshot of the system. When a restarting follower sent a synch request to the leader, the leader was not blocked from making an update after it took a snapshot of the system. Thus a restarting follower could occasionally receive inconsistent service-data from the leader.

A pattern definition consists of class definitions, declaration of variables (if any), and then the pattern itself.

```
Synch := [$1, Synch_Leader, $2];
Snapshot := [$2, Take_Snapshot, ''];
Update := [$2, Make_Update, ''];
Forward := [$2, Take_Snapshot, $1];
Snapshot $Diff;
Update $Write;
pattern := (Synch → $Diff) ∧  ($Diff → $Write)
    ∧ ($Write → Forward);
```

The pattern above tries to detect a situation when a snapshot taken on a *synch* request is followed by an *update* before it gets forwarded to the follower. The variable-binding for the events ensures the proper causal order. It also shows how variables can be used inside the class definition for better precision. The text field can be used for various purpose and this particular pattern is using it to encode the corresponding trace for a particular Synch/Forward pair.

IV. Online Event-Pattern Monitoring

We use a tree-based mechanism because a tree closely matches the causality structure specified in a pattern.

A. Pattern Tree

The specified pattern is first parsed to create a *pattern tree* as shown in Figure 2. The leaf nodes represent the primitive events in the pattern and the internal nodes represent the compound-event expressions.

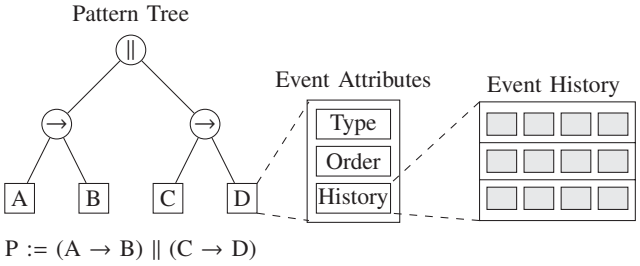


Fig. 2: The Structure of the Pattern Tree

Each leaf node has three main attributes:

- *Type* specifies the event class for the primitive event.
- *Order* defines the order of evaluation.
- *History* is the list of matched primitive events grouped by traces.

Every time POET reports an event that matches a leaf node of the pattern tree, it is added to the corresponding leaf node's *history* of events. This history is grouped by traces and is totally ordered for each individual trace. The OCEP algorithm is then triggered, which tries to build a complete match at the root in a bottom-up fashion. It uses backtracking and the iterative stages for each leaf node are stored in *order*. Thus the runtime of the matching algorithm is only affected by the events that are actually in the pattern, not by all the events that are being monitored.

B. Specifying a Representative Subset

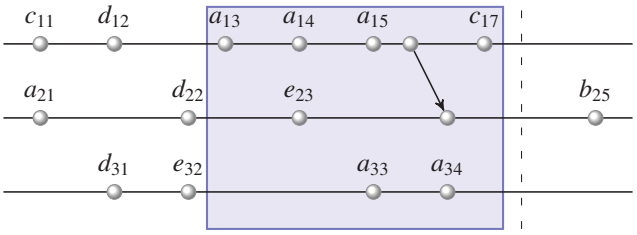
Event-pattern search executes on a set of potentially unbounded processes and thus the amount of memory required to report all possible matches may also grow without bound. We also have to report these matches in a timely manner to provide an effective online monitor.

In Section II we discussed existing approaches to tackle this by limiting the pattern's expressiveness or its event-domain. A restricted pattern with aggregate operators (e.g. average, count, maximum, etc.) can provide valuable information but it only provides a summarized view of the system. If a system requirement can be specified as a pattern of events, finding the specific set of events that violates this pattern and where they occur provides further insight into the behaviour of the system. On the other hand, the sliding-window-based approaches that

restrict the event-domain are susceptible to omission problems as when no matches are found it can also be because a match spans multiple windows.

Our approach is to report a representative subset of all matches that spans the entire execution time. A good representative subset should provide the most information about the matches to the pattern. Each event in a pattern is an instrumented activity of interest occurring on a process. So it is plausible to assume that it is important for the user to know if such an event has occurred anywhere in the system.

A representative subset should also be low in redundancy, so we can exclude multiple occurrences of similar events on the same process. Considering these two objectives, we propose



All: $a_{13}b_{25}$, $a_{14}b_{25}$, $a_{15}b_{25}$, $a_{21}b_{25}$
Window: $a_{13}b_{25}$, $a_{14}b_{25}$, $a_{15}b_{25}$
Desired: $a_{15}b_{25}$, $a_{21}b_{25}$ (not unique)

Fig. 3: Choosing a Representative Subset for $A \rightarrow B$

a subset that has *at least one occurrence per process for each primitive event class*. Figure 3 shows a simple process-time diagram where the dotted vertical line is the *current cut*. On arrival of the new event b, there are four matches for the pattern $A \rightarrow B$.

Figure 3 also shows a sliding window and the set of matches that it will report. We chose the window to have n^2 events where n is the number of processes. If we look at the reported matches, it fails to return a match that involves an event a on P_2 ($a_{21}b_{25}$). Thus the returned subset is not a proper representative of the set of all matches. Considering the pattern as a safety condition, the action that we take based on the returned subset will be incomplete as it misses the matches that span beyond the maintained window.

Our representative subset will report if any of the constituent events in the pattern has occurred on any of the processes and is part of a complete match. Thus if there is only one match, it will definitely be reported. If there are many matches, it can be proved that there exists a subset according to our definition that has cardinality of at most kn. Here k is the number of events in the pattern and n is the number of processes.

C. OCEP Algorithm

OCEP uses *backtracking* to find the subset of matches to the pattern, as shown in Algorithm 1. We start with the newly matched event as our initial assignment (e_1). At every stage of backtracking, the algorithm tries to extend the partial match by finding a match to the event e_i (*goForward*) so that it is causally related to the existing assignment $(e_1, e_2, \ldots, e_{i-1})$ as specified by the pattern. If a complete match is found with an event e_i on trace t or there is no unexplored match on it, the algorithm continues with matches for e_i on trace $t+1$. If there is no unexplored match on any of the traces the algorithm backtracks (*goBackward*). The variable *consistent* controls the direction of backtracking by keeping track of the return value from the two traversal functions.

Algorithm 1 OCEP Algorithm

Precondition: M is a partial match of length one: $\{e_1\}$

```
1: level ← 2          ▷ level is updated inside called functions
2: consistent ← true
3: while level ! = 1 do
4:     if consistent then
5:         consistent ← goForward(M, level)
6:     else
7:         consistent ← goBackward(M, level)
8:     if M is a complete match then
9:         updateSubset(M)
```

A very basic implementation of *goForward* can use chronological backtracking, which will start with the latest match on a trace and chronologically go back in time. That is not very efficient in practice as it explores the entire search space until a solution is found or a conflict is reached.

When dealing with a causal pattern, given a partial match $(e_1, e_2, \ldots, e_{i-1})$, it is possible to use the causality relationship of the instantiated events to restrict the domain of event e_i on any trace. The *greatest predecessor* (GP) of an event a on a trace t is the most-recent event e on that trace that happens before a. An interesting property of $GP(a,t)$ is that it shows in which parts of the process t an event b can occur so as to happen before a. The *least successor* (LS) of an event a on a trace t is the least-recent event e on that trace that happens after a. $LS(a,t)$ shows in which portion of the process t an event b can occur so as to happen after a. When used together, $GP(a,t)$ and $LS(a,t)$ can also identify the portions with which a is concurrent.

Suppose we are trying to instantiate the event e_i on trace l that is concurrent with event e on trace m, which is already instantiated. The earliest event on trace l that can be concurrent with e is the event that happens immediately after its GP on trace l. The latest concurrent event is the event that occurs immediately before its LS on trace l. Thus the domain of e_i on l that can extend the partial match with respect to e is given by $(GP(e,l), LS(e,l))$. The open interval denotes that the GP and LS themselves are not included in the domain. We summarize how we restrict the domain for all relations in Figure 4.

Causality	Domain
$e \parallel e_i$	$(GP(e,l), LS(e,l))$
$e \rightarrow e_i$	$[LS(e,l), \infty)$
$e_i \rightarrow e$	$(-\infty, GP(e,l)]$

Fig. 4: Restricting Domain of e_i with Respect to e

This technique is similar to the *forward checking* algorithm, which uses the instantiated variables to restrict the domains of all future variables [33]. In contrast, we are restricting only the domain of the variable that is currently being instantiated.

Algorithm 2 Forward Algorithm

Precondition: M is a partial match of length $i-1$, i is the backtracking level, and n is the total number of traces.

```
1:  function goForward(M, i)
2:      if D_i is not initialized then
3:          while trace_i < n do
4:              for prev ← 1 ... i − 1 do
5:                  D_i ← restrictDomain(M_prev, trace_i)
6:                  if D_i == ϵ then
7:                      bt[prev][i] ← getTS(M_prev, trace_i)
8:                      break
9:              trace_i ← trace_i + 1
10:     if D_i ! = ϵ then
11:         M_i ← nextMatch(D_i, trace_i)
12:         i ← i + 1
13:         return true
14:     else
15:         return false
```

The pseudo-code for the function *goForward*, which instantiates an event in each backtracking level, is given in Algorithm 2. The variable i is the passed backtracking level. The variables $trace_i$ and D_i store the trace that is currently being traversed and the domain of matched events on it. The function starts by initializing the domain D_i by restricting it using the already instantiated events (a summary of *restrictDomain* is given in Figure 4). If a conflict is found, i.e., a previously instantiated event constrains D_i to empty, *getTS* determines the desired timestamp for the event at the previous level that can possibly resolve this conflict.

After all the traces are tried, it returns to *goBackward*, which uses the recorded timestamps in *bt* to *backtrack* to the closest event that can resolve one of the conflicts. A simple backtrack to the previous event would have caused repeated failure from the same conflicting event.

Suppose e_i has an empty domain on trace m with respect to the instantiated event e_2 on trace l. The vector timestamps of the conflicting events can then be used for updating the domain at the earlier level during backtracking. This is done in the function *getTS* and we illustrate its operation in Figure 5 for each of the causal relations. For simplicity we have shown a truncated process-time diagram and fragmented vector timestamps that only show entries for the relevant traces.

If we are looking for $e_2 \rightarrow e_i$, then the conflict happens because LS(e_2, m), if it exists, happens after the latest e_i on m, Figure 5(a). The l-th entry in the vector timestamp of e_i (t_i^l) indicates $GP(e_i, l)$. Any matched e_2 that happens after the $GP(e_i, l)$ will again lead us to the same conflict. So t_i^l is stored for later use and during backtracking, *updateDomain* uses this to restrict e_2's domain to $(-\infty, GP(e_i, l)]$.

If the conflict is for $e_i \rightarrow e_2$, GP(e_2, m) happens before the earliest e_i on m, Figure 5(b). Thus we can prune all the matches for e_2 on l as none of them will be able to create a complete match.

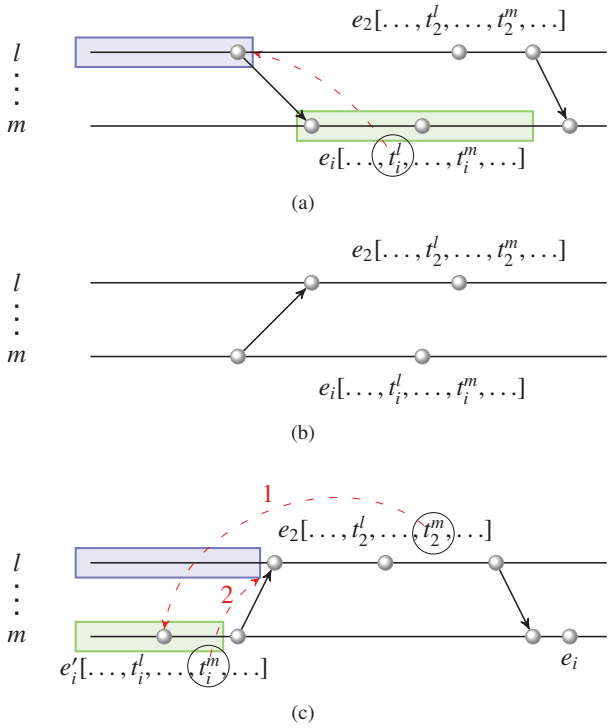


Fig. 5: Using Causality to Update Domain when Backtracking: a) Precedence, b) Follow, c) Concurrency

When we have a conflict for $e_2 \| e_i$, Figure 5(c), all the matches for e_i on m either happen after e_2 or happen before it. So e_2 cannot create a complete match with an e_i on trace m but an earlier e_2' on trace l may still create one. Then the domain of e_i on m that can be of possible interest is $(-\infty, GP(e_2, m))$. We can use t_2^m to find GP(e_2, m), say e_i'. The backtracked level then needs to restrict e_2's domain to $(-\infty, LS(e_i', l))$ and we can use t_i^m to find LS(e_i', l).

Algorithm 3 Backward Algorithm

Precondition: M is a partial match of length $i-1$ and i is the backtracking level

```
1:  function goBackward(M, i)
2:      jumpTS ← getClosest(i, bt[i])
3:      if JumpTS exists then
4:          D_i ← updateDomain(i, jumpTS)
5:      else
6:          i ← i − 1
7:      return true
```

The pseudo-code for the function *goBackward* is given in Algorithm 3. It first tries to determine if any conflict with the uninstantiated events is recorded for the instantiated event at this level. If any exists, it finds the one that will make it backtrack to the latest match in the current domain. It extracts the required timestamp and uses it to restrict its domain. If no conflict was found, it simply backtracks to the previous level.

V. Performance Evaluation

A. Partial-Order Event Tracer

This work is built on top of the various techniques and algorithms in an existing tool, the Partial-Order Event Tracer (POET) [21]. POET itself is a distributed application that allows a user to instrument a distributed application and collect information about it. It is target-system independent as it can collect data from a wide variety of execution (target) environments with a minimum amount of effort [39].

The core information stored by POET is a set of events grouped by traces and the partial-order relationships among those events. The occurrence of a POET event indicates that one of a predefined set of important actions has occurred in the monitored application. This set of important actions is entirely dependent on the target environment being used.

A client can connect to the POET server in a way that it receives the arriving events in a linearization of the partial order of the events. A linearization of a partial order $\rightarrow$ on a set X is a sequence containing each element of X once such that any x occurs before x' whenever $x \rightarrow x'$. Our monitor application connects to POET as one such client and tries to detect the specified pattern as the events appear.

B. Evaluation Methodology

We evaluate the effectiveness of OCEP using some representative bug patterns that we have chosen from the existing literature [13, 27]. These test patterns were run on event data collected from μC++ and MPI environments. μC++ [11] extends C++ with additional constructs to incorporate concurrent programming. MPI [38] is a widely used library for writing parallel applications.

Each test case is executed until the number of events generated exceeds one million. We used the *dump* feature in POET to save the collected trace-event data in a file. The *reload* feature in POET allows us to reuse this file with the saved events passed to POET via the same interface used to collect events from a running application. OCEP connects to POET as a client in order to receive the events in a linearization of the partial order.

All measurements are performed on a workstation with an Intel Core 2 Duo 2GHz CPU and 2GB memory running Linux kernel 3.0.0. OCEP is executed with each set of trace-event data five times and the average is used for the evaluation.

The main performance metric that we used for our evaluation is the *execution time*, which is measured as the wall clock time taken by the monitor to find the set of matches on arrival of an event. The events can be divided into three categories depending on a given pattern: i) events that do not match the pattern, ii) events that match the pattern but will not generate a complete match, and iii) events that can possibly generate a complete match, which we call *terminating event*s. For the pattern $A \rightarrow B$, only a newly arrived b is a terminating event as it can generate a complete match if a causally related a is already found. In contrast, for the pattern $A||B$, any newly arrived a or b is a terminating event.

C. Case Studies

We use boxplots to show the measured execution time taken for each of our case studies in Figures 6-9. The centre rectangle spans the *inter quartile range* (IQR), which is the likely range of variation, with the inner segment representing the median. The whisker marks are placed $1.5 \times IQR$ above the third quartile and below the first quartile, while the crosses mark the outliers.

1) Deadlock: In MPI, when a process executes a blocking point-to-point communication, it does not return until it is complete and the buffer can be reused. So an application can deadlock if there exists a send-receive cycle due to incorrect usage of the communication routine. A commonly used method for detecting such a deadlock is to build a dependency graph and check for cycles [2]. Event patterns are not able to detect a generic cycle as opposed to a dependency graph, but they can be used to identify a deadlock of specific length.

We simulate deadlock using a parallel algorithm for *random walk* which has many applications in statistical and scientific computation. It divides a domain among the parallel processes and each process has a number of walkers traversing a contiguous sub-domain. The processes communicate among themselves to exchange the walkers that move across process boundaries. We deliberately leave a deadlock in the code for this point-to-point communication. Interestingly enough, this deadlock is rarely visible as *MPI_Send*, although a blocking operation, only gets blocked when the network cannot buffer the message completely [38].

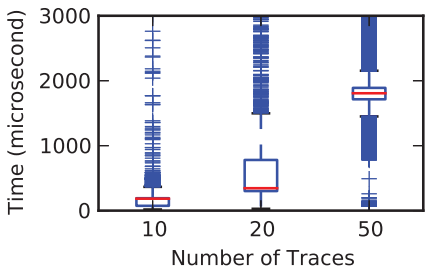


Fig. 6: Execution Time for Deadlock

Figure 6 shows that our algorithm is still exponential in terms of the length of the pattern, which is expected since we are using a backtracking algorithm. The efficiency of our backtracking algorithm depends on its ability to prune the domain for an event on a trace based on its causality relation with the other events. As we explained in Figure 5, we used vector timestamps of the causally related events to constrain the searched domain. In contrast, building and maintaining a dependency graph is costly, which is apparent from the runtime of 35 seconds to detect a cycle of length 30 [2].

2) Message Race: When two or more concurrent messages are sent to the same process they may arrive in a random order, causing nondeterministic execution of a parallel program. This may lead to sporadically occurring errors that are difficult to reproduce. Even when a message race happens by design, it

is critical to detect it for debugging in order to ensure that all possible executions of a program are examined [20].

A common method for detecting message races is to keep track of the *receive* events on a trace and compare their vector timestamps for causality [30]. If any two incoming messages to a process are concurrent then the two messages race. A causal-event-pattern can express this pattern, which we use to express message races.

We use a benchmark program in which all processes but one concurrently send message to the remaining process while the latter accepts them using a blocking receive with *MPI_ANY_SOURCE* wild-card.

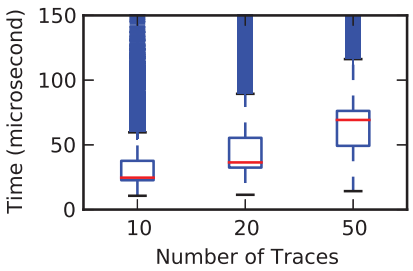


Fig. 7: Execution Time for Message Races

Existing methods for detecting message races pass the vector timestamp among the processes by attaching them to messages [32]. OCEP receives a vector timestamp constructed in POET, not in the application, so there is minimal extra overhead on the application itself.

3) Atomicity Violation: An atomic code segment is protected in the sense that when a process executes the first event in the segment, no other process can start executing the segment before the first process executes the last event. The approaches for detecting an atomicity violation rely on finding unserializable patterns of operations by searching the events that are related to shared-variable access and synchronization primitives [40]. Our approach is similar in the sense that we search for a causal pattern among these same events, but we also monitor the synchronization primitives as separate traces, which allows us to represent an atomicity violation as a causal pattern.

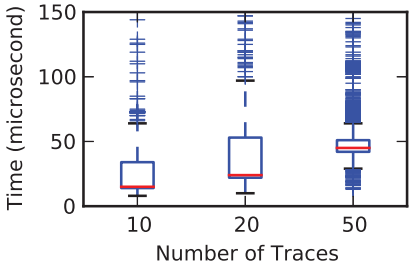


Fig. 8: Execution Time for Atomicity Violation

We demonstrate this with a μC++ program that has a method protected by a semaphore so that there is never more than one thread executing it. There is an intentional bug for which, when a thread attempts to execute the method, the semaphore will not be acquired properly with 1% probability. Existing approaches often build a conflict graph with the monitored events and synchronization primitives, which has previously taken 0.4-40 seconds for detecting similar violation [40].

We chose the μC++ environment as a POET plugin for μC++ already adds semaphores as separate traces. It is possible to add different semaphore implementation, such as a *pthreads* semaphore, which will require additional plugins.

4) Undesired Order: Studies have found that most non-deadlock concurrency bugs in real-world applications belong to one of two categories: atomicity-violation and order-violation [27]. Existing concurrency-bug detection tools do not address bugs that are manifested by the violation of order implied by the developer.

There are various ways to analyze the expected behaviour of a system. A developer can represent the violation of intended ordering with a causally-related pattern of events. Large software systems often use software classification tools to identify repetitive patterns of events from program execution traces [26]. Over the last decade, *design patterns* have become one of the most powerful tools in designing large software systems [16]. *Design patterns* are reusable software modules for recurring problems and both formal [9] and informal [16] specifications exist defining precise semantics.

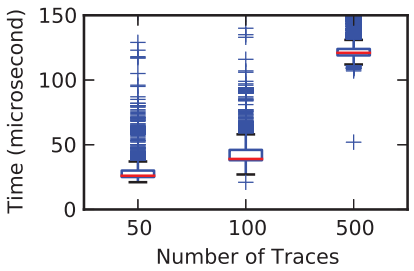


Fig. 9: Execution Time for Ordering Bug

We use the ordering-bug pattern shown in Section III-D that has a leader-and-follower coherence issue. We simulate a distributed application that handles a replicated service. We left a deliberate bug in the synchronization procedure so that there is a 1% chance that a leader may make an update after it takes a snapshot of the system and then it will forward this stale snapshot to the synchronizing follower.

D. Results and Discussion

The strength of our event monitor lies in its ability to detect the occurrences of a pattern that will include each trace that has a constituent event matching to it. This makes it an excellent tool for identifying safety conditions in a system. The outliers in the boxplots for test cases show that in order to identify such a subset we occasionally have to traverse a large section of the process-time diagram. We limited the number of outliers shown in Figures 6-9 so that the IQR and whisker marks are clearly shown. We summarize the quartiles, top whisker, and the maximum time taken by an event in the Figure 10.

Test Case	Q1	Med	Q3	Top Whisker	Max
Deadlock	1712	1805	1888	2153	14931
Races	49	69	76	117	10830
Atomicity	42	45	51	65	6819
Ordering	119	121	124	132	7668

Fig. 10: Detailed Runtime for Test Cases (μsecond)

The second performance metric that we used is completeness. Since we used known violation cases in simulated applications, we also knew all such occurrences. Our OCEP algorithm is complete as it correctly reported all violations for the test cases. OCEP also did not report any false positives for any of the test cases.

The major question that we wanted to answer in our simulation was whether our algorithm is efficient enough to provide fast detection of violation patterns with low overhead in realistic application settings.

We demonstrated this by choosing some of the prevalent concurrency bug patterns in real-world applications. We find that OCEP is able to find a match to most of the tested violation patterns in less than one millisecond. The only test case where the whisker marks are above this limit is when we are searching for a long deadlock cycle. We could not evaluate the existing graph-based approaches as their implementations are not publicly available. As shown in Section V-C, OCEP runs orders of magnitude faster to detect similar violation compared to previously published results. Additionally, we support generic patterns which are able to detect various undesired behaviours as opposed to detecting a specific violation.

Survey results often show that most concurrency bugs involve only a small number of processes [27]. We use variable binding for both an event and its attributes inside the class definition to clearly specify the relationship among the constituent events in the pattern. For example, a complete match of the ordering bug in Section III-D only involves the leader and the follower. Figure 9 shows almost a linear increase in runtime with the number of traces. This signifies that our algorithm was effectively able to isolate the relevant traces from the pattern specification. So OCEP can provide fast detection for a range of violation patterns when the pattern involves a moderate number of traces.

That leaves us the question of the overhead. In this work we used a very simple approach of discarding multiple occurrences of the same event on a trace which have no send or receive events between them. As we are dealing with potential causality, how an event is causally related to the other events is only affected by the messages. So two events which do not have any send or receive events between them have the same causal relation with events on other traces. This approach is computationally simpler, O(1), but does not guarantee a minimal subset size. In the worst case it will store all the events since the start-up. So OCEP is not scalable to an arbitrarily large number of long-running processes. The main challenge in maintaining a minimal overhead is that there are cases in which it is always possible to extend a partial match with future events. Comparison of timestamps can help but each such operation is O(n) and hence prohibitively expensive. We think a technique that exploits the causality present in the pattern can work better in this situation rather than a general solution. We leave this for future work.

Finally, the detection of violation with OCEP requires knowledge of the causality structure among the monitored events that represents an undesired behaviour. An unexpected behaviour with a different causality structure may remain undetected although it has the same effect on the global property.

VI. Conclusion

Online detection of causal-event-patterns is a complex problem, particularly for long-running applications. In this paper, we introduced OCEP, an efficient online causal-event-pattern-matching framework that can be used to identify safety-condition violations in distributed applications. On arrival of each event, OCEP returns a representative subset of matches that spans the entire execution time. It uses vector timestamps and the causality relation among the events in the pattern to efficiently prune the search space.

We evaluated our approach with some of the most frequently found concurrency bug patterns. OCEP successfully finds all occurrences of the pattern in each application within a millisecond in almost all cases. This is the first available online tool that detects safety violations using generic causally related sets of events that represents event-patterns.

Our work can be extended by exploring the possibility of pruning events from the event history that can no longer generate new matches for the subset. A future plugin for POET, which will allow a client to retrieve the timestamp values of any previously matched event in constant time, can also be used to restrict the size of the event history maintained at the monitor. Also, at each backtracking level, the traces are traversed sequentially. Each of these traces represents a subtree in the total search space. This parallelism can be exploited to improve the performance of the algorithm.

References

[1] "ZooKeeper bug# 962," https://issues.apache.org/jira/browse/ZOOKEEPER-962, accessed: 10/10/2012.

[2] R. Agarwal, L. Wang, and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," in *HVC*, Haifa, Israel, Nov 2005, pp. 191–207.

[3] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *SIGMOD*, Vancouver, BC, Jun 2008, pp. 147–160.

[4] M. K. Aguilera, J. C. Mogul *et al.*, "Performance debugging for distributed systems of black boxes," in *SOSP*, Bolton Landing, NY, Oct 2003, pp. 74–89.

[5] A. Arasu, B. Babcock *et al.*, "Characterizing memory requirements for queries over continuous data streams," *ACM T. Database Syst.*, vol. 29, no. 1, pp. 162–194, 2004.

[6] R. Baldoni, J.-M. Hélary, and M. Raynal, "Rollback-dependency trackability: A minimal characterization and its protocol," *Inf. Comput.*, vol. 165, no. 2, pp. 144–173, 2001.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling," in *OSDI*, Seattle, WA, Dec 2004, pp. 259–272.

[8] T. Basten, T. Kunz *et al.*, "Vector time and causality among abstract events in distributed computations," *Distrib. Comput.*, vol. 11, pp. 21–39, 1997.

[9] I. Bayley and H. Zhu, "Formal specification of the variants and behavioural features of design patterns," *J Syst. Software*, vol. 83, no. 2, pp. 209–221, 2010.

[10] K. P. Birman, "Overcoming failures in a distributed system," in *Reliable Distributed Systems*. Springer New York, 2005, pp. 247–275.

[11] P. A. Buhr, G. Ditchfield *et al.*, "μC++: Concurrency in the object-oriented language C++," *Softw. Pract. Exper.*, vol. 22, no. 2, pp. 137–172, 1992.

[12] R. Cooper and K. Marzullo, "Consistent detection of global predicates," in *PADD*, May 1991, pp. 167–174.

[13] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *IPDPS*, Apr 2003.

[14] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 28–33, Aug 1991.

[15] M. Fox, "Event-predicate detection in the monitoring of distributed applications," Master's thesis, University of Waterloo, Waterloo, Ontario, 1998.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison Wesley, 1995.

[17] D. Geels, G. Altekar *et al.*, "Friday: Global comprehension for distributed replay," in *NSDI*, Cambridge, MA, Apr 2007, pp. 285–298.

[18] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM T Progr. Lang. Sys.*, vol. 12, pp. 463–492, 1990.

[19] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for Internet-scale systems," in *USENIX ATC*, Boston, MA, Jun 2010.

[20] D. Kranzlmüller and M. Schulz, "Notes on nondeterminism in message passing programs," in *9th European PVM/MPI Users' Group Meeting*, Linz, Austria, Sep/Oct 2002, pp. 357–367.

[21] T. Kunz, J. P. Black, D. Taylor, and T. Basten, "POET: Target-system independent visualizations of complex distributed application executions," *Comput. J.*, vol. 40, no. 8, pp. 499–512, 1997.

[22] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul 1978.

[23] L. Lamport, "On interprocess communication, part I: Basic formalism, part II: Algorithms," *Distrib. Comput.*, vol. 1, pp. 77–101, 1986.

[24] K. H. Lee, N. Sumner, X. Zhang, and P. Eugster, "Unified debugging of distributed systems with Recon," in *ICDSN*, Hong Kong, China, Jun 2011, pp. 85–96.

[25] X. Liu, Z. Guo *et al.*, "D^{3}S: Debugging deployed distributed systems," in *NSDI*, San Francisco, CA, Apr 2008, pp. 423–437.

[26] D. Lo, H. Cheng *et al.*, "Classification of software behaviors for failure detection: A discriminative pattern mining approach," in *SIGKDD*, Paris, France, Jun/Jul 2009, pp. 557–566.

[27] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *ASPLOS XIII*, Seattle, WA, Mar 2008, pp. 329–339.

[28] F. Mattern, "Virtual time and global states of distributed systems," in *I W Parall. Distrib. Alg.*, North-Holland / Elsevier, Dec 1988, pp. 215–226.

[29] N. Mittal and V. K. Garg, "On detecting global predicates in distributed computations," in *ICDCS*, Phoenix, AZ, Apr 2001.

[30] R. H. B. Netzer and B. P. Miller, "Optimal tracing and replay for debugging message-passing parallel programs," in *Supercomputing*, Minneapolis, MN, Nov 1992, pp. 502–511.

[31] M. Nichols, "Efficient pattern search in large, partial-order data sets," Ph.D. dissertation, University of Waterloo, Waterloo, Ontario, 2008.

[32] M.-Y. Park, S. J. Shim, Y.-K. Jun, and H.-R. Park, "MPIRace-Check: Detection of message races in MPI programs," in *GPC*, May 2007, pp. 322–333.

[33] P. Prosser, "Hybrid algorithms for the constraint satisfaction problem," *Computat. Intell.*, vol. 9, no. 3, pp. 268–299, 1993.

[34] P. Reynolds, C. Killian *et al.*, "Pip: Detecting the unexpected in distributed systems," in *NSDI*, San Jose, CA, May 2006, pp. 115–128.

[35] S. Savage, M. Burrows *et al.*, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov 1997.

[36] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distrib. Comput.*, vol. 7, no. 3, pp. 149–174, Mar 1994.

[37] A. Singh, P. M. T. Roscoe, and P. Druschel, "Using queries for distributed monitoring and forensics," in *EuroSys*, Leuven, Belgium, Apr 2006, pp. 389–402.

[38] M. Snir, J. Dongarra *et al.*, *MPI: The Complete Reference*, 2nd ed. The MIT Press, 1998.

[39] D. Taylor, T. Kunz, and J. P. Black, "A tool for debugging OSF DCE applications," in *COMPSAC*, Seoul, Korea, Aug 1996, pp. 440–446.

[40] L. Wang and S. D. Stoller, "Accurate and efficient runtime detection of atomicity errors in concurrent programs," in *PPoPP*, New York, NY, Mar 2006, pp. 137–146.

[41] P. Xie, "Convex-event based offline event-predicate detection," Master's thesis, University of Waterloo, Waterloo, Ontario, 2003.