Randomized Testing of Distributed Systems with Probabilistic Guarantees

BURCU KULAHCIIOGLU OZKAN, Max Planck Institute for Software Systems (MPI-SWS), Germany
RUPAK MAJUMDAR, Max Planck Institute for Software Systems (MPI-SWS), Germany
FILIP NIKSIC, Max Planck Institute for Software Systems (MPI-SWS), Germany
MITRA TABAEI BEFROUEI, Vienna University of Technology, Austria
GEORG WEISSENBACHER, Vienna University of Technology, Austria

Several recently proposed randomized testing tools for concurrent and distributed systems come with theoretical guarantees on their success. The key to these guarantees is a notion of bug depth—the minimum length of a sequence of events sufficient to expose the bug—and a characterization of d-hitting families of schedules—a set of schedules guaranteed to cover every bug of given depth d. Previous results show that in certain cases the size of a d-hitting family can be significantly smaller than the total number of possible schedules. However, these results either assume shared-memory multithreading, or that the underlying partial ordering of events is known statically and has special structure. These assumptions are not met by distributed message-passing applications.

In this paper, we present a randomized scheduling algorithm for testing distributed systems. In contrast to previous approaches, our algorithm works for arbitrary partially ordered sets of events revealed online as the program is being executed. We show that for partial orders of width at most w and size at most n (both statically unknown), our algorithm is guaranteed to sample from at most $w^2 n^{d-1}$ schedules, for every fixed bug depth d. Thus, our algorithm discovers a bug of depth d with probability at least $1/(w^2 n^{d-1})$. As a special case, our algorithm recovers a previous randomized testing algorithm for multithreaded programs. Our algorithm is simple to implement, but the correctness arguments depend on difficult combinatorial results about online dimension and online chain partitioning of partially ordered sets.

We have implemented our algorithm in a randomized testing tool for distributed message-passing programs. We show that our algorithm can find bugs in distributed systems such as Zookeeper and Cassandra, and empirically outperforms naive random exploration while providing theoretical guarantees.

CCS Concepts: · Mathematics of computing → Combinatorics; · Software and its engineering → Software testing and debugging; · Theory of computation → Generating random combinatorial structures;

Additional Key Words and Phrases: distributed systems, random testing, hitting families, partially ordered sets, online chain partitioning

Authors’ addresses: Burcu Kulahcioglu Ozkan, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Str. 26, Kaiserslautern, Rheinland-Pfalz, 67663, Germany, burcu@mpi-sws.org; Rupak Majumdar, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Str. 26, Kaiserslautern, Rheinland-Pfalz, 67663, Germany, rupak@mpi-sws.org; Filip Niksic, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Str. 26, Kaiserslautern, Rheinland-Pfalz, 67663, Germany, fnksic@mpi-sws.org; Mitra Tabaei Befrouei, Vienna University of Technology, Vienna, Austria, tabaei@forsyte.at; Georg Weissenbacher, Vienna University of Technology, Vienna, Austria, georg.weissenbacher@tuwien.ac.at.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2018 Copyright held by the owner/author(s).
2475-1421/2018/11-ART160
https://doi.org/10.1145/3276530

1 INTRODUCTION
Concurrent programming is error-prone because the programmer has to reason about a large number of possible interleavings of events, any one of which may cause the program to fail. Systematic testing procedures for concurrent systems aim at finding errors by exploring the space of interleavings. Each interleaving is called a schedule. An arbitrary program of bounded execution length can have exponentially many schedules and thus, systematic testing is exponential in the worst case. However, empirically, many bugs in concurrent systems are exposed by considering the sequencing of a small number of events—the “bug depth”—independent of the ordering of the rest of the events. This observation is the basis for several systematic testing procedures [Burckhardt et al. 2010; Chistikov et al. 2016; Majumdar and Niksic 2018]: instead of exploring all possible schedules, the goal is to cover a sufficient family of schedules that guarantees that every execution of a certain depth is covered.

Formally, a concurrent system can be modeled as the generator of partially ordered events. An event can be a message send or receipt (in message-passing applications), a read from or write to a shared memory (in shared-memory applications), or a crash or reboot of a process. The partial ordering tracks the happens-before relation among the events. A schedule then is a specific linearization of the partial order. For a fixed $d$, a schedule hits an ordered sequence of $d$ events if the schedule orders these events according to the sequence, and a set of schedules is said to be $d$-hitting if for every possible ordering of $d$ events allowed by the partial order, there is a schedule that hits it. Clearly, the set of all schedules is $d$-hitting for any $d$. In practice, one would like to construct a small family of $d$-hitting schedules. Moreover, one can use a $d$-hitting family to derive a randomized testing procedure, which samples uniformly from the space of $d$-hitting families and which can provide exponentially better guarantees on the probability of finding an error of depth $d$ than naive random search.

When the partial order is given explicitly and has a known structure, such as an antichain or a tree, one can provide explicit combinatorial constructions of $d$-hitting schedules [Chistikov et al. 2016]; for antichains and trees, the size of a $d$-hitting family can be exponentially smaller than the number of events. Unfortunately, when testing a concurrent system implementation, it is unrealistic or impossible to know the partial ordering up front, e.g., if the events are exposed incrementally as the program executes, or to assume a specific “nice” structure. Thus, a challenge in systematic testing is to come up with small $d$-hitting family online (i.e., along with the execution) and for an arbitrary partial ordering.

An online construction for $d$-hitting families was demonstrated by Burckhardt et al. [2010] for multithreaded, shared-memory programs. Their algorithm, called PCT (Probabilistic Concurrency Testing), instruments a program with randomized schedule points such that the resulting program is guaranteed to uniformly sample a $d$-hitting family of schedules. In fact, PCT guarantees its schedules are sampled from a stronger variant of $d$-hitting family, which we call a strong $d$-hitting family. The key idea underlying the PCT construction is to represent the underlying partial ordering of events as a decomposition of $k$ chains, one per thread. The events are then cleverly scheduled from these chains so that each $d$-tuple of events is hit with probability at least $1/(kn^{d-1})$, where $n$ is the total number of instructions. Unfortunately, it was not known how this construction could be generalized for concurrency models in which the decomposition cannot be computed based on syntactic structures like threads. For example, an efficient PCT procedure was not known for
distributed programs communicating via asynchronous message passing, where a naive mapping of each asynchronous task to a thread would lead to a very pessimistic procedure.

In this paper, we provide a general construction for online construction of \(d\)-hitting families for arbitrary partial orders. In the general case, proving guarantees is significantly harder and requires combinatorial insights from recent results in the theory of posets. Our construction uses the combinatorial notion of adaptive chain covering [Felsner 1997]; we connect this notion with strong hitting families. In adaptive chain covering, the partial ordering is provided one element at a time, in an “upgrowing” manner. That is, the new element is guaranteed to be maximal among the elements seen so far. The adaptive chain covering algorithm must incrementally maintain a set of chains that form a chain covering—a decomposition of the partial order into a (not necessarily disjoint) union of chains. A sequence of deep results show that the optimal number of chains in an adaptive chain covering algorithm is exactly the size of an optimal strong 1-hitting family [Felsner 1997; Kloch 2007]. We generalize this result to show that the size of an optimal strong \(d\)-hitting family is bounded above by the optimal number of chains times \(n^{d-1}\), where \(n\) is the number of elements in the partial order. In particular, we re-derive the PCT result in this very general setting, since the size of the chain covering is \(k\) for \(k\) threads. The best known adaptive chain covering algorithms are in fact online chain partitioning algorithms—they decompose the partial order into a disjoint union of chains. It is not known how to effectively exploit the fact that we do not need partitions, but merely coverings [Bosek et al. 2012]. Optimal online chain partitioning algorithms use at most \(w^2\) chains, where \(w\) is the width of the partial order [Agarwal and Garg 2007]. (Recall, by Dilworth’s theorem, that \(w\) is a lower bound.) Thus, we get online hitting families of size \(w^2n^{d-1}\) for partial orders of width \(w\) and \(n\) elements. Using a general instrumentation technique, we get a randomized testing algorithm, named PCTCP (Probabilistic Concurrency Testing with Chain Partitioning), with a \(1/(w^2n^{d-1})\) probability of hitting each \(d\)-tuple for arbitrary partial orders, presented online, with (unknown) width \(w\).

While the proof of correctness is involved, the final algorithm is surprisingly simple: it involves maintaining prioritized chains of events, where the priorities are assigned randomly, picking the highest priority events at all times, and reducing the priorities of chains at \(d - 1\) randomly chosen points in the execution.

We have implemented this algorithm for distributed protocol implementations written in P# [Deligiannis et al. 2016], as well as for distributed applications such as Zookeeper and Cassandra, on top of the SAMC model checker [Leesatapornwongsa et al. 2014]. We show empirically that PCTP is effective in finding bugs in these applications and usually outperforms naive random exploration.

Our contributions are summarized as follows.

- We develop an algorithm for the online construction of hitting families of schedules for arbitrary partial orders. The construction incorporates online partitioning of a partially ordered set into a number of disjoint linearizations and enables generalizing the PCT algorithm and its probabilistic guarantees to work with arbitrary partially ordered sets. The algorithm is the basis for a simple randomized testing procedure with guaranteed lower bounds on the probability of finding depth-\(d\) bugs.
- We implement PCT with chain partitioning (PCTCP) for programs written in the P# framework, as well as the real word distributed systems Zookeeper and Cassandra. We provide our practical design choices such as modeling node crashes as events in the system or handling livelocks that are likely to occur in some distributed systems.
2 OVERVIEW OF THE APPROACH

In this section, we informally introduce some of the notions used in the rest of the paper, and we present the PCTCP algorithm and demonstrate it on a simple example.

Example 1. Consider a distributed system with three nodes\(^1\): a Handler which processes client requests, a Logger which logs transaction information, and a Terminator which terminates the system. When the Handler processes a request message from the client, it sends a log message to the Logger and a terminate message to the Terminator. When the Terminator receives a terminate message, it sends a flush message to the Logger. On receiving a flush, the Logger flushes the logs (i.e., writes the logs into the database and deallocates the file descriptors) and sends an acknowledgement flushed message back to the Terminator. The messages by the Handler are sent concurrently to Logger and Terminator. Hence, the flush message sent by Terminator and the log message sent by Handler arrive concurrently at the Logger. If the log message is processed before the terminate message, the system behaves as expected. However, if the log message is delayed and the terminate message is processed before the log message, the Logger accesses an invalid descriptor and crashes.

Partially Ordered Set of Events and Online Chain Partitioning. PCTCP abstracts messages in the system as partially ordered events. The partial order relation corresponds to the causality relation on the events in the execution. Note that the causality relation between the events of a system depends on the semantics and the guarantees of the system. In our example, the log and terminate messages depend on the request message, as they are created in response to request. They are concurrent to each other since they are sent to different receivers and they will be processed concurrently.

PCTCP intercepts all events in a running system and maintains the poset of events in an execution online as well as the current schedule. In each step, PCTCP selects an unexecuted event and schedules it. The execution of this event can cause further events in the system. These are intercepted and added to the partial order. The partial order of events is maintained as a chain decomposition. That is, the elements of the partial order are partitioned into a set of chains. Each chain is a linear ordering of events according to the partial order. When a new event is intercepted, it is added to one of these chains (or put in a new chain by itself) by an online chain partitioning algorithm.

The key to the theoretical properties of PCTCP is that the chain decomposition has a small number of chains, bounded by a function of the width of the partial order. PCTCP forms chains of events based on the causal dependency relation between them. It inserts the concurrently executable events into different chains, which bounds the number of chains to a function of the number of concurrently executable events. Therefore, the theoretical bug detection guarantee of PCTCP is not tied to the number of nodes in a system (some of which may be inactive in some parts of the execution) but to the width of the partial order, i.e., the maximum number of simultaneously executable events. (Note that since the partial order is revealed one element at a time, the chain partition is constructed online. Thus, while there always exists a chain partition whose size is the width of the ordering, we may not achieve this bound.) PCTCP uses an online chain partitioning algorithm [Agarwal and Garg 2007] that guarantees that we use at most \(O(w^2)\) chains, where \(w\) is the (unknown) width of the partial order.

PCTCP Algorithm. PCTCP is a randomized scheduling algorithm for distributed programs. It takes as input the maximum number \(n\) of messages to be scheduled and a parameter \(d\) which determines the bug depth to be explored. It guarantees a lower bound on the probability of covering every execution of depth \(d\) based on \(n, d,\) and the width of the underlying partial order. PCTCP maintains a priority list of chains partitioning the partial order of events, where lower numbers

---

\(^1\)The example is adapted from Tasharofi et al. [2013], and it shows a simplified version of a bug found in a performance testing tool called Gatling [2018].
Randomized Testing of Distributed Systems with Probabilistic Guarantees

**Input:** number of events $n$, depth bound $d$

**Data:**
- $\text{chains}$ // chain partition of events, the first $d-1$ positions are initialized to null
- $\text{eventsAdded}$ // number of events already added, initially 0
- $\text{priorityChangePt}$ // vector of $d-1$ distinct integers, initialized randomly between 1 and $n$
- $\text{schedule}$ // the current execution

**Procedure** `addNewEvent(e)`
1. insert $e$ into the poset using online chain partitioning (Alg. 2)
2. if a new chain is created then
3. insert the new chain into a random position between $d$ and $|\text{chains}|$ in $\text{chains}$
4. increment $\text{eventsAdded}$
5. if $\exists j : \text{eventsAdded} = \text{priorityChangePt}[j]$ then
6. // assign a label to the event
7. $e.\text{label} \leftarrow j$

**Procedure** `scheduleNextEvent()`
8. while $\exists j : \text{chains}[j] = \alpha \cdot e \cdot \alpha' \land e.\text{isEnabled} \land e.\text{hasLabel} \land e.\text{label} \neq j$ do
9. // we are at a priority change point
10. // note that $\text{chains}[e.\text{label}] = \text{null}$ due to the labels being distinct
11. swap $\text{chains}[j]$ and $\text{chains}[e.\text{label}]$
12. // select an enabled event from the chain with the highest priority
13. if $\exists j : \text{chains}[j] = \alpha \cdot e \cdot \alpha' \land e.\text{isEnabled}$ then
14. $e \leftarrow$ the event $e$ corresponding to the highest index $j$ s.t. $\text{chains}[j] = \alpha \cdot e \cdot \alpha' \land e.\text{isEnabled}$
15. $\text{schedule}.\text{append}(e)$
16. return $e$

**Algorithm 1:** PCTCP algorithm: adding new events and scheduling the next event from the poset

The algorithm is given in Algorithm 1. It maintains three main data structures. The first is a list of chains of events (called $\text{chains}$), which maintains a chain decomposition of events seen so far, where each chain in the list is assigned a priority. The chain decomposition data structure has two logical parts. The first $d-1$ indices in the list are reserved for chains with reduced priority and are all initialized to null. These positions are populated later during execution when a priority change point is encountered. The rest of the list maintains a prioritized list of chains, and higher indices in the list denote higher priority.

The second data structure, the priority change points $\text{priorityChangePt}$, is a list of $d-1$ distinct integers picked randomly from the range $[1, n]$ at the beginning of the algorithm and used to randomly change the priority of certain chains at run time. The third data structure, $\text{schedule}$, is a schedule of events executed so far.

The algorithm has two main procedures. Procedure `addNewEvent` inserts a new event into the chain decomposition by either inserting it at the end of an existing chain or creating a new chain, according to the online chain decomposition algorithm. If a new chain is created, the new chain is assigned a random priority by inserting it into the chain decomposition at a random position at or after the $d$th position. Additionally, this procedure uses the variable $\text{eventsAdded}$ to keep track of the number of events added to the poset. Once $\text{eventsAdded}$ becomes equal to $\text{priorityChangePt}[j]$ for some $j$, the procedure assigns a label $j$ to the event that is being added to the poset. The label is

used to adjust the priority of the chain containing the event once the event becomes ready to be scheduled.

Procedure scheduleNextEvent selects an enabled event and schedules it by appending it to schedule. We say an event $e$ is enabled (denoted by $e.isEnabled$ in the pseudocode) if it is not yet scheduled, but all of its predecessors have been scheduled. To select an enabled event, the procedure first adjusts the priorities of chains: if there is an enabled event $e$ carrying a label $i$ (the predicate $e.hasLabel$ is true in this case) that is placed in chain $c$ currently in position $j \neq i$ in chains, the procedure moves $c$ to position $i$ in chains. Once the priorities are adjusted, the procedure picks the highest priority chain containing an enabled event, appends this event to schedule, and returns it to be executed. All new events resulting from the execution are added to the chain decomposition (using addNewEvent), and scheduleNextEvent is called again until $n$ events are scheduled.

**PCTCP on the Example.** Figure 1 shows the online construction of the poset in our example for the bug depth parameter $d = 1$. In each step, the event that is executed is crossed out. (i) Initially, the poset contains only the request event in a single chain. The event is scheduled since it is the only event in the system. (ii) Executing request causes two new events: log and terminate. PCTCP extends the chain decomposition with these new events. Since the events are concurrent, the width of the partial order at this point is 2, and the chain partitioning algorithm needs to allocate a new chain. Say that in this example the chain partitioning algorithm inserts log into the same chain with request and terminate into a fresh chain. PCTCP now has two chains to select the next event from: $C_1 = \{\text{request, log}\}$ and $C_2 = \{\text{terminate}\}$, and randomly decides the priority between them. We follow the algorithm first with the ordering that prioritizes $C_1$ over $C_2$. In this case, PCTCP schedules the log event. (iii) Processing log does not lead to more events, so we do not insert any events into the poset. Since all the events in the highest priority chain $C_1$ are executed, the PCTCP scheduler schedules the next event in $C_2$, i.e., the terminate event. (iv) Processing this event creates a flush even sent from the Terminator to the Handler. Since flush depends on terminate, PCTCP extends the chain $C_2$ with flush. (v) Since $C_1$ still does not have any events to schedule, PCTCP continues with $C_2$ and schedules flush. Similar to the previous step, the flushed message is inserted.
into the same chain. (vi) PCTCP schedules the \textit{flushed} event and processes it. No more events are created and the execution ends.

Now assume $C_2$ was given a priority higher than $C_1$. In this case, PCTCP schedules the \textit{terminate}, \textit{flush}, and \textit{flushed} events in this order, before the \textit{log} event. This hits the buggy execution. Since each possible ordering between $C_1$ and $C_2$ is picked with probability $1/2$, we hit the bug with probability $1/2$.

On the other hand, a naive random strategy that uniformly picks one of the enabled events at each step would detect the same bug with probability $1/4$. The random scheduler would have to select \textit{terminate} among the two concurrent events \textit{log} and \textit{terminate}, and then select \textit{flush} among \textit{log} and \textit{flush} to be able to hit the bug. As the length of the chain in which \textit{flush} is inserted increases, the probability of naive random testing to hit the bug decreases exponentially. On the other hand, the probabilities of detecting a bug with PCTCP and naive random testing intuitively get closer to each other as the width of the poset approaches the number of events in the system, i.e., when most of the newly added events are concurrent to each other. In our experimental evaluation in Section 4, we compare the performance of PCTCP and naive random testing on real-world benchmarks.

\textbf{Priority Change Points.} So far, we have ignored the priority change points, because exposing the bug in this example requires a single ordering constraint between two events. Hence, this bug can be detected without changing the initially assigned priorities of the chains. In a more complex setting, the priorities of chains may need to change in order to hit a bug, and this is handled by the priority change points.

Consider a modified version of our example, where the bug is exposed not just with the relative ordering of \textit{flush} $\rightarrow$ \textit{log} events, but also the ordering \textit{flush} $\rightarrow$ \textit{log} $\rightarrow$ \textit{flushed}. Since two additional constraints trigger the bug, the PCTCP scheduler needs to be called with the bug depth parameter $d = 2$, causing it to change chain priorities at one randomly chosen priority change point. If initially $C_2$ has a higher priority than $C_1$ and the priority change point is picked to be 5, then the fifth event added to the poset, i.e. \textit{flushed}, is assigned a label, and after \textit{terminate} and \textit{flush} events are executed, the priority of the chain $C_2$ is reduced. At this point, the \textit{log} event from the currently higher priority chain $C_1$ is scheduled. There are no more events in $C_1$ and PCTCP continues with scheduling \textit{flushed} from $C_2$, hitting the buggy ordering of events. The probability of hitting the bug in this case is $1/10$: the probability that $C_2$ initially has higher priority than $C_1$ is $1/2$, and the desired priority change point is picked with probability $1/5$.

\textbf{Guarantees.} Having generated a $d$-tuple of event labels $(x_0, \ldots, x_{d-1})$, the PCTCP algorithm produces a schedule which "strongly hits" this $d$-tuple. In other words, PCTCP schedules an event labeled $x_i$ at the last possible point in the execution, before the events labeled $x_{i+1}, \ldots, x_{d-1}$. Briefly, the PCTCP algorithm guarantees this by keeping a list of reduced-priority chains which are ordered based on the order of event labels in the $d$-tuple, e.g., the chain which has $x_0$ as the first unexecuted event is inserted as the first chain in the list of reduced-priority chains. When all the chains with initial priorities either finished or were reduced to a lower priority, the reduced-priority chains are executed in an order which preserves the relative order of event labels in the tuple. The crucial theoretical property we can ensure is that every possible $d$-tuple of events is hit with probability at least $1/(w^2n^{d-1})$. The proof of this result appears in the next section.

\section{Online Strong Hitting Schedulers}

\subsection{Preliminaries}

A \textit{partially ordered set} (poset) is a pair $\mathcal{P} = (X, \leq)$ where $X$ is a set and $\leq$ is a partial order (i.e., reflexive, anti-symmetric, and transitive binary relation) on $X$. With slight abuse of notation, we
write $x \in \mathcal{P}$ to denote $x \in X$. We write $\mathcal{P}_1 \subseteq \mathcal{P}_2$ if $X_1 \subseteq X_2$ and $\leq_1$ is a restriction of $\leq_2$; then $\mathcal{P}_1$ is a subposet of $\mathcal{P}_2$. We write $\mathcal{P}_2 \setminus \mathcal{P}_1$ for the partial order on the set $\{x \mid x \in \mathcal{P}_2 \setminus \mathcal{P}_1\}$ and the restriction of $\leq_2$ onto this set. A poset $\mathcal{P}_1$ is a prefix of a poset $\mathcal{P}_2$ if $\mathcal{P}_1 \subseteq \mathcal{P}_2$ and for every $x \in \mathcal{P}_2 \setminus \mathcal{P}_1$ and $y \in \mathcal{P}_1$, we have $x \not\geq y$.

Given a poset $\mathcal{P}$, an element $x \in \mathcal{P}$ is said to be minimal if no other element is smaller than $x$, that is, $\forall y \in \mathcal{P}. y \leq x \implies y = x$. Analogously, $x$ is maximal if no other element is greater than $x$, that is $\forall y \in \mathcal{P}. y \geq x \implies y = x$. We write min $\mathcal{P}$ and max $\mathcal{P}$ to denote the set of minimal and maximal elements of $\mathcal{P}$.

A linearization (or schedule) of a poset $\mathcal{P} = (X, \leq)$ is a total order $\leq_a$ for $X$, such that for all $x, y \in X$, we have $x \leq y$ implies $x \leq_a y$. We often identify schedules with a linear sequence of the elements in $X$. For a schedule $\alpha$, we write $\leq_a$ for the total order induced by $\alpha$. We denote an empty schedule by $\epsilon$.

A poset $\mathcal{P}_2 = \mathcal{P}_1 \cup \{z\}$ is an extension of $\mathcal{P}_1$ with an element $z$ if $\mathcal{P}_1$ is a subposet of $\mathcal{P}_2$. We denote the linear extension of a schedule $\alpha$ with an element $z$ as $\alpha \cdot z$, where $z$ is the greatest element in the linear extension.

Let $\mathcal{P} = (X, \leq)$ be a poset and let $Y \subseteq X$; then $Y$ is a chain if $\forall x_0, x_1 \in Y.(x_0 \leq x_1) \lor (x_1 \leq x_0)$ and an antichain if $\forall x_0, x_1 \in Y.(x_0 \not\leq x_1) \land (x_1 \not\leq x_0)$. The greatest possible size of an antichain in a poset $\mathcal{P}$ is the width of $\mathcal{P}$. Dilworth’s theorem [Dilworth 1950] states that the width $w$ of a finite poset $\mathcal{P}$ is equal to the minimal number of chains that cover $\mathcal{P}$ (i.e., $\mathcal{P}$ can be partitioned into $w$ chains).

### 3.2 Scheduling Games

To formalize our scheduling task, we treat it as a scheduling game played by two players: Program, who reveals a poset of elements in the upgrowing fashion—each element being maximal when it appears—and Scheduler, who schedules the elements while adhering to the partial order.

We describe and analyze two versions of the scheduling game. In the first version, called online hitting for upgrowing posets, Scheduler maintains a family of schedules. In each step Program introduces a single new element, maximal among the old elements, and Scheduler responds by inserting the element into existing schedules without changing the order of the old elements. Scheduler is allowed to duplicate schedules before inserting the element. In this version of the game, Program has full freedom to select the relation between the new and old elements, as long as the new element is maximal at the moment it is introduced.

In the second version of the game we will introduce a structure called scheduling poset. Thus, we call the game online hitting for scheduling posets. In this version, Scheduler maintains a single partial schedule, which it extends by appending elements at its end. Each time Scheduler schedules an additional element $x$, Program may extend the poset with one or more new elements, again in the upgrowing fashion, but with an additional restriction that each new element must be greater than $x$. This is to prevent Program in adding an element that could have been scheduled earlier in the partial schedule.

In both versions of the game, Scheduler’s objective is to construct a strong $d$-hitting family of schedules for a fixed parameter $d \geq 1$, containing as few schedules as possible. The strong $d$-hitting property roughly says that for every $d$-tuple of elements $(x_0, \ldots, x_{d-1})$ in the poset there is a schedule constructed by Scheduler in which $x_i$ appears at the last possible moment before $x_{i+1}, \ldots, x_{d-1}$, that is, if an element $y$ is scheduled after some $x_i$, then it is scheduled there only because $y \geq x_j$ for some $j \geq i$. As we shall see, defining the property rigorously for scheduling posets is rather tricky.

Online hitting for scheduling posets closely corresponds to the execution model of distributed message passing programs. Scheduling an element corresponds to executing a receive event, that
is, choosing a message that can be received and executing its receive handler. As a response, the handler may send new messages, inducing new receive events that can only be executed later, and never before the current receive event. This version of the game also straightforwardly generalizes the execution model of multithreaded programs and the PCT scheduler from Burckhardt et al. [2010]. In this setting, scheduling an element corresponds to executing an instruction, to which the program responds by “making available” the next instruction from the same thread.

Online hitting for upgrowing posets extends the results about (weak) hitting families [Chistikov et al. 2016], as well as the results about online dimension of upgrowing posets [Bosek et al. 2012; Felsner 1997; Kloch 2007]. The (weak) $d$-hitting property requires that for every $d$-tuple $(x_0, \ldots, x_{d-1})$, if there exists a schedule $\alpha$ that schedules the elements in the order $x_0 <_\alpha \ldots <_\alpha x_{d-1}$, then such a schedule also exists in a $d$-hitting family of schedules. As we shall see, every strong $d$-hitting family is a $(d + 1)$-hitting family. In the context of online dimension of upgrowing posets, online dimension can be defined as the smallest size of a 2-hitting family achievable by Scheduler.

### 3.3 Online Hitting for Upgrowing Posets

In the first version of the scheduling game, Program is arbitrarily extending a poset with new elements, and Scheduler is maintaining a strong $d$-hitting family of schedules for the poset in each step, while trying to keep the number of schedules as small as possible. We start by precisely defining the objects constructed by each player.

**Definition 2 (Upgrowing Poset).** An upgrowing poset of size $n$ is a sequence of posets $P = (P_k)_{0 \leq k \leq n}$ that satisfies the following conditions: (1) $P_0 = \emptyset$, (2) $P_{k+1} = P_k \cup \{x\}$ for $k < n$, where $x$ is a new elements such that $x \not\in P_k$, and (3) $x$ is maximal in $P_{k+1}$, that is, for every $y \in P_{k+1}$, $y \not> x$.

**Definition 3 (Strong Hitting Family).** Let $d \geq 1$ be a fixed integer.

- Given a poset $P$, we say a schedule $\alpha$ for $P$ strongly hits a $d$-tuple of elements $(x_0, \ldots, x_{d-1})$ if for every $y \in P$, $y \geq_{\alpha} x_i$ in $\alpha$ for some $i \in \{0, \ldots, d-1\}$ implies $y \geq x_j$ in $P$ for some $j \geq i$.
- We call a set of schedules $F$ a strong $d$-hitting family for $P$ if for every $d$-tuple of elements in $P$ there is a schedule in $F$ that strongly hits it.
- Given an upgrowing poset $P = (P_k)_{0 \leq k \leq n}$ of size $n$, we call a sequence of sets of schedules $F = (F_k)_{0 \leq k \leq n}$ an online strong $d$-hitting family for $P$ if each $F_k$ is a strong $d$-hitting family for $P_k$, and each schedule in $F_{k+1}$ is an extension of a schedule in $F_k$.

**Remark 4.** Strong hitting families are a stronger version of hitting families defined by Chistikov et al. [2016], hence the name. Given a poset $P$ and $d \geq 1$, a $d$-hitting family $F$ is a set of schedules such that every admissible tuple $(x_0, \ldots, x_{d-1})$ in $P$ is hit by a schedule $\alpha \in F$, that is, ordered by $\alpha$ as $x_0 <_\alpha \ldots <_\alpha x_{d-1}$. A tuple is admissible if it is hit by at least one schedule (not necessarily from $F$).

Every strong $d$-hitting family is a $(d + 1)$-hitting family. To show this, let $F$ be a strong $d$-hitting family, and let $(x_0, \ldots, x_d)$ be an admissible $(d + 1)$-tuple. There is a schedule $\alpha \in F$ that strongly hits $(x_1, \ldots, x_d)$. We show that $\alpha$ hits $(x_0, \ldots, x_d)$. Suppose it does not, and let $i, j$ be indices such that $0 \leq i < j \leq d$ and $x_i \geq x_j$. Since $\alpha$ strongly hits $(x_1, \ldots, x_d)$ and $j \geq 1$, there exists $j' \geq j$ such that $x_i \geq x_{j'}$. But then, since $i < j'$, the tuple cannot be hit by any schedule, contradicting the admissibility.

The results of Felsner [1997] and Kloch [2007] (see also the survey by Bosek et al. [2012]) show that there is a close connection between constructing a strong 1-hitting family and an adaptive chain covering of an upgrowing poset. In the adaptive chain covering game, Scheduler constructs a decomposition of the poset into a (not necessarily disjoint) union of chains. That is, whenever
Program adds a new element, Scheduler places it into several chains. Later on, the element may be removed from some chains to better accommodate new elements, but it must always remain in at least one chain. We formalize these requirements in the following definition.

**Definition 5 (Adaptive Chain Covering).** Let \( P = (P_k)_{0 \leq k \leq n} \) be an upgrowing poset of size \( n \) and \( \Lambda \) a set of chain colors. A sequence of functions \( C = (C_k)_{0 \leq k \leq n} \), where \( C_k : P_k \rightarrow 2^\Lambda \), is called an adaptive chain covering for \( P \) if the following conditions hold for all \( 0 \leq k \leq n \) and \( x \in P_k \):

1. \( C_{k+1}(x) \subseteq C_k(x) \) for \( k < n \),
2. \( C_k(x) \neq \emptyset \), and
3. the set \( \{ x \in P_k \mid \lambda \in C_k(x) \} \) is a chain for every \( \lambda \in \Lambda \).

The result of Felsner and Kloch can be stated as follows. Let \( \text{hit}(w) \) be the least integer \( m \) such that Scheduler has a strategy for strong 1-hitting that uses at most \( m \) schedules, and let \( \text{adapt}(w) \) be the least integer \( m \) such that Scheduler has a strategy for adaptive chain covering that uses at most \( m \) chain colors, both on upgrowing posets of width at most \( w \).

**Theorem 6 (Felsner, Kloch).** \( \text{hit}(w) = \text{adapt}(w) \).

Theorem 6 was never explicitly stated by Felsner and Kloch. In fact, they prove a stronger result that \( \dim(w) = \text{adapt}(w) \), where \( \dim(w) \) is the maximal online dimension of upgrowing posets of width at most \( w \). Felsner’s proof of the stronger claim [Felsner 1997] originally had a flaw that was later corrected by Kloch [2007]. In his correction, Kloch isolates strong 1-hitting under the name “property \((\star)\)” as the key property, and essentially shows \( \dim(w) = \text{hit}(w) \) and \( \text{hit}(w) = \text{adapt}(w) \).

We emphasize the latter in Theorem 6 because Felsner and Kloch prove this claim by showing that a strategy for adaptive chain covering can be straightforwardly converted into a strategy for strong 1-hitting and vice versa. Thus, strong 1-hitting and adaptive chain covering are essentially the same problems.

In this paper, we want to bound the number of schedules Scheduler needs to use to achieve strong \( d \)-hitting for arbitrary \( d \geq 1 \). Let \( \text{hit}_d(w, n) \) be the least integer \( m \) such that Scheduler has a strategy for strong \( d \)-hitting that uses at most \( m \) schedules on upgrowing posets of width at most \( w \) and size at most \( n \). Our main result on online hitting for upgrowing posets is the following theorem.

**Theorem 7.** \( \text{hit}_d(w, n) \leq \text{adapt}(w) \cdot (\frac{n}{d-1})(d-1)! \).

**Proof sketch.** Given an upgrowing poset \( P \) of size \( n \) and width at most \( w \), and an adaptive chain covering \( C \) for \( P \) with at most \( m \) colors, the idea is to transform \( C \) step by step into an online strong \( d \)-hitting family \( \mathcal{F} \). The schedules in \( \mathcal{F} \) are indexed by \( d \)-tuples of the form \((\lambda, n_1, \ldots, n_{d-1})\), where \( \lambda \) is a chain color, and \( n_1, \ldots, n_{d-1} \in \{1, \ldots, n\} \) are distinct numbers. The construction ensures that in every step \( k \), for every \( d \)-tuple \((x_0, \ldots, x_{d-1})\) in \( P_k \) there is a schedule index \((\lambda, n_1, \ldots, n_{d-1})\) such that \( \alpha_{\lambda, n_1, \ldots, n_{d-1}} \in \mathcal{F}_k \) strongly hits the tuple. In the index, \( \lambda \) is a chain color such that \( \lambda \in C_k(x_0) \), and \( n_1, \ldots, n_{d-1} \) are steps in which the elements \( x_1, \ldots, x_{d-1} \) were added to the poset. The number of schedule indices is \( m \cdot (\frac{n}{d-1})(d-1)! \), hence the bound on the size of \( \mathcal{F}_k \).

A detailed proof can be found in Sect. A.1. \( \square \)

### 3.4 Online Hitting for Scheduling Posets

In the second version of the scheduling game, Scheduler maintains a single partial schedule of the upgrowing poset presented by Program. Scheduler takes a turn by scheduling an element \( x \) that is minimal among the non-scheduled elements. Program responds by introducing zero or more elements \( y \) such that \( x < y \). New elements are introduced in the upgrowing fashion, that is, each element \( y \) is maximal in the step it is introduced.

There are two key complications in this version of the game. First, there is a mutual dependency of the upgrowing poset constructed by Program and the schedule constructed by Scheduler. And
second, since only one schedule is constructed, it is unclear how to define strong hitting families. We deal with the first complication first: we upgrade the upgrowing poset to a new structure called scheduling poset that encodes all possible ways Scheduler can extend the schedule and Program can extend the poset.

Definition 8 (Scheduling Poset). A scheduling poset is a pair \( SP = (S, \mathcal{P}) \), where \( S \) is a set of schedules, each schedule \( \alpha \in S \) has an associated number \( n_\alpha \geq 0 \), and \( \mathcal{P} = \{ \mathcal{P}_{\alpha, k} \mid \alpha \in S, 0 \leq k \leq n_\alpha \} \) is a set of posets satisfying the following conditions.

Initial conditions:

\[
\begin{align*}
(1) & \quad \epsilon \in S \\
(2) & \quad \mathcal{P}_{\epsilon, 0} = \emptyset
\end{align*}
\]

Extending the schedule:

\[
\begin{align*}
(3) & \quad \alpha \cdot x \in S \text{ if and only if } \alpha \in S \text{ and } x \in \min(\mathcal{P}_{\alpha, n_\alpha} \setminus \alpha) \\
(4) & \quad \mathcal{P}_{\alpha \cdot x, 0} = \mathcal{P}_{\alpha, n_\alpha}
\end{align*}
\]

Extending the poset:

\[
\begin{align*}
(5) & \quad \mathcal{P}_{\alpha, k+1} = \mathcal{P}_{\alpha, k} \cup \{x\}, \text{ where } k < n_\alpha \text{ and } x \notin \mathcal{P}_{\alpha, k} \\
(6) & \quad x \text{ is maximal in } \mathcal{P}_{\alpha, k+1}, \text{ that is, for every } y \in \mathcal{P}_{\alpha, k+1}, y \neq x \\
(7) & \quad x \text{ is greater than the last scheduled element, that is, if } \alpha = \alpha' \cdot y, \text{ then } y < x
\end{align*}
\]

The numbers \( n_\alpha \) in Definition 8 represent the number of new elements Program adds into the poset after the Scheduler extends the schedule to \( \alpha \). The poset \( \mathcal{P}_{\alpha, k} \) for \( 0 \leq k \leq n_\alpha \) is the poset in the \( k \)-th step after scheduling \( \alpha \). We will also be referring to the cumulative step for \( \alpha \) and \( k \): Let \( l \) be the length of \( \alpha \), and let \( a_i \) for \( 0 \leq i \leq l \) denote the prefix of \( \alpha \) of length \( i \). The cumulative step for \( \alpha \) and \( k \) is the number \( t = n_{a_0} + \ldots + n_{a_l} + k \). It is not difficult to see that a scheduling poset in cumulative step \( t \) has precisely \( t \) elements.

In online testing of upgrowing posets, we need to extend the definition of adaptive chain covering to posets.

Definition 9 (Adaptive Chain Covering for Scheduling Posets). Let \( \Lambda \) be a set of chain colors, and \( SP = (S, \mathcal{P}) \) a scheduling poset. A set of functions \( C = \{ C_{\alpha, k} : \mathcal{P}_{\alpha, k} \to 2^\Lambda \mid \alpha \in S, 0 \leq k \leq n_\alpha \} \) is called an adaptive chain covering for \( SP \) if the following conditions hold for all \( \alpha \in S \), \( 0 \leq k \leq n_\alpha \), and \( x \in \mathcal{P}_{\alpha, k} \): (1) \( C_{\alpha, k+1}(x) \subseteq C_{\alpha, k}(x) \) if \( k < n_\alpha \), (2) \( C_{\alpha, y, 0}(x) = C_{\alpha, n_\alpha}(x) \), (3) \( C_{\alpha, k}(x) \neq \emptyset \), and (4) the set \( \{ x \in \mathcal{P}_{\alpha, k} \mid \lambda \in C_{\alpha, k}(x) \} \) is a chain for every \( \lambda \in \Lambda \).

By defining scheduling posets, we have solved the first of the two complications mentioned earlier. We have also solved part of the second complication: given a scheduling poset \( SP = (S, \mathcal{P}) \), a strong \( d \)-hitting family will be some subset \( F \subseteq S \). But how do we define the strong \( d \)-hitting property? Note that we cannot quantify over \( d \)-tuples \((x_0, \ldots, x_{d-1})\), because as soon as we fix a domain for some \( d \)-tuple, say \( \mathcal{P}_{\alpha, k} \), we have fixed the schedule \( \alpha \), and this schedule does not necessarily hit the tuple. We deal with this complication by employing a trick from Burckhardt et al. [2010]: instead of tuples, we quantify over auxiliary functions called labelings that indirectly select the tuples for us.

Definition 10 (Labeling). Let \( SP = (S, \mathcal{P}) \) be a scheduling poset, \( d \geq 1 \) a fixed integer, and \( L = \{ x_0, \ldots, x_{d-1} \} \) an ordered set of labels. A \( d \)-labeling for \( SP \) is a set of partial functions \( \mathcal{L} = \{ L_{\alpha, k} : L \to \mathcal{P}_{\alpha, k} \mid \alpha \in S, 0 \leq k \leq n_\alpha \} \) satisfying the following conditions for every \( \alpha \in S \) and \( 0 \leq k \leq n_\alpha \):

\[
\begin{align*}
(1) & \quad \mathcal{L}_{\alpha, k} \text{ is injective}
\end{align*}
\]
(2) \( L_{a,x,0} = L_{a,n_a} \) for \( x = \min(\mathcal{P}_{a,n_a} \setminus \alpha) \), and if \( k < n_a \), then \( \text{dom}(L_{a,k}) \subseteq \text{dom}(L_{a,k+1}) \) and \( L_{a,k+1}(x_1) = L_{a,k}(x_1) \) for every \( x_1 \in \text{dom}(L_{a,k}) \).

(3) If \( k < n_a \) and \( \mathcal{P}_{a,k+1} = \mathcal{P}_{a,k} \cup \{x\} \), then \( \text{dom}(L_{a,k+1}) \setminus \text{dom}(L_{a,k}) \) contains at most one label \( x_1 \), for which \( L_{a,k+1}(x_1) = x \).

(4) For every adaptive chain covering \( C \) for \( \mathcal{P} \), there exists a chain color \( \lambda \) such that \( \lambda \in C_{a,k}(L_{a,k}(x_0)) \) for every schedule \( \alpha \) and step \( 0 \leq k \leq n_a \) in which \( x_0 \in \text{dom}(L_{a,k}) \).

When \( \alpha \) and \( k \) are clear from the context, we usually write \( x_1 \) instead of \( L_{a,k}(x_1) \).

Intuitively, the conditions in Definition 10 require the labels to be assigned to distinct elements; they require them to be stable, and only assigned to newly added elements. Condition 4 requires that for every adaptive chain covering there is a chain that contains \( x_0 \) irrespective of the way we schedule the elements.

**Definition 11 (Strong Hitting Family for Scheduling Posets).** Let \( \mathcal{P} = (S, \mathcal{P}) \) be a scheduling poset, \( d \geq 1 \) a fixed integer, \( L \) a \( d \)-labeling for \( \mathcal{P} \), and \( \alpha \in S \) a schedule.

- We say \( \alpha \) partially hits \( L_{a,k} \) for \( 0 \leq k \leq n_a \) if for every \( x_1 \in \text{dom}(L_{a,k}) \) scheduled by \( \alpha \) and every \( x \in \mathcal{P}_{a,k} \) such that either \( x \geq_x x_1 \) or \( x \) is not scheduled, there exists \( x_j \in \text{dom}(L_{a,k}) \) with \( j \geq i \) such that \( x \geq_x x_j \) in \( \mathcal{P}_{a,k} \). We say \( \alpha \) partially hits \( L \) if it partially hits \( L_{a,k} \) for every \( 0 \leq k \leq n_a \).

- If \( \alpha \) is complete, that is, it schedules the whole \( \mathcal{P}_{a,n_a} \), and it partially hits \( L \), we say it strongly hits \( L \).

- We say \( L \) is complete if for each of its strongly hitting schedules \( \alpha \) all labels are assigned in \( \mathcal{P}_{a,n_a} \), that is, \( L_{a,n_a} \) is a total function.

- A set of complete schedules \( \mathcal{F} \subseteq S \) is a strong \( d \)-hitting family for \( \mathcal{P} \) if for every complete \( d \)-labeling \( L \) for \( \mathcal{P} \) there is a schedule \( \alpha \in \mathcal{F} \) that strongly hits \( L \).

The following lemma shows that in order to maintain partial hitting, it suffices for Scheduler to preserve the property on their move. In other words, Program cannot break the property by cleverly introducing a new element.

**Lemma 12.** Let \( \mathcal{P} = (S, \mathcal{P}) \) be a scheduling poset, \( d \geq 1 \) a fixed integer, \( L \) a \( d \)-labeling for \( \mathcal{P} \), and \( \alpha \in S \) a schedule. The following statements are equivalent:

1. \( \alpha \) partially hits \( L \),
2. \( \alpha \) partially hits \( L_{a,k} \) for some \( 0 \leq k \leq n_a \),
3. \( \alpha \) partially hits \( L_{a,0} \).

**Proof.** Clearly (1) implies (2). In order to show that (2) implies (3), assume \( \alpha \) partially hits \( L_{a,k} \) for some \( k > 0 \). We show \( \alpha \) partially hits \( L_{a,k-1} \) and conclude by downward induction on \( k \). Let \( \mathcal{P}_{a,k} = \mathcal{P}_{a,k-1} \cup \{x\} \), let \( x_i \in \text{dom}(L_{a,k-1}) \) be an element scheduled by \( \alpha \), and let \( y \in \mathcal{P}_{a,k-1} \) be an element such that either \( y \geq_x x_i \) or \( y \) is not scheduled. Since \( \alpha \) partially hits \( L_{a,k} \), there exists \( x_j \in \text{dom}(L_{a,k}) \) with \( j \geq i \) such that \( y \geq_x x_j \). If \( x_j \in \mathcal{P}_{a,k-1} \), we are done. Suppose \( x_j \notin \mathcal{P}_{a,k-1} \); then \( x_j = x \). But then \( x < y \) in \( \mathcal{P}_{a,k} \), contradicting the maximality of \( x \).

We show that (3) implies (1) by (upward) induction on \( k \). The statement (3) is the base case. Assume \( \alpha \) partially hits \( L_{a,k} \) for some \( k < n_a \), let \( \mathcal{P}_{a,k+1} = \mathcal{P}_{a,k} \cup \{x\} \), let \( x_i \in \text{dom}(L_{a,k+1}) \) be an element scheduled by \( \alpha \), and let \( y \in \mathcal{P}_{a,k+1} \) be an element such that either \( y \geq_x x_i \) or \( y \) is not scheduled. Note that \( x_i \in \mathcal{P}_{a,k} \). If \( y \in \mathcal{P}_{a,k} \), we are done; otherwise \( y = x \). Since \( \alpha \) schedules \( x_i \), we know that \( \alpha = \alpha' \cdot z \) for some \( z \in \mathcal{P}_{a,k} \), and moreover \( z \geq_x x_i \). By the induction hypothesis, there exists \( x_j \in \text{dom}(L_{a,k}) \) with \( j \geq i \) such that \( z \geq_x x_j \). Since \( y = x > z \), by transitivity we have \( y \geq x_j \). \( \square \)
In contrast to Theorem 7, which states the result for online hitting for upgrowing posets using quantities $\text{hit}_d(w, n)$ and $\text{adapt}(w)$, we state the result in this subsection in a more operational way. To that end, we define an auxiliary notion of schedule indices: Given a scheduling poset $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ of size at most $n$, a fixed integer $d \geq 1$, and an adaptive chain covering $C$ for $\mathcal{SP}$ with the set of chain colors $\Lambda$, we say a schedule index is a $d$-tuple of the form $(\lambda, n_1, \ldots, n_{d-1})$, where $\lambda \in \Lambda$ is a chain color, and $n_i \in \{1, \ldots, n\}$ for $1 \leq i \leq n$ are distinct numbers. Intuitively, given a labeling $L$ and a schedule $\alpha$, the numbers $n_i$ represent the cumulative steps in which $L$ assigns new elements in the poset, and $\lambda$ represents the color of a chain containing $x_0$. If $L$ assigns labels in this way by following $\alpha$, we say $L$ conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on $\alpha$.

**Lemma 13.** Let $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ be a scheduling poset of size at most $n$, $d \geq 1$ a fixed integer, and $C$ an adaptive chain covering for $\mathcal{SP}$. For every schedule index $(\lambda, n_1, \ldots, n_{d-1})$ there is a schedule $\alpha = \alpha_{\lambda, n_1, \ldots, n_{d-1}}$ such that $\alpha$ strongly hits every complete $d$-labeling that conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on $\alpha$.

**Proof.** Let $(\lambda, n_1, \ldots, n_{d-1})$ be a schedule index. We construct the schedule $\alpha = \alpha_{\lambda, n_1, \ldots, n_{d-1}}$ inductively. The invariant maintained in the construction is that $\alpha$ partially hits every labeling that conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on $\alpha$.

1. **Base case:** $\alpha = \epsilon$. Since $\mathcal{P}_{\epsilon, 0} = \emptyset$, $\epsilon$ trivially hits $\mathcal{L}_{\epsilon, 0}$ for any labeling $\mathcal{L}$. By Lemma 12, $\epsilon$ partially hits every labeling $\mathcal{L}$.

2. **Induction step.** Assume we have constructed some $\alpha = \alpha_{\lambda, n_1, \ldots, n_{d-1}}$ that satisfies the invariant. If all elements have been scheduled, we are done. Otherwise, we show how to select $x \in \min(\mathcal{P}_{\alpha, n_i}) \setminus \alpha$ to extend $\alpha$ into $\alpha' = \alpha \cdot x$ without breaking the invariant. There are three cases:

   1. There exists $x \in \min(\mathcal{P}_{\alpha, n_i}) \setminus \alpha$ such that $\lambda \notin C_{\alpha, n_i}(x)$ and $x$ was not added in cumulative step $n_i$ for any $1 \leq i < d$. We extend $\alpha$ with any such $x$.

   2. Otherwise, there exists $x \in \min(\mathcal{P}_{\alpha, n_i}) \setminus \alpha$ such that $\lambda \in C_{\alpha, n_i}(x)$ and $x$ was not added in cumulative step $n_i$ for any $1 \leq i < d$. We extend $\alpha$ with any such $x$.

   3. Otherwise, every $x \in \min(\mathcal{P}_{\alpha, n_i}) \setminus \alpha$ was added in cumulative step $n_i$ for some $1 \leq i < d$. We extend $\alpha$ with $x$ added in step $n_i$ for the least index $i$.

Let $\mathcal{L}$ be a labeling conforming to $(\lambda, n_1, \ldots, n_{d-1})$ on $\alpha'$. Since it also conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on $\alpha$, it is partially hit by $\alpha$. We may have broken the partial hitting property if we have extended the schedule with $x_0$ in the second case, or with $x_i$ for $1 \leq i < d$ in the third case.

In the second case, let $y$ be some element that is not yet scheduled, and let $y' \in \min(\mathcal{P}_{\alpha, n_i}) \setminus \alpha$ be an element such that $y' \leq y$ (such $y'$ always exists). Since we are in the second case, either $\lambda \in C_{\alpha, n_i}(y')$, implying $y \geq y' \geq x_0$, or $y' = x_j$ for some $1 \leq j < d$. In either case, $y \geq x_j$ for some $0 \leq j < d$.

In the third case, let $y$ be some element that is not yet scheduled, and again, let $y' \in \min(\mathcal{P}_{\alpha, n_i}) \setminus \alpha$ be an element such that $y' \leq y$. Since we are in the third case, $y' = x_j$ for some $1 \leq j < d$. Since we have extended $\alpha$ with $x_j$ having the least index $i$, we have $j \geq i$.

This shows that $\alpha'$ partially hits $\mathcal{L}_{\alpha', 0}$. By Lemma 12, $\alpha'$ partially hits $\mathcal{L}$.\qed

**Lemma 14.** Let $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ be a scheduling poset of size at most $n$, $d \geq 1$ a fixed integer, and $C$ an adaptive chain covering for $\mathcal{SP}$. For every complete $d$-labeling $\mathcal{L}$ there is a schedule index $(\lambda, n_1, \ldots, n_{d-1})$ such that $\mathcal{L}$ conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on $\alpha_{\lambda, n_1, \ldots, n_{d-1}}$.

**Proof.** Let $\mathcal{L}$ be a complete $d$-labeling, and let $\lambda$ be a chain color such that $\lambda \in C_{\alpha, k}(x_0)$ for every schedule $\alpha$ and step $0 \leq k \leq n_a$ in which $x_0$ is defined. Note that we can repeat the construction from the proof of Lemma 13 with the knowledge of $\lambda$ and the actual elements $x_1, \ldots, x_{d-1}$ selected by $\mathcal{L}$ instead of the knowledge of the schedule index. During the construction, we take note of...
cumulative steps $n_i$ in which $L$ assigns labels $x_i$. By the invariant, the schedule $\alpha$ obtained at the end strongly hits $L$. Since $L$ is complete, all labels have been assigned at the end, hence $C_{\alpha,n_i}(x_0)$ is a well-defined set of chain colors containing $\lambda$ and $n_i$ are well-defined for all $1 \leq i < d$. By construction, $\alpha = \alpha_{\lambda,n_i,...,n_{d-1}}$ and $L$ conforms to $(\lambda,n_1,\ldots,n_{d-1})$ on $\alpha$.

**Theorem 15.** Let $SP = (S,P)$ be a scheduling poset of size at most $n$, $d \geq 1$ a fixed integer, and $C$ an adaptive chain covering for $SP$. The set $$\mathcal{F} = \{\alpha_{\lambda,n_1,...,n_{d-1}} \mid \lambda,n_1,...,n_{d-1}\}$$ is a schedule index for $SP$. If $C$ uses $m$ chain colors, then $\mathcal{F}$ has size at most $m\binom{n}{d-1}(d-1)!$.

**Proof.** Let $L$ be a complete $d$-labeling for $SP$. By Lemma 14, there exists a schedule index $(\lambda,n_1,\ldots,n_{d-1})$ such that $L$ conforms to $(\lambda,n_1,\ldots,n_{d-1})$ on $\alpha = \alpha_{\lambda,n_1,...,n_{d-1}}$. By Lemma 13, $\alpha$ strongly hits $L$, and by definition, $\alpha \in \mathcal{F}$. Finally, the size of $\mathcal{F}$ is bounded by the total number of schedule indices.

### 3.5 Online Chain Partitioning

Our two main results, Theorem 7 and Theorem 15, show that Scheduler can construct strong hitting families of bounded size provided they have a strategy for adaptive chain covering. Adaptive chain covering is essentially an online decomposition of an upgrowing poset into a (not necessarily disjoint) union of chains. In particular, any strategy for online chain partitioning, which decomposes the poset into a disjoint union of chains, is a strategy for adaptive chain covering.

By Dilworth’s theorem, the optimal chain partition of a poset of width $w$ uses $w$ chains. In the online setting, the optimal partition may not be achievable. As shown by Felsner [1997], for upgrowing posets of width at most $w$, Scheduler always has a strategy for chain partitioning that uses at most $\binom{w+1}{2}$ chains, and Program can force Scheduler to use $\binom{w+1}{2}$ chains. This bound translates into an upper bound for adapt$(w)$, the minimal number of chains needed for adaptive chain covering over all upgrowing posets of width at most $w$. By plugging the bound into our main theorems, we can bound the size of strong hitting families.

**Corollary 16.** Given $d \geq 1$, for any upgrowing or scheduling poset of width at most $w$ and size at most $n$, there is a strong $d$-hitting family of schedules of size at most $\binom{w+1}{2}\binom{n}{d-1}(d-1)! \leq w^2n^{d-1}$.

There is a surprisingly elegant algorithm for online chain partitioning given by Agarwal and Garg [2007], given in Algorithm 2. The algorithm is optimal in the sense that it uses at most $\binom{w+1}{2}$ chains for upgrowing posets of width at most $w$. The algorithm maintains $w$ sets of chains.

**Algorithm 2:** Chain partitioning algorithm: adding a new element into a chain

**Input:** A new element $y$

**Data:** Sets of chains $B_1,\ldots,B_w$

**Invariant:** $\forall i : 1 \leq i \leq w \implies |B_i| \leq i$.

**Invariant:** $\forall i : 1 \leq i \leq w \implies \text{Last}(B_i) := \{x \mid \alpha \cdot x \in B_i\}$ is an antichain

**Procedure** `addNewElement(y)`

```plaintext
for i = 1 to w do
  if (\exists \alpha' \cdot x \in B_i : x < y) or |B_i| < i then
    \alpha \leftarrow \alpha' \cdot x if it exists, or a new empty chain otherwise
    \alpha \leftarrow \alpha \cdot y
  if i > 1 then
    (B_{i-1},B_i) \leftarrow (B_i \setminus \{\alpha\}, B_{i-1} \cup \{\alpha\})
  return
```

B₁, ..., Bₜ such that each Bᵢ contains at most i chains. Moreover, define Last(Bᵢ) := {x | α ∙ x ∈ Bᵢ}. The algorithm maintains the invariant that Last(Bᵢ) is an antichain for every 1 ≤ i ≤ t. Let y be the new maximal element added to the poset. The algorithm finds the least index i such that y is comparable with some x ∈ Last(Bᵢ) or Bᵢ has less than i chains. (Such i exists, otherwise Last(Bᵢ) ∪ {y} is an antichain of size t + 1.) Let α = α' ∙ x if y is comparable with x for α' ∙ x ∈ Bᵢ, otherwise let α = e. The algorithm extends α to α ∙ y in Bᵢ. If i > 1, the algorithm swaps the chains in Bᵢ₋₁ and Bᵢ so that in the next step Bᵢ’₋₁ = Bᵢ \ {α ∙ y} and Bᵢ’ = Bᵢ₋₁ ∪ {α ∙ y}. It is not difficult to see that the invariant of the algorithm is preserved [Agarwal and Garg 2007]. Notice that the width of the poset w need not be known upfront. As the algorithm inserts elements to the poset, it creates new chains as needed. By means of the invariant enforcing each Last(Bᵢ) to be an antichain, the largest set of chains Bᵢ is at most of size as the width of the poset, w.

Note that since adaptive chain covering allows decompositions of the poset into non-disjoint chains, it is possible that there exist strategies which use fewer than \( \binom{w+1}{2} \) chains. Unfortunately, no better strategies than online chain partitioning are currently known [Bosek et al. 2012]. However, in case of future progress on adaptive chain covering, any new bounds on adapt(w) will automatically translate into new bounds on the size of online strong hitting families.

### 3.6 PCTCP—PCT with Chain Partitioning

We now relate our algorithm PCT—Probabilistic Concurrency Testing with Chain Partitioning, introduced as Algorithm 1 and described informally in Sec. 2, to the results discussed in Sec. 3.4 and Sec 3.5. PCTCP incorporates Agarwal and Garg’s online chain partitioning algorithm into the construction of strong hitting families for scheduling posets. However, instead of constructing the whole strong hitting family, it selects a scheduling index uniformly at random and constructs only the corresponding schedule. Therefore, it provides a bound on the probability of hitting a bug of depth d:

**Corollary 17.** Given a scheduling poset \( SP \) of size at most n and width at most w, a schedule constructed by PCTCP strongly hits a \( d \)-complete labeling for \( SP \) with probability at least \( 1/(w^n d^{d-1}) \).

To pick a scheduling index uniformly at random, PCTCP uses a priority-based randomized scheduler similar to PCT—the randomized scheduler for multithreaded programs by Burckhardt et al. [2010]. The PCTCP scheduler assigns a priority uniformly at random to each chain as it is constructed and added to the partition on-the-fly. It then, at each step of computation, schedules an enabled event from a chain with the highest priority. An event is enabled if it is not scheduled and all of its predecessors have been scheduled. The priority of a chain may change during the execution when it passes a priority change point. These points are steps in an execution with associated priorities which are lower than the priorities assigned to chains initially. When the execution reaches a priority change point, the scheduler adjusts the priority of the corresponding chain to the priority associated with the change point. More specifically, given inputs \( d \) and \( n \), PCTCP assigns priority values \( d, d + 1, \ldots, d + w^2 - 1 \) to chains which are constructed dynamically (we can have up to \( w^2 \) chains). It also picks initially \( d - 1 \) random priority change points \( n₁, \ldots, n_{d-1} \) in the range \([1, n]\), where each \( nᵢ \) has an associated priority value of \( i \). Combining the initial priority assignments and the priority change points, PCTCP generates a schedule index \((\lambda, n₁, \ldots, n_{d-1})\), where \( \lambda \) is the chain with the lowest initial priority.

We argue that PCTCP is a natural generalization of PCT for multithreaded programs [Burckhardt et al. 2010]. The main difference is that PCTCP uses a chain partition constructed on-the-fly, and PCT uses a chain partition provided by threads. Hence the difference in the probabilistic guarantee is \( 1/(w^n d^{d-1}) \) for PCTCP versus \( 1/(kn^{d-1}) \) for PCT on a program with \( k \) threads.
Another difference between the two is in the phrasing of the objective of the generated schedule. PCTCP’s objective is to “strongly hit a labeling,” whereas PCT’s objective is to “satisfy a directive that guarantees a bug.” PCT’s directive for a given \( d \geq 1 \) is a tuple \( D = (L, A_0, x_0, \ldots, A_{d-1}, x_{d-1}) \), where \( L \) is a labeling whose set of labels \( L \) can contain labels other than \( x_0, \ldots, x_{d-1} \), and each \( A_i \subseteq L \) is a set of labels. A schedule \( \alpha \) satisfies a directive if it schedules all \( a \in A_i \) before \( x_i \), for every \( 0 \leq i < d \), and schedules \( x_0, \ldots, x_{d-1} \) in the order \( x_0 <_a \ldots <_a x_{d-1} \). Thus, a directive represents a set of additional ordering constraints a schedule should satisfy in order to expose a bug. The constraints are implicitly assumed to be consistent with the program’s partial order.

It is not difficult to see that the strong hitting property subsumes PCT’s directives. Let \( D = (L, A_0, x_0, \ldots, A_{d-1}, x_{d-1}) \) be a directive, and assume a schedule \( \alpha \) strongly hits \((x_0, \ldots, x_{d-1})\). We first show that the elements \( x_0, \ldots, x_{d-1} \) are ordered as \( x_0 <_a \ldots <_a x_{d-1} \). Suppose they are not, that is, suppose \( x_j \geq_a x_i \) for some \( j < i \). By the strong hitting property, there exists \( j' \geq i \) such that \( x_j \geq x_{j'} \). But then, since \( j < j' \), the directive is inconsistent with the partial order. We conclude that \( \alpha \) correctly schedules \( x_0 <_a \ldots <_a x_{d-1} \). Suppose now that \( a \geq_a x_i \) for some \( 0 \leq i < d \) and \( a \in A_j \). By the strong hitting property, there exists \( j \geq i \) such that \( a \geq x_j \). Again, since \( i \leq j \), according to the directive the element \( a \) should be scheduled before \( x_j \), which is inconsistent with the partial order. We conclude all \( a \in A_i \) are scheduled before \( x_i \) for every \( 0 \leq i < d \). Thus, \( \alpha \) satisfies the directive \( D \).

4 EXPERIMENTAL EVALUATION

4.1 P# Benchmarks

We implemented PCTCP\(^2\) to randomly test distributed applications written in Microsoft’s P# framework\(^3\) for building asynchronous message passing systems [Deligiannis et al. 2015, 2016; Mudduluru et al. 2017]. A P# program consists of a number of state machines that communicate by sending and receiving messages. Each P# machine executes a message handling loop and runs in parallel with other machines. Handling of a message can result in a state transition, creating new machines, sending messages to other machines, or updating local fields. The systematic testing engine of P# instruments a program at synchronization points, which are send, create-machine, and receive events. Upon execution of one of these events, the P# runtime calls the scheduler, which blocks the current machine and releases a possibly different machine for execution. Therefore, a machine may be interrupted in the middle of handling an incoming message in case the handling causes sending a new message or creating a new machine.

The original P# runtime does not keep track of causal dependencies between events, and thus does not have an explicit notion of chains. At synchronization points, the scheduler only knows the set of currently executing machines, and chooses one of them as the next one to schedule. The choice is determined by the scheduling strategy; among others, the implemented strategies include the “random walk,” which selects the next machine uniformly at random, and “prioritized strategy,” which randomly selects \( d \) scheduling points during execution order and prioritizes them to make sure they are ordered in a particular way. The latter strategy is similar to PCT, and it is called PCT in the P# source code, but without the notion of chains it does not provide the same probabilistic guarantee. Therefore we call it “prioritized strategy” to avoid confusion.

In order to keep track of causal dependencies, we implemented our own version of the P# runtime called “PCTCP runtime”. Additionally, we simplified the scheduler to only schedule the receive events. In fact, the underlying concurrency model of PCTCP runtime is coarser as it introduces fewer synchronization points. Therefore, it may miss behaviors arising from interleavings of different

\(^2\)The source code is available at https://gitlab.mpi-sws.org/fnksic/PSharp/tree/PCTCP.

\(^3\)https://github.com/p-org/PSharp
Randomized Testing of Distributed Systems with Probabilistic Guarantees

Table 1. Characteristics of benchmarks (LOC includes comments and blank lines, “Type of bug” refers to known bugs)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#LOC</th>
<th>#Machine type</th>
<th>#Message type</th>
<th>Type of bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundedAsync</td>
<td>288</td>
<td>2</td>
<td>7</td>
<td>safety</td>
</tr>
<tr>
<td>ChainReplication</td>
<td>1,562</td>
<td>5</td>
<td>46</td>
<td>safety</td>
</tr>
<tr>
<td>Raft</td>
<td>1,302</td>
<td>5</td>
<td>29</td>
<td>safety</td>
</tr>
<tr>
<td>Chord</td>
<td>917</td>
<td>3</td>
<td>22</td>
<td>liveness</td>
</tr>
<tr>
<td>ReplicatingStorage</td>
<td>978</td>
<td>7</td>
<td>37</td>
<td>liveness</td>
</tr>
<tr>
<td>FailureDetector</td>
<td>674</td>
<td>4</td>
<td>19</td>
<td>both</td>
</tr>
<tr>
<td>TwoPhaseCommit</td>
<td>725</td>
<td>5</td>
<td>29</td>
<td>-</td>
</tr>
<tr>
<td>MultiPaxos</td>
<td>1,095</td>
<td>5</td>
<td>26</td>
<td>-</td>
</tr>
<tr>
<td>CacheCoherence</td>
<td>420</td>
<td>3</td>
<td>17</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2. Results of applying PCTCP to P# benchmarks including number of buggy schedules, average number of computed chains, maximum number of produced messages and the running time.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Event labels (d)</th>
<th>#Runs</th>
<th>%Buggy</th>
<th>Avg #chains</th>
<th>Max #msgs (n)</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundedAsync</td>
<td>1</td>
<td>10,000</td>
<td>98.97</td>
<td>12</td>
<td>128</td>
<td>71.48</td>
</tr>
<tr>
<td>ChainReplication</td>
<td>5</td>
<td>1,000</td>
<td>12.60</td>
<td>18</td>
<td>362</td>
<td>635.18</td>
</tr>
<tr>
<td>Raft</td>
<td>1</td>
<td>1,000</td>
<td>0.20</td>
<td>37</td>
<td>590</td>
<td>210.53</td>
</tr>
<tr>
<td>Chord</td>
<td>1</td>
<td>1,000</td>
<td>6.10</td>
<td>5</td>
<td>62</td>
<td>6.62</td>
</tr>
<tr>
<td>ReplicatingStorage</td>
<td>1</td>
<td>100</td>
<td>11.00</td>
<td>24</td>
<td>899</td>
<td>504.73</td>
</tr>
<tr>
<td>FailureDetector</td>
<td>1</td>
<td>5,000</td>
<td>0.36</td>
<td>27</td>
<td>172</td>
<td>360.87</td>
</tr>
<tr>
<td>TwoPhaseCommit</td>
<td>1</td>
<td>10,000</td>
<td>0.00</td>
<td>9</td>
<td>42</td>
<td>50.63</td>
</tr>
<tr>
<td>MultiPaxos</td>
<td>1</td>
<td>10,000</td>
<td>0.00</td>
<td>32</td>
<td>754</td>
<td>552.82</td>
</tr>
<tr>
<td>CacheCoherence</td>
<td>1</td>
<td>10,000</td>
<td>0.00</td>
<td>6</td>
<td>465</td>
<td>988.96</td>
</tr>
</tbody>
</table>

message handlers. However, it considers all possible reorderings between concurrent events which may lead to a concurrency bug. On top of this simplified concurrency model, we implemented the PCTCP and the random walk scheduling strategies. The random walk strategy selects the next event uniformly at random among the enabled chains.

We evaluate our method on 9 sample implementations of distributed algorithms in the P# framework, which were also used in previous work [Deligiannis et al. 2015; Mudduluru et al. 2017]. Table 1 shows the characteristics of the P# benchmarks including lines of code (LOC), number of machines and message types, and the type (safety or liveness) of the underlying (known) bug(s).

Table 2 shows the result of applying PCTCP on P# benchmarks. For each benchmark we ran PCTCP for a number of times (#Runs) with a given value of parameter d (#Event labels), and measured the average number of computed chains (#Event labels), the maximum number of messages in the partial order (Max #msgs), the number of buggy schedules (%Buggy), and the execution time in seconds (Time(s)).

To choose the value of parameter d, we start with the minimum value of 1 and increment it only if it is not sufficient to catch at least one bug in 10,000 runs. As we can see in Table 2, for ChainReplication we increased the value of d up to 5 to catch the two underlying safety bugs. For
Table 3. Comparison of effectiveness of PCTCP in bug detection with three other methods.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PCTCP runtime</th>
<th>Original P# runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>%Buggy</td>
<td>Time(s)</td>
</tr>
<tr>
<td>BoundedAsync</td>
<td>98.97</td>
<td>71.48</td>
</tr>
<tr>
<td>ChainReplication</td>
<td>12.60</td>
<td>635.18</td>
</tr>
<tr>
<td>Raft</td>
<td>0.2</td>
<td>210.53</td>
</tr>
<tr>
<td>Chord</td>
<td>6.1</td>
<td>6.62</td>
</tr>
<tr>
<td>ReplicatingStorage</td>
<td>11.0</td>
<td>504.73</td>
</tr>
<tr>
<td>FailureDetector</td>
<td>0.36</td>
<td>360.87</td>
</tr>
<tr>
<td>TwoPhaseCommit</td>
<td>0</td>
<td>50.63</td>
</tr>
<tr>
<td>MultiPaxos</td>
<td>0</td>
<td>552.82</td>
</tr>
<tr>
<td>CacheCoherence</td>
<td>0</td>
<td>988.96</td>
</tr>
</tbody>
</table>

the last three benchmarks in this table, we only experimented with \( d = 1 \). These examples do not have bugs discoverable by our reference methods from the P# framework as we will see in the following. Note that the bugs found with some specific value of \( d \) do not necessarily have the bug depth of \( d \).

From Table 2, it can be inferred that the measured probability of catching a bug of depth \( d \) is higher than the guaranteed probability of PCTCP. This is mainly due to the fact that some benchmarks had more than one bug (assertion violation). For example, we observed two different assertion violations in ChainReplication and FailureDetector. Moreover, due to symmetry in these protocols various \( d \)-tuples can result in the same assertion violation.

Table 3 reports the result of comparing the effectiveness of PCTCP in detecting bugs (column 2) with other three reference methods: the random walk strategy in the PCTCP runtime (column 3), the prioritized strategy (column 4) and the random walk (column 5) in the original P# runtime.

Recall that the two runtimes differ in the underlying concurrency model—the PCTCP runtime only schedules message receive events. The number of runs and the parameter \( d \) for each benchmark are the same as in Table 2.

Table 3 shows that both PCTCP and its random walk version are more effective in detecting bugs than the original random walk and prioritized strategies of P#. The assertion violation in the Process machine of BoundedAsync was caught by the PCTCP runtime in nearly all runs (using either strategy). However, both the prioritized and the random walk strategies under the original P# runtime failed to reveal this bug by exploring up to 10,000 schedules. PCTCP found two assertion violations in ChainReplication. The one in ChainReplicationMaster machine was detected with \( d = 1 \); however for detecting the violation in InvariantMonitor we had to increase the value of \( d \) to 5 (Table 2). Both the prioritized (with \( d \leq 5 \)) and the random walk strategies of the P# framework did not find these assertion violations by exploring up to 10,000 schedules (the running times given in Table 3 for this benchmark are for 1,000 runs for the sake of comparison). Only for some specific random seed values, the prioritized scheduler of P# runtime could find the assertion violation in InvariantMonitor in 1 out of 10,000 runs (0.01% buggy schedules). PCTCP also performed more effectively than any strategy under the original P# runtime by detecting one liveness and one safety bug in FailureDetector. Both strategies of the P# runtime failed to detect these bugs in 5,000 runs. The prioritized strategy could find this bug in 70 out of 5,000 when applying \( d = 2 \) and the specific random seed value given in the test suite of P# for this benchmark.
Note that, for ReplicatingStorage, we also compared the P# prioritized scheduler and the PCTCP scheduler based on similar time budgets. However, no bug was caught by the P# prioritized scheduler after exploring 100,000 schedules in 698.65 sec. We did the same comparison for Raft.

As Table 3 shows, PCTCP is effective in detecting bugs in practice. It may not necessarily always outperform random walk, but in contrast it provides theoretical guarantees on finding bugs.

Livelocks. Because the PCTCP scheduler always schedules events from the chain with highest priority, it lacks fairness. This can result in livelocks in message-passing systems. For example, the Raft benchmark may livelock under the PCTCP scheduler due to new messages always being placed in the chain with the highest priority.

To avoid livelocks while searching for safety bugs, PCTCP identifies the chain which causes a livelock by detecting a cycle and using heuristics such as comparing the number of times a chain is consecutively scheduled with a given threshold. It then temporarily disables the identified chain and enables it again as soon as a new message is added to it.

The livelock problem also affects the original PCT algorithm and is discussed in Burckhardt et al. [2010]. A more formal treatment of the issue is left for future work.

4.2 Case Study: Cassandra

In this and the next subsection we evaluate the effectiveness of PCTCP algorithm on two complex real-world systems. The bugs in real-world distributed systems are known to be hard to detect since in addition to message reorderings they often involve other kinds of faults, like node crashes and reboots [Leesatapornwongsa et al. 2016]. We demonstrate that PCTCP can effectively find bugs even in such realistic scenarios.

We start with Cassandra [Lakshman and Malik 2010]—a distributed NoSQL database management system, which provides lightweight transactions based on the Paxos consensus protocol.4 Cassandra’s Paxos protocol implementation in version 2.0.0 [Apache 2012] has a bug CASSANDRA-60235 which exposes in some subtle reorderings on the synchronization messages exchanged between the nodes. The bug is detected in a scenario where the nodes process different client transactions concurrently. In an execution where some commit messages of some transactions arrive at some nodes after the synchronization messages of other transactions, it is possible to commit a transaction more than once. This results in corrupting data and propagating it to the other nodes. The bug is deep and hard to detect as it requires several message reorderings in several transactions and nodes.

We tested Cassandra on our PCTCP implementation6 which we build on top of the SAMC/DMCK [Leesatapornwongsa et al. 2014] model checker. Our PCTCP scheduler collects the distributed system events intercepted by SAMC and partitions them into chains. It selects the next event to be scheduled based on the chain priorities and sends this selected event to SAMC to enforce its execution in the distributed system.

We tested different schedules of a use case scenario with three concurrent client transactions using both PCTCP and a naive random scheduler which randomly selects one of the enabled events in the system. Table 4 shows the parameters and the results of our tests. The first row shows the results for the random walk and each of the next rows shows the PCTCP results for different values of $d$, the number of event labels. The columns list the maximum number of events produced by the benchmark (Max #events), the size of the tuple of event labels (#Event labels), the number of runs (#Runs), the number of buggy schedules (#Buggy) and the total running time of the tests.

4http://cassandra.apache.org
5https://issues.apache.org/jira/browse/CASSANDRA-6023
6The source code is available at https://gitlab.mpi-sws.org/burcu/pctcp-cass/tree/PCTCP.
Table 4. Test parameters and results for the Cassandra system.

<table>
<thead>
<tr>
<th></th>
<th>Max # events</th>
<th>#Event labels</th>
<th>Avg of max #chains</th>
<th>#Runs</th>
<th>#Buggy</th>
<th>Total time (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random walk</td>
<td>54</td>
<td>-</td>
<td>6.97</td>
<td>1000</td>
<td>0</td>
<td>481.95</td>
</tr>
<tr>
<td>PCTCP</td>
<td>54</td>
<td>4</td>
<td>5.65</td>
<td>1000</td>
<td>0</td>
<td>505.73</td>
</tr>
<tr>
<td>PCTCP</td>
<td>54</td>
<td>5</td>
<td>5.73</td>
<td>1000</td>
<td>1</td>
<td>503.81</td>
</tr>
<tr>
<td>PCTCP</td>
<td>54</td>
<td>6</td>
<td>5.80</td>
<td>1000</td>
<td>1</td>
<td>512.00</td>
</tr>
</tbody>
</table>

in minutes (Time(min)). The column Avg of max #chains shows the average of the maximum number of concurrently enabled chains for PCTCP and the average of the maximum number of concurrently enabled events in the naive random tests.

The PCTCP algorithm hits the bug in one of the schedules determined by 5 and 6-tuples of events over 54 events. PCTCP detects the bug with a higher probability (0.1% in this evaluation) than its theoretical guarantee ($1/(w^2n^{d-1})$). This can be explained by several facts: (i) Considering the poset width parameter $w$, we can say that the number of concurrently enabled events (around 7 on average in our benchmark) is lower than the width of the poset (around 10 in our benchmark) in general. During the execution of PCTCP, it is typical to have several chains in which all the events are already executed and the algorithm selects from a smaller number of chains than the poset width. As an example consider a distributed system execution where the highest priority chain has some events pertaining to some protocol communication between a sender and receiver. The PCTCP scheduler moves to another chain (without a reduction in priorities) when all the events of this chain are executed. The first chain gets enabled only after some events in the other chains, e.g., when processing other receivers’ responses in other chains cause the sender to send an event that is inserted into the first chain. (ii) Now let us consider the generation of a $d-1$ tuple of events from $n$ events to characterize a bug. In practice, a bug is not only hit by a specific tuple of events but several tuples lead to the buggy schedule. First, more than one ordering of events in the tuple can expose the same bug, since some events in the tuple might be commutative. For example, a bug hit by a tuple of three events $(a,b,c)$ might require only the relative ordering of $a-b$ and $a-c$, hence hit also by the tuple $(a,c,b)$. In our experiment, the detected 4-tuple of events which leads to a buggy execution of Cassandra has two commutative events. Second, the problematic relative ordering of the events can be generated by tuples of different events. A distributed system bug which is exposed by scheduling an event at a later point of execution can be exposed by reducing the priority of a chain at this event as well as at an earlier event that creates that event. As an example, reducing the priority of a chain at an event pertaining to a certain part of the protocol communication between two nodes delays the execution of the next events of the communication protocol as well.

The naive random approach cannot detect the bug in a total of 1000 schedules which takes around eight hours to run. As shown in Table 4, the maximum number of concurrently enabled events to select from randomly in random testing is higher than the maximum number of concurrently enabled chains in PCTCP on average. Therefore, random testing has a lower probability of hitting the bug by naively selecting the next event among the enabled events.
4.3 Case Study: Zookeeper

Our next case study involves a system called Zookeeper\(^7\). Zookeeper is a distributed key-value store used by large distributed systems for maintaining configuration and naming information, providing distributed synchronization, and for other purposes that arise while coordinating nodes in a distributed setting. Its intended usage requires Zookeeper to provide strong consistency guarantees, which it does by running a distributed consensus algorithm called ZAB—Zookeeper Atomic Broadcast \cite{Junqueira2011}. In our case study we only focus on the first phase of the algorithm—the leader election—and show that in the presence of node crashes and reboots, PCTCP can effectively find executions resulting in multiple node leaders.

In the experiments we are using Zookeeper v3.4.3, even though the most recent stable version was 3.4.11 at the time of writing this text. The reason is that the older version has some known bugs which can be detected by tools like SAMC. In fact, the authors of SAMC were kind enough to provide us with a version of their tool specifically tailored to catching bug ZK-1419\(^8\) in Zookeeper v3.4.3. The bug in question is a liveness bug—the nodes fail to ever elect a leader. In a typical leader election involving 3 nodes, the nodes elect a leader by exchanging 15 to 18 messages. Based on this observation, the authors of SAMC set a bound of 50 events (including messages, but also node crashes and reboots)—any execution that goes beyond this bound is marked as faulty. Following the instructions for reproducing the bug, we ran SAMC in its semantic-aware exploration mode on 3 nodes, allowing 1 node crash and 1 reboot. Out of 174 executions explored in 2,798 seconds, 7 of them were marked as faulty: 4 of them because they reached the bound, but interestingly, 3 of them failed because they resulted in multiple leaders. Note that by itself the situation when multiple nodes believe they are leaders does not constitute a bug in the leader election protocol: the true leader has to be supported by a quorum of followers (cf. bug report ZK-1912\(^9\)). However, such situations may still indicate underlying issues, e.g. the bug ZK-975\(^10\) involves a node that unnecessarily goes into the leading state only to restart the leader election after a delay, thus affecting availability of the system. We set to reproduce the executions leading to multiple leaders, as well as followers following nodes that are not leaders, using PCTCP. We refer to such situations as inconsistencies.

We built our own scheduler called HitMC\(^11\). Like SAMC, HitMC can orchestrate Zookeeper nodes by imposing the order of messages during leader election, and it can crash and reboot nodes. Additionally, HitMC tracks causal dependencies among messages, allowing us to form more general chain decompositions than simple per-node chaining of messages. Unlike SAMC, which knows about commutativity among messages and can thus employ partial-order reduction techniques, HitMC does not have any semantic awareness—it treats messages opaquely and schedules them according to PCTCP.

We model the execution of the system with three kinds of events: node start, node crash, and message. Fig. 2 shows an example of a scheduling poset for Zookeeper with 3 nodes. In the figure, Start($n$), Crash($n$), and Msg($n$, $n'$) designate the start event for node $n$, the crash event for node $n$, and an event for a message from $n$ to $n'$, respectively. For each run of the system we specify a crash budget and a reboot budget. Each node’s initial start event gets a corresponding crash event as a successor. We only effectively allow the execution of these events if they are within the crash budget. Later, each executed crash event gets a node start event as a successor and vice versa, as long as the corresponding budget is still positive. Nodes send messages either automatically after

\(^7\)https://zookeeper.apache.org/

\(^8\)https://issues.apache.org/jira/browse/ZOOKEEPER-1419

\(^9\)https://issues.apache.org/jira/browse/ZOOKEEPER-1912

\(^10\)https://issues.apache.org/jira/browse/ZOOKEEPER-975

\(^11\)The source code is available at https://gitlab.mpi-sws.org/rupak/hitmc.
they are started, or in response to received messages. In the first case, a message is ordered after the last node start event or the last message sent from the same node, and in the second case a message is additionally ordered after the message it is responding to.

By default, HitMC schedules events according to PCTCP. The user specifies parameters $d$, the number of event labels, and $n$, the total number of events. If the number of events in an actual run exceeds $n$, HitMC makes sure to first schedule the $n$ events that were added to the scheduling poset first. It then continues executing in the order specified by the chain priorities. No priority changes happen after the $n$-th event, and the probabilistic guarantee of PCTCP only applies to the first $n$ events. In addition to PCTCP, HitMC provides the “random walk” scheduling strategy, which selects the next event in each step uniformly at random among the enabled events.

Like with SAMC, our experiments consist of executing Zookeeper’s leader election on 3 nodes, with the crash and reboot budget of 1 each. We experiment with the random walk strategy, as well as PCTCP with the parameter $d$ ranging from 1 to 5. In each case, we execute a total of 1,000 random runs. In addition, we execute 1,000 runs of PCTCP with $d = 2$ and no crashes and reboots, in order to see whether we can find inconsistent election results even without node crashes. Following the SAMC’s bound for the number of events within which the leader should be elected, we set the parameter $n$ for the maximal number of events to 50. Unlike SAMC, we do not stop the execution when the maximal number of events is reached. Instead, we let the system run for up to 1,000 events.

The results of the experiments are summarized in Table 5. With the random walk strategy we observe inconsistent election results in 12 out of 1,000 runs. With PCTCP we observe inconsistencies in at least 26 runs for $d = 4$, and up to 42 runs for $d = 5$. Note that the number of inconsistent runs for $d = 5$ amounts to 4.2% of all the runs. Even in the experiment without crashes and reboots, we detect inconsistencies in 34 runs. Except for $d \geq 4$, almost all runs finish within the predicted limit of 50 events. For $d \geq 4$, there are 10 runs that exceed this limit and go up to 63 and 70 events. Additionally, for $d = 4$ we observe a run that failed to terminate even after 1,000 events, possibly exhibiting the non-termination issue ZK-1419. However, it is also possible that the non-termination occurs due to the non-fair nature of our scheduler. Finally, we note that a run takes between 2 and 7 seconds on average, which allows us to execute a set of 1,000 runs in 30 to 120 minutes.

5 DISCUSSION AND OTHER RELATED WORK

As we have already discussed in earlier sections, we extend the notion of hitting families of schedules, formalized by Chistikov et al. [2016], to strong hitting families. Chistikov et al. show that for certain partial orders such as antichains and trees, one can explicitly construct hitting

Fig. 2. Scheduling poset for Zookeeper with 3 nodes. Gray events are executed and black events are enabled.

Table 5. Zookeeper results. For each strategy we executed 1,000 runs. The experiment marked by † had no crashes or reboots; in all other cases there was 1 node crash and 1 reboot per run. In the case of PCTCP with \( d = 4 \), we observed a non-terminating run, most likely exhibiting bug ZK-1419. The run was terminated after reaching 1,000 events. Numbers marked by † are given with the non-terminating run excluded.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Avg max enabled events</th>
<th>Max enabled events</th>
<th>Avg enabled events</th>
<th>Max enabled events</th>
<th>Avg max chains</th>
<th>Max chains</th>
<th>Faulty runs</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random walk</td>
<td>6</td>
<td>6</td>
<td>20.9</td>
<td>36</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>7,040</td>
</tr>
<tr>
<td>PCTCP (( d = 1, n = 50 ))</td>
<td>6</td>
<td>6</td>
<td>20.6</td>
<td>31</td>
<td>6.4</td>
<td>10</td>
<td>39</td>
<td>2,059</td>
</tr>
<tr>
<td>PCTCP (( d = 2, n = 50 )*</td>
<td>3</td>
<td>5</td>
<td>18.2</td>
<td>52</td>
<td>4.5</td>
<td>11</td>
<td>34</td>
<td>5,107</td>
</tr>
<tr>
<td>PCTCP (( d = 2, n = 50 ))</td>
<td>6</td>
<td>6</td>
<td>20.7</td>
<td>34</td>
<td>7.5</td>
<td>11</td>
<td>32</td>
<td>2,060</td>
</tr>
<tr>
<td>PCTCP (( d = 3, n = 50 ))</td>
<td>6</td>
<td>6</td>
<td>21.0</td>
<td>34</td>
<td>8.6</td>
<td>12</td>
<td>26</td>
<td>2,103</td>
</tr>
<tr>
<td>PCTCP (( d = 4, n = 50 ))</td>
<td>6</td>
<td>7</td>
<td>23.4†</td>
<td>63</td>
<td>10.1†</td>
<td>18†</td>
<td>40+1†</td>
<td>6,835†</td>
</tr>
<tr>
<td>PCTCP (( d = 5, n = 50 ))</td>
<td>6</td>
<td>6</td>
<td>23.7†</td>
<td>70</td>
<td>11.2†</td>
<td>19</td>
<td>42</td>
<td>7,111†</td>
</tr>
</tbody>
</table>

families exponentially smaller than the size of the partial order. However, the partial order has to be known in advance. In this work, we provide an online construction of strong hitting families of bounded size for arbitrary partial orders. Our algorithm PCTCP generalizes PCT, a randomized scheduler for multithreaded programs [Burckhardt et al. 2010], to concurrency models where chain partition of the partial order is not implicitly given by syntactic structures like threads. Instead, PCTCP partitions the partial order on the fly using online chain partitioning.

The online chain partitioning algorithm we use is by Agarwal and Garg, and it appears in their work on chain clocks [Agarwal and Garg 2007]: they compare vector clocks, where each component in the clock corresponds to a thread in the program, and chain clocks, where each component in the clock corresponds to a chain in a chain partition obtained by online chain partitioning. They show that in many cases chain clocks are considerably more efficient than vector clocks. Similarly to this, it would make sense to try running PCTCP side-by-side with PCT on multithreaded programs, and see whether there are scenarios in which PCTCP would find a better chain partition than the one induced by threads. An experiment along these lines is left for future work.

Online chain partitioning and its connection to online dimension for upgrowing posets was studied by Felsner [1997] and Kloch [2007]. Their work is part of a larger context of studying unrestricted posets. In the unrestricted setting, bounds on the optimal number of chains are much worse than for upgrowing posets. While an upgrowing poset can be partitioned online into at most \( \binom{w+1}{2} \) chains, for a long time the best known upper bound for an unrestricted poset was \( (5^w - 1)/4 \) [Kierstead 1981]. Bosek and Krawczyk [2010] found a subexponential upper bound of \( w^{16 \log_2 w} \), which was recently improved to \( w^{6.5 \log_2 w + 7} \) [Bosek et al. 2018]. A nice survey of the results in this setting was done by Bosek et al. [2012].

On the application side, our work is related to a number of tools for finding bugs in concurrent and distributed systems. We start by mentioning Bita, a testing tool for actor programs implemented within Akka framework in Scala [Tasharofi et al. 2013]. By using an arbitrary execution of the program as the initial schedule, Bita systematically reverses the order of pairs of concurrent messages in a given schedule to produce new schedules, and then executes these schedules. The exploration is guided by coverage goals similar to our 2-hitting goal. Unlike Bita, our tool randomly samples schedules from a strong \( d \)-hitting family for arbitrary \( d \), and does not rely on any initial execution of the program.

Another related tool is EventRacer, a race detector for client-side web applications [Raychev et al. 2013]. Similarly to Bita, EventRacer explores schedules by pairwise reversal of concurrent events,
but with the goal of detecting races—concurrent conflicting accesses to the same memory location. One of the key contributors to EventRacer’s efficiency is the use of chain clocks instead of vector clocks. EventRacer’s chain clocks are based on a greedy chain decomposition. The authors report a 33-fold reduction in the average length of chain clocks compared to standard vector clocks. Even with a highly optimized bit-vector representation of vector clocks, chain clocks are reported to consume on average 6.6 times less memory, which significantly improves overall performance. In a different paper on race detection, Dimitrov et al. [2015] also use insights from theoretical work on partial orders to show that partially ordered sets of dimension two admit efficient race detection.

In Section 4 we have discussed P# [Deligiannis et al. 2016], a framework for building and systematic testing of asynchronous programs, and SAMC [Leesatapornwongsa et al. 2014], a tool for systematic, semantic-aware testing of distributed systems. In both P# and SAMC, as well as our own work, the testing is done by taking control of the scheduler and imposing a particular order of events in the program. Another approach is taken by frameworks like Jepsen [Kingsbury 2018], where a system under test is run as a black box, while the framework alters the environment in which the system is running in order to expose abnormal behavior. In particular, Kingsbury [2013] found a large number of bugs in many production systems by randomly introducing network partitions while the system under test is running. Jepsen itself does not provide any guarantees on finding bugs. However, by doing an a posteriori analysis, Majumdar and Niksic [2018] showed that for many bugs discovered by Jepsen, one can derive guarantees on the probabilities that they are found, and these probabilities are sufficiently high to justify Jepsen’s empirically observed success.

Different notions of bug depth are defined in the literature. These notions aim to parametrize the search space by a depth d, so that a d-bounded exploration provides a high coverage of the executions that are likely to be buggy. Context bounding [Qadeer and Rehof 2005] characterizes the depth as the number of context switches between threads required to hit a bug. Preemption bounding [Musuvathi and Qadeer 2007] bounds only the preemptive switches between the tasks. Although this notion yields a smaller depth for the cases where tasks run to completion, it is still not efficient for exploring schedules where the many tasks need to be preempted to hit a schedule. Delay bounding [Emmi et al. 2011], which defines the depth as the number of deviations from a given deterministic scheduler, and phase bounding [Bouajjani and Emmi 2012], which bounds the number of process communication cycles, are applicable to distributed system setting, since the bug depth parameter does not limit the number of tasks/nodes involved in the execution. The work by Desai et al. [2015] presents a randomized algorithm for asynchronous systems based on delay bounded exploration. Their algorithm is parametrized by a delaying scheduler where the depth parameter does not characterize the bug but the search space. The bug depth we use in this work (which is also used in Burckhardt et al. [2010] and Chistikov et al. [2016]) is defined only by the ordering constraints between the events in the execution, and it is independent of the exploration strategy.

Partial order reduction techniques [Abdulla et al. 2014; Flanagan and Godefroid 2005; Godefroid 1996] perform systematic search of behaviors of concurrent systems where only one representative behavior among equivalent behaviors, i.e., those differing only in the ordering of independent events, is required to be executed. In their recent work, Yuan et al. [2018] introduce a randomized scheduling algorithm that takes partial order reduction into account. These techniques do not restrict the search space to depth-d bugs, as we do. We do not know how partial order reduction can be modified to efficiently explore depth-d bugs.
A APPENDIX

A.1 Proof of Theorem 7

Proof. Let \( \mathcal{P} = (\mathcal{P}_k)_{0 \leq k \leq n} \) be an upgrowing poset of size \( n \) and width at most \( w \), and let \( C = (C_k)_{0 \leq k \leq n} \) be an adaptive chain covering for \( \mathcal{P} \) with \( m \) chain colors. We show how to transform 
\[ C \] step by step into an online strong \( d \)-hitting family \( \mathcal{F} = (\mathcal{F}_k)_{0 \leq k \leq n} \) with at most \( m \cdot \binom{n}{d-1}(d-1)! \) schedules. 

The schedules in families \( \mathcal{F}_k \) will be indexed by \( d \)-tuples \((\lambda, n_1, \ldots, n_{d-1})\), where \( \lambda \in \Lambda \) is a chain color, and \( n_i \in \{1, \ldots, n\} \) for \( 1 \leq i < d \) are distinct numbers. We do not require the indexing scheme to be injective, that is, multiple indices may denote a single schedule. Intuitively, the numbers \( n_1, \ldots, n_{d-1} \) serve to single out elements \( x_1, \ldots, x_{d-1} \) added to the poset in steps \( n_1, \ldots, n_{d-1} \), respectively. The schedule \( \alpha = \alpha_{\lambda, n_1, \ldots, n_{d-1}} \) will be constructed so that it strongly hits \((x_0, x_1, \ldots, x_{d-1})\) for every element \( x_0 \) with \( \lambda \in C_k(x_0) \). More precisely, the construction will satisfy the following invariant: For \( 0 \leq k \leq n \), let \( x_{i_1}, \ldots, x_{i_l} \) with \( i_1 < \ldots < i_l \) be all the singled-out elements in step \( k \), and let \( x_0 \) be an element such that \( \lambda \in C_k(x_0) \). The schedule \( \alpha \) strongly hits \((x_0, x_{i_1}, \ldots, x_{i_l})\).

Clearly, a family \( \mathcal{F}_0 \) consisting of a single empty schedule satisfies the invariant. Let \( x \) be the element added to \( \mathcal{P}_k \) in step \( k > 0 \), and let \( \mathcal{F}_{k-1} \) be a family of schedules that satisfies the invariant in step \( k - 1 \). We show how to extend the schedule \( \alpha = \alpha_{\lambda, n_1, \ldots, n_{d-1}} \in \mathcal{F}_{k-1} \) with \( x \) to obtain a schedule \( \alpha' = \alpha'_{\lambda, n_1, \ldots, n_{d-1}} \in \mathcal{F}_k \) so that \( \mathcal{F}_k \) satisfies the invariant in step \( k \). We distinguish several cases:

1. If \( k = n_i \) for some \( i \in \{1, \ldots, d-1\} \), then \( x \) needs to be singled out: we define \( x_{i_j} := x \). Let \( x_{i_1}, \ldots, x_{i_j} \) with \( i_1 < \ldots < i_j \) be all previously singled-out elements. If \( l = 0 \) or \( i > i_j \) or \( x \geq x_{i_j} \), we schedule \( x \) as the last element in \( \alpha' \). (This is possible because \( x \) is maximal in \( \mathcal{P}_k \)). Otherwise, let \( i_j \) be the least index such that \( i < i_j \) and \( x \) is incomparable with all of \( x_{i_1}, x_{i_1+1}, \ldots, x_{i_j} \). We schedule \( x \) right before \( x_{i_j} \). (This is possible because of the invariant in step \( k - 1 \): if \( x \geq y \) for some \( y \geq x \), then \( x \geq y \geq x_{j'} \) for some \( j' \leq l \).)

To see that the invariant holds for \( \alpha' \), let \( x_0 \) be an element such that \( \lambda \in C_k(x_0) \) and set \( i_0 := 0 \). If \( x \geq x_{i_j} \) for some \( j \in \{0, \ldots, l\} \), then either \( i < i_j \) and \( x \geq x_{j'} \) for some \( j' \geq j \) by construction, or \( i > i_j \) and \( x \geq x_{i_j} \) trivially. Since \( x \) is singled out, we also need to inspect the case when \( y \geq x_{i_j} \) for some \( y \in \mathcal{P}_{k-1} \). By construction, \( x \) immediately precedes some previously singled-out element \( x_{j'} \) such that \( i < i_j \). Therefore, \( y \geq x_{j'} \), and by the invariant in step \( k - 1 \), \( y \geq x_{j'} \) for some \( j' \geq j \).

2. If \( x \) is not to be singled out and \( \lambda \in C_k(x) \), let again \( x_{i_1}, \ldots, x_{i_l} \) with \( i_1 < \ldots < i_l \) be all previously singled-out elements. If \( l = 0 \) or \( x \geq x_{i_l} \), we schedule \( x \) as the last element in \( \alpha' \). Otherwise, let \( i_l \) be the least index such that \( x \) is incomparable with all of \( x_{i_1}, x_{i_1+1}, \ldots, x_{i_l} \). We schedule \( x \) right before \( x_{i_l} \).

As before, to see that the invariant holds for \( \alpha' \), let \( x_0 \) be an element such that \( \lambda \in C_k(x_0) \) and set \( i_0 := 0 \). Assume \( x \geq x_{i_j} \) for some \( j \in \{0, \ldots, l\} \). If \( j = 0 \), then \( x \geq x_{i_0} \) because both elements are in chain \( \lambda \). If \( j > 0 \), by construction there exist \( j' \geq j \) such that \( x \geq x_{i_{j'}} \). Since \( \lambda \in C_k(x) \), we also need to inspect the case when \( y \geq x_{i_{j'}} \) for some \( y \in \mathcal{P}_{k-1} \). By construction, \( x \) immediately precedes some previously singled-out element \( x_{i_{j'}} \). Therefore, \( y \geq x_{i_{j'}} \), and by the invariant in step \( k - 1 \), \( y \geq x_{i_{j'}} \) for some \( j' \geq j \).

3. Finally, if \( x \) is not to be singled out and \( \lambda \notin C_k(x) \), we schedule \( x \) right after the last \( y \) such that \( y \leq x \). To show the invariant, let \( x_{i_1}, \ldots, x_{i_l} \) with \( i_1 < \ldots < i_l \) be all previously singled-out
elements, let $x_0$ be an element such that $\lambda \in C_k(x_0)$, and set $i_0 := 0$. If $x \geq x'_j$ for some $j \in \{0, \ldots, l\}$, then we also have $y \geq x_j$. Hence $x \geq y \geq x_j$, for some $j' \geq j$.

Now let $(x_0, \ldots, x_{d-1})$ be a $d$-tuple in $\mathcal{P}_k$. Since $C$ is an adaptive chain covering, there exists $\lambda \in C_k(x_0)$. Moreover, there exist steps $n_1, \ldots, n_{d-1}$ in which the elements $x_1, \ldots, x_{d-1}$ were added to $\mathcal{P}_k$. Because of the invariant, the schedule $\alpha_{n_1, \ldots, n_{d-1}} \in T_k$ strongly hits the tuple.}

\section{Schedules in Scheduling Posets}

Note that Definition 8 of scheduling posets never explicitly requires a schedule $\alpha$ to extend a prefix of $\mathcal{P}_{\alpha,k}$. However, this fact can easily be derived from the definition.

\textbf{Lemma 18.} Let $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ be a scheduling poset. For every $\alpha \in \mathcal{S}$ and $k$ such that $0 \leq k \leq n_\alpha$, $\alpha$ is a schedule of a prefix of $\mathcal{P}_{\alpha,k}$.

\textbf{Proof.} The claim clearly holds for $\alpha = \epsilon$ and all $k$ such that $0 \leq k \leq n_\epsilon$. Assume the claim holds for some $\alpha \in \mathcal{S}$ and all $k$ such that $0 \leq k \leq n_\alpha$ and let $x \in \min(\mathcal{P}_{\alpha,n_\alpha})$. Since $\mathcal{P}_{\alpha-x,0} = \mathcal{P}_{\alpha,n_\alpha}$, and since $x$ is minimal in $\mathcal{P}_{\alpha,n_\alpha}$, there exists no $y \in \mathcal{P}_{\alpha-x,0} \setminus \alpha \cdot x$ such that $y \leq x$. Therefore, $\alpha \cdot x$ is a schedule of a prefix of $\mathcal{P}_{\alpha-x,0}$. Now assume $\alpha \cdot x$ is a schedule of a prefix of $\mathcal{P}_{\alpha-x,k}$ for some $k$ such that $0 \leq k < n_{\alpha \cdot x}$. Let $\mathcal{P}_{\alpha-x,k+1} = \mathcal{P}_{\alpha-x,k} \cup \{y\}$. Since $y$ is maximal in $\mathcal{P}_{\alpha-x,k+1}$, we cannot have $y \leq x$. (This also follows from $x < y$). Therefore, $\alpha \cdot x$ is a schedule of a prefix of $\mathcal{P}_{\alpha-x,k+1}$. The claim now follows by sub-induction on $k$ and induction on $\alpha$.

\section*{Acknowledgments}

This research was sponsored in part by the European Research Council Grant Agreement No. 610150 (ERC Synergy Grant ImPACT (http://www.impact-erc.eu/)), by the Vienna Science and Technology Fund (WWTF) through the projects Heisenbugs (VRG11-005) and APALACHE (ICT15-103), and by the Austrian Science Fund (FWF) through the projects Heisenbugs (VRG11-005) and APALACHE (ICT15-103), and by the Austrian National Research Network S11403-N23 (RiSE). We thank Haryadi S. Gunawi, Jeffrey F. Lukman, and Tanakorn Leesatapornwongsa for their help with the SAMC tool.

\section*{References}


