Fast Detection of Stable and Count Predicates in Parallel Computations

Himanshu Chauhan[1] **and Vijay K. Garg**[1]

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.

1998 ACM Subject Classification C.2.4 Distributed Systems

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

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.0

1 Introduction

Predicate detection [18, 11] 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 [10, 14, 21, 25].

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 [24] 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 = $ *all promises have been delivered*, and $B = $ *at least k 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

21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão; Article No. 0; pp. 0:1–0:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

parallel programs. For example, B = *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 [11], DFS [1], and Lex [16, 17] 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 = *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. Thus, when compared to existing algorithms for enumerating such consistent cuts, the time complexities of our algorithms are significantly — exponentially for many cases — better.

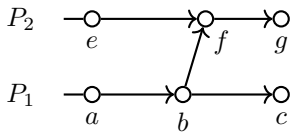


Figure 1 Computation on two processes with six events

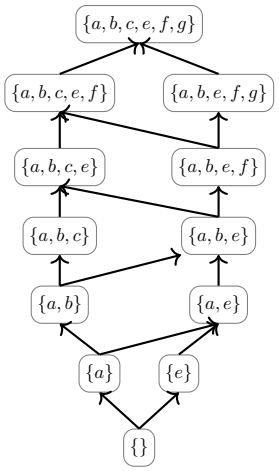


Figure 2 Lattice of Consistent Cuts for Figure 1

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 [23]. 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 α is a chain partition of poset $P = (E, \rightarrow)$ then α maps every event to a natural number such that

$$\forall x, y \in E : \alpha(x) = \alpha(y) \Rightarrow (x \rightarrow y) \vee (y \rightarrow x).$$

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 [26] and Fidge [13] 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: $[e_n, e_{n-1}, ..., e_i, ..., e_2, e_1]$ 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} 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: $e \rightarrow f \Leftrightarrow \forall j : e.V[j] \leq f.V[j] \wedge \exists k : e.V[k] < f.V[k]$. A pair of events, e and f, is concurrent iff $e \not\rightarrow f \wedge f \not\rightarrow e$. We denote this relation by $e||f$. Note that concurrent events must occur on separate processes. Figure 3 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, $e \in C \wedge f \rightarrow e \implies f \in C$.

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.

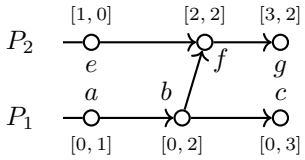


Figure 3 Vector clocks of events for computation of Figure 1

Consider the computation shown in Fig 1. The subset of events $\{a, b, e\}$ forms a consistent cut, whereas the subset $\{a, e, f\}$ 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, $G <_l H \equiv \exists k : (G[k] < H[k]) \wedge (\forall i : n \geq i > k : G[i] = H[i])$

▶ Theorem 1. [12, 26] Let $\mathcal{C}(E)$ denote the set of all consistent cuts of a computation $(E, \rightarrow)$. $\mathcal{C}(E)$ forms a finite distributive lattice under the relation $\subseteq$.

Figure 9 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 $rank(G) = \sum G[i]$. 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 a poset 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 μ is uniflow if

$$\forall x, y \in P : \mu(x) < \mu(y) \Rightarrow (y \not\rightarrow x) \tag{1}$$

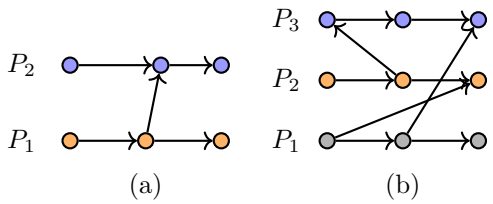


Figure 4 Posets in Uniflow Partitions

Visually, in a uniflow chain partition all the edges between separate chains always point upwards. Figure 4 shows two posets with uniflow partitions. Whereas Figure 5 shows two posets with partitions that do not satisfy the uniflow property. The poset in Figure 5a can be transformed into a uniflow partition of three chains as shown in Figure 5b. Similarly, Figure 5c can be transformed into a uniflow partition of two chains shown in Figure 5d. Observe that:

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

Proof. Any total order that is an extension of 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 μ. 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 μ 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, $e <_u f \equiv (\mu(e) < \mu(f)) \vee (\mu(e) = \mu(f) \wedge pos(e) < pos(f))$

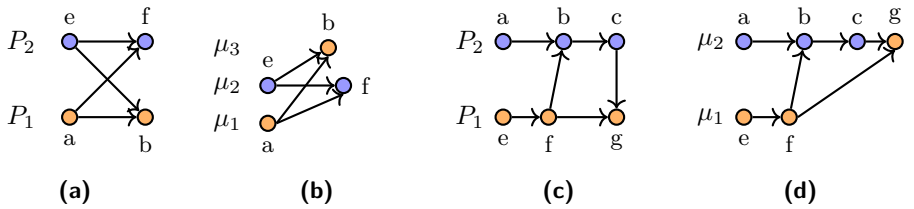


Figure 5 Posets in (a) and (c) are not in uniflow partition: but (b) and (d) respectively are their equivalent uniflow partitions

For example, in Figure 5b 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 [9, 2, 32]. 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 [9, 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 [7] 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 [7].

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 $\{\mu_i \mid 1 \leq i \leq n_u\}$, 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 4b, consider the cut $G = [1, 2, 1]$[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 5c, $G = [2, 2]$ 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 μ 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 μ. Then, we call G a *l-base cut* if $\forall j \leq l : G[j] = size(\mu_j)$

Thus, in a l-base cut we must include all the events from each chain that is same or lower than μ_l in the uniflow partition μ. In Figure 5b, $\{a, e, b\}$ (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 μ_1, but it is not a 2-base cut as it does not include all the events from the chain μ_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 $\mathcal{C}$ be the set of all consistent cuts of a computation. A predicate B defined on $\mathcal{C}$ is called stable if and only if $\forall G, H \in \mathcal{C} : G \subseteq H$ 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 10 (in Appendix, page 18) 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 $\mathcal{C}(E)$ represents the set of all consistent cuts of the computation $P = (E, \rightarrow)$. Let $S_B \subset \mathcal{C}(E)$ be the set of all consistent cuts of P that satisfy a stable predicate B. We use P's uniflow partition μ 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 $G <_l H \equiv \exists k : (G[k] < H[k]) \wedge (\forall i : n_u \geq i > k : G[i] = H[i])$.

We use the 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.

[1] Recall that in our vector clock notation ith entry from the right in the vector clock represents the events included from ith chain from the bottom in the uniflow chain partition.

Algorithm 1 EnumerateStable$((E, \rightarrow), B)$

Input: Computation $(E, \rightarrow)$ 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 >_l G satisfying B
```

Algorithm 2 GetMinCut(B, G)

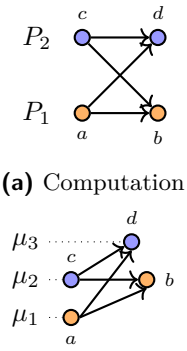
Input: B: a stable predicate, G: a consistent cut
Output: lexically smallest consistent cut $>_l G$ that satisfies B

```
1: ⟨H, c⟩ = GetBiggerBaseCut(B, G)
2: return BackwardPass(B, c − 1, H)
```

Given a consistent cut G, and a stable predicate B, we use the 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: GetBiggerBaseCut and BackwardPass.

The GetBiggerBaseCut routine in Algorithm 3 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 we return $\langle \mathsf{null}, -1 \rangle$. We start by copying G into H, and from the lowest chain, $i = 1$, add events to H that are not included in it. Each time we add an event e (not already present in H) to H, we form a bigger consistent cut, and then check if this H satisfies B. Note that we move from lower chains to higher, and by the property of uniflow chain partition, we know that adding events in this order will not violate any causal dependencies and keep the cut consistent. At the first instance of finding a bigger cut that satisfies B, we stop and return the pair $\langle H, i \rangle$, where i is the chain number in μ on which we found e. If we consume all the events from a chain, we move to the chain immediately above and repeat this process.

Let us illustrate the execution with an example. Consider the computation in Figure 6b and the predicate B=P_2 *has executed two or more events*, and the call GetBiggerBaseCut $(B, \{c\})$. We use the uniflow partition, and starting at μ_1, 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 P_1 in the computation. We now jump to chain μ_2, and find the first event on μ_2 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 $\langle H = \{a, b, c, d\}, i = 3 \rangle$.



(a) Computation

(b) Uniflow Partition

Figure 6 A computation on two processes in: (a) its original non-uniflow partition, (b) equivalent uniflow partition

The BackwardPass routine (in Algorithm 4) 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: $\{G' \subseteq G : G'[j] =$

Algorithm 3 GetBiggerBaseCut(B, G)

Input: B: a stable predicate, G: a consistent cut
Output: pair $\langle H, i\rangle$: 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:     // H is guaranteed to be lexically greater than G now
7:     if B(H) then // if H satisfies B
8:       return ⟨H, i⟩ // return H and chain number of the event
9: return ⟨null,-1⟩ // no cut lexically greater than G and satisfying B was found
```

$G[j], start + 1 \leq j \leq n_u\}$. Thus, H is the lexically smallest consistent cut $H \leq_l G$ that satisfies B such that H and G include the same set of events from chains $start + 1$ and higher. Note that whenever $start = n_u$, we have $start + 1 > n_u$, 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 6b and the predicate B=P_2 *has executed two or 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 $\leq G$ that satisfies B and has $H[k] = G[k]$ for $start + 1 \leq k \leq n_u$.

```
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:     // K must be consistent now
8:     if B(K) then // K is consistent, smaller than G, and satisfies B
9:       H = K // update H to this cut
10: return H
```

For the computation in Figure 6b 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 $\langle H = \{a, b, c, d\}, i = 3\rangle$ 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 6b 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 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 for these algorithms in this paper's extended version [6]. 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 B.

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 [24] algorithm, a programmer might only be interested in analyzing possible system states when kth *propose* message has been sent, or k' *promise* messages have been delivered. Another scenario is when a programmer knows that a 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 one 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:

▶ Definition 7 ($view(G, c)$). Let each event e of the computation $P = (E, \rightarrow)$ be colored with a color c from the set of colors C. Then for a consistent cut G of P we define $view(G, c)$ as the set of events that are included in G and are colored c.

For example, consider the computation shown in Figure 7. The events in this computation are colored either white or blue. Given the cut $G = \{a, b, e\}$ in this computation, we

have $view(G, white) = \{a, b, e\}$, and $view(G, blue) = \{\}$. For $G = \{a, b, c, d, e, f, g\}$, we get $view(G, white) = \{a, b, c, e, g\}$, and $view(G, blue) = \{d, f\}$.

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

▶ Definition 8 (Count Predicate). Let $P = (E, \rightarrow)$ be a computation, and c be a color from the set of colors C. A predicate B is called a count predicate if it can be written in the form: $|view(G, c)| = k \in \mathbb{N}$, for any consistent cut G of P.

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

- $count_B(G) \leq rank(G)$.
- If H is a consistent cut such that $G \subseteq H$ then $count_B(H) \geq count_B(G)$.
- If K is a consistent cut such that $G \subset K$ and $count_B(K) > count_B(G)$, then $\exists H : (G \subset H \subseteq K) \ \wedge count_B(H) = count_B(G) + 1$.

Given that B is defined with respect to one color c, for brevity and ease of notation we usually write $view(G)$ for $view(G, c)$ 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 μ for enumerating these cuts in their lexical order. Algorithm 5 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 $count_B(G)$ or keep it same. Thus, using the uniflow chain partition μ 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 $count_B(G) = k$ is also the lexically smallest cut that satisfies the stable predicate $count_B(G) > k - 1$. We then repeatedly enumerate lexically bigger cuts that satisfy B using two sub-routines: EnumSameViewCuts and GetSuccessor.

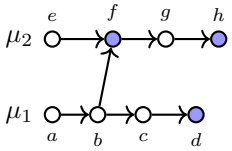


Figure 7 A computation in uniflow partition

EnumSameViewCuts in Algorithm 6 takes two arguments: a count predicate B, and a consistent cut G that satisfies B. It uses the uniflow chain partition μ to enumerate all the consistent cuts that satisfy the predicate and have the same *view* with respect to the color c used to define B. For example, consider the predicate B=*number of blue events is* 1, and the computation in Figure 7. 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 *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 *view*, then we enumerate it. Otherwise, if the *view* is different, by the properties of B we know that adding more events from the same chain will also give a different *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 7 finds a consistent cut H such that H satisfies B and $view(G) \neq view(H)$. For example, suppose $B =$ *number of blue events is* 2. Then for the computation in Figure 7, given $G = \{a, b, c, d, e, f\}$, we have GetSuccessor $(B, G) = \{a, b, e, f, g, h\}$. This is because $view(\{a, b, c, d, e, f\})$ is the set with two blue events: $\{d, f\}$. The next lexically bigger consistent cut that has two

Algorithm 5 EnumerateCount$((E, \rightarrow), B)$

Input: Computation $(E, \rightarrow)$ in its uniflow chain partition μ, B: a count predicate
Output: Enumerate all consistent cuts satisfying B.

```
1: G = GetMinCut(B, {}) // now G is the smallest cut satisfying B
2: while G ≠ null do
3:    EnumSameViewCuts(B, G)
4:    G = GetSuccessor(B, G)
```

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 $\geq_l G$ and satisfies $view(G) == view(H)$.

```
1: enumerate(G)
2: H = G
3: K = G
4: for (i = 1; i ≤ n_u; i = i + 1) do // go from lowest chain to highest
5:    j = index of the first event on chain μ_i that is not included in H
6:    for (; j ≤ size(μ_i); j = j + 1) do // use events not included in G
7:       H = H ∪ {μ_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 $>_l G$ 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 ≤ n_u; i++) do  // lower chains to higher
5:    ind = index of the first event on chain μ_i that is not included in K
6:    for (; ind ≤ size(μ_i); ind = ind + 1) do // move forward on chain
7:       K = K ∪ {μ_i[ind]} // add event to cut
8:       if view(K) ≠ 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 μ_j from K
11:         //K may not be consistent: fix causual dependencies on all lower chains
12:         for (j = i + 1; j ≤ n_u; j + +) do
13:            for (k = i − 1; k > 0; k − −) do
14:               S = causal dependencies of events from chain μ_j on chain μ_k
15:               K = K ∪ S
16:         //K is a consistent cut now, and view(K) ≠ 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
```

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 $count_B$. If that is the case, we make $count_B(K) == count_B(G)$ by calling the 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 8 which is in a uniflow partition. Given the predicate $B =$ *number of blue events is* 2, and consistent cut $G = \{a, b, c, d, e, f\}$ that satisfies B, consider the call of GetSuccessor (B, G). We find $V = view(G) = \{c, f\}$, and $r = count_B(G) = 2$, and create $K = G$. We start from the bottom chain μ_1 but there is no event in μ_1 that is not included in K. We move on to μ_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 $view(K) == V$ as g is not a blue event. We then move on to the next event in μ_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 $view(K) = \{c, f, h\}$ which is different than V. We now remove all the events (lines 9–10) from lower chain μ_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 $K = \{a, b, e, f, g, h\}$. At line 17, we get $count_B(K)$ which is 2; thus we have our result and we return this K. Hence, GetSuccessor $(B, G) = \{a, b, e, f, g, h\}$ whose *view* is $\{f, h\}$. If we call GetSuccessor $(B, \{a, b, e, f, g, h\})$, we get $\{a, b, c, i, j\}$ whose *view* is $\{c, j\}$.

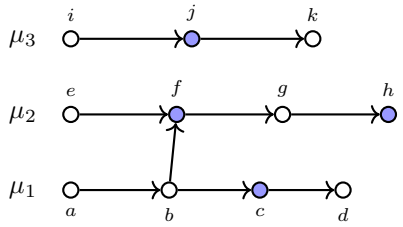


Figure 8 A computation in uniflow partition

We present the proof of correctness for these algorithms in this paper's extended version [6]. 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 B.

6 Complexity Analysis

Consider the computation $P = (E, \rightarrow)$ whose uniflow partition μ 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 B.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 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 B.2, BackwardPass takes

Algorithm	Space Required
BFS [11]	$\mathcal{O}(\frac{m^{n-1}}{n})$
DFS [1]	$\mathcal{O}(\|E\|)$
Lex [17]	$\mathcal{O}(n)$
QuickLex [5]	$\mathcal{O}(n)$
This paper*	$\mathcal{O}((n_u + \|E\|) \cdot n)$

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

$\mathcal{O}(n_u \cdot n^2 \cdot \log m)$ time, where $m = \max_{1 \le j \le n_u} size(\mu_j)$, in the worst case. Hence, getting a consistent cut result from 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 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 at $|E - G|$ events that are not present in G. Hence, in the worst case the two for loops at lines 4 and 6 perform $\mathcal{O}(|E - G|)$ 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 $\mathcal{O}(n)$ time. Thus, EnumSameViewCuts takes $\mathcal{O}(n \cdot |E - G|)$ in the worst case.

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

7 Related Work

We first discuss the algorithms for traversal of cuts in the lattice of consistent cuts of a computation. Cooper and Marzullo [11] gave the first algorithm for global states enumeration

Algorithm	Time
BFS [11]	$\mathcal{O}(n^2 \cdot \|\mathcal{C}(E)\|)$
DFS [1]	$\mathcal{O}(n^2 \cdot \|\mathcal{C}(E)\|)$
Lex [17]	$\mathcal{O}(n^2 \cdot \|\mathcal{C}(E)\|)$
QuickLex [5]	$\mathcal{O}(n \cdot \Delta \cdot \|\mathcal{C}(E)\|)$
This paper*	$\mathcal{O}(n \cdot \|S_B\| \cdot (n_u \cdot n \cdot \log m + \log \|E\|))$

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

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 $\mathcal{O}(n^2 \cdot i(P))$ 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 $\mathcal{O}(|E|)$. Steiner [31] gave an algorithm that uses $\mathcal{O}(|E| \cdot i(P))$ time, and Squire [30] further improved the computation time to $\mathcal{O}(log|E| \cdot i(P))$. Pruesse and Ruskey [29] gave the first algorithm that generates global states in a combinatorial Gray code manner. The algorithm uses $\mathcal{O}(|E| \cdot i(P))$ time and can be reduced to $\mathcal{O}(\Delta(P) \cdot i(P))$ time, where $\Delta(P)$ is the in-degree of an event; however, the space grows exponentially in $|E|$. Later, Jegou et al. [22] and Habib et al. [20] improved the space complexity to $\mathcal{O}(n \cdot |E|)$. Ganter [16] presented an algorithm, which uses the notion of lexical order, and Garg [17] gave the implementation using vector clocks. The lexical algorithm requires $\mathcal{O}(n^2 \cdot i(P))$ 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 $\mathcal{O}(n \cdot \Delta(P) \cdot i(P))$ time, and $\mathcal{O}(n^2)$ 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 [27, 28, 8]. 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.

8 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 predi-

cates. 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 [15, 19]. 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

1. S. Alagar and S. Venkatesan. Techniques to Tackle State Explosion in Global Predicate Detection. *IEEE Transactions on Software Engineering (TSE)*, 27(8):704–714, August 2001.
2. Lucio Bianco, Paolo Dell 'Olmo, and Stefano Giordani. An optimal algorithm to find the jump number of partially ordered sets. *Computational Optimization and Applications*, 8(2):197–210, 1997.
3. K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
4. Yen-Jung Chang and Vijay K. Garg. A parallel algorithm for global states enumeration in concurrent systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 140–149. ACM, 2015.
5. Yen-Jung Chang and Vijay K. Garg. Quicklex: A fast algorithm for consistent global states enumeration of distributed computations. In *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, pages 25:1–25:17, 2015.
6. Himanshu Chauhan and Vijay K. Garg. Fast detection of stable and counting predicates in parallel computations. *Extended Version*, 2017. URL: `http://users.ece.utexas.edu/~garg/dist/opodis17.pdf`.
7. Himanshu Chauhan and Vijay K. Garg. Space efficient breadth-first and level traversals of consistent global states of parallel programs. In *17th International Conference on Runtime Verification (RV 2017)*, pages 138–154, 2017.

8 Himanshu Chauhan, Vijay K Garg, Aravind Natarajan, and Neeraj Mittal. A distributed abstraction algorithm for online predicate detection. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 101–110. IEEE, 2013.

9 M Chein and M Habib. The jump number of dags and posets: an introduction. *Annals of Discrete Mathematics*, 9:189–194, 1980.

10 Feng Chen, Traian Florin Serbanuta, and Grigore Roşu. jPredictor: a predictive runtime analysis tool for java. In *Proceedings of the International Conference on Software Engineering*, pages 221–230, 2008.

11 R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

12 B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

13 C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial-Ordering. In K. Raymond, editor, *Proceedings of the 11th Australian Computer Science Conference (ACSC)*, pages 56–66, February 1988.

14 Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

15 David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

16 Bernhard Ganter. Two basic algorithms in concept analysis. In *Proceedings of the International Conference on Formal Concept Analysis*, pages 312–340, 2010.

17 Vijay K. Garg. Enumerating global states of a distributed computation. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 134–139, 2003.

18 Vijay K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, 1994.

19 Dan Gusfield and Robert W Irving. *The stable marriage problem: structure and algorithms*. MIT press, 1989.

20 Michel Habib, Raoul Medina, Lhouari Nourine, and George Steiner. Efficient algorithms on distributive lattices. *Discrete Appl. Math.*, 110(2-3):169–187, 2001.

21 Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

22 Roland Jegou, Raoul Medina, and Lhouari Nourine. Linear space algorithm for on-line detection of global predicates. In *Proc. of the International Workshop on Structures in Concurrency Theory*, pages 175–189, 1995.

23 L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

24 Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

25 Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

26 F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 215–226, 1989.

27 N. Mittal and V. K. Garg. Techniques and Applications of Computation Slicing. *Distributed Computing (DC)*, 17(3):251–277, March 2005.

28 N. Mittal, A. Sen, and V. K. Garg. Solving Computation Slicing using Predicate Detection. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 18(12):1700–1713, December 2007.

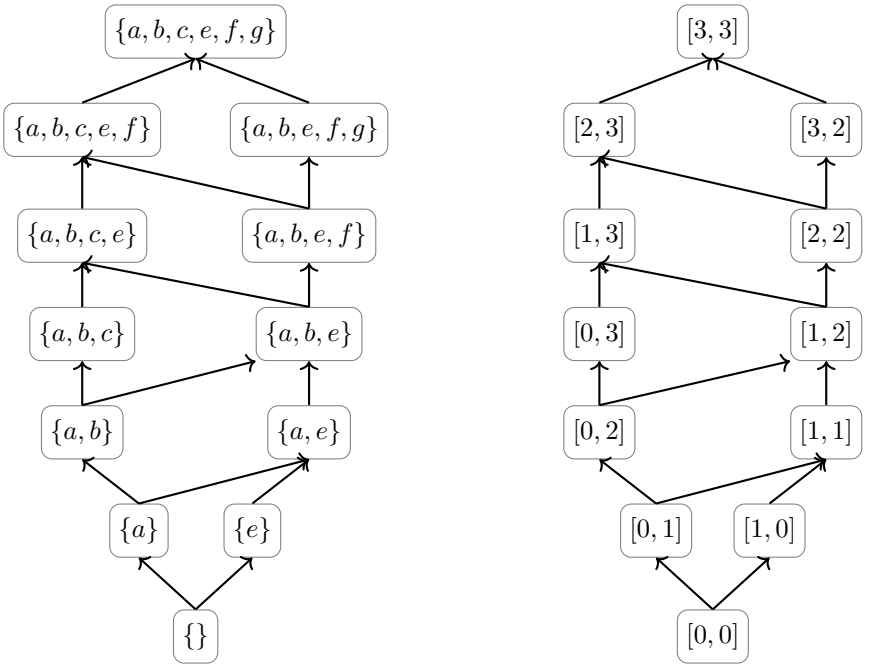
29 Gara Pruesse and Frank Ruskey. Gray codes from antimatroids. *Order 10*, pages 239–252, 1993.

30 Matthew B. Squire. Enumerating the ideals of a poset. In *PhD Dissertation, Department of Computer Science, North Carolina State University*, 1995.

31 George Steiner. An algorithm to generate the ideals of a partial order. *Oper. Res. Lett.*, 5(6):317–320, 1986.

32 Maciej M Sysło. Minimizing the jump number for partially ordered sets: A graph-theoretic approach. *Order*, 1(1):7–19, 1984.

A Visual Illustrations for Lattice of Consistent Cuts



(a) Set of Events Representation **(b)** Vector Clock Representation

Figure 9 Lattice of Consistent Cuts for Figure 1

B 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, μ, 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 12a is not in uniflow partition. Figure 12b shows its uniflow partition on

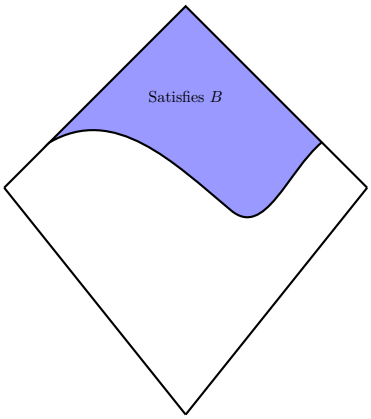
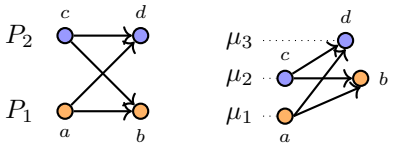


Figure 10 Illustration: Visual representation for some stable predicate B: the cuts in the blue region of the lattice satisfy a stable predicate, and cuts in the white region do not.

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 [7]. In short, we use a vector G_u, called *indicator vector*, of length n_u, to keep track of which event is included in G. In Figure 11, we show an illustration with multiple G cuts, and their respective indicator vectors. Whenever we add an event e from chain μ_i to G we update $G_u[i]$ to the index of e. Thus, finding the index of the first event on chain μ_i not included in G can be implemented as $ind = G_u[i] + 1$, and takes constant time. Given the indicator vector G_u, we can find its equivalent cut G in $\mathcal{O}(n_u + n^2)$ time. For details, we refer the interested reader to Section 4.2 of [7].



(a) Computation

(b) Uniflow Partition

$G = \{a, b, c, d\} \implies G_u[0] = 1, G_u[1] = 2, G_u[2] = 1$
$G = \{a, c, d\} \implies G_u[0] = 1, G_u[1] = 1, G_u[2] = 1$
$G = \{a, c, b\} \implies G_u[0] = 1, G_u[1] = 2, G_u[2] = 0$
$G = \{a, c\} \implies G_u[0] = 1, G_u[1] = 1, G_u[2] = 0$
$G = \{a\} \implies G_u[0] = 1, G_u[1] = 0, G_u[2] = 0$

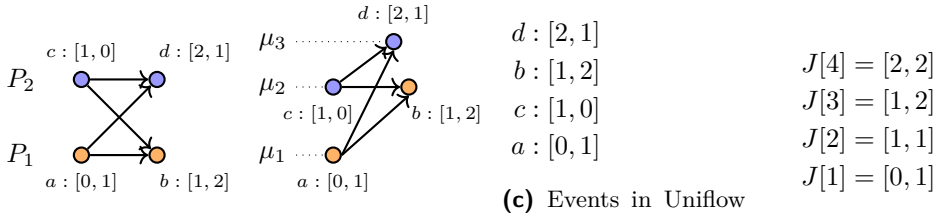
(c) G values and their respective G_u vectors

Figure 11 Illustration: Maintaining indicator vector G_u for a cut G

B.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 μ, we can optimize the runtime for this routine by using additional $\mathcal{O}(n \cdot |E|)$ 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 $J[i-1]$



(a) Computation **(b)** Uniflow Partition **(c)** Events in Uniflow Order **(d)** J Vector

Figure 12 Illustration: Computing J vector for optimizing GetBiggerBaseCut

are both vector of length n, and we have:

$$J[i][k] = \max\left(J[i][k], J[i-1][k]\right), \quad 2 \leq i \leq |E|, 1 \leq k \leq n.$$

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 $count_B$ to guide the binary search, and return the smallest entry in J for which $count_B$ matches the requirement in B.

Consider the computation in Figure 12a that has four events, and its uniflow partition in Figure 12b. The increasing order on the vector clocks of all the four events is in Figure 12c. Starting from the bottom (vector $[0, 1]$), and performing the joins, we get J as shown in Figure 12d. 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.

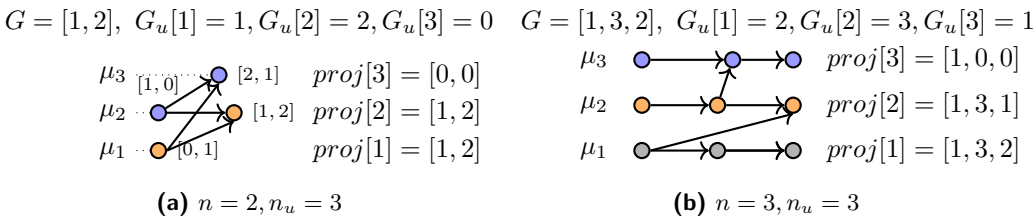
Computing and storing the vector J requires $\mathcal{O}(n \cdot |E|)$ time and space. After computing J, each call to GetBiggerBaseCut takes $\mathcal{O}(n \cdot \log |E|)$ time with binary search: there are $\mathcal{O}(n \cdot \log |E|)$ iterations, and for each such iteration we take $\mathcal{O}(n \cdot \log |E|)$ time to check the consistent cut satisfies the predicate.

B.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 4 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 $\mathcal{O}(n_u \cdot n^2 \cdot \log m)$ time, where $m = \max_{1 \leq j \leq n_u} size(\mu_j)$, in the worst case. This is because the outer loop on the uniflow chains takes $\mathcal{O}(n_u \cdot n^2 \cdot \log m)$ 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 $\mathcal{O}(n^2)$ time. There are $\mathcal{O}(\log m)$ search iterations for such an event in the worst case.

B.3 GetSuccessor

We optimize the routine GetSuccessor by replicating the strategy of computing projections as per [7]. Whenever the routine is called, we compute the causal dependencies, called projections, of the input consistent cut on each chain in μ, and store them in a vector called $proj$. We then use this vector to fix the causal dependencies on each chain in $\mathcal{O}(n)$ time (see [7] for details). For this optimization, we require $\mathcal{O}(n_u \cdot n)$ space to store the computed projections, and by using them we can find the result of GetSuccessor in $\mathcal{O}((n_u + \log|E| + n_u \log m) \cdot n)$ time in the worst case. As $\log m > 1$ for most of the computations, we can simplify this bound to $\mathcal{O}((\log|E| + n_u \log m) \cdot n)$.



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

Figure 13 Illustration: Projections of cuts on uniflow chains

We illustrate with the computation shown in Figure 13a 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 $G_u[3] = 0, G_u[2] = 2, G_u[1] = 1$. Hence, on the top-most chain, the projection is empty and we have $proj[3] = [0,0]$. On chain μ_2, the projection must include the combined vector clocks of events included form chain μ_3, and μ_2. As $G_u[2] = 2$, we take the vector clock of second event on μ_2, and perform a element-wise max operation for its entries and $proj[3]$. We thus get $proj[2] = [1,2]$. We then move to chain μ_1 and find the vector clock of event against entry $G_u[1] = 1$ which is the first event on μ_1, with vector clock $[0,1]$. We then set $proj[1] = \max(proj[2], [0,1])$, which is element-wise max of two arrays $[1,2]$, and $[0,1]$. Thus, we get $proj[1] = [1,2]$.

Figure 13b shows another illustration of computing the projection vector $proj$ on a computation with three processes that forms a uniflow chain partition by default.