Fast Detection of Stable and Count Predicates in Parallel Computations

Himanshu Chauhan\textsuperscript{1} and Vijay K. Garg\textsuperscript{1}

\textsuperscript{1} University of Texas at Austin
\{himanshu, garg\}@ece.utexas.edu

Abstract

Enumerating all consistent states of a parallel computation that satisfy a given predicate is an important problem in debugging and verification of parallel programs. We give a fast algorithm to enumerate all consistent states of a parallel computation that satisfy a stable predicate. In addition, we define a new category of global predicates called count predicates and give an algorithm to enumerate all consistent states (of the computation) that satisfy it. All existing predicate detection algorithms, such as BFS, DFS and Lex algorithms, do not exploit the knowledge about the nature of the predicates, and thus may visit all global states of the computation in the worst case. In comparison, our algorithms only visit the states that satisfy the given predicate, and thus take time and space that is a polynomial function of the number of states of interest. In doing so, they provide a significant reduction — exponential in many cases — in time complexities in comparison to existing algorithms.

Keywords and phrases Algorithms, Theory, Predicate Detection, Parallel Programs

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Predicate detection [17, 10] is a powerful technique for verifying parallel programs. It allows inference based analysis to check many possible system states based on a single execution of the program. It involves three key steps: (a) obtaining an execution trace of the program, (b) modeling this trace as a partial order, and (c) checking all possible states of the model that are consistent with the partial order against a predicate that encodes the violation of any constraint or invariant. A large body of work uses this approach to verify distributed applications, as well as to detect data-races and other concurrency related bugs in shared memory parallel programs [9, 13, 20, 24].

In many debugging/analysis applications, we may be interested in analyzing each consistent global state — often called a consistent cut — of a parallel program that satisfies a given predicate. For example, while debugging an implementation of the Paxos [23] algorithm, a programmer might only be interested in analyzing consistent cuts when all the promise messages of a particular round have been delivered. Another scenario is when a programmer knows that her program exhibits a bug only after the system has executed a certain number of, let us say $k$, events. For these two scenarios, our predicate definitions are: $B = \text{all promises have been delivered}$, and $B = \text{at least } k \text{ events have been executed}$.

Both of these predicates fall under the category of stable predicates. A stable predicate is a predicate that remains true once it becomes true. In addition, some predicates are defined on the count of some specific types of events in the system. We call such global predicates count predicates. This category of predicates encodes many useful conditions for debugging/verification of parallel programs. For example, $B = \text{exactly two promise messages have been received}$ is a count predicate.
Let us call the partial order model of the execution trace a computation. If we are interested in enumerating all possible consistent cuts of a computation that satisfy a global predicate \( B \) that is of either a stable or a count predicate, then we currently only have one choice: traverse all the cuts using existing traversal algorithms (such as BFS [10], DFS [1], and Lex [15, 16] and check which ones satisfy \( B \). This is generally wasteful because we traverse many more cuts than needed — especially if the subset of cuts satisfying \( B \) is relatively small. For example, consider the computation in Figure 1, and the predicate \( B = \text{at least 4 events have been executed} \). Figure 2 shows all the consistent cuts of the computation as a distributive lattice using the vector clock notation. There are five such cuts in which at least four events have been executed. Using the BFS, DFS, or Lex traversal algorithms, however, we will have to visit all the twelve cuts to find these five.

We present the first algorithms to efficiently enumerate the subset of consistent cuts that satisfy stable or count predicates without enumerating other consistent cuts that do not satisfy them. Our algorithms take time and space that is a polynomial function of the number of consistent cuts of interest, and in doing so provide a significant — possibly exponential — reduction in time complexities in comparison to existing algorithms.

2 Background

We model a computation \( P = (E, \rightarrow) \) on \( n \) processes \( \{P_1, P_2, \ldots, P_n\} \) as a partial order on the set of events, \( E \). The events are ordered by Lamport’s happened-before (\( \rightarrow \)) relation [22]. This partially ordered set (poset) of events is generally partitioned into chains:

Definition 1 (Chain Partition). A chain partition of a poset places every element of the poset on a chain that is totally ordered. Formally, if \( \alpha \) is a chain partition of poset \( P = (E, \rightarrow) \) then \( \alpha \) maps every event to a natural number such that

\[
\forall x, y \in E : \alpha(x) = \alpha(y) \Rightarrow (x \rightarrow y) \lor (y \rightarrow x).
\]

Generally, a computation on \( n \) processes is partitioned into \( n \) chains such that the events executed by process \( P_i \) (\( 1 \leq i \leq n \)) are placed on \( i^{th} \) chain.

Mattern [25] and Fidge [12] proposed vector clocks, an approach for time-stamping events in a computation such that the happened-before relation can be tracked. For a program on \( n \) processes, each event’s vector clock is a \( n \)-length vector of integers. Note that vector clocks are dependent on chain partition of the poset that models the computation. For an event \( e \), we denote \( e.V \) as its vector clock. Throughout this paper, we use the following representation for interpreting chain partitions and vector clocks:

- If there are \( n \) chains in the chain partition of the computation, then the lowest chain (process) is always numbered 1, the second lowest chain is numbered 2, and so on, with the highest chain being numbered \( n \).
A vector clock on \( n \) chains is represented as a \( n \)-length vector: \([e_n, e_{n-1}, \ldots, e_i, \ldots, e_2, e_1]\) such that the rightmost (first from right) entry denotes the number of events executed on \( P_1 \), second from right entry denotes events executed on \( P_2 \), and so on, such that the left most \((n^{th} \text{ from right})\) entry holds the number of events executed on \( P_n \).

Hence, if event \( e \) was executed on process \( P_i \), then \( e.V[i] \) is \( e \)'s index (starting from 1) on \( P_i \). Also, for any event \( f \) in the computation: \( e \rightarrow f \Leftrightarrow \forall j : e.V[j] \leq f.V[j] \land \exists k : e.V[k] < f.V[k] \). A pair of events, \( e \) and \( f \), is concurrent if \( e \not\rightarrow f \land f \not\rightarrow e \). We denote this relation by \( e||f \). Note that concurrent events must occur on separate processes. Figure ?? shows the corresponding vector clocks of the computation shown in Figure 1. Event \( b \) is the second event on process \( P_1 \), and thus using the notation described above its vector clock is \([0, 2]\). Event \( g \) is the third event on \( P_2 \), but it is preceded by \( f \), which in turn is causally dependent on \( b \) on \( P_1 \), and thus the vector clock of \( f \) is \([3, 2]\).

**Definition 2 (Consistent Cut).** Given a computation \((E, \rightarrow)\), a subset of events \( C \subseteq E \) forms a consistent cut if \( C \) contains an event \( e \) only if it contains all events that happened-before \( e \). Formally, \( e \in C \land f \rightarrow e \implies f \in C \).

A consistent cut captures the notion of a global state of the system at some point during its execution [3]. We use the term cut for a possible global state of the computation which may or may not be consistent. When the global state is consistent, we use the term consistent cut.

Consider the computation shown in Fig 1. The subset of events \( \{a, b, e\} \) forms a consistent cut, whereas the subset \( \{a, c, f\} \) does not; because \( b \rightarrow f \) (\( b \) happened-before \( f \)) but \( b \) is not included in the subset.

**Vector Clock Notation of Cuts:** So far we have described how vector clocks can be used to time-stamp events in the computation. We also use them to represent cuts of the computation. If the computation is partitioned into \( n \) chains, then for any cut \( G \), its vector clock is a \( n \)-length vector such that \( G[i] \) denotes the number of events from \( P_i \) included in \( G \). Note that in our vector clock representation the events from \( P_i \) are at the \( i^{th} \) index from the right.

For example, consider the state of the computation in Figure 1 when \( P_1 \) has executed events \( a \) and \( b \), and \( P_2 \) has only executed event \( e \). The consistent cut for this state, \( \{a, b, e\} \), is represented by \([1, 2]\). Note that cut \([2, 1]\) is not consistent, as it indicates execution of \( f \) on \( P_2 \) without \( b \) being executed on \( P_1 \).

**Definition 3 (Lexical Order on Consistent Cuts).** Given any chain partition of poset \( P \) that partitions it into \( n \) chains, we define a total order called lexical order on all consistent cuts of \( P \) as follows. Let \( G \) and \( H \) be any two consistent cuts of \( P \). Then, \( G <_l H \equiv \exists k : (G[k] < H[k]) \land \forall i : n \geq i > k : G[i] = H[i] \)

**Theorem 1.** [11, 25] Let \( C(E) \) denote the set of all consistent cuts of a computation \((E, \rightarrow)\). \( C(E) \) forms a finite distributive lattice under the relation \( \subseteq \).

Figure 7 in Appendix A, shows the twelve consistent cuts of the computation in Figure 1 in set notation and their corresponding vector clock notation.

**Rank of a Cut:** Given a cut \( G \), we define \( \text{rank}(G) = \sum G[i] \). The rank of a cut corresponds to the total number of events, across all processes, that have been executed to reach the cut.
## 3 Uniflow Chain Partition

We now discuss a special chain partition of called *uniflow chain partition*. A uniflow partition of a poset $P$ is its partition into $n_u$ chains $\{\mu_i \mid 1 \leq i \leq n_u\}$ such that no element in a higher numbered chain is smaller than any element in lower numbered chain; that is if any element $e$ is placed on a chain $i$ then all elements smaller than $e$ must be placed on chains numbered lower than $i$. For poset $P$, chain partition $\mu$ is uniflow if

$$\forall x, y \in P : \mu(x) < \mu(y) \Rightarrow \neg(y \nrightarrow x)$$

(1)

![Figure 4 Posets in Uniflow Partitions](image)

Visually, in a uniflow chain partition all the edges between separate chains always point upwards. Figure 3 shows two posets with uniflow partitions. Whereas Figure 4 shows two posets with partitions that do not satisfy the uniflow property. The poset in Figure 4a can be transformed into a uniflow partition of three chains as shown in Figure 4b. Similarly, Figure 4c can be transformed into a uniflow partition of two chains shown in Figure 4d. Observe that:

▶ **Lemma 1.** Every poset has at least one uniflow chain partition.

**Proof.** Any total order derived from the poset is a uniflow chain partition in which each element is a chain by itself. In this trivial uniflow chain partition the number of chains is equal to the number of elements in the poset. 

For any poset $P$, the number of chains in any of its uniflow partition is always less than or equal to $|P|$ (the number of elements in poset). Let us now focus on finding a uniflow chain partition of our poset model of a parallel computation.

We define a total order, called *uniflow order*, on the events of the computation based on its uniflow chain partition. Recall from Equation 1 that for any event $e$, $\mu(e)$ denotes its chain number in $\mu$. Let $pos(e)$ denote the index of event $e$ on chain $\mu(e)$. Note that a chain is totally ordered, and thus for any two events on the same chain one event’s index will be greater than the other’s.

▶ **Definition 4 (Uniflow Order on Events, $<_u$).** Let $\mu$ be uniflow chain partition of a computation $P = (E, \rightarrow)$ that partitions it into $n_u$ chains. We define a total order called *uniflow order* on the set of events $E$ as follows. Let $e$ and $f$ be any two events in $E$. Then, $e <_u f \equiv (\mu(e) < \mu(f)) \lor (\mu(e) = \mu(f) \land pos(e) < pos(f))$

For example, in Figure 4b we have $a <_u e$ as $\mu(a) = 1$ and $\mu(e) = 2$; and $e <_u f$ as $\mu(e) = \mu(f) = 2$, $pos(e) = 1$ and $pos(f) = 2$.

The problem of finding a uniflow chain partition is a direct extension of finding the *jump number* of a poset [8, 2, 31]. Multiple algorithms have been proposed to find the jump number of a poset; which in turn arrange the poset in a uniflow chain partition. Finding an optimal (smallest number of chains) uniflow chain partition of a poset is a hard problem.
[8, 2]. Bianco et al. [2] present a heuristic algorithm to find a uniflow partition, and show in their experimental evaluation that in most of the cases the resulting partitions are relatively close to optimal. We use the online algorithm given in [6] to find a uniflow partition for a computation.

Recall that for the vector clock notation of a cut $G$, $G[i]$ denotes the number of events included from chain $i$. The vector clocks of events, and cuts, of a computation are dependent on the underlying chain partition, and hence we re-generate the vector clocks of the events for the uniflow partition. This is a simple task using existing vector clock implementation techniques, and we omit these details.

Note that the set of consistent cuts of a computation remains the same irrespective of the chain partition used. Hence, if the computation’s uniflow partition is different from its original chain partition, we re-map the vector clock of consistent cuts in uniflow partition to the vector clocks of cuts in original partition. For the details of this step, we refer the reader to [6].

The structure of uniflow chain partitions can be used for efficiently obtaining bigger consistent cuts. After we find a uniflow chain partition of a computation, and regenerate the vector clocks of events as per this partition, we have the following result.

**Lemma 2 (Uniflow Cuts Lemma).** Let $P$ be a poset with a uniflow chain partition $\{\mu_i | 1 \leq i \leq n_u\}$, and $G$ be a consistent cut of $P$. Then any $H_k \subseteq P$ for $1 \leq k \leq n_u$ is also a consistent cut of $P$ if it satisfies:

\[
\forall i : k < i \leq n_u : H_k[i] = G[i], \text{ and } \\
\forall i : 1 \leq i \leq k : H_k[i] = |\mu_i|.
\]

**Proof.** Using Equation 1, we exploit the structure of uniflow chain partitions: the causal dependencies of any element $e$ lie only on chains that are lower than $e$’s chain. As $G$ is consistent, and $H_k$ contains the same elements as $G$ for the top $(n_u - k)$ chains, all the causal dependencies that need to be satisfied to make $H_k$ have to be on chain $k$ or lower. Hence, including all the elements from all of the lower chains will naturally satisfy all the causal dependencies, and make $H_k$ consistent.

For example, in Figure 3b, consider the cut $G = [1, 2, 1]$ that is a consistent cut of the poset. Then, picking $k = 1$, and using Lemma 2 gives us the cut $[1, 2, 3]$ which is consistent; similarly choosing $k = 2$ gives us $[1, 3, 3]$ that is also consistent. Note that the claim may not hold if the chain partition does not have uniflow property. For example, in Figure 4c,
Fast Detection of Stable and Count Predicates in Parallel Computations

\( G = [2, 2] \) is a consistent cut. The chain partition, however, is not uniflow and thus applying Lemma 2 with \( k = 1 \) gives us \([2, 3] \) which is not a consistent cut as it includes the third event on \( P_1 \), but not its causal dependency — the third event on \( P_2 \).

We now define the notion of a base cut: a consistent cut that is formed by including events from \( \mu \) in a bottom-up manner.

**Definition 5 (l-Base Cut).** Let \( G \) be a consistent cut of a computation \( P = (E, \rightarrow) \) with uniflow partition \( \mu \). Then, we call \( G \) a \( l \)-base cut if \( \forall j \leq l : G[j] = \text{size}(\mu_j) \)

Thus, in a \( l \)-base cut we must include all the events from each chain that is same or lower than \( \mu_l \) in the uniflow partition \( \mu \). In Figure 4b, \( \{a, e, b\} \) (or \([1, 1, 1]\) in its vector clock notation) is a consistent cut. It is a \( 1 \)-base cut as it includes all the elements from chain \( \mu_1 \), but it is not a \( 2 \)-base cut as it does not include all the events from the chain \( \mu_2 \).

## 4 Enumerating Consistent Cuts Satisfying Stable Predicates

A predicate \( B \) is stable if once it becomes true it stays true. Some examples of stable predicates are: deadlock, termination, loss of message, at least \( k \) events have been executed, and at least \( k' \) messages have been sent.

**Definition 6 (Stable Predicate).** Let \( C \) be the set of all consistent cuts of a computation. A predicate \( B \) defined on \( C \) is called stable if and only if \( \forall G, H \in C : G \subseteq H \) implies that if \( B(G) \) is true then \( B(H) \) is also true.

Thus, for any stable predicate \( B \) the lattice of consistent cuts can be split in two parts using a boundary: every consistent cut higher than the boundary satisfies \( B \), and no consistent cut lower than the boundary satisfies \( B \). Figure 8 in Appendix B presents a visualization for this concept. Our goal is to enumerate all consistent cuts of a computation \( P = (E, \rightarrow) \) that satisfy a stable predicate \( B \). Note that if the empty cut, \( \{\} \) satisfies \( B \), then by the stability property of \( B \) all the consistent cuts of the computation satisfy \( B \). In this case, the problem is equivalent to traversing all the consistent cuts of a computation. We can use a fast traversal algorithm such as QuickLex [5] to do so. We now focus on the non-trivial case, and present our algorithm that enumerates only the consistent cuts that satisfy \( B \), and does not enumerate the remaining parts of the lattice of consistent cuts.

Recall that \( C(E) \) represents the set of all consistent cuts of the computation \( P = (E, \rightarrow) \). Let \( S_B \subset C(E) \) be the set of all consistent cuts of \( P \) that satisfy a stable predicate \( B \). We use \( P \)'s uniflow partition \( \mu \) to enumerate them in their lexical order based on the uniflow partition. Let \( G \) and \( H \) be two consistent cuts of \( P \), then applying the definition of lexical order (Definition 3) over \( n_u \) chains, we get \( G <_l H \equiv \exists k : (G[k] < H[k]) \land (\forall i : n_u \geq i > k : G[i] = H[i]) \).

We use the \texttt{EnumerateStable} routine in Algorithm 1 for this enumeration. We first find the lexically smallest consistent cut \( G \) that satisfies \( B \). We then find the next cut that is lexically greater than \( G \) and satisfies \( B \), and repeat the process after re-assigning \( G \) to this cut. We stop when no such lexically greater cut satisfying \( B \) is found.

Given a consistent cut \( G \), and a stable predicate \( B \), we use the \texttt{GetMinCut} routine in Algorithm 2 to find the lexically smallest cut that is greater than \( G \) and satisfies \( B \). We use two sub-routines for this task: \texttt{GetBiggerBaseCut} and \texttt{BackwardPass}.

The \texttt{GetBiggerBaseCut} routine in Algorithm ?? takes a consistent cut, \( G \), and returns a pair: the first entry is the lexically smallest \( l \)-base cut (Definition 5) \( H \) lexically greater than \( G \) that satisfies \( B \), and the second entry is the chain number from which we added the last event to \( H \) before returning the result. If no such cut \( H \) can be found, then
Algorithm 1: EnumerateStable((E,→), B)

Input: Computation (E,→) in its uniflow chain partition µ, B: a stable predicate
Output: Enumerate all consistent cuts satisfying B.
1: G = GetMinCut(B, {}) // find the lexically smallest consistent cut satisfying B
2: while G ≠ null do
3: enumerate(G) // enumerate the cut
4: G = GetMinCut(B, G) // find the next lexically smallest consistent cut >1 G satisfying B

Algorithm 2: GetMinCut(B, G)

Input: B: a stable predicate, G: a consistent cut
Output: lexically smallest consistent cut >1 G that satisfies B
1: (H, c) = GetBiggerBaseCut(B, G)
2: return BackwardPass(B, c − 1, H)

Let us illustrate the execution with an example. Consider the computation in Figure ?? and the predicate B=P2 has executed two or more events, and the call GetBiggerBaseCut (B, {c}). We use the uniflow partition, and starting at µ1, with H = G = {c}, we add the first and only event of this chain, a, to H and get {a, c} that is greater than G but does not satisfy B, as a was executed on P1 in the computation. We now jump to chain µ2, and find the first event on µ2 that is not included in H. This event is b, the second event on chain. We add it to H and get H = {a, b, c} that still does not satisfy B. We now move to the third chain, and add its only event d to H. We now have H = {a, b, c, d} and it satisfies B. We return (H = {a, b, c, d}, i = 3).

The BackwardPass routine (in Algorithm ??) takes three arguments: a stable predicate B, a chain number start, and a consistent cut G that satisfies B. It returns a consistent cut H such that H satisfies B, and H is the lexically smallest member of the set: \{G’ ⊆ G : H[j] = G[j], start + 1 ≤ j ≤ nu\}. Thus, H is the lexically smallest consistent cut H ≤I G that satisfies B such that H and G include the same set of events from chains start + 1 and higher. Note that whenever start = nu, we have start + 1 > nu, and the routine returns without changing the passed cut. We start from the given chain number and traverse backwards on it removing the events as long as the resulting cut continues to satisfy B. If removing an event will cause the cut to become inconsistent or not satisfy B, we do not remove the event and move to the chain immediately below. Consider the computation in Figure ?? and the predicate B=P2 has executed two or

![Figure 6](image_url) A computation on two processes in: (a) its original non-uniflow partition, (b) equivalent uniflow partition.
Fast Detection of Stable and Count Predicates in Parallel Computations

Algorithm 3 GetBiggerBaseCut(B, G)

Input: B: a stable predicate, G: a consistent cut
Output: pair ⟨H, i⟩: H is the smallest base cut that is lexically greater than G and satisfies B, i is the chain number in µ from which we added the last event to H.

1: H = G
2: for (i = 1; i ≤ n_u; i = i + 1) do // go from lowest chain to highest
3:    j = index of the smallest event on chain µ_i that is not included in H
4:    for (; j ≤ size(µ_i); j = j + 1) do // use events on chain i not included in G
5:       H = H ∪ {µ_i[j]} // add event to cut H
6:    end for
7: end for
8: return ⟨H, i⟩ // return H and chain number of the event

For the computation in Figure ?? and the predicate B = P_2 has executed two or more events, let us find the lexically smallest cut that satisfies B. We use the GetMinCut routine, and since we are interested in finding the lexically smallest cut, we start with G = {}. Calling GetBiggerBaseCut (B, {}) returns ⟨H = {a, b, c, d}, i = 3⟩ as shown earlier. Now calling BackwardPass (B, 2, {a, b, c, d}) returns {a, c, d}. This is the lexically smallest cut of the computation that satisfies B.

Let us now go through a run of EnumerateStable routine. For the computation in Figure ?? and the predicate B = P_2 has executed two or more events, we have already seen that lexically smallest cut that satisfies B is {a, c, d}. We enumerate this cut at line 3 (in Algorithm 1) and then call GetMinCut (B, {a, c, d}). This in turn will first call

more events, and the call BackwardPass (B, 2, {a, b, c, d}). We start at chain i = 2, and remove the last event on this chain, b, from H, to get K = {a, c, d}. This cut satisfies B as it has two events c and d that were executed on P_2. We now update H = K = {a, c, d}. We then try to remove c the first event on chain µ_2 from H, but get the cut K = {a, d} that is not consistent — d’s causal dependency c is not included in this cut. Hence, H is not changed, and kept as {a, c, d}. We now move to the lower chain µ_1. We again cannot remove the only event from this chain (event a) as it will make the cut inconsistent. We now have exhausted all the chains, and thus at the end return H = {a, c, d} which is lexically smaller than G = {a, b, c, d} and satisfies B.

Algorithm 4 BackwardPass(B, start, G)

Input: B: a stable predicate, start: a chain number (from µ) such that 0 < start < n_u, G: a base cut that satisfies B.
Output: H: Lexically smallest consistent cut ≤ G that satisfies B and has H[k] = G[k] for start + 1 ≤ k ≤ n_u

1: H = G
2: for (j = start; j ≥ 1; j = j - 1) do // iterate from start argument chain to lower chains
3:    for (e = H[j]; e ≥ 1; e = e - 1) do // from last event on chain to first
4:       K = H \ {µ_j[e]} // remove event from cut
5:    if K is inconsistent then // removing the event violated consistency
6:       break // break inner loop on events to move to the lower chain
7: end if
8: end for
9: return H
GetBiggerBaseCut \((B, \{a, c, d\})\), and the result is \(\langle H = \{a, b, c, d\}, i = 2 \rangle\). The second call (in GetMinCut) is BackwardPass \((B, 1, \{a, b, c, d\})\) that returns \(G = \{a, b, c, d\}\), and we enumerate it. The next call of GetMinCut \((B, \{a, b, c, d\})\) will return null as there is no cut greater than \(\{a, b, c, d\}\). Hence, the loop will now terminate, and we have enumerated all the cuts that satisfy \(B\).

We present the proof of correctness in Appendix C. The routines presented here can be implemented without the regeneration of vector clocks for uniflow partition. We can also optimize them for improved runtime by using some additional space. We present these implementation details, and their optimizations in Appendix E.

5 Enumerating Consistent Cuts satisfying Count Predicates

Many applications involve analysis of computations based on some specific type of events. The type of an event is defined either in the context of the system under consideration, or in the context of the analysis problem. For example, we can categorize events in a message-passing computation in three base types: send event, receive event, and local event. Similarly, in a shared memory parallel computation that uses locks, we can define three base types: acquire-lock event, release-lock event, and thread-local event. Analyzing such computations may require us to check all consistent cuts that satisfy counting conditions on a type of event. For example, we may be interested in analyzing the computation when a certain number of send events have occurred, or a certain number of messages have been received.

We call such predicates count predicates. Count predicates are used in multiple debugging and analysis applications. For example, while debugging an implementation of Paxos [23] algorithm, a programmer might only be interested in analyzing possible system states when \(k\)th propose message has been sent, or \(k'\) promise messages have been delivered. Another scenario is when a programmer knows that her program exhibits a bug only after the system has executed a certain number of events. We use the notion of colors to represent types. We assume that by default each event in a computation is colored white. Then, every event of interest is assigned a color where each color represents a type categorization. Note that an event can have only when color, and on assigning a color \(c\) to it, we replace its previously assigned color. For example, in the Paxos implementation scenario discussed earlier, we may assign the color blue to all the events that send a propose message, and the color red to all the events that deliver promise messages. We then define the notion of a view of a consistent cut with respect to a color:

\begin{definition}[view(G, c)] \(\text{Let each event } e \text{ of the computation } P = (E, \to) \text{ be colored with a color } c \text{ from the set of colors } C. \text{ Then for a consistent cut } G \text{ of } P \text{ we define } \text{view}(G, c) \text{ as the set of events that are included in } G \text{ and are colored } c.\end{definition}

For example, consider the computation shown in Figure 5. The events in this computation are colored either white or blue. Given the cut \(G = \{a, b, e\}\) in this computation, we have \(\text{view}(G, \text{white}) = \{a, b, e\}\), and \(\text{view}(G, \text{blue}) = \{\}\). For \(G = \{a, b, c, d, e, f, g\}\), we get \(\text{view}(G, \text{white}) = \{a, b, c, e, g\}\), and \(\text{view}(G, \text{blue}) = \{d, f\}\).

We now use the view with respect to a color to define a count predicate.

\begin{definition}[Count Predicate] \(\text{Let } P = (E, \to) \text{ be a computation, and } c \text{ be a color from the set of colors } C. \text{ A predicate } B \text{ is called a count predicate if it can be written in the form: } |\text{view}(G, c)| = k \in \mathbb{N}, \text{ for any consistent cut } G \text{ of } P.\end{definition}
Algorithm 5 EnumerateCount((E, \rightarrow), B)

Input: Computation (E, \rightarrow) in its uniflow chain partition \(\mu\), B: a count predicate
Output: Enumerate all consistent cuts satisfying B.
1: \(G = \text{GetMinCut}(B, \{\})\) // now \(G\) is the smallest cut satisfying \(B\)
2: while \(G \neq \text{null}\) do
3: EnumSameViewCuts(B, G)
4: \(G = \text{GetSuccessor}(B, G)\)

If \(c\) is the color used in defining \(B\), then we use the notation \(\text{count}_B(G) = |\text{view}(G, c)|\).
Observe that for a count predicate \(B\), we get:

\[
\begin{align*}
&\text{count}_B(G) \leq \text{rank}(G), \\
&\text{If } H \text{ is a consistent cut such that } G \subseteq H \text{ then } \text{count}_B(H) \geq \text{count}_B(G), \\
&\text{If } K \text{ is a consistent cut such that } G \subset K \text{ and } \text{count}_B(K) > \text{count}_B(G), \text{ then } \exists H : (G \subset H \subseteq K) \land \text{count}_B(H) = \text{count}_B(G) + 1.
\end{align*}
\]

Given that \(B\) is defined with respect to one color \(c\), for brevity and ease of notation we usually write \(\text{view}(G)\) for \(\text{view}(G, c)\) when \(c\) is obvious from the context.

We now present an algorithm to enumerate all consistent cuts of a computation \((E, \rightarrow)\) that satisfy a count predicate \(B\). We use the computation’s uniflow partition \(\mu\) for enumerating these cuts in their lexical order. Algorithm 3 shows our approach outline. First we find the lexically smallest cut that satisfies \(B\). Given the properties of \(B\), we know that adding new events to any consistent cut \(G\) can either increase \(\text{count}_B(G)\) or keep it same. Thus, using the uniflow chain partition \(\mu\) we can use the GetMinCut routine from Algorithm 2 to find the lexically smallest cut that satisfies \(B\). This works because the lexically smallest cut that satisfies the count predicate \(\text{count}_B(G) = k\) is also the lexically smallest cut that satisfies the stable predicate \(\text{count}_B(G) > k - 1\). We then repeatedly enumerate lexically bigger cuts that satisfy \(B\) using two sub-routines: EnumSameViewCuts and GetSuccessor.

EnumSameViewCuts in Algorithm 4 takes two arguments: a count predicate \(B\), and a consistent cut \(G\) that satisfies \(B\). It uses the uniflow chain partition \(\mu\) to enumerate all the consistent cuts that satisfy the predicate and have the same \(\text{view}\) with respect to the color \(c\) used to define \(B\). For example, consider the predicate \(B = \text{number of blue events is 1}\), and the computation in Figure 5. Calling EnumSameViewCuts with \(G = \{a, b, e, f\}\) will enumerate three cuts: \(\{a, b, e, f\}\), \(\{a, b, e, f, g\}\), \(\{a, b, c, e, f, g\}\) as they have the same \(\text{view}\) — the same blue event \(f\) has been executed in all of them. The routine goes from lower chains to higher, and on each chain adds events in their increasing order to the cut. We know from the structure of uniflow chain partition that the resulting cut will be consistent. If it has the same \(\text{view}\), then we enumerate it. Otherwise, if the \(\text{view}\) is different, by the properties of \(B\) we know that adding more events from the same chain will also give a different \(\text{view}\) than the one we seek. Hence, we move to the chain above, and repeat the steps.

Given a consistent cut \(G\) that satisfies \(B\), GetSuccessor routine in Algorithm 5 finds a consistent cut \(H\) such that \(H\) satisfies \(B\) and \(\text{view}(G) \neq \text{view}(H)\). For example, suppose \(B = \text{number of blue events is 2}\). Then for the computation in Figure 5, given \(G = \{a, b, c, d, e, f\}\), we have GetSuccessor \((B, G) = \{a, b, e, f, g, h\}\). This is because \(\text{view}(\{a, b, c, d, e, f\})\) is
Algorithm 6 EnumSameViewCuts($B, G$)

Input: $B$: a count predicate, $G$: a consistent cut that satisfies $B$.
Output: Enumerate each consistent cut $H$ that is $\geq_1 G$ and satisfies $view(G) == view(H)$.
1. $enumerate(G)$
2. $H = G$
3. $K = G$
4. for $(i = 1; i \leq n_u; i = i + 1)$ do // go from lowest chain to highest
5.     $j = \text{index of the first event on chain } \mu_i \text{ that is not included in } H$
6. for $(j; j \leq \text{size}((\mu_i)); j = j + 1)$ do // use events not included in $G$
7.     $H = H \cup \{\mu_i[j]\}$ // add event to cut
8. if $view(H) == view(G)$ then // same view
9.     $K = H$ // update cut
10. $enumerate(K)$
11. else // $B(H) = false; count_B(H)$ must have increased
12.     $H = K$ // retain old cut
13.     break // break the inner loop on events; move to the chain above

Algorithm 7 GetSuccessor($B, G$)

Input: $B$: a count predicate, $G$: a consistent cut satisfying $B$
Output: $K$: lexically smallest consistent cut $\geq_1 G$ that satisfies $B$ and $view(G) \neq view(K)$
1. $V = view(G)$
2. $r = count_B(G)$
3. $K = G$ // Create a copy of $G$ in $K$
4. for $(i = 1; i \leq n_u; i = i + 1)$ do // lower chains to higher
5.     $ind = \text{index of the first event on chain } \mu_i \text{ that is not included in } K$
6. for $(j; ind \leq \text{size}((\mu_i)); ind = ind + 1)$ do // move forward on chain
7.     $K = K \cup \{\mu_i[ind]\}$ // add event to cut
8. if $view(K) \neq V$ then // $K$ is lexically greater than $G$ and has a different view than $G$
9.     for $(j = i - 1; j > 0; j --)$ do // first reset lower chains
10.     remove all elements on $\mu_j$ from $K$
11. // $K$ may not be consistent: fix causal dependencies on all lower chains
12. for $(j = i + 1; j \leq n_u; j = j + 1)$ do
13.     for $(k = i - 1; k > 0; k --)$ do
14.         $S = \text{causal dependencies of events from chain } \mu_j \text{ on chain } \mu_k$
15.         $K = K \cup S$
16. // $K$ is a consistent cut now, and $view(K) \neq view(G)$
17. if $B(K) == true$ then
18.     return $K$ // $K$ satisfies $B$, and is the successor cut we want
19. if $count_B(K) < r$ then // $K$ can be used to construct the lexically bigger cut that satisfies $B$
20.     return GetMinCut($B, K$)
21. return null // could not find a candidate cut

the set with two blue events: $\{d, f\}$. The next lexically bigger consistent cut that has two blue events and has a different view is the cut $\{a, b, e, f, g, h\}$ with two blue events: $f$ and $h$.

In this routine, we start at the lowest chain in a uniflow poset, and if possible increment the cut by one event on this chain. If the new cut has the same view, we move on to the next event. When we encounter an event whose addition changes the view of the resulting cut $K$, we reset the entries on lower chains, and then make $K$ consistent by satisfying all the causal dependencies. Note that at this point $view(K)$ is guaranteed to be different than
view\( (G) \). However, \( K \) may not satisfy \( B \) as it may have a lower \( \text{count}_B \). If that is the case, we make \( \text{count}_B (K) = \text{count}_B (G) \) by calling the \text{GetMinCut} routine to find lexically smallest cut that is greater than \( K \) and satisfies \( B \). If we have tried all chains and did not find a suitable cut, then \( G \) is the largest consistent cut satisfying \( B \) and we return null.

Consider the computation in Figure 6 which is in a uniflow partition. Given the predicate \( B = \text{number of blue events is 2} \), and consistent cut \( G = \{a, b, c, d, e, f, g\} \) that satisfies \( B \), consider the call of \text{GetSuccessor} \((B, G)\). We find \( V = \text{view} (G) = \{c, f\} \), and \( r = \text{count}_B (G) = 2 \), and create \( K = G \). We start from the bottom chain \( \mu_1 \) but there is no event in \( \mu_1 \) that is not included in \( K \). We move on to \( \mu_2 \) and find the next event not in \( K \): event \( g \). We add it to \( K \) at line 7, to make \( K = \{a, b, c, d, e, f, g\} \), which is bigger than \( G \) but \( \text{view} (K) = V \) as \( g \) is not a blue event. We then move on to the next event in \( \mu_2 \) which is \( h \). Adding it to \( K \) makes \( K = \{a, b, c, d, e, f, g, h\} \). Now \( K \) is bigger than \( G \) and \( \text{view} (K) = \{c, f, h\} \) which is different than \( V \). We now remove all the events (lines 9–10) from lower chain \( \mu_1 \), and get \( K = \{e, f, g, h\} \). This cut is not consistent, and we make it consistent by executing lines 12–15 and add all the causal dependencies required: \( \{a, b\} \). We now have \( K = \{a, b, c, f, g, h\} \). At line 17, we get \( \text{count}_B (K) \) which is 2; thus we have our result and we return this \( K \). Hence, \text{GetSuccessor} \((B, G) = \{a, b, c, f, g, h\}\) whose \text{view} is \( \{f, h\} \). If we call \text{GetSuccessor} \((B, \{a, b, e, f, g, h\}\), we get \( \{a, b, c, i, j\} \) whose \text{view} is \( \{c, j\} \).

We present the proof of correctness in Appendix D. The algorithms presented in this section can be implemented without regeneration of vector clocks for uniflow partition. In addition, we can optimize them for improved runtime performance using some additional space. We discuss these implementation details in Appendix E.

6 Complexity Analysis

Consider the computation \( P = (E, \rightarrow) \) whose uniflow partition \( \mu \) has \( n_u \) chains. We now present the time and space complexity of the optimized versions of our algorithms for detecting stable and count predicate for \( P \).

Using the optimized implementations discussed in Appendix E.1, we know that computing and storing the vector \( J \) requires \( \mathcal{O}(n \cdot |E|) \) time and space. This task is only performed once. After computing \( J \), each call to \text{GetBiggerBaseCut} takes \( \mathcal{O}(n \log |E|) \) time with binary search: there are \( \mathcal{O}(\log |E|) \) search iterations, and for each such iteration, we require \( \mathcal{O}(n) \) time to check if the consistent cut under consideration satisfies the predicate. Similarly, using the optimized implementation from Appendix E.2, \text{BackwardPass} takes \( \mathcal{O}(n_u \cdot n^2 \cdot \log m) \) time, where \( m = \max_{1 \leq j \leq n_u} \text{size}(\mu_j) \), in the worst case. Hence, getting a consistent cut result from \text{GetMinCut} in the representation corresponding to original chain partition takes \( \mathcal{O}((n_u \cdot n \cdot \log m + \log |E|) \cdot n) \) time in the worst case.

Based on this, we can state that for a stable predicate \( B \) enumerating all consistent cuts of \( P = (E, \rightarrow) \) that satisfy \( B \) takes \( \mathcal{O}((n_u \cdot n \cdot \log m + \log |E|) \cdot n) \) time per cut.

Let us now analyze the \text{EnumSameViewCuts} routine. Given a cut \( G \), the routine adds events not already present in \( G \) to form bigger cuts, and then checks if the cut satisfies the predicate \( B \). There are \(|E - G|\) events that are not present in \( G \). Hence, in the worst
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Space Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS [10]</td>
<td>(O\left(\frac{m^{n-1}}{n}\right))</td>
</tr>
<tr>
<td>DFS [1]</td>
<td>(O(</td>
</tr>
<tr>
<td>Lex [16]</td>
<td>(O(n))</td>
</tr>
<tr>
<td>QuickLex [5]</td>
<td>(O(n))</td>
</tr>
<tr>
<td>This paper*</td>
<td>(O((n_u +</td>
</tr>
</tbody>
</table>

**Table 1** Space complexities of algorithms for detecting a stable or count predicate in the lattice of consistent cuts; here \(m = \frac{|E|}{n}\).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS [10]</td>
<td>(O(n^2 \cdot</td>
</tr>
<tr>
<td>DFS [1]</td>
<td>(O(n^2 \cdot</td>
</tr>
<tr>
<td>Lex [16]</td>
<td>(O(n^2 \cdot</td>
</tr>
<tr>
<td>QuickLex [5]</td>
<td>(O(n \cdot \Delta \cdot</td>
</tr>
<tr>
<td>This paper*</td>
<td>(O(n \cdot</td>
</tr>
</tbody>
</table>

**Table 2** Time complexities for enumerating all consistent cuts of \(C(E)\) that satisfy a stable predicate \(B\). Here \(\Delta\) is the maximum in-degree of an event in the computation.

In the two for loops at lines 4 and 6 perform \(O(|E - G|)\) iterations in combination. Each time we form a bigger cut by adding an event, we check if the view of the cuts remains the same (at line 8). Finding \(view(H)\) requires \(O(n)\) time. Thus, \(\text{EnumSameViewCuts}\) takes \(O(n \cdot |E - G|)\) in the worst case.

We now analyze the optimized version of \(\text{GetSuccessor}\) routine. Recall that with the projection based optimization, we first call the \(\text{ComputeProjections}\) routine that takes \(O(n \cdot n_u)\) time. We need \(O(n \cdot n_u)\) space to store the computed projections. We then iterate over \(n_u\) chains, and perform \(O(n)\) work in finding \(\text{viewK}\) and then \(O(n)\) work in taking the component-wise maximum of \(\text{proj}[i-1]\) and the vector clock of event being included. Thus, in the worst case we perform \(O(n \cdot n_u)\) work before returning a result. Note that, we may call \(\text{GetMinCut}\) routine at the end to return the correct result. As per our earlier analysis, that requires additional \(O((n_u \cdot n \cdot \log m + \log |E|) \cdot n)\) time. Hence, in the worst case \(\text{GetSuccessor}\) takes \(O(n_u \cdot n \cdot \log m + \log |E|) \cdot n)\) time and requires \(O(n \cdot n_u)\) space.

In Table 1, we compare the worst-case space complexities of our optimized algorithm against those of detecting the predicate using the BFS, DFS, and Lex traversal algorithms.

Let \(S_B \in C(E)\) denote the set of consistent cuts that satisfy the stable or count predicate \(B\) for the computation \(P = (E, \rightarrow)\). Then, based on our analysis we have the following result:

**Theorem 2.** Enumerating all consistent cuts in \(S_B\) takes \(O(f \cdot |S_B|)\) time using the algorithms given in this paper; where \(f\) is a polynomial function of \(|E|\) and \(n\).

In comparison, enumerating all the cuts of \(S_B\) using the existing algorithms such as BFS, DFS, Lex (or QuickLex) may take \(O(|C(E)|)\) time in the worst case. Note that the \(|C(E)|\) can be exponentially bigger than \(|S_B|\). Table 2 compares the worst-case time complexities of these algorithms to enumerate all consistent cuts in \(S_B\) when \(B\) is stable.
Related Work

We first discuss the algorithms for traversal of cuts in the lattice of consistent cuts of a computation. Cooper and Marzullo [10] gave the first algorithm for global states enumeration which is based on breadth first search (BFS). Let \( i(P) \) denote the total number of consistent cuts of a poset \( P \). Cooper-Marzullo algorithm requires \( O(n^2 \cdot i(P)) \) time, and exponential space in the size of the input computation. Alagar and Venkatesan [1] presented a depth first algorithm using the notion of global interval which reduces the space complexity to \( O(|E|) \). Steiner [30] gave an algorithm that uses \( O(|E| \cdot i(P)) \) time, and Squire [29] further improved the computation time to \( O(log|E| \cdot i(P)) \). Pruesse and Ruskey [28] gave the first algorithm that generates global states in a combinatorial Gray code manner. The algorithm uses \( O(|E| \cdot i(P)) \) time and can be reduced to \( O(\Delta(P) \cdot i(P)) \) time, where \( \Delta(P) \) is the in-degree of an event; however, the space grows exponentially in \(|E|\). Later, Jegou et al. [21] and Habib et al. [19] improved the space complexity to \( O(n \cdot |E|) \). Ganter [15] presented an algorithm, which uses the notion of lexical order, and Garg [16] gave the implementation using vector clocks. The lexical algorithm requires \( O(n^2 \cdot i(P)) \) time but the algorithm itself is stateless and hence requires no additional space besides the poset. Paramount [4] gave a parallel algorithm to traverse this lattice in lexical order, and QuickLex [5] provides an improved implementation for lexical traversal that takes \( O(n \cdot \Delta(P) \cdot i(P)) \) time, and \( O(n^2) \) space overall.

For enumerating only the consistent cuts that satisfy a given predicate, computation slicing is an abstraction technique to reduce the state space for model checking a single program trace [26, 27, 7]. But the known efficient algorithms for computation slicing generally require the predicate to be regular. Hence, this technique does not apply to stable or count predicates.

Conclusion and Applications to Other Fields

The ubiquity of multicore and cloud computing has significantly increased the degree of parallelism in programs. This change has in turn made verification and analysis of large parallel programs even more challenging. For such verification and analysis tasks, the algorithms presented in this paper can provide much faster runtimes in comparison to existing algorithms. In many cases, this reduction in runtime can be exponential, and also allows us allows us to analyze computation with high degree of parallelism with relatively small memory footprint.

Our algorithm for detecting count predicates has a wide-ranging potential scope in analysis of parallel computations. In addition to predicate detection for verifying correctness, it can also be used to analyze logs of distributed protocols such as Paxos, and various distributed systems for performance related analysis. Further optimizations of this algorithm can provide improved runtimes for its implementation which can make it an appealing choice as a lightweight and fast component in online runtime verification systems.

Observe that many useful analysis criterion can be written in the form of stable predicates. For example, if we are interested in analyzing logs of a distributed system to identify causes of a system failure or performance degradation, we can create stable predicates that include either thresholds or upper bounds for performance load factors. By using these predicates, we can then use our algorithm (Algorithm 1) to efficiently find only those system states that are of interest to us without going through the states that came before them. A promising future application of our work is implementation of a system that accepts either a stable or count predicate and returns the set of consistent cuts satisfying it.
The applications of our algorithms extend to the problem of stable marriage [14, 18]. The stable marriage problem involves finding a stable matching of women and men and ensure that there is no pair of woman and man such that they are not married to each other but prefer each other over their matched partners. Many variations of the problem with additional constraints have been studied. Some examples include man-optimal or woman-optimal matchings, and introducing the notion of regret. We can use the algorithms developed in this paper to enumerate matchings that meet a given lower-bound or upper-bound, or any other combination of such criteria on the overall cumulative regret of the matching, or individual regrets of actors.

The notion of consistent cut of a computation, directly maps to the notion of order ideals in a lattice. Multiple problems in the field of lattice theory require enumeration of a specific level of order ideals, or a range of levels. Our algorithms can be used to enumerate order ideals of a lattice that satisfy some stable properties without visiting other levels of the lattice. Our algorithm for enumerating cuts satisfying count predicate can also be used to traversing order ideals of a sub-lattice without visiting ideals outside the sub-lattice. No known algorithm in lattice theory has the ability to perform such traversals without visiting other ideals of the lattice — whose total number can be exponentially bigger than the size of the sub-lattice of interest.

References

Fast Detection of Stable and Count Predicates in Parallel Computations


A  Representation for Lattice of Consistent Cuts

![Diagram of a lattice showing consistent cuts with set of events representation and vector clock representation.](image)

- **Figure 9** Lattice of Consistent Cuts for Figure 1

B  Visualization: Cuts satisfying a stable predicate

![Visualization showing cuts satisfying a stable predicate.](image)

- **Figure 10** Illustration: Visual representation for some stable predicate $B$: the cuts in the blue region of the lattice satisfy a stable predicate, and cuts in the white region do not.
Proof of Correctness for Algorithm 1

Lemma 3. Let \( P = (E, \rightarrow) \) be a computation, and \( B \) be a stable predicate. If \( B \) is true for any consistent cut \( G \) of \( P \), then there exists a \( l \)-base cut \( H \) where \( 1 \leq l \leq n_u \) that satisfies \( B \).

Proof. By the stability property of \( B \), we know that if \( B \) is true for any consistent cut \( K \) of a computation, then it will be true for each consistent cut \( K' \) such that \( K \subseteq K' \). We use this property to create \( H \). Since \( B(G) \) is true, we know that \( B(E) \) is true as \( G \subseteq E \) and \( B \) is stable. \( E \) itself is a \( l \)-base cut, with \( l = n_u \), as it includes all the events from all the chains. Hence, just setting \( H = E \), we have a \( l \)-base cut that satisfies \( B \). □

If \( B(G) \) is true, and \( G \) does not include all the events from the lowest chain \( \mu_1 \), then the smallest \( l \)-base cut lexically greater than \( G \) satisfying \( B \) can be formed by adding the first event from \( \mu_1 \) that is not included in \( G \). If \( G \) includes all the events from \( \mu_1 \) but not all from \( \mu_2 \), then we can find the desired \( l \)-base cut by adding the first non-included event from \( \mu_2 \), and so on (moving up chains). The steps of \texttt{GetBiggerBaseCut} encode this process.

Lemma 4. Let \( G \) be a consistent cut of a computation \( P = (E, \rightarrow) \), and \( B \) be a stable predicate. Then the cut \( H \) returned in \((H, i) = \texttt{GetBiggerBaseCut}(B, G)\) is the lexically smallest \( l \)-base cut with \( l = i - 1 \) that is lexically greater than \( G \) and satisfies \( B \).

Proof. Note that we return \( H \) = \texttt{null} only if we have added all the events to \( H \) such that \( H = E \) and \( B(H) \) is still not true. In this case, we know from the previous lemma that \( B \) never becomes true in the computation. Hence, for this case the claim trivially holds.

In \texttt{GetBiggerBaseCut}, we start with \( H = G \), and add the events of the lowest chain to \( H \) as long as the resulting cut does not satisfy \( B \). If we have added all the events from the lowest chain to \( H \) and \( B(H) \) is still false, we move to the chain immediately above and repeat the process. Given that we do not skip any event that is not already present in \( H \), and only move to a higher chain \( k \) if we have added all the events from chain \( k - 1 \) we are guaranteed that the returned cut is \( l \)-base cut for \( l = i - 1 \).

We return a non-empty \( H \) only at line 8 that is executed under the condition that \( B(H) \) is true. Hence we have established that \( H \) is a \( l \)-base cut, with \( l = i - 1 \), that satisfies \( B \). Line 8, however, is executed only once in the routine, and is executed at the first instance the condition in line 7 is true. The if condition in line 7 checks if \( B \) is satisfied for each new formed cut. Hence, we are guaranteed that if \( H \) is non-empty, it must be returned at the first instance we found an event on chain \( i \) whose addition to the \( i - 1 \)-base cut satisfies \( B \). Hence, the returned \( H \) is guaranteed to be the lexically smallest \( l \)-base cut for \( l = i - 1 \) that is greater than \( G \) and satisfied \( B \). □

Lemma 5. Let \( H \) be a consistent cut that satisfies a stable predicate \( B \). Then \( H' = \texttt{BackwardPass}(B, i - 1, H) \) is the lexically smallest consistent cut that has \( H'[j] = H'[j], i \leq j \leq n_u \) and satisfies \( B \).

Proof. If \( H \) is null, then \( H' \) will also be null as the routine \texttt{BackwardPass} only removes events from \( H \) and does not add any event to it. In this case, the claim is trivially true.

We start in \texttt{BackwardPass} with \( H' = H \) that satisfies \( B \). Subsequently in the routine, we possibly remove some events from \( H' \). In case no event was removed \( H' \), our claim holds.

Now we only need to consider the case when \( H' \neq H \). Hence, we must have removed some events from \( H \) to construct \( H' \). The outer loop (at line 2) starts the iteration with
we do not change $H'$. Hence, $H[j] = H'[j], i \leq j \leq n_u$. We remove an event $e$ from $H'$ if the resulting cut, $H' \setminus \{e\}$ remains consistent and still satisfies $B$, otherwise we retain the older version of $H'$. Hence, we know that returned $H'$ is consistent and satisfies $B$.

In the routine, we move top-down starting from chain $i - 1$. At each iteration of the outer loop on chains, our construction ensures that the cut $H'$ is consistent, and satisfies $B$.

Hence, for any consistent cut $W$ that satisfies $B$ and $W[j] = H[j]$ for $i \leq j \leq n_u$, we must have $W \geq H'$. If not, then that means that our algorithm did not remove an event on some chain numbered $k$ where $i - 1 \geq k \geq 1$ that could have been removed. But, by construction, in the inner loop on the events of chain $k$, we remove events from $H'$ in their decreasing order, and in this order remove every single event that can be removed while keeping $H'$ consistent and satisfying $B$. Hence, the assumption that $W[k] < H'[k]$ contradicts the construction of the algorithm.

\[
\text{ lemma 6.} \quad \text{Let } G \text{ be a consistent cut of a computation } P = (E, \rightarrow), \text{ and } B \text{ be a stable predicate. Then } \text{GetMinCut} (B, G) \text{ returns the lexically smallest consistent cut that is lexically greater than } G \text{ and satisfies } B.
\]

**Proof.** Let $K$ be the cut returned by $\text{GetMinCut} (B, G)$. If we have $K = \emptyset$ (ie. $\{\}$), then by Lemma 3 we know that $B$ is not satisfied by any consistent cut in the computation. The claim trivially holds.

Now, we focus on the case when $K$ is non-empty. We first show that $K$ is consistent and satisfies $B$. Let $(H, i) = \text{GetBiggerBaseCut} (B, G)$, then by definition of $\text{GetMinCut}$, we know that $K = \text{BackwardPass} (B, i - 1, H)$. Then by Lemma 4, and Lemma 5 we know that $K$ is consistent and satisfies $B$.

From $\text{GetBiggerBaseCut}$ construction $i$ is the highest chain from which we added an event to $G$. Hence, we know that $H[i] > G[i]$, and $H[j] = G[j]$ for $i < j \leq n_u$. To get $K = \text{BackwardPass} (B, i - 1, H)$, we only remove some events, if any, from chains numbered $i - 1$ or lower in $H$ to form $K$. Thus, we have $K[j] = G[j]$ for $i < j \leq n_u$, and $K[i] > G[i]$.

We now show that $K$ is the lexically smallest cut lexically greater than $G$ that satisfies $B$. Suppose not, and let $W$ be the lexically smallest consistent cut lexically greater than $G$ that satisfies $B$. Thus, we have $K > W \uparrow G$. Recall that $i$ is the chain number returned in $(H, i) = \text{GetBiggerBaseCut} (B, G)$. We claim that $W[j] = H[j] = G[j]$, for $i < j \leq n_u$. This claim is valid since we have already established earlier that that $K[j] = G[j]$ for $i < j \leq n_u$. Thus, the three consistent cuts, that is $G$, $K$, and $W$, contain same events from every chain that is higher than chain $i$. Let us now analyze the cuts for chain number $i$. Since $W > G$, we are guaranteed that $W[i] \geq G[i]$. There are only two possible cases:

**Case (1):** $W[i] \geq G[i] \land W[i] < K[i]$. Naturally, this will ensure that $W <_{l} K[i]$. Then, we form a consistent cut $W'$ by setting $W'[j] = W[j]$ for $i \leq j \leq n_u$, and $W'[j] = \text{size}(H[j])$ for $1 \leq j \leq i - 1$. This will make $W'$ a $l$-base cut with $l = i - 1$, and will give us $W' >_{l} H$. In addition, since $W$ satisfies $B$ which is a stable predicate, then $W' \supset W$ must also satisfy $B$. This makes $W'$ the lexically smallest $(i - 1)$-base cut that satisfies $B$ and is lexically greater than $G$ — a contradiction with Lemma 4.

**Case (2):** $W[i] > G[i] \land W[i] = K[i]$. Since $K[i] = H[i]$, we now have $W[j] = H[j]$ for $i \leq j \leq n_u$. Since we assumed that $W <_{l} K$, and $W$ satisfies $B$, we now have $W$ as the lexically smallest cut that is: less than or equal to $H$, satisfies $B$, and has $W[j] = H[j]$ for $i \leq j \leq n_u$. This contradicts Lemma 5.

**Lemma 7.** For a computation $P = (E, \rightarrow)$, and a stable predicate $B$, $\text{EnumerateStable}$ in Algorithm 1 enumerates all consistent cuts of $P$ that satisfy $B$. 

Proof. In EnumerateStable we start with the empty cut \( G = \{ \} \) and call GetMinCut \((B, G)\). Note that if \( B \) never becomes true, then we get the lexically smallest cut satisfying \( B \) as null and the result trivially holds.

If \( B \) ever becomes true in the computation, then by Lemma 6 we know that in the result of \( G = \text{GetMinCut} (B, \{ \}) \), at line 1, we get the lexically smallest non-trivial consistent cut that satisfies \( B \). We enumerate this cut at the first execution of line 3. We then find and enumerate the next lexically smallest cut lexically greater than \( G \) that satisfies \( B \). Proceeding in this manner, we enumerate the consistent cuts satisfying \( B \) in the lexical order — which is a total order over all consistent cuts. We continue enumerating subsequent lexically bigger cuts satisfying \( B \) without stopping unless we have reached the cut \( E \). Thus, we are guaranteed to enumerate all consistent cuts that satisfy \( B \).

D Proof of Correctness for Algorithm 3

Lemma 8. Let \( G \) be a consistent cut of a computation \( P = (E, \rightarrow) \), and \( B \) be a count predicate. Then GetMinCut \((B, G)\) in Algorithm 2 returns the lexically smallest consistent cut that is lexically greater than \( G \) and satisfies \( B \).

Proof. Note that our construction of GetMinCut is with respect to stable predicates. In this case, \( B \) is a count predicate that is of the form: \( \text{count}_{B}(G) = k \). We construct a stable predicate \( B' \) from \( B \) by setting: \( B' = \text{count}_{B}(G) > (k - 1) \). The lexically smallest cut that satisfies the count predicate \( B \) is also the lexically smallest cut that satisfies the \( B' \). Hence, we can use the GetMinCut routine to find the lexically smallest cut satisfying \( B \). The result then follows from Lemma 6.

Lemma 9. For a computation \( P = (E, \rightarrow) \) and a count predicate \( B \), let \( G \) be a consistent cut of \( P \) that satisfies \( B \). Then, EnumSameViewCuts in Algorithm 4 enumerates all consistent cuts of \( P \) that are lexically greater than \( G \), satisfy \( B \), and have same \text{view} as that of \( G \).

Proof. Algorithm 4 enumerates a cut \( H \) only if \( \text{view}(G) == \text{view}(H) \): line 1 enumerates \( G \) itself, and the only other line that enumerates a cut is line 10 that is executed only if \( \text{view}(G) == \text{view}(H) \).

We now show that the algorithm does not miss any consistent cut of \( P \) that satisfies \( B \) and has the same \text{view} as that of \( G \). We know that \( G \) is already enumerated. Suppose \( W \succ G \) is a consistent cut that satisfies \( B \) and \( \text{view}(W) == \text{view}(G) \) and is not enumerated by the algorithm. Thus, there exists an event \( e \) on some chain \( i, 1 \leq i \leq n \), that is not included in \( G \), and \( G \cup \{ e \} \) is consistent and \( \text{view}(G) == \text{view}(G + \{ e \}) \). But starting from the lowest chain (chain number 1), the algorithm adds each event not in \( G \) to \( H \), where \( H \) is initially same as \( G \). The cut \( H \) is only updated at line 9 under the condition \( H \succ G \land \text{view}(H) == \text{view}(G) \). Hence, it is impossible that iterating through all the chains, we did not find \( e \) to construct \( W \) and subsequently enumerate it.

Lemma 10. Let \( G \) be any consistent cut of computation \( P = (E, \rightarrow) \), that satisfies a count predicate \( B \). Then GetSuccessor \((B, G)\) returns the lexically smallest consistent cut greater than \( G \) that satisfies \( B \) and has a different \text{view} than that of \( G \).

Proof. Let \( W \) be the cut returned by GetSuccessor. We consider two cases. Suppose that \( W \) is null. This means that for all values of \( i \), either all event in chain \( \mu_i \) are already included in \( G \), or on inclusion of the next event in \( \mu_i, z \), the smallest consistent cut that
includes \( z \) has the same view as that of \( G \). Hence, \( G \) is lexically biggest consistent cut satisfying \( B \) such that no other bigger cut has a different view.

Now consider the case when \( W \) is the consistent cut returned at line 18 by GetMinCut\((B,K)\). We first observe that after executing line 15, \( K \) is the next lexically bigger consistent cut (of any view) after \( G \). If \( \text{count}_B(K) \) is at most \( r \), then by Lemma 6 we know that GetMinCut\((B,K)\) returns the smallest lexical consistent cut greater than \( G \) that satisfies \( B \). If for any consistent cut \( K \), \( \text{count}_B(K) \) is greater than \( r \), then by the properties of count predicates, there is no consistent cut of \( \text{count}_B \) equal to \( r \) such that it includes more events from the same chain \( i \). Thus, when calling GetMinCut at line 18 we use the largest possible value of \( i \) for which there exists a lexically bigger consistent cut than \( G \) that satisfies \( B \), and this line is executed under the if condition (of line 8) that this cut has a different view than \( \text{view}(G) \).

\[ \text{Lemma 11.} \text{ Given a computation } P = (E, \rightarrow) \text{ with its uniflow chain partition } \mu, \text{ and a count predicate } B, \text{ Algorithm 3 enumerates all consistent cuts of } P \text{ that satisfy } B. \]

\[ \text{Proof.} \text{ At line 1 in Algorithm 3 we find the lexically smallest consistent cut } G \text{ that satisfies } B. \text{ If its not null we pass it to EnumSameViewCuts that will enumerate it at its first line. By Lemma ??, we know that all subsequent cuts satisfying } B \text{ with the } \text{view}(G) \text{ will be enumerated. After this, the only consistent cuts that satisfy } B \text{ and have not been enumerated are the cuts that have a different view. In the first iteration of loop of lines 2–4, we find the lexically smallest consistent cut that is bigger than } G, \text{ satisfies } B, \text{ and has a different view. We then enumerate it and all the cuts that have the same view. Repeating this unless we cannot find a cut with a new view ensures that at the end we have enumerated all the consistent cuts that satisfy } B. \]

\[ \text{E Optimized Implementation} \]

In this section, we discuss optimized implementations of our algorithms for detecting stable and counting predicates.

First, note that we do not need to regenerate the vector clocks of the computation for its uniflow chain partition. In implementing our algorithms based on the uniflow chain partition, \( \mu \), we only reposition the events on their respective uniflow chains. There are \( n_u \) such chains, and each of them is stored as an array in which whose entries store the original vector clocks, and the state variables for each event. For example, the computation on two processes in Figure ??a is not in uniflow partition. Figure ??b shows its uniflow partition on three chains. Note that we have retained the original vector clocks of the events, and only repositioned them on three chains.

We achieve this by replicating the process described in [6]. In short, we use a vector \( G_u \) called \textit{indicator vector}, of length \( n_u \), to keep track of which event is included in \( G \). In Figure 9, we show an illustration with multiple \( G \) cuts, and their respective indicator vectors. Whenever we add an event \( e \) from chain \( \mu_i \) to \( G \) we update \( G_u[i] \) to the index of \( e \). Thus, finding the index of the first event on chain \( \mu_i \) not included in \( G \) can be implemented as \( \text{ind} = G_u[i] + 1 \), and takes constant time. Given the indicator vector \( G_u \), we can find its equivalent cut \( G \) in \( O(n_u + n^2) \) time. For details, we refer the interested reader to Section 4.2 of [6].
Fast Detection of Stable and Count Predicates in Parallel Computations

Now, for a predicate $B$ that is stable or counting, we can perform the binary search on this $J$ to find the result of GetBiggerBaseCut for this computation.

Computing and storing the vector $J$ requires $O(n \cdot |E|)$ time and space. After computing $J$, each call to GetBiggerBaseCut takes $O(n \cdot \log |E|)$ time with binary search: there are

**E.1 GetBiggerBaseCut**

In the GetBiggerBaseCut routine we add events to any cut in increasing uniflow order (Definition 4). We do not skip any event, and only return $\langle H, c \rangle$ when the cut satisfies a predicate $B$. Given a uniflow chain partition $\mu$, we can optimize the runtime for this routine by using additional $O(\log |E|)$ space.

The computation $P = (E, \rightarrow)$ on $n$ processes has $|E|$ events, and each event has a vector clock of length $n$. We first collect and store all the events in the uniflow order. Let $J$ represent the array that stores the vector clocks of events in their increasing uniflow order. Now, for $2 \leq i \leq |E|$ we compute element-wise max of vector clocks in entries $J[i]$ and $J[i-1]$, and store the result in $J[i]$. Thus, for a computation on $n$ processes $J[i]$ and $J[i-1]$ are both vector of length $n$, and we have:

$$J[i][k] = \max \{ J[i][k], J[i-1][k] \}, \quad 2 \leq i \leq |E|, \quad 1 \leq k \leq n.$$  

We can now use this vector $J$ to find the result of GetBiggerBaseCut for any predicate $B$. Moreover, given that $J$ will contain entries (vector clocks) in increasing order, we can perform binary search on it to find the result. If a predicate $B$ is stable, we perform the binary search using its evaluation (true or false) on the cuts, and return the smallest entry in $J$ on which $B$ evaluates to true.. If $B$ is a counting predicate, then we use $\text{count}_B$ to guide the binary search, and return the smallest entry in $J$ for which $\text{count}_B$ matches the requirement in $B$.

Consider the computation in Figure ??a that has four events, and its uniflow partition in Figure ??b. The increasing order on the vector clocks of all the four events is in Figure ??c. Starting from the bottom (vector $[0,1]$), and performing the joins, we get $J$ as shown in Figure ??d. Now, given a predicate $B$ that is stable or counting, we can perform the binary search on this $J$ to find the result of GetBiggerBaseCut for this computation.

(a) $n = 2, n_u = 3$


(b) $n = 3, n_u = 3$

Figure 13 Illustration: Projections of cuts on uniflow chains

$O(n \cdot \log |E|)$ iterations, and for each such iteration we take $O(n \cdot \log |E|)$ time to check the consistent cut satisfies the predicate.

E.2 BackwardPass

In BackwardPass routine, we iterate on chains in top to bottom manner, and try to remove as many events from a cut $G$ from the end of the chain as possible. We only stop removing events from a chain $i$ if $G$ becomes inconsistent or $B(G)$ becomes false on removal. Then, we move to chain $i - 1$. We can exploit the properties of stable and counting predicates, and use binary search, instead of linear search used in Algorithm ?? to remove events on each chain. This is possible possible because for a stable or counting predicate, if removal of an event from a chain makes the predicate become false (from true) then we know that removing any smaller events on that chain will never make it true. Using this implementation, BackwardPass takes $O(n_u \cdot n^2 \cdot \log m)$ time, where $m = \max_{1 \leq j \leq n_u} \text{size}(\mu_j)$, in the worst case. This is because the outer loop on the uniflow chains takes $O(n_u \cdot n^2 \cdot \log m)$ iterations in the worst case. In the inner body of this loop, we check if removal of an event makes the resulting cut inconsistent, and this check requires $O(n^2)$ time. There are $O(\log m)$ search iterations for such an event in the worst case.

E.3 GetSuccessor

We optimize the routine GetSuccessor by replicating the strategy of computing projections as per [6]. Whenever the routine is called, we compute the causal dependencies, called projections, of the input consistent cut on each chain in $\mu$, and store them in a vector called $proj$. We then use this vector to fix the causal dependencies on each chain in $O(n)$ time (see [6] for details). For this optimization, we require $O(n_u \cdot n)$ space to store the computed projections, and by using them we can find the result of GetSuccessor in $O((n_u + \log |E| + n_u \log m) \cdot n)$ time in the worst case. As $\log m \gg 1$ for most of the computations, we can simplify this bound to $O((\log |E| + n_u \log m) \cdot n)$.

We illustrate with the computation shown in Figure 10a that was originally on two processes. Suppose we want the lexical successor of $G = [1, 2]$. Then, for each chain, starting from the top, using the vector $G_u$ we compute the projection of events included in $G$ on lower chains. For the consistent cut $G = [1, 2]$, we have $G_u[3] = 0, G_u[2] = 2, G_u[1] = 1$. 

$proj[3] = [0, 0]$ $proj[2] = [1, 2]$ $proj[1] = [1, 2]$
Hence, on the top-most chain, the projection is empty and we have $\text{proj}[3] = [0, 0]$. On chain $\mu_2$, the projection must include the combined vector clocks of events included from chain $\mu_3$, and $\mu_2$. As $G_u[2] = 2$, we take the vector clock of second event on $\mu_2$, and perform a element-wise max operation for its entries and $\text{proj}[3]$. We thus get $\text{proj}[2] = [1, 2]$. We then move to chain $\mu_1$ and find the vector clock of event against entry $G_u[1] = 1$ which is the first event on $\mu_1$, with vector clock $[0, 1]$. We then set $\text{proj}[1] = \max(\text{proj}[2], [0, 1])$, which is element-wise max of two arrays $[1, 2]$, and $[0, 1]$. Thus, we get $\text{proj}[1] = [1, 2]$.

Figure 10b shows another illustration of computing the projection vector $\text{proj}$ on a computation with three processes that forms a uniflow chain partition by default.