ASYNCCLOCK: Scalable Inference of Asynchronous Event Causality

Chun-Hung Hsiao Satish Narayanasamy
Essam Muhammad Idris Khan[1]
University of Michigan
{chhsiao,nsatish,essam}@umich.edu

Cristiano L. Pereira
Gilles A. Pokam
Intel, Inc.
{cristiano.l.pereira,gilles.a.pokam}@intel.com

Abstract

Asynchronous programming model is commonly used in mobile systems and Web 2.0 environments. Asynchronous race detectors use algorithms that are an order of magnitude performance and space inefficient compared to conventional data race detectors. We solve this problem by identifying and addressing two important problems in reasoning about causality between asynchronous events.

Unlike conventional signal-wait operations, establishing causal order between two asynchronous events is fundamentally more challenging as there is no common handle they operate on. We propose a new primitive named ASYNCCLOCK that addresses this problem by explicitly tracking causally preceding events, and show that ASYNCCLOCK can handle a wide variety of asynchronous causality models. We also address the important scalability problem of efficiently identifying heirless events whose metadata can be reclaimed.

We built the first single-pass, non-graph-based Android race detector using our algorithm and applied it to find errors in 20 popular applications. Our tool incurs about 6x performance overhead, which is several times more efficient than the state-of-the-art solution. It also scales well with the execution length. We used our tool to find 147 previously unknown harmful races.

CCS Concepts • **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → **Program analysis**

Keywords Data Races; Causality; Happens-before; Event-Driven; Asynchronous; Android

[1] He is currently at Amazon, Inc. This work was done when he was at the University of Michigan.

ASPLOS '17, April 08 - 12, 2017, Xi'an, China

ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037712

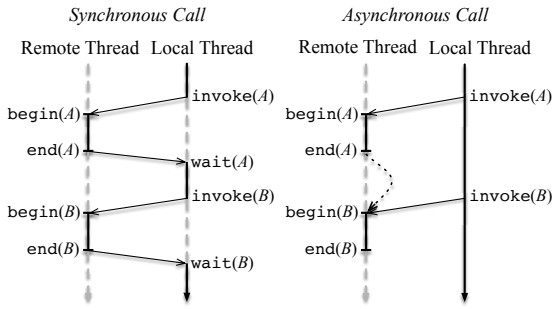


Figure 1: Synchronous versus asynchronous events.

1. Introduction

Asynchronous or event-driven programming has become the dominant programming model in the last few years. In this model, computations are posted as events to an event queue from where they get processed asynchronously by the application. A huge fraction of computing systems built today use asynchronous programming. There are now more than two million mobile applications available between the Apple App Store and Google Play, which are all written using asynchronous programming. All the Web 2.0 JavaScript applications (e.g., Gmail, Facebook) use asynchronous programming. Distributed servers (e.g., Twitter, LinkedIn, PayPal) built using actor-based languages (e.g., Scala) and platforms such as `node.js` rely on asynchronous events for scalable communication. Internet-of-Things (IoT), wearable computers, embedded systems, sensor networks, desktop GUI applications, etc., all rely on the asynchronous programming model. Even database transactions are driven by user queries.

Asynchronous programming models enables programmers to effectively express concurrency in these I/O intensive applications. Not surprisingly, the high degree of concurrency in these programs lead to a number of concurrency errors. In spite of the importance of asynchronous programming models, concurrency tools that help programmers ensure the correctness of these programs has generally been lacking.

Unlike conventional multithreaded programs, the happens-before relations, or *causal orders* between events in asynchronous programs are more sophisticated. Recent work has defined causality models for JavaScript [16, 17] and Android [6, 11, 13] for ordering asynchronous events and finding races (unordered conflicting memory accesses). However, these asynchronous race detectors use algorithms that are an order of magnitude performance and space inefficient compared to conventional data race detectors [10]. We solve this problem by identifying and addressing two important problems in reasoning about causality between asynchronous events.

Establishing happens-before relation between asynchronous events. We observe that inferring happens-before relation in an asynchronous programming model is fundamentally harder than in a synchronous model. In a synchronous program, happens-before relations between concurrent threads are trivially established by deriving the logical time of a wait (equivalently, lock, begin, *etc*.) from the time of its corresponding signal (release, send, *etc*.). These operations can be related through the handle (*e.g.*, synchronization variable) they operate on. In an asynchronous program, however, it is more challenging, as illustrated in Figure 1. Asynchronous programming models such as Android and JavaScript guarantee that two events are casually ordered if their sends are ordered. Establishing the happens-before relation between two ordered events, such as A and B in Figure 1, is challenging as there is no explicit program handle to relate them. Note that we cannot assume a program order between two events executed in a thread as their order may change in another execution, if their sends are not guaranteed to be ordered. To determine an event's causal predecessors, prior solutions [6, 11, 13, 16, 17] built a happens-before graph of all events and synchronization operations, and iteratively traversed this graph for every event, which is inefficient, especially as the graph size grows with the program execution.

We introduce a primitive called AsyncClock to realize an efficient non-iterative algorithm. An AsyncClock keeps track of causally preceding events, whose end time may not yet have been resolved. A newly created event inherits the AsyncClock from the parent thread. When the event begins its execution, it computes its logical time by taking a join of the end times of causally preceding events in the AsyncClock, which are guaranteed to have completed by then. We show that AsyncClock can be generalized to handle a wide variety of causality rules for asynchronous events.

Scalability. The second challenge is keeping track of times of events such that it scales in terms of performance and space with the program execution length. Unlike synchronous programs that execute only dozens of threads, asynchronous programs may execute hundreds to thousands of events every minute. Since the times of past events cannot be simply discarded as they will be used to compute the logical time of their future succeeding events, it is important to identify events that have no future *immediate causal successors* and reclaim their metadata (end time and AsyncClocks) to achieve good scalability. Prior work [6, 11, 13, 16, 17] did not address this problem, and requires large memory to analyze long executions.

We provide several solutions to address the scalability problem noted above. We propose methods to efficiently identify *heirless* events that can no longer have immediate successors. Unfortunately, not all heirless events can be identified efficiently. To address this issue, we exploit the intuition that an old event and a recent event are unlikely to be re-ordered in an alternative execution. Any concurrency error between two such events is unlikely to manifest, and thus low priority for developers to fix. Based on this assumption, we choose to free old events. We conservatively set the time threshold for classifying old events so high that this optimization does not result in any false negatives in our empirical evaluation. Finally, to further reduce the space overhead, we propose a sparse representation for an AsyncClock to take advantage of our observation that an event often have few predecessors.

We built the first single-pass, non-graph-based data race detector for Android's asynchronous programming model. Compared to a recent solution, EventRacer [6], we show that AsyncClock is 8x faster, used 87% less memory, and scales well in terms of both performance and space. The average performance overhead of collecting and analyzing the traces of 20 common Android applications is about 6x. Thus, we also meet our goal in realizing a tool for asynchronous programs that is as efficient as conventional data race detectors [10].

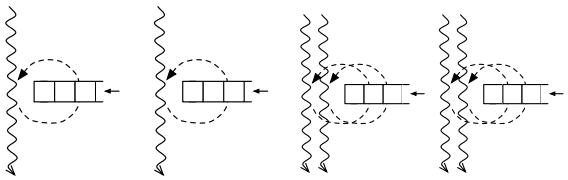
Finally, not every race between events is a harmful race. In this work, we study common Java and Android libraries to identify commutative library operations to effectively filter harmless races. We found 147 new harmful races in 8 popular Android applications such as Firefox, VLCPlayer, etc. We filed bug reports for some of the new harmful races discovered, one of them has been acknowledged by the developers and others are still under investigation.

2. Background

This section provides a brief description of an event-driven programming model, and discusses the challenges in realizing a scalable race detector for this model. While our discussion is centered around the Android model, the problems and solutions are generally applicable to other asynchronous models such as JavaScript and iOS.

2.1 Event-driven Programming Model

An event is a lightweight asynchronous task, which can be generated by the operating system in response to an external input, or by an application's thread (by invoking `send`



(a) Looper framework (b) Binder framework

Figure 2: Event execution models.

in Android). An event is enqueued into an *event queue*. A *looper* thread is associated with each event queue. It is responsible for dequeuing an event in FIFO order and executing it to completion before dequeuing another event. We refer to this event execution model as the *looper framework* (Figure 2). In another model, the *binder framework*, several *binder* threads may be associated with an event queue. Though the binder events are dequeued in FIFO order, they may be concurrently executed in different binder threads. In addition to looper and binder threads that execute events, conventional *worker* threads may also be created by the application.

Most event-driven models use a combination of the above three types of thread models. The looper framework provides a strong ordering guarantee between events. It is widely used in UI frameworks of modern mobile systems such as Android and iOS. These systems often provide the ability to overrule the FIFO order by allowing programmers to assign *priority tags* to events. The binder framework is more efficient for exploiting concurrency in multi-core processors. Android OS uses this framework to handle inter-process communications efficiently.

2.2 Program Trace

A program trace is a sequence of operations listed below, where T is a thread, E is an event, S and S' is either a thread or an event. If an operation is part of an event, it is attributed to that event instead of its looper thread.

- $\texttt{begin}(S), \texttt{end}(S)$: start or end of S.
- $\texttt{rd}(S, x), \texttt{wr}(S, x)$: read or write to data variable x in S.
- $\texttt{fork}(S, T), \texttt{join}(S, T)$: S spawns or waits for T.
- $\texttt{signal}(S, m)$ and $\texttt{wait}(S', m)$ S' waits till S signals through handle m. Callback registration and its execution is also modeled as signal and wait, respectively. We omit S and S' when there is no ambiguity.
- $\texttt{send}(S, q, E)$: S enqueues event E to queue q. We omit S and/or q when there is no ambiguity.

For two operations α and β in a trace, we say $\alpha < \beta$ if α is executed before β. α *happens-before* β ($\alpha \prec \beta$) if there is a programmer-enforced causal order between α and β. We use the term *causally precedes* and happens-before

$$\frac{task(\alpha) = task(\beta) \quad \alpha < \beta}{\alpha \prec \beta}\ (\text{PO}) \qquad \frac{\alpha = \texttt{end}(E_1) \quad \beta = \texttt{begin}(E_2) \quad \texttt{send}(\_, q, E_1) \prec \texttt{send}(\_, q, E_2)}{\alpha \prec \beta}\ (\text{FIFO})$$

$$\frac{\alpha = \texttt{begin}(T) \quad \beta = \texttt{begin}(E) \quad looper(E) = T}{\alpha \prec \beta}\ (\text{LoopBegin}) \qquad \frac{\alpha = \texttt{end}(E) \quad \beta = \texttt{end}(T) \quad looper(E) = T}{\alpha \prec \beta}\ (\text{LoopEnd})$$

$$\frac{\alpha = \texttt{fork}(\_, T) \quad \beta = \texttt{begin}(T)}{\alpha \prec \beta}\ (\text{Fork}) \qquad \frac{\alpha = \texttt{end}(T) \quad \beta = \texttt{join}(\_, T)}{\alpha \prec \beta}\ (\text{Join})$$

$$\frac{\alpha = \texttt{signal}(\_, m) \quad \beta = \texttt{wait}(\_, m) \quad \alpha < \beta}{\alpha \prec \beta}\ (\text{Signal}) \qquad \frac{\alpha = \texttt{send}(\_, \_, E) \quad \beta = \texttt{begin}(E)}{\alpha \prec \beta}\ (\text{Send})$$

Figure 3: Causality rules. $task(\alpha)$ is the worker thread or event that executes operation α. $looper(E)$ is the looper thread that executes event E. _ is a don't-care.

interchangeably. For two events E and E' of the same queue, we shorten $\texttt{end}(E) \prec \texttt{begin}(E')$ as $E \prec E'$.

2.3 Causality Model

Figure 3 summarizes a simplified causality model for an event-driven system with looper threads handling FIFO events (events without priority tags) and worker threads. In Section 5, we will generalize our solutions to handle a wide variety of causality rules, including event atomicity, events with priority tags, and binder threads in Android's causality model [6, 11, 13].

In an asynchronous model, program order (Rule PO) is assumed only for operations within a worker thread or an event, but not between events executed in the same looper thread. This is because the order of events in a looper thread may differ across executions. Rule FIFO is unique to an asynchronous programming model. It says that two events must be causally ordered, if they are of the same queue, and their sends are causally ordered. Inferring this happens-before relation is significantly more challenging than other conventional happens-before relations. We address this problem in Section 3 using AsyncClock.

Rule Send states that an event happens-after its send. Also an event clearly happens-between the beginning and end of its looper thread (Rules LoopBegin, LoopEnd). Rules Fork, Join, and Signal are the same as those in a conventional causality model for multi-threaded programs. We also assume a closed system and thus a total order between input events. Locks induce no causal relation, as their semantics only guarantee mutual exclusion.

2.4 Challenges in Race Detection

A race consists of two memory accesses that access the same location, at least one of them is a write, and there is no causal order between them. However, not all races in the asynchronous model are harmful bugs. Only races that indicate order violations are harmful. Commutative races

are not. Commutative races are common in this model as atomicity is guaranteed between events executed in a thread. We address this issue in Section 6.

To find races in the asynchronous model, we need to address two problems: 1. establish happens-before relations; 2. maintain logical time of operations. We now briefly describe the challenges in addressing these problems.

Establishing Asynchronous Event Causality As shown in Figure 1, the happens-before relation in a synchronous program is defined by program order and pairs of synchronization operations operating on a common handle, and the logical time of immediate causal predecessors (`invokes` and `ends`) of `begin` and `wait` operations could be trivially determined through the handles they operate on. However, in an asynchronous program, determining the immediate causal predecessors of the `begin` operation of an event is harder. It requires identifying which among the past events satisfy the FIFO rule (Figure 3).

EVENTRACER [6] solves this problem by tracking the entire *happens-before graph* of all past synchronization and event operations (`send`, `begin`, `end`) with their logical time. When an event is about to begin (say, `begin`(B) in Figure 1), it traverses the happens-before graph backward from `send`(B) to find causally preceding `send` operations (`send`(A)). The events sent by these operations are the predecessors of B. To speed up the search of predecessors, EVENTRACER uses *graph traversal pruning* to stop the search along certain paths when a `send` is encountered (`send`(A)). However, in the worst case, the entire graph may need to be traversed. We empirically show that this current solution does not scale as the number of events increases with the program trace length.

Maintaining Time *Vector clock* [14] is a succinct data structure to express happens-before relations, and thus a standard way to maintain logical times of all operations in conventional data race detectors. A vector clock is a vector of logical timestamps, where each timestamp identifies the time of the causally preceding operation in a thread. The vector clock of an operation is computed by *inheriting* the vector clocks of its immediate causal predecessors, so for any two operations, one happens-before the other if and only if the vector clock of the former is less than the latter elementwise. However, there are two problems in adapting vector clocks to maintain logical times in asynchronous programs.

The first problem is that, naively adapting vector clocks for asynchronous programs requires dedicating a timestamp in the vector for each event. Unfortunately, there could be an unbounded number (typically dozens of thousands) of events, so this approach clearly does not scale. This problem could be partly alleviated using *chain decomposition* [12, 17], where events are partitioned into *chains*, each consisting of a sequence of causally ordered events. Each chain can be thought of as a logical thread, so only one timestamp in a vector clock is needed to track the logical time in a chain

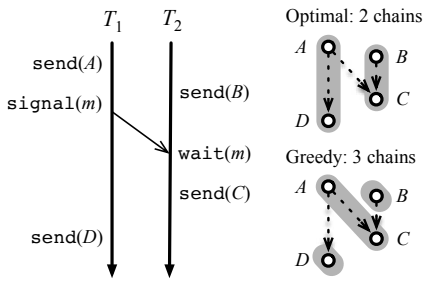


Figure 4: Illustration of chain decomposition. The optimal chain decomposition partitions the events into 2 chains, while an online greedy approach might partition them into 3 chains.

of events. Figure 4 shows an optimal chain decomposition partitioning 4 events into 2 chains, so it is sufficient to use 4 timestamps in a vector clock to track the logical times in all threads and chains. Using vector clocks with chain decomposition, however, requires online partitioning of events upon being processed. Instead of finding the optimal chain decomposition (which requires the knowledge of the complete happens-before graph in prior), past work [17] takes an online greedy approach to assign each event to an existing chain ending with one of its immediate causal predecessors, or to a new chain if there exists no such chain. Since there is no conceptual difference between chains of events an worker threads, both would be referred as chains in the following text.

The second problem is in determining when to free the logical time maintained for an event. Conventional multithreaded programs usually use a limited number of synchronization variables, so conventional data race detectors can keep track of the logical times of all synchronization handles at a low cost. However, there are usually an unbounded number of events in asynchronous programs. Prior work [6] tracks the times of all past events, which clearly does not scale with trace length. We discuss and address challenges in identifying *heirless* events — events that have no immediate successors in future — so that their metadata can be reclaimed to ensure scalability.

3. ASYNCCLOCK Design

In this section, we describe the ASYNCCLOCK algorithm that efficiently establishes happens-before relations between asynchronous events for an *event-driven system having only FIFO events*, whose causality rules are summarized in Figure 3. In Section 5, we will generalize our solutions to handle more types of events and causality rules, such as event atomicity and events with priority tags.

3.1 Overview

To establish happens-before relations between any operations, we could maintain the logical time at each operation

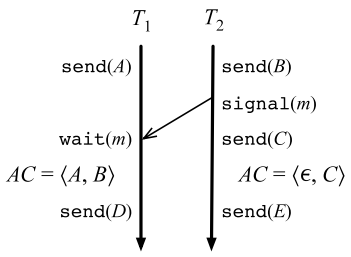


Figure 5: Illustration of AsyncClock. The ACs depict the AsyncClock of T_1 and T_2 after processing their preceding operations. ϵ indicates that there is no predecessor in the corresponding chain.

via vector clocks. The vector clock (VC) at an operation inherits the vector clocks at its *immediate causal predecessors*. According to Rule FIFO in Figure 3, the immediate causal predecessors of an event E must be posted to the same queue by the immediately causally preceding `send` operations of $\texttt{send}(E)$. Therefore, at a `send` operation, we seek to determine the set of events posted by its immediately causally preceding `send`s to the same queue. AsyncClock helps track this information in each chain. The vector clock of an event is then computed at its `begin` operation, where the times of its immediate causal predecessors are guaranteed to have been resolved, by inheriting the end times of the events tracked in the AsyncClock at its `send`.

3.2 AsyncClock Primitive

For any `send` operation, *there is at most one immediately causally preceding* `send` *to the same queue in each chain*. Therefore, an AsyncClock (AC) for a queue can be represented as a vector of events posted by the latest causally preceding `send` to the queue in each chain.

Figure 5 illustrates AsyncClock using an example with a single queue. AC at $\texttt{send}(D)$ consists of A and B as they are posted by the immediately causally preceding `send`s in the two threads. AC at $\texttt{send}(E)$ contains only C, as $\texttt{send}(B)$ does not immediately precedes $\texttt{send}(E)$ in T_2, and there is no causally preceding `send` in T_1.

Formal Definition For an event queue q, the AsyncClock $AC_q : \mathit{Chains} \to \mathit{Events}$ at operation x is formally defined as a vector of events, one for each chain, such that

$$AC_q(i) = \begin{cases} E, & \text{if } \texttt{send}(\_, q, E) \text{ is the latest } \texttt{send} \text{ operation in chain } i \text{ s.t. } \texttt{send}(\_, q, E) \prec x; \\ \epsilon, & \text{otherwise.} \end{cases}$$

Note that for an event E, AC at $\texttt{send}(E)$ may track more than the immediate causal predecessors of E, since not all of the *latest* causally preceding `send`s in every chain are *immediately* causally preceding `send`s of $\texttt{send}(E)$. For example, in Figure 5, if $\texttt{send}(A)$ happens after $\texttt{wait}(m)$ in T_1, then $\texttt{send}(B)$ would no longer be an immediately causally preceding `send` of $\texttt{send}(D)$, but B is still tracked in AC at $\texttt{send}(D)$. In Section 3.3, we will describe a simple optimization to avoid tracking such events.

Resolving Event Time The logical time of an event E of queue q is resolved at $\texttt{begin}(E)$, where all immediate causal predecessors of E are guaranteed to have finished. VC at $\texttt{begin}(E)$ must inherit: 1. VC at $\texttt{send}(E)$, and 2. VC at $\texttt{end}(E')$ for each E' in AC_q at $\texttt{send}(E)$.

3.3 Maintaining AsyncClocks

For each chain, we maintain a AsyncClock for each queue to keep track of the events posted by the immediately causally preceding `send`s in all chains. The following defines the join operation ($\sqcup$) to inherit another AsyncClock, and the identity function ($\mathcal{I}_{AC}$) which constructs an AsyncClock containing only one event. They are used to maintain AsyncClocks at synchronization operations, `send`s and `begin`s. Here, $\mathit{sender}(E)$ denotes the chain that $\texttt{send}(E)$ is in.

$$\begin{aligned} AC \sqcup AC' &= \lambda i.\ \text{if } \texttt{send}(AC(i)) \prec \texttt{send}(AC'(i)) \\ &\quad \text{then } AC'(i) \text{ else } AC(i) \\ \mathcal{I}_{AC}(E) &= \lambda i.\ \text{if } i = \mathit{sender}(E) \text{ then } E \text{ else } \epsilon \end{aligned}$$

Synchronization Operations For each event queue q, AC_q of chain i at a `wait` operation is joined with AC_q at the corresponding `signal` operation. Note that since the join operation performs happens-before queries only between respective `send` operations in the same chain, each query can be implemented as a simple integer comparison between their logical times in that chain. So the join takes only $O(n)$ time, where n is the number of chains.

Event Creation The AsyncClock data structure described above sometimes tracks more than immediate causal predecessors. For example, in Figure 5, any event sent from T_1 after $\texttt{send}(D)$ would have only one immediate causal predecessor, D. So it is redundant and inefficient to keep B in AC of T_1 after $\texttt{send}(D)$. Instead of maintaining the full AsyncClock, our algorithm maintains a "reduced" AC, which becomes $\langle D, \epsilon \rangle$, for T_1 after $\texttt{send}(D)$. More generally, after an event E of queue q is sent from chain i, our algorithm reduces its AC_q to $\mathcal{I}_{AC}(E)$.

Event Begin At the beginning of an event E of queue q, its AsyncClock needs to be computed to track the immediate causal predecessors of any event sent from E. For each queue q', $AC_{q'}$ at $\texttt{begin}(E)$ must inherit: 1. $AC_{q'}$ at $\texttt{send}(E)$, and 2. $AC_{q'}$ at $\texttt{end}(E')$ for each E' in AC_q at $\texttt{send}(E)$. Especially, our algorithm removes all causal predecessors of E from AC_q, as their logical times have been already inherited by VC of E. For a system of n chains and l queues, the computation takes $O(n^2 l)$ time.

3.4 Race Detection

Our race detector maintains a vector clock and a set of AsyncClocks, one for each queue, for each chain, past

event, and synchronization variable, to resolve the logical time of an event. Once the time of an event is resolved, our algorithm uses *greedy chain decomposition* [17] to choose the chain that the event should be part of according to its time. Read and write operations inherit the logical time of their attributed chains. We use the FASTTRACK [10] algorithm to optimize metadata stored for data variables and find races between their accesses.

4. Improving Scalability

The ASYNCCLOCK algorithm described in Section 3 maintains VCs and ACs for all past events, as these metadata might be used to compute the logical times of immediate causal successors of past events in future. However, this algorithm has the same problem as previous work [6]: it does not scale since the memory profile grows with the program execution length.

To address this problem, we present several solutions to identify *heirless events* — events that have no immediate causal successors in future — and reclaim their metadata. We also describe optimizations to make the metadata slim, thus improving memory efficiency of our algorithm.

4.1 Reclaiming Heirless Events

A heirless event cannot have any immediate causal successor in future. That means, its `send` operation cannot be an immediately causally preceding `send` of: 1. any ongoing thread or event; 2. any future event. Since this condition is hard to check, we present two efficient solutions, *reference counting* and *multi-path reduction*, to find a majority of heirless events based on two sufficient but not necessary conditions.

Unfortunately, these solutions do not fully address the problem of memory scalability because: 1. they cannot find all heirless events; 2. there could be unbounded number of events that are not heirless. Figure 6a shows a such example, where $A_2, B_2, C_2, ...$ are not heirless, as a possible future execution (shown in gray) creates immediate causal successors $(A_3, B_3, C_3, ...)$ for each of them.

We propose *time window approximation* to resolve this issue. It exploits the following intuition: *events that are far apart in time are unlikely to be re-ordered in alternative executions*. Concurrency bugs between such events are unlikely to manifest, and hence low priority for developers to fix. We assume happens-before relations between such events to make *old events* heirless and reclaim their metadata, thus ensuring constant memory usage.

Reference Counting If an event is not heirless, it would be in the AC of some chain or future event, whose AC is also computed from the ACs of past events. Hence *events that are not in any AC must be heirless*. So an AC can be implemented as a vector of *reference counting pointers* to the metadata of events. For a newly created event, the reference count of its metadata is 1, as it is in the AC of the sending chain. This count may increment when a containing AC is inherited by another AC, and may decrement when a containing AC inherits another AC, or is reclaimed, or an immediate causal successor is created. When the reference count becomes 0, the event is heirless and its metadata is reclaimed.

Multi-path Reduction Not all heirless events can be reclaimed by reference counting, as can be seen in Figure 6b. In this example, A_1 is in AC_q of B_1 and thus has a positive reference count. However, since B_1 does not send any event, its immediate causal successor can only be sent from either T, A_2, or some causal successor of A_2 in future. This means that $\texttt{send}(A_2)$ causally precedes any future event sent to queue q. Therefore, A_1 becomes heirless at the moment B_1 finishes.

Such heirless events could be identified as follows. If an event E is not heirless, it would be in the AC of a chain (this case is already handled by reference counting), or in the AC of a future event F. In the latter case, E is either in the AC at $\texttt{send}(F)$, or in the AC at the end of some immediate causal predecessor of F (say, F'). Especially, when $\texttt{send}(E) \prec \texttt{send}(F')$ (Figure 6c), E is in the AC of F only if there is no causal successor of E created along *both* paths from $\texttt{send}(F')$ to $\texttt{begin}(F)$, and thus the reference count of E's metadata is at least 2, one for each path. Therefore, for any two events E and F' (not necessarily of the same queue) such that $\texttt{send}(E) \prec \texttt{send}(F')$ and E is in the AC at $\texttt{end}(F')$, if E's reference count is 1, then E is heirless, and thus we can remove E from the AC to relinquish E's metadata.

Time Window Approximation Since an old event and a recent event are unlikely to be executed in reversed order in alternative executions, we assume a happens-before relation between these two events, making the old event heirless. Figure 6d illustrates this idea. For each looper thread, we maintain a sliding time window of recently finished events, up to a predefined time threshold t, and a *time window clock* (TC). Happens-before relations between events in the time window are precisely maintained. When an event is moved out from the time window (E_2) it becomes *old*, so TC is updated to TC' by inheriting the end time and AC of the old event. The newly started event (E_{n+1}) then inherits TC'. Since TC' is a causal successor of every old event and a causal predecessor of every new event, an old event no longer has any immediate causal successor in future and thus becomes heirless.

To actively reclaim the metadata of old events, the reference counting pointers to the metadata are implemented with an *invalidate* operation: when an event becomes old, we invalidate an arbitrary pointer to its metadata, so that the metadata is immediately relinquished, and all other pointers to the same metadata become null pointers.

Time window approximation ensures memory scalability in twofold. First, it limits the number of events tracked by the algorithm: if a looper thread processes events with rate

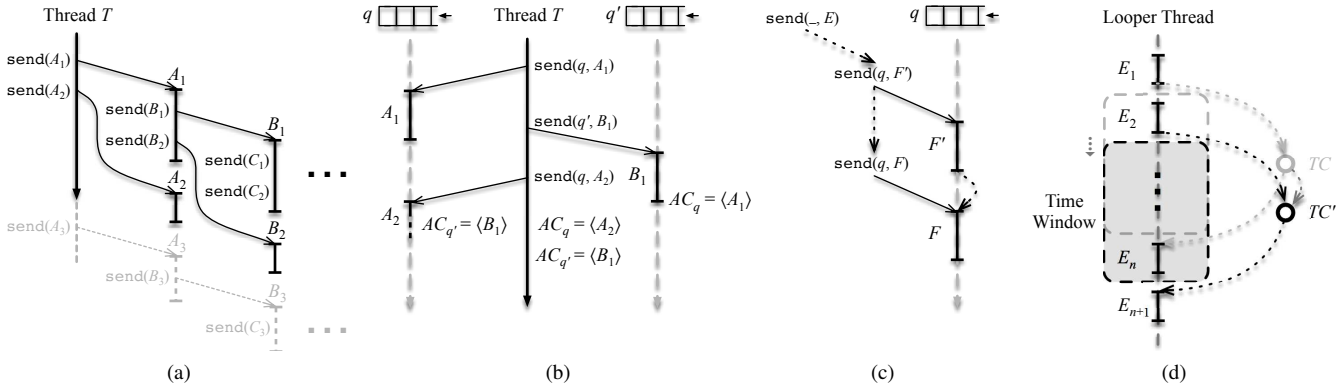


Figure 6: (a) Example of infinitely many non-heirless events. The executed trace is shown in black, and a possible future execution is shown in gray. (b) A heirless event (A_1) with positive reference count. Only nonempty ACs are shown. (c) Illustration of multi-path reduction. (d) Illustration of time window approximation.

r, at most $(rt + 1)$ events are tracked for this looper thread. Second, the number of chains required to place all events in the looper thread (and hence the size of VCs and ACs) is also bounded by $(rt + 1)$.

4.2 Reducing AsyncClock Size

Another important factor that affects both time and space efficiency of the AsyncClock algorithm is the size of AsyncClocks. In the following, we present optimizations to reduce the size of the metadata.

Sparse Vectors The number of chains, which affects the dimension of VC and AC, is unbounded, as the example in Figure 6a also shows that. However, as described in Section 3.3, the size of an AC of a chain is reset to 1, once the chain sent a new event to the corresponding queue. Therefore, *most ACs are rather sparse*. So we implemented a sparse vector representation [7] with hash tables for ACs to significantly reduce the space, and also the time of join operations.

Garbage Collection Time window approximation and removal of events (discussed in Section 5) leave residual null pointers in the AsyncClocks and thus reduce their sparsity. Therefore, we perform a *garbage collection* process periodically to scan through every AsyncClock maintained in the system and remove all null pointers.

FIFO Chain Decomposition Different chain decompositions may affect both the total number of chains and the sparsity of AsyncClocks, and thus time and memory performance. We propose a new chain decomposition method, *FIFO chain decomposition*, that improves AsyncClock's performance. It is based on the following two observations: 1. our empirical study shows that many events in mobile applications are FIFO events that are either children or grandchildren of worker threads or input events; 2. All FIFO events sent from a chain to an event queue are sequentially ordered.

The chain decomposition works as follows. For an event queue, all FIFO events sent from a worker thread are placed in a *level-1 FIFO chain*. All input events also form a level-1 FIFO chain. FIFO events sent from a level-1 FIFO chain are placed in a *level-2 FIFO chain*, and so on. Among all events, about 54% are level-1 FIFO events, 4.8% are level-2 FIFO events, and 1.7% are level-3 FIFO events. Since there are very few FIFO events beyond level 3, we fall back to greedy chain decomposition for other FIFO and non-FIFO events.

5. Generalizing AsyncClock

In this section, we start with the extended causality model for Android, which includes event atomicity and event with *priority tags* that overrule the FIFO ordering. Then, we formulate a generalized form of the extended causality rules and present a generalization of AsyncClock that can handle *any* causality rule that follows this generalized form. In the end, we describe realizations of AsyncClock to handle Android's causality model and other features such as event removal and binder events.

5.1 Extended Causality Rules

Figure 7 and Table 1 summarize Android's causality rules for event atomicity and events with priority tags. We made a few revisions on the causality rules in [6, 11, 13].

Rule Atomic states that two events in a looper thread are atomic with respect to each other, and thus are causally ordered by the synchronization operations they perform. Here, we made a revision shown in Figure 8a: instead of claiming $E_1 \prec E_2$, we only order E_1 with the part *after* `wait`(m) in E_2, since the synchronization operations provide no ordering semantics before `wait`(m).

Rule Priority summarizes various ways that Android provides to overrule the FIFO ordering. An event can have

E_1 \ E_2	Delayed, Async	Delayed, Sync	AtTime, Async	AtTime, Sync	AtFront, Async	AtFront, Sync
Delayed, Async	$E_1.\texttt{time} \leq E_2.\texttt{time}$	$E_1.\texttt{time} \leq E_2.\texttt{time}$	false	false	false	false
Delayed, Sync	false	$E_1.\texttt{time} \leq E_2.\texttt{time}$	false	false	false	false
AtTime, Async	false	false	$E_1.\texttt{time} \leq E_2.\texttt{time}$	$E_1.\texttt{time} \leq E_2.\texttt{time}$	false	false
AtTime, Sync	false	false	false	$E_1.\texttt{time} \leq E_2.\texttt{time}$	false	false
AtFront, Async	true	true	true	true	false	false
AtFront, Sync	false	true	false	true	false	false

Table 1: The priority function for Rule PRIORITY. $E_1 \prec E_2$ if $\texttt{send}(E_1) \prec \texttt{send}(E_2)$ and the corresponding cell is true.

$$\frac{\begin{array}{c}\alpha = \texttt{end}(E_1) \\ \texttt{begin}(E_1) \prec \beta \\ \exists E_2.\ E_2 \in Events \wedge E_2 = task(\beta) \wedge \\ looper(E_2) = looper(E_1)\end{array}}{\alpha \prec \beta}\ (\textsc{Atomic})$$

$$\frac{\begin{array}{c}\alpha = \texttt{end}(E_1) \\ \beta = \texttt{begin}(E_2) \\ \texttt{send}(\_, q, E_1) \prec \texttt{send}(\_, q, E_2) \\ priority(E_1, E_2)\end{array}}{\alpha \prec \beta}\ (\textsc{Priority})$$

$$\frac{\begin{array}{c}\alpha = \texttt{end}(E_1) \\ \beta = \texttt{begin}(E_2) \\ \texttt{send}(\_, q, E_2) \prec \texttt{send}(\_, q, E_1, \texttt{AtFront}) \prec \beta\end{array}}{\alpha \prec \beta}\ (\textsc{AtFront})$$

Figure 7: Extended causality rules for Android. *priority* is the priority function defined in Table 1.

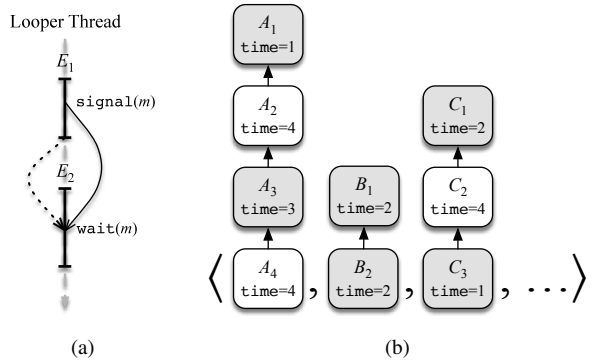


Figure 8: (a) Illustration of Rule ATOMIC. (b) Illustration async-before lists for `Delayed` events. The causal predecessors of an event with $\texttt{time} = 3$ are shown in gray.

the following priority tags to change its queuing policy: 1. being dequeued after a certain delay (`Delayed`); 2. being dequeued after a specific time (`AtTime`); 3. being enqueued to the front of the queue (`AtFront`); 4. having an *asynchronous message* [1] (`Async`) to precede ordinary events (`Sync`) when a *barrier message* is placed in the queue. Table 1 lists all combinations of priority tags an event can have. Events with the same priority tags and non decreasing time constraints are dequeued in FIFO order, and FIFO events are considered as `Delayed` events with zero delay when inferring their causal relations with non-FIFO events. Here, we also made a revision over prior work [6], to infer that two `AtTime` events are ordered by Android's model if one is sent causally earlier than the other with a smaller time constraint.

Rule ATFRONT says that an `AtFront` event happens-before another event, if that other event is guaranteed to be in the queue when this event is enqueued. It is the only rule showing that *a causally later* `send` *could create a causal predecessor of an event sent earlier*.

5.2 Generalized Asynchronous Causality Rule

Rules FIFO, ATOMIC and PRIORITY all follow the following form of *generalized asynchronous causality rule*:

$$\frac{\alpha \in \Lambda \quad \gamma(\alpha) \prec \eta(\beta) \quad \rho(\alpha, \beta)}{\alpha \prec \beta}$$

defined on the set of operations Op in a trace with parameters $\Lambda \subseteq Op, \gamma : \Lambda \to Op, \eta : Op \to Op, \rho : Op \times Op \to \{\text{true}, \text{false}\}$ such that $\forall \alpha \in \Lambda.\, \gamma(\alpha) \preceq \eta(\alpha)$ and $\forall \beta \in Op.\, \eta(\beta) \preceq \beta$. For example, for Rule PRIORITY, Λ is the set of `end` operations of all events of a queue, γ, η return the `send` operation of the containing event, and ρ is the priority function; for Rule ATOMIC, Λ is the set of event `end`s in a looper thread, γ returns the `begin` operation of the containing event, η is the identity function, and ρ checks if both operations are in the same looper thread.

For any causality rule following the above generalized form, we can generalize ASYNCCLOCK to track the immediately causally preceding γ-operations of $\eta(\beta)$ to find the immediate causal predecessors of β as follows: at operation x, the generalized ASYNCCLOCK $AC_\Lambda : Chains \to \Lambda$ is a vector over Λ, one for each chain, such that

$$AC_\Lambda(i) = \begin{cases} \alpha, & \text{if } \gamma(\alpha) \text{ is in chain } i \text{ s.t. } \gamma(\alpha) \prec x \text{ and} \\ & \gamma' \prec \gamma(\alpha) \text{ for all } \gamma' \in \gamma(\Lambda) \text{ in chain } i; \\ \epsilon, & \text{otherwise.} \end{cases}$$

If ρ is a function satisfying that: 1. ρ is commutative and transitive, and 2. $\forall \alpha, \alpha' \in \Lambda.\, \gamma(\alpha) \prec \gamma(\alpha') \implies \rho(\alpha, \alpha')$, then one can show that for any operations α, α' and β, if $\gamma(\alpha) \prec \gamma(\alpha')$, then either both of α, α' or none of them are β's causal predecessors. So one can easily generalize the algorithm described in Section 3 to compute the logical time of β. Moreover, since $\gamma(\alpha), \eta(\beta), \alpha$ and β form a "diamond-structure" similar to the one in Figure 6c, multi-path reduction can also be generalized.

If ρ does not satisfy the above properties, we can still use AC_Λ with a helper data structure, *async-before lists*, to find

immediate causal predecessors. We illustrate async-before lists with an example for `Delayed` events shown in Figure 8b. We store all `Delayed` events sent from each chain in a list ordered by their `sends` (*e.g.*, events A_i form a list, and B_i form another). To find the immediate causal predecessors of an event E with $\texttt{time} = 3$, we traverse each list, starting from the events in the AC at $\texttt{send}(E)$ $(A_4, B_2, C_3, ...)$, to visit all events whose `sends` happens-before $\texttt{send}(E)$. The set of visited events satisfying the priority function (shown in gray) would contain all immediate causal predecessors of E.

5.3 Handling Extended Causality Model

Here we describe how to use generalized AsyncClocks to handle each extended causality rule. We use the rule evaluation order described in [6] to obtain the minimal closure of all causality rules. We also describes solutions for Rule AtFront, event removal, and binder threads.

Event Atomicity For an operation β in an event, we use generalized AsyncClocks to track its *immediately causally preceding* `begin` operations, thus their corresponding `end` operations are the immediate causal predecessors of β. Then, the logical time of β is resolved by inheriting the time of these `ends`.

Events with Priority Tags For each tag combination in Table 1, we use generalized AsyncClocks to track *immediately causally preceding* `send` operations along with async-before lists to find immediate causal predecessors. A `Sync` event should inherit the time from both its `Sync` and `Async` predecessors. For an `Delayed` or `AtTime` event E, we can speed up the search of predecessors by early-stopping the traversal of each list after visiting some event E' in the list such that 1. $E'.\texttt{time} = E.\texttt{time}$, or 2. $E''.\texttt{time} \leq E'.\texttt{time}$ for all preceding events E'' in the list. For example, in Figure 8b, we can prune A_1 (Case 1) and B_1 (Case 2), since they happen-before A_3 and B_2 respectively and thus not immediate causal predecessors of the new event. The search of `AtFront` predecessors can also be avoided by caching the join of the logical times of all `AtFront` events sent along a chain.

Sent-at-Front Events Rule AtFront does not follow the form of generalized asynchronous causality rule, and thus cannot be addressed with AsyncClock. Instead, we maintain a *sent-at-front list* for each event in a queue. When an `AtFront` event is executed, it is added to all sent-at-front lists. When an event is dequeued, we iterate through its sent-at-front list to find its causal predecessors.

Event Removal An event removed explicitly by the programmer cannot be relinquished, since it might have an immediate causal successor. In such case, we first resolve the time of this removed event, and use it to resolve the time of the successor. Removed events are reclaimed through reference counting, or along with garbage collection when all of its predecessors are old.

Binder Events We apply the speculative happens-before rules in [6] to order binder events, and use a separated pool of chains for greedy chain decomposition.

6. Race Detection

All data races reported by AsyncClock are true races, meaning that the accesses in the races are not ordered by the causality model and might be re-ordered in alternative executions. In conventional multithreaded models, data races always happen between different threads and are disallowed in a DRF0 memory model [5]. However, data races defined by an causality model for event-driven systems could happen between different events executed in the same looper thread. These races are either *harmful races* that lead to *order violation* bugs, or *harmless races* that produce "correct" results irrespective of their execution order and do not lead to any concurrency bug. Android applications usually incur many harmless races, which significantly reduce the usability of a race detector.

However, without a formal correctness specification for any Android program, we have to rely on manual reasoning of a race and its events to determine if it is harmful or harmless, which is a common approach used in similar tools such as EventRacer [6]. We manually investigated the races reported by our tool, and observed that the following races are usually harmless: 1. races in Android's framework that are not induced by the user code; 2. races in certain OS-generated events that are *commutative* with other events (*e.g.*, screen refresh events). 3. race in libraries that provides *commutative operations* (*e.g.*, increments to `size` when adding two items into a list). Therefore, our tool only reports races between *user-induced* accesses (accesses in user code, or in libraries called by user code), and uses a *commutativity filter* that whitelists pairs of commutative accesses and events described above to remove races that are highly likely to be harmless. We built the whitelist conservatively such that the filter did not remove harmful races in our experience. Our whitelist currently contains around 400 entries that mark the code in the Android framework, OS, and libraries that are likely to be commutative, but not in the user code. So the manual effort required to create this whitelist is not necessary for every application.

To further improve usability, our tool reports races in *race groups*, similar to EventRacer: races that are induced by the same library invocations in the user code are grouped together. This reduce a fair amount of work for manual investigations on race reports.

7. Evaluation

In this section, we evaluate the performance AsyncClock and the effectiveness of the proposed solutions.

7.1 Experimental Setup

We used a Google Nexus 4 device with an instrumented Dalvik runtime of Android OS v4.3 to record the execution traces, then offloaded the traces to a machine equipped with 2.67G Intel Xeon X5650 CPU and 48GB of memory for race detection. We evaluated ASYNCCLOCK on 20 popular Android applications picked from [3, 4, 6, 11, 13]. We used *Android Monkey* [2] to generate traces that run in 10–30 minutes on an uninstrumented system (longer traces were collected for applications that generated fewer events per minute.)

7.2 Overall Performance

Trace Collection The statistics of collected traces and the overheads of trace recording for all 20 applications are shown in Table 2. The overhead is primarily incurred by recording all read and write accesses to heap objects. On average, instrumented runs were about 5x slower than uninstrumented runs with respect to CPU time.

Offline Analysis The *analysis* columns in Table 2 show the time and memory of our race detector based on ASYNCCLOCK, with a 2-minute time window and FIFO chain decomposition. Note that we run the offline analyses on a local machine. The analysis overhead is computed as the analysis time (on local machine) normalized by the application CPU time (on phone), indicating how much slowdown a user would experience to use our tool to analyze an application. All analyses finished in 9 minutes, and averaged 3 minutes and around 700MB of memory.

7.3 Comparison with EVENTRACER

EVENTRACER [6] is the state-of-the-art race detector for Android applications. Since EVENTRACER for Android [4] is close-sourced, we could not use its publicly available binary executable for various reasons. First, we were unable to adapt our revisions to the causality model (Section 5.1) to EVENTRACER's implementation. Second, we wanted to compare ASYNCCLOCK and EVENTRACER for the exact same trace, and this was not possible without access to their source code. Also, we couldn't get to run or obtain long enough traces for several applications used in our analysis. Therefore, we re-implemented its happens-before graph construction algorithm in our framework, including graph traversal pruning and other optimizations presented in [6]. This enabled us to do an end-to-end comparison between the two algorithm with the same trace and the same framework overheads.

The rightmost two columns compare the performance in time and memory between our tool and EVENTRACER. Since EVENTRACER scales super-linearly (Figure 9a), its overhead is not fixed, but grows with the number of events, so our tool achieved greater speedup when more events were analyzed. For the collected traces, our tool was at least twice as fast as EVENTRACER, and averaged 8x faster. The memory performance was also significant: with heirless events reclaimed and a 2-minute time window, our tool used only 0.7GB of memory on average, where EVENTRACER used 5.5GB.

We further studied the scalability of EVENTRACER and ASYNCCLOCK in Figure 9a. As can be seen, EVENTRACER scaled super-linearly in time. We observed that its graph traversal pruning did not handle the scenario in Figure 9b well because: 1. it nearly pruned nothing for `AtTime` events since their `time`s are usually different; 2. the entire parent chain of B_3 (*i.e.*, I_1, I_2, I_3) in Figure 9b was traversed to find its predecessors, which is proportional to the trace length. In contrast, the time spent in finding predecessors of an event in our tool remained low due to the sparsity of ASYNCCLOCKs and early-stopping in the async-before lists of `Delayed`, `AtTime`, and `AtFront` events. EVENTRACER also does not scale in memory with number of events because it maintains the whole happens-before graph, where our tool shows better memory scalability due to storing only ASYNCCLOCKs for events and aggressively reclaiming events with our scalability optimizations described in Section 4.

Since both EVENTRACER's graph traversal algorithm and ASYNCCLOCK (with no time window) are sound, the races reported by both methods are the same under our causality model. However, both methods use an additional step that employs different heuristics to classify the found races into harmless and harmful races, which is orthogonal to finding races under a causality model and thus not part of the performance analysis.

7.4 Effectiveness of Reclaiming Events

Figure 9a also shows how effective the event reclaiming optimizations are in 5 applications. As can be seen, ASYNCCLOCK had good time performance even without any scalability optimization described in Section 4, but the memory usage grew with the number of events being processed. The two sound optimizations for reclaiming heirless events (reference counting and multi-path reduction) reduced half of the memory usage for Firefox and VLCPlayer. But for AardDict and ConnectBot, there were too many non-heirless concurrent events, which made the number of chains and metadata kept in memory increase rapidly. These two applications showed the need of time window approximation. With a 2-minute time window, our tool scaled reasonably well in memory.

7.5 Recall of Time Window Approximation

We evaluated how many false negatives were introduced by assuming happens-before relations between old and recent events in time window approximation. Figure 10 shows the trade-off between its recall and resource usage with different window sizes for 8 applications (listed in Table 3). As can be seen, only 4% of the races were missing when we used a 2-minutes or larger window, and all of them are manually

Application	CPU Time	Operations		Threads			Events		Trace Overhead	Analysis		vs. EventRacer	
		Sync	Mem	Looper	Binder	Other	Looper	Binder		Overhead	Mem	Speedup	Mem Saved
AnyMemo	145s	760k	67M	24	5	158	244584	1110	5.49x	2.41x	2095M	28.33x	81%
ConnectBot	279s	722k	166M	3	6	60	86056	4819	4.58x	0.69x	474M	17.03x	97%
Firefox	1448s	720k	195M	7	4	269	78719	2673	2.31x	0.16x	248M	10.38x	97%
NPRNews	283s	556k	198M	8	5	87	77619	50011	5.48x	0.79x	672M	9.38x	85%
K9Mail	369s	507k	164M	6	5	46	48493	8136	2.95x	0.44x	785M	5.85x	87%
OpenSudoku	216s	651k	121M	1	4	20	47062	2810	4.39x	0.55x	478M	5.17x	87%
SGTPuzzles	188s	702k	115M	3	5	65	42110	1938	4.95x	0.74x	592M	5.49x	90%
AardDict	148s	382k	80M	3	4	117	37345	4331	4.10x	1.04x	963M	4.78x	82%
BarcodeScanner	114s	208k	127M	2	3	17	34792	949	7.61x	1.24x	197M	5.49x	98%
FlymNews	251s	338k	122M	4	6	151	31690	1579	4.21x	0.81x	982M	4.08x	81%
RemindMe	157s	280k	56M	10	6	57	31637	1391	2.83x	0.34x	310M	2.49x	87%
AdobeReader	216s	418k	75M	14	4	307	31301	1751	3.47x	0.61x	831M	3.95x	79%
FlipKart	387s	811k	242M	34	4	233	31054	1264	10.78x	0.92x	1438M	2.92x	71%
OIFileManager	205s	593k	124M	76	5	227	30841	6694	5.05x	0.84x	723M	2.70x	76%
VLCPlayer	170s	1045k	90M	31	25	360	26241	28133	5.34x	0.98x	616M	3.44x	87%
ASQLiteManager	78s	421k	64M	1	4	34	25597	1529	6.27x	0.87x	455M	4.43x	88%
Twitter	273s	355k	159M	128	9	149	24333	2615	8.34x	0.61x	1134M	2.92x	70%
Tomdroid	143s	465k	70M	2	6	104	22121	3441	4.01x	0.72x	429M	2.21x	82%
FBReader	223s	632k	107M	14	5	56	21300	4064	3.85x	0.61x	548M	2.32x	73%
ATimeTracker	112s	291k	52M	1	6	22	19620	1880	3.35x	0.57x	462M	2.48x	67%
Average	270s	543k	120M	–			49626	6556	4.97x	0.80x	722M	7.99x	87%

Table 2: Summary of trace collection and analysis sorted by looper events. The analysis columns show the performance of AsyncClock with a 2-minute time window and FIFO chain decomposition. All analyses are run offline on a local machine.

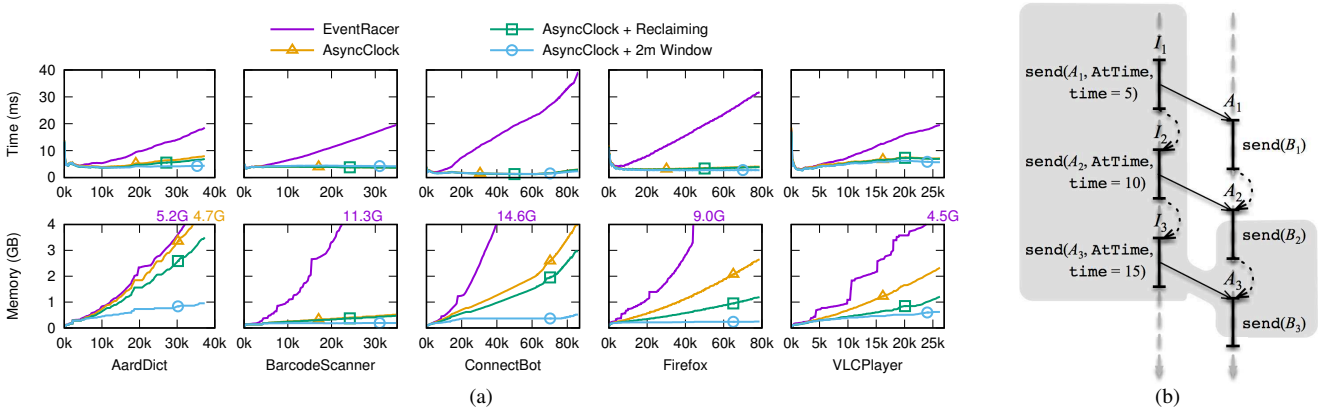


Figure 9: (a) Comparison between EventRacer and AsyncClock. The x-axes show the number of looper events. The top row shows the average time spent *per event*. The bottom row shows total memory used during the analyses. The results of 3 configurations of AsyncClock are shown: no event reclaiming (△), reclaiming heirless events (□), and reclaiming events with a 2-minute time window (○). (b) An event pattern in BarcodeScanner, where I_n are input events. EventRacer traversed the shaded area to find predecessors of B_3.

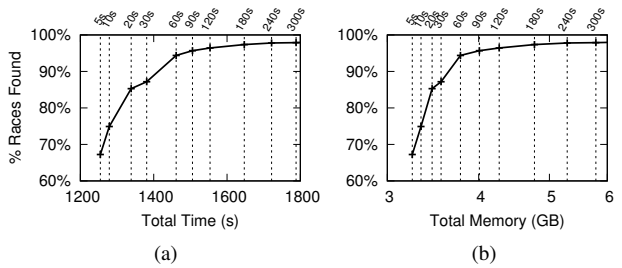


Figure 10: The percentages of races reported from 8 selected applications versus the total time and memory used for various time windows. The memory axis is in log scale.

identified as harmless. Therefore, we used a 2-minute time window in all other experiments.

7.6 FIFO Chain Decomposition

We compared FIFO chain decomposition with a naive greedy approach [17]. Our method achieved 5% improvement in memory, and 10% in time (due to finding chains with table-lookups for FIFO events).

7.7 Reported Races

Our tool found 1437 user-induced race groups in 8 selected applications. All of them are true races under the causality model, but many are harmless races and removed by our commutativity filter. The results are shown in Table 3. As can be seen, 1106 race groups are removed by the filter (Row *Fil-*

Application		Aard Dict	Barcode Scanner	Connect Bot	FB Reader	Firefox	OIFile Manager	Tom droid	VLC Player
All Races Groups		184	33	218	190	282	74	79	377
Filtered		99	25	175	164	217	51	65	310
Harmful		67	5	22	9	28	6	2	8
Harmless	Type I	6	0	8	0	10	2	1	7
	Type II	4	1	8	3	20	1	4	30
	Other	8	2	5	14	7	14	7	22

Table 3: The user-induced race groups reported in 8 applications. The *filtered* row shows how many race groups were removed by our filter. The *harmful* and *harmless* columns are the actual number of race groups reported by our tool.

tered), and of the remaining race groups (Rows *Harmful* and *Harmless*), 44% of them are harmful. Compared to EVENTRACER [6], our tool reported more harmless races. We speculate the following reason: EVENTRACER reported only *uncovered races* and assumed that those races are potentially *custom synchronization operations* that order *covered races*. Since this heuristic does not guarantee soundness and could lead to false negatives, we decided not to apply this heuristics but only employed a conservative commutativity filter. Therefore, our tool reported more true and false positives.

Harmful Races After manual inspection, we found 147 harmful races from the 8 applications listed in Table 3, including one that was confirmed by the developers of Firefox. Here are some harmful races we discovered:

- Firefox uses the main *UI thread* to process user input, and another *compositor thread* to render web pages. While rendering a page, the compositor thread would read the locale settings, which might be been updating by a UI event. So the browser might render locale-specific contents incorrectly till the next UI refresh.
- BarcodeScanner initializes the `CameraManager` in the `onResume` event. The `CameraManager` is then used in the `surfaceCreated` event, which usually comes after `onResume`, but the order is not guaranteed by the Android system, and it might use a wrong `CameraManager` object, which is not cleaned up correctly in a previous `onPause` event, to initialize the camera.
- VLCPlayer switches from the audio player mode to the video player mode when the next item in the playlist is a video without checking if the next item has been nullified because of loading a new playlist, which would lead to a `NullPointerException`.

Harmless Races Although the problem of identifying harmless races is orthogonal to this work, we manually investigated the types of harmless races to understand the causes to give us some insights to develop better heuristics in the future. In our experience, over 50% of the harmless fall in the following two categories:

Type I (*Delayed update*): Many races are between an event that modified the UI components or internal data structures in reaction to user input, and an event independently generated by Android to update the UI. The changes made in the earlier events would not be observed until the later event executed.

Type II (*Control-dependent races*): Since events are executed atomically with respected to each other, programmers usually write to a flag variable in an earlier event to change the execution of a later event. Some of the races in this categories might be able to remove by the *If-Guard check* proposed in CAFA [11].

8. Related Work

Prior solutions for finding races in asynchronous programs [6, 11, 13, 16, 17] did not address the scalability challenges that we address in this paper. EventRacer [6, 17] is the most closely related work to which we provided a detailed comparison. Dimitrov [8] introduced the concept of commutativity races, and developed a framework to systematically describe high-level operation commutativity for library functions. We adapted their idea and came up with a simplified commutativity spec that helped us to remove races between events in the same thread. CAFA [11] proposed several heuristics to distinguish harmful *use-after-free* violations from harmless one. However, the scope of checking object *use-after-free* is too narrow. Our work is able to report all read and write races, with a commutativity spec for Android and Java libraries to remove some benign ones.

There have been many studies on detecting races in multithreaded programs [9, 10, 15, 18]. These techniques only infer synchronous happens-before relations. Also these techniques suffer from the scalability issues when dealing with large number of threads.

9. Conclusion

Asynchronous programming models are becoming increasingly common. Yet, efficient tools for finding concurrency errors in asynchronous programs have been lacking. In this paper, we presented a new primitive, ASYNCCLOCK, to infer happens-before relations between asynchronous events for a wide variety of causality models for event-driven systems. We also addressed an important scalability problem: reclaiming heirless events. We built the first single-pass, non-graph-based Android data race detector, and show that our algorithm is scalable in time and space. We used it to find 147 previously unknown harmful races in popular Android applications.

Acknowledgments

We are most grateful to Prof. Peter Chen and Prof. Jason Flinn for helpful advice. We also thank the anonymous reviewers for comments that improved this paper. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. It was also supported by NSF CAREER 1149773, NSF CCF 1527301, and Intel, Inc.

References

[1] Message — android developers. https://developer.android.com/reference/android/os/Message.html#setAsynchronous(boolean). [Online; accessed 2016-08-15].

[2] Ui/application exerciser monkey — android studio. https://developer.android.com/studio/test/monkey.html, . [Online; accessed 2016-08-15].

[3] droidracer - software engineering and analysis lab (seal), iisc bangalore. http://www.iisc-seal.net/droidracer, . [Online; accessed 2016-08-15].

[4] Event racer for android. http://eventracer.org/android/, . [Online; accessed 2016-08-15].

[5] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. *SIGARCH Comput. Archit. News*, 19(3):234–243, Apr. 1991. ISSN 0163-5964. doi: 10.1145/115953.115976. URL http://doi.acm.org/10.1145/115953.115976.

[6] P. Bielik, V. Raychev, and M. Vechev. 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, pages 332–348, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814303. URL http://doi.acm.org/10.1145/2814270.2814303.

[7] M. Christiaens and K. Bosschere. *Euro-Par 2001 Parallel Processing: 7th International Euro-Par Conference Manchester, UK, August 28–31, 2001 Proceedings*, chapter Accordion Clocks: Logical Clocks for Data Race Detection, pages 494–503. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-44681-1. doi: 10.1007/3-540-44681-8_73. URL http://dx.doi.org/10.1007/3-540-44681-8_73.

[8] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 305–315, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594322. URL http://doi.acm.org/10.1145/2594291.2594322.

[9] D. R. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, pages 237–252, 2003.

[10] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[11] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594330. URL http://doi.acm.org/10.1145/2594291.2594330.

[12] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, Dec. 1990. ISSN 0362-5915. doi: 10.1145/99935.99944. URL http://doi.acm.org/10.1145/99935.99944.

[13] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594311. URL http://doi.acm.org/10.1145/2594291.2594311.

[14] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[15] R. H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[16] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *PLDI*, pages 251–262, 2012.

[17] V. Raychev, M. T. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *OOPSLA*, pages 151–166, 2013.

[18] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/SIGSOFT FSE*, pages 205–214, 2007.