# Efficient Computation of Happens-Before Relation for Event-Driven Programs

Pallavi Maiya
Indian Institute of Science, India
pallavi.maiya@csa.iisc.ernet.in

Aditya Kanade
Indian Institute of Science, India
kanade@csa.iisc.ernet.in

## ABSTRACT

An emerging style of programming is to use both threads and events to achieve better scalability. The improved scalability comes at the price of increased complexity, as both threads and events can follow non-deterministic schedules. The happens-before (HB) relation captures the space of possible schedules and forms the basis of various concurrency analyses. Improving efficiency of the HB computation can speed up these analyses.

In this paper, we identify a major bottleneck in computation of the HB relation for such event-driven programs. Event-driven programs are designed to interact continuously with their environment, and usually receive a large number of events even within a short span of time. This increases the cost of discovering the HB order among the events. We propose a novel data structure, called *event graph*, that maintains a subset of the HB relation to efficiently infer order between any pair of events. We present an algorithm, called EventTrack, which improves efficiency of vector clock based HB computation for event-driven programs using event graphs.

We have implemented EventTrack and evaluated it on traces of eight Android applications. Compared to the state-of-the-art technique, EventTrack gave an average speedup of 4.9X. The speedup ranged from 1.8X to 10.3X across the applications.

## CCS CONCEPTS

•**Software and its engineering** → **Dynamic analysis;** *Publish-subscribe / event-based architectures; Concurrent programming structures;* •**Theory of computation** → Design and analysis of algorithms;

## KEYWORDS

Concurrency Analysis, Happens-before Reasoning, Vector Clock, Event-driven Programs, Android

**ACM Reference format:**
Pallavi Maiya and Aditya Kanade. 2017. Efficient Computation of Happens-Before Relation for Event-Driven Programs. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
DOI: 10.1145/3092703.3092733


*ISSTA'17, Santa Barbara, CA, USA*

DOI: 10.1145/3092703.3092733

## 1 INTRODUCTION

In recent years, event-driven programming has become prominent. Programs make use of both threads and events and thereby, accrue scalability benefits of both. The improved scalability comes at the price of increased complexity, as both threads and events can follow non-deterministic schedules. Android applications [19] are prototypical of this programming model.

Analyzing the partial order among operations of a program is the key to understand the space of possible schedules. This partial order is formally captured using the *happens-before relation* [12]. Different linearizations of the happens-before (HB) relation give rise to different schedules. The HB relation therefore forms the basis of various concurrency analyses for event-driven programs such as race detection [2, 8, 9, 17, 21, 23] and model checking [11, 13]. Improving the efficiency of the HB computation can help speed up these analyses.

In this paper, we investigate efficient computation of the HB relation for execution traces of event-driven programs. In specific, we focus on Android applications whose HB relation is well studied in literature [2, 8, 17]. Similar to multi-threaded programs, operations of different threads of an event-driven program may be partially ordered with each other. In addition, since events may get reordered across executions, operations of different event handlers executing on the same thread may also be partially ordered instead of being totally ordered. We identify a major bottleneck in computation of the HB relation for event-driven programs, that of discovering the HB order among the event handlers on the same thread. The order among event handlers is subject to a number of HB rules [2, 8, 17]. These depend on the order between the operations that post the events, or the operations that mark start and end of the event handlers. A simple solution to evaluate these HB rules is to check HB ordering between relevant operations of *every pair of events* posted to the same thread, in a brute force manner. With thousands of events to analyze per execution, this becomes fairly expensive.

Let us consider the execution trace of an event-driven program shown in Figure 1. The operations executed by a thread are vertically aligned and appear in the order of execution. An operation `post(`$e$`)` means that the event $e$ is enqueued to a target thread. A thread dequeues events from its queue and runs their handlers one after the other. We use unique names for all the events in the examples in this paper so that the thread to which an event is posted is evident. The operations `deq(`$e$`)` and `end(`$e$`)` show when the handler of $e$ starts and finishes its execution. Alternatively, we may say $e$ is dequeued and ends. We do not show any memory operations in the traces because they are not relevant to the HB computation.

We use vector clocks [18] to represent the HB relation of event-driven programs. A vector clock is a vector of logical clocks where each logical clock corresponds to a thread or an event. The order

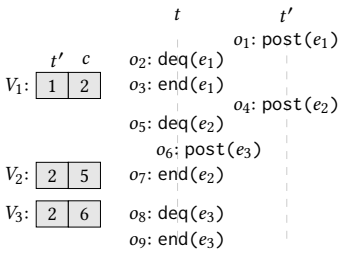


**Figure 1: An execution trace of an event-driven program.**

between two operations in a trace is inferred by comparing their vector clock timestamps. Event-driven programs are designed to interact continuously with their environment, and usually receive a large number of events even within a short span of time. Assigning one logical clock per event in a vector clock therefore becomes space inefficient. This problem is addressed in [23] by applying chain decomposition [10], wherein all events corresponding to a chain of totally ordered event handlers share the same logical clock.

Let $V_1$, $V_2$ and $V_3$ be the vector clocks of the events $e_1$, $e_2$ and $e_3$ in Figure 1. The value of each of these vector clocks reflects the timestamp of the most recent operation processed by a HB computation algorithm, in the corresponding event handler. In Figure 1, $V_1$ and $V_2$ contain timestamps of operation $o_3$ in the handler of $e_1$ and $o_7$ in the handler of $e_2$ respectively. $V_3$ is the vector clock of $e_3$ after the operation $o_8$ finishes. Each of these vector clocks have two logical clocks tagged as $t'$ and $c$. The first corresponds to the thread $t'$. The second corresponds to the chain $c$ of events $e_1$-$e_2$-$e_3$. Each logical clock has an integer value which is incremented after processing an operation corresponding to its thread or chain. We discuss chain decomposition later in Section 3.1. In this example, as the algorithm discovers that $e_2$ and $e_3$ are totally ordered with $e_1$, it assigns them to the same chain $c$ used for $e_1$.

For the event-driven programs we consider, events are dequeued in a FIFO order and handlers are run to completion. In Figure 1, the handler of $e_2$ posts the event $e_3$ and therefore, $e_2$ happens before $e_3$. In terms of operations, this means that `end(`$e_2$`)` happens before `deq(`$e_3$`)`. As the post of $e_1$ happens before the post of $e_2$, $e_1$ happens before $e_2$. Thus, both $e_1$ and $e_2$ happen before $e_3$. The vector clock $V_3$ is computed so that $V_3 \sqsupseteq V_1 \sqcup V_2$ where $x \sqsupseteq y$ (the partial order) indicates that $x[i] \geq y[i]$ for all components $i$, and $x \sqcup y$ (the join operation) returns a vector $z$ such that $z[i] = max(x[i], y[i])$ for all $i$. As $e_1$ happens before $e_2$, $V_2 \sqsupseteq V_1$ and therefore, it is unnecessary to use $V_1$ in the computation of $V_3$. In other words, we can compute the vector clock $V_3$ of the event $e_3$ more efficiently if we know that it is sufficient to order $e_3$ only with $e_2$. In general, an event $e$ may be HB ordered w.r.t. a large set $A$ of events. However, ordering $e$ only with a small subset of $A$ may be sufficient to order it with all events in $A$. Our aim is to efficiently compute such subsets and thereby, reduce the overall time required for HB computation.

With this aim, we augment the vector clock based HB computation by maintaining another data structure, called the *event graph*. For an event $e$, we call the set of events that could happen before $e$ as *the set of candidate events* for $e$. The HB rules must be evaluated on this set to identify all events that $e$ is ordered with. The event graph maintains a subset of the HB relation required for efficient computation of the set of candidate events. We design an algorithm, called EventTrack, which queries the event graph for the set of candidate events for each event $e$ and computes the vector clock of $e$ only with respect to a subset of its candidate set. To achieve efficiency, we exploit the already established order between events to prune the set of candidate events without sacrificing soundness of the HB computation. In the example above, the event graph has only $\{e_2\}$ as the set of candidate events for $e_3$, as desired.

EventRacer [2] – a state-of-the-art race detector for Android applications, also addresses the problem of efficient computation of candidate events. However, for the trace in Figure 1, it computes $\{e_1, e_2\}$ as the candidate events when only $\{e_2\}$ is sufficient.

We have implemented EventTrack as a standalone tool and evaluated it on execution traces obtained from eight popular Android applications. Its performance is compared with a baseline implementation, called ER-baseline, which uses the HB computation algorithm of EventRacer. We show that the parameters affecting the time complexity of the two techniques are incomparable. Nevertheless, in our experiments, EventTrack yielded a speedup ranging from 1.8X to 10.3X compared to ER-baseline. In addition, the cumulative size of sets of candidate events identified by EventTrack using event graph was on an average 4.8 times smaller than those identified by ER-baseline. This shows the effectiveness of event graph in identifying already ordered events and avoiding unnecessary computations to discover or inspect such events. EventTrack was found to consume 1.4 times more memory than ER-baseline on an average, which we consider to be a reasonable memory overhead incurred in exchange for significant speedup.

The contributions of this paper are as follows:

- We present EventTrack which maintains the event graph, in addition to vector clocks, to efficiently compute HB relation for event-driven programs.
- We implement EventTrack as a standalone tool that can be integrated into various concurrency analyses. EventTrack is available at http://bitbucket.org/iiscseal/eventtrack.
- We experimentally evaluate EventTrack and show that it achieves an average speedup of 4.9X compared to the state-of-the-art technique on traces of eight Android programs.

## 2 EXAMPLE

In this section, we explain the limitations of EventRacer [2] using the example discussed in the Introduction, and illustrate how event graph augmented HB computation by EventTrack works on it. Android allows an event to be posted with a delay, in which case an event is dequeued only after the delay has elapsed. In this example, we assume that all the events are posted with the same delay and hence, dequeued in the same order in which they are posted.

### 2.1 HB Graph Augmented HB Computation

EventRacer [2] builds an HB graph to help compute HB order between events, apart from maintaining vector clocks. The nodes of the HB graph are all the (concurrency) operations in the trace and edges indicate HB ordering. Even though the HB relation is transitively closed, EventRacer does not saturate the HB graph with the transitive edges as transitive closure is expensive. When it comes to deciding the set of candidate events for an event $e$, it performs depth first search (DFS) on the HB graph starting from

the dequeue operation of $e$ and in the reverse direction to that of the HB edges, inspecting if the operations visited satisfy the criteria for a corresponding event to be in the set of candidate events of $e$.

In Figure 2(a), we reproduce the trace from Figure 1 and show some HB edges by solid arrows. The operations of an event handler are totally ordered with each other due to sequential semantics (program order). The same holds for the operations of the thread $t'$. We do not show program-order edges explicitly.

EventRacer analyzes the trace starting from the first operation. A common reason for HB ordering two events posted to the same thread is that the post operations are themselves ordered while the events are posted with the same delay. This is called the FIFO rule [2, 8, 17]. When at the operation $o_8$, EventRacer identifies the set of candidate events of $e_3$ by visiting the matching post operation $o_6$ and starting a (backward) DFS in the HB graph. This is shown by the edge marked ①. From the operation $o_5$, it goes to the post operation $o_4$ via the edge labeled ②. Now, it has found a post in the trace that happens before the post $o_6$ and posting to thread $t$. Hence, event $e_2$ posted by $o_6$ is a candidate event of $e_3$. Also, since $e_2$ and $e_3$ are posted with the same delay, $e_2$ is identified to happen before $e_3$ as per the FIFO rule. When identifying the set of candidate events of an event $e$ due to FIFO rule, EventRacer prunes DFS along the current path on reaching an operation posting a candidate event with the same delay as $e$. Due to this pruning, this search does not follow the dashed edge in Figure 2(a). However, another DFS from $o_5$, going along the edges ③ and ④ reaches the post at $o_1$. By the FIFO rule, the event $e_1$ posted by $o_1$ happens before $e_3$. Before propagating the value of vector clock of $e_1$ to that of $e_3$, the two vector clocks are compared and EventRacer finds that $V_3 \sqsupseteq V_1$ since $e_3$ has been transitively ordered with $e_1$ via $e_2$. Yet, it has performed an unnecessary DFS traversal to discover a redundant event $e_1$. Thus, its pruning strategy is not effective in this case.

The performance of EventRacer is also susceptible to the order of DFS traversal. In case the backward DFS follows the edge ③ prior to edge ②, event $e_1$ would be discovered before $e_2$. In this case, EventRacer would use vector clocks of both $e_1$ and $e_2$ to compute vector clock of $e_3$, thus incurring even more cost. HB order between events can be established through various rules [2, 8, 17]. EventRacer may have to perform multiple DFS traversals to evaluate each of these rules. Along with identifying new candidate events, such repeated DFS traversals may redundantly reach events already established to happen before $e$ by a prior DFS.

## 2.2 Event Graph Augmented HB Computation

In addition to vector clocks, our algorithm, EventTrack, maintains a data structure called event graph. Event graph is a directed and labeled graph with events and threads as nodes. The event graph is maintained in a way that when identifying events which happen before an event $e$, it suffices to inspect the incoming edges of $e$. In other words, the set of candidate events for $e$ can be computed by simply looking up its incoming edges in the event graph.

EventTrack analyzes the trace in Figure 1 starting from the first operation $o_1$. It assigns vector clocks to the thread $t'$ and events $e_1$, $e_2$ and $e_3$. A snapshot of the vector clocks $V_1$, $V_2$ and $V_3$ of the events $e_1$, $e_2$ and $e_3$ (after analyzing the operation $o_8$) is shown in Figure 1. In this example, the event graph has four nodes: the node

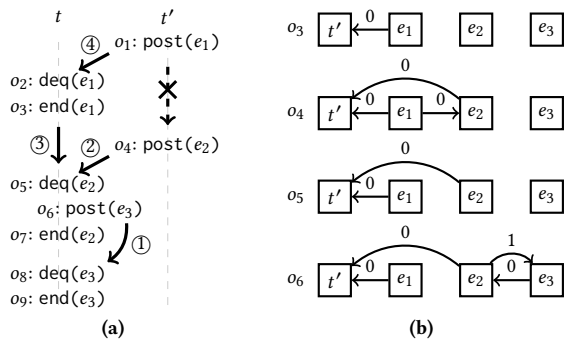


**Figure 2: (a) The trace from Figure 1 with some HB edges and annotations, and (b) The evolution of the event graph across a subset of operations of the trace from Figure 1.**

labeled $t'$ denotes the thread $t'$, and the nodes labeled $e_1$, $e_2$ and $e_3$ correspond to the events in the trace.

In Figure 2(b), we show the event graph computed by EventTrack after analyzing the operations $o_3$–$o_6$. We omit the event graphs at the other operations for brevity. The event graph obtained after analyzing the operation $o_3$, has only one edge $e_1 \xrightarrow{0} t'$. The meaning of this edge is that, `post(`$e_1$`)` happens before some operation in $t'$. An edge $e_1 \xrightarrow{0} e'$ for an event $e'$ could also mean that `post(`$e_1$`)` happens before `post(`$e'$`)`. We use $e \xrightarrow{1} e'$ to indicate that `deq(`$e$`)` happens before some operation in the event $e'$.

As the operation $o_4$ executed on the thread $t'$ posts the event $e_2$, EventTrack adds the edge $e_2 \xrightarrow{0} t'$ (see Figure 2(b)). At the same time, it propagates the incoming edge $e_1 \xrightarrow{0} t'$ of $t'$ to $e_2$ as the edge $e_1 \xrightarrow{0} e_2$. The addition of the incoming edge to $e_2$ indicates that the post of $e_1$ happens before the post of $e_2$.

When the event $e_2$ is dequeued at $o_5$, EventTrack queries the event graph to identify the set of candidate events that $e_2$ could be ordered with. This is accomplished by inspecting the incoming edges to $e_2$. Since there is an edge from $e_1$ to $e_2$, where $e_2$ is posted with the same delay as $e_1$, the vector clock $V_2$ of $e_2$ is updated to incorporate value of vector clock $V_1$ of $e_1$ due to FIFO rule. The information maintained in event graph thus aided in identifying HB ordering from $e_1$ to $e_2$.

Now that the ordering from $e_1$ to $e_2$ is encoded in the vector clock of $e_2$, there is no need to retain the edge $e_1 \xrightarrow{0} e_2$. EventTrack therefore prunes this edge as seen in the event graph tagged with $o_5$ in Figure 2(b). As we discuss in Section 3.2, pruning such edges helps maintain only the information that might be required in the future without sacrificing soundness of the HB computation.

The operation $o_6$ executed by the handler of $e_2$ posts the event $e_3$, resulting in an edge $e_3 \xrightarrow{0} e_2$. The post operation $o_6$ happens after the dequeue operation $o_5$ of $e_2$. Hence, `deq(`$e_2$`)` happens before every operation in $e_3$. To record this fact, EventTrack adds the edge $e_2 \xrightarrow{1} e_3$. The operation $o_7$ does not modify the event graph.

At the operation $o_8$, EventTrack inspects the incoming edges of $e_3$ in the event graph. There is only one edge from $e_2$ labeled with 1. As noted before, in the event-driven programs we consider, an event handler is run to completion before processing the next event in the queue. As $e_2$ is dequeued before $e_3$ (as per the edge label 1), EventTrack identifies that $e_2$ would have finished before

$e_3$. It therefore uses the vector clock $V_2$ of $e_2$ to compute the vector clock $V_3$ of $e_3$, resulting in the value shown in Figure 1.

The event $e_1$ also happens before the event $e_3$. But as remarked in the Introduction, it is unnecessary to use $V_1$ to compute $V_3$. As seen in the example, the event graph pruned the ordering between $e_1$ and $e_2$ as soon as it got reflected in the vector clock of $e_2$. Thus, this information is not propagated further in the event graph and we do not have an edge from $e_1$ to $e_3$. This saves the effort expended by a simple strategy in iterating over all prior events, or a technique such as EventRacer which may perform redundant DFS traversal to discover already ordered events.

## 3 DEFINITIONS

In this section, we first present some preliminary concepts and then define the event graph data structure.

### 3.1 Preliminary Concepts

***Program Model***. We consider an event-driven program $P$, in which, different threads communicate with each other through shared memory. Some threads in $P$ have associated event queues. Any thread or the environment can communicate with these threads by posting asynchronous events to their event queues. An event loop of a thread $t$ dequeues events from its event queue and runs their handlers one after the other on $t$. Thus, operations of different event handlers on the same thread do not interleave, but operations of different threads do. The order in which events are handled on a thread can change across executions.

Our goal in this work is to design an efficient algorithm to perform happens-before (HB) analysis of execution traces of event-driven programs. We use certain primitive operations to model the concurrency-relevant aspects of each trace. We have already presented the `post`, `deq` and `end` operations in the Introduction. We now introduce the remaining operations:

- `tinit`($t$) and `texit`($t$) indicate the start and end of a thread $t$.
- `fork`($t$,$t'$) denotes the creation of a new thread $t'$ by a thread $t$, and `join`($t$,$t'$) denotes that a thread $t$ is consuming a thread $t'$ which has completed its execution.
- `wait`($o$) indicates unblocking of a thread blocked on an object $o$, and `notify`($o$) indicates that a thread waiting on $o$ should be unblocked.
- `enable`($e$) denotes that an event $e$ has been enabled or registered, and `trigger`($e$) is emitted from within an event handler to indicate that the handler of $e$ is invoked. These are used to order a few environment events.

Two prominent modes in which events can be posted are: using delays or by posting to the front of the event queue. The case of posting an event without delay is a special case of the former. For an event $e$, $delay(e)$ gives the value of delay used when posting $e$. $foq(e)$ is true for an event $e$ iff $e$ is posted to the front of the queue of a thread. When we use $delay(e)$, we implicitly mean that $e$ is not posted to the front of the queue. An execution trace $\tau$ of $P$ is a sequence of primitive operations that is feasible at runtime.

We now introduce some notation. We will use the term *task* to refer to an event or a thread. We refer to an event handler simply by its event. Let $Th$, $Ev$, $H$ and $Op$ respectively represent the set of threads, events, tasks and operations in a trace $\tau$ where $H = Th \cup Ev$. Function $dest(e)$ returns the thread to whose queue the event $e$ is posted. Given an $a \in Op \cup H$, $thread(a)$ returns the thread that executes $a$. Function $task(z)$ returns the task which executes operation $z$. If $b = task(z)$ then we may say $z \in b$.

***Vector Clocks with Chain Decomposition***. Consider an execution trace $\tau$ of an event-driven program. Let $Chains$ be a finite set of discrete values called *chains*. The chain decomposition [10] of $\tau$ is a function $C$ that maps each operation in $\tau$ to a unique chain $c \in Chains$ such that the operations assigned to the same chain are totally ordered with each other. A *vector clock* $V$ is a vector of logical clocks with one clock assigned to each chain [23].

The operations in $\tau$ are assigned to chains in a greedy manner. We assign the same chain to all the operations of an event handler. We therefore also use $C(e)$ to refer to the chain of an event $e$. Suppose two events $e_1$ and $e_2$ executed on the same thread are totally ordered and every operation of $e_1$ is assigned a chain $c$. We assign all operations of $e_2$ the chain $c$, if `end`($e_1$) is the most recent operation that is assigned the chain $c$ when `deq`($e_2$) is assigned a chain. Operations on different threads are assigned different chains. Any other chain decomposition strategy can be applied in place of this, so long as the resulting map $C$ is consistent with the definition of chain decomposition. In Figure 1, operations of the events $e_1$, $e_2$ and $e_3$ are assigned the same chain and those of the thread $t'$ are assigned another. We therefore require only two components (logical clocks) in the vector clocks, one per chain.

Formally, a vector clock (VC) is a function $V : Chains \to \mathbb{N}$. Let $v : Op \to \mathbb{V}$ map operations $Op$ of a trace $\tau$ to their VC timestamps. Let function $hbSrc(z)$ return the set of operations in the trace that happen-before an operation $z$ due to some HB rule except purely due to transitive closure. We use $V$, $V'$ or $V_i$ where $i \in \mathbb{N}$ to denote a VC. We use the following basic operations on vector clocks [5, 23] where $c \in Chains$.

$$\bot_V = \lambda c.0 \qquad \texttt{[VC-BOT]}$$

$$V_1 \sqcup V_2 = \lambda c.max(V_1[c], V_2[c]) \qquad \texttt{[VC-JOIN]}$$

$$V_1 \sqsubseteq V_2 \text{ iff } \forall c.V_1[c] \leq V_2[c] \qquad \texttt{[VC-ORDER]}$$

$$inc_c(V) = \lambda d.\text{if } d = c \text{ then } V[c] + 1 \text{ else } V[d] \qquad \texttt{[VC-INC]}$$

$$v(z) = inc_{C(z)}(\sqcup\{v(b) \mid b \in hbSrc(z)\}) \qquad \texttt{[VC-TSTAMP]}$$

The vector clocks are partially ordered under $\sqsubseteq$. The timestamp of an operation $z$ computed using `VC-TSTAMP` ensures that for all operations $y$ that happen-before the operation $z$, $v(y) \sqsubseteq v(z)$. Whether $v(y) \sqsubseteq v(z)$ can be checked in constant time by checking if $v(y)[C(y)] \leq v(z)[C(y)]$.

***Event-ordering Rules***. Establishing HB order among operations of an event-driven program involves both 1) multi-threaded HB analysis and 2) HB analysis for ordering events posted to the same thread. The former is similar to the well-known HB analysis for pure multi-threaded programs. Efficient computation of the latter is the focus of this work. For reference, we present the rules to establish order among events posted to the *same thread* in Table 1. The binary relation $\prec_{hb}$ denotes the HB order.

The rule `FIFO(a)` states that if $e_1$ is posted before $e_2$ (according to the HB ordering $\prec_{hb}$) and the delay associated with $e_1$ is also smaller-or-equal-to that of $e_2$ then $e_1$ finishes execution before $e_2$

**Table 1: Happens-before rules for ordering events.**

| | |
|---|---|
| FIFO(a) | If post($e_1$) $\prec_{hb}$ post($e_2$) & $delay(e_1) \leq delay(e_2)$ then end($e_1$) $\prec_{hb}$ deq($e_2$). |
| FIFO(b) | If $foq(e_1)$ & $!foq(e_2)$ & post($e_1$) $\prec_{hb}$ post($e_2$) then end($e_1$) $\prec_{hb}$ deq($e_2$). |
| FoQ(a) | If $foq(e_1)$ & $!foq(e_2)$ & post($e_1$) $\prec_{hb}$ deq($e_2$) then end($e_1$) $\prec_{hb}$ deq($e_2$). |
| FoQ(b) | If $foq(e_1)$ & $foq(e_2)$ & post($e_2$) $\prec_{hb}$ post($e_1$) & post($e_1$) $\prec_{hb}$ deq($e_2$) then end($e_1$) $\prec_{hb}$ deq($e_2$). |
| NO-PRE | If deq($e_1$) $\prec_{hb}$ end($e_2$) then end($e_1$) $\prec_{hb}$ deq($e_2$). |

is dequeued. The rule FIFO(b) is similar but handles the case that $e_1$ is posted to the front and $e_2$ is not. The rule FoQ(a) has similar constraints about *foq* status of $e_1$ and $e_2$, but states that if $e_1$ is posted before $e_2$ is dequeued then $e_1$ finishes before $e_2$ starts. As an aside, we mention that this rule is a generalization of similar rules presented in [2, 8]. The rule FoQ(b) reasons about the case when both $e_1$ and $e_2$ are posted to the front. Finally, the rule NO-PRE states that if $e_1$ is dequeued before any operation of $e_2$, even end($e_2$), then $e_1$ finishes before $e_2$ starts. This is because an event handler runs to completion before another can be scheduled on that thread.

## 3.2 Event Graph

An *event graph* $G = (H, \xi)$ is a directed graph with tasks as the set of nodes and the set of edges being $\xi \subseteq H \times \{0, 1\} \times H$ where

- An edge $b \xrightarrow{0} d \in \xi$ indicates that post($b$) happens before the operation that initializes task $d$ (namely, post or fork), or post($b$) happens before some operation in task $d$.
- An edge $b \xrightarrow{1} d \in \xi$ indicates that deq($b$) happens before the operation that initializes task $d$, or deq($b$) happens before some operation in task $d$.

Thus, the event graph maintains a subset of HB relation pertaining to posts and deqs. By definition, all the edges originate from nodes corresponding to events. Threads can only have incoming edges.

Table 2 lists the operators that we shall use for manipulating an event graph $G$. Apart from the *edge addition* and *edge deletion* operators, $\oplus$ and $\ominus$, we also define an operator $\otimes$ (called *edge addition with subsumption*) which takes an edge $b \xrightarrow{l} d$ as input and updates the edge label of the edge $b \xrightarrow{l'} d$ to $l$ if the edge $b \xrightarrow{l'} d$ already exists in $G$ and $l' < l$. Otherwise, it just adds the edge $b \xrightarrow{l} d$ to the event graph. The replacement of $l'$ with $l$ happens only if $l' = 0$ and $l = 1$. This is because if deq($b$) happens before an operation $z$ in $d$ then implicitly post($b$) too happens before $z$, *i.e.*, label 1 subsumes label 0. Given an edge $b \xrightarrow{l} d$, the operator $\otimes_{Ev}$ applies $\otimes$ only if $b$ is an event (and not a thread). Otherwise, it does not change the event graph $G$. For a task $b$, $G(b)$ (called *edge retrieval*) returns the set of incoming edges to $b$ in $G$. To maintain uniformity, we use $G(b)$ instead of the more precise notation $\xi(b)$.

Assume that an HB ordering is established from an operation in a task $d$ to an operation $z$ in a task $b$. Then, event graph captures the transitive HB orderings by adding edges from sources of incoming edges of $d$ to node $b$ in the event graph $G$. Let $E = G(d)$. Operator $\uplus$ called *edge propagation with pruning* performs a propagation of edges. Let $a \xrightarrow{l} d \in E$. If $a \xrightarrow{l'} b \in G(b)$ then $l'$ needs to be replaced

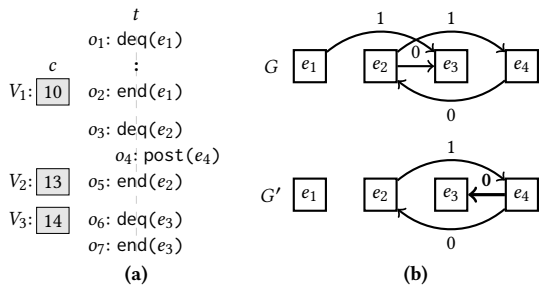


**Figure 3: Illustration of edge propagation with pruning.**

by $l$ if $l' < l$, as discussed before. Hence, $\uplus$ uses $\otimes$ to propagate the new incoming edges to $b$. To keep the graph $G$ sparse, $\uplus$ prunes existing incoming edges into $b$ which satisfy certain conditions. If end($a$) $\prec_{hb} z$, the operator $\uplus$ 1) prunes the edge $a \xrightarrow{l} b$ from $G(b)$, if it exists or 2) does not add an edge $a \xrightarrow{l} b$ to $G(b)$, even if a matching edge $a \xrightarrow{l} d$ exists in $E$. This is because, an edge $a \xrightarrow{l} b$ indicates that either deq($a$) $\prec_{hb} z$ or post($a$) $\prec_{hb} z$. If end($a$) $\prec_{hb} z$ is established prior to the application of $\uplus$, then it subsumes the ordering information captured by $a \xrightarrow{l} b$. As shown in Table 2, the operator $\uplus$ applies $\ominus$ or $\otimes$ for every edge in $E \cup G(b)$ as discussed above. As $\uplus$ iterates over all edges in $E \cup G(b)$, the complexity of $\uplus$ is linear in $|E \cup G(b)|$. Also, if VC timestamps of end($a$) and $z$ are known then as stated in Section 3.1, the VC comparison to check for HB ordering between end($a$) and $z$ is a $O(1)$ operation.

***Overloading $\uplus$ .*** In the rest of the paper, we pass the VC timestamp of the operand $z$ to $\uplus$ instead of $z$, since we will be using VC timestamps to check the HB ordering end($a$) $\prec_{hb} z$ in $\uplus$.

***Example.*** We demonstrate edge propagation with pruning on the execution trace in Figure 3(a). Let $e_1$, $e_2$ and $e_3$ be totally ordered events on thread $t$, all assigned to a chain $c$. Let $V_1$, $V_2$ and $V_3$ be the VC timestamps of end($e_1$), end($e_2$) and deq($e_3$) respectively. The event $e_2$ posts an event $e_4$. Figure 3(b) shows the event graphs tagged $G$ and $G'$ obtained before and after processing operation $o_6$ respectively. Note that not all the operations responsible for the state of the event graph before $o_6$ have been depicted. The incoming edges $G(e_3)$ of $e_3$ correspond to the subset of HB orderings with post($e_3$) as target. In particular, $G(e_3)$ indicates that deq($e_1$) $\prec_{hb}$ post($e_3$) and post($e_2$) $\prec_{hb}$ post($e_3$). Assume that an HB ordering is established from $e_2$ to $e_3$ when analyzing $o_6$, resulting in the value of $V_3$ shown in the figure. Due to this HB ordering, $G(e_2)$ should be propagated to $e_3$. Let $E = G(e_2) = \{e_4 \xrightarrow{0} e_2\}$. Then, performing $G \uplus (e_3, \{e_4 \xrightarrow{0} e_2\}, V_3)$ adds a new edge $e_4 \xrightarrow{0} e_3$ depicted in bold in $G'$. Thus, event graph has rightly established that post($e_4$) $\prec_{hb}$ deq($e_3$). From VC values, $V_1 \sqsubseteq V_3$ and $V_2 \sqsubseteq V_3$. This essentially denotes that end($e_1$) $\prec_{hb}$ deq($e_3$) and end($e_2$) $\prec_{hb}$ deq($e_3$). Thus, the incoming edges of $e_3$ from $e_1$ and $e_2$ are no more needed and $\uplus$ prunes these edges resulting in event graph $G'$.

# 4 ALGORITHM

We now present the EventTrack algorithm which uses event graphs in vector clock based HB computation. EventTrack takes an execution trace $\tau$ of an event-driven program as input and processes operations of $\tau$ in the order in which they appear in the trace. It

**Table 2: List of operators for an event graph $G = (H, \xi)$.**

| | | |
|---|---|---|
| $G \oplus (b \xrightarrow{l} d)$ | Add the edge $b \xrightarrow{l} d$ to $\xi$. | [edge addition] |
| $G \ominus (b \xrightarrow{l} d)$ | Delete the edge $b \xrightarrow{l} d$ from $\xi$ if such an edge exists. | [edge deletion] |
| $G \otimes (b \xrightarrow{l} d)$ | If $\exists l' \cdot b \xrightarrow{l'} d \in \xi$ and $l' < l$ then replace the edge label $l'$ by $l$. Else $G \oplus (b \xrightarrow{l} d)$. | [edge addition with subsumption] |
| $G \otimes_{Ev} (b \xrightarrow{l} d)$ | Perform $G \otimes (b \xrightarrow{l} d)$ only if $b \in Ev$. | [conditional edge subsumption] |
| $G(b)$ | Return the set of incoming edges to $b$ in $G$. | [edge retrieval] |
| $G \uplus (b, E, z)$ | For each $(a \xrightarrow{l} d) \in E \cup G(b)$, if $\texttt{end}(a) \prec_{hb} z$ then $G \ominus (a \xrightarrow{l} b)$ else $G \otimes (a \xrightarrow{l} b)$. | [edge propagation with pruning] |

**Table 3: List of source and target tasks for each operation $z$. For operations not listed, $source(z) = target(z) = task(z)$.**

| Operation $z$ | $d = source(z)$ | $b = target(z)$ |
|---|---|---|
| `fork(`$t$`,`$t'$`)` | $task(z)$ | $t'$ |
| `join(`$t$`,`$t'$`)` | $t'$ | $task(z)$ |
| `post(`$e$`)` | $task(z)$ | $e$ |
| `notify(`$o$`)` | $task(z)$ | task with `wait` notified by $z$ |
| `trigger(`$e$`)` | $task$(`enable(`$e$`)`) | $e$ |

maintains a pair $(T, G)$ as the *analysis state*, where $T : H \to \mathbb{V}$ maps tasks to vector clocks and $G = (H, \xi)$ is an event graph.

The vector clocks for all tasks in $\tau$ are initialized to $\bot_V$. Let $\alpha$ be the prefix of $\tau$ that has been analyzed. Let $z$ be the next operation of $\tau$ to be analyzed. Let $c = chain(z)$ be the chain that $z$ is assigned to. For ease of presentation, when processing an operation $z$, we refer to $d = source(z)$ as the task from which the HB information is to be propagated to the task $b = target(z)$. Table 3 lists the source and target tasks for all the concurrency operations. At any analysis state, the components $T$ and $G$ are maintained such that for any task $a$, $T(a)$ is the VC timestamp of the most recent operation of $a$ evaluated in $\alpha$, and $G(a)$ is the incoming edges of $a$ corresponding to a subset of `post` and `deq` operations that happen before the most recent operation of $a$ evaluated in $\alpha$.

## 4.1 Multi-threading and Post Operations

Table 4 gives the transfer functions to update the analysis state $(T, G)$ for multi-threaded operations such as fork-join, delayed posts and event dequeue operations. Recall that a post operation without delay is a special case of delayed posts. Each row of the table gives the transfer function for the operation(s) mentioned in the first column. The second column computes helper variables, which are used to update the vector clock map $T$ and the event graph $G$. Note that the algorithm assigns an operation $z$ to a chain and increments the clock corresponding to that chain only after the transfer function computes $z$'s VC timestamp.

***Multi-threaded Operations.*** The multi-threaded operations are `tinit`, `texit`, `wait`, `notify`, `fork` and `join`. The `tinit` and `texit` operations do not involve any inter-thread communication. The vector clock update for them therefore simply increments the logical clock corresponding the chain $c$ in the vector clock $T(b)$. The task $b$ is same as the thread $t$. These operations do not update the event graph $G$. As shown in Table 4, the transfer function of the `wait(`$o$`)` operation is similar. The task $b$ refers to the task that executes the `wait` operation, that is, $task(z)$.

While analyzing an operation $z = \texttt{notify}(o)$, EventTrack updates the vector clock of a task with the matching `wait` operation. In this case, $d = task(z)$ and $b$ is the task with the matching `wait` (see Table 3). As shown in Table 4, $T(b)$ is updated to $V = T(b) \sqcup T(d)$ and the clock for the chain $c$ (that $z$ is assigned to) in $T(d)$ is incremented. If $d$ is an event then due to `notify-wait` HB order and transitivity, `deq(`$d$`)` happens before `wait(`$o$`)` in task $b$. Therefore, an edge $d \xrightarrow{1} b$ is added to $G$. If $d \xrightarrow{0} b$ is already present in $G$ then the operator $\otimes_{Ev}$ updates the edge label of this edge to 1. Further, the incoming edges to $d$, denoted by $G(d)$, must be propagated to $b$ to account for transitive ordering. This edge propagation prunes any edges in $G(b) \cup G(d)$ which are already accounted for in the vector clock of $b$. The operand $V$ passed to $\uplus$ is the vector clock of the target task $b$ (as defined by the transfer function itself).

The transfer function of the `fork` operation is similar to that of `notify`. Let $z = \texttt{join}(t, t')$. For the join operation, $b = task(z)$ and $d = t'$. Note that `join` may be executed even within an event handler. By the semantics of multi-threaded programs, `texit(`$t'$`)` happens before $z$. Analyzing operations executed after `texit(`$t'$`)` in $\tau$ does not modify $T(t')$ which was last updated when evaluating `texit(`$t'$`)`. The transfer function for the join operation therefore updates $T(b) = inc_c(V)$ where $V = T(d) \sqcup T(b)$. It also propagates edges in $G(d)$ to $b$ while pruning them using the vector clock $V$.

***Delayed Posts.*** Let $z = \texttt{post}(e)$ be a delayed post. Recall that a post operation is either a delayed post or a post to front of the queue. That is, for $e$, $!foq(e)$ holds. The transfer function for the delayed post is shown in the fourth row of Table 4. The vector clock updates are similar to that of `notify` operation. As the target task $b$ learns that $d$ has been dequeued, we add $d \xrightarrow{1} b$ in $G$ (using the operator $\otimes_{Ev}$). Since $d$ is the task that posted event $b$, we also add $b \xrightarrow{0} d$ as per the definition of event graph. As usual, we propagate the incoming edges of $d$ to $b$.

***Front-of-queue (FoQ) Posts.*** Let $z = \texttt{post}(e)$ be a front-of-queue (FoQ) post. Table 5 gives the transfer function of such a post which is identical to the transfer function of a delayed post. The additional part is an update of a lookup table $\pi$. Let $d$ be the task that performs the FoQ post $z$. Let $t$ be the thread to which the event $e$ has been posted by $z$. We identify each event $a$ that is posted to the front of the queue of thread $t$ such that the post of $a$ happens before that of $d$ according to the event graph $G$. We record the set of such events into a lookup table entry $\pi(z)$ for later use.

## 4.2 Event Dequeue and Environment Events

The event-ordering rules of Table 1 are evaluated by EventTrack when analyzing the `deq` operation. The event graph constructed by

**Table 4: Transfer functions for multi-threaded operations, delayed posts and event dequeue operations. Let $z$ be the operation to be analyzed and $d = source(z)$, $b = target(z)$, $t = thread(b)$ and $c = chain(z)$.**

| Operation $z$ | Helper variables | Update $T$ | Update $G$ |
|---|---|---|---|
| $\texttt{tinit}(t)$, $\texttt{texit}(t)$, $\texttt{wait}(o)$, $\texttt{end}(e)$ | $V = T(b)$ | $T[b \mapsto inc_c(V)]$ | $G$ |
| $\texttt{fork}(t,t')$, $\texttt{notify}(o)$ | $V = T(b) \sqcup T(d)$ | $T[b \mapsto V, d \mapsto inc_c(T(d))]$ | $(G \otimes_{Ev} (d \xrightarrow{1} b)) \uplus (b, G(d), V)$ |
| $\texttt{join}(t,t')$ | $V = T(d) \sqcup T(b)$ | $T[b \mapsto inc_c(V)]$ | $G \uplus (b, G(d), V)$ |
| $\texttt{post}(e)$ s.t. $!foq(e)$ | $V = T(b) \sqcup T(d)$ | $T[b \mapsto V, d \mapsto inc_c(T(d))]$ | $((G \otimes_{Ev} (d \xrightarrow{1} b)) \oplus (b \xrightarrow{0} d)) \uplus (b, G(d), V)$ |
| $\texttt{deq}(e)$ s.t. $!foq(e)$ | $A = \{a \in Ev \mid dest(a) = t \wedge (a \xrightarrow{l} b) \in G(b) \wedge (l = 1 \vee foq(a) \vee delay(a) \leq delay(e))\}$ $V = T(b) \sqcup \left( \bigsqcup_{a \in A} T(a) \right)$ | $T[b \mapsto inc_c(V)]$ | $G \uplus \left( \biguplus_{a \in A} (b, G(a), V) \right)$ |
| $\texttt{deq}(e)$ s.t. $foq(e)$ | $A = \{a \in Ev \mid dest(a) = t \wedge (a \xrightarrow{1} b) \in G(b)\}$ $V = T(b) \sqcup \left( \bigsqcup_{a \in A} T(a) \right)$ | $T[b \mapsto inc_c(V)]$ | $G \uplus \left( \biguplus_{a \in A} (b, G(a), V) \right)$ |

analyzing operations in prefix $\alpha$ of trace $\tau$ is queried to compute the set of candidate events of an event $e$ being dequeued.

***Evaluation of FIFO and NO-PRE Rules.*** Consider a dequeue operation $z = \texttt{deq}(e)$ where $e$ has not been posted to the front of the event queue. From Table 3, $b = e$. The fifth row of Table 4 gives the transfer function to compute a set $A$ of events w.r.t. which $e$ should be ordered. The only operation whose transfer function adds incoming edges into $e$ in event graph $G$ prior to processing $\texttt{deq}(e)$, is $\texttt{post}(e)$. Thus each incoming edge $a \xrightarrow{l} b \in G(b)$ indicates either $\texttt{deq}(a) \prec_{hb} \texttt{post}(b)$ or $\texttt{post}(a) \prec_{hb} \texttt{post}(b)$, therefore satisfying all the happens-before constraints in the antecedent of FIFO(a), FIFO(b) or NO-PRE rules in Table 4. Hence, we consider the source event corresponding to each incoming edge in $G(b)$ and posted to $t$ (where $t = thread(b)$), as a member of the set of candidate events of $b$. A candidate event $a$ is included in the set $A$ only if it satisfies all other constraints in the antecedent of some rule in Table 1. For example, if $a \xrightarrow{l} b \in G(b)$ such that $l = 0$, then it indicates that $\texttt{post}(a) \prec_{hb} \texttt{post}(b)$. Additionally, if $a$ is posted to front of $t$'s queue then $a$ happens before $b$ by FIFO(b) rule. If $l = 0$ but $a$ is posted with a delay then $a$ can be added to $A$ only if the delay constraint of the rule FIFO(a) is satisfied. However, if $l = 1$ then $\texttt{deq}(a) \prec_{hb} \texttt{post}(b)$ which implies $\texttt{deq}(a) \prec_{hb} \texttt{deq}(b)$. In this case, $a$ is included in $A$ since it satisfies the NO-PRE rule.

Let $V$ be the join of $T(b)$ with $T(a)$ for all $a \in A$. As shown in Table 4, VC timestamp of $\texttt{deq}(e)$ is computed as $inc_c(V)$ and the edges $G(a)$ for each $a \in A$ are propagated to $b$.

For $z = \texttt{deq}(e)$ where $e$ is posted to the front of the event queue, the set $A$ of events that happen before $e$ is computed as those events which execute on the same thread as $e$ and from which $b$ has incoming edges labeled 1. Such edges satisfy the antecedent of the NO-PRE rule. Note that the FIFO rules cannot order an event posted to front of the queue of a thread $t$ w.r.t. any other event posted to $t$.

***Evaluation of FoQ Rules.*** As seen in Table 1, the FoQ rules require an ordering of the form $\texttt{post}(a) \prec_{hb} \texttt{deq}(e)$ in the antecedents. Therefore, EventTrack evaluates them immediately after the transfer function of the dequeue operation $\texttt{deq}(e)$ has been computed. It identifies each event $a$ posted to $thread(e)$ such that $a \xrightarrow{0} e \in G(e)$ as a candidate event since this edge means that $\texttt{post}(a) \prec_{hb} \texttt{deq}(e)$. Let $t$ be $thread(e)$. If $a$ is posted to the front of the queue of $t$ and $!foq(e)$, then the order between $a$ and $e$ is inferred as per FoQ(a). If $e$ is itself posted to the front of queue then we also need to check whether $\texttt{post}(e) \prec_{hb} \texttt{post}(a)$ to decide whether FoQ(b) applies. We cannot simply look up the event graph for this because analysis of the intermediate operations in the trace may have updated the edges incident on $a$. We therefore evaluate the ordering of post operations of $a$ and $e$ on the edges stored in the lookup table $\pi$ for $\texttt{post}(a)$. As discussed in Section 4.1, these are stored when the FoQ post operations are analyzed. Hence, they capture the HB orderings that hold for the post operation.

If an FoQ rule is applicable, the VC of $e$ and the set of incoming edges to $e$ in the event graph are updated similar to the dequeue operations. As new HB orders may be established by the FoQ rules, they are applied iteratively until no new HB orders can be derived.

***Environment Events.*** As described in Section 3.1, $\texttt{enable}(e)$ indicates that an environment event $e$ is enabled and $\texttt{trigger}(e)$ indicates that $e$ is being handled. As shown in Table 5, when an operation $z = \texttt{enable}(e)$ is analyzed, EventTrack increments the logical clock of the chain $c$ that $z$ belongs to. The event graph remains unchanged. Subsequent operations may update the vector clock and incoming edges to $b = task(z)$. However, we need them to order the matching trigger operation. Therefore, EventTrack stores the vector clock $V$ and the edges $G(b)$ in a lookup table $\Pi$, with $\texttt{enable}(e)$ as the key. When an operation $z = \texttt{trigger}(e)$ is analyzed, EventTrack identifies corresponding enable operation, say $y$, and takes join of $T(b)$ with $\Pi(y).\mathbb{V}$ where $b = task(z)$ and

**Table 5: Transfer functions and auxiliary computations for front-of-queue (FoQ) posts and environment events. Let $z$ be the operation to be analyzed and $d = source(z)$, $b = target(z)$, $t = thread(b)$ and $c = chain(z)$.**

| Operation $z$ | Helper variables | Update $T$ | Update $G$ | Update lookup table |
|---|---|---|---|---|
| `post(`$e$`)` s.t. $foq(e)$ | $V = T(b) \sqcup T(d)$, $I = G(d)$ | $T[b \mapsto V,$ $d \mapsto inc_c(T(d))]$ | $((G \otimes_{Ev} (d \xrightarrow{1} b)) \oplus (b \xrightarrow{0} d))$ $\uplus (b, G(d), V)$ | $\pi[z \mapsto \{a \in Ev \mid dest(a) = t$ $\wedge (a \xrightarrow{0} d) \in I \wedge foq(a)\}]$ |
| `enable(`$e$`)` | $V = T(b)$ | $T[b \mapsto inc_c(V)]$ | $G$ | $\Pi[z \mapsto \langle V, G(b)\rangle]$ |
| `trigger(`$e$`)` | $y =$ `enable(`$e$`)`, $V = T(b) \sqcup \Pi(y).\mathbb{V}$ | $T[b \mapsto inc_c(V)]$ | $(G \otimes_{Ev} (d \xrightarrow{1} b)) \uplus (b, \Pi(y).\xi, V)$ | – |

$\Pi(y).\mathbb{V}$ refers to the vector clock stored for $y$ in the lookup table $\Pi$. Let $d = source(z)$ be the task that executed the enable operation $y$. The event graph is updated by adding an edge $d \xrightarrow{1} b$ and propagating the edges $\Pi(y).\xi$ stored in the lookup table to $b$.

### 4.3 Vector Clock Join over Event-covering Set

Let $E$ be the set of events that happen before an event $e$. We define a subset $S \subseteq E$ as an *event-covering set of $e$ w.r.t. $E$*, if ordering $e$ w.r.t. the events in $S$ in turn orders $e$ w.r.t. all the events in $E$. An obvious way to reduce the cost of updating the VC of $e$ w.r.t. $E$ is to identify a smaller event-covering set $S$ and update VC of $e$ only w.r.t. $S$. We call this as the *vector clock join over event-covering set* and propose an iterative solution to accomplish it as follows.

We first define a map $\Sigma : Ev \rightarrow \mathbb{N}$ which maps each event in the trace $\tau$ to an integer ID such that for every pair of events $e_1$ and $e_2$, if `deq(`$e_1$`)` is executed prior to `deq(`$e_2$`)` in $\tau$ then $\Sigma(e_1) < \Sigma(e_2)$. Let $V$ be the vector clock of an event $e$. Given a set of events $A$ identified to happen before $e$ and sorted by their $\Sigma$ IDs, the set $A$ is processed in the descending order. Assume that an event $e_i \in E$ selected this way has a vector clock $V_i$ such that $V_i[C(e_i)] > V[C(e_i)]$. Then, $e_i$ is the most recently dequeued event among the unprocessed events of $A$, with which $e$ is not yet ordered. Then, we update the vector clock $V$ of $e$ as $V \leftarrow V \sqcup V_i$. However, if $V_i[C(e_i)] \leq V[C(e_i)]$ then we do not join $V$ and $V_i$, since $e_i$ is already established to happen before $e$, and move to the next event in $A$. The subset of events in $A$ with which $V$ is joined, is an event-covering set of $e$ w.r.t. $A$.

When evaluating `FIFO`, `NO-PRE` and `FoQ` rules on an event $e$ as described in Section 4.2, we perform VC join over an event-covering set of $e$ w.r.t. $A$ instead of the entire set $A$. Also, we update $G(e)$ through edge propagation only w.r.t. the event-covering set. The set of incoming edges to $e$ is sorted by the $\Sigma$ IDs of the source events. A new edge is inserted in a way that the edge list remains sorted. Since $G(e)$ is sorted, the VC join over an event-covering set of $e$ is linear in $|G(e)|$.

### 4.4 Correctness and Complexity

The vector clock updates that EventTrack does are standard for event-driven programs [2]. The main difference is that EventTrack queries the event graph to identify the events that may happen before an event $e$. The algorithm therefore must ensure that if $e' \prec_{hb} e$ then there is an edge $e' \xrightarrow{l} e$ in the event graph or that, the vector clocks of $e'$ and $e$ are already ordered appropriately. To avoid spurious HB orderings, the converse is also required.

The complexity of computing candidate events by EventTrack is linear in the maximum in-degree of nodes in the event graph and size of event-covering sets. The complexity of computing candidate events for EventRacer is linear in the number of events and the number of DFS traversals per event. The complexities of EventTrack and EventRacer are incomparable. We empirically show that EventTrack can be more efficient than EventRacer in practice. The correctness and complexity arguments are available at [14].

## 5 IMPLEMENTATION

EventTrack is implemented as a standalone `C++` program which analyzes execution traces consisting of concurrency operations described in Section 3.1. The vector clock data structure is implemented using a `C++` container `std::vector`. Event graph is implemented as an adjacency list with the incoming edges of every task ordered as per the $\Sigma$ map (defined in Section 4.3). This facilitates efficient computation of event graph operations such as edge propagation with pruning, and identification of event-covering sets. We have also implemented a simple check to prune some edges in the event graph based on the DFS pruning strategy of EventRacer.

We generate execution traces using a tool called DroidRacer [16], a dynamic race detector for Android programs (apps). The race detection aspect of DroidRacer was disabled, and its Android instrumentation was modified to only log operations relevant for the computation of HB relation. In addition to the delayed posts and posts to front of queue discussed earlier, Android apps can perform post at-time and can post IdleHandlers. EventTrack analyzes them using the relevant HB rules from [2]. Further, there can be framework-specific orderings between environment events, which are handled as described in [17].

We compare the cost of HB computation of EventTrack with the DFS-based analysis of EventRacer. We could not directly use EventRacer's HB computation engine because it is integrated in the race detector which is only available as an executable [1]. We therefore implemented the algorithm in [2] and call it ER-baseline. Our technique of vector clock join over event-covering sets is orthogonal to the use of event graphs. We integrated it into ER-baseline to make it compute smaller event-covering sets for better performance. We call it ER-opt. Source code of EventTrack, ER-baseline and ER-opt, and our dataset are publicly available at [15].

## 6 EVALUATION

### 6.1 Experimental Setup

We collected execution traces from eight popular Android apps belonging to various categories such as social networking, online shopping and email client. The traces were collected by exercising hundreds of UI events on these apps either by a human user, or by

**Table 6: Characteristics of execution traces.**

| Android apps | #Op | #Th | #Ev | #loop | #Chain |
|---|---|---|---|---|---|
| Facebook (fb) | 176486 | 104 | 58380 | 18 | 3487 |
| Messenger (mms) | 108790 | 1149 | 33491 | 134 | 13704 |
| Remind Me (rm) | 82322 | 22 | 26535 | 19 | 4415 |
| Twitter (twtr) | 56700 | 567 | 16057 | 520 | 4669 |
| SGT Puzzles (sgt) | 55578 | 205 | 16102 | 3 | 4727 |
| BBC News (bbc) | 38963 | 203 | 12557 | 56 | 3125 |
| K-9 Mail (k9) | 27950 | 51 | 8684 | 22 | 2460 |
| Flipkart (kart) | 24460 | 147 | 7442 | 27 | 1956 |

the automated UI exploration engine of DroidRacer. Traces of varied lengths were collected to test the scalability of EventTrack.

Table 6 reports the concurrency characteristics of a representative trace of each app. It gives the number of operations, threads, events and the count #loop of subset of threads that have event loops. Note that the operations reported only include concurrency operations. We can see that these traces exhibit high degree of concurrency in both multi-threaded and event-driven aspects. The table also reports the number of chains created to group the operations of each trace. The number of chains of events identified by both EventTrack and ER-baseline is the same since both the techniques compute equivalent HB relation and ER-baseline uses the same chain assignment rules as EventTrack.

All the experiments were performed on a single core of a 64-bit machine with Intel Core i7-4700MQ 3.2GHz CPU and 16GB RAM.

## 6.2 Experimental Results

***Efficiency of EventTrack.*** Figure 4 shows the time taken by EventTrack, ER-baseline and ER-opt to compute HB relation for the traces given in Table 6. Note that the reported time does not include the time taken for pre-processing the trace, which was found to be negligible and equal for all the techniques.

For most of the traces, EventTrack computes HB relation within hundreds of milliseconds. Even for longer traces from apps such as Facebook, Messenger and Remind Me, EventTrack finishes its run within 1 to 3.5 seconds. Thus, the event graph augmented HB computation of EventTrack is quite efficient in practice.

***Comparison of Runtimes of Different Tools.*** As can be seen from Figure 4, EventTrack is faster than ER-baseline and ER-opt on all the traces. Over the longer traces, ER-baseline takes 16 seconds for Facebook and about 10 seconds for traces from Messenger, Remind Me and SGT Puzzles. This is significantly more than the time taken by EventTrack on the same traces. EventTrack has a speedup ranging from 1.8X to 10.3X across these traces. The average speedup of EventTrack compared to ER-baseline is 4.9X. Recall that ER-opt is a version of ER-baseline that we optimized by using our technique of vector clock join over event-covering sets (see Section 5). EventTrack is faster than even ER-opt on all traces as per Figure 4, and has an average speedup of 4.3X.

***Comparison of Number of Candidate Events.*** Both EventTrack and ER-baseline compute the same HB relation once they finish the analysis. Yet, there is difference in the runtimes of the two as stated above. We did a fine grained analysis of the reasons for this. Both EventTrack and ER-baseline use a common data structure, vector clocks, to represent the HB relation, and use event graph and HB graph respectively to compute candidate events required to ultimately identify HB ordered events.

In Figure 5, we plot the cumulative count of candidate events identified by EventTrack and ER-baseline across all events in the traces analyzed. We observe that the cumulative count of candidate events identified by ER-baseline by traversing the HB graph, is on an average 4.8 times larger than the count of candidate events computed by EventTrack by inspecting the incoming edges of events in the event graph. Thus, the use of event graph in EventTrack is more effective in pruning unnecessary events from the sets of candidate events than DFS pruning used by ER-baseline on the HB graph. Thus, ER-baseline performs unnecessary DFS traversals on evaluated traces, that may identify redundant candidate events.

***Effectiveness of Event-covering Sets.*** Both the techniques inspect the candidate events to identify pairs of events to be ordered. A pair of events is ordered by performing a vector clock join which is expensive and reducing the number of vector clock joins improves efficiency. Figure 6 shows the number of pairs of events ordered by EventTrack and ER-baseline by joining their vector clocks. This is same as the cumulative size of event-covering sets identified by the two techniques for each of the traces in Table 6. Not only did EventTrack identify smaller sets of candidate events (Figure 5), it also computed smaller event-covering sets than ER-baseline. As seen in Figure 6, as a result, EventTrack performed 30% lesser number of vector clock joins on an average.

ER-opt computes the same event-covering sets as EventTrack. Figure 4 shows that ER-opt outperforms ER-baseline. However, as the number of candidate events identified by ER-baseline (and consequently, ER-opt) is much higher, the computation of smaller event-covering sets alone is not enough to outperform EventTrack.

***Memory.*** Both the techniques have a peak memory consumption within a few hundreds of MB as measured using Valgrind, except for Messenger for which both consume more than 1 GB due to larger size of VCs than others. On an average EventTrack consumed 1.4 times more memory than EventRacer to maintain the event graph, but offered speedup in return.

## 6.3 Threats to Validity

The primary threats to internal validity are faults in the implementation of the two techniques and failure to incorporate relevant optimizations for ER-baseline. To mitigate the former threat, we extensively tested both the techniques on many execution traces, and validated that both were computing the same VC timestamps for all the operations in each of the traces. The algorithm and data structures of ER-baseline are based on [2]. In order to address the latter threat, firstly we implemented ER-baseline in such a way so that it does not suffer from any overhead of race detection, since our focus was only to compute the HB relation. Apart from implementing all the optimizations described in [2], we have also carefully incorporated additional strategies to prune DFS so that ER-baseline can efficiently evaluate various event-ordering rules.

Threats to external validity may be present since we cannot guarantee that EventTrack performs better than ER-baseline on

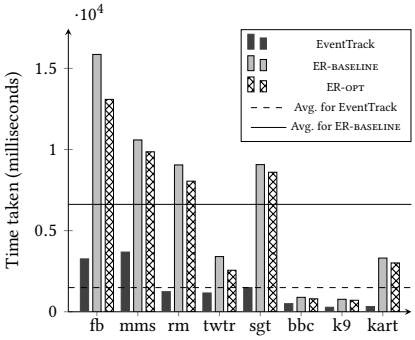


**Figure 4: Time taken to compute HB relation for traces in Table 6.**

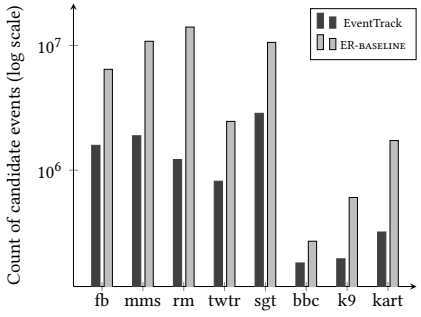


**Figure 5: Cumulative count of candidate events computed for each trace.**

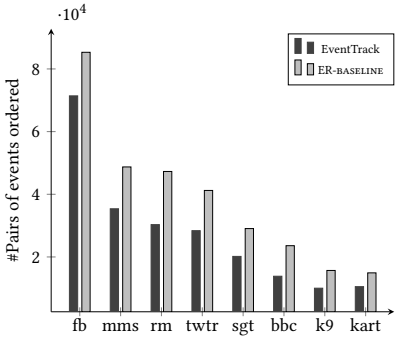


**Figure 6: Number of pairs of events ordered through vector clock joins.**

execution traces other than those listed in Table 6. However, to minimize this threat, we have evaluated the two techniques on popular and mature apps belonging to various categories. Also, each of the tested execution traces has been collected by performing long interactions with the corresponding app and subjecting it to real usage scenarios typical for the app. In the process, we have obtained long and complex traces. We believe that for traces with similar HB characteristics our results should hold.

## 7 RELATED WORK

Among concurrency bugs common in multi-threaded programs, data races which occur due to absence of ordering between conflicting operations are typically detected with the help of HB relation over program traces [4, 5, 20, 22, 26]. Since HB relation is transitively closed, computing it can be expensive. Vector clock [18] is a data structure which reduces the time complexity of reasoning about transitive ordering between operations. Vector clock has been adapted to efficiently compute HB relation for the detection of data races in multi-threaded programs [5, 22]. A few atomicity violation checkers [3, 6] use HB relation to capture ordering of synchronization, conflicting and causally ordered operations so that a cycle detected on the HB graph indicates an atomicity violation.

Smaragdakis et al. [25] define a relation called causally-precedes (CP) which is weaker than the classical HB relation defined for multi-threaded programs but effective in detecting more data races quickly. While vector clock is efficient at recursively identifying HB orderings due to transitivity ($a \prec_{hb} b$ and $b \prec_{hb} c$ implies $a \prec_{hb} c$), it does not maintain sufficient information to efficiently compute orderings due to rules of the form $a \prec_{hb} b$ implies $c \prec_{hb} d$. The reason for this is discussed earlier in the paper in the context of event-ordering rules which have this structure. Since computing CP relation involves recursively evaluating rules of both these kinds, the technique presented in [25] does not use vector clocks. Instead, they implement the CP computation algorithm as a Datalog program. It would be interesting to explore this approach for computing HB relation for event-driven programs.

Existing literature present rules to compute HB relation for event-driven framework such as client side web-applications having a single thread with an event loop [21, 23], smartphone platforms such as Android with multiple threads and event loops [2, 8, 17], and GUI libraries whose event handlers are capable of spawning programmatic event loops [24], aiding in the detection of concurrency bugs manifested due to data races. These existing work use varied techniques to compute HB relation. For example, DroidRacer [17] and SparseRacer [24] compute HB relation as a directed graph over operations or sets of operations. Both the techniques maintain a worklist of computed HB edges and recursively derive new edges using applicable HB rules. Absence of vector clock in these techniques makes transitive closure and consequently computation of candidate events expensive. EventRacer [2] augments vector clocks with HB graph over operations, and performs DFS traversals over this graph to identify candidate events.

In a concurrent work, Hsiao et al. [7] propose a data structure called AsyncClock to compute the set of candidate events for an event. While EventTrack maintains a subset of HB orderings relevant to evaluate event-ordering rules in the form of event graph, the technique proposed in [7] embeds relevant HB orderings as an AsyncClock and computes event-ordering rule specific AsyncClocks for each event in a given trace. It would be nice to compare EventTrack and AsyncClock approaches in future.

## 8 CONCLUSIONS AND FUTURE WORK

Event-driven programming is widely used for developing scalable programs. Concurrency analysis of event-driven programs has therefore become an active area of research. Happens-before analysis is a central analysis that forms the basis of various other analyses. We have identified that computing ordering among events is a major bottleneck in HB computation of event-driven programs and presented the EventTrack algorithm which uses a novel data structure, called event graph, for efficient HB computation. In our experiments on traces of eight Android apps, EventTrack yielded an average speedup of 4.9X compared to the state-of-the-art.

In future, we plan to implement event graphs using different data structures that can enable better pruning of event graph edges whose source events are guaranteed to be ordered due to transitivity. It would also be interesting to identify more opportunities to combine the pruning strategies of EventTrack and EventRacer.

## ACKNOWLEDGMENTS

Pallavi Maiya thanks Google India for its support through a PhD Fellowship in Programming Languages and Compilers.

# REFERENCES


[1] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. EventRacer for Android. (2015). Retrieved May 10, 2017 from http://eventracer.org/android

[2] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable Race Detection for Android Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 332–348. DOI: http://dx.doi.org/10.1145/2814270.2814303

[3] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. 2014. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 28–39. DOI: http://dx.doi.org/10.1145/2594291.2594323

[4] Mark Christiaens and Koen De Bosschere. 2001. TRaDe, a Topological Approach to On-the-fly Race Detection in Java Programs. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1267847.1267862

[5] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133. DOI: http://dx.doi.org/10.1145/1542476.1542490

[6] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 293–303. DOI: http://dx.doi.org/10.1145/1375581.1375618

[7] Chun-Hung Hsiao, Satish Narayanasamy, Essam Muhammad Idris Khan, Cristiano L. Pereira, and Gilles A. Pokam. 2017. AsyncClock: Scalable Inference of Asynchronous Event Causality. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 193–205. DOI: http://dx.doi.org/10.1145/3037697.3037712

[8] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race Detection for Event-driven Mobile Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 326–336. DOI: http://dx.doi.org/10.1145/2594291.2594330

[9] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. 2016. Automatically Verifying and Reproducing Event-based Races in Android Apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 377–388. DOI: http://dx.doi.org/10.1145/2931037.2931069

[10] H. V. Jagadish. 1990. A Compression Technique to Materialize Transitive Closure. *ACM Trans. Database Syst.* 15, 4 (Dec. 1990), 558–598. DOI: http://dx.doi.org/10.1145/99935.99944

[11] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. 2015. Stateless Model Checking of Event-driven Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 57–73. DOI: http://dx.doi.org/10.1145/2814270.2814282

[12] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. DOI: http://dx.doi.org/10.1145/359545.359563

[13] Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. 2016. Partial Order Reduction for Event-Driven Multi-threaded Programs. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. Springer-Verlag New York, Inc., New York, NY, USA, 680–697. DOI: http://dx.doi.org/10.1007/978-3-662-49674-9_44

[14] Pallavi Maiya and Aditya Kanade. 2017. Correctness Proofs and Complexity Analysis of EventTrack Algorithm. (2017). Retrieved May 10, 2017 from http://www.iisc-seal.net/publications/eventtrack-appendix.pdf

[15] Pallavi Maiya and Aditya Kanade. 2017. EventTrack. (2017). Retrieved May 10, 2017 from http://bitbucket.org/iiscseal/eventtrack

[16] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. DroidRacer. (2014). Retrieved May 10, 2017 from http://www.iisc-seal.net/droidracer

[17] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race Detection for Android Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 316–325. DOI: http://dx.doi.org/10.1145/2594291.2594311

[18] Friedemann Mattern. 1989. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms* 1, 23 (1989), 215–226.

[19] Zigurd Mednieks, Laird Dornin, G Blake Meike, and Masumi Nakamura. 2012. *Programming Android*. O'Reilly Media, Inc.

[20] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*. ACM, New York, NY, USA, 167–178. DOI: http://dx.doi.org/10.1145/781498.781528

[21] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 251–262. DOI: http://dx.doi.org/10.1145/2254064.2254095

[22] Eli Pozniansky and Assaf Schuster. 2003. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*. ACM, New York, NY, USA, 179–190. DOI: http://dx.doi.org/10.1145/781498.781529

[23] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA '13)*. ACM, New York, NY, USA, 151–166. DOI: http://dx.doi.org/10.1145/2509136.2509538

[24] Anirudh Santhiar, Shalini Kaleeswaran, and Aditya Kanade. 2016. Efficient Race Detection in the Presence of Programmatic Event Loops. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 366–376. DOI: http://dx.doi.org/10.1145/2931037.2931068

[25] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 387–400. DOI: http://dx.doi.org/10.1145/2103656.2103702

[26] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and Surviving Data Races Using Complementary Schedules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 369–384. DOI: http://dx.doi.org/10.1145/2043556.2043590