Tailoring Stateless Model Checking for Event-Driven Multi-Threaded Programs

Parosh Aziz Abdulla[1], Mohammed Faouzi Atig[1], Frederik Meyer Bønneland[2],
Sarbojit Das[1], Bengt Jonsson[1], Magnus Lång[1], and Konstantinos Sagonas[1,3]

[1] Uppsala University, Uppsala, Sweden
[2] Aalborg University, Denmark
[3] NTUA, Grece

Abstract. Event-driven multi-threaded programming is an important idiom for structuring concurrent computations. Stateless Model Checking (SMC) is an effective verification technique for multi-threaded programs, especially when coupled with Dynamic Partial Order Reduction (DPOR). Existing SMC techniques are often ineffective in handling event-driven programs, since they will typically explore all possible orderings of event processing, even when events do not conflict. We present Event-DPOR, a DPOR algorithm tailored to event-driven multi-threaded programs. It is based on Optimal-DPOR, an optimal DPOR algorithm for multi-threaded programs; we show how it can be extended for event-driven programs. We prove correctness of Event-DPOR for all programs, and optimality for a large subclass. One complication is that an operation in Event-DPOR, which checks for redundancy of new executions, is NP-hard, as we show in this paper; we address this by a sequence of inexpensive (but incomplete) tests which check for redundancy efficiently. Our implementation and experimental evaluation show that, in comparison with other tools in which handler threads are simulated using locks, Event-DPOR can be exponentially faster than other state-of-the-art DPOR algorithms on a variety of programs and

1 Introduction

Event-driven multi-threaded programming is an important idiom for structuring concurrent computations in distributed message-passing applications, file systems [31], high-performance servers [10], systems programming [11], smartphone applications [33], and many other domains. In this idiom, multiple threads execute concurrently and can communicate through shared objects. In addition, some threads, called *handler threads*, have an associated event pool to which all threads can post events. Each handler thread executes an event processing loop in which events from its pool are processed sequentially, one after the other, interleaved with the execution of other threads. An event is processed by invoking an appropriate handler, which can be, e.g., a callback function.

Testing and verification of event-driven multi-threaded programming faces all the usual challenges of testing and verification for multi-threaded programs, and

furthermore suffers from additional complexity, since the order of event execution is determined dynamically and non-deterministically. A successful and fully automatic technique for finding concurrency bugs in multithreaded programs (i.e., defects that arise only under some thread schedulings) and for verifying their absence is *stateless model checking* (SMC) [15]. Given a terminating program and fixed input data, SMC systematically explores the set of all thread schedulings that are possible during program runs. A special runtime scheduler drives the SMC exploration by making decisions on scheduling whenever such choices may affect the interaction between threads. SMC has been implemented in many tools (e.g., VeriSoft [16], CHESS [34], Concuerror [9], NIDHUGG [2], rInspect [42], CDSCHECKER [35], RCMC [22], and GENMC [26]), and successfully applied to realistic programs (e.g., [17] and [25]). To reduce the number of explored executions, SMC tools typically employ *dynamic partial order reduction* (DPOR) [12,1]. DPOR defines an equivalence relation on executions, which preserves relevant correctness properties, such as reachability of local states and assertion violations, and explores at least one execution in each equivalence class.

Existing DPOR techniques for multi-threaded programs lack effectiveness in handling the complications brought by event-driven programming, as has been observed by e.g., Jensen et al. [20] and Maiya et al. [28]. A naïve way to handle such a program is to consider all pairs of events as conflicting, implying that different orderings of event executions by a handler thread will be considered inequivalent. A major drawback is then that a DPOR algorithm cannot exploit the fact that different orderings of event executions by a single handler thread can be considered equivalent in the case that events are non-conflicting. In this way, a program in which n non-conflicting events are posted to a handler thread by n concurrent threads can give rise to $n!$ explorations by a standard DPOR algorithm, whereas all of them are in fact equivalent. On the other hand, some events may be conflicting, so a DPOR algorithm for event-driven programs should explore only the necessary inequivalent orderings between conflicting events. This can be achieved by defining an equivalence on executions, which respects only the ordering of conflicting accesses to shared variables, irrespective of the order in which events are executed. For plain multi-threaded programs, this equivalence is the basis for several effective DPOR algorithms [12,1]. The challenge is to develop an effective DPOR algorithm also for event-driven programs.

In this paper, we present Event-DPOR, a DPOR algorithm for event-driven multi-threaded programs where handlers can execute events from their event pool in arbitrary order (i.e., the event pool is viewed as a multiset). The multiset semantics is used in many works [21,37,20], often with the significant restriction that there is only one handler thread; we consider the more general situation with an arbitrary number of handler threads. Event-DPOR is based on Optimal-DPOR [1,3], a DPOR algorithm for multi-threaded programs. The basic working mode of Optimal-DPOR is similar to several other DPOR algorithms: Given a terminating program, one of its executions is explored and then analyzed to construct initial fragments of new executions; each fragment that is not redundant (i.e., which can be extended to an execution that is not equivalent

to a previously explored execution), is subsequently extended to a maximal execution, which is analyzed to construct initial fragments of new executions, and so on. Event-DPOR employs the same basic mode of operation as Optimal-DPOR, but must be extended to cope with the event-driven execution model. One complication is that the constructed initial fragments must satisfy the constraints imposed by the fact that event executions on a handler are serialized; this may necessitate reordering of several events when constructing new executions from an already explored one. Another complication is that the check whether a new fragment is redundant is NP-hard in the event-driven setting, as we prove in this paper. We alleviate this by defining a sequence of inexpensive but incomplete rendundancy checks, using a complete decision procedure only as a last resort.

We prove that the Event-DPOR algorithm is *correct* (explores at least one execution in each equivalence class) for event-driven programs. We also prove that it is *optimal* (explores exactly one execution in each equivalence class) for the class of so-called *non-branching* programs, in which the possible sequences of shared variable accesses that can be performed during execution of an event, whose handler also executes other events, does not depend on how its execution is interleaved with other threads.

We have implemented Event-DPOR in an extension of the NIDHUGG tool [2]. Our experimental evaluation shows that, when compared with other SMC tools in which event handlers are simulated using locks, Event-DPOR incurs only a moderate constant overhead, but can be exponentially faster than other state-of-the-art DPOR algorithms. The same evaluation also shows that, unlike other algorithms that can achieve analogous reduction, Event-DPOR manages to completely avoid unnecessary exploration of executions that cannot be serialized. Moreover, in all the programs we tried, also those that are not non-branching, Event-DPOR explored the optimal number of traces, suggesting that Event-DPOR is optimal not only for non-branching programs but also for a good number of branching ones. Also, our sequence of inexpensive checks for redundancy was sufficient in all tried programs, i.e., we never had to invoke the decision procedure for this NP-hard problem.

2   Related Work

Stateless model checking has been implemented in many tools for analysis of multithreaded programs (e.g., [16,34,9,2,42,35,22,26]). It often employs DPOR, introduced by Flanagan and Godefroid [12] to reduce the number of schedulings that must be explored. Further developments of DPOR reduce this number further, by being optimal (i.e., exploring only one scheduling in each equivalence class) [1,3,6,23] or by weakening the equivalence [6,5,8,4].

DPOR has been adapted to event-driven multi-threaded programs. Jensen et al. [20] consider an execution model in which events are processed in arbitrary order (multiset semantics) and apply it to JavaScript programs. Maiya et al. [28] consider a model where events are processed in the order they are received (FIFO semantics), and develop a tool, EM-Explorer, for analyzing Android applications

which, given a particular sequence of event executions, produces a set of reorderings of its events which reverses its conflicts. The above works are based on the algorithm of Flanagan and Godefroid [12], implying that they do not take advantage of subsequent improvements in DPOR algorithms [1,3,23], nor do they employ techniques such as sleep sets for avoiding redundant explorations. It is known [3]that even with sleep sets, the algorithm of Flanagan and Godefroid [12] can explore an exponential number of redundant execution compared to the algorithms of [1,3,23]. Without sleep sets, the amount of redundant exploration will increase further. Recently, Trimananda et al. [39] have proposed an adaptation of stateful DPOR [41,40] to non-terminating event-driven programs, which has been implemented in Java PathFinder. For analogous reason as for [20,28], also this approach does not avoid to perform redundant explorations.

For actor-based programs, in which processes communicate by message-passing, Aronis et al. [6] have presented an improvement of Optimal-DPOR in which two postings of messages to a mailbox are considered as conflicting only if their order affects the subsequent behavior of the receiver. Better reduction can then be achieved if the receiver selects messages from its mailbox based on some criterion, such as by pattern matching on the structure of the message. However, this execution model differs from the one we consider.

Event-driven programs where handlers select messages in arbitrary order from their mailbox can be analyzed by modeling messages (mini-)threads that compete for handler threads by taking locks, and applying any SMC algorithm for shared-variable programs with locks. Since typical SMC algorithms always consider different lock-protected code sections as conflicting, this approach has the drawback of exploring all possible orderings of events on a handler. There exists a technique to avoid exploring of all these orderings in programs with locks, in which lock sections can be considered non-conflicting if they do not perform conflicting accesses to shared variables. This LAPOR technique [24] is based on optimistically executing lock-protected code regions in parallel, and aborting executions in which lock-protected regions cannot be serialized. This can led to significant useless exploration, as also shown in our evaluation in Section 8.

The problem of detecting potentially harmful data races in single executions of event-driven programs has been addressed by several works. The main challenge for data race detection is to capture the often hidden dependencies for applications on Android [18,30,7,19] or on other platforms [36,37,38,29]. Detecting data races is a different problem than exploring all possible executions of a program, in that it considers only one (possibly long) execution, but tries to detect whether it (or some other similar execution) exhibits data races.

3   Main Concepts and Challenges

In this section, we informally present core concepts of our approach by examples[4]

---

[4] Note that in the remainder of the paper, we will use the term *message* to refer to what was called *event* in Sections 1 and 2, for the reason that the literature on

Writer-readers program.

s	t	u
x = 1	a = y;	c = z;
	b = x	d = x

Fig. 1. A program and its execution tree with the four executions that Optimal-DPOR will explore. In E_1, the red arcs show the conflict order; the blue arrows the program order. The first wakeup sequence is shown in green; the remaining two continue with blue.

3.1 Review of Optimal-DPOR

Our DPOR algorithm for event-driven programs is an extension of Optimal-DPOR [1]. Let us illustrate Optimal-DPOR on the program snippet shown in Fig. 1. In this code, three threads s, t, and u access three shared variables x, y, and z,[5] whereas a, b, c, and d are thread-local registers. Optimal-DPOR first explores a maximal execution, which it inspects to detect races. From each race, it constructs an initial fragment of an alternative execution which reverses the race and branches off from the explored execution just before the race. Let us illustrate with the program in Fig. 1. Assume that the first execution is E_1 (cf. the tree in Fig. 1). The DPOR algorithm first computes its happens-before order, denoted $\xrightarrow{\text{hb}}_{E_1}$, which is the transitive closure of the union of: (i) the *program order*, which totally orders the events in each thread (small blue arrows to the left of E_1), and (ii) the *conflict order* which orders conflicting events: two events are conflicting if they access a common shared variable and at least one is a write (red arcs left of E_1). A *race* consists of two conflicting events in different threads that are adjacent in the $\xrightarrow{\text{hb}}_{E_1}$-order. The execution E_1 contains two races (red arcs in Fig. 1). Let us consider the first race, in which the first event is s: x=1 and the second event is t: b=x. The alternative execution is generated by concatenating the sequence of events in E_1 that do not succeed the first event in the $\xrightarrow{\text{hb}}_{E_1}$ order (i.e., t: a = y; u: c = z) with the second event of the race t: b=x. This forms a *wakeup sequence*, which branches off from E_1 just before the race, i.e., at the beginning of the exploration (green in Fig. 1). The second race, between s: x=1 and u: d=x induces the wakeup sequence $t.u.u$ formed from the sequence t: a = y; u: c = z and the second event u: d = x, also branching off at the beginning (note that $t.u.u$ does not contain the second event t: b=x of t since it succeeds s: x=1 in the $\xrightarrow{\text{hb}}_{E_1}$-ordering). When attempting to insert $t.u.u$, the algorithm will discover that this sequence is *redundant*, since its events are

---

DPOR has reserved the term *event* to denote an execution of a program statement. We will also use *mailbox* instead of event pool.

[5] Throughout this paper, we assume that threads are spawned by a main thread, and that all shared variables get initialized to 0, also by the main thread.

Fig. 2. An event-driven program with non-conflicting messages (top left). A program with non-atomic conflicting messages (bottom left) and its tree of executions (right).

consistently contained in a continuation $(t.u.t.u)$ of the already inserted wakeup sequence $t.u.t$, and it will therefore not insert $t.u.u$. After this, the algorithm will reclaim the space for E_1, extend $t.u.t$ into a maximal execution E_2, in which races are detected that generate two new wakeup sequences (which start in green and continue in blue), which are extended to two additional executions (cf. Fig. 1).

3.2   Challenges for Event-driven Programs

A naïve way in which existing DPOR algorithms can handle event-driven programs is to consider all pairs of messages as conflicting. However, such an approach is *not* effective, since it will lead to exploration of all different serialization orders of the messages, even if they are non-conflicting, as is the case for the top left program of Fig. 2 in which two threads s and t post two messages p_1 and p_2 to a handler thread h. (We show messages labeled by the message identifier and wrapped in brackets.) Since the events of p_1 and p_2 are non-conflicting, exploring only one execution suffices. In general, some messages of a program may be conflicting and some may not be, so a DPOR algorithm for event-driven programs should explore only the necessary inequivalent orderings between conflicting messages. Event-DPOR achieves this by extending Optimal-DPOR's technique for reversing races between events in different threads to a mechanism for reversing races between events in different messages.

We illustrate this mechanism on the program at the bottom left of Fig. 2. Assume that the first explored execution is E_1. It contains two races between events in the two messages, one on x and one on y. According to Optimal-DPOR's principle for race reversal, the race on x should induce an alternative execution composed of the sequence of events that do not happen-after the first event (i.e., $h: p_1: \mathtt{u = 1}$ $h: p_2: \mathtt{v = 2}$) and the second event $h: p_2: \mathtt{a = x}$ (for brevity, we do not show the two post events). However, since message execution is serialized, these events cannot form an execution. Therefore, Event-DPOR forms the alternative

Fig. 3. A program with messages that branch on read values and its exploration tree.

execution (shown in blue) by appending the second event h: p_2: $\mathtt{a=x}$ to a maximal subset of the events of E_1 which is closed under $\xrightarrow{\text{hb}}_{E_1}$-predecessors (i.e., if it contains an event e then it also contains all its $\xrightarrow{\text{hb}}_{E_1}$-predecessors), and which can form an execution that does not contain the first event. Later, this wakeup sequence is extended to execution E_2. Let us then consider the race on $\mathtt{y}$. The constructed wakeup sequence should append the second event h: p_2: $\mathtt{b=y}$ to a maximal subset of events that do not happen-after the first event h: p_1: $\mathtt{y=1}$. However, there is no execution that satisfies these constraints, since it would have to include h: p_2: $\mathtt{a=x}$ before its $\xrightarrow{\text{hb}}_{E_1}$-predecessor h: p_1: $\mathtt{x=1}$. The conclusion is that the race on $\mathtt{y}$ cannot (and should not) be considered for reversal, whereas that on $\mathtt{x}$ should be reversed. More generally, if two messages executing on the same handler thread are in conflict, then a wakeup sequence is constructed consisting of only the second message up until and including its first conflicting event.

When messages can branch on values read from shared variables, reversing the order of two messages may change the control flow of each involved message. Also in this case, Event-DPOR's principles for reversing races work fine. We illustrate this on the program in Fig. 3, consisting of two threads s and t and a handler thread h. Thread s posts a message p_1 to h and thereafter writes to $\mathtt{x}$. Thread t posts message p_2 to h that reads from $\mathtt{x}$ and *may* then read from $\mathtt{y}$.

Assume that the first execution is E_1, where s's access to $\mathtt{x}$ goes last. The execution has two races: one on $\mathtt{y}$ between p_1: $\mathtt{y=2}$ and p_2: $\mathtt{b=y}$, and one on $\mathtt{x}$ between p_2: $\mathtt{a=x}$ and s: $\mathtt{x=1}$. The race on $\mathtt{x}$ can be handled in the same way as in Optimal-DPOR: the wakeup sequence is s: $\mathtt{x=1}$, which branches off after the prefix $s.t.p_1$ (green in Fig. 3), and will subsequently be extended to execution E_2. The race on $\mathtt{y}$ is a race between events in two messages on the same handler thread. As in the previous example, the wakeup sequence will include the second message up until and including the first racing event, which is p_2: $\mathtt{b=y}$. Included in the events that do not happen-after the first event is also s: $\mathtt{x=1}$, which must be placed after its predecessor p_2: $\mathtt{a=x}$, yielding the wakeup sequence p_2: $\mathtt{a=x}$; s: $\mathtt{x=1}$; p_2: $\mathtt{b=y}$, which branches off after s: $\mathtt{post}(p_1,h)$, t: $\mathtt{post}(p_2,h)$. This is the blue rightmost branch of the tree in Fig. 3, and is later extended into the

t	h's messages	k's messages
$\texttt{post}(p_1,h)$	$p_1: \begin{bmatrix} \texttt{d = 1;} \\ \texttt{a = y} \end{bmatrix}$	$q_1: \begin{bmatrix} \texttt{y = 1;} \\ \texttt{x = 1} \end{bmatrix}$
$\texttt{post}(p_2,h)$		
$\texttt{post}(q_1,k)$	$p_2: [\texttt{z = 1}]$	$q_2: \begin{bmatrix} \texttt{b = z;} \\ \texttt{c = x} \end{bmatrix}$
$\texttt{post}(q_2,k)$		

Fig. 4. A program in which a reversal of the race on $\texttt{x}$ will reorder messages on the handler k, and two executions that will be explored.

execution E_3. Execution E_3 has a race on $\texttt{x}$. Its reversal produces the wakeup sequence s: $\texttt{x = 1}$, which is a tentative branch next to p_2: $\texttt{a = x}$. However, this wakeup sequence is not in conflict with the left branch labeled p_1: $\texttt{b = y}$, which means that it will not be inserted for the reason that it is equivalent to a subsequence of an execution starting with p_1: $\texttt{b = y}$, namely E_2.

Reordering Messages when Reversing Races Event-DPOR's principles for reversing races may necessitate reordering of messages on handlers that are not involved in the race. Consider the program in Fig. 4. Assume that the first explored execution is E_1, where we have omitted the initial sequence of post events of thread t for succinctness. In E_1, message p_1 is processed before p_2, and q_1 is processed before q_2. There are three races in E_1, one on each of the shared variables $\texttt{x}$, $\texttt{y}$, $\texttt{z}$. Let us consider the race on $\texttt{x}$, shown by the red arrow. A wakeup sequence which reverses this race must include all events of q_2, since these are the $\xrightarrow{\text{hb}}_{E_1}$-predecessors of q_2: $\texttt{c = x}$. It must also include the write to $\texttt{z}$ by p_2 since it is a $\xrightarrow{\text{hb}}_{E_1}$-predecessor of events in q_2. On the other hand, it cannot include any part of the message q_1, since q_1 must now occur after q_2, and therefore it also cannot include the read of $\texttt{y}$ by p_1 since its predecessor in q_1 is missing. In summary, the wakeup sequence contains two fully processed messages p_2 and q_2, the event h: p_1: $\texttt{d = 1}$ of p_1, but no events from q_1. Such a wakeup sequence must branch off after the post events of t, i.e., from the root of the tree to the right in Fig. 4. Later, this wakeup sequence is extended to a full execution E_2. In total, the program of Fig. 4 has eight inequivalent executions (the other six are not shown).

4 Computation Model

4.1 Programs

We consider programs consisting of a finite set of *threads* that interact via a finite set of *(shared) variables*. Each thread is either a *normal thread* or a *handler*

thread. A normal thread has a finite set of local registers and runs a deterministic code, built in a standard way from expressions and atomic statements, using standard control flow constructs (sequential composition, selection and bounded iteration). Atomic statements read or write to shared variables and local registers, including read-modify-write operations, such as compare-and-swap. A handler thread has a *mailbox* to which all threads (also handler threads) can post messages. A mailbox has unbounded capacity, implying that the posting of a message to a mailbox can never block. A message consists of a deterministic code, built in the same way as the code of a thread. We let $\texttt{post}(p, h)$ denote the statement which posts the message p into the mailbox of handler thread h. A handler thread repeatedly extracts a message from its mailbox, executes the code of the message to completion, then extracts a next message and executes its code, and so on. Messages are extracted from the mailbox in arbitrary order. The execution of a message is interleaved with the statements of other threads.

The local state of a thread is a valuation of its local registers together with the contents of its mailbox. A global state of a program consists of a local state of each thread together with a valuation of the shared variables. The program has a unique initial state, in which mailboxes are empty.

Recall that we use *message* to denote what is called *event* in Section 1.

4.2   Events, Executions, Happens-before Ordering, and Equivalence

We use $s, t, \ldots$ for threads, $p, q, \ldots$ for messages and non-handler threads, $\texttt{x}$, $\texttt{y}$, $\texttt{z}$ for shared variables, and $\texttt{a}$, $\texttt{b}$, $\texttt{c}$, $\texttt{d}$ for local registers. We assume, wlog, that the first event of a message does not access a shared variable, but only performs a local action, e.g., related to initialization of message execution. In order to simplify the presentation, we henceforth extend the term *message* to refer not only to a message but also to a non-handler thread.

The execution of a program statement is an *event*, which affects the global state of the program. An event is denoted by a pair $\langle p, i \rangle$, where p denotes the message containing the event and i is a positive integer, denoting that the event results from the i-th execution step in message p. An *execution sequence* E is a finite sequence of events, starting from the initial state of the program. Since thread and message codes are deterministic, an execution sequence E can be uniquely characterized by the sequence of messages (and non-handler threads) that perform execution steps in E, where we use dot(.) as concatenation operator. Thus $p.p.q$ denotes the execution sequence consisting first of two events of p, followed by an event of q.

We let $enabled(E)$ denote the set of messages that can perform a next event in the state to which E leads. A sequence E is *maximal* if $enabled(E) = \emptyset$. We use $u, v, w, \ldots$ to range over sequences of events. We introduce the following notation, where E is an execution sequence and w is a sequence of events.

- $\langle \rangle$ denotes the empty sequence.
- $E \vdash w$ denotes that $E.w$ is an execution sequence.
- $w \backslash p$ denotes the sequence w with its first occurrence of p (if any) removed.

- $dom(E)$ denotes the set of events $\langle p, i \rangle$ in E, that is, $\langle p, i \rangle \in dom(E)$ iff E contains at least i events of p. We also write $e \in E$ to denote $e \in dom(E)$.
- $next_{[E]}(p)$ denotes the next event to be performed by the message p after the execution E if $p \in enabled(E)$, otherwise $next_{[E]}(p)$ is undefined.
- $\widehat{e}$ denotes the message that performs e, i.e., e is of form $e = \langle \widehat{e}, i \rangle$ for some i.
- $E' \leq E$ denotes that E' is a (not necessarily strict) prefix of E.

We say that p *starts after* E if p has been posted in E, but not yet performed any events in E. We say that p *is active after* E if p has been posted in E, but not finished its execution in E.

Definition 1 (Happens-before). *Given an execution sequence E, we define the* happens-before *relation on E, denoted $\xrightarrow{\text{hb}}_E$, as the smallest irreflexive partial order on $dom(E)$ such that $e \xrightarrow{\text{hb}}_E e'$ if e occurs before e' in E and either*

- *e and e' are performed by the same message p,*
- *e and e' access a common shared variable $\boldsymbol{x}$ and at least one writes to $\boldsymbol{x}$, or*
- *$\widehat{e'}$ is the message that is posted by e and e' is the first event of $\widehat{e'}$.*   □

The hb-*trace* (or *trace* for short) of E is the directed graph $(dom(E), \xrightarrow{\text{hb}}_E)$.

Definition 2 (Equivalence). *Two execution sequences E and E' are* equivalent, *denoted $E \simeq E'$, if they have the same trace. We let $[E]_\simeq$ denote the equivalence class of E.*   □

Note that for programs that do not post or process messages, $\simeq$ is the standard Mazurkiewicz trace equivalence for multi-threaded programs [32,14,12,1]. We say that two sequences of events, w and w', with $E \vdash w$ and $E \vdash w'$, are *equivalent after E*, denoted $w \simeq_{[E]} w'$ if $E.w \simeq E.w'$.

5   The Event-DPOR Algorithm

In this section, we present *Event-DPOR*, a DPOR algorithm for event-driven programs. Given a terminating program on given input, the algorithm explores different maximal executions resulting from different thread interleavings.

5.1   Central Concepts in Event-DPOR

Definition 3 (Happens-before Prefix). *Let E and E' be execution sequences. We say that E' is a* happens-before prefix *of E, denoted $E' \sqsubseteq E$, if (i) $dom(E') \subseteq dom(E)$, (ii) $\xrightarrow{\text{hb}}_{E'}$ is the restriction of $\xrightarrow{\text{hb}}_E$ to E', and (iii) whenever $e \xrightarrow{\text{hb}}_E e'$ for some $e' \in dom(E')$, then $e \in dom(E')$. We let $w' \sqsubseteq_{[E]} w$ denote that $E.w' \sqsubseteq E.w$.*   □

Intuitively, $E' \sqsubseteq E$ denotes that the execution E' is "contained" in the execution E in such a way that it is not affected by the events in E that are not in E'. [6] To illustrate, for the top left program of Fig. 2, the execution E' consisting of t: $\texttt{post}(p_2, h)$ h: p_2: $\texttt{y=2}$ is a happens-before prefix of any maximal execution of the program, since the event of p_2 cannot happen-after any other event than the event that posts p_2, which is already in E'.

Definition 4 (Weak Initials). *Let E be an execution sequence, and w be a sequence with $E \vdash w$. The set $WI_{[E]}(w)$ of weak initials of w after E is the set of messages p such that $E \vdash p.w'$ for some w' with $w \sqsubseteq_{[E]} p.w'$.* □

Intuitively, p is in $WI_{[E]}(w)$ if p can execute the first event in a continuation of E which "contains" w, in the sense of $\sqsubseteq$. In Event-DPOR, the concept of weak initials is used to test whether a new sequence is redundant, i.e., is "contained in" an execution that have been explored or in a wakeup sequence that is scheduled for exploration. Note that in Definition 4, we can generally not choose w' as $w \backslash p$. This happens, e.g., if p does not occur in w but instead w contains another message p' which executes on the same handler as p and does not conflict with p; in this case w' must contain a completed execution of p inserted before p'.

We illustrate using the program shown on the right. If we let E be the execution $s.t$ and w be the sequence p_1, we have $p_2 \in WI_{[E]}(w)$, since $w \sqsubseteq_{[E]} p_2.p_2.p_1$. This illustration shows that in order to determine whether $p \in WI_{[E]}(w)$ for a message p, one must know which shared-

s	t	h's messages	
$\texttt{post}(p_1, h)$	$\texttt{post}(p_2, h)$	p_1: $[\texttt{x=1}]$	
		p_2: $\begin{bmatrix}\texttt{y=2;}\\\texttt{z=2}\end{bmatrix}$	

Fig. 5. Illustrating weak initials

variable access will be performed by $next_{[E]}(p)$, and, in case p starts after E but will execute after some other message on its handler, also the sequences of shared-variable accesses that p will perform when executing to completion.

The weak initial check problem consists in checking whether $p \in WI_{[E]}(w)$.

Theorem 1. *The weak initial check problem is NP-hard.*

The proof of the above theorem can be found in Appendix B.1. In Appendix A.3, we propose a sequence of inexpensive rendundancy checks, which have shown to be sufficient for all our benchmarks.

Definition 5 (Races). *Let E be a maximal execution sequence. Two events e and e' in different messages are in a race, denoted $e \lesssim_E e'$, if $e \xrightarrow{\text{hb}}_E e'$ and*

(i) e and e' access a common shared variable and at least one is a write, and
(ii) there is no event e'' with $e \xrightarrow{\text{hb}}_E e''$ and $e'' \xrightarrow{\text{hb}}_E e'$. □

Intuitively, a race arises between conflicting accesses to a shared variable, by events which are in different messages but adjacent in the $\xrightarrow{\text{hb}}_E$ order.

---

[6] The relation $w' \sqsubseteq_{[E]} w$ is also introduced in [28], as "w is a dependence-covering sequence of w'."

5.2   The Event-DPOR Algorithm

The Event-DPOR algorithm, shown as pseudocode in Algorithm 1, performs a depth-first exploration of executions using the recursive procedure $Explore(E)$, where E is the currently explored execution, which also serves as the stack of the exploration. In addition the algorithm maintains three mappings from prefixes of E, named $done$, wut, and $parkedWuS$. For each prefix E' of E,

- $done(E')$ is a mapping whose domain is the set of messages p for which the call $Explore(E'.p)$ has returned. If p does not start after E', then $done(E')(p)$ is the shared variable-access performed by $next_{[E']}(p)$. If p starts after E', then $done(E')(p)$ is the set of sequences of shared variable-accesses that can be performed in a completed execution of p after E'. The information in $done(E')(p)$ is collected during the call $Explore(E'.p)$ (Lines 22 to 31).
- $wut(E')$ is a *wakeup tree*, i.e., an ordered tree $\langle B, \prec \rangle$ where B is a prefix-closed set of sequences, whose leaves are wakeup sequences. For each sequence $u \in B$, the order $\prec$ orders its children (of form $u.p$) by the order in which they were added to $wut(E')$. This is also the order in which the sequences of form $E'.u.p$ will be visited in the recursive exploration.
- $parkedWuS(E')$ is a set of wakeup sequences v that were previously being inserted into some wakeup tree $wut(E'')$, but were "parked" at the sequence E' because at that time there was not enough information to determine where in $wut(E'')$ to place v. Later, when a branch of $wut(E'')$ has been extended to a maximal execution, it should be possible to determine where to insert v.

Each call to $Explore(E)$ first initializes $done(E)$ and $parkedWuS(E)$ ($wut(E)$ was initialized before the call), and thereafter enters one of two phases: *race detection* (Lines 4 to 11) or *exploration* (Lines 13 to 31). The race detection phase is invoked when E is a maximal execution sequence. First, for each wakeup sequence v parked at a prefix E' of E it invokes $InsertParkedWuS(v, E')$ to insert v into the appropriate wakeup tree (Lines 5 to 7). Thereafter, each race (of form $e \lesssim_E e'$) in E is analyzed by $ReverseRace(E, e, e')$, which returns a set of executions that reverse the race. Each such execution $E'.v$ is returned as a pair $\langle E', v \rangle$, where v is a wakeup sequence that should be considered for insertion in the wakeup tree at E'. Each wakeup sequence v is checked for redundancy (Line 10), using the information in $done$. If v is not redundant, it is inserted into the wakeup tree at E' for future exploration (Line 11).

The exploration phase (Lines 13 to 33) is entered if exploration has not reached the end of a maximal execution sequence. First, if $wut(E)$ only contains the empty sequence, then an arbitrary enabled message is entered into $wut(E)$ (Lines 14 and 15). Thereafter, each sequence in $wut(E)$ is subject to recursive exploration. We find the $\prec$-minimal child p of the root of $wut(E)$ (Line 19), and make the recursive call $Explore(E.p)$ (Line 21). Before the call, $wut(E.p)$ is initialized (Line 20). During the call $Explore(E)$, information is also collected about the sequences of shared-variable accesses that can be performed by each message that is active after E, and subsequently stored in the mapping $done$.

Algorithm 1: Event-DPOR

Initial call: $Explore(\langle\rangle)$ with $wut(\langle\rangle) = \langle\{\langle\rangle\}, \emptyset\rangle$

---

1 $Explore(E)$ // Returns access sequences of messages
2     $done(E) := \emptyset$;
3     $parkedWuS(E) := \emptyset$;
4   **if** $enabled(E) = \emptyset$ **then**   // When E is maximal, enter race detection
5   **foreach** $E' \leq E$ **do**
6   **foreach** $v \in parkedWuS(E')$ **do**   // Parked wakeup sequences
7           $InsertParkedWuS(v, E')$;   // are inserted at the appropriate place
8   **foreach** e, e' such that $e \lesssim_E e'$ **do**   // For each race in E
9   **foreach** $\langle E', v \rangle \in ReverseRace(E, e, e')$ **do**   // For each race reversal
10   **if** $\neg\exists E'', w, p \;\; s.t. \;\; E''.w = E' \wedge p \in dom(done(E'')) \wedge p \in$
            $WI_{[E'']}(w.v)$ **then**   // If v is not redundant
11          $Insert(v, E', \langle\rangle)$;   // insert v into the wakeup tree at E'
12   **else**   // If not at a maximal execution sequence, explore...
13   **if** $wut(E) = \langle\{\langle\rangle\}, \emptyset\rangle$ **then**
14   **choose** $p \in enabled(E)$;   // ... or by selecting an arbitrary p...
15        $wut(E) := \langle\{\langle\rangle, p\}, \{(p, \langle\rangle)\}\rangle$;   // Adapt wakeup tree accordingly
16   **foreach** message q that is active after E **do**
17        $msgAccesses(q) := \emptyset$;   // Initialize the sequences of accesses for messages
18   **while** $\exists q \in wut(E)$ **do**   // While the wakeup tree is not empty...
19   **let** $p = \min_{\prec}\{q \in wut(E)\}$;   // ... pick next branch, ...
20        $wut(E.p) := subtree(wut(E), p)$;   // extract next wakeup tree)
21   **let** $tmpAccesses = Explore(E.p)$;   // ... and make a recursive call
22   **if** $next_{[E]}(p)$ is the last event of message p **then**
23   add p to $dom(tmpAccesses)$ with $tmpAccesses(p) = \{\langle\rangle\}$
24   **if** $next_{[E]}(p)$ performs a global access **then**
25   prepend $next_{[E]}(p)$'s access to each sequence in $tmpAccesses(p)$
26   **foreach** message q that is active after E **do**
27          $msgAccesses(q) \cup= tmpAccesses(q)$
28   add p to the domain of $done(E)$;   // Mark p as explored
29   **if** p starts after E **then**   // If p starts
30          $done(E)(p) := msgAccesses(p)$;   // ... store p's accesses
31   **else** $done(E)(p) := next_{[E]}(p)$'s access;   // ... store $next_{[E]}(p)$'s access
32   remove all sequences of form $p.w$ from $wut(E)$;   // At end, cleanup
33   **return** $msgAccesses$

The information is collected in the variable $msgAccesses$, which is initialized at Line 17. Each recursive call $Explore(E.p)$ returns the sets of access sequences performed by messages that are active after $E.p$ (Line 21). After prepending the access performed by $next_{[E]}(p)$ to the sets of access sequences performed by p (Line 25), the sets returned by $Explore(E.p)$ are added to the corresponding sets in $msgAccesses$ (Line 27). Finally, p is added to the domain of $done(E)$ (Line 28). If p starts a message after E, then $done(E)(p)$ is assigned the set of access sequences performed by p (Line 30), otherwise only the access of $next_{[E]}(p)$. Thereafter, the subtree rooted at p is removed from $wut(E)$ (Line 33). When

all recursive calls of form $Explore(E.p)$ have returned, the accumulated sets of access sequences are returned (Line 33).

Event-DPOR calls functions that are briefly described in the following paragraphs. More elaborate descriptions (with pseudocode) are in Appendix A.

$ReverseRace(E, e, e')$ is given a race $e \lesssim_E e'$ in the execution E (Line 8), and returns a set of executions that reverse the race in the sense that they perform the second event e' of the race without performing the first one, and (except for e') only contain events that are not affected by the race. More precisely, it returns a set of pairs of form $\langle E', u.e' \rangle$, such that (i) $E'.u$ is a maximal happens-before prefix of E such that $E'.u.e'$ is an execution, and (ii) $dom(E')$ is a maximal subset of $dom(E'.u)$ such that $E' \leq E$. An illustration of the $ReverseRace$ function was given for the race on x in the program of Fig. 4.

$Insert(v, E', \langle\rangle)$ inserts the wakeup sequence v into the wakeup tree $wut(E')$. If there is already some sequence u in $wut(E')$ such that $u \sqsubseteq_{[E']} v$ or $v \sqsubseteq_{[E']} u$, then the insertion leaves $wut(E')$ unaffected. Otherwise $Insert(v, E', \langle\rangle)$ attempts to find the $\prec$-minimal non-leaf sequence u in $wut(E')$ with $u \sqsubseteq_{[E']} v$, and insert a new leaf of form $u.v'$ into $wut(E')$, such that $v \sqsubseteq_{[E']} u.v'$, which is ordered after all existing descendants of u in $wut(E')$. The function finds such a u by descending into $wut(E')$ one event at a time; from each node u' it finds a next node $u'.p$ as the $\prec$-minimal child with $u'.p \sqsubseteq_{[E']} v$. If, during this search, the message p starts after $E'.u'$ it may happen that the wakeup tree does not contain enough subsequent events to determine whether $u'.p \sqsubseteq_{[E']} v$; in this case the sequence v is "parked" at the node $u'.p$: the insertion of v will be resumed when $E'.u'.p$ is extended to a maximal execution (at Line 7 with E' being $E'.u'$).

$InsertParkedWuS(v, E')$ inserts a wakeup sequence v, which is parked after a prefix E' of the execution E, into an appropriate wakeup tree. The function first decomposes E' as $E''.p$, and checks whether $p \in WI_{[E'']}(v)$, using information about the accesses of p that can be found in E. If the check succeeds, then insertion proceeds recursively one step further in the execution E, otherwise v conflicts with p and should be inserted into the wakeup tree after E''.

Checking for Redundancy Tests of form $p \in WI_{[E]}(w)$ for a message p and an execution $E.w$ appear at Line 10 and in the functions $InsertWuS$ and $InsertParkedWuS$. If p does not start after E, then the check can be straightforwardly performed using sleep sets [14]. If p starts after E, then checking whether $p \in WI_{[E]}(w)$ is NP-hard in the general case (see Theorem 1). To avoid expensive calls to a decision procedure, Event-DPOR employs a sequence of incomplete checks, starting with simple ones, and proceeding with a next test only if the preceding was not conclusive. These tests are in order: 1) If p is the first message (if any) on its handler in w, then $p \in WI_{[E]}(w)$ is trivially true. 2) If the happens-before relation precludes p from executing first on its handler, then $p \in WI_{[E]}(w)$ is false; checking this may require w to be extended so that p (and possibly other messages) are executed to completion. 3) An attempt is made to construct an actual execution in which p is the first message on its handler, which respects the happens-before ordering. 4) If all previous tests were inconclusive, a decision procedure is invoked as a final step.

6   Correctness and Optimality

A program is defined to be *non-branching* if each message, which executes on the same handler as some other message, performs the same sequence of accesses (reads or writes) to shared variables during its execution, regardless of how its execution is interleaved with other threads and messages. Note that the "non-branching" restriction does not apply to non-handler threads nor to messages that are the only ones executing on their handler.

The following theorems state that Event-DPOR is *correct* (explores at least one execution in each equivalence class) for *all* event-driven programs and *optimal* (explores exactly one execution in each equivalence class) for non-branching programs. Proofs can be found in Appendix C.

Theorem 2 (Correctness).   *Whenever the call to $Explore(\langle\rangle)$ returns during Algorithm 1, then for all maximal execution sequences E, the algorithm has explored some execution sequence in $[E]_{\simeq}$.*

Theorem 3 (Optimality).   *When applied to a non-branching program, Algorithm 1 never explores two maximal execution sequences which are equivalent.*

7   Implementation

Event-DPOR was implemented on top of NIDHUGG. NIDHUGG [2] is a state-of-the-art stateless model checker for C/C++ programs with Pthreads, which works at the level of the LLVM Intermediate Representation. NIDHUGG comes with a selection of DPOR algorithms. One of them is Optimal-DPOR, which we have used as a basis for Event-DPOR's implementation.

We have extended the data structures of NIDHUGG with the information needed by Event-DPOR. For instance, nodes in wakeup trees contain new information, such as the set of parked wakeup sequences, and events in executions include the information in *tmpAccesses*, used to compute the *done* set as shown in Lines 23 to 30 of Algorithm 1. The relation $\xrightarrow{\text{hb}}_E$ is represented by a vector clock per event, containing the set of preceding events. When reversing races (in *ReverseRace*) and checking for redundancy (Line 10 of Algorithm 1), the relation $\xrightarrow{\text{hb}}_E$ is extended by a saturation operation (Definition 6 in Appendix A) that captures ordering constrained induced by serialized message execution.

Concerning race reversal, instead of reversing multiple races between messages executed on the same handler, our implementation detects and reverses only the race induced by the first conflict, since other races cannot be reversed, as explained using the example in Fig. 2. Moreover, in cases where *ReverseRace* would return several maximal executions that reverse a race, our implementation instead returns their union, even though it may not form an execution (e.g., since it may contain several incomplete executed messages on a handler). From this union, events will be removed adaptively during wakeup tree insertion to extract only those maximal executions that generate new leaves in a wakeup tree.

8   Evaluation

In this section, we evaluate the performance of our implementation and put it into context. Since currently there is no other SMC tool for event-driven programs to compare against,[7] we have created an API, in the form of a C header file, that implements event handlers as pthread mutexes (locks) and simulates messages as threads that wait for their event handler to be free. This API allows us to use plain C/pthread programs to compare Event-DPOR with the Optimal-DPOR algorithm implemented in Nidhugg as baseline, but also with the *Lock-Aware Partial Order Reduction* (*LAPOR*) algorithm [24], implemented in GenMC. The LAPOR algorithm is often analogous to Event-DPOR w.r.t. the amount of reduction that it can achieve when event handlers are modeled as global locks. We also include in our comparison the baseline DPOR algorithm of GenMC that tracks the modification order (`-mo`) of shared variables. For Nidhugg, we used its `master` branch at the end of 2022; for GenMC, we used version 0.6.1.[8] We have run all benchmarks on a Ryzen 5950X desktop running Arch Linux.

We will compare implementations of different DPOR algorithms based on the number of executions that they explore, as well as the time that this takes. For some programs, LAPOR also examines a fair amount of *blocked* executions (i.e., executions that cannot be serialized and need to be aborted), which naturally affects its time performance. In Table 1, we show the number of executions explored by an entry of the form $T+B$, where T is the number of complete traces and B is the number of blocked executions. (We omit the B part when it is zero.)

All the benchmark programs we use are parametric, typically on the number of threads used (and thus messages posted); their parameters are shown inside parentheses. In the first program (posters), each thread posts to a single event handler two messages containing stores to some atomic global variable, and then the value of this variable is checked by an assertion. This simple program allows us to establish the baseline speed of all implementations. We can see that GenMC `-mo` is the fastest here. The reason is that it does not perform any checks whether the explored executions are sequentially consistenct, which allows it to be five times faster than LAPOR, and seven to nine times faster than Nidhugg's algorithm implementations. We can also notice that Event-DPOR incurs a small but noticeable overhead over Optimal-DPOR for the extra machinery that its implementation requires.

The next two benchmarks were taken from a paper by Kragl et al. [27]. In buyers, n "buyer" threads coordinate the purchase of an item from a "seller" as follows: one buyer requests a quote for the item from the seller, then the buyers coordinate their individual contribution, and finally if the contributions are enough to buy the item, the order is placed. In ping-pong, the "pong" handler

---

[7] All our attempts to use R^4 failed miserably; the tool has not been updated since 2016.

[8] GenMC v0.6.1 (released July 2021) warns that LAPOR usage with `-mo` is experimental; in fact, LAPOR support has been dropped in more recent GenMC versions.

Table 1. Performance of different DPOR algorithm implementations.

| | Executions (Traces+Blocked) | | | | Time (secs) | | | |
| | GenMC | | Nidhugg | | GenMC | | Nidhugg | |
Benchmark	-mo	-lapor	-optimal	-event	-mo	-lapor	-optimal	-event
posters(3)	90	90	90	90	0.02	0.03	0.09	0.09
posters(4)	2520	2520	2520	2520	0.18	0.81	0.94	1.42
posters(5)	113400	113400	113400	113400	9.43	47.11	50.87	84.64
buyers(6)	720	720+2383	720	720	0.08	2.51	0.36	0.51
buyers(7)	5040	5040+20301	5040	5040	0.56	25.80	2.53	3.96
buyers(8)	40320	40320+191369	40320	40320	5.03	306.95	23.59	37.70
ping-pong(6)	3276	3276+8271	3276	3276	0.23	3.99	1.45	2.61
ping-pong(7)	27252	27252+79435	27252	27252	2.01	44.51	13.78	26.42
ping-pong(8)	253296	253296+835509	253296	253296	20.63	572.07	149.26	299.12
consensus(2)	4	4+4	4	4	0.01	0.01	0.06	0.06
consensus(3)	216	125+347	216	125	0.04	0.29	0.20	0.20
consensus(4)	331776	50625+242828	331776	50625	75.43	293.91	419.90	177.63
prolific(5)	120	30+26	120	30	0.17	5.34	0.21	0.18
prolific(7)	5040	126+120	5040	126	16.12	98.14	11.79	2.12
prolific(9)	362880	510+502	362880	510	2462.83	1132.65	1363.31	26.28
sparse-mat(4,3)	204	34	204	34	0.16	0.06	0.16	0.09
sparse-mat(4,5)	185520	1546	185520	1546	212.51	3.56	126.06	1.66
sparse-mat(4,7)	🕐	130922	🕐	130922	🕐	603.31	🕐	234.27
plb(4)	105	1	105	1	0.02	0.01	0.10	0.06
plb(6)	10395	1	10395	1	1.99	0.02	6.61	0.06
plb(8)	2027025	1	2027025	1	556.46	0.02	1808.24	0.06

thread receives messages with increasing numbers from the "ping" thread, which are then acknowledged back to the "ping" event handler.

Looking at Table 1, we notice that, in both buyers and ping-pong, all algorithms explore the same number of traces, but LAPOR also explores a significant number of executions that cannot be serialized and need to be aborted. In fact, for both benchmarks, the aborted executions significantly outnumber the traces explored. This affects negatively the time that LAPOR takes, and GenMC -lapor becomes the slowest implementation. In contrast, Event-DPOR does not suffer from this problem and shows similar scalability as baseline GenMC and Optimal-DPOR.

With the four remaining benchmarks, we evaluate all implementations in programs where algorithms tailored to event-driven programming, either natively (Event-DPOR) or which are lock-aware (when handlers are implemented as locks), have an advantage. The first program (consensus), again from the paper by Kragl et al. [27], is a simple *broadcast consensus* protocol for n nodes to agree on a common value. For each node i, two threads are created: one thread executes a `broadcast` method that sends the value of node i to every other node, and the other thread is an event handler that executes a `collect` method which receives n values and stores the maximum as its decision. Since every node receives the values of all other nodes, after the protocol finishes, all nodes have decided on the same value. The next program (prolific) is synthetic: n threads send n messages with an increasing number of stores to and loads from an atomic global variable to one event handler. The sparse-mat program com-

putes the number of non-zero elements of a sparse matrix of dimension $m \times n$, by dividing the work into n tasks sent as messages to different handlers, which compute and join their results. The last benchmark (plb) is taken from a paper by Jhala and Majumdar [21]. A fixed sequence of task requests is received by the main thread. Upon receiving a task, the main thread allocates a space in memory and posts a message with the pointer to the allocated memory that will be served by a thread in the future.

Refer again to Table 1. In consensus, all algorithms start with the same number of traces, but LAPOR and Event-DPOR need to explore fewer and fewer traces than the other two algorithms, as the number of nodes (and threads) increases. Here too, LAPOR explores a significant number of executions that need to be aborted, which hurts its time performance. On the other hand, Event-DPOR's handling of events is optimal here. The prolific program shows a case where algorithms not tailored to events (or locks) explore $(n-1)!$ traces, while LAPOR and Event-DPOR explore only $2^n - 2$ consistent executions, when running the benchmark with parameter n. It can also be noted that Event-DPOR scales *much* better than LAPOR here in terms of time, due to the extra work that LAPOR needs to perform in order to check consistency of executions (and abort some of them). The sparse-mat program shows another case where algorithms that are not tailored to events explore a large number of executions unnecessarily ($\odot$ denotes timeout). This program also shows that Event-DPOR beats LAPOR time-wise even when LAPOR does not explore executions that need to be aborted. Finally, plb shows a case on which Event-DPOR and LAPOR really shine. These algorithms need to explore only one trace, independently of the size of the matrices and messages exchanged, while DPOR algorithms not tailored to event-driven programs explore a number of executions which increases exponentially and fast.

We remark that, in all benchmarks, the inexpensive checks for redundancy were sufficient, and Event-DPOR explored the optimal number of traces. Results from an extended set of benchmarks appear in Appendix D.

9   Concluding Remarks

In this paper, we presented a novel SMC algorithm, Event-DPOR, tailored to the characteristics of event-driven multi-threaded programs running under the SC semantics. The algorithm was proven correct and optimal for event-driven programs in which the variable accesses of events do not depend on how their execution is interleaved with other threads.

We have implemented Event-DPOR in the NIDHUGG tool, and we will open-source our implementation. With a wide range of event-driven programs, we have shown that Event-DPOR incurs only a moderate constant overhead over its baseline implementation (Optimal-DPOR), it is exponentially faster than existing state-of-the-art SMC algorithms in time and number of traces examined on programs where events' actions do not conflict, and does not suffer from performance degradation caused by having to examine non-serializable executions.

Event-DPOR assumes that handlers can process their events in arbitrary order. Directions for future work include to retarget Event-DPOR for event-driven programs with other policies (e.g., FIFO), and for specific event-driven execution models.

10   Reproducible Artifact

An anonymous artifact containing the benchmarks and all the tools used in the evaluation, including our Nidhugg with Event DPOR, is available at `https://doi.org/10.5281/zenodo.7929004`.

References

1. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: Symposium on Principles of Programming Languages. pp. 373–384. POPL 2014, ACM, New York, NY, USA (2014). `https://doi.org/10.1145/2535838.2535845`, `http://doi.acm.org/10.1145/2535838.2535845`
2. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 9035, pp. 353–367. Springer, Berlin, Heidelberg (2015). `https://doi.org/10.1007/978-3-662-46681-0_28`, `http://dx.doi.org/10.1007/978-3-662-46681-0_28`
3. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: A foundation for optimal dynamic partial order reduction. Journal of the ACM **64**(4), 25:1–25:49 (Sep 2017). `https://doi.org/10.1145/3073408`, `http://doi.acm.org/10.1145/3073408`
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. **3**(OOPSLA), 150:1–150:29 (Oct 2019). `https://doi.org/10.1145/3360576`, `https://doi.org/10.1145/3360576`
5. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Computer Aided Verification. LNCS, vol. 10426, pp. 526–543. Springer, Berlin Heidelberg (Jul 2017). `https://doi.org/10.1007/978-3-319-63387-9_26`, `https://doi.org/10.1007/978-3-319-63387-9_26`
6. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference. LNCS, vol. 10806, pp. 229–248. Springer, Cham (Apr 2018). `https://doi.org/10.1007/978-3-319-89963-3_14`, `https://doi.org/10.1007/978-3-319-89963-3_14`
7. Bielik, P., Raychev, V., Vechev, M.T.: Scalable race detection for android applications. In: Aldrich, J., Eugster, P. (eds.) Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 332–348. OOPSLA 2015, ACM (Oct 2015). `https://doi.org/10.1145/2814270.2814303`, `https://doi.org/10.1145/2814270.2814303`

8. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Data-centric dynamic partial order reduction. Proc. ACM on Program. Lang. **2**(POPL), 31:1–31:30 (Jan 2018). `https://doi.org/10.1145/3158119`, `http://doi.acm.org/10.1145/3158119`

9. Christakis, M., Gotovos, A., Sagonas, K.: Systematic testing for detecting concurrency errors in Erlang programs. In: Sixth IEEE International Conference on Software Testing, Verification and Validation. pp. 154–163. ICST 2013, IEEE, Los Alamitos, CA, USA (Mar 2013). `https://doi.org/10.1109/ICST.2013.50`, `https://doi.org/10.1109/ICST.2013.50`

10. Dabek, F., Zeldovich, N., Kaashoek, M.F., Mazières, D., Morris, R.T.: Event-driven programming for robust software. In: Muller, G., Jul, E. (eds.) Proceedings of the 10th ACM SIGOPS European Workshop. pp. 186–189. ACM (Jul 2002). `https://doi.org/10.1145/1133373.1133410`, `https://doi.org/10.1145/1133373.1133410`

11. Desai, A., Gupta, V., Jackson, E.K., Qadeer, S., Rajamani, S.K., Zufferey, D.: P: safe asynchronous event-driven programming. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 321–332. PLDI '13, ACM (Jun 2013). `https://doi.org/10.1145/2491956.2462184`, `https://doi.org/10.1145/2491956.2462184`

12. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of Programming Languages, (POPL). pp. 110–121. ACM, New York, NY, USA (Jan 2005). `https://doi.org/10.1145/1040305.1040315`, `http://doi.acm.org/10.1145/1040305.1040315`

13. Gibbons, P.B., Korach, E.: Testing shared memories. SIAM J. Comput. **26**(4), 1208–1244 (1997). `https://doi.org/10.1137/S0097539794279614`, `https://doi.org/10.1137/S0097539794279614`

14. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Ph.D. thesis, University of Liège (1996). `https://doi.org/10.1007/3-540-60761-7`, `http://www.springer.com/gp/book/9783540607618`, also, volume 1032 of LNCS, Springer.

15. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Principles of Programming Languages, (POPL). pp. 174–186. ACM Press, New York, NY, USA (Jan 1997). `https://doi.org/10.1145/263699.263717`, `http://doi.acm.org/10.1145/263699.263717`

16. Godefroid, P.: Software model checking: The VeriSoft approach. Formal Methods in System Design **26**(2), 77–101 (Mar 2005). `https://doi.org/10.1007/s10703-005-1489-x`, `http://dx.doi.org/10.1007/s10703-005-1489-x`

17. Godefroid, P., Hanmer, R.S., Jagadeesan, L.: Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 124–133. ISSTA, ACM, New York, NY, USA (Mar 1998). `https://doi.org/10.1145/271771.271800`, `https://doi.org/10.1145/271771.271800`

18. Hsiao, C., Pereira, C., Yu, J., Pokam, G., Narayanasamy, S., Chen, P.M., Kong, Z., Flinn, J.: Race detection for event-driven mobile applications. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 326–

336. PLDI '14, ACM (Jun 2014). `https://doi.org/10.1145/2594291.2594330`, `https://doi.org/10.1145/2594291.2594330`

19. Hu, Y., Neamtiu, I., Alavi, A.: Automatically verifying and reproducing event-based races in android apps. In: Zeller, A., Roychoudhury, A. (eds.) Proceedings of the 25th International Symposium on Software Testing and Analysis. pp. 377–388. ISSTA 2016, ACM (Jul 2016). `https://doi.org/10.1145/2931037.2931069`, `https://doi.org/10.1145/2931037.2931069`

20. Jensen, C.S., Møller, A., Raychev, V., Dimitrov, D., Vechev, M.T.: Stateless model checking of event-driven applications. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 57–73. OOPSLA 2015, ACM, New York, NY, USA (2015). `https://doi.org/10.1145/2814270.2814282`, `https://doi.org/10.1145/2814270.2814282`

21. Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: Hofmann, M., Felleisen, M. (eds.) Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007. pp. 339–350. ACM (2007). `https://doi.org/10.1145/1190216.1190266`, `https://doi.org/10.1145/1190216.1190266`

22. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM on Program. Lang. **2**(POPL), 17:1–17:32 (Jan 2018). `https://doi.org/10.1145/3158105`, `https://doi.org/10.1145/3158105`

23. Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. **6**(POPL), 1–28 (2022). `https://doi.org/10.1145/3498711`, `https://doi.org/10.1145/3498711`

24. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Effective lock handling in stateless model checking. Proc. ACM Program. Lang. **3**(OOPSLA), 173:1–173:26 (Oct 2019). `https://doi.org/10.1145/3360599`, `https://doi.org/10.1145/3360599`

25. Kokologiannakis, M., Sagonas, K.: Stateless model checking of the Linux kernel's hierarchical read-copy-update (tree RCU). In: Proceedings of International SPIN Symposium on Model Checking of Software. pp. 172–181. SPIN 2017, ACM, New York, NY, USA (2017). `https://doi.org/10.1145/3092282.3092287`, `https://doi.org/10.1145/3092282.3092287`

26. Kokologiannakis, M., Vafeiadis, V.: GenMC: A model checker for weak memory models. In: Computer Aided Verification - 33rd International Conference, CAV 2021, Proceedings, Part I. LNCS, vol. 12759, pp. 427–440. Springer (Jul 2021). `https://doi.org/10.1007/978-3-030-81685-8_20`, `https://doi.org/10.1007/978-3-030-81685-8_20`

27. Kragl, B., Enea, C., Henzinger, T.A., Mutluergil, S.O., Qadeer, S.: Inductive sequentialization of asynchronous programs. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 227–242. PLDI 2020, ACM (Jun 2020). `https://doi.org/10.1145/3385412.3385980`, `https://doi.org/10.1145/3385412.3385980`

28. Maiya, P., Gupta, R., Kanade, A., Majumdar, R.: Partial order reduction for event-driven multi-threaded programs. In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016). LNCS, vol. 9636, pp. 680–697. Springer, Berlin,

Heidelberg (Apr 2016). `https://doi.org/10.1007/978-3-662-49674-9_44`, `https://doi.org/10.1007/978-3-662-49674-9_44`

29. Maiya, P., Kanade, A.: Efficient computation of happens-before relation for event-driven programs. In: Bultan, T., Sen, K. (eds.) Proceedings of the 26th International Symposium on Software Testing and Analysis. pp. 102–112. ISSTA 2017, ACM, New York, NY, USA (Jul 2017). `https://doi.org/10.1145/3092703.3092733`, `https://doi.org/10.1145/3092703.3092733`

30. Maiya, P., Kanade, A., Majumdar, R.: Race detection for android applications. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 316–325. ACM (2014). `https://doi.org/10.1145/2594291.2594311`, `https://doi.org/10.1145/2594291.2594311`

31. Mazières, D.: A toolkit for user-level file systems. In: Park, Y. (ed.) Proceedings of the General Track: 2001 USENIX Annual Technical Conference. pp. 261–274. USENIX (Jun 2001), `http://www.usenix.org/publications/library/proceedings/usenix01/mazieres.html`

32. Mazurkiewicz, A.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Applications and Relationships to Other Models of Concurrency. LNCS, vol. 255, pp. 279–324. Springer, Berlin Heidelberg (1987). `https://doi.org/10.1007/3-540-17906-2_30`, `http://dx.doi.org/10.1007/3-540-17906-2_30`

33. Mednieks, Z., Dornin, L., Meike, G.B., Nakamura, M.: Programming Android. "O'Reilly Media, Inc." (2012)

34. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation. pp. 267–280. OSDI '08, USENIX Association, Berkeley, CA, USA (Dec 2008), `http://dl.acm.org/citation.cfm?id=1855741.1855760`

35. Norris, B., Demsky, B.: A practical approach for model checking C/C++11 code. ACM Trans. Program. Lang. Syst. **38**(3), 10:1–10:51 (May 2016). `https://doi.org/10.1145/2806886`, `http://doi.acm.org/10.1145/2806886`

36. Petrov, B., Vechev, M.T., Sridharan, M., Dolby, J.: Race detection for web applications. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 251–262. ACM (2012). `https://doi.org/10.1145/2254064.2254095`, `https://doi.org/10.1145/2254064.2254095`

37. Raychev, V., Vechev, M.T., Sridharan, M.: Effective race detection for event-driven programs. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013. pp. 151–166. ACM (2013). `https://doi.org/10.1145/2509136.2509538`, `https://doi.org/10.1145/2509136.2509538`

38. Santhiar, A., Kaleeswaran, S., Kanade, A.: Efficient race detection in the presence of programmatic event loops. In: Zeller, A., Roychoudhury, A. (eds.) Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20,

2016. pp. 366–376. ACM (2016). https://doi.org/10.1145/2931037.2931068, https://doi.org/10.1145/2931037.2931068

39. Trimananda, R., Luo, W., Demsky, B., Xu, G.H.: Stateful dynamic partial order reduction for model checking event-driven applications that do not terminate. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13182, pp. 400–424. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_20, https://doi.org/10.1007/978-3-030-94583-1_20

40. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient stateful dynamic partial order reduction. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5156, pp. 288–305. Springer (2008). https://doi.org/10.1007/978-3-540-85114-1_20, https://doi.org/10.1007/978-3-540-85114-1_20

41. Yi, X., Wang, J., Yang, X.: Stateful dynamic partial-order reduction. In: Liu, Z., He, J. (eds.) Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4260, pp. 149–167. Springer (2006). https://doi.org/10.1007/11901433_9, https://doi.org/10.1007/11901433_9

42. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: Programming Language Design and Implementation (PLDI). pp. 250–259. ACM, New York, NY, USA (Jun 2015). https://doi.org/10.1145/2737924.2737956, http://doi.acm.org/10.1145/2737924.2737956

A   Detailed Descriptions of Auxiliary Functions

In this section, we describe in detail the functions that are called by Event-DPOR, and were briefly described at the end of Section 5.2 Some of these functions extend the happens-before relation $\xrightarrow{\text{hb}}_E$ on an execution with additional ordering constraints that are enforced in the event-driven execution model, stemming from the fact that each handler must execute its messages in some sequential order. The following *saturation operation* adds such additional orderings imposed by any ordering relation on events.

Definition 6 (Saturation).   *Let E be a sequence of events, and $\rightarrow_E$ be an irreflexive partial order on the events of E. We define $\langle\!\langle \rightarrow_E \rangle\!\rangle$ as the smallest transitive relation $\xrightarrow{\text{st}}_E$ on the events of E which includes $\rightarrow_E$ and satisfies the constraint that whenever e and e' are events in different messages on the same handler, and there is an event e'' in the same message as e and an event e''' in the same message as e' with $e'' \xrightarrow{\text{st}}_E e'''$, then $e \xrightarrow{\text{st}}_E e'$.*   □

In the above definition, note that it is not required that e is distinct from e'', nor that e' is distinct from e'''.

A.1   Reversing Races

A key procedure of Event-DPOR is $ReverseRace(E, e, e')$ which constructs new executions by analyzing and reversing a race in an explored execution. This procedure is given a race $e \lesssim_E e'$ in the currently explored execution E (at Line 8 of Algorithm 1), and returns a set of maximal executions that reverse the race. More precisely, it returns a set of pairs of form $\langle E', u.e' \rangle$, such that (i) $E'.u$ is a maximal happens-before prefix of E such that $E'.u.e'$ is an execution, and (ii) $dom(E')$ is a maximal subset of $dom(E'.u)$ such that $E' \leq E$.

   The procedure $ReverseRace(E, e, e')$ is shown in Algorithm 2. Let E'' be the set of events of E that are not affected by the race (Line 2): this is the set of events e''' with $e \xrightarrow{\text{hb}}_E e'''$. If E'' can be reordered to form an execution, the code at Lines 4 to 23 will have no effect; $ReverseRace$ will terminate and returns its linearization. However, there are situations in which E'' cannot be reordered into an execution. For instance, E'' may contain two incomplete messages on the same handler because the remaining parts of these messages happen-after e in E. Since an execution may contain at most one incomplete message per handler, $ReverseRace$ then performs a sequence of message removals and reorderings to produce a set of maximal wakeup sequences. The procedure employs the saturation operation of Definition 6 to constrain the ordering between messages on the same handler. The procedure maintains an ordering relation $\xrightarrow{sc}_{E''}$ on E'', initialized to $\langle\!\langle \xrightarrow{hb}_{E''} \rangle\!\rangle$ (Line 4). It thereafter performs a sequence of steps in which messages are removed from E'' and/or the ordering relation $\xrightarrow{sc}_{E''}$ is extended. Some steps may be resolved nondeterministically: in such cases the procedure pursues all possible alternatives, potentially resulting in several returned sequences. The steps of Algorithm 2 are the following.

Algorithm 2: Reversal of a Race.

---

1 $ReverseRace(E, e, e')$

2   **let** E'' be the subsequence of E consisting of the events e''' with $e \overset{\text{hb}}{\nrightarrow}_E e'''$;

3   **let** e'' be the last event performed by $\widehat{e'}$;

4 $\overset{sc}{\longrightarrow}_{E''} := \langle\!\langle \overset{hb}{\longrightarrow}_{E''} \rangle\!\rangle$;

5   **let** $S = \{E''\}$;

6   **foreach** $F \in S$ **do**

7   **repeat**

8   **if** *an incomplete message includes an event e''' with $e''' \overset{\text{hb}}{\longrightarrow}_F e''$* **then**

9   remove all other incomplete messages on same handler from F;

10   **if** *several incomplete messages execute on one handler* **then**

11   **foreach** *incomplete message p in the same handler* **do**

12   construct a sequence U where all the messages except p from this handler are removed;

13   add U to the set S;

14   delete F from S;

15   pick another sequence F from S;

16   **foreach** *incomplete message p* **do**

17   add relation $\overset{sc}{\longrightarrow}_F$ from all other messages on same handler to p and saturate;

18   **foreach** *cycle in $\overset{sc}{\longrightarrow}_F$* **do**

19   remove a message in the cycle;

20   remove events that follow already removed events in the $\overset{\text{hb}}{\longrightarrow}_F$ ordering;

21   **if** *an event e''' s.t. $e''' \overset{\text{hb}}{\longrightarrow}_F e''$ is deleted* **then**

22   remove F from S and exit the loop;

23   **until** *each handler in F has at most one incomplete message*;

24   **let** $WSS = \emptyset$ // *set of wakeup sequences*;

25   **foreach** $F \in S$ **do**

26   **foreach** *two messages from the same handler that are not ordered by $\overset{sc}{\longrightarrow}_F$* **do**

27   add relation $\overset{sc'}{\longrightarrow}_F$ as they appear in F;

28   **repeat**

29   nondeterministically pick two messages ordered by the relation $\overset{sc'}{\longrightarrow}_F$ and reverse the order;

30   **until** *until $\overset{sc}{\longrightarrow}_F$ and $\overset{sc'}{\longrightarrow}_F$ together are acyclic*;

31   topologically sort F respecting $\overset{sc}{\longrightarrow}_F$ and $\overset{sc'}{\longrightarrow}_F$;

32   extract largest common prefix E' of F and E;

33   add $\langle E', .u.e' \rangle$ to WSS, where $F = E'.u$;

34   **return**(WSS);

Line 5 After the loop from Line 7 to Line 23, the set S will contain all the possible sequences with at most one incomplete message per handler.

Line 8 If an incomplete message includes an event e''' with $e''' \xrightarrow{hb}_F e''$, then any other message on the same handler which is not completely executed in F must be removed.

Line 10 If several incomplete messages execute on the same handler, then finds all the possible sequence where only one of the incomplete messages is present and saves them to S.

Line 16 Whenever a handler has an incomplete message p, any other message p' on that handler must be executed before p, represented by extending $\xrightarrow{sc}_F$ from the last event of p' to the first event of p and then saturating.

Line 18 If $\xrightarrow{sc}_F$ becomes cyclic during the filtering and ordering procedure, then each cycle should be broken by removing the events in a suitable message.

Line 21 It is possible to have two or more incomplete messages from the same handler in F each having at least one event that happens-before e''. Because of this reason or non-deterministic choice during message deletion process described previously, an event e''' such that $e''' \xrightarrow{hb}_F e''$ might be deleted from E''. Then the algorithm removes F from S.

Line 27 By adding additional relation $\xrightarrow{sc'}_F$, the algorithm determines a total order on the messages from the same handler.

Line 29 If $\xrightarrow{sc}_F$ and $\xrightarrow{sc'}_F$ together form a cycle, the algorithm tries to guess another order $\xrightarrow{sc'}_F$. Systematic search of $\xrightarrow{sc'}_F$ is a NP-complete problem in general case (see Theorem 4 below). But for the programs we have tried so far, doing Line 27 is sufficient.

Line 31 The sequence u is linearized by topological sort procedure while respecting $\xrightarrow{sc}_F$ and $\xrightarrow{sc'}_F$.

As an illustration, consider the race on x in the program of Fig. 4. Here, there is a unique (up to equivalence) maximal execution which reverses the race, which consists of all events that post messages, all events in messages p_2 and q_2, and the assignment to d by p_1. The read of x by q_2 should be ordered last, since it corresponds to the racing event e'. Message q_1 is removed by the rule at Line 8, whereby also the second of event of p_1 is removed, since it reads from the first event in q_1.

Event-driven Consistency. When describing Line 29 above, we stated that the problem of determininig whether a given happens-before relation can be obtained from some execution is NP-complete. This follows from NP-completeness of the event-driven consistency problem. The event-driven consistency problem consists in checking whether, for a given directed graph $(S, \xrightarrow{hb}_S)$ where S is a set of events and $\xrightarrow{hb}_S$ is a set of edges, there is an execution sequence E such that $(S, \xrightarrow{hb}_S)$ is the hb-*trace* of E.

Theorem 4. *The event-driven consistency problem is NP-complete.*

Algorithm 3: Insertion into Wakeup Tree.

---

1 $Insert(v, E', u)$
2   **if** v *is the empty sequence* **or** u *is a leaf in* $wut(E')$ **then return**;
3   **foreach** child $u.p$ of u, in order (from left to right) **do**
4   **if** p *does not start after* $E'.u$ **then**   // If a new message is not started ...
5   **if** $p \in WI_{[E'.u]}(v)$ **then**
6   **if** $next_{[E'.u]}(p) \in v$ **then** $Insert(v \backslash p, E', u.p)$ **else return**;
7   **else**
8   **if** p *is the first (if any) message on its handler in* v **then**
9   **if** $next_{[E'.u]}(p) \in v$ **then** $Insert(v \backslash p, E', u.p)$ **else return**;
10   **else if** p *is fully present in* v **then**
11   **if** $p \in WI_{[E'.u]}(v)$ **then** $Insert(v \backslash p, E', u.p)$ **else continue**;
12   **else**
13   add v to $parkedWuS(E'.u.p)$;
14   **return**;
15   insert v as a new branch from u, ordered after the existing children of u;
16   **return**;

The proof of the above theorem can be found in Appendix B.2. Given this NP-hardness result, we define a procedure to reverse races (Appendix A.1) that makes use of a saturation procedure to constrain the ordering between messages and therefore reduces the number of cases to consider.

A.2   Wakeup Tree Insertion

In this section, we formally define wakeup trees, and present the procedure *InsertWuS* for inserting wakeup sequences, and *InsertParkedWuS* for inserting parked wakeup sequences.

Definition 7 (Wakeup Tree). *A* wakeup tree *is an ordered tree* $\langle B, \prec \rangle$*, where B (the set of* nodes*) is a finite prefix-closed set of sequences of messages, with the empty sequence* $\langle \rangle$ *being the root. The children of a node u, of form $u.p$ for some set of messages p, are ordered by $\prec$. In the tree* $\langle B, \prec \rangle$*, such an ordering between children is extended to a total order $\prec$ on B by letting $\prec$ be the induced post-order relation between the nodes in B (i.e., if the children $u.p_1$ and $u.p_2$ are ordered as $u.p_1 \prec u.p_2$, then $u.p_1 \prec u.p_2 \prec u$ in the induced post-order).*   □

Insertion of a wakeup sequence v into the wakeup tree $wut(E')$ is performed by calling the function $Insert(v, E', u)$ with parameters v and E', and the parameter u being the empty sequence. The call $Insert(v, E', \langle \rangle)$ will, if v conflicts with all its current leaves, extend the wakeup tree $wut(E')$ by a new leaf v' such that $v' \simeq E'v$. The recursive function $Insert(v, E', u)$, shown in Algorithm 3, traverses the wakeup tree $wut(E')$ from the root downwards, where u is the current point of the traversal. The initial call is performed with u being the empty sequence. Each invocation of $Insert(v, E', u)$ first checks whether a leaf has been reached or all of v has already been examined, in which case nothing new should

---

Algorithm 4: Insertion of Parked Wakeup Sequences.

---

1 $InsertParkedWuS(v, E')$
2   let $E''.p$ be E';
3   **if** v *is the empty sequence* **then return**;
4   **if** E'' *was formerly a leaf in a wakeup tree* **then return**;
5   **if** $p \in WI_{[E'']}(v)$ **then**
6   let q be the message following E' in E;
7 $InsertParkedWuS(v \backslash p, E'.q)$;
8   **else**
9 $Insert(v, E'', \langle \rangle)$;   // If conflict w. next event, insert into the wakeup tree

---

be added to $wut(E')$ (Line 2). Thereafter, it considers the children of u (of form $u.p$) from left to right. For each child $u.p$, the algorithm tries to determine whether or not $p \in WI_{[E.u]}(v)$. If p does not start after $E'.u$ then $p \in WI_{[E.u]}(v)$ then $p \in WI_{[E.u]}(v)$ can be checked by simple inspection at Lines 4 to 6 (as described in the second paragraph of Appendix A.3). The algorithm traverses to $u.p$ by a call to $Insert(v \backslash p, E', u.p)$ if $p \in WI_{[E.u]}(v)$, otherwise it considers the next child of u if $p \notin WI_{[E.u]}(v)$. If $p \in WI_{[E.u]}(v)$ but p does not appear in v, then actually no wakeup sequence need be inserted (Line 6). If p starts after $E'.u$ (Line 7), then

- the case in which p is the first (if any) message on its handler in v, considered at Line 9 is performed according to the **Simple Check** in Appendix A.3;
- if p executes to completion in the sequence v (Line 10), then v contains sufficient information to decide whether $p \in WI_{[E.u]}(v)$ using the remaining sequence of checks in Appendix A.3;
- if none of these two cases apply, then more information is needed about which accesses p performs when it is executed. Therefore the sequence v is "parked" at the node $u.p$: the insertion of v will be resumed when the node $u.p$ is extended to a maximal execution starting with $E'.u.p$, which happens at Line 7 of Algorithm 1 with E' being $E'.u$.

If all children $u.p$ of u have been traversed with failing tests for $p \in WI_{[E.u]}(v)$, then v is added as a new branch from u, ordered after the already existing children (Line 15).

It remains to define the procedure for inserting parked wakeup sequences (called at Line 7 of Algorithm 1). This insertion is described in Algorithm 4, as the function $InsertParkedWuS(v, E')$, which inserts a wakeup sequence v which is parked after a prefix E' of the execution E. The function first decomposes E' as $E''.p$, and checks whether $p \in WI_{[E'']}(v)$. Information about the accesses of p can now be found in the execution E, so that the check $p \in WI_{[E'']}(v)$ can be performed. The check will be exact for non-branching programs, but possibly conservative in general. If the check succeeds, then insertion proceeds one step further in the execution E (Line 7), otherwise v conflicts with p and so should be inserted at the wakeup tree after E'' (Line 9). As an additional optimization, Line 4 checks whether E'' was the leaf that is extended to the currently explored

execution. If so, the insertion can return without inserting anything, in analogy with how leaves are handled in wakeup tree insertion (Line 2 of Algorithm 3).

A.3   Checking for Redundancy

Let us now consider the problem of deciding whether $p \in WI_{[E]}(w)$ for a message p and an execution $E.w$.

If p does not start after E, then $p \in WI_{[E]}(w)$ can be checked by simple inspection, as follows. If $next_{[E]}(p)$ is a local event or posts a message, then $p \in WI_{[E]}(w)$ holds trivially. If $next_{[E]}(p)$ accesses a shared variable, then (i) if p appears in w, we have $p \in WI_{[E]}(w)$ precisely when there is no event e in w such that $e \xrightarrow{hb}_{E.w} next_{[E]}(p)$, and (ii) if p does not appear in w, we have $p \in WI_{[E]}(w)$ precisely when no event in w conflicts with $next_{[E]}(p)$.

If p starts after E, then checking whether $p \in WI_{[E]}(w)$ is NP-hard in the general case, as we show in Theorem 1. However, in many cases, the check can be performed by tests that run in polynomial time. Event-DPOR employs the following sequence of checks, starting with simple ones, and resorting to an exact decision procedure only as a last step. We assume that the event which posts message p appears in E, otherwise $p \in WI_{[E]}(w)$ is trivially false.

Simple Check If p is the first message (if any) on its handler in w, then $p \in WI_{[E]}(w)$ is trivially true (recall our assumption that the first event of a message does not access a shared variable).

Happens-Before Check If p is not the first message on its handler in w, we check whether there is a happens-before dependency from a message p' which precedes p on its handler, as follows.

1. If p is not executed to completion in w, we extend w by a sequence of events performed by p which performs all the shared-variable accesses that p did not perform in w. If after this extension, some event of p happens-after an event in a message q on another handler which is not executed to completion in w, then w is further extended by events of q in the same way. If an event of q again happens-after an event in an incomplete message on some other handler, this procedure is repeated recursively until convergence, resulting in an extension w' of w.

2. Thereafter, the happens-before relation $\xrightarrow{hb}_{E.w'}$ is extended to include ordering constraints induced by the event-driven execution model.

   (i) First $\xrightarrow{hbp\langle E \rangle}_{E.w'}$ is constructed as the smallest transitive relation which includes $\xrightarrow{hb}_{E.w'}$ and in addition enforces $e \xrightarrow{hbp\langle E \rangle}_{E.w'} e'$ whenever e is in a message whose first event is in E and e' occurs after e on the same handler as e.

   (ii) Thereafter, $\xrightarrow{sc}_{E.w'}$ is defined as the saturation $\langle\!\langle \xrightarrow{hbp\langle E \rangle}_{E.w'} \rangle\!\rangle$ of $\xrightarrow{hbp\langle E \rangle}_{E.w'}$ (see Definition 6).

   If now $e' \xrightarrow{sc}_{E.w'} e$ for some event e in p and event e' in a message which precedes p on the same handler, then $p \in WI_{[E]}(w)$ must be false.

Witness Construction If the Happens-Before Check was not negative, the next step is to construct an actual execution in which p is the first message on its handler. First, $\xrightarrow{sc}_{E.w'}$ is extended, by ordering the events in p before any event in a message that precedes p in w' on the same handler, and thereafter saturated by the saturation operation $\langle\!\langle \cdot \rangle\!\rangle$. If the result contains a cycle, then $p \in WI_{[E]}(w)$ must be false. Otherwise we extend the saturated extension of $\xrightarrow{sc}_{E.w'}$ to a total order on the messages of each handler, by ordering messages that are still unordered to execute in the same order as they appear in w'. If this can be done without creating a cycle then $p \in WI_{[E]}(w)$ is true.

Decision Procedure If a cycle is created, then a decision procedure is invoked as a final step.

B   Proofs of Complexity Results

In this section, we prove the complexity results of Theorems 1 and 4 but first we need to define the happens-before relation on the events of each execution sequence.

Given an execution sequence E, we define the *happens-before relation* on E, denoted $\xrightarrow{hb}_E$, as the irreflexive partial order on $dom(E)$ induced by the union of three sub-relations, $\xrightarrow{po}_E$, $\xrightarrow{cnf}_E$, and $\xrightarrow{pb}_E$. Each of these is a sub-relation of $<_E$, defined as follows.

$e \xrightarrow{po}_E e'$ if e and e' are performed by the same message p.

$e \xrightarrow{cnf}_E e'$ if e and e' access a common shared variable x, and at least one of them writes to x.

$e \xrightarrow{pb}_E e'$ if $\widehat{e'}$ is the message that is posted by e and e' is the first event of $\widehat{e'}$.

Intuitively, $\xrightarrow{po}_E$ (*program order*) is the total order of events of each message. Note that $\xrightarrow{po}_E$ does not order events of different messages relative to each other. The relation $\xrightarrow{cnf}_E$ (*conflicts with*) captures data flow constraints arising from reads and writes to shared variables. The relation $\xrightarrow{pb}_E$ (*posted by*) captures the causal dependency from message posting to message execution.

B.1   Proof of Theorem 1

We prove the lower bound by reduction from the VSC-read problem. The reduction is similar to the one from the event-driven consistency problem to the VSC-read problem. The idea is to start from an execution sequence and reversing the order of two messages will lead to the pattern used in the hardness proof of the event-driven consistency problem. In this proof, we will replace the conflict relation from p_e to $p_{e,x}$ by a sequence of conflict relations that go through two particular messages $p'_{e,x}$ and $p''_{e,x}$ if they are executed in a certain order. Otherwise, there is no conflict relation from p_e to $p_{e,x}$, and so the happens-before relation is acyclic.

We use the same set of assumptions as in the hardness proof of the event-driven consistency problem. We now reduce the VSC-read problem to the order reversing problem. Let $(S, \xrightarrow{\text{hb}}_S = \xrightarrow{\text{po}}_S \cup \xrightarrow{\text{rf}}_S)$ be a directed graph. As in the previous proof, we associate a message p_e for each event e. The message p_e will be executed by the handler h. For every write event e in S executed by a thread t, we have a message p_e of the form $[x_e = 1;\ x_t = 1;\ x'_e = 1]$. For a read event e' in S executed by a thread t and reading from the write event e, we have a message $p_{e'}$ which is of the form $[a = y_e;\ x_t = 1;\ y_{e'} = 0]$.

We also use an extra handler h_x for each variable x (used in the events of S). For each write event e on the variable x, we have a message $p_{e,x}$ that will run the following sequence of statements $[x_e = 0;\ z_e = 0;\ y_e = 0;\ z'_e = 0]$. We then add a conflict relation from the first write of $p_{e,x}$ to the first write of p_e. This will force the message p_e to start after $p_{e,x}$. For a read event e' reading from the write event e, we also add a conflict relation from the third write of $p_{e,x}$ to the first read of $p_{e'}$. Observe that we do not impose a direct conflict relation from p_e or $p_{e'}$ to $p_{e,x}$.

For each write event e on the variable x, we have two messages $p'_{e,x}$ and $p''_{e,x}$ that run on a fresh handler h_e the following sequence of statements $[a = z;\ x'_e = 1]$ and $[z_e = 0;\ a = x]$ respectively. We add a conflict relation from the second last event of p_e to the second event of $p'_{e,x}$ and from the first write event of $p''_{e,x}$ to the last event of the second write event of $p_{e,x}$. Observe that in the case that $p'_{e,x}$ is executed before $p''_{e,x}$, we have an indirect conflict relation from the last write of p_e to the second write of $p_{e,x}$ through $p'_{e,x}$ and $p''_{e,x}$. In the case where we execute $p''_{e,x}$ before $p'_{e,x}$, there is no (indirect) happens-before relation from p_e to $p_{e,x}$.

In similar manner, for each read event e' on x in S reading from the write event e, we have two messages $p'_{e',x}$ and $p''_{e',x}$ that run on a fresh handler $h_{e'}$ the following sequence of statements $[a = z;\ y_{e'} = 1]$ and $[a = z'_e;\ a = x]$ respectively. We add a conflict relation from the last event of $p_{e'}$ to the second event of $p'_{e',x}$ and from the first read event of $p''_{e',x}$ to the last write event of $p_{e,x}$. Observe that in the case that $p'_{e',x}$ is executed before $p''_{e',x}$, we have an indirect conflict relation from the last write of $p_{e'}$ to the last write of $p_{e,x}$ through $p'_{e',x}$ and $p''_{e',x}$.

To set the order of all $p'_{e,x}$ and $p''_{e,x}$ ($p'_{e',x}$ and $p''_{e',x}$), we will use two messages p and p' on a fresh handler h' that run the following statements $[x_p = 1;\ z = 1]$ and $[x_{p'} = 1;\ x = 1]$ respectively. We add then a conflict relation from the first read event of $p'_{e,x}$ (resp. $p'_{e',x}$) to the event of p and from the write event of p' to the last event of $p''_{e,x}$ (resp. $p''_{e',x}$). Note that if p is executed before p' then $p'_{e,x}$ (resp. $p'_{e',x}$) is executed before $p''_{e,x}$ (resp. $p''_{e',x}$).

Let $(S', \xrightarrow{\text{hb}'}_{S'} = \xrightarrow{\text{po}}_{S'} \cup \xrightarrow{\text{cnf}}_{S'} \cup \xrightarrow{\text{pb}}_{S'})$ be the constructed hb-trace from $(S, \xrightarrow{\text{hb}}_S = \xrightarrow{\text{po}}_S \cup \xrightarrow{\text{rf}}_S)$. It is easy to see that there is an execution sequence E such that $(S', \xrightarrow{\text{hb}}_{S'})$ is the hb-*trace* of E and where the message p' is executed before p and $p''_{e,x}$ (resp. $p''_{e',x}$) is executed before $p'_{e,x}$ (resp. $p'_{e',x}$).

Lemma 1. *There is an execution sequence E' such that $(S', \xrightarrow{\text{hb}}_{S'})$ is the* hb*-trace of E' and where the first event of message p is the first executed event in*

E if and only if there is an execution E'' such that $S = dom(E)$, $\xrightarrow{\text{po}}_S = \xrightarrow{\text{po}}_E$ and $\xrightarrow{\text{rf}}_S \subseteq \xrightarrow{\text{cnf}}_E$.

Imposing p to be executed before p' will impose that every $p'_{e,x}$ (resp. $p'_{e',x}$) is executed before $p''_{e,x}$ (resp. $p''_{e',x}$) and so there will be an indirect relation from the last write of (resp. p_e) $p_{e'}$ to the last (second) write of $p_{e,x}$ through $p'_{e,x}$ and $p''_{e,x}$ ($p'_{e',x}$ and $p''_{e',x}$). Thus, we are in similar case as in the hardness proof of the event-driven consistency problem. Furthermore, we have the first event of E' can be the first event of p since it is independent from any other event.

Lemma 2. $p \in WI_{[\langle\rangle]}(E)$ *if and only if there is an execution sequence* E' *such that* $(S', \xrightarrow{\text{hb}}_{S'})$ *is the* hb*-trace of* E' *and where the message* p *is executed before* p'.

Finally, Theorem 1 can be seen as an immediate corollary of Lemmas 1 and 2.

B.2   Proof of Theorem 4

Upper-bound Let $(S, \xrightarrow{\text{hb}}_S)$ be a directed graph (i.e., hb-trace) where S is a set of events and $\xrightarrow{\text{hb}}_S = \xrightarrow{\text{po}}_S \cup \xrightarrow{\text{cnf}}_S \cup \xrightarrow{\text{pb}}_S$ is a set of labeled edges. To show that the event-driven consistency problem is NP, it suffices to first guess a total ordering $<_S$ between the messages on the same thread handler. Observe that we can have at most one incomplete message per handler which should be scheduled last with respect to $<_S$. We then use the total order relation $<_S$ to extend the *program order* relation $\xrightarrow{\text{po}}_S$ into a total order relation $\xrightarrow{\text{po}}$ on the set of events executed by the same handler such that: (1) $e \xrightarrow{\text{po}} e'$ if $e \xrightarrow{\text{po}}_S e'$, and (2) $e \xrightarrow{\text{po}} e'$ whenever e and e' are events in two different messages p and p' on the same handler and $p <_S p'$. Finally, the extended happens-before relation $\xrightarrow{\text{hb}} = \xrightarrow{\text{po}} \cup \xrightarrow{\text{cnf}}_S \cup \xrightarrow{\text{pb}}_S$ is acyclic (which is equivalent to checking sequential consistency of the extended graph $(S, \xrightarrow{\text{hb}})$) if and only if there is an execution sequence E such that $(S, \xrightarrow{\text{hb}})$ is the hb-*trace* of E (i.e., $S = dom(E)$, $\xrightarrow{\text{po}} = \xrightarrow{\text{po}}_E$, $\xrightarrow{\text{cnf}}_S = \xrightarrow{\text{cnf}}_E$ and $\xrightarrow{\text{pb}}_S = \xrightarrow{\text{pb}}_E$). Observe that checking the acyclicity of the extended happens-before relation $\xrightarrow{\text{hb}}_S$ can be done in polynomial time. Furthermore, the execution sequence E can be obtained via the linearlization of the extended happens-before relation $\xrightarrow{\text{hb}}$ (since the extend the *program order* relation $\xrightarrow{\text{po}}$ forces the messages on the same handler to be executed one after the other).

Lower-bound We prove the lower bound by reduction from the problem of verifying the sequential consistency of traces when only the read-from relation is given. Hereafter, we call this problem VSC-read. The VSC-read problem consists in checking whether, given a directed graph $(S, \xrightarrow{\text{hb}}_S = \xrightarrow{\text{po}}_S \cup \xrightarrow{\text{rf}}_S)$ where S is a set of write and read events, $\xrightarrow{\text{po}}_S$ is the program order relation that totally orders all the events of each thread, and $\xrightarrow{\text{rf}}_S$ is the *read-from* relation that maps each read event to the write event from which it gets its value, there is an execution sequence E such that $S = dom(E)$, $\xrightarrow{\text{po}}_S = \xrightarrow{\text{po}}_E$ and $\xrightarrow{\text{rf}}_S \subseteq \xrightarrow{\text{cnf}}_E$. The

VSC-read problem is known to be NP-complete in the size of the program [13, Theorem 4.1].

We now reduce the VSC-read problem to the event-driven consistency problem. Let $(S, \xrightarrow{\text{hb}}_S = \xrightarrow{\text{po}}_S \cup \xrightarrow{\text{rf}}_S)$ be a directed graph. To simplify the presentation[9], we assume w.l.o.g. that each write event is read by at least one read event. The main idea of our reduction is to associate a message p_e for each event e. The message p_e will be executed by the handler h. The order of the execution of these messages will correspond to a linearization of the set of event S (since all these messages will be executed by the same handler h). However, this poses a challenge since such reduction from the VSC-read problem to the event-driven consistency problem will fix the order of write events on the same variable (as it is implied by the conflict relation $\xrightarrow{\text{cnf}}$). To address this challenge, we rename the shared variables used by each event in S and thus there will be no conflict relation between write-write events (and therefore between read-write events too). However, this leads to a new challenge which is how to make sure that between a write event $e \in S$ and a read event $e' \in S$ that is reading from e there is no other scheduled write event in S on the same variable between e and e'. To address this second challenge, we use an extra handler h_x per variable x that executes a number of independent messages (one per write event on x in S). The order in which these messages are executed corresponds to the order in which the write events on the same variable are scheduled. Furthermore, we make sure that each read event is scheduled after the write event it reads from and before the next scheduled write event on the same variable.

Formally, for every write event e in S executed by a thread t, we create a message p_e running on the thread handler h. The message p_e will be of the form $[x_e$ = 1; x_t = 1; x_e = 1$]$. For a read event e' in S executed by a thread t and reading from the write event e, we create a message $p_{e'}$ running on the thread handler h. The message $p_{e'}$ will be of the form $[$a = y_e; x_t = 1; a = $y_e]$. We use the write event on x_t to order the messages corresponding to events running on the same thread t in S. In fact, we will simulate $\xrightarrow{\text{po}}_S$ using $\xrightarrow{\text{cnf}}_{S'}$ that will totally order all the write events on x_t. This results in adding a conflict relation $\xrightarrow{\text{cnf}}_{S'}$ between every two events corresponding to the writes on x_t in two different messages p_e and $p_{e'}$ iff $e \xrightarrow{\text{po}}_S e'$.

The statements on x_e and y_e are used to force a total order on the messages corresponding to events on the same variable such that all the read messages are scheduled just after their corresponding write messages. To that aim, we use an extra handler h_x for each variable x (used by the events of S). For each write event e on the variable x, we create a message $p_{e,x}$ that will run the following sequence of statements $[x_e$ = 0; x_e = 0; y_e = 0; y_e = 0$]$. We then add a conflict relation from the first write of $p_{e,x}$ to the first write of p_e and from the last write of p_e to the second write of $p_{e,x}$. This will force the message

---

[9] We assume that threads/messages are spawned/posted by a main thread, and that all shared variables get initialized to 0, also by the main thread. To make the presentation simple, we omit the events of the main thread.

p_e to start and end before its corresponding reads. For a read event e' reading from the write event e, we also add a conflict relation from the third write of $p_{e,x}$ to the first read of $p_{e'}$ and from the last read of $p_{e'}$ to the last write of $p_{e,x}$. This conflict relation will force that the entire message $p_{e'}$ will be executed just after the message p_e without the interleaving of any other message that corresponds to a write event on x between p_e and p'_e.

Observe that the messages p_e are run one after the other (since they are on the same handler h). Furthermore, the constraints between the messages of handler h and those of handler h_x impose that the read message $p_{e'}$ is scheduled just after its corresponding write message p_e but before the next scheduled write message on the same variable. Let $(S', \xrightarrow{\text{hb}'}_{S'} = \xrightarrow{\text{po}}_{S'} \cup \xrightarrow{\text{cnf}}_{S'} \cup \xrightarrow{\text{pb}}_{S'})$ be the constructed hb-trace from $(S, \xrightarrow{\text{hb}}_S = \xrightarrow{\text{po}}_S \cup \xrightarrow{\text{rf}}_S)$. It is then easy to see that:

Lemma 3. *There is an execution sequence E' such that $(S', \xrightarrow{\text{hb}}_{S'})$ is the hb-trace of E' if and only if there is an execution E such that $S = dom(E)$, $\xrightarrow{\text{po}}_S = \xrightarrow{\text{po}}_E$ and $\xrightarrow{\text{rf}}_S \subseteq \xrightarrow{\text{cnf}}_E$.*

C   Proof of Correctness and Optimality

In this section, we prove correctness (Theorem 2) and optimality (Theorem 3) of the Event-DPOR algorithm.

C.1   Proof of Theorem 2

Let us first prove Theorem 2. This theorem follows from the more general Theorem 5, which we state and prove in this section.

Let us assume a particular completed execution of Event-DPOR. This execution consists of a number of terminated calls to $Explore(E)$ for some values of the parameters E and WuT. Let $\mathcal{E}$ denote the set of execution sequences E that have been explored in some call $Explore(E)$. Define the ordering $\propto$ on $\mathcal{E}$ by letting $E \propto E'$ if $Explore(E)$ returned before $Explore(E')$. Intuitively, if one were to draw an ordered tree that shows how the exploration has proceeded, then $\mathcal{E}$ would be the set of nodes in the tree, and $\propto$ would be the post-order between nodes in that tree. Theorem 2 follows from the more general Theorem 5, stated here

Theorem 5 (Correctness   of   Event-DPOR).   *Whenever   a   call   to $Explore(E)$ returns during Algorithm 1, then for all maximal execution sequences $E.w$, the algorithm has explored some execution sequence in $[E.w]_\simeq$.*

Since the initial call to the algorithm, $Explore(\langle\rangle)$, starts with the empty sequence as argument, Theorem 5 implies that for all maximal execution sequences E the algorithm explores some execution sequence E' which is in $[E]_\simeq$. Note also that a sequence of form $E.w$ need not have been explored inside

the call $Explore(E)$, but can have been explored in some earlier call, of form $Explore(E'.p)$ for some prefix E' of E.

The proof of Theorem 5 proceeds by induction on the set $\mathcal{E}$ of execution sequences E that are explored during the considered execution, using the ordering $\propto$ (i.e., the order in which the corresponding calls to $Explore(E)$ return).

We first state and prove a small lemma.

Lemma 4. *Let $\mathcal{E}$ be the tree of explored execution sequences. and let $\propto$ be the order in which the corresponding calls to $Explore(E)$ return. Consider some point in the execution, and let $wut(E)$ be the wakeup tree at E at that point, for some $E \in \mathcal{E}$.*

1. If $w \in wut(E)$ for some w, then $E.w \in \mathcal{E}$.
2. If $w \prec w'$ for $w, w' \in wut(E)$ then $E.w \propto E.w'$

Proof. The lemma follows by noting how the exploration from any $E \in \mathcal{E}$ is controlled by the wakeup tree $wut(E)$ at Lines 13 to 21 of Algorithm 1. □

We now continue with the proof of Theorem 5.

Base Case: This case corresponds to the first execution sequence E for which the call $Explore(E)$ returns. By the algorithm, E is already maximal, so the theorem trivially holds.

Inductive Hypothesis: The theorem holds for all execution sequences E' with $E' \propto E$.

Inductive Step: Proof by contradiction. Let us assume that there exists an execution E such that when the call to $Explore(E)$ returns, there is a maximal execution sequence $E.w$ such that Algorithm 1 has not explored any execution sequence in $[E.w]_\simeq$. We will show that this leads to a contradiction. So, let E be the smallest such execution in the $\propto$ order. Let $done$ be the value of the mapping $done$ when the call to $Explore(E)$ returns. Note that for such w to exist, E cannot be maximal, so $done(E)$ contains at least one message.

For each message p such that $p \in done(E')$ for some E' with $E' \le E$, define E'_p to be the longest such E'. Thus, if $p \in done(E)$ then $E'_p = E$, otherwise if E'_p is defined it is a strict prefix of E with $p \in done(E'_p)$. It follows that $E'_p.p \propto E$. We further define w'_p by $E = E'_p.w'_p$. For each message p such that E'_p is defined and $p \in WI_{[E'_p]}(w'_p)$, define

- w_p as the longest prefix of w such that $p \in WI_{[E'_p]}(w'_p.w_p)$ (such a prefix must exist since one candidate is the empty sequence),
- e_p as the first event in w which is not in w_p. Such an event e_p must exist, otherwise $w_p = w$, which implies $p \in WI_{[E'_p]}(w'_p.w)$, which together with the Inductive Hypothesis contradicts the assumption that the algorithm has not explored any execution sequence in $[E.w]_\simeq$,
- w''_p as a sequence such that $w'_p.w_p \sqsubseteq_{[E'_p]} p.w''_p$.

Among the messages p for which E'_p is defined and $p \in WI_{[E'_p]}(w'_p)$, select q such that w_q is the longest prefix among w_p. If there are several such messages q with equally long prefixes w_q, then among these pick q such that $E'_q.q$ is minimal with respect to $\propto$. Let w''_q be a sequence with $w'_q.w_q \sqsubseteq_{[E'_q]} q.w''_q$.

Let p' be the message $\widehat{e}_q$ of e_q. We first note that e_q must be a shared-variable access. To see why, note that if e_q would start the message p', then no event of the message p' can be in $w'_q.w_q$. Moreover, the handler of p' must be free after $E'_q.w'_q.w_q$, and $E'_q.w'_q.w_q$ must contain the event which posts p'. We can simply extend w''_q until it starts message p' and then we have a sequence w'''_q with $w'_q.w_q.e_q \sqsubseteq_{[E'_q]} q.w'''_q$, contradicting the choice of w_q.

There are now two cases to consider.

1. q does not start a message after E'_q. Then E'_q contains the first part of message q (up until but not including $next_{[E'_q]}(q)$). Since $w'_q.w_q \sqsubseteq_{[E'_q]} q.w''_q$, it follows that $next_{[E'_q]}(q)$ does not conflict with any event in $w'_q.w_q$, and that we can choose w''_q as $w'_q.w_q$. The only reason for $q \notin WI_{[E'_q]}(w'_q.w_q.e_q)$ (which implies $w'_q.w_q.e_q \not\sqsubseteq_{[E'_q]} q.w''_q.e_q$) is that $next_{[E'_q]}(q)$ conflicts with e_q. This implies that the execution $E'_q.q.w'_q.w_q.e_q$ contains a race between $next_{[E'_q]}(q)$ and e_q. Let w'''_q be $w'_q.w_q.e_q$ and let $E'_q.q.w'''_q.z$ be a maximal extension of $E'_q.q.w'''_q$. Then $next_{[E'_q]}(q) \lesssim_{E'_q.q.w'''_q.z} e_q$. By the Inductive Hypothesis, $Explore(E'_q.q)$ has then explored some sequence $E'_q.q.z'$ in $[E'_q.q.w'''_q.z]_{\simeq}$. When exploring it, the race $next_{[E'_q]}(q) \lesssim^m_{E'_q.q.z'} e_q$ between $next_{[E'_q]}(q)$ and e_q will be detected (at Line 8). Then $ReverseRace(E'_q.q.z', next_{[E'_q]}(q), e_q)$ will return maximal executions, one of which must contain $E'_q.w'_q.w'_q.e_q$ as a happens-before prefix.

2. q starts a message after E'_q. Since e_q is a shared-variable access, it can be simply added to p' in w''_q, obtaining w'''_q. Since $q \notin WI_{[E'_q]}(w'_q.w_q.e_q)$, w''_q must contain an event e, which is not in $w'_q.w_q$, which conflicts with e_q. This implies that the execution $E'_q.q.w'''_q$ contains a race between e and e_q. Let $E'_q.q.w'''_q.z$ be a maximal extension of $E'_q.q.w'''_q$. Then $e \lesssim_{E'_q.q.w'''_q.z} e_q$. By the Inductive Hypothesis, $Explore(E'_q.q)$ has then explored some sequence $E'_q.q.z'$ in $[E'_q.q.w''_q.z]_{\simeq}$. When exploring it, the race $e \lesssim^m_{E'_q.q.z'} e_q$ between e and e_q will be detected (at Line 8). Then $ReverseRace(E'_q.q.z', e, e_q)$ will return maximal executions, one of which must contain $E'_q.w'_q.w'_q.e_q$ as a happens-before prefix.

Let $E'_q.w'_q.w'_q.e_q$ be reordered as $E'_q.v$. It follows that $q \notin WI_{[E'_q]}(v)$, from the assumptions made when selecting q. Moreover, there cannot be any E'', w, p such that $E''.w = E'_q$ and $p \in dom(done(E''))$ and $p \in WI_{[E'']}(w.v)$, also by the assumptions made when selecting q. Thus, the wakeup sequence v will be inserted into the wakeup tree $wut(E'_q)$ (Line 11) by the call $Insert(v, E'_q, \langle\rangle)$. We claim that this insertion will add a sequence of form $E.p$ with $p \in WI_{[E]}(w_q.\widehat{e}_q)$. To see why, we consider the definition of $Insert(v, E'_q, u)$ in Algorithm 3. We first claim that during the insertion, the sequence u will always satisfy $E_q.u \leq E$ and v will satisfy $u'.w_q.\widehat{e}_q \sqsubseteq_{[E_q.u]} v$, where $u.u' = w'_q$. This is trivially true initially.

To see that it is preserved by each iteration of the loop starting at Line 3, we consider the possible children of form $u.p$. Let r be the message such that $E'_q.u.r \leq E$ (if still $E'_q.u < E$). We know that $E'_q.u.r$ is in $\mathcal{E}$ when $Explore(E)$ is returns. Furthermore, for each branch $u.p$ with $E_q.u'.p \propto E_q.u'.r$ we have that $p \notin WI_{[E.u]}(u'.w_q.\widehat{e_q})$ by the Inductive Hypothesis and the assumption that $E.w$ has not been explored. On the other hand $r \in WI_{[E.u]}(u'.w_q.\widehat{e_q})$, implying that either $u.r$ is already in $wut(E'_q)$ during the insertion, in which case the loop will move to the next iteration with invariants preserved, or $u.r$ is not already in $wut(E'_q)$ in which case it must be added during the current insertion and produce a branch $u.v$ such that $u'.w_q.\widehat{e_q} \sqsubseteq_{[E_q.u]} v$. Thus, when the insertion of v has completed, possibly after having been parked, the exploration tree will contain an execution of form $E.v'$ with $w_q.\widehat{e_q} \sqsubseteq_{[E]} v'$, thereby contradicting the assumption that w_q is the longest extension of E that has been explored. This concludes the proof of the inductive step, and Theorem 5 is proven.   □

C.2   Proof of Theorem 3

Let us next prove Theorem 3. This theorem depends on Event-DPOR being able to the following property P:

P: whenever the exploration tree $\mathcal{E}$ contains a node of form $E.p$, then the algorithm will not add an execution of form $E.w$ which is contained in some execution of form $E.p.w'$ for some w', i.e., for which $p \in WI_{[E]}(w)$.

If P is enforced, then Algorithm 1 cannot explore two equivalent maximal executions. To see this, let E be the longest common prefix of the two executions. Let the execution of the two, which is explored first, be of form $E.p.w'$. The other execution will then be the continuation of a wakeup sequence, which is inserted as a new sequence w from the node E in the exploration tree $\mathcal{E}$, and thereafter extended to $E.w.v$. But if now $E.p.w' \simeq E.w.v$, then $E.w \sqsubseteq E.p.w'$, which implies $p \in WI_{[E]}(w)$, which contradicts P.

It thus remains to check that property P is enforced. By inspection of Algorithm 1, we see that whenever a new sequence is inserted into $\mathcal{E}$, which happens before inserting a new wakeup sequence (Line 10), inside procedure *InsertWuS* (Algorithm 3) for wakeup tree insertion, and in the procedure *InsertParkedWuS* (Algorithm 4) for inserting parked wakeup sequences. Furthermore, for non-branching programs the test for $p \in WI_{[E]}(w)$, described in Appendix A.3, is exact. This concludes the proof of the theorem.   □

D   Complete Set of Benchmark Tables

In this appendix, we include the complete set of benchmark results comparing the performance of the Event-DPOR with that of the Optimal-DPOR algorithm, with the LAPOR technique implemented in GenMC and also with the baseline algorithm of GenMC which tracks the modification order (`-mo`) of shared variables. A subset of these results appears in the main body of the paper.

Table 2. Performance on programs where different DPOR algorithms implemented in NIDHUGG and GENMC explore the same number of complete and blocked executions. Times (in seconds) show the relative speed of their implementations.

| Benchmark | Executions (Traces+Blocked) | | | | Time (secs) | | | |
| | GENMC | | NIDHUGG | | GENMC | | NIDHUGG | |
	`-mo`	`-lapor`	`-optimal`	`-event`	`-mo`	`-lapor`	`-optimal`	`-event`
writers(4)	24	24	24	24	0.01	0.01	0.07	0.07
writers(6)	720	720	720	720	0.05	0.16	0.26	0.37
writers(8)	40320	40320	40320	40320	3.14	12.31	16.19	25.96
posters(3)	90	90	90	90	0.02	0.03	0.09	0.09
posters(4)	2520	2520	2520	2520	0.18	0.81	0.94	1.42
posters(5)	113400	113400	113400	113400	9.43	47.11	50.87	84.64
2PC(6)	720+720	720+720	720+720	720+720	0.41	3.57	1.22	2.14
2PC(7)	5040+5040	5040+5040	5040+5040	5040+5040	3.46	33.83	9.92	19.13
2PC(8)	40320+40320	40320+40320	40320+40320	40320+40320	33.86	359.59	96.56	210.57

Baseline Comparison First, we measure the performance of algorithm implementations on three programs where all algorithms explore the same number of executions. The first two of them are simple programs where a number of threads post racing messages to a *single* event handler. Both programs are parametric on the number of threads (and messages posted); the value of this parameter is shown inside parentheses. The messages of the first program (writers) consist of a store to the same atomic global variable followed by an assertion that checks for the value written. The second program (posters) is similar but between the write and the assertion check the messages also post, to the same handler, another message with an atomic store to the same global variable; this increases the number of executions to examine. Finally, the third program (2PC) is a two-phase commit protocol used by a coordinator and n participant threads (i.e., $n+1$ handler threads in total) to decide whether to commit or abort a transaction, by broadcasting and receiving messages.

Results from running these benchmarks for increasing number of threads are shown in Table 2. As can be seen, all algorithms explore the same number of executions here. This allows us to establish that:

(i) GENMC `-mo` is fastest overall; in particular, it is 3–7 times faster than NIDHUGG `-optimal` and about 8–9 times faster than NIDHUGG `-event`.
(ii) The overhead that LAPOR incurs over its baseline implementation in GENMC is significant. Still, for the first two programs, which involve just one event handler and no blocked or aborted executions, GENMC `-lapor` beats NIDHUGG `-event`. However, NIDHUGG `-event` is faster than GENMC `-lapor` on the third program (2PC).
(iii) The overhead that Event-DPOR incurs over Optimal-DPOR for the extra machinery that its implementation requires is small but quite noticeable.

The results from 2PC corroborate these conclusions. The blocked executions in this benchmark are due to *assume-blocking* and affect all algorithms equally in terms of additional executions examined. However, notice that GENMC `-lapor` is affected more in terms of time overhead compared to its baseline.

Table 3. Performance on programs where different DPOR algorithms implemented in NIDHUGG and GENMC examine the same number of traces, but LAPOR also explores a significant number of executions that need to be aborted. This negatively affects the runtime that SMC using LAPOR takes.

| | Executions (Traces+Blocked) | | | | Time (secs) | | | |
| | GENMC | | NIDHUGG | | GENMC | | NIDHUGG | |
Benchmark	-mo	-lapor	-optimal	-event	-mo	-lapor	-optimal	-event
buyers(6)	720	720+2383	720	720	0.08	2.51	0.36	0.51
buyers(7)	5040	5040+20301	5040	5040	0.56	25.80	2.53	3.96
buyers(8)	40320	40320+191369	40320	40320	5.03	306.95	23.59	37.70
ping-pong(6)	3276	3276+8271	3276	3276	0.23	3.99	1.45	2.61
ping-pong(7)	27252	27252+79435	27252	27252	2.01	44.51	13.78	26.42
ping-pong(8)	253296	253296+835509	253296	253296	20.63	572.07	149.26	299.12

Performance on More Involved Event-Driven Programs The next two benchmarks were taken from a recent paper by Kragl et al. [27]. In buyers, n "buyer" threads coordinate the purchase of an item from a "seller" as follows: one buyer requests a quote for the item from the seller, then the buyers coordinate their individual contribution, and finally if the contributions are enough to buy the item, the order is placed. In ping-pong, the "pong" handler thread receives messages with increasing numbers from the "ping" thread, which are then acknowledged back to the "ping" event handler.

Results from running these benchmarks are shown in Table 3. In these two programs, all algorithms explore the same number of traces, but LAPOR also explores a significant number of executions that cannot be serialized and need to be aborted. This negatively affects the time that SMC using LAPOR requires; GENMC -lapor becomes the slowest configuration here. In contrast, NIDHUGG -event shows similar scalability as baseline GENMC and NIDHUGG -optimal.

Performance on Event-Driven Programs Showing Complexity Differences Between DPOR Algorithms Finally, we evaluate all algorithms in programs where algorithms tailored to event-driven programming, either natively (Event-DPOR) or which are lock-aware (when handlers are implemented as locks), have an advantage. We use six benchmarks. The first (consensus), again from the paper by Kragl et al. [27], is a simple *broadcast consensus* protocol for n nodes to agree on a common value. For each node i, two threads are created: one thread executes a `broadcast` method that sends the value of node i to every other node, and the other thread is an event handler that executes a `collect` method which receives n values and stores the maximum as its decision. Since every node receives the values of all other nodes, after the protocol finishes, all nodes have decided on the same value. The second benchmark (db-cache) is a key-value store system inspired from Memcached, a well known distributed cache application. There are n clients requesting a fixed sequence of storage accesses to a server via UDP sockets (modeled as threads with mailboxes). On the server side there is one worker thread per client to fulfill these requests. So multiple worker threads on the server threads may race. The third benchmark (prolific)

Table 4. Performance on programs that show complexity differences in the number of traces that different DPOR algorithms implemented in Nidhugg and GenMC explore.

| Benchmark | Executions (Traces+Blocked) | | | | Time (secs) | | | |
| | GenMC | | Nidhugg | | GenMC | | Nidhugg | |
	-mo	-lapor	-optimal	-event	-mo	-lapor	-optimal	-event
consensus(2)	4	4+4	4	4	0.01	0.01	0.06	0.06
consensus(3)	216	125+347	216	125	0.04	0.29	0.20	0.20
consensus(4)	331776	50625+242828	331776	50625	75.43	293.91	419.90	177.63
db-cache(2)	4608+480	32	4608	32	0.33	0.04	2.24	0.09
db-cache(3)	3048192+401484	1764	3048192	1711	262.62	2.29	2322.91	2.02
db-cache(4)	⊙	235224	⊙	218527	⊙	402.31	⊙	418.34
prolific(5)	120	30+26	120	30	0.17	5.34	0.21	0.18
prolific(7)	5040	126+120	5040	126	16.12	98.14	11.79	2.12
prolific(9)	362880	510+502	362880	510	2462.83	1132.65	1363.31	26.28
sparse-mat(4,3)	204	34	204	34	0.16	0.06	0.16	0.09
sparse-mat(4,5)	185520	1546	185520	1546	212.51	3.56	126.06	1.66
sparse-mat(4,7)	⊙	130922	⊙	130922	⊙	603.31	⊙	234.27
mat-mult(4,3,5)	13824	1	13824	1	4.52	0.04	21.82	0.07
mat-mult(4,4,5)	331776	1	331776	1	157.49	0.05	828.91	0.07
mat-mult(4,5,5)	⊙	1	⊙	1	⊙	0.08	⊙	0.07
plb(4)	105	1	105	1	0.02	0.01	0.10	0.06
plb(6)	10395	1	10395	1	1.99	0.02	6.61	0.06
plb(8)	2027025	1	2027025	1	556.46	0.02	1808.24	0.06

is synthetic: n threads send n messages with an increasing number of stores to and loads from an atomic global variable to one event handler. The fourth benchmark (sparse-mat) computes sparseness (number of non-zero elements) of a sparse matrix of dimension $m \times n$. The work is divided among n tasks/messages and sent to different handlers, which then compute and join these results. The fifth benchmark (mat-mult) implements concurrent matrix multiplication taking two matrices of dimensions $m \times k$ and $k \times n$ as inputs. The work is divided among n tasks/messages and sent to different handlers, which then compute and join these results. The last benchmark (plb) is taken from a paper by Jhala and Majumdar [21]. The main thread receives a fixed sequence of task requests. Upon receiving a task, the main thread allocates a space in memory and posts a message with the pointer to the allocated memory that will be served by a thread in the future.

Results from running these six benchmarks are shown in Table 4.

In consensus, all algorithms start with the same number of traces, but LAPOR and Event-DPOR need to explore fewer and fewer traces than the other two algorithms, as the number of nodes (and threads) increases. Here too, LAPOR explores a significant number of executions that need to be aborted, which hurts its time performance. On the other hand, Event-DPOR's handling of events is optimal in this program, even though it is not non-branching.

The db-cache program shows a case where GenMC, both when running with -mo but also with -lapor, is non-optimal. In contrast, Event-DPOR, even with employing the inexpensive redundancy checks, manages to explore an optimal number of traces.

The prolific program shows a case where algorithms not tailored to events (or locks) explore $(n-1)!$ traces, while LAPOR and Event-DPOR explore only $2^n - 2$ consistent executions, when running the benchmark with n nodes. We briefly explain why the number of feasible executions are $2^n - 2$. Because of the access patterns of global variables in this program, each message is conflicting with the previous and the next messages. In an execution, these conflicts can be represented by n directed edges. So there are 2^n possible reorderings when both directions of each edge are considered. But two of these reorderings are not possible because they create a cycle, hence the $2^n - 2$. On this program, it can also be noted that Event-DPOR scales *much* better than LAPOR here in terms of time, due to the extra work that LAPOR needs to perform in order to check consistency of executions (and abort some of them).

The sparse-mat program shows another case where algorithms that are not tailored to events explore a large number of executions unnecessarily ($\odot$ denotes timeout). This program also shows that Event-DPOR beats LAPOR time-wise even when LAPOR does not explore executions that need to be aborted.

Finally, plb shows a case on which Event-DPOR and LAPOR really shine. These algorithms need to explore only one trace, independently of the size of the matrices and messages exchanged, while DPOR algorithms not tailored to event-driven programs explore a number of executions which increases exponentially and fast.