The Dissertation Committee for Himanshu Chauhan
certifies that this is the approved version of the following dissertation:

Algorithms for Analyzing Parallel Computations

Committee:

Vijay Garg, Supervisor

Christine Julien

Neeraj Mittal

Evdokia Nikolova

Keshav Pingali

Vijay Reddi

Algorithms for Analyzing Parallel Computations

by

Himanshu Chauhan, B.Tech., M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2017

Dedicated to my parents Beena and Devendra.

Acknowledgments

I am indebted to my advisor Vijay Garg for making my PhD an enjoyable, fulfilling, and humbling experience. His infectious curiosity, and his relentless passion for research have left an indelible mark on me. Having him as an advisor has improved my knowledge tremendously, and at the same time has made me acutely aware of how little I know. Over these past six years, I have often found myself marveling at his ability to remain unflappable in instances — so many that I stopped keeping track — in which I made preposterous conjectures, or wrote proofs riddled with careless mistakes. How he manages to do so consistently with everyone eludes me to this day, but I will always aspire to achieve this quality. In multiple ways, he has made me a better person and for that I owe him enormous gratitude.

I thank the members of my dissertation committee for their influence on my research as well as on me. I want to thank Keshav for being a remarkable teacher who distills deep theoretical and practical knowledge in his lectures, and inspires ambition in every student. As I write this dissertation, I am filled with relief and gratitude towards him for disabusing me of my misplaced goal of improving both theory and practice of the field in one thesis. I thank Christine for asking questions whose answers revealed the importance of contributing to research that makes something better for someone somewhere on this planet. I thank Eddie for allowing me an opportunity to assist her in teaching. Interacting with her significantly improved my understanding of algorithmic theory, and made me a better teacher. Writing papers with Neeraj was an education in combining thoroughness with elegance. I also learnt many aspects of lock-free concurrency from him and from his papers, and for all of these learnings I am thankful to him. I thank Vijay (Reddi) for insisting that I take a course to learn lower level aspects of computer architecture, which in turn influenced me to design algorithms while keeping their practical implementations in mind.

I am grateful to Greg Plaxton for his phenomenal teachings in algorithms and game theory. In addition to teaching me the fundamental concepts of these fields, he also left a lasting impression on me by his meticulous attention to detail and emphasis on clarity of expression. Among all the critical remarks and wisdom I have ever received, he gave probably the best one in: "Nobody in the history of humankind has ever published a paper in which Figure 3 comes before Figure 2."

Having Yen-Jung Chang and Wei-Lun Hung as colleagues in the lab, and working with them, was a fantastic experience and I thank them for it. I thank RoseAnna Goewey, Cayetana Garcia, and Melanie Gulick for their patient help and support in numerous administrative tasks.

I feel extremely fortunate that in life I met and befriended: Abhinav Parate, Anuj Madaria, Ashwin Kulkarni, Bharath Balasubramanian, Engin Hassamanci, Gurpreet Singh, Hannah Bronsnick, Natalia Arzeno, Natalie

Hansen, Nivedita Singh, Raghavendra Singh, Sangeetha Iyer, Stephanie Taylor, Susannah Volpe, and Vinit Ogale. At one juncture or the other during my PhD, with admonitions or encouragements, with presence silent or loud, with conversations long or short, with gestures big or small, they have enhanced the joys of life and made its travails less difficult. I am grateful to every single one of them for being the person they are, and I will forever cherish their friendship.

My parents, along with my aunts Meena and Nisha, gave up a lot in their own lives to provide me with the luxury of following my dreams. I cannot find words to do justice to the magnitude of their sacrifices. This dissertation would not be possible without them, and wherever I reach in life I will owe it to them.

Algorithms for Analyzing Parallel Computations

Publication No. ________________

Himanshu Chauhan, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Vijay Garg

Predicate detection is a powerful technique to verify parallel programs. Verifying correctness of programs using this technique involves two steps: first we create a partial order based model, called a *computation*, of an execution of a parallel program, and then we check all possible global states of this model against a predicate that encodes a faulty behavior. A partial order encodes many total orders, and thus even with one execution of the program we can reason over multiple possible alternate execution scenarios. This dissertation makes algorithmic contributions to predicate detection in three directions.

Enumerating all consistent global states of a computation is a fundamental problem requirement in predicate detection. Multiple algorithms have been proposed to perform this enumeration. Among these, the breadth-first search (BFS) enumeration algorithm is especially useful as it finds an erroneous consistent global state with the least number of events possible. The traditional algorithm for BFS enumeration of consistent global states was given more than two decades ago and is still widely used. This algorithm, however, requires space that in the worst case may be exponential in the number of processes in the computation. We give the first algorithm that performs BFS based enumeration of consistent global states of a computation in space that is polynomial in the number of processes.

Detecting a predicate on a computation is a hard problem in general. Thus, in order to devise efficient detection and analysis algorithms it becomes necessary to use the knowledge about the properties of the predicate. We present algorithms that exploit the properties of two classes of predicates, called *stable* and *counting* predicates, and provide significant reduction — exponential in many cases — in time and space required to detect them.

The technique of *computation slicing* creates a compact representation, called *slice*, of all global states that satisfy a class of predicates called *regular* predicate. We present the first distributed and online algorithm to create a slice of a computation with respect a regular predicate. In addition, we give efficient algorithms to create slices of two important temporal logic formulas even when their underlying predicate is not regular but either the predicate or its negation is efficiently detectable.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel programming has become integral to modern computing. Whether it is through multiple cores on a single machine, or through harvesting the power of many machines in a distributed system, employing parallelism is now essential to create scalable software systems that solve practical problems of computing. Parallel programs, however, are not only difficult to design and implement, but once implemented are also difficult to debug and verify. This difficulty arises from the *state space explosion*: the combination of independent local states of the involved processes causes a multiplicative effect that leads to exponentially many possible system states that need to be verified. Thus, verifying correctness of parallel programs using the traditional approaches can be a hard problem. Let us first discuss the two traditional approaches that are primarily used to verify software systems: *formal methods* and *runtime verification/testing*.

In the methodology of formal methods, we model a system and its properties with mathematical constructs and then analyze the resulting model for correctness. Formal methods have two key branches: *model checking* and *theorem proving*. Model checking [20, 21] models the system as a finite state machine whose specifications are encoded using the language of *temporal logic* [18, 19]. Theorem proving [22, 27] admits a wider variety of logic languages for specifying the system, and proves the validity of the system as theorems under its specifications. Despite the exhaustive nature of both of these approaches, they suffer from drawbacks that limit their practical applicability. Model checking is prone to the state space explosion, and does not scale well with the size of the components involved in the problem. Theorem proving, on the other hand, requires intensive manual effort, and is difficult to automate. In addition, and somewhat counter-intuitively, formally verified implementations remain prone to errors and bugs. In performing formal verification of programs, we generally make certain assumptions about the context, as well as parameters of programs. When these programs are deployed the interaction involved with other components of the system or users may invalidate those assumptions. For example, multiple formally verified implementations of distributed systems have been shown to invalidate verification guarantees and exhibit bugs due to incorrect assumptions [29].

Runtime verification, or testing, involves monitoring the system execution, and extracting information from this execution to detect violation of critical properties. We verify the system for correctness by comparing the observed states of the system against those that are expected as per its specification. The verification is called *online* when system is monitored and verified during its execution itself, and *offline* if we only collect the information during the execution and perform the analysis later. This methodology is simple, and is performed on the actual system implementation — thus we directly verify the correctness of the program that gets executed and not its abstract model. Testing parallel programs, however, is not straight-forward: a single run of the program may not exhibit a concurrency related bug, and multiple runs of the same program may lead to different observations. The primary cause of this problem is the inherent non-determinism of parallel executions. On shared memory based parallel programs this non-determinism is introduced by thread scheduling; whereas in distributed systems is is caused by the asynchrony between process clocks and instruction cycles. A possible solution is to execute the program repeatedly, with the hope that multiple separate executions will produce at least a few different observations. On shared memory machines, we can control these executions by using a controllable thread scheduler to ensure that each new run of the program explores some new thread interleavings [53, 67, 46]. We can further prune away already explored states by using techniques such as partial order reduction [53, 67]. Even then, we are forced to execute the program repeatedly to increase the *coverage* of the test cases.

Let us now discuss a third technique that combines the benefits of model checking and runtime verification. This technique is called *predicate detection* [36, 23]. It allows inference based analysis to check many possible system states based on a single execution of the program. In this way, it combines the simplicity and effectiveness of runtime verification with the aspects of model checking — not only the observed execution but other possible executions are also verified. Verifying parallel programs that use paradigms such as lock-free data structures [39] or delegated critical sections [8, 56, 25, 41, 42] is even more difficult in comparison to verifying traditional lock-based parallel programs. This is because absence of lock-based critical sections generally leads to increased concurrency. Hence, even if the algorithms and data structures using these paradigms are proven to be correct, their actual implementations may exhibit bugs due to the weak consistency guarantees of lower level hardware instructions used in them. Thus, detecting bugs in the actual implementations of parallel programs becomes even more crucial. Given that predicate detection does not make assumptions about the implementations, and performs predictive analysis on observed as well as inferred executions, it can be extremely beneficial for these verification tasks. We now expand upon the details involved in the technique.

1.1 Predicate Detection

A global predicate, often just called predicate, is a boolean formula on a global state of the system. Hence, for any state of the system during the program execution, the evaluation of the predicate will either be false or true. In the technique of predicate detection, we require as input the predicate(s) that specifies the constraint(s) or invariant(s) using the system properties. We then use a single run, often called a *trace*, of a parallel program, and from it construct all possible valid states of the system. For each state we check if the predicate could possibly become true. If yes, then we output that state as a counter-example.

We observe a trace of a parallel program as the events executed by the processes. On these events, we impose a partial order based on Lamport's *happened-before*, [44] relation which is denoted by $\rightarrow$. This relation captures causal dependencies between the events. On the set of events of a computation, it is the smallest relation that satisfies the following three conditions: (1) If a and b are events in the same process and a is executed before b, then $a \rightarrow b$. (2) For a distributed system, if a is the sending of a message and b is the receipt of the same message, then $a \rightarrow b$. For a shared memory system, if a is the release of a lock by some thread and b is the subsequent acquisition of that lock by any thread then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. We call the partially ordered set of events, ordered using the $\rightarrow$ relation, a *computation*. Note that one partial order can encode exponentially many total orders. We now check all the *possible* — and not just the observed — global states of this computation, and if any of them violates a safety constraint, we can infer that the program is not correct.

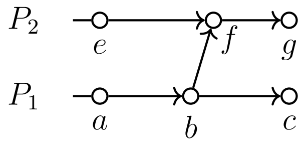


Figure 1.1: A computation on two processes with six events

Let us illustrate this with an example. Consider the computation shown in Figure 1.1 in which two processes P_1 and P_2 execute three events each. Suppose this computation was obtained by executing a distributed computation in which the two processes communicate by message passing. The arrows denote the happened-before relation. P_1 executes three events, and given that they are executed on the same process, we observe their order as: $a \to b \to c$. P_2 also executes three events in the order: $e \to f \to g$. The event f is receipt of a message (on process P_2) that was sent at event b (on process P_1). Let $<_t$ denote the *observed before in real-time* relation between two events, and suppose from the point of view of an outside observer the events were observed in the following order: $a <_t e <_t b <_t f <_t c <_t g$. Consider a hypothetical safety constraint defined as: *the third event on P_1 must happen before the third event on P_2*. In the observed order this constraint is satisfied as, c, the third event on P_1 happens before g the third event on P_2. But observing carefully, we can verify that there exists a possible ordering of the events that is consistent with the computation, and violates this constraint. This order is: $a <_t e <_t b <_t f <_t g <_t c$. Given that there is no happened-before relation between events c and g, it is possible that in a different execution g gets executed before c.

To summarize, the technique of predicate detection involves three main steps: (1) modeling an execution of a parallel as partial order based computation (2) generating global states of the system that are consistent with the partial order, and (3) evaluating if a predicate — encoding the constraint violation or system invariant — becomes true in any or some of these states. Observe that depending on the application, we may be interested in all the states that satisfy a predicate.

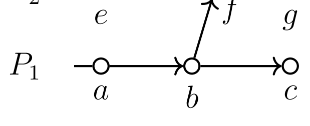
A large body of work uses this approach to verify distributed applications, as well as to detect data-races and other concurrency related bugs in shared memory parallel programs [17, 28, 40, 47]. Finding consistent global states of an execution also has critical applications in snapshotting of modern distributed file systems [1, 63].

We now discuss our contributions to the technique of predicate detection in three directions.

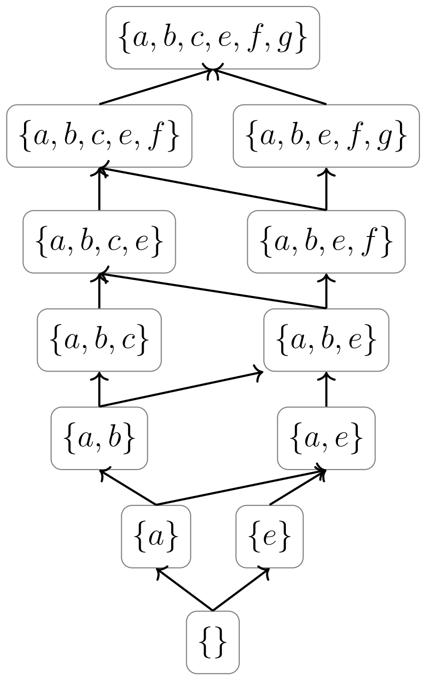
1.2 Space Efficient Breadth First Traversal of Consistent Global States

Given a computation, a consistent global state, or *consistent cut*, of the computation is a possible global state of the system that is consistent with the happened-before partial order. Informally, a consistent cut of a computation is a subset of its events such that all causal dependencies of each event in this subset are satisfied. We present a formal definition in the next chapter. The *empty* global state ($\{\}$) is the one in which no event has been executed, and is trivially consistent. For example, consider the computation in Figure 1.1. This computation has eleven non-empty consistent global states. They are: $\{a\}, \{e\}, \{a, b\}, \{a, e\}, \{a, b, c\}, \{a, b, e\}, \{a, b, c, e\}, \{a, b, e, f\}, \{a, b, c, e, f\}$, $\{a, b, e, f, g\}, \{a, b, c, e, f, g\}$. Note that any subset of events that includes event f but does not include its causal dependencies $\{a, b\}$ is not a consistent cut.

A fundamental requirement for predicate detection is the traversal of all possible consistent cuts of the system. The set of all consistent cuts of a computation can be represented as a directed acyclic graph in which each vertex represents a consistent cut, and the edges mark the transition from one global state to another by executing one event. Moreover, this graph has a special structure: it is a *distributive lattice* [48]. For example, Figure 1.2b shows the distributive lattice of consistent cuts of the computation in Figure 1.2a. Multiple algorithms have been proposed to traverse the lattice of consistent cuts of a parallel execution. Cooper and Marzullo's algorithm[23] starts from the source — a consistent cut in which no operation has been executed by any process — and performs a breadth-first-search (BFS) visiting the lattice level by level. Alagar and Venkatesan's algorithm[2] performs a depth-first-search (DFS) traversal of the lattice, and Ganter's algorithm [31] enumerates global states in lexical order.



(a) Computation



(b) Lattice of Consistent Cuts

The BFS traversal of the lattice is particularly useful in solving two key problems. First, suppose a programmer is debugging a parallel program to find a concurrency related bug. The global state in which this bug occurs is a counter-example to the programmer's understanding of a correct execution, and we want to halt the execution of the program on reaching the first state where the bug occurs. Naturally, finding a small counter example is quite useful in such cases. The second problem is to check all consistent cuts of given rank(s). For example, a programmer may observe that her program crashes only after k events have been executed, or while debugging an implementation of Paxos [45] algorithm, she might only be interested in analyzing the system when all processes have sent their *promises* to the leader. Among the existing traversal algorithms, the BFS algorithm provides a straightforward solution to these two problems. It is guaranteed to traverse the lattice of consistent cuts in a level by level manner where each level corresponds to the total number of events executed in the computation. In contrast, DFS or Lexical order based traversals may have to traverse the complete lattice to find all the cuts in which a specific number of events have been executed, and thus are ill-suited to solve the above problems. The traditional BFS traversal, however, requires space proportional to the size of the biggest level of the lattice which, in general, is *exponential* in the number of processes.

We present a new algorithm to perform BFS traversal of the lattice of consistent cuts in space that is polynomial in the size of the processes [14]. We use a partitioning scheme for partial orders, called *uniflow chain partition*, to design our algorithm to traverse any given level of the lattice of consistent cuts. Our algorithm traverses the cuts of only the given level, and no other level in the lattice. None of the existing traversal algorithms in the literature can do so. In short, our contributions are:

- For a computation on n processes such that each process has m events on average, our algorithm requires $\mathcal{O}(m^2n^2)$ space in the worst case, whereas the traditional BFS algorithm requires $\mathcal{O}(\frac{m^{n-1}}{n})$ space (exponential in n).
- Our evaluation on seven benchmark computations shows the traditional BFS runs out of the maximum allowed 2 GB memory for three of them, whereas our implementation can traverse the lattices by using less than 60 MB memory for each benchmark.

1.3 Detecting Stable and Count Predicates

In many debugging/analysis applications, we are only interested in global states of a system that satisfy a given predicate. For example, while debugging an implementation of Paxos [45] algorithm, a programmer might only be interested in analyzing possible system states when all the *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, 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 parallel programs. For example, $B =$ *exactly two messages have been received* is a count predicate.

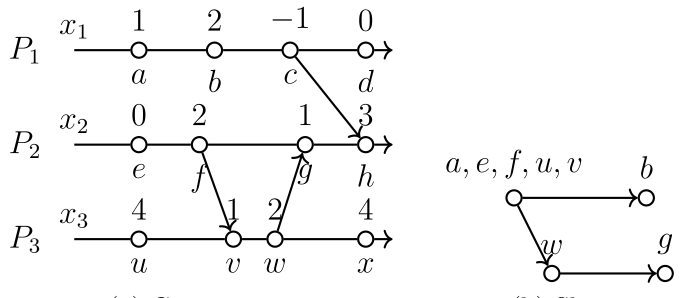
If we are interested in enumerating all the consistent cuts of a trace that satisfy a global predicate B that is of either a stable or a count predicate, then we currently only have one choice: traverse all the cuts using existing traversal algorithms (such as BFS, DFS, and Lex) and check which ones satisfy B. This is 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 1.1, and the predicate $B =$ *at least* 4 *events have been executed*. Figure 1.2b 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 [13] to efficiently enumerate 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 an exponential reduction in time complexities in comparison to existing algorithms.

1.4 Slicing Algorithms

Mathematical abstractions play a crucial role in design and analysis of computational tasks. In the context of predicate detection, we can apply an abstraction on a computation that removes the parts that are not relevant to the predicate under consideration and produces a smaller computation . This abstract computation may be exponentially smaller than the original computation, and thus our analysis becomes significantly faster. *Computation slicing* is an abstraction technique for efficiently finding all global states of a computation that satisfy a given global predicate without explicitly enumerating all such global states [51]. The *slice* of a computation with respect to a predicate is a sub-computation that satisfies the following properties: (a) it contains all global states of the computation for which the predicate evaluates to true, and (b) of all the sub-computations that satisfy condition (a), it has the least number of global states.

As an illustration, consider the computation shown in Fig. 1.2(a). The computation consists of three processes P_1, P_2, and P_3 hosting integer variables x_1, x_2, and x_3, respectively. An event, represented by a circle is labeled with the value of the variable immediately after the event is executed. Suppose we want to determine whether the property (or the predicate) $(x_1 * x_2 + x_3 < 5)$ $\wedge$ $(x_1 \geq 1) \wedge (x_3 \leq 3)$ ever holds in the computation. In other words, does there exist a global state of the computation that satisfies the predicate? The



(a) Computation (b) Slice

Figure 1.2: A Computation, and its slice with respect to predicate $(x_1 \geq 1) \wedge (x_2 \leq 3)$

predicate could represent the violation of an invariant. Without computation slicing, we are forced to examine all global states of the computation, twenty-eight in total, to ascertain whether some global state satisfies the predicate. Alternatively, we can compute a slice of the computation automatically with respect to the predicate $(x_1 \geq 1) \wedge (x_3 \leq 3)$ as shown in Fig. 1.2(b). We can now restrict our search to the global states of the slice, which are only six in number, namely:

$\{a, e, f, u, v\}, \{a, e, f, u, v, b\}, \{a, e, f, u, v, w\}$,

$\{a, e, f, u, v, b, w\}, \{a, e, f, u, v, w, g\}$, and $\{a, e, f, u, v, b, w, g\}$.

The slice has much fewer global states than the computation itself — exponentially smaller in many cases—resulting in substantial savings.

We focus on abstracting distributed computations with respect to *regular* predicates (defined in Sec. 2). The family of *regular* predicates contains many useful predicates that are often used for runtime verification in distributed systems. Some such predicates are:

Conjunctive Predicates: Global predicates which are conjunctions of local predicates. For example, predicates of the form, $B = (l_1 \geq x_1 \geq u_1) \wedge (l_2 \geq x_2 \geq u_2) \wedge \ldots \wedge (l_n \geq x_n \geq u_n)$, where x_i is the local variable on process P_i, and l_i, u_i are constants, are conjunctive predicates. Some useful verification predicates that are in conjunctive form are: detecting mutual exclusion violation in pairwise manner, pairwise data-race detection, detecting if each process has executed some instruction, etc.

Monotonic Channel Predicates [32]: Some examples are: all messages have been delivered (or all channels are empty), at least k messages have been sent/received, there are at most k messages in transit between two processes, the leader has sent all "prepare to commit" messages, etc.

We make two key contributions to the problem of computational slicing:

1.4.1 Distributed Slicing Algorithm for Regular Predicates

Centralized offline [50] and online [61] algorithms for *slicing* based predicate detection have been presented previously. For systems with large number of processes, centralized algorithms require a single process to perform high number of computations, and to store very large data. In comparison, a distributed online algorithm significantly reduces the per process costs for both computation and storage. Additionally, for predicate detection, the centralized online algorithm requires at least one message to the slicer process for every relevant event in the computation, resulting in a bottleneck at the slicer process. A method of devising a distributed algorithm from a centralized algorithm is to decompose the centralized execution steps into multiple steps to be executed by each process independently. However, for performing online abstraction using computation slicing, such an incremental modification is inefficient as direct decomposition of the steps of the centralized online algorithm requires that each process sends its local state information to all the other processes whenever the local state (or state interval) is updated. In addition, a simple decomposition leads to a distributed algorithm that wastes significant computational time as multiple processes may end up visiting (and enumerating) the same global state. Thus, the task of devising an efficient distributed algorithm for *slicing* is non-trivial.

We present the first *distributed online* slicing algorithm for *regular* predicates in distributed systems [15, 54]. Our algorithm exploits not only the nature of the predicates, but also the collective knowledge across processes. The optimized version of our algorithm reduces the required storage per *slicing* process, and computational workload per *slicing* process by $\mathcal{O}(n)$.

1.4.2 Slicing Algorithms for EF and AG Temporal Operators on Non-Regular Predicates

Computation tree logic (CTL) [18] is a temporal logic specification language to describe properties of computation trees. Formulae written in CTL can reason about many possible executions with the notion of time and future in executions. Two key temporal operators in CTL are: EF and AG . For a predicate B, $\mathsf{EF}(\mathrm{B})$ encodes the expression *for some execution path starting* *from the current state, B becomes true*, and $\mathsf{AG}(\mathrm{B})$ encodes the expression *starting from the current state, B is true for all execution paths.* It is due to such expressive power, and ability to capture a wide range of temporal properties that are otherwise difficult or impossible to capture using global state based predicates, CTL operators have become a popular choice for writing specifications in verification tasks.

Previous research [51, 59] has focused on devising slicing algorithms for regular state based predicates, and their CTL based temporal formulae. When a predicate B is regular, the temporal operators $\mathsf{EF}(\mathrm{B})$ and $\mathsf{AG}(\mathrm{B})$ are also regular [60]. In many scenarios, however, the predicate B is not regular, and thus $\mathsf{EF}(\mathrm{B})$ and $\mathsf{AG}(\mathrm{B})$ may not be regular. For this case, when B is not regular, we present two offline algorithms: (1) to efficiently compute the slice with respect to $\mathsf{AG}(\mathrm{B})$ when $\neg B$ (B's negation) can be detected efficiently; and (2) to efficiently compute the slice with respect to $\mathsf{EF}(\mathrm{B})$ when B is efficiently detectable. Both of these algorithms require that the slice of the computation with respect to B is available to us as input.

1.5 Applications of Developed Algorithms to Other Fields

In developing our algorithms, we have focussed primarily on the technique of predicate detection for parallel computations. The applications of our body of work, however, are not limited to just this field. We now discuss how our algorithms apply to the problem of stable marriage [30], and problems in lattice theory.

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 dissertation 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 rank traversal algorithm in Section 3.4.1 can be used to enumerate order ideals of any given level without visiting other levels of the lattice. Our algorithm for enumerating cuts satisfying counting predicate (in Section 4.2) 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.

1.6 Overview of the Dissertation

The rest of this dissertation is organized as follows. In Chapter 2 we discuss the background on concepts used for developing our algorithms, including a special chain partition of posets, called *uniflow chain partition*. Using this chain partition, we present an algorithm to enumerate consistent cuts of a computation in breadth-first manner in Chapter 3, and then evaluate its runtime performance against that of existing enumeration algorithms on five benchmark computations. In chapter Chapter 4, we present algorithms that use uniflow chain partitions to enumerate consistent cuts that satisfy detect stable and counting predicates. In Chapter 5, we present a distributed online algorithm to perform slicing with respect to regular predicates, and in Chapter 6 discuss slicing with respect to temporal logic operators when the underlying predicate is not regular and we have already obtained a slice of the computation with respect the predicate. We then present concluding remarks and future work in Chapter 7.

Chapter 2

Background

In this chapter, we present the concepts and notation used in the rest of this dissertation.

We use the term *computation* to denote an execution trace of a parallel program. Unless specified, we restrict our focus to finite computations. Thus, a computation is a collection of events executed by processes in the system. An event may denote — depending on the context of the problem — the execution of a single instruction or a collection of instructions together. It is possible that the instructions executed by different processes/threads are different. Our model of parallel computation is based on the happened-before relation, and is applicable to both distributed as well as shared memory parallel computations. A shared memory parallel computation, often called a *concurrent computation*, involves multiple processes/threads controlled by a scheduler on a single machine. In shared memory computations, we use the term process for an operating system process, and also a thread. In shared memory computations processes execute their program instructions independently and use mechanisms such as mutexes or semaphores for synchronization. A distributed computation is a parallel computation without shared memory in which interprocess communication is possible only through message-passing.

We order the events of a computation using Lamport's *happened-before* ($\rightarrow$) relation [44]. The relation $\rightarrow$ on the set of events of a computation is the smallest relation that satisfies the following three conditions:

1. **Process Order**: If a and b are events executed by the same process, and a is executed before b, then $a \rightarrow b$.

2. **Causal Order**: Causal order between events on different processes is imposed by either of the following:

    - *Message Dependency in Distributed Computations*: If a is a message send event, and b is an event corresponding to the receipt of the same message, then $a \rightarrow b$.

    - *Synchronization Dependency in Shared Memory Computations*: In a shared memory computation, if a is the release of a lock by some process and b is the subsequent acquisition of that lock by any process then $a \rightarrow b$. Or, if a corresponds to a fork call by a process and b is the first event in the execution of the child process, then $a \rightarrow b$. Similarly, if a is a termination of a child process and b is the actual join event with its parent process then $a \rightarrow b$. Or, if a process goes into wait on some monitor at event a and is notified on the same monitor at event b, then $a \rightarrow b$.

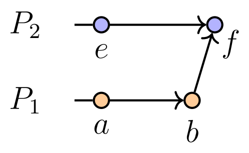
3. **Transitivity**: If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

The happened-before relation imposes a partial order the set of events in a computation.

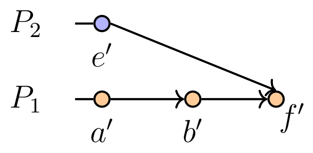
2.1 Computation as Partially Ordered Set of Events

Let X be a set, and R be a binary relation on X. If R is irreflexive, antisymmetric, and transitive then it imposes a partial order on the elements of X. $\langle X, R\rangle$, the pair of set X along with relation R, is called a partially ordered set, or *poset* in short.

If E is the set of events of a parallel computation, then the happened-before relation, $\rightarrow$, is an irreflexive, antisymmetric, and transitive binary relation on E. Thus a computation under $\rightarrow$ relation forms a poset. We use the notation $P = (E, \rightarrow)$ to denote this poset. It is important to note that multiple computations could have the identical posets as their model. For example, the two computations shown in Figure 2.1 will lead to the identical posets.



(a) Computation on two processes



(b) Computation on three processes

Figure 2.1: Illustration: Two different computations that will lead to identical posets as their models

Let $P = (E, \rightarrow)$ be a computation on n processes $\{P_1, P_2, \ldots, P_n\}$. Then we use E_i to denote the set of events executed by process P_i where $1 \leq i \leq n$. Note that E_i is a totally ordered set. Consider two events $a, b \in E$.

If either $a \rightarrow b$ or $b \rightarrow a$, we say that a and b are *comparable*; otherwise, we say a and b are *concurrent*, and denote this relation by $a \mid\mid b$. Observe that $\mid\mid$ is not a transitive relation.

Let $proc(e)$ denote the process on which event e occurs. The predecessor and successor events of e on $proc(e)$ are denoted by $pred(e)$ and $succ(e)$, respectively, if they exist.

2.1.1 Chains and Antichains

Let $P = (E, \rightarrow)$ be a computation whose set of events is E. A subset $Y \subseteq E$ is called a *chain*, if every pair of distinct events from Y is comparable in P, that is: $\forall a, b \in Y : (a \rightarrow b) \vee (b \rightarrow a)$. Similarly, a subset $W \subseteq E$ is called an *antichain*, if every pair of distinct events from W is concurrent in P, that is: $\forall a, b \in W : a||b$. Thus, E_i, the set of events executed by process P_i, is a chain. The *height* of a poset is defined to be the size of a largest chain in the poset. The *width* of a poset is defined to be the size of a largest antichain in the poset. Consider the computation shown in Figure 2.2. Chains $\{a, b, c\}$ and $\{e, f, g\}$ are two chains of three events each that are formed by the events executed by processes P_1 and P_2 respectively. Note that $\{a, b, f, g\}$ is also a chain; moreover it is a largest chain, and thus the height of the poset is four. For this computation, $\{a, e\}$ is an antichain, and so is $\{e, c\}$. Note that $\{a, g\}$ is not an antichain. The width of this computation/poset is two.

Generally, a computation n processes $\{P_1, P_2, \ldots, P_n\}$ is partitioned into n chains such that the events executed by process P_i $(1 \leq i \leq n)$ are

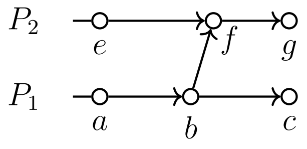


Figure 2.2: A computation on two processes

placed on i^{th} chain. This leads to the notion of chain partitions.

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*

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

For a computation $P = (E, \rightarrow)$ on n processes, we can identify each event e with a tuple (i, k) which represents the kth event on the ith process, where $1 \leq i \leq n$. Similarly, if we use a different chain partition for P whose width is w, then every event e in the computation can be identified with a tuple (i, k) which represents the kth event on the ith chain; $1 \leq i \leq w$.

2.2 Vector Clocks

Mattern [48] and Fidge [26] 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, we maintain an event e's vector clock, denoted by $e.V$, as a n-length vector of non-negative integers. Note that vector clocks are dependent on chain partition of the poset that models the computation. If a chain partition of a computation has width w, then each vector clock is an array of length w. If f is the kth event executed by process P_i, then we set $f.V[i] = k$. For $j \neq i$, $f.V[j]$ is the number of events that must have happened on process j before f is executed. Thus, $f.V[j]$ is the index of event e on P_j that is the maximal event such that $e \rightarrow f$.

We use the following representation for interpreting chain partitions and vector clocks: a vector clock on n chains is represented as a n-length vector: $[c_n, c_{n-1}, ..., c_i, ..., c_2, c_1]$ such that c_i denotes the number of events executed on process P_i.

Figure 2.3 shows a sample computation with six events and their corresponding vector clocks. Event b is the second event on process P_1, and 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 g is $[3, 2]$.

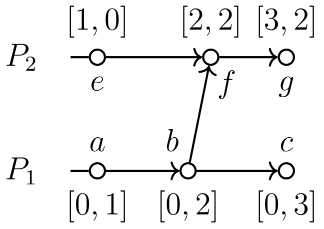


Figure 2.3: A computation with vector clocks of events

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 (denoted by $e \mid\mid f$) iff $e \not\rightarrow f \wedge f \not\rightarrow e$.

2.3 Consistent Cuts

A consistent global state, or *consistent cut*, of a computation is its snapshot view in which all causal dependencies are satisfied. Formally:

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 *possible* global state of the system at some point during its execution [9]. Consider the computation shown in Figure 2.3. 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.

Frontiers: The *frontier* of a consistent cut G, denoted by *frontier*(G), is defined as the set of those events in G whose successors are not in G. Formally,

$$\mathit{frontier}(G) \quad \triangleq \quad \{\, e \in G \mid \implies \mathit{succ}(e) \notin G \,\} \tag{2.1}$$

For example, in Figure 2.3 for the consistent cut $G = \{a, b, e\}$ we have *frontier*$(G) = \{b, e\}$. Similarly, for $G = \{a, b, c, e, f\}$, we have *frontier*$(G) = \{c, f\}$.

2.3.1 Vector Clock Notation of Consistent Cuts

We earlier described how vector clocks can be used to time-stamp events in the computation. We also use them to represent consistent 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 2.3 when P_1 has executed events a and b, and P_2 has only executed event e. The consistent cut for the 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.

2.3.2 Lattice of Consistent Cuts

The set of all consistent cuts of a computation also forms a poset (partially ordered set) under the containment order. Given two consistent cuts G and H of a computation $P = (E, \rightarrow)$, we say that $G \leq H$ iff $G \subseteq H$. For example, in Figure 2.3 the consistent cut for this state, $G = \{a, b, e\}$ contains a smaller consistent cut $H = \{a, b\}$, thus we have $H \leq G$. The consistent cut $H' = \{a, e\}$ similarly is contained in G, however, it is not comparable to H. Thus, we state that $H \leq G$, $H' \leq G$, but $H \nleq H' \wedge H' \nleq H$.

Let us now discuss meet and join operators on elements of a poset. Let $P = \langle X, \leq \rangle$ be a poset.

Definition 3 (Join)**.** *For any two elements $x, y \in X$, j is the join of x and y iff:*

1. $x \leq j \wedge y \leq j$,
2. $\forall j' \in X, (x \leq j' \wedge y \leq j') \implies j \leq j'$.

We denote the join with $\sqcup$ symbol, and write $x \sqcup y = j$.

Thus, the join — if it exists — of any two elements in a poset is their least upper bound.

The meet operator is the dual operator of join, and corresponds — if it exists — to the greatest lower bound.

Definition 4 (Meet)**.** *For any two elements $x, y \in X$, m is the meet of x and y iff:*

1. $m \leq x \wedge m \leq y$,
2. $\forall m' \in X, (m' \leq x \wedge m' \leq y) \implies m' \leq m$.

We denote the meet with $\sqcap$ symbol, and write $x \sqcap y = m$.

A *lattice* is poset that is closed under both join and meet operators, that is: if joins and meets exist for all finite subsets of X.

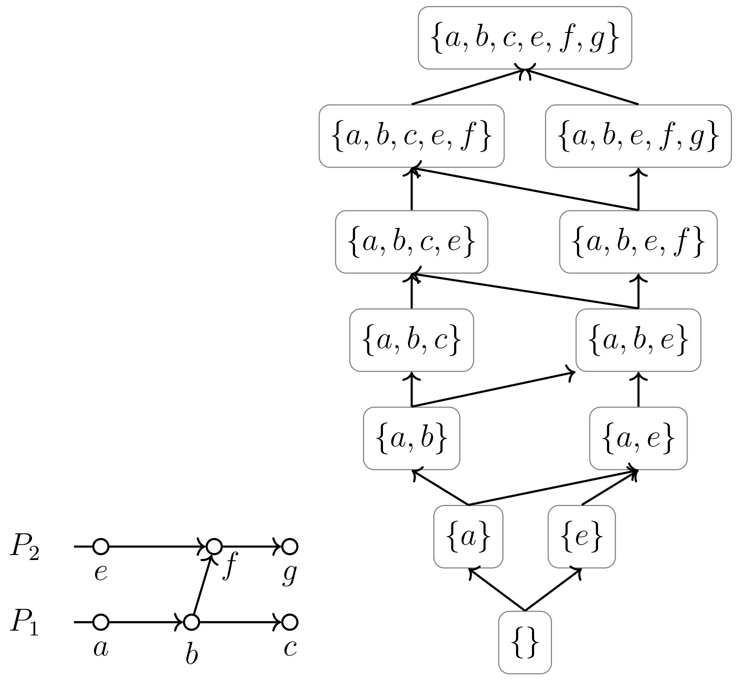
Definition 5 (Lattice)**.** *A poset $P = \langle X, \leq \rangle$ is a lattice iff $\forall x, y \in X$, we have $x \sqcup y \in X$ and $x \sqcap y \in X$.*

Moreover, if the join and meet operators distribute over each other then the lattice is called a *distributive lattice*.

It has been shown [24, 48] that:

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

In Figure 2.4b, we show a computation and its lattice of consistent cuts in their set notation. We show the same lattice in the equivalent vector clock notation of consistent cuts in Figure 2.5.



(a) Computation (b) Lattice of Consistent Cuts

Figure 2.4: A computation and its lattice of consistent cuts

$$\begin{tikzcd}
 & {[3,3]} & \\
{[2,3]} \arrow[ur] & & {[3,2]} \arrow[ul] \\
{[1,3]} \arrow[u] & & {[2,2]} \arrow[ull] \arrow[u] \\
{[0,3]} \arrow[u] & & {[1,2]} \arrow[ull] \arrow[u] \\
{[0,2]} \arrow[u] \arrow[urr] & & {[1,1]} \arrow[u] \\
{[0,1]} \arrow[u] \arrow[urr] & & {[1,0]} \arrow[u] \\
 & {[0,0]} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 2.5: Lattice of Consistent Cuts for Figure 2.4a in Vector Clock notation

We now define the notion of rank of a cut.

Definition 6 (Rank of a Cut)**.** *Given a cut G, we define its* rank*, written $rank(G)$, to be the total number of events, across all processes, that have been executed to reach the cut.*

Recall that given a consistent cut G, we use $G[i]$ to denote the number of events included from process/chain P_i in G's vector clock notation. Based on this, we have $rank(G) = \sum G[i]$.

Consider the lattice of consistent cuts in Figure 2.5. There is one cut ($[0, 0]$) with rank 0, then there are two cuts each of ranks 1 to 5, and finally there is one cut ($[3, 3]$) with rank 6.

2.4 Uniflow Chain Partition

We now discuss a special chain partition of called *uniflow chain partition*. In later chapters of this dissertation, we will use this partition to construct our algorithms for predicate detection and breadth-first traversal of lattice of consistent cuts.

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

$$\forall x, y \in P : \mu(x) < \mu(y) \Rightarrow \neg(y \not\rightarrow x) \tag{2.2}$$

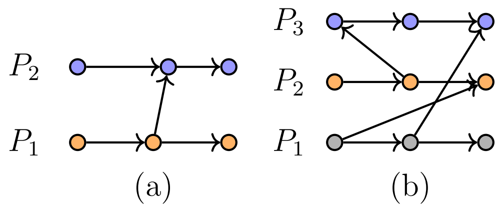


Figure 2.6: Posets in Uniflow Partitions

Visually, in a uniflow chain partition all the edges between separate chains always point upwards. Figure 2.6 shows two posets with uniflow partitions. Whereas Figure 2.7 shows two posets with partitions that do not satisfy the uniflow property. The poset in Figure 2.7a can be transformed into a uniflow partition of three chains as shown in Figure 2.7b. Similarly, Figure 2.7c

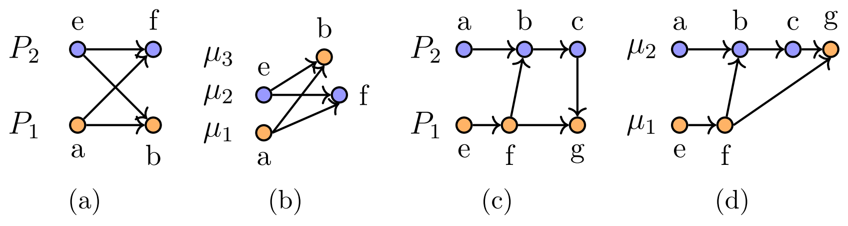


Figure 2.7: Posets in (a) and (c) are not in uniflow partition: but (b) and (d) respectively are their equivalent uniflow partitions

can be transformed into a uniflow partition of two chains shown in Figure 2.7d. 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 2.2 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 7 (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, $e <_u f \equiv (\mu(e) < \mu(f)) \vee (\mu(e) = \mu(f) \wedge pos(e) < pos(f))$*

For example, in Figure 2.7b 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 $pos(f) = 2$.

2.5 Uniflow Chain Partitioning: Online Algorithm

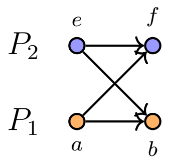
The problem of finding a uniflow chain partition is a direct extension of finding the *jump number* of a poset [16, 6, 66]. Multiple algorithms have been proposed to find the jump number of a poset; which in turn arrange the poset in a uniflow chain partition. Finding an optimal (smallest number of chains) uniflow chain partition of a poset is a hard problem [16, 6]. Bianco et al. [6] 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 present an online algorithm to find a uniflow partition for a computation.

Our algorithm processes events of the computation $P = (E, \rightarrow)$ in an online manner: when a process P_i executes event e it sends the event information to our partitioning algorithm. We require that the event information contains its vector clock in the computation. Recall from Section 2.2 that the vector clocks are dependent on the chain partition of the poset. Let us assume that the computation involves n processes, thus each event's vector clock in the original partition is a vector of length n — the original computation is partitioned n chains where chain P_i contains the executed by ith process. We can regenerate vector clocks for a uniflow chain partition of the computation using the vector clock generation algorithms given in [48, 11].

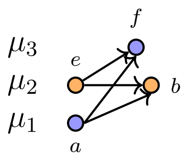
Algorithm 1 FINDUNIFLOWCHAIN(e)

```
Input: An event $e$ of the computation $P = (E, \rightarrow)$ on $n$ processes
Output: $e$ is placed at the end of a chain in the uniflow chain partition $\mu$
1: $maxid$: id of highest uniflow chain till now
2: $eventChainMap$: hashtable of events against their uniflow chain number
3: $uid = e.procid$  // start with chain that executed $e$
4: for each direct causal dependency $dep$ of $e$ do
5:     $uid = \text{MAX}(eventChainMap[dep], uid)$ // max of $uid$ and the chain of direct causal dependency
6: //now check if there exists any chain with the same id
7: if $\exists$ a chain in $\mu$ with $id = uid$ then
8:     $f$ = last event on this chain
9:     if $e \parallel f$ then  // $e$ is concurrent with $f$, cannot add to this chain
10:        $uid = ++maxid$  // increment max used chain id
11:        create new chain with id $= uid$
12: else // chain with required number doesn't exist
13:     create new chain with id $= uid$
14:     $maxid = uid$  // updated max assigned chain id
15: add $e$ at the end of chain with $id = uid$
16: $eventChainMap[e] = uid$ // store mapping of event to chain
```

Algorithm 1 shows the steps of finding an appropriate chain for e in the uniflow partition, and appending e to the end of that chain. Note that in the online setting, e's causal dependencies are guaranteed to be processed to the algorithm before e is processed. Given an event e, we start by setting its uniflow chain, *uid*, to the *e.procid* that is the id of process (chain) in the original computation on which e was executed. Then, we go over all its direct causal dependencies, and in case any of the dependencies were placed on higher numbered chains, we update the *uid* (lines 4–5). We know that to maintain the uniflow chain partitioning, e must be placed either on a chain with id *uid*, or above it. Lines 7–9 check if there already exists a chain with that id, and if the last event on this chain is concurrent with e. If so, we cannot place e on this chain and must put it on a chain above — possibly by creating a new chain (lines 10–11). If e is not concurrent with f, then we can place it on the existig chain numbered *uid*. If no chain has been created with *uid* as its number, that means e is the first event on some chain (process) in P and we must create a new chain in our uniflow partition for it. This is done in lines 12–14. Finally, we place e on the correct chain at line 15, and then store the mapping of this event against the chain number on which it was placed.



(a) Original Computation



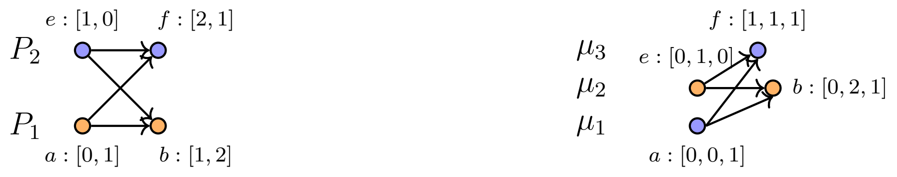
(b) Uniflow Partition with Algorithm 1

Figure 2.8: Illustration: Finding uniflow chain partition of a computation

Let us illustrate the execution of the algorithm on the poset of Figure 2.8a. Initially, our uniflow chain partition μ has no chains. Suppose, a, the first event on process P_1 is first event sent to this algorithm. As there is no event in μ, a will be placed on chain id 1. In an online setting, e the first event on P_2 is going to be presented next. This event also has no direct causal dependencies, and thus the *uid* value for it at line 7 will be 2 — the id of the process that executed it. However, there is no chain with id 2 yet, and thus we execute lines 12–14 to create a new chain and place e on chain 2 in μ. Suppose the next event to arrive is b the second event on P_2. As b is causally dependent on a and e both its *uid* value after the loop of line 4–5 is 2 as we take the maximum of the uids assigned to all the causal dependencies. There is a chain with id 2 in μ, and its only event is e which not concurrent with b. Hence, we skip to line 15, and place b at the end of chain 2. The last event to arrive will be f. After executing lines 4–5, the *uid* value for f will be 2. As there is a chain with id 2 in μ, at line 9 we will compare f with b — the last event on chain 2. However, b and f are concurrent. Hence, we have to create a new chain with id 3 as per lines 10–11. We then place f on this chain. The resulting uniflow partition μ is shown in Figure 2.8b. With this uniflow chain partition, we regenerate the vector clocks of the events as per [48, 11]. The vector clocks in the original computation and in the new uniflow chain partition are shown in Figure 2.9.

2.5.1 Proof of Correctness

Lemma 2. *Given a computation $P = (E, \rightarrow)$ and events $e, f \in E$ such that $e \rightarrow f$, If Algorithm 1 places e on chain k, and f on chain k' in P's uniflow*



(a) Original Computation (b) Uniflow Chain Partition

Figure 2.9: Illustration: Regenerated vector clocks for uniflow chain partition

chain partition μ, then $k \leq k'$.

Proof. Suppose not, and $k > k'$. Since $e \rightarrow e'$ and our algorithm is online, we know that e must be processed before f. By lines 4–5, we are guaranteed that *uid* for f in μ will be greater than or equal to k as e is a causal dependency of f. Thus we have $k \leq k'$ when we reach line 7. All subsequent paths of execution in lines 7–14 either keep the value of *uid* same, or increment it by at least one. Thus, when we reach line 15 for placing f in μ we are guaranteed that *uid* value for f is greater than or equal to k. At lines 15 and 16, we place f at the end of chain numbered *uid*, and then store the mapping of f against this number. Thus, we maintain $k \leq k'$. □

2.5.2 Complexity Analysis

For a computation on n processes, there can be at most n events that are direct causal dependencies of any event. Hence, lines 4–5 of Algorithm 1 take $\mathcal{O}(n)$ time for any event. Checking for existence of a chain id at line 6 is a constant time operation as we can use a hash-table for storing the chains against their ids. The check for concurrency of two events is $\mathcal{O}(n)$ as we can use the original vector clocks of the two events. Lines 9–11 then take constant time. If the events e and f are not concurrent at line 9, we skip to line 15. We take constant time in appending the event at the end of a chain and storing its mapping against the chain number at line 16. Hence, in the worst case our algorithm takes $\mathcal{O}(n) + \mathcal{O}(n) \equiv \mathcal{O}(n)$ time per event.

We require $\mathcal{O}(|E|)$ space for the hash-table that stores the mapping of each event and its uniflow chain number.

2.6 Consistent Cuts in Uniflow Chain Partitions

The structure of uniflow chain partitions can be used for efficiently obtaining bigger consistent cuts. From now on we use the vector clock notation of consistent cuts for our discussion. Recall that in vector clock notation $G[i]$ denotes the number of events included from chain i. 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 3 (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:*

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

Proof. Using Equation 2.2, 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 2.6b, consider the cut $G = [1, 2, 1]$[1] that is a consistent cut of the poset. Then, picking $k = 1$, and using Lemma 3 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 2.7c, $G = [2, 2]$ is a consistent cut. The chain partition, however, is not uniflow and thus applying Lemma 3 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 8 (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 2.9b, $\{a, e, f\}$

[1]Recall (from Section 2.2) 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 a chain partition.

(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.

2.7 Global Predicates

A *global predicate* (or simply a *predicate*), in our model, is either a *state-based* predicate or a *path-based* predicate. State-based predicates are boolean-valued function on variables of processes. Given a consistent cut, a state-based predicate is evaluated on the state resulting after executing all events in the cut. A global state-based predicate is *local* if it depends on variables of a single process. If a predicate B (state or path based) evaluates to true for a consistent cut C, we say that "C satisfies B" and denote it by $C \models B$.

Consider the computation in Figure 2.10. It shows a distributed computation on three processes in which processes send messages to each other. For example, P_1 sends a message at event c, this message is received at event h on to P_2. Processes P_1, P_2, and P_3 have local integer variables x_1, x_2, and x_3, respectively. The value of these local variables, after execution of each event is shown immediately above the event. Assume that all variables are initialized to 0. The consistent cut $U = \{a, e, f, u\}$ with $\mathit{frontier}(U) = [a, f, u]$ satisfies the predicate $x_1 + x_2 \leq x_3$. However, the consistent cut $V = \{a, e, f, u, v\}$ with $\mathit{frontier}(V) = [a, f, v]$ does not.

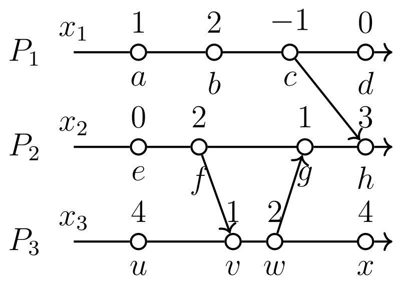


Figure 2.10: Illustration: Computation with local variables

2.7.1 Stable, Linear, and Regular Predicates

The problem of predicate detection requires us to check if a given predicate could be satisfied by any consistent cut of a computation. This problem is intractable in general [49, 50]. To obtain a polynomial-time detection algorithm, it becomes necessary to exploit some structural properties of the predicate. The *stability* of a predicate is one such property. 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. We discuss stable predicates in detail in Section 4.1.

Another property that allows us to detect predicates efficiently is the *linearity property*:

Definition 9 (Linearity Property of Predicates)**.** *A predicate B is said to have the linearity property, if for any consistent cut C that does not satisfy predicate B, there exists a process P_i such that we must advance along P_i to reach a consistent cut that is reachable from C and satisfies B.*

Predicates that have the linearity property are called *linear predicates*. The process P_i in the above definition is called a *forbidden process*.

Consider the computation shown in Figure 2.10. The cut denoted by frontier $[b, f, u]$ does not satisfy the linear predicate "all channels are empty", as b sends a message and is only received at v, hence the channel between P_2 and P_3 is not empty. Thus, progress must be made on P_3 to reach the cut with frontier $[b, f, v]$ which satisfies the predicate. Here P_3 is the forbidden process.

Detecting a linear predicate efficiently requires an additional property called efficient advancement property. A linear predicate has the efficient advancement property if given a cut that does not satisfy this predicate, we can find a forbidden process efficiently. For a computation involving n processes, given a consistent cut that does not satisfy the predicate, the forbidden process P_i can be found in $\mathcal{O}(n)$ time for most linear predicates used in practice. To find a *forbidden process* given a consistent cut, a process first checks if the cut needs to be advanced on itself; if not it checks the states in the total order defined using process identifiers, and picks the first process whose state makes the predicate false on the cut.

An important subclass of linear predicates is the class of *regular predicates*. They exhibit a stronger structural property:

Definition 10 (Regular Predicates)**.** *A predicate is called* regular *if for any two consistent cuts C and D that satisfy the predicate, the consistent cuts given by $C \sqcap D$ (meet) and $C \sqcup D$ (join) also satisfy the predicate.*

Examples of regular predicates include local predicates (*e.g.*, $x \leq 4$), conjunction of local predicates (*e.g.*, $(x \leq 4) \wedge (y \geq 2)$ where x and y are variables on different processes) and monotonic channel predicates (*e.g.*, there are at most k messages in transit from P_i to P_j) [50].

2.7.2 Temporal Logic Predicates

A path-based or temporal logic predicate is one that includes temporal operators [18] such as AG , EG and EF . For a consistent cut C, the temporal operators are defined as follows:

- $C \models \mathsf{AG}(B)$ iff for *all* consistent cut sequences $C_0, \ldots, C_k$ such that (i) $C_0 = C$, and (ii) $C_i \leq C_{i+1}$ (iii) $C_k = E$, we have: $C_i \models B$ for *all* $0 \leq i \leq k$. Thus, $\mathsf{AG}(B)$ means that in the lattice of consistent cuts, all cuts reachable from cut C satisfy B.

- $C \models \mathsf{EG}(B)$ iff for *some* consistent cut sequence $C_0, \ldots, C_k$ such that (i) $C_0 = C$, and (ii) $C_i \leq C_{i+1}$ (iii) $C_k = E$, we have: $C_i \models B$ for *all* $0 \leq i \leq k$. Thus, $\mathsf{EG}(B)$ means that in the lattice of consistent cuts, there exists a path starting with cut C till the biggest consistent cut E on which each consistent cut satisfies B.

- $C \models \mathsf{EF}(B)$ iff for *some* consistent cut sequence $C_0, \ldots, C_k$ such that (i) $C_0 = C$, and (ii) $C_i \leq C_{i+1}$ (iii) $C_k = E$, we have: $C_i \models B$ for *some* $0 \leq i \leq k$. Thus, $\mathsf{EF}(B)$ means that in the lattice of consistent cuts, there exists a consistent cut that satisfies B, and we can reach this cut by starting with the cut C and then executing *some* sequence of events on the way.

Consider a system of two processes P_1 and P_2 trying to execute a critical section in a mutually exclusive manner. Let B_1 and B_2 be the predicates that P_1 and P_2 are, respectively, in their critical section. A safe state, from which the system will never violate mutual exclusion, can be determined by detecting the predicate $\mathsf{EF}\ (B_1 \wedge B_2)$. If the predicate evaluates to false at the current state, then there is no future state where both P_1 and P_2 are in the critical section simultaneously, indicating a safe state. Otherwise, the current state is unsafe.

It was shown in [60] that, when predicate B is regular, the three temporal logic predicates $\mathsf{AG}(B)$, $\mathsf{EG}(B)$ and $\mathsf{EF}(B)$ are also regular predicates.

2.8 Computation Slicing

Computation slicing, is an abstraction technique for efficiently finding all global states of a computation that satisfy a given global predicate without explicitly enumerating all such global states [51]. The result is a computation slice, often just called *slice*: a concise representation of all the consistent cuts of a computation that satisfy a predicate. The slice of a computation with respect to a predicate is a sub-computation that satisfies the following properties: (a) it contains all global states of the computation for which the predicate evaluates to true, and (b) of all the sub-computations that satisfy condition (a), it has the least number of global states.

We alter our model of computation slightly for computation slicing. We have, till now, modeled a computation as a poset of events using the happened-before relation. For slicing, we use directed graphs to model computations as well as their slices. This allows us to handle both of them in a *uniform* and convenient manner. The set of vertices in our equivalent directed graph includes the set of events, while the edges are derived from the traditional model. In addition, we allow strongly connected components in our model, which are not possible in the traditional model. To obtain the directed graph from a computation, we perform the following steps.

We assume the presence of *fictitious* initial and final events on each process. The initial event on process P_i, denoted by $\bot_i$, occurs before any other event on P_i and initializes the state of that process. Likewise, the final event on process P_i, denoted by $\top_i$, occurs after all other events on P_i. We use final events only to ease the exposition of the slicing algorithms. It *does not imply* that processes have to synchronize with each other at the end of the computation. For convenience, let $\bot$ and $\top$ denote the set of all initial events and final events, respectively. We assume that all initial events belong to the same strongly connected component. Likewise, all final events also belong the same strongly connected component.

After this, we model a computation as a directed graph represented by the tuple $\langle E, \mapsto \rangle$, where E now is the set of events including fictitious events, and edges are given by the precedence relation $\mapsto$. The precedence relation on the set of non-fictitious events is defined by the *happened-before* relation $\rightarrow$. Note that, for two non-fictitious events e and f, $e \mapsto f$ if and only if $e \rightarrow f$. The $\perp$ events precede all other events in the computation. All initial events precede all non-fictitious events and all non-fictitious events precede all final events. Figure 2.11 shows the resulting directed graph representation after performing these steps on the computation shown in Figure 2.10.

Any consistent cut of a distributed computation that contains all initial events ($\perp$) and none of the final events ($\top$) is referred to as a *non-trivial consistent cut*. Only non-trivial consistent cuts are of interest to us they correspond to *real* system states. We denote the largest non-trivial consistent cut of the computation, which is given by $E \setminus \top$, by $\widehat{E}$.

As mentioned earlier, we allow non-singleton strongly connected components in our model. In a computation, however, they consist entirely of fictitious events. We use directed graphs to model the computation slices too. A strongly connected component in a computation slice can contain non-fictitious events. A strongly connected component in the slice of a computation that contains two non-fictitious events e and f implies that both events must be present in a consistent cut of the computation for that cut to satisfy the predicate. We define a *non-trivial strongly connected component* as a strongly connected component that contains (a) at least two non-fictitious events, and (b) none of the final events.

For a computation $\langle E, \mapsto\rangle$ and a predicate B, we use $\mathcal{C}_B(E)$ to denote the consistent cuts of that satisfy B. Note that $\mathcal{C}_B(E)$ is a subset of $\mathcal{C}(E)$ —

the set of all consistent cuts of the computation. Let $\mathcal{I}_B(E)$ denote the set of all graphs on vertices E such that for every graph $G \in \mathcal{I}_B(E)$, $\mathcal{C}_B(E) \subseteq \mathcal{C}(G) \subseteq \mathcal{C}(E)$.

Definition 11 (Slice [50])**.** *A slice of a computation with respect to a predicate B is a directed graph that contains the fewest consistent cuts, such that every consistent cut of the computation that satisfies B is contained in it. Formally, given a computation $\langle E, \mapsto\rangle$ and a predicate B,*

$$\begin{aligned} &S \textit{ is a slice of } \langle E, \mapsto\rangle \textit{ for } B \quad \triangleq \\ &S \in \mathcal{I}_B(E) \wedge \forall G : G \in \mathcal{I}_B(E) : |\mathcal{C}(S)| \leq |\mathcal{C}(G)| \end{aligned} \tag{2.3}$$

We denote the slice of computation $\langle E, \mapsto\rangle$ with respect to predicate B by $\langle E, \mapsto\rangle_B$. A slice is *empty* if it does not contain any non-trivial consistent cuts. In general, there can be multiple directed graphs on the same set of consistent cuts [50]. As a result, more than one graph may constitute a valid representation of a given slice. Using lattice theory, it was shown in [50] that all such graphs have the same transitive closure of edges, and thus the same set of consistent cuts. The slice of a computation with respect to a predicate contains two types of edges: (a) those that were present in the original computation, and (b) those added to the computation to eliminate consistent cuts that do not satisfy the predicate.

Consider the computation shown in Figure 2.11. Its slice with respect to the predicate $(x_1 \geq 1) \wedge (x_3 \leq 3)$ is shown in Figure 2.12. The edges added to the computation to eliminate irrelevant consistent cuts are shown as dotted

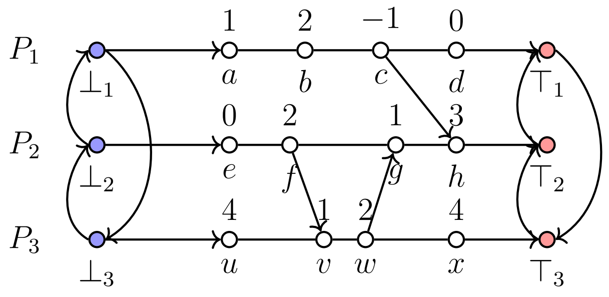


Figure 2.11: Computation of Figure 2.10 as a directed graph under the slicing model

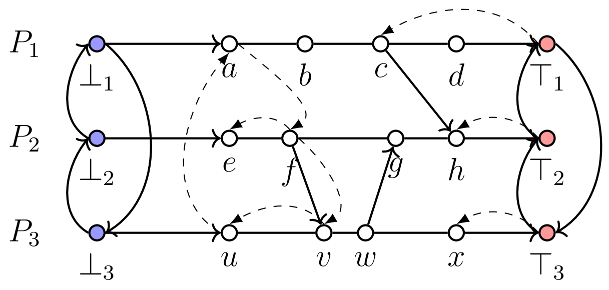


Figure 2.12: Slice of Figure 2.11 as a directed graph with respect to $B = (x_1 \geq 1) \wedge (x_3 \leq 3)$

edges. For example, by adding a dotted edge from v to u, any consistent cut that contains u but not v is eliminated.

To generate the slice of a computation $\langle E, \mapsto \rangle$ with respect to a regular (state-based) predicate B, we compute consistent cut $J_B(e)$ for every event e, which is defined as the smallest non-trivial consistent cut of the computation that contains e and satisfies B [50]. If no such cut of the computation exists, then $J_B(e)$ is set to the default cut. Recall that, in the slice with respect to B, we are only interested in those consistent cuts of the computation that satisfy B. Hence, every consistent cut of $\langle E, \mapsto \rangle_B$ that contains e, will include all events in $J_B(e)$. By adding an edge from all events in $\mathit{frontier}(J_B(e))$ to e, those cuts of the computation that contain e but not other events in $J_B(e)$, are eliminated. These are the consistent cuts that do not satisfy B. *Note that e does not have to be the maximal event in $J_B(e)$.* Intuitively, the set of consistent cuts given by $J_B(e)$ for all events e form the *join-irreducible elements* (or basis elements) of the lattice of consistent cuts generated by the slice [50]. The slice $\langle E, \mapsto \rangle_B$ contains the set of events E as the set of vertices and has the following edges [50]: $\forall e : e \notin \top$, there is an edge from e to $\mathit{succ}(e)$, and for each event e, there is an edge from every event $f \in \mathit{frontier}(J_B(e))$ to e.

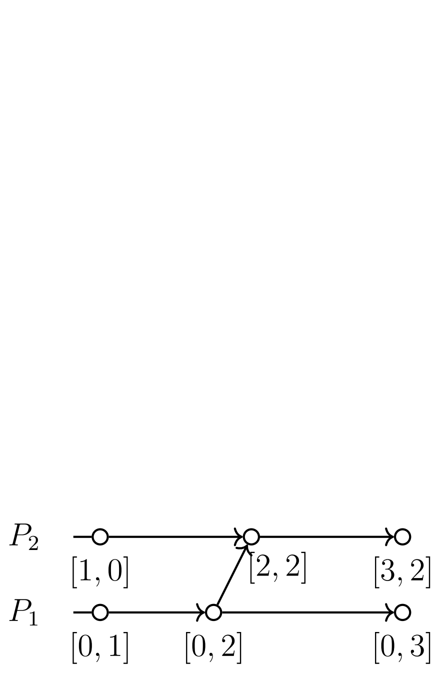
We have now completed the overview of all the required background concepts. In the next chapter, we use the uniflow chain partition of a computation to perform breadth-first traversal of its lattice of consistent cuts.

Chapter 3

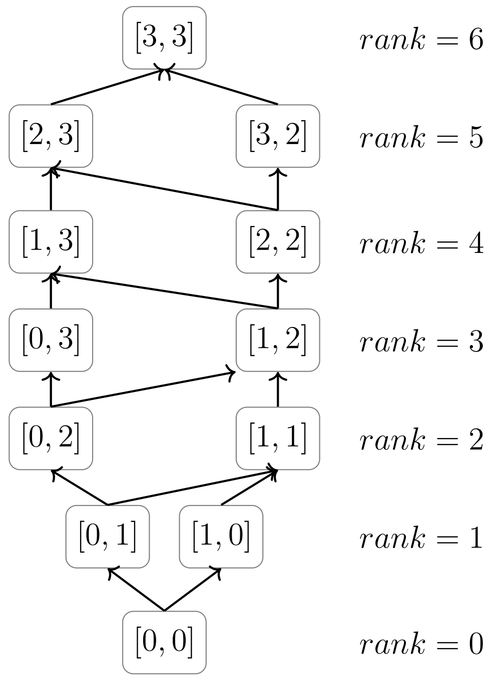
Polynomial Space Breadth-First Traversal of Consistent Cuts

In this chapter, we present algorithms for breadth-first traversal of the lattice of consistent cuts of a computation using space that is polynomial in the size of computation.

Given a computation $P = (E, \rightarrow)$, the set of its consistent cuts, $\mathcal{C}(E)$, forms a distributive lattice [24, 48]. In many scenarios, analysis of a parallel computation may require us to visit all the cuts in this lattice in the worst case. Such scenarios occur when we do not have specific knowledge of the predicate, or we cannot exploit its structure to detect it efficiently. The lattice, $\mathcal{C}(E)$, is a directed acyclic graph (DAG) whose vertices are the consistent cuts, and there is a directed edge from vertex u to vertex v if state represented by v can be reached by executing one event on u. Recall that the *rank* of a consistent cut is the total number of events executed in it, hence we also have $rank(v) = rank(u) + 1$. The source of $\mathcal{C}(E)$ is the empty set: a consistent cut in which no events have been executed on any process. The sink of this DAG is E: the consistent cut in which all the events of the computation have been executed. Figure 3.1b presents a visual illustration of such a lattice.



(a) Computation



(b) Lattice of Consistent Cuts

Figure 3.1: Illustration: Level by Level (BFS) Traversal of Lattice of Consistent Cuts

Cooper and Marzullo [23] gave the first algorithm for enumerating consistent cuts 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. The exponential space requirement is due to the standard BFS approach in which consistent cuts of rank r must be stored to traverse the cuts of rank $r + 1$.

There is also a body of work on enumeration of consistent cuts in order different than BFS. Alagar and Venkatesan [2] presented a depth first algorithm using the notion of global interval which reduces the space complexity to $\mathcal{O}(|E|)$. Steiner [65] gave an algorithm that uses $\mathcal{O}(|E| \cdot i(P))$ time, and Squire [64] further improved the computation time to $\mathcal{O}(\log |E| \cdot i(P))$. Pruesse and Ruskey [57] 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. [43] and Habib et al. [38] improved the space complexity to $\mathcal{O}(n \cdot |E|)$.

Ganter [31] presented an algorithm, which uses the notion of lexical order, and Garg [33] 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 [12] gave a parallel algorithm to traverse this lattice in lexical order, and QuickLex [10] 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.

3.1 Traditional BFS Traversal Algorithm

Cooper and Marzullo's algorithm [23] enumerates all the consistent cuts in $\mathcal{C}(E)$ in a breadth-first manner. Even though they focussed on distributed systems, their algorithm has been subsequently adopted for verification of shared-memory parallel programs too [17, 28]. Breadth-first search (BFS) of this lattice starts from the source vertex and visits all the cuts of rank 1; it then visits all the cuts of rank 2 and continues in this manner till reaching the last consistent cut of rank $|E|$. For example, in Figure 3.1b the BFS algorithm will traverse cuts in the following order: $[0,0],[0,1],[1,0],[0,2],[1,1],[0,3],[1,2],[1,3],[2,2],[2,3],[3,2],$

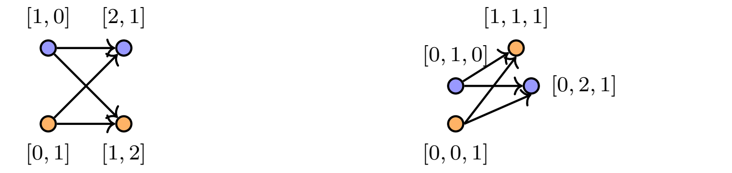
The standard BFS on a graph needs to store the vertices at distance d from the source to be able to visit the vertices at distance $d+1$ (from the source). Hence, in performing a BFS on $\mathcal{C}(E)$ we are required to store the cuts of rank r in order to visit the cuts of rank $r+1$. Observe that in a parallel computation there may be exponentially many — in the number of processes — cuts of rank r. Thus, traversing the lattice $\mathcal{C}(E)$ requires space which is exponential in the number of processes.

BFS based traversal of lattice of consistent cuts provides two key advantages in analysis of parallel programs: it is guaranteed to find an erroneous global state (consistent cut) — that violates an invariant — with the least number of events. In addition, it can also be used to enumerate consistent cuts of a given rank(s). The worst case space requirement of BFS based traversal, however, is exponential in the number of processes involved in the computation. This space requirement can be often prohibitive in analyzing parallel computations.

3.2 BFS Traversal Algorithm using Uniflow Partition

We now show that BFS traversal of the lattice of consistent cuts of any computation can be performed in space that is polynomial in the size of the input. We do this by extending the algorithm given in [34]. We use a computation's uniflow chain partition and enumerate its consistent cuts in increasing order of ranks. We start from the empty cut, and then traverse all consistent cuts of rank 1, then all consistent cuts of rank 2 and so on. In this chapter, we use the vector clock notation of consistent cuts for the presentation of our algorithms. For any rank r, $1 \leq r \leq |E|$, we traverse the consistent cuts in the following lexical order:

Definition 12 (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,* $G <_l H \equiv \exists k : (G[k] < H[k]) \wedge (\forall i : n \geq i > k : G[i] = H[i])$



(a) Original vector clocks

(b) Renegerated vector clocks for uniflow partition

Figure 3.2: Vector clocks of a computation in its original form, and in its uniflow partition

Recall from our vector clock notation (Section 2.2) that the right most entry in the vector clock is for the lowest chain. Also, the vector clocks are dependent on chain partition. Consider the poset with a non-uniflow chain partition in Figure 3.2a. The vector clocks of its events are shown against the four events. The lexical order on the consistent cuts of this chain partition is:

$[0, 0] <_l [0, 1] <_l [1, 0] <_l [1, 1] <_l [1, 2] <_l [2, 1] <_l [2, 2]$. For the same poset, Figure 3.2b shows the equivalent uniflow partition, and the corresponding vector clocks that are regenerated using Algorithm 1. The lexical order on the consistent cuts for this uniflow chain partition is: $[0, 0, 0] <_l [0, 0, 1] <_l [0, 1, 0] <_l [0, 1, 1] <_l [0, 2, 1] <_l [1, 1, 1] <_l [1, 2, 1]$.

Note that the number of consistent cuts remains same for both of these chain partitions, and there is a one-to-one mapping between the consistent cuts in the two partitions.

Algorithm 2 TraverseBFSUniflow(P)

```
Input: A poset $P = (E, \rightarrow)$ that has been partitioned into a uniflow chain partition
    of $n_u$ chains, and the vector clock of the events have been regenerated for this
    partition.
1: $G$ = new int[$n_u$]  // initial consistent cut with rank 0
2: enumerate($G$)  // evaluate the predicate on empty cut $G$.
3: for ($r = 1; r \leq |E|; r++$) do
4:     //make $G$ lexically smallest cut of given rank
5:     $G$ = GetMinCut($G, r$)
6:     while $G \neq$ null do
7:         enumerate($G$)  // evaluate the predicate on $G$.
8:         //find the next bigger lexical cut of same rank
9:         $G$ = GetSuccessor($G, r$)
```

Algorithm 2 shows the steps of our BFS traversal using a computation in a uniflow chain partition. In generating the input for this algorithm, we perform two pre-processing steps: (a) finding a uniflow partition, and (b) regenerating vector clocks for this partition. These steps are performed only once for a computation, and are relatively inexpensive in comparison to the traversal of lattice. Later, in Section 3.3, we show how to implement our uniflow partition based BFS algorithm without regeneration of the vector clocks.

Algorithm 3 GetMinCut(G, r)

```
Input: $G$: a consistent cut of poset $P$ from Algorithm 2
Output: Smallest consistent cut of rank $r$ that is lexically greater than or equal to
    $G$.
1: $d = r - rank(G)$  // difference in ranks
2: for ($j = 1; j \leq n_u; j = j + 1$) do
3:     if  $d \leq size(\mu_j) - G[j]$ then
4:         $G[j] = G[j] + d$
5:         return $G$
6:     else  // take all the elements from chain $j$
7:         $G[j] = G[j] + size(\mu_j)$
8:         $d = d - size(\mu_j)$
```

For each rank r, $1 \leq r \leq |E|$, Algorithm 2 first finds the lexically smallest consistent cut at of rank r. This is done by the GetMinCut (shown in Algorithm 3) routine that returns the lexically smallest consistent cut of P bigger than G of rank r. For example, in Figure 3.3, GetMinCut($[0, 0, 0], 4$) returns $[0, 1, 3]$. Given a consistent cut G of rank r, we repeatedly find the next lexically bigger consistent cut of rank r using the routine GetSuccessor given in Algorithm 4. For example, in Figure 3.3, GetSuccessor($[0, 0, 3], 3$) returns the next lexically smallest consistent cut $[0, 1, 2]$.

The GetMinCut routine on poset P assumes that the rank of G is at most r and that G is a consistent cut of the P. It first computes d as the difference between r and the rank of G. We need to add d elements to G to find the smallest consistent cut of rank r. We exploit the Uniflow Cut Lemma (Lemma 3) by adding as many elements from the lowest chain as possible. If all the elements from the lowest chain are already in G, then we continue with the second lowest chain, and so on.

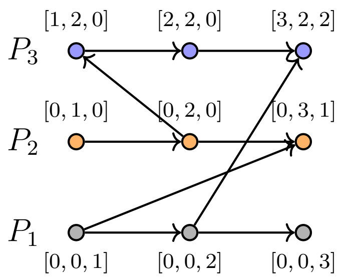


Figure 3.3: Illustration for GetSuccessor: Computation in uniflow partition on three processes

Algorithm 4 GetSuccessor(G, r)

Input: G: a consistent cut of rank r

Output: K: lexical successor of G of rank r

```
1:  $K = G$  // Create a copy of $G$ in $K$
2:  for $(i = 2; i \leq n_u; i{+}{+})$ do  // lower chains to higher
3:    if next element on $P_i$ exists then
4:      $K[i] = K[i] + 1$ // increment cut
5:      for $(j = i - 1; j > 0; j - -)$ do
6:        $K[j] = 0$ // reset lower chains
7:      //fix dependencies on lower chains
8:      for $(j = i + 1; j \leq n_u; j + +)$ do
9:        for $(k = i - 1; k > 0; k - -)$ do
10:         $vc$ = vector clock of event number $G[j]$ on $P_j$
11:         $K[k] = \mathrm{MAX}(vc[k], K[k])$
12:     if $rank(K) \leq r$ then
13:       return GetMinCut$(K, r)$
14: return null  // no candidate cut
```

For example, consider the computation in Figure 3.3, and its consistent cut $G = [0, 0, 2]$. Now let us try to find G's lexical successor at rank 5. In this case, we add all three elements from P_1 to reach $[0, 0, 3]$, and then add first two elements from P_2 to get the answer as $[0, 2, 3]$.

The GetSuccessor routine (Algorithm 4) finds the lexical successor of G at rank r. The approach for finding a lexical successor is similar to counting numbers in a decimal system: if we are looking for successor of 2199, then we can't increment the two 9s (as we are only allowed digits 0-9), and hence the first possible increment is for entry 1. We increment it to 2, but we must now reset the entries at lesser significant digits. Hence, we reset the two 9s to 0s, and get the successor as 2200.

In our GetSuccessor routine, we start at the second lowest chain in a uniflow poset, and if possible increment the cut by one event on this chain. We then reset the entries on lower chains, and then make the cut consistent by satisfying all the causal dependencies. If the rank of the resulting cut is less than or equal to r, then calling the GetMinCut routine gives us the lexical successor of G at rank r.

Line 1 copies cut G in K. The for loop covering lines 2–13 searches for an appropriate element not in G such that adding this element makes the resulting consistent cut lexically greater than G. We start the search from chain 2, instead of chain 1, because for a non-empty cut G adding any event from the lowest chain to G will only increase G's rank as there are no lower chains to reset. Line 3 checks if there is any possible element to add in P_i. If yes, then lines 4–6 increment K at chain i, and then set all its values for lower chains to 0. To ensure that K is a consistent cut, for every element in

K, we add its causal dependencies to K in lines 7–11. Line 12 checks whether the resulting consistent cut is of rank $\leq r$. If $rank(K)$ is at most r, then we have found a suitable cut that can be used to find the next lexically bigger consistent cut and we call GetMinCut routine to find it. If we have tried all values of i and did not find a suitable cut, then G is the largest consistent cut of rank r and we return null.

In Figure 3.3, consider the call of GetSuccessor $([1,2,3],6)$. As there is no next element in P_1, we consider the next element in P_2. After line 5, the value of K is $[1,3,0]$, which is not consistent. Lines 7–10 make K a consistent cut, now $K = [1,3,1]$. Since $rank(K)$ is 5, we call GetMinCut at line 13 to find the smallest consistent cut of rank 6 that is lexically bigger than $[1,3,1]$. This consistent cut is $[1,3,2]$.

3.2.1 Proof of Correctness

Lemma 4. *Let G be any consistent cut of rank at most r. Then, $H =$ Get-MinCut is the lexically smallest consistent cut of rank r greater than or equal to G.*

Proof. We first show that H is a consistent cut. Initially, H is equal to G which is a consistent cut. We show that H continues to be a consistent cut after every iteration of the *for* loop. At iteration j, we add elements from the j^{th} chain from the bottom to H. Since all elements from higher numbered chains are already part of H, and all elements from lower numbered chains cannot be smaller than any of the newly added element, we get that H continues to be a consistent cut.

By construction of our algorithm it is clear that rank of H is exactly r. We now show that H is the lexically smallest consistent cut of rank r greater than or equal to G. Suppose not, and let $W <_l H$ be the lexically smallest consistent cut of rank r greater than or equal to G. Since $W <_l H$, let k be the smallest index such that $W[k] < H[k]$. Since $G \leq_l W$, k is one of the indices for which we have added at least one event to G. Because rank of W equals rank of H, there must be an index k' lower than k such that $W[k'] > H[k']$. However, our algorithm forces that for H for any index k' lower than k, $H[k']$ equals $|P_{k'}|$. Hence, $W[k']$ cannot be greater than $H[k']$. □

Lemma 5. *Let G be any consistent cut of rank at most r, Then* GetSuccessor *returns the least consistent cut of rank r that is lexically greater than 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 elements in chain P_i are already included in G, or on inclusion of the next element in P_i, z, the smallest consistent cut that includes z has rank greater than r. Hence, G is lexically biggest consistent cut of rank r.

Now consider the case when W is the consistent cut returned at line 16 by GetMinCut(K, r). We first observe that after executing line 11, K is the next lexical consistent cut (of *any* rank) after G. If $rank(K)$ is at most r, then by Lemma 4 we know that GetMinCut(K, r) returns the smallest lexical consistent cut greater than or equal to G of rank r. If $rank(K)$ is greater than r, then there is no consistent cut of rank r such that $\forall k : i + 1 \leq k \leq n_u$: $K[k] = G[k]$ and $K[i] > G[i]$ and $rank(K) \leq r$. Thus, at line 16 we use the largest possible value of i for which there exists a lexically bigger consistent cut than G of rank r. □

3.2.2 Complexity Analysis

We regenerate vector clocks of the events in the computation $P = (E, \rightarrow)$ after finding its uniflow chain partition μ. If μ has n_u chains then each event's regenerated vector clock in μ will have length n_u. Hence, we require $\mathcal{O}(n_u \cdot |E|)$ space to store the computation in the uniflow partition.

The GetMinCut routine goes over n_u chains at most, and for each iteration performs constant work. Thus, the time complexity of GetMinCut is $\mathcal{O}(n_u)$. Finding the lexical successor of a cut using the GetSuccessor routine takes $\mathcal{O}(n_u^3)$ time. This is due to the three nested for-loops — at lines 2, 8 and 9 — that iterate over the chains of μ.

3.2.3 GetSuccessor in $\mathcal{O}(n_u^2)$ Time

We now present an optimization to find the lexical successor of any consistent cut in $\mathcal{O}(n_u^2)$ time, instead of $\mathcal{O}(n_u^3)$ time taken in GetSuccessor. We do so by using additional $\mathcal{O}(n_u^2)$ space.

Observe that GetSuccessor routine iterates over $n_u - 1$ chains in the

Algorithm 5 ComputeProjections(G)

```
Input: $G$: a consistent cut of rank $r$
for ($i = n_u; i \geq 1; i - -$) do  // go top to bottom
    $val = G[i]$  // event number in $G$ on chain $i$
    $vc =$ vector clock of event num $val$ on chain $i$
    if $i == n_u$ then  // on highest chain
        $proj[i] = vc$
    else // process relevant entries in vector
        for ($j = i; j > 0; j - -$) do
            //projection on chain $i$:
            $proj[i][j] =$MAX$(vc[j], proj[i+1][j])$
```

outer loop at line 2, and the two inner loops at lines 8 and 9 perform $\mathcal{O}(n_u^2)$ work in the worst case. When we cannot find a suitable cut of rank less than or equal to r (check performed at line 12), we move to a higher chain (with the outer loop at line 2). Thus, we repeat a large fraction of the $\mathcal{O}(n_u^2)$ work in the two inner loops at lines 8 and 9 for this higher chain. We can avoid this repetition by storing the combined causal dependencies from higher chains on each lower chain.

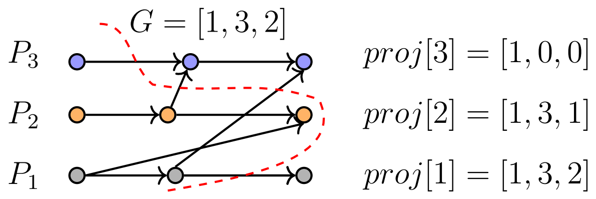


Figure 3.4: Illustration: Projections of a cut on chains

Let us illustrate this with an example. Consider the uniflow computation shown in Figure 3.4. Suppose we want the lexical successor of $G = [1, 3, 2]$. Then, for each chain, starting from the top we compute the projection of events included in G on lower chains. For example, $G[3] = 1$, and thus on the top-most chain, the projection is only the vector clock of the first event on P_3, which is $[1, 0, 0]$. Thus $proj[3] = [1, 0, 0]$. On P_2, the projection must include the combined vector clocks of $G[3]$ and $G[2]$ — the events from top two chains. As $G[2] = 3$, we use the vector clock of third event on P_2, which is $[0, 3, 1]$ as that event is causally dependent on first event on P_1. Combining the two vectors gives us the projection on P_2 as $proj[2] = [1, 3, 1]$.

Algorithm 5 shows the steps involved in computing the projections of a cut on each chain. We create an auxiliary matrix, $proj$, of size $n_u \times n_u$, to store these projections. In GetSuccessor routine, once we have computed a new successor by using some event on chain i, we need to update the stored projections on chains lower than i; and not all n_u chains. This is because the projections for unchanged entries in G above chain i will not change on chain i, or any chain above it. Hence, we only update the relevant rows and columns — rows and columns with number i or lower — in $proj$; i.e. only the upper triangular part of the matrix $proj$. We keep track of the chain that gave us the successor cut, and pass it as an additional argument to Algorithm 5. We read and update $n_u^2/2$ entries in the matrix, and not all n_u^2 of them.

Hence, the optimized implementation of finding the lexical successor of G requires two changes. First, every call of GetSuccessor (G, r) starts with computing the projections of G using Algorithm 5. Second, we replace the two inner for loops at lines 8 and 9 in GetSuccessor by one $\mathcal{O}(n_u)$ loop to compute the max of the two vector clocks: vector clock of $K[i]$, and $proj[i]$.

The optimized implementation with these changes is shown in Algorithm 6 .

Algorithm 6 GetSuccessorOptimized(G, r)

```
Input: $G$: a consistent cut of rank $r$
Output: $K$: lexical successor of $G$ with rank $r$
1: ComputeProjections($G$)  // $G$'s projections
2: $K = G$  // Create a copy of $G$ in $K$
3: for $(i = 2; i \le n_u; i{+}{+})$ do
4:   if next element on $P_i$ exists then
5:     $K[i] = K[i] + 1$ // increment cut in $P_i$
6:     //fix dependencies using projections
7:     $vc =$ vector clock of event number $K[i]$ on $P_i$
8:     //take component-wise max
9:     for $(k = i - 1; k > 0; k - -)$ do
10:      $K[k] = \text{MAX}(vc[k], proj[i][k])$
11:    if $rank(K) \le r$ then
12:      return GetMinCut($K, r$)  // make $K$'s rank equal to $r$
13: return null  // could not find a candidate cut
```

3.2.4 Re-mapping Consistent Cuts to Original Chain Partition

The number of consistent cuts of a computation is independent of the chain partition used. Their vector clock representation, however, varies with chain partitions as the vector clocks of events in the computation depend on the chain partition used to compute them. There is a one-to-one mapping between a consistent cut in the original chain partition of the computation on n chains (processes), and its uniflow chain partition on n_u chains. We now show how to map a consistent cut in a uniflow chain partition to its equivalent cut in the original chain partition of the computation. Let $P = (E, \rightarrow)$ be a computation on n processes, and let n_u be the number of chains in its uniflow chain partition. If G_u is a consistent cut in the uniflow chain partition, then its equivalent consistent cut G for the original chain partition (of n chains) can be found in $\mathcal{O}(n_u + n^2)$ time.

Algorithm 7 REMAP(G_u, n_u, n)

```
Input: $G_u$: a consistent cut in uniflow chain partition on $n_u$ chains
Output: $G$: equivalent consistent cut in original chain partition on $n$ chains
 1: $G$ = new int[$n$]  // allocate memory for $G$
 2: $I$ = new int[$n_u$]  // reduction vector
 3: for ($i = n_u; i \geq 1; i - -$) do  // go over all the uniflow chains
 4:     $uvc$ =event number $G_u[i]$'s vector-clock on uniflow chain $i$
 5:     //chain of this event in original poset
 6:     $c$ = ORIGINALCHAIN($uvc$)
 7:     //$uvc$'s event number on chain $c$ in original poset
 8:     $e$ = ORIGINALEVENT($uvc$)
 9:     if $I[c] < e$ then  // update indicator with $e$
10:         $I[c] = e$
11: for ($j = n; i \geq 1; i - -$) do  // go over chains in original poset
12:     $vce$ =event number $I[j]$'s vector-clock on chain $j$ in original poset
13:     for ($k = n; k \geq 1; k - -$) do  // update $G$ entries
14:         $G[k]$ =MAX($G[k], vce[k]$)
15: return $G$
```

We do so by mapping two additional entries with the new vector clock of each event for uniflow chain partition: the chain number c, and event number e from the original chain partition over n chains. For example, in Figure 3.2b, for uniflow vector clock $[1, 1, 1]$, its chain number in original poset is 1, and its event number on that chain is 2. When generating the uniflow vector clocks, we populate these entries in a map. Given a uniflow vector clock uvc, the call to ORIGINALCHAIN(uvc) returns c, and ORIGINALEVENT(uvc) returns e. To compute G from G_u, we use these two values from the corresponding event for each entry in G_u. We start with I as an all-zero vector of length n. Now, we iterate over G_u, and we update I by setting $I[c] = max(I[c], e)$. As vector G_u has length n_u, this step takes $\mathcal{O}(n_u)$ time. We now initiate G as an all-zero vector clock of length n, and for each entry $I[k]$, $1 \leq k \leq n$, we get the vector clock, vce, of event $I[k]$ on chain k in the original computation. We then set G to the component-wise maximum of G and vce. As there are n entries in I, and for each non-zero entry we perform $\mathcal{O}(n)$ work in updating G (in lines 11–14 in Algorithm 7) the total work in this step is $\mathcal{O}(n^2)$.

3.3 Implementation without Regeneration of Vector Clocks

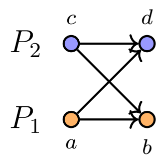
Our discussion of GetMinCut (Algorithm 3) and GetSuccessor (Algorithm 4) required that the vector clocks of the events must be regenerated for the uniflow chain partition. We now discuss how to implement the algorithms presented earlier in this chapter without regenerating the vector clocks for the uniflow chain partition of the computation.

Suppose the original computation under analysis, $P = (E, \rightarrow)$, is on n processes. Then, this computation is stored as vector clocks, and state variables of $|E|$ events on n chains. Note that a chain partition is only a way of positioning elements of the poset. Thus, after finding the uniflow chain partition μ, we can only reposition the events on their respective uniflow chains, and do not need to regenerate their vector clocks. In our implementation under this approach, there are n_u chains in μ, and each of them is stored as an array whose entries store the original vector clocks, and the state variables for each event. For example, the computation on two processes in Figure 3.6a is not in uniflow partition. Figure 3.6b 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.

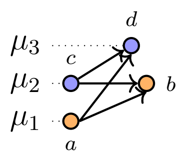
3.3.1 Retaining Original Vector Clocks in Uniflow Partition

Our presentation of algorithms uses $G[i]$ to denote the number of events from chain μ_i that are included in a consistent cut G. We maintain this information by assigning an index (in the range 1 to $size(\mu_i)$) to each event on the chain. 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 3.5, 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 using the optimized approach of Section 3.2.4 in $\mathcal{O}(n_u + n^2)$ time.



(a) Computation

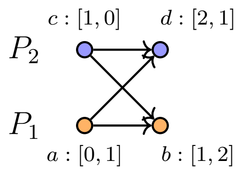


(b) Uniflow Partition

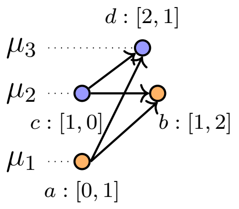
$$G = \{a, b, c, d\} \implies G_u[0] = 1, G_u[1] = 2, G_u[2] = 1$$
$$G = \{a, c, d\} \implies G_u[0] = 1, G_u[1] = 1, G_u[2] = 1$$
$$G = \{a, c, b\} \implies G_u[0] = 1, G_u[1] = 2, G_u[2] = 0$$
$$G = \{a, c\} \implies G_u[0] = 1, G_u[1] = 1, G_u[2] = 0$$
$$G = \{a\} \implies G_u[0] = 1, G_u[1] = 0,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

(c) G values and their respective G_u vectors

Figure 3.5: Illustration: Maintaining indicator vector G_u for a cut G



(a) Computation



(b) Uniflow Partition

$$\begin{array}{l} d : [2,1] \\ b : [1,2] \\ c : [1,0] \\ a : [0,1] \end{array}$$

(c) Events in Uniflow Order

$$\begin{array}{l} J[4] = [2,2] \\ J[3] = [1,2] \\ J[2] = [1,1] \\ J[1] = [0,1] \end{array}$$

(d) J Vector

Figure 3.6: Illustration: Computing J vector for optimizing GetMinCut

3.3.2 GetMinCut

Observe that in the GetMinCut routine we add events to any cut in increasing uniflow order (Definition 7). We do not skip any event, and only return when the cut has the required rank r. 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 $J[i-1]$ 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 GetMinCut (G, r). If G is empty, then we return the entry $J[r]$ as the result. This takes constant time. When G is non-empty, given that J will contain entries (vector clocks) in increasing order, we can perform binary search on it to find the result. We use the rank of the resulting cut formed by joining G with the entry in J to guide our binary search.

Consider the computation in Figure 3.6a that has four events, and its uniflow partition in Figure 3.6b. The increasing order on the vector clocks of all the four events is in Figure 3.6c. Starting from the bottom (vector $[0, 1]$), and performing the joins, we get J as shown in Figure 3.6d.

Computing and storing the vector J requires $\mathcal{O}(n \cdot |E|)$ time and space. After computing J, each call to GetMinCut (G, r) takes $\mathcal{O}(n \log |E|)$ time with binary search when G is non-empty. This is because there are at most $\log |E|$ iterations to find the result, and at each iteration we do $\mathcal{O}(n)$ work to find the join of two vector clocks and compute its rank. As we discussed earlier, when G is the empty cut a call to GetMinCut (G, r) takes $\mathcal{O}(1)$ time irrespective of the value of r.

3.3.3 ComputeProjections

The optimized GetSuccessor algorithm presented in Section 3.2.3 requires $\mathcal{O}(n_u^2)$ time and space. This is because the ComputeProjections routine requires $\mathcal{O}(n_u^2)$ time and space to compute the projections of cuts as each vector clock is of length n_u after its regeneration under the uniflow chain partition. When we do not regenerate the vector clocks, and only use the indicator vector G_u as discussed above, we only require $\mathcal{O}(n_u \cdot n)$ time and space to compute the projections. We use the indicator vector G_u to compute the projections, and each of these projections now takes $\mathcal{O}(n)$ space — the space taken by a vector clock in the original computation. We show the modified routine in Algorithm 8.

Algorithm 8 ComputeProjections(G_u) with original vector clocks

```
1: for $(i = n_u; i \geq 1; i--)$ do  // start from top, move down
2:   $val = G_u[i]$  // event number in $G_u$ on chain $i$
3:   $vc = \mu_i[val].VC$  // vector clock of event number $val$ on chain $i$
4:   if $i == n_u$ then  // on highest chain
5:     $proj[i] = vc$
6:   else
7:     for $(j = n; j > 0; j--)$ do  // projection on chain $i$ is max of two vectors
8:       $proj[i][j] = \max(vc[j], proj[i+1][j])$
```

Let us illustrate this with an example. Consider the uniflow computation shown in Figure 3.7 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

$G = [1, 2],\ G_u[1] = 1, G_u[2] = 2, G_u[3] = 0$

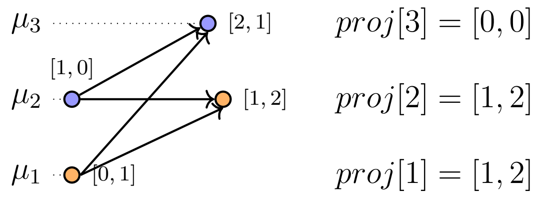


Figure 3.7: Illustration: Projections of cuts on uniflow chains without regeneration of vector clocks

$G_u[3] = 0, G_u[2] = 2, G_u[1] = 1$. 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 $proj[1] = [1, 2]$.

After the modifications discussed above, the modified GetSuccessor algorithm takes $\mathcal{O}((n_u + \log |E|) \cdot n)$ time in the worst case. To achieve this improved time complexity, we require $\mathcal{O}((|E| + n_u) \cdot n)$ additional space: $\mathcal{O}(n \cdot |E|)$ space to store the computed J vector, and $\mathcal{O}(n \cdot n_u)$ to store the projections, $proj$ vector, of a consistent cut on n_u chains.

3.4 Comparison with Other Traversal Algorithms

Based on the optimized implementation of our algorithms, we have the following result:

Theorem 2. *Given a computation $P = (E, \rightarrow)$ on n processes, Algorithm 2 performs breadth-first traversal of its lattice of consistent cuts using $\mathcal{O}((n_u + |E|) \cdot n)$ space which is polynomial in the size of the computation.*

Proof. Storing the original computation, and the computed J vector requires $\mathcal{O}(n \cdot |E|)$ space — each event's vector clock has n integers. Storing the projections requires $\mathcal{O}(n \cdot n_u)$ space. The G_u vector takes $\mathcal{O}(n_u)$ space.

As, $n_u \leq |E|$, the worst case space complexity of our BFS traversal algorithm is $\mathcal{O}(n \cdot |E|)$ which is polynomial in the size of the input. $\square$

3.4.1 Traversing Consistent Cuts of Specific Rank(s)

A key benefit of our algorithm is that it can traverse all the consistent cuts of a given rank, or within a range of ranks, without traversing the cuts of lower ranks. In contrast, the traditional BFS traversal must traverse, and store, consistent cuts of rank $R-1$ to traverse cuts of rank R, which in turn requires it to traverse cuts of rank $R-2$ and so on. Other algorithms such as DFS [2], and Lex [31, 33] may traverse the all the consistent cuts of the lattice in the worst case to enumerate cuts of a specified rank.

To traverse all the cuts of rank R, we just change the loop bounds at line 3 in Algorithm 2 to for $(r = R; r \leq R; r{+}{+})$. Thus, starting with an empty cut

Algorithm	Space Required
Traditional BFS [23]	$\mathcal{O}(\frac{m^{n-1}}{n})$
DFS [2]	$\mathcal{O}(\|E\|)$
Lex [31, 33]	$\mathcal{O}(n)$
Original Uniflow-BFS*	$\mathcal{O}((n_u + \|E\|) \cdot n_u)$
Optimized Uniflow-BFS*	$\mathcal{O}((n_u + \|E\|) \cdot n)$

Table 3.1: Space complexities of algorithms for traversing lattice of consistent cuts; here $m = \frac{|E|}{n}$. * denotes algorithms in this dissertation.

we find the lexically smallest consistent cut of rank R with the GETMINCUT routine. Then we repeatedly find its lexical successor of the same rank, until we have traversed the lexically biggest cut of rank R. Similarly, consistent cuts between the ranks of R_1 and R_2 can be traversed by changing the loop at line 3 in Algorithm 2 to: for $(r = R_1; r \leq R_2; r++)$.

Consider a computation $P = (E, \rightarrow)$ on n processes, whose uniflow chain partition μ has n_u chains. In Table 3.1, we compare the worst-case space complexities of BFS, DFS, and Lex traversal algorithms, against that of our uniflow partition based BFS algorithm.

Let L_r denote the number of consistent cuts of rank r for the computation. In Table 3.2, we compare the worst-case time complexities of these traversal algorithms to traverse this level of the lattice.

In the next chapter, we extend the notion of lexical order based traversal to enumerate consistent cuts that satisfy two important categories of predicates.

Algorithm	Time	
	Per cut	Level r
Traditional BFS [23]	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2 \sum_{i=1}^{r} L_i)$
DFS [2]	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2 \sum_{i=1}^{\lvert E\rvert} L_i)$
Lex [31, 33]	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2 \sum_{i=1}^{\lvert E\rvert} L_i)$
Original Uniflow-BFS*	$\mathcal{O}(n_u^2)$	$\mathcal{O}(n_u^2 \cdot L_r)$
Optimized Uniflow-BFS*	$\mathcal{O}((n_u + \log \lvert E\rvert) \cdot n)$	$\mathcal{O}((n_u + \log \lvert E\rvert) \cdot n \cdot L_r)$

Table 3.2: Space and Time complexities for traversing level r of the lattice of consistent cuts. * algorithm denotes in this dissertation.

3.5 Experimental Evaluation

We conduct an experimental evaluation to compare the space and time required by traditional BFS, Lex [33, 12], and our uniflow based traversal algorithm to traverse consistent cuts of specific ranks, as well as all consistent cuts up to a given rank. We do not evaluate DFS [3] implementation as previous studies have shown that Lex implementation outperforms DFS based traversals in both time and space [33, 12, 10]. Lexical enumeration is significantly better for enumerating all possible consistent cuts of a computation [12, 10]. However, it is not well suited for only traversing cuts of a specified ranks, or finding the smallest counter example. For these tasks, BFS traversal remains the algorithm of choice. We optimize the traditional BFS implementation as per [33] to enumerate every global state exactly once. We use seven benchmark computations from recent literature on traversal of consistent cuts [12, 10]. The details of these benchmarks are shown in Table 3.3. Benchmarks *d-100, d-300* and *d-500* are randomly generated posets for modeling distributed computations. Each of them simulates a distributed computation

Name	n	$\|E\|$	Approx. # of cuts
d-100	10	100	1.2×10^6
d-300	10	300	4.3×10^7
d-500	10	500	4.9×10^9
bank	8	96	8.2×10^8
hedc	12	216	4.5×10^9
w-4	4	480	9.3×10^6
w-8	8	480	7.3×10^9

Table 3.3: Benchmark details

on $n = 10$ processes, with varying number of events: *d-100* has 100 events, and *d-300*, *d-500* have 300 and 500 events respectively. After each internal event, every processes sends a message to a randomly selected process with a probability of 0.3. The benchmarks *bank*, and *hedc* are computations obtained from real-world concurrent programs that are used by [17, 28, 68] for evaluating their predicate detection algorithms. The benchmark *bank* contains a typical error pattern in concurrent programs, and *hedc* is a web-crawler. Benchmarks *w-4* and *w-8* have 480 events distributed over 4 and 8 processes respectively, and help to highlight the influence of degree of parallelism on the performance of enumeration algorithms.

We conduct two sets of experiments: (a) complete traversal of lattice of consistent cuts (of the computation) in BFS manner, and (b) traversal of cuts of specific ranks. We conduct all the experiments on a Linux machine with an Intel Core i7 3.4GHz CPU, with L1, L2 and L3 caches of size 32KB, 256KB, and 8192KB respectively. We compile and run the programs on Oracle Java 1.7, and limit the maximum heap size for Java virtual machine (JVM) to 2GB.

For each run of our traversal algorithm, we use Algorithm 1 to find the uniflow chain partition of the poset. The runtimes and space reported for our uniflow traversal implementation include the time and space needed for finding and storing the uniflow chain partition of the poset.

3.5.1 Results with Regenerated Vector Clocks

Our initial presentation of uniflow based BFS traversal algorithms required that vector clocks of events be regenerated for uniflow chain partitions. Under this setup, we require $\mathcal{O}((n_u + |E|) \cdot n_u)$ space and take $\mathcal{O}(n_u^2)$ time to enumerate a consistent cut of the computation. We first present the results of our experiments under this setup.

Table 3.4 compares runtimes, and the sizes of JVM heap for traditional BFS and our uniflow based BFS traversal of lattice of consistent cuts of the benchmarks. The traditional BFS implementations runs out of memory on *hedc, bank*, and *w-8*. Our implementation requires significantly less memory despite regenerating vector clocks whose lengths are usually bigger than the original vector clocks. Even though our implementation is slower, it enables us to do BFS traversal on large computations — something that is impossible with traditional BFS due to its memory requirement.

Table 3.5 highlights the strength of our algorithm in traversing consistent cuts of specific ranks. We compare our implementation with traditional BFS as well as the implementation of Lexical traversal. For traversing consistent cuts of three specified ranks (equal to quarter, half, and three-quarter of

Name	n_u	T_{part}	Traditional BFS		Uniflow BFS	
			Space	Time	Space	Time
d-100	26	0.030	108	0.48	31	0.37
d-300	68	0.031	842	16.84	33	46.20
d-500	112	0.033	893	108.07	34	607.55
bank	8	0.023	×	×	59	73.2
hedc	26	0.028	×	×	56	1129
w-4	121	0.036	258	0.99	25	8.59
w-8	63	0.032	×	×	40	1445.57

Table 3.4: Heap-space consumed (in MB) and runtimes (in seconds) for two BFS implementations to traverse the full lattice of consistent cuts. T_{part} = time (seconds) to find uniflow partition; × = out-of-memory error.

Name	$r = \frac{\vert E\vert}{4}$			$r = \frac{\vert E\vert}{2}$			$r = \frac{3\vert E\vert}{4}$		
	tbfs	lex	uni	tbfs	lex	uni	tbfs	lex	uni
d-100	0.12	0.10	0.06	0.22	0.11	0.05	0.20	0.89	0.04
d-300	0.39	1.23	0.05	2.70	1.15	0.07	6.33	1.25	0.13
d-500	2.29	5.73	0.11	7.83	6.52	0.33	67.59	6.86	1.48
bank	3.36	16.80	0.27	×	16.34	3.07	×	17.02	0.32
hedc	4.72	16.50	0.50	×	152.76	15.70	×	153.54	0.51
w-4	0.09	0.18	0.07	0.53	0.18	0.10	0.93	0.19	0.09
w-8	26.39	143.08	0.72	×	171.23	12.27	×	169.21	3.09

Table 3.5: Runtimes (in seconds) for tbfs: Traditional BFS, lex: Lexical, and uni: Uniflow BFS implementations to traverse cuts of given ranks; × = out-of-memory error.

number of events) our algorithm is consistently and significantly faster than both traditional BFS, as well as Lex algorithm. Thus, it can be extremely helpful in quickly analyzing traces when the programmer has knowledge of the conditions when an error/bug occurs.

In addition, there are many cases when we are not interested in checking

Name	$r \leq 32$		
	tbfs	lex	uni
d-100	0.19	0.93	0.12
d-300	0.20	1.22	0.14
d-500	0.19	4.93	0.19
bank	45.43	16.87	5.70
hedc	0.23	128.60	0.12
w-4	0.01	0.13	0.05
w-8	0.02	196.21	0.05

Table 3.6: Runtimes (in seconds) for tbfs: Traditional BFS, lex: Lexical, and uni: Uniflow BFS implementations to traverse cuts of ranks upto 32.

all consistent cuts of a computation. It has been argued that most concurrency related bugs can be found relatively early in execution traces [53, 4]. As highlighed by Table 3.6, we also perform well in visiting all consistent cuts of rank less than or equal to 32. Hence, our implementation is faster on most benchmarks for smaller ranks, and has a much smaller memory footprint (see Table 3.7). These results emphasize that our algorithm is useful for practical debugging tasks while consuming less resources.

3.5.2 Results without Regenerated Vector Clocks

We now present the results of our experiments for the implementations of our algorithms in which we use the original vector clocks of the computation. Recall that under this setting we require $\mathcal{O}((n_u + |E|) \cdot n)$ additional space, and we take $\mathcal{O}((n_u + \log |E|) \cdot n)$ time in the worst case to enumerate a consistent cut of the computation. From here on, we use the term UniR for the implementation that regenerates vector clocks of events for the uniflow chain

Name	$r = \frac{\lvert E\rvert}{4}$			$r = \frac{\lvert E\rvert}{2}$			$r = \frac{3\lvert E\rvert}{4}$			$r \leq 32$		
	tbfs	lex	uni	tbfs	lex	uni	tbfs	lex	uni	tbfs	lex	uni
d-100	95	32	41	121	29	41	134	32	42	112	32	42
d-300	107	33	53	342	32	54	583	32	54	113	31	42
d-500	299	33	56	695	32	55	1604	34	55	112	32	41
bank	1014	21	52	×	22	54	×	21	54	1312	22	54
hedc	934	33	61	×	34	62	×	34	62	602	31	60
w-4	83	21	49	313	22	49	301	21	51	36	20	49
w-8	1786	27	44	×	28	43	×	28	45	1240	28	43

Table 3.7: Heap Memory Consumed (in MB) for tbfs: Traditional BFS, lex: Lexical, and uni: Uniflow BFS implementations to traverse cuts of ranks up to 32. ×= out-of-memory error

partition, and UniNR for the implementation that does not regenerate them, and uses the original vector clocks.

Table 3.8 compares the runtimes and sizes of the JVM heap for traversing the lattice of consistent cuts for the benchmarks with UniR and UniNR implementations. Note that the heap space usage of the two uniflow based traversals remains more or less same. This is because the measured heap size includes the size of all the objects allocated from it, including the original computation and its state information, and not just the uniflow chain partition and its vector clocks. In addition, the memory footprint of regenerated vector clocks for the benchmarks computations is a few kilobytes at most. Thus, we do not see a considerable saving in heap memory usage with to the UniNR implementation. However, if the computation has a really large number of events, and its resulting uniflow chain partition also has a many more chains than the number of processes then we may see a big saving in memory

Name	Traditional BFS		UniR BFS		UniNR BFS	
	Space	Time	Space	Time	Space	Time
d-100	108	0.48	31	0.37	31	0.33
d-300	842	16.84	33	46.20	33	26.62
d-500	893	108.07	34	607.55	33	301.72
bank	×	×	59	73.24	58	87.97
hedc	×	×	56	1129.35	56	1304.74
w-4	258	0.99	25	19.59	25	21.48
w-8	×	×	40	1445.57	40	1880.45

Table 3.8: Heap-space consumed (in MB) and runtimes (in seconds) for traversing the full lattice of consistent cuts using traditional BFS, UniR: uniflow BFS that regenerates vector clocks, and UniNR: uniflow BFS that does not regenerate vector clocks.

usage with the UniNR implementation. Note that the for many cases UniNR implementation is significantly faster than our earlier implementation, and is relatively close to traditional BFS in runtimes.

Name	$r = \frac{\|E\|}{4}$		$r = \frac{\|E\|}{2}$		$r = \frac{3\|E\|}{4}$		$r \leq 32$	
	UniR	UniNR	UniR	UniNR	UniR	UniNR	UniR	UniNR
d-100	0.06	0.04	0.05	0.01	0.04	0.05	0.16	0.12
d-300	0.05	0.04	0.07	0.07	0.13	0.12	0.20	0.14
d-500	0.11	0.09	0.33	0.34	1.48	1.61	0.16	0.15
bank	0.27	0.25	3.07	2.97	3.12	0.31	5.28	5.88
hedc	0.50	0.41	15.70	16.61	0.51	0.52	0.15	0.09
w-4	0.07	0.06	0.10	0.10	0.09	0.09	0.05	0.01
w-8	0.72	0.71	11.27	12.69	3.09	3.08	0.05	0.02

Table 3.9: Runtimes (in seconds) to traverse cuts of given ranks with UniR and UniNR implementations

Table 3.9 shows the runtimes of the two implementations in traversing consistent cuts of specific ranks. For some cases, the improvement in runtime performance with UniNR over the UniR is considerable. Note that UniR is already significantly faster than Lex and traditional BFS for traversing specific ranks of the lattice.

Chapter 4

Detecting Stable and Counting Predicates

In this chapter, we give an algorithm to enumerate all consistent cuts satisfying a stable predicate. In addition, we define a new category of global predicates called *counting predicates* and give an algorithm to enumerate all consistent cuts that satisfy it.

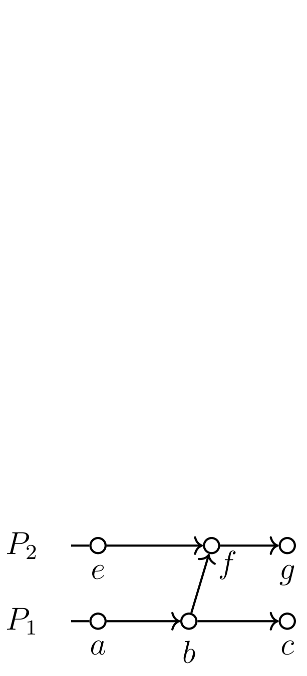
In the previous chapter, we focused on enumerating all consistent cuts of the computation lattice. In many practical scenarios, we may only be interested in a subset of consistent cuts that satisfy a given predicate. Moreover, knowledge about the properties and structure of this predicate can be helpful in enumerating the states that satisfy it. We first focus on a subclass of global predicates called *stable predicates*. Predicates such as $B =$ *all promises have been delivered*, and $B =$ *at least k events have been executed* fall under the category of stable predicates. In this chapter, we introduce another category of global predicates called *counting predicates*. This category encodes many useful conditions for debugging/verification of parallel programs. For example, $B =$ *exactly two messages have been received* is a counting predicate.

If we are interested in enumerating all the consistent cuts of a computation that satisfy a global predicate B that is of the type stable or counting, then we currently only have one choice: traverse all the cuts using existing traversal algorithms (such as BFS, DFS, and Lex) and enumerate each visited cut that satisfies B. This is 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 shown in Figure 4.1a, and the predicate B = *at least* 4 *events have been executed*. Figure 4.1b shows all the consistent cuts of the computation as a distributive lattice. There are five cuts in which at least four events have been executed; we have highlighted these cuts with a gray background. The BFS, DFS, or Lex traversal algorithms, however, will have to visit all the twelve cuts to find these five.

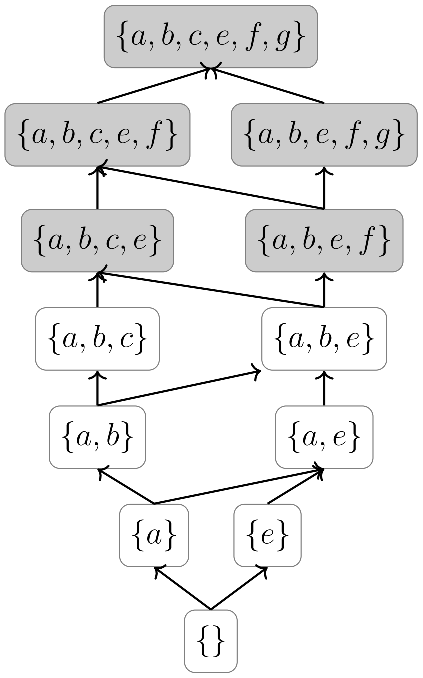
We now present algorithms to efficiently enumerate subset of consistent cuts that satisfy stable or counting predicates without enumerating other consistent cuts that do not satisfy them. Our algorithms take time and space that is a polynomial function of the number consistent cuts of interest, and in doing so provide an exponential reduction in time complexities in comparison to existing algorithms. For the earlier example of the computation Figure 4.1a, and the predicate B = *at least* 4 *events have been executed*, our algorithm only visits and enumerates the five gray cuts in Figure 4.1b.

4.1 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



(a) Computation



(b) Lattice of Consistent Cuts

Figure 4.1: A computation and its lattice of consistent cuts. Cuts with gray background satisfy predicate $B =$ *at least* 4 *events have been executed.*

least k events have been executed, and at least k' messages have been sent.

Definition 13 (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 4.2 presents a visualization for this concept.

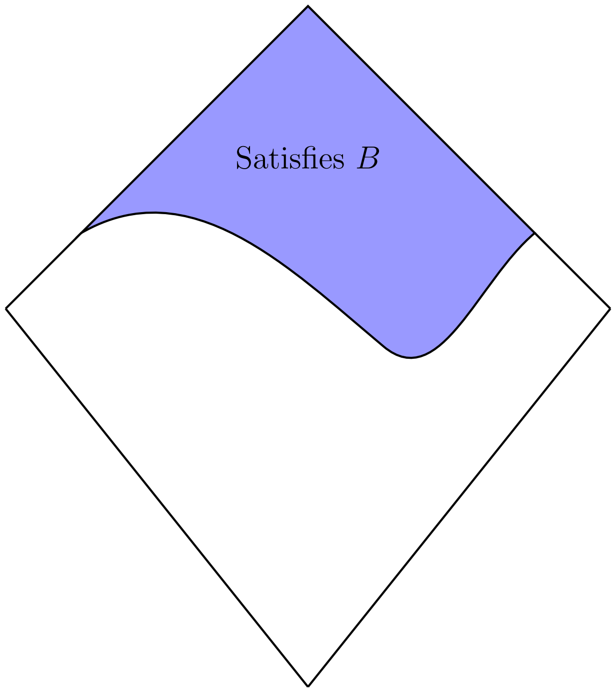


Figure 4.2: Illustration: Visual representation for some stable predicate B: the cuts in the blue region of the lattice satisfy a stable predicate, and cuts in the white region do not.

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 [10] 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 =$(E, \to)$. 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 12) 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 9 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.

Algorithm 9 EnumerateStable$((E, \to), B)$

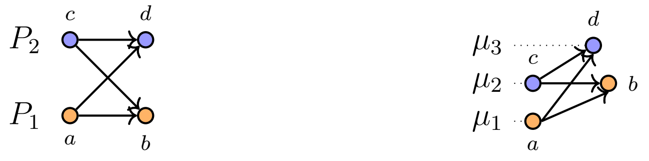
```
Input: Computation $(E, \to)$ in its uniflow chain partition $\mu$, $B$: a stable predicate
Output: Enumerate all consistent cuts satisfying $B$.
1: $G$ = GetMinCut$(B, \{\})$ // find the lexically smallest consistent cut satisfying $B$
2: while $G \neq$ null do
3:     enumerate$(G)$ // enumerate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

Algorithm 10 GetMinCut(B, G)

```
Input: $B$: a stable predicate, $G$: a consistent cut
Output: lexically smallest consistent cut $>_l G$ that satisfies $B$
1: $\langle H, c \rangle$ = 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 10 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 11 takes a consistent cut, G, and returns a pair: the first entry is the lexically smallest l-base cut (Definition 8) 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 $\langle H, c \rangle$, 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.



(a) Original Computation (b) Uniflow Partition

Figure 4.3: A computation on two processes in: (a) its original non-uniflow partition, (b) equivalent uniflow partition

Let us illustrate the execution with an example. Consider the computation in Figure 4.3b 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 $\langle H = \{a, b, c, d\}, i = 3\rangle$.

Algorithm 11 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 $(i = 1; i \leq n_u; i = i + 1)$ do // go from lowest chain to highest
3:   $j$ = index of the smallest event on chain $\mu_i$ that is not included in $H$
4:   for $(; j \leq size(\mu_i); j = j + 1)$ do // use events on chain $i$ not included in $G$
5:     $H = H \cup \{\mu_i[j]\}$ // 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 $\langle H, i\rangle$ // return $H$ and chain number of the event
9: return ⟨null,-1⟩ // no cut lexically greater than $G$ and satisfying $B$ was found
```

The BackwardPass routine (in Algorithm 12) 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: $\{G' \subseteq G : H[j] = G[j], start + 1 \leq j \leq n_u\}$.

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 4.3b and the predicate B=P_2 *has executed two or more events*, and the call BACKWARD-PASS $(B, 2, \{a, b, c, d\})$. 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 causual 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.

For the computation in Figure 4.3b 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

Algorithm 12 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 ($j = start; j \geq 1; j = j - 1$) do // iterate from start argument chain to lower chains
3:    for ($e = H[j]; e \geq 1; e = e - 1$) do // from last event on chain to first
4:      $K = H \setminus \{\mu_j[e]\}$ // 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 $H$
```

($B, \{\}$) 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 4.3b 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 9) 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.

4.1.1 Proof of Correctness

Lemma 6. *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. $\square$

If $B(G)$ is true, and G does not include all the events from the lowest chain μ_1, then the smallest l-base cut lexically greater than G satisfying B can be formed by adding the first event from μ_1 that is not included in G. If G includes all the events from μ_1 but not all from μ_2, then we can find the desired l-base cut by adding the first non-included event from μ_2, and so on (moving up chains). The steps of GetBiggerBaseCut encode this process.

Lemma 7. *Let G be a consistent cut of a computation $P = (E, \rightarrow)$, and B be a stable predicate. Then the cut H returned in $\langle H, i \rangle =$*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=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 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 8. *Let H be a consistent cut that satisfies a stable predicate B. Then H' =*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 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 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 $j = \mathit{start}$, and we call the routine with $\mathit{start} = i-1$. Hence, for each higher chain $j \geq i$, 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, $H' - \{e\}$ 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 $W \geq_l H'$. 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. $\square$

Lemma 9. *Let G be a consistent cut of a computation $P = (E, \rightarrow)$, and B be a stable predicate. Then* GetMinCut (B, G) *returns the lexically smallest consistent cut that is lexically greater than G and satisfies B.*

Proof. Let K be the cut returned by GetMinCut (B, G). If we have $K = \mathsf{null}$ (ie. $\{\}$), then by Lemma 6 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 $\langle H, i\rangle =$ GetBiggerBaseCut (B, G), then by definition of GetMinCut, we know that $K =$BackwardPass $(B, i-1, H)$. Then by Lemma 7, and Lemma 8 we know that K is consistent and satisfies B.

From 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 $K =$ 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 $K >_l W >_l G$. Recall that i is the chain number returned in $\langle H, i\rangle = \textsc{GetBiggerBaseCut}$ (B, G). We claim that $W[j] = H[j] = G[j]$, for $i < j \le n_u$. This claim is valid since we have already established earlier that that $K[j] = G[j]$ for $i < j \le n_u$. 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 $W >_l G$, we are guaranteed that $W[i] \ge G[i]$. There are only two possible cases:

Case (1): $W[i] \ge G[i] \wedge W[i] < K[i]$. Naturally, this will ensure that $W <_l K[i]$. Then, we form a consistent cut W' by setting $W'[j] = W[j]$ for $i \le j \le n_u$, and $W'[j] = size(\mu_j)$ for $1 \le j \le i-1$. 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 7.

Case (2): $W[i] > G[i] \wedge W[i] = K[i]$. Since $K[i] = H[i]$, we now have $W[j] = H[j]$ for $i \le j \le n_u$. 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 $i \le j \le n_u$. This contradicts Lemma 8.

□

Lemma 10. *For a computation P =$(E, \rightarrow)$, and a stable predicate B,* Enu-

MERATESTABLE in Algorithm 9 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 9 we know that in the result of $G =$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. □

4.2 Enumerating Consistent Cuts satisfying Counting 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 *counting predicates.* Counting predicates are used in multiple debugging and analysis applications. For example, while debugging an implementation of Paxos [45] 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 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:

Definition 14 ($view(G, c)$)**.** *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 4.4. 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 $view(G, blue) = \{d, f\}$.

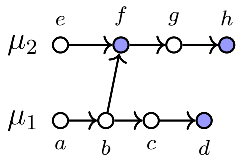


Figure 4.4: A computation in uniflow partition

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

Definition 15 (Counting Predicate). *Let $P = (E, \to)$ be a computation, and c be a color from the set of colors C. A predicate B is called a counting 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 counting 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 counting predicate B. We use the computation's uniflow partition μ for enumerating these cuts in their lexical order. Algorithm 13 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 10 to find the lexically smallest cut that satisfies B. This works because the lexically smallest cut that satisfies the counting 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.

EnumSameViewCuts in Algorithm 14 takes two arguments: a counting predicate B, and a consistent cut G that satisfies B. It uses the uniflow

Algorithm 13 ENUMERATECOUNTING$((E, \rightarrow), B)$

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

```
1: $G$ = GETMINCUT($B$, {}) // now $G$ is the smallest cut satisfying $B$
2: while $G \neq$ null do
3:     ENUMSAMEVIEWCUTS($B, G$)
4:     $G$ = GETSUCCESSOR($B, G$)
```

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 4.4. 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 15 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 4.4, given $G = \{a, b, c, d, e, f\}$, we have GETSUCCESSOR

Algorithm 14 EnumSameViewCuts(B, G)

Input: B: a counting 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 $(i = 1; i \leq n_u; i = i + 1)$ do // go from lowest chain to highest
5:     $j$ = index of the first event on chain $\mu_i$ that is not included in $H$
6:     for $(; j \leq size(\mu_i); j = j + 1)$ 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
```

$(B, G) = \{a, b, e, f, g, h\}$. 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 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

Algorithm 15 GetSuccessor(B, G)

```
Input: $B$: a counting 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 $(i = 1; i \leq n_u; i{+}{+})$ do  // lower chains to higher
5:   $ind$ = index of the first event on chain $\mu_i$ that is not included in $K$
6:   for $(; ind \leq size(\mu_i); ind = ind + 1)$ 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 causual dependencies on all lower chains
12:      for $(j = i + 1; j \leq n_u; j + +)$ do
13:        for $(k = i - 1; k > 0; k - -)$ do
14:          $S$ = causal dependencies of events from chain $\mu_j$ 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
```

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 4.5 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).

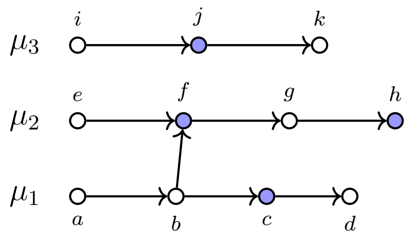


Figure 4.5: A computation in uniflow partition

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 $\{c, j\}$.

4.2.1 Proof of Correctness

Lemma 11. *Let G be a consistent cut of a computation $P = (E, \rightarrow)$, and B be a counting predicate. Then* GetMinCut (B, G) *in Algorithm 10 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 counting predicate that is of the form: $count_B(G) = k$. We construct a stable predicate B' from B by setting: $B' = count_B(G) > (k-1)$. The lexically smallest cut that satisfies the counting 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 9. $\square$

Lemma 12. *For a computation $P = (E, \rightarrow)$, and a counting predicate B, let G be a consistent cut of P that satisfies B. Then,* EnumSameViewCuts *in Algorithm 14 enumerates all consistent cuts of P that are lexically greater than G, satisfy B, and have same view as that of G.*

Proof. Algorithm 14 enumerates a cut H only if $view(G) == view(H)$; line 1 enumerates G itself, and the only other line that enumerates a cut is line 10 that is executed only if $view(G) == view(H)$.

We now show that the algorithm does not miss any consistent cut of P that satisfies B and has the same *view* as that of G. We know that G is already enumerated. Suppose $W >_l G$ is a consistent cut that satisfies B and $view(W) == view(G)$ and is not enumerated by the algorithm. Thus, there exists an event e on some chain i, $1 \leq i \leq n_u$, that is not included in G, and $G \cup \{e\}$ is consistent and $view(G) == view(G + \{e\})$. 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 $H >_l G \wedge view(H) == view(G)$. Hence, it is impossible that iterating through all the chains, we did not find e to construct W and subsequently enumerate it. □

Lemma 13. *Let G be any consistent cut of computation P =$(E, \rightarrow)$, that satisfies a counting predicate B Then* GetSuccessor *(B, G) returns the lexically smallest consistent cut greater than G that satisfies B and has a different 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 μ_i are already included in G, or on inclusion of the next event in μ_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 $count_B(K)$

is at most r, then by Lemma 9 we know that GetMinCut(B, K) returns the smallest lexical consistent cut greater than G that satisfies B. If for any consistent cut K, $count_B(K)$ is greater than r, then by the properties of counting predicates, there is no consistent cut of $count_B$ 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 *view*(G). □

Lemma 14. *Given a computation P =$(E, \rightarrow)$ with its uniflow chain partition μ, and a counting predicate B Algorithm 13 enumerates all consistent cuts of P that satisfy B.*

Proof. At line 1 in Algorithm 13 we find the lexically smallest consistent cut G that satisfies B. If its not null we pass it to EnumSameViewCuts that will enumerate it at its first line. By Lemma 12, we know that all subsequent cuts satisfying B with the *view*(G) will be enumerated. After this, the only consistent cuts that satisfy B 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, satisfies B, 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. □

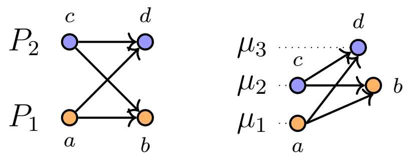
4.3 Optimized Implementation

We now 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 4.7a is not in uniflow partition. Figure 4.7b 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 Section 3.3.1. 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 4.6, 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 using the optimized approach of Section 3.2.4 in $\mathcal{O}(n_u + n^2)$ time.



(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 4.6: Illustration: Maintaining indicator vector G_u for a cut G

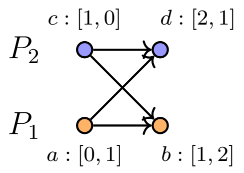
4.3.1 GetBiggerBaseCut

In the GetBiggerBaseCut routine we add events to any cut in increasing uniflow order (Definition 7). 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.

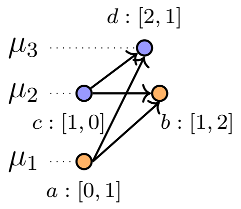
The computation P =$(E, \rightarrow)$ on n processes has $|E|$ events, and each event has a vector clock of length n. We first collect and store all the events in the uniflow order. Let J represent the array that stores the vector clocks of events in their increasing uniflow order. Now, for $2 \leq i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



(a) Computation



(b) Uniflow Partition

$d : [2,1]$
$b : [1,2]$
$c : [1,0]$
$a : [0,1]$

(c) Events in Uniflow Order

$J[4] = [2,2]$
$J[3] = [1,2]$
$J[2] = [1,1]$
$J[1] = [0,1]$

(d) J Vector

Figure 4.7: Illustration: Computing J vector for optimizing GetBiggerBaseCut

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 4.7a that has four events, and its uniflow partition in Figure 4.7b. The increasing order on the vector clocks of all the four events is in Figure 4.7c. Starting from the bottom (vector $[0,1]$), and performing the joins, we get J as shown in Figure 4.7d. 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.

4.3.2 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 12 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.

4.3.3 GetSuccessor

We optimize the routine GetSuccessor by replicating the strategy of computing projections as per Section 3.3.3. 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 Section 3.3.3 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 $\mathcal{O}((\log|E| + n_u \log m) \cdot n)$.

4.4 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 counting predicate for P.

From Section 4.3.1, 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. From Section 4.3.2, we know that optimized version of BackwardPass takes $\mathcal{O}(n_u \cdot n^2 \cdot \log m)$ 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

Algorithm	Space Required
Traditional BFS	$\mathcal{O}(\frac{m^{n-1}}{n})$
DFS	$\mathcal{O}(\lvert E\rvert)$
Lex	$\mathcal{O}(n)$
Optimized Uniflow-BFS*	$\mathcal{O}((n_u + \lvert E\rvert) \cdot n)$

Table 4.1: Space complexities of algorithms for detecting a stable or counting predicate in the lattice of consistent cuts; here $m = \frac{|E|}{n}$. * denotes algorithm in this dissertation.

Algorithm	Time
Traditional BFS	$\mathcal{O}(n^2 \cdot \lvert\mathcal{C}(E)\rvert)$
DFS	$\mathcal{O}(n^2 \cdot \lvert\mathcal{C}(E)\rvert)$
Lex	$\mathcal{O}(n^2 \cdot \lvert\mathcal{C}(E)\rvert)$
Optimized Uniflow-BFS*	$\mathcal{O}(n \cdot \lvert S_B\rvert \cdot (n_u \cdot n \cdot \log m + \log \lvert E\rvert))$

Table 4.2: Time complexities for enumerating all consistent cuts of $\mathcal{C}(E)$ that satisfy a stable predicate B. * denotes algorithm in this dissertation.

$\mathcal{O}((n_u \cdot n \cdot \log m + \log |E|) \cdot n)$ 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 4.1, we compare the worst-case space complexities of our optimized algorithm against those of BFS, DFS, and Lex algorithms, to detect a predicate that is either of type stable or counting.

Let $S_B \in \mathcal{C}(E)$ denote the set of consistent cuts that satisfy the stable predicate B. Then, Table 4.2 compares the worst-case time complexities of these algorithms to enumerate all consistent cuts in S_B. Note that the $|\mathcal{C}(E)|$ can be exponentially bigger than $|S_B|$.

We now move on to computation slicing, and in the next chapter present a distributed algorithm for slicing with respect to regular predicates.

Chapter 5

Distributed Online Algorithm for Slicing

In this chapter, we give a distributed online algorithm to compute slice of a computation with respect to a state based regular predicate.

A computation slice of a computation with respect to a predicate B is a concise representation of all the consistent cuts of the computation that satisfy the predicate B. When the predicate B is regular, the set of consistent cuts satisfying B, $\mathcal{C}_B(E)$, forms a sublattice of $\mathcal{C}(E)$ that is the lattice of all consistent cuts of the computation. $\mathcal{C}_B(E)$ can equivalently be represented using its join-irreducible elements [24]. Intuitively, join-irreducible elements form the basis of a lattice, such that the lattice can be generated by taking joins of its basis elements. Let J_B be the set of all join-irreducible elements of $\mathcal{C}_B(E)$, and let $J_B(e)$ denote the least consistent cut that includes an event e and satisfies predicate B. Then, it has been shown [35] that

$$J_B = \{J_B(e) \mid e \in E\}$$

Remark 1. *Observe that for an event e, $J_B(e)$ may not necessarily exist because there may not be any consistent cut that includes e and satisfies B. Also, multiple events may have the same $J_B(e)$.*

For the predicate $B =$ "*all channels are empty*", the $J_B(event)$ values for each event of the computation in Figure 5.1 are: $J_B(a) = \{a\}$, $J_B(b) = \{a, b, e, f\}$, $J_B(c) = \{a, b, c, e, f\}$, $J_B(e) = \{e\}$, $J_B(f) = \{a, b, e, f\}$, $J_B(g) = \{a, b, e, f, g\}$. In the vector clock notation of consistent cuts, these cuts can be representation as: $J_B(a) = [0, 1]$, $J_B(b) = [2, 2]$, $J_B(c) = [2, 3]$, $J_B(e) = [1, 0]$, $J_B(f) = [2, 2]$, $J_B(g) = [3, 2]$.

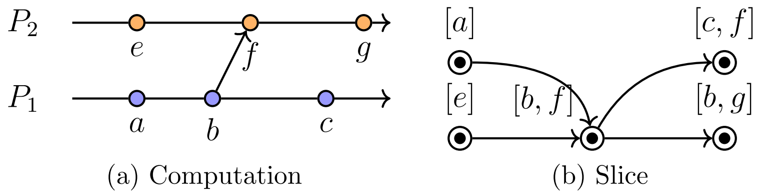


Figure 5.1: A Computation, and its slice with respect to predicate $B =$"all channels are empty"

Intuitively, a join-irreducible element of a lattice is one that cannot be represented as the join of two distinct elements of the lattice, both different from itself. For the computation of Figure 5.1, the join-irreducible consistent cuts are: $\{a\}, \{a, b\}, \{a, b, c\}, \{e\}, \{a, b, e, f\}, \{a, b, e, f, g\}$. Figure 5.2 shows the join-irreducible consistent cuts of the sub-lattice induced by predicate "all channels empty" for computation of Figure 5.1.

A centralized online algorithm to compute J_B was proposed in [51]. In the online version of this centralized algorithm, a pre-identified process, called *slicer* process, plays the role of the slice computing process. All the processes in the system send their event and local state values whenever their local states change. The *slicer* process maintains a queue of events for each

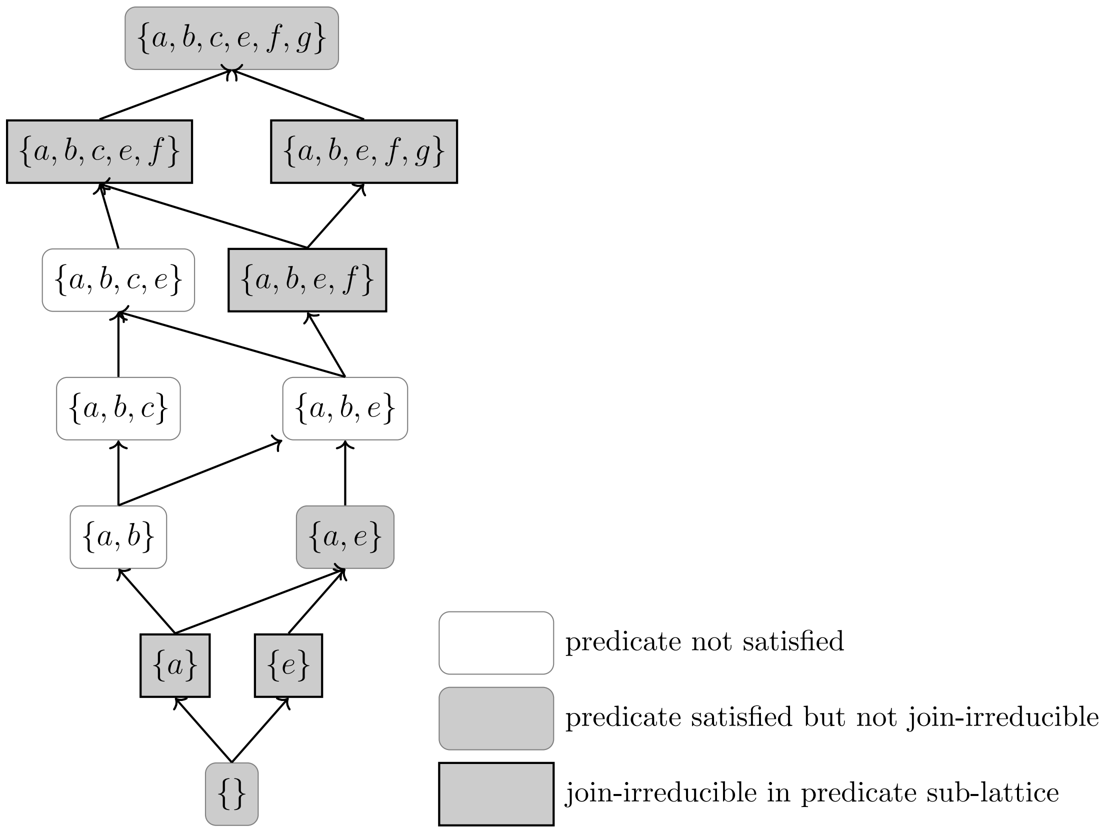


Figure 5.2: Illustration: Join-irreducible elements of the lattice of consistent cuts for Figure 5.1 with respect to predicate B = "*all channels are empty*".

process in the system, and on receiving the data from a process adds the event to the relevant queue. In addition, the slicer process also keeps a map of events and corresponding local states for each process in the system. For each received event, the slicer appends the event and local state mapping to the respective map. For every event e it receives, the slicer computes $J_B(e)$ using the *linearity property.* This centralized algorithm, however, suffers from drawbacks that apply to almost most centralized algorithms: they are not fault tolerant, and push all the message and computational load to one processes and thus scale poorly.

Online algorithms for detecting certain classes of predicates, such as stable, termination and conjunctive predicates, have been proposed (*cf.*, [32]). Using the equivalence result described in [51], these algorithms can also be used to derive online slicing algorithms for those predicates. However, in the resultant slicing algorithms, the incremental cost of updating the slice on arrival of a new event is quite high due to the generic nature of the transformation.

Distributed algorithms for monitoring a program execution have been proposed previously [62, 5]. The algorithm in [62], however, can only detect a subset of safety properties, whereas the algorithm in [5] requires the underlying system to be synchronous.

We propose a distributed algorithm that significantly reduces the computational load, as well as the message load on any process. For our distributed slicing algorithm, we require that message channels between processes impose first-in-first-out (FIFO) order. In our distributed online slicing algorithm, we have n slicer processes (running as local threads on application processes), $S_1, S_2, ..., S_n$, one for every application process $P_1, ..., P_n$. For a computation $P = (E, \to)$, E_i denotes the set of events executed by process P_i. All slicer processes cooperate to compute the task of slicing $(E, \to)$. In our algorithm, S_i computes

$$J_i(B) = \{J_B(e) | e \in E_i\}$$

where $J_B(e)$ is the join-irreducible consistent cut that satisfies B and includes event e. Observe that by the definition of join-irreducible consistent cut, $e \rightarrow f$ implies $J_B(e) \subseteq J_B(f)$. Since all events in a process are totally ordered, the set of consistent cuts generated by any S_i is also totally ordered.

Algorithm 16 presents the distributed algorithm for online slicing with respect to a regular predicate B. Each slicer process has a token assigned to it that goes around in the system. Other slicer processes cooperate in maintaining and processing the token. The goal of the token for the slicer process S_i is to compute $J_B(e)$ for all events $e \in E_i$. Whenever the token has computed $J_B(e)$ it returns to its original process, reports $J_B(e)$ and starts computing $J_B(succ(e))$, $succ(e)$ being the immediate successor of event e. The token T_i carries with it the following data:

- *pid*: Process id of the slicer process to which it belongs.
- *event*: Details of event e, specifically the event id and event's vector clock, at P_i for which this token is computing $J_B(e)$. The identifier for event e is the tuple $\langle pid, eid \rangle$ that identifies each event in the computation uniquely.
- *gcut*: The vector clock corresponding to the cut which is under consideration (a candidate for $J_B(e)$).
- *depend*: Dependency vector for events in *gcut*. The dependency vector is updated each time the information of an event is added to the token

(steps explained later), and is used to decide whether or not some cut being considered is consistent. On any token, its vector *gcut* is a consistent global state *iff* for all i, $depend[i] \leq gcut[i]$.

- *gstate*: Vector representation of global state corresponding to vector *gcut*. It is sufficient to keep only the states relevant to the predicate B.
- *eval*: Evaluation of B on *gstate*. The evaluation is either true or false; in our notation we use the values: $\{predtrue, predfalse\}$.
- *target*: A pointer to the unique event in the computation for which a token has to wait. The event need not belong to the local process.

A token waits at a slicer process P_i under three specific conditions:

(C1) The token is for process S_i and it has computed $J_B(pred(e))$, $pred(e)$ being the immediate predecessor event of e, and is waiting for the arrival of e.

(C2) The token is for process S_i and it is computing $J_B(f)$, where f is an event on P_i prior to e. The computation of $J_B(f)$ requires the token to advance along process P_i.

(C3) The token is for process S_j such that $j \neq i$, and it is computing $J_B(f)$ which requires the token to advance along process P_i.

On occurrence of each relevant event $e \in E_i$, the computation process P_i performs a *local* enqueue to *slicer* S_i, with the details of this event. Note

Algorithm 16 Distributed Slicing Algorithm at S_i

```
Input:  1. An ongoing computation; each event $e \in E_i$ reported to $S_i$
        2. Regular predicate $B$
Output: Online slice of computation with respect to $B$
function ReceiveEvent(Event $e$, State $localstate_e$)
    save $\langle e.eid, localstate_e \rangle$ in local state map $procstates$
    for each waiting token $t$ at $S_i$ do
        if ($t.target = e$) then // $t$ waiting for event $e$
            AddEventToToken($t$,$e$)
            ProcessToken($t$)
function AddEventToToken(Token $t$, Event $e$)
    $t.gstate[e.pid] = procState[e.eid]$
    $t.gcut[e.pid] = e.eid$
    if $t.pid == i$ then // my token: update token's $event$ pointer
        $t.event = e$
    $t.depend = max(t.depend, e.V)$ // set causal dependency
function ProcessToken(Token $t$)
    if $t.gcut$ is $inconsistent$ then
        // find lowest $k$ for which $t.gcut[k] < t.depend[k]$
        $t.target = t.gcut[k] + 1$ // set desired event
        send $t$ to $S_k$
    else EvaluateToken($t$) // $t.gcut$ is consistent
function EvaluateToken(Token $t$)
    if $B(t.gstate)$ then // $B$ is true on cut given by $t.gcut$
        $t.eval = predtrue$
        send $t$ to process $S_{t.pid}$
    else // $B$ is false on $t.gstate$
        $t.eval = predfalse$
        // $P_k$: forbidden process in $t.gstate$ for $B$
        $t.target = t.gcut[k] + 1$
        send $t$ to $S_k$
```

that P_i and its slicer S_i are modeled as two threads on the same process, and therefore the *local* enqueue is simply an insertion into the queue — that is shared between the threads on the same process — of the *slicer*. The inserted information contains the event identifier $\langle pid, eid \rangle$, the corresponding vector

Algorithm 17 Continued: Distributed Slicing Algorithm at S_i

```
28: function ReceiveToken(Token $t$)
29:   if $(t.eval == predtrue) \wedge (t.pid == i)$ then // my token, $B$ true
30:     output$(t.pid, t.eid, t.gcut)$
31:      // token waits for the next event
32:     $t.target = t.gcut[i] + 1$
33:     $t.waiting = true$
34:   else // either inconsistent cut, or predicate false
35:     $newid = t.target$ // id of event $t$ requires
36:     if $\exists f \in localEvents : f.id == newid$ then // required event has happened
37:       AddEventToToken($t$,$f$)
38:       EvaluateToken($t$)
39:      // else, the token remains in waiting state
40: function ReceiveStopSignal
41:   for each token $t : t.pid \neq i$ do
42:      // not my token, send back to parent
43:     send $t$ to $S_{t.pid}$
```

clock $e.V$, and P_i's local state $localstate_e$ corresponding to e. We now explain each function of the algorithm in detail:

ReceiveEvent (Lines 1–6): On receiving the details of event e from P_i, S_i adds them in the mapping of P_i's local states *procstates* (line 2). It then iterates over all the waiting tokens, and checks their *target*. For each token that has e as the target (required event to make progress), S_i updates the state of the token, and then processes it.

AddEventToToken (Lines 7–12): To update the state of some token t on S_i, we advance the candidate cut to include the new event by setting $t.gcut[i]$ to the id of event e. If S_i is the parent process of the token (T_i), then the $t.event$ pointer is updated to indicate the event id for which token is computing the join–irreducible cut that satisfies the predicate. The causal-dependency is updated at line 12, which is required for checking whether or not the cut is consistent.

ProcessToken (Lines 13–18): To process any token, S_i first checks that the global state in the token is consistent (line 14) and at least beyond the global states that were earlier evaluated to be false. For t's evaluation of a global cut $t.gcut$ to be consistent, $t.gcut$ must be at least $t.depend$. This is verified by checking the component-wise values in both these vectors. If some index k is found where $t.depend > t.gcut$, the token's cut is inconsistent, and $t.gcut$ must be advanced by at least one event on P_k, by sending the token to *slicer* of P_k. If the cut is consistent, the predicate is evaluated on the variables stored as part of $t.gstate$ by calling the EvaluateToken routine.

EvaluateToken (Lines 19–27): The cut represented by $t.gstate$ is evaluated; if the predicate is true, then the token has computed $J_B(e)$ for the event $e = \langle t.pid, t.eid \rangle$. The token is then sent to its parent *slicer*. If the evaluation of the predicate on the cut is false, the $target$ pointer is updated, at line 26, and the token is sent to the *forbidden* process on which the token must make progress.

ReceiveToken (Lines 28–39 in Algorithm 17): On receiving a token, the *slicer* checks if the predicate evaluation on the token is true, and the token is owned by the *slicer*. In such a case, the *slicer* outputs the cut information, and now uses the token to find $J_B(succ(e))$, where $succ(e)$ denotes the event that locally succeeds e. This is done by setting the new event id in $t.target$ at line 32, and then setting the waiting flag (line 33). If the predicate evaluation on the token is false, then the $target$ pointer of the token points to the event required by the token to make progress. S_i looks for such an event (line 36), and if it has been reported to S_i by P_i, then adds that event (line 37) to the token and processes it (line 38). In case the desired event has not been reported yet to the *slicer* process, the token is retained at the process S_i and is kept in the waiting state until the required event arrives. Upon arrival of the required event, its details are added to the token and the token is processed.
Note: The notation of $target = t.gcut[i] + 1$ means that if the $t.gcut[i]$ holds the event id $\langle pid, eid\rangle$, then the $target$ pointer is set to $\langle pid, eid + 1\rangle$.

ReceiveStopSignal (Lines 40-43 in Algorithm 17): For finite computations, a single token based termination detection algorithm is used in tandem. When termination is detected, a pre-determined *slicer* sends the 'stop' signal to all the *slicer* processes, including itself. On receiving the 'stop' signal, S_i sends all the *slicing* tokens that do not belong to it back to their parent processes.

Note that the functions in our algorithm require atomic updates and reads on the local queues, as well as on tokens present at S_i. These atomic updates can be easily implemented using common local synchronization techniques.

5.1 Example of Algorithm Execution

This example illustrates the algorithm execution steps for one possible run (real time observations) of the computation shown in Figure 5.1, with respect to the predicate $B =$ "all channels empty".

The algorithm starts with two slicing processes S_1 and S_2, each with token T_1 and T_2 respectively. The *target* pointer for each token T_i is initialized to the event $\langle i, 1\rangle$. When event a is reported, S_1 adds its details to T_1, and on its evaluation finds the predicate "all channels empty" to be true, and outputs this information. It then updates $T_1.target$ pointer and waits for the next event to arrive. Similar steps are performed by S_2 on T_2 when e is reported.

When b is reported to S_1, and T_1 is evaluated with the updated information, the predicate is false on the state $[b]$. Given that b is a message send event, it is obvious that for the channel to be empty, the message receive event should also be incorporated. Thus, S_1 sends T_1 to S_2 after setting the target pointer to the first event on S_2. On receiving T_1, S_2 fetches the information of its first event (e) and updates T_1. The subsequent evaluation still leads to the predicate being false. Thus S_2 retains T_1 and waits for the next event.

When f is reported, S_2 updates both T_1 and T_2 with f's details. S_2's evaluation on $T_1.gstate$, represented by $[b, f]$ is true, and as per line 22, T_1 is sent back to S_1 where the consistent cut $[b, f]$ is output. T_1 now waits for the next event. However, after being updated with the details of event f, the resulting cut on T_2 is inconsistent, as the message-receive information is present but the information regarding the corresponding send event is missing. By using the vector clock values, T_2's target would be set to the id of message-send event b. S_2 would then send T_2 to S_1. On receiving T_2, S_1 finds the required event (looking at $T_2.target$) and after updating T_2 with its details, evaluates the token. The predicate is true on $T_2.gstate$ now, and T_2 is sent back to S_2. On receiving T_2, S_2 outputs the consistent cut $[b, f]$, and waits for the next event. On receiving details of event c, and adding them to the waiting token T_1, the predicate is found to be true again on T_1, and S_1 outputs $[c, f]$. Similarly on receiving g, S_2 performs similar steps and outputs $[b, g]$. Note that the consistent cuts $[a, b]$ and $[c, g]$, both of which satisfy the predicate are not enumerated as they are not join-irreducible, and can be constructed by the unions of $[a]$, $[b]$ and $[c, f]$, $[b, g]$ respectively.

5.2 Proof of Correctness

We now prove the correctness and termination of the distributed algorithm of Algorithm 16 for finite computations. The correctness argument can be easily extended to infinite computations.

Lemma 15. *The algorithm presented in Algorithm 16 does not deadlock.*

Proof. The algorithm involves n tokens, and none of the tokens wait for any other token to complete any task. With non-lossy channels, and no failing processes, the tokens are never lost. The progress of any token depends on the *target* event, and as per lines 4–6, whenever an event is reported to a *slicer*, it always updates the tokens with their *target* being this event. Thus, the algorithm can not lead to deadlocks. □

Lemma 16. *If a token T_i is evaluating $J_B(e)$ for $e \in E_i$, assuming $J_B(e)$ exists, and if $T_i.gcut < J_B(e)$, then $T_i.gcut$ would be advanced in finite time.*

Proof. If during the computation of $J_B(e)$, at any instance $T_i.gcut < J_B(e)$, then there are two possibilities for *gcut*:

(a) *gcut* is consistent: This means that the evaluation of predicate B on *gcut* must be false, as by definition $J_B(e)$ is the least consistent cut that satisfies B and includes e. In this case, by line 26 and subsequent steps, the token would be forced to advance on some process.

(b) *gcut* is inconsistent: The token is advanced on some process by execution of lines 14–17. □

Lemma 17. *While evaluating $J_B(e)$ for event $e \in E_i$ on token T_i, if $T_i.gcut < J_B(e)$ currently and $J_B(e)$ exists then the algorithm eventually outputs $J_B(e)$.*

Proof. By Lemma 16, the global cut of T_i would be advanced in finite time. Given that $J_B(e)$ exists, we know that by the linearity property, there must exist a process on which T_i should progress its *gcut* and *gstate* vectors in order to reach the $J_B(e)$; lines 26–27 ensure that this forbidden process is found and T_i sent to this process. By the previous Lemma, the cut on the T_i would be advanced until it matches $J_B(e)$. By line 30 of the algorithm, whenever $J_B(e)$ is reached, it would be output. □

Lemma 18. *For any token T_i, the algorithm never advances $T_i.gcut$ vector beyond $J_B(e)$ on any process, when searching $J_B(e)$ for $e \in E_i$.*

Proof. The search for $J_B(e)$ starts with either an empty global state vector, or from the global state that is at least $J_B(pred(e))$, where $pred(e)$ is the immediate predecessor event of e on S_i. Thus, till $J_B(e)$ is reached, the global cut under consideration is always less than $J_B(e)$. From the linearity property of advancing on the forbidden process, and Lemma 16, the cut would be advanced in finite time. Whenever the cut reaches $J_B(e)$, it would be output as per Lemma 17 and the token would be sent back to its parent slicer, to either begin the search for $succ(e)$ or to wait for $succ(e)$ to arrive ($succ(e)$ being the immediate successor of e). Thus, $T_i.gcut$ would never advance beyond $J_B(e)$ on any process when searching for $J_B(e)$ for any event e. □

Lemma 19. *If token T_i is currently not at S_i, then T_i would return to S_i in finite time.*

Proof. Assume T_i is currently at S_j ($j \neq i$). S_j would advance $T_i.gcut$ in finite time as per Lemma 16. With no deadlocks (Lemma 15), and by Lemmas 17 and 18, we are guaranteed that if $J_B(T_i.event)$ exists then within a finite time,

$T_i.gcut$ vector would be advanced to $J_B(T_i.event)$ and T_i would be sent back to S_i. If $J_B(T_i.event)$ does not exist then at least one slicer process S_k would run out of all its events while attempting to advance on $T_i.gcut$. In such a case, knowing that there are no more events to process, S_k would send T_i back to S_i (lines 40-43). □

Lemma 20. (Termination)*: For a finite computation, the algorithm terminates in finite time.*

Proof. We first prove that for any event $e \in E_i$, computation of finding $J_B(e)$ with token T_i takes finite time. By Lemma 16, T_i always advances in finite time while computing $J_B(e)$. If $J_B(e)$ exists, then based on this observation within a finite time the token T_i would advance its *gcut* to $J_B(e)$, if it exists. By Lemma 17, the algorithm would output this cut, thus finishing the $J_B(e)$ search and as per Lemma 18 would not advance any further for $J_B(e)$ computation. Thus, if $J_B(e)$ exists then it would be output in finite time. By Lemma 19 the token would be returned to its parent process and the $J_B(e)$ computation for $e \in E_i$ would finish in finite time.

If $J_B(e)$ does not exist, then as we argued in Lemma 19 some *slicer* would run out of events to process in the finite computation, and thus return the token to S_i, which would result in search for $J_B(e)$ computation to terminate. As each of these steps is also guaranteed to finish in finite time as per above Lemmas, we conclude that $J_B(e)$ computation for $e \in E_i$ finishes in finite time.

Applying this result to all the events in E leads to the desired result of termination in finite time. $\square$

Lemma 21. *The algorithm outputs all the elements of J_B.*

Proof. Whenever any event $e \in E$ occurs, it is reported by some process P_i on which it occurs, to the corresponding slicer process S_i. Thus e can be represented as $e \in E_i$. If at the time e is reported to S_i, T_i is held by S_i then by Lemmas 16 and 17, it is guaranteed that the algorithm would output $J_B(e)$. If S_i does not hold the token T_i when e is reported to it, then by Lemma 19, T_i would arrive on S_i within finite time. If S_i has any other events in its processing queue before e, then as per Lemma 20, S_i would finish those computations in finite time too. Thus, within a finite time, the computation for finding $J_B(e)$ with T_i would eventually be started by S_i. Once this computation is started, the results of Lemmas 16 and 17 can be applied again to guarantee that the algorithm would output $J_B(e)$, if it exists.

Repeatedly applying this result to all the events in E, we are guaranteed that the algorithm would output $J_B(e)$ for every event $e \in E$. Thus the algorithm outputs all the join-irreducible elements of the computation, which by definition together form J_B. $\square$

Lemma 22. *The algorithm only outputs join-irreducible global states that satisfy predicate B.*

Proof. By Lemma 18, while performing computations for $e \in E_i$ on token T_i, the algorithm would not advance on token T_i beyond $J_B(e)$. Since only token

T_i is responsible for computing $J_B(e)$ for all the events $e \in E_i$, the algorithm would not advance beyond $J_B(e)$ on any token. In order to output a global state that is not join-irreducible we must advance the cut of at least one token beyond a least global state that satisfies B. The result follows from the above assertions. $\square$

Lemma 20 guarantees termination, and correctness follows from Lemmas 21, and 22.

5.3 Complexity Analysis

Each token T_i processes every event $e \in E_i$ once for computing its $J_B(e)$. If there are $|E|$ events in the system, then in the worst case T_i does $\mathcal{O}(n \cdot |E|)$ work, because it takes $\mathcal{O}(n)$ to process one event. We are assuming here that evaluation of B takes $\mathcal{O}(n)$ time given a global state. There are n tokens in the system, hence the total work performed is $\mathcal{O}(n^2 \cdot |E|)$. Since there are n slicing processes and n tokens, the average work performed is $\mathcal{O}(n \cdot |E|)$ per process. In comparison, the centralized algorithm (either online or offline) requires the *slicer* process to perform $\mathcal{O}(n^2 \cdot |E|)$ work.

Let $|S|$ be the maximum number of bits required to represent a local state of a process. The actual value of $|S|$ is subject to the predicate under consideration, as the resulting number/type of the variables to capture the necessary information for predicate detection depends on the predicate. The centralized online algorithm requires $\mathcal{O}(|E| \cdot |S|)$ space in the worst case;

however it is important to notice that all of this space is required on a single (central slicer) process. For a large computation, this space requirement can be limiting. The distributed algorithm proposed above only consumes $\mathcal{O}(|E_i| \cdot |S|)$ space per slicer. Thus, we have a reduction of $\mathcal{O}(n)$ in per slicer space consumption.

The token can move at most once per event. Hence, in the worst case the message complexity is $\mathcal{O}(|E|)$ per token. Therefore, the message complexity of the distributed algorithm presented here is $\mathcal{O}(n \cdot |E|)$ total for all tokens. The message complexity of the centralized online slicing algorithm is $\mathcal{O}(|E|)$ because all the event details are sent to one (central) slicing process. However, for conjunctive predicates, it can be observed that the message complexity of the *stalling*-based implementation of the distributed algorithm is also $\mathcal{O}(|E|)$. With speculative stalling of tokens, only unique join-irreducible cuts are computed. This means that for conjunctive predicates, a token only leaves (and returns to) S_i, $\mathcal{O}(|E_i|)$ times. As there are n tokens, the overall message complexity of the *stalling*-based implementation for conjunctive predicates is $\mathcal{O}(|E|)$.

Algorithm	Total Work	Work/Slicer	Messages	Space/Slicer
Centralized	$\mathcal{O}(n^2 \cdot |E|)$	$\mathcal{O}(n^2 \cdot |E|))$	$\mathcal{O}(|E|)$	$\mathcal{O}(|E| \cdot |S|)$
Distributed (this chapter)	$\mathcal{O}(n^2 \cdot |E|)$	$\mathcal{O}(n \cdot |E|)$	$\mathcal{O}(n \cdot |E|)$	$\mathcal{O}(|E_i| \cdot |S|)$

Table 5.1: Comparison of Centralized and Distributed Online Slicing Algorithms

In the next chapter, we present algorithms to create slices of two temporal logic operators with respect to predictaes that are not regular.

Chapter 6

Slicing for Non-Regular Predicates

Computation slicing is an abstraction technique for efficiently finding all global states of computation that satisfy a given global predicate, without explicitly enumerating all such global states [51]. The *slice* of a computation with respect to a predicate is a sub-computation that satisfies the following properties: (a) it contains all global states of the computation for which the predicate evaluates to true, and (b) of all the sub-computations that satisfy condition (a), it has the least number of global states. The slice has much fewer global states than the computation itself — exponentially smaller in many cases — resulting in substantial savings. Multiple algorithms [51, 55, 58] have been presented for computing the slice for temporal logic predicates where B is a regular state-based predicate. In many scenarios, however, the predicate B is not regular. Note that if B is not regular, then their temporal versions $\mathsf{AG}(B)$, $\mathsf{EG}(B)$, and $\mathsf{EF}(B)$ may not be regular. In this chapter, we present offline algorithms for computing slices for $\mathsf{AG}(B)$, and $\mathsf{EF}(B)$ when B is not a a regular predicate but either $\neg B$ or B itself is efficiently detectable, and we are given the slice of the computation with respect to B.

6.1 Slicing Algorithm for $\mathsf{AG}(B)$

For a predicate B, we say a consistent cut C satisfies the temporal logic predicate $\mathsf{AG}(B)$ iff in the lattice of consistent cuts, all cuts reachable from C satisfy B. That is: $C \models \mathsf{AG}(B)$ iff for *all* consistent cut sequences $C_0, \ldots, C_k$ such that (i) $C_0 = C$, and (ii) $C_k = \widehat{E}$, we have: $C_i \models B$ for *all* $0 \leq i \leq k$. Thus,

When predicate B is not regular, we require that $\neg B$ be efficiently detectable for efficient computation of the slice with respect to $\mathsf{AG}(B)$. Algorithm 18 shows the algorithm for computing the slice for $\mathsf{AG}(B)$ when this condition is met.

Recall that a slice of a computation is its equivalent directed graph containing additional directed edges. Hence, the directed graph for the slice will have at least as many edges as in original computation. When constructing a slice for a computation $G = \langle E, \mapsto \rangle$, the slice contains two types of directed edges:

1. all the edges of G,
2. edges added by the slicing algorithm.

In constructing the slice for $\mathsf{AG}(B)$, edges of type (2) eliminate consistent cuts of the original computation that do not satisfy $\mathsf{AG}(B)$. To do so, we consider all possible pairs of events (e, f) such that $e \not\mapsto f$ in G. Let A be the set of consistent cuts of G that contain f but not e. Adding an edge from e

to f eliminates those and only those consistent cuts of G that are in A. To determine if such an edge can be added, we need to ascertain that no consistent cut in A satisfies $\mathsf{AG}(B)$. Let C be the largest consistent cut of A. Note that C exists and is well defined; and every other consistent cut of A can reach C. It is sufficient to check that C satisfies $\mathsf{AG}(B)$. If C does not satisfy $\mathsf{AG}(B)$ then there exists a consistent cut D in G such that $C \subseteq D$ and D does not satisfy B. Such a consistent cut will be reachable from every other consistent cut of A as well. As a result, none of the consistent cuts of A will satisfy $\mathsf{AG}(B)$. On the other hand, if C does satisfy $\mathsf{AG}(B)$, then clearly the slice of the computation w.r.t. $\mathsf{AG}(B)$ must contain C. Hence, the slice cannot contain an edge from e to f because that will eliminate the cut C.

Algorithm 18 Slicing algorithm for $\mathsf{AG}(B)$ when B is not regular.

```
Input: (1) computation graph $G = \langle E, \mapsto \rangle$, (2) predicate $B$ such that $\neg B$ is
    efficiently detectable
Output: the slice of $\langle E, \mapsto \rangle$ with respect to $\mathsf{AG}(B)$
1: $M = G$
2: for each event pair $(e, f)$ such that $e \not\mapsto f$ in $\langle E, \mapsto \rangle$ do
3:     // Find $C$ and $H$ as follows:
4:    $C$: largest consistent cut of $G$ that contains $f$ but not $e$
5:    $H$: a reduced computation of $G$ such that $C$ is the initial consistent cut of $H$
6:    if some cut in $H$ satisfies $\neg B$ then
7:       add $e \mapsto f$ in $M$
8: return $M$
```

We now explain how to efficiently check whether or not C satisfies $\mathsf{AG}(B)$. This is done by starting from the largest consistent cut C that contains event f but not e. If starting from this cut, anywhere in the future — in the remaining computation — we detect that B is not satisfied for some cut (hence

$\neg B$ is true) then we are guaranteed that $\mathsf{AG}(B)$ cannot hold for C.

6.1.1 Proof of Correctness

Let S be a graph that represents an actual slice of G with respect to $\mathsf{AG}(B)$. We show that M returned by Algorithm 18 is same as S. As our algorithm only adds edges to the original computational graph, we only need to show the following:

Lemma 23. *Each edge of M is also an edge of S, and vice-versa.*

Proof. Note that M and S both must contain all the edges of the original computation. Hence, if they differ in their edges, it must be due to the type (2) edges: the additional edges added to construct the slice. Let E_M and E_S be the set of edges of type (2) in M and S respectively. There are the following two possibilities.

(a) $\exists$ edge $e \mapsto f \in E_M : e \not\mapsto f \in E_S$: Hence, our algorithm added the edge $e \mapsto f$ to eliminate all the consistent cuts that contain f but not e when constructing the slice. Observe that our algorithm only adds such an edge after ensuring that no consistent cut that contains f but not e can satisfy $\mathsf{AG}(B)$ in the G. Let us construct a graph S' by adding this edge $e \mapsto f$ to S. Note that S' still contains all the consistent cuts of G that satisfy $\mathsf{AG}(B)$ but will form a smaller sub-lattice than that of S. This leads to a contradiction of S being a slice of G with respect of $\mathsf{AG}(B)$ as it violates the definition of slice.

(b) $\exists$ edge $e \mapsto f \in E_S : e \not\mapsto f \in E_M$: As this edge is of type (2) — an edge that is not originally present in G, but added to construct a slice — we know that $e \not\mapsto f$ in G. Hence, our algorithm must have considered the event pair e, f (at line 2) and subsequently checked if there exists a largest consistent cut containing f but not e such that it satisfies $\mathsf{AG}(B)$. But, by having the edge $e \mapsto f$ the graph S will not contain at least one consistent cut of G that satisfies $\mathsf{AG}(B)$. Hence, we again have a contradiction that S is a slice of G with respect to $\mathsf{AG}(B)$. $\square$

6.1.2 Complexity Analysis

Suppose the complexity of detecting $\neg B$ is $\mathcal{O}(T)$ where $T = g(n, |E|)$, and g is a polynomial. Then, we have the following result.

Theorem 3. *The time complexity of the algorithm in Algorithm 18 is* $\mathcal{O}(\max(T.|E|^2, |E|^3))$.

Proof. There are $\mathcal{O}(|E|^2)$ possible pairs of events e, and f (line 2). Finding the largest consistent cut C (at line 4) takes $\mathcal{O}(|E|)$ time. Detecting $\neg B$ (line 6) in the computation is $\mathcal{O}(T)$. Hence, the overall time complexity is $\mathcal{O}(\max(T.|E|^2, |E|^3))$. $\square$

6.2 Slicing Algorithm for $\mathsf{EF}(B)$

A consistent cut C satisfies the temporal logic predicate $\mathsf{EF}(B)$ iff in the lattice of consistent cuts, there exists some consistent cut $C' \supseteq C$ that satisfies B, and we can reach C' by starting with the cut C and then executing *some* sequence of events on the way. That is: $C \models \mathsf{AG}(B)$ iff for *some* consistent cut sequences $C_0, \ldots, C_k$ such that (i) $C_0 = C$, and (ii) $C_k = \widehat{E}$, we have: $C_i \models B$ for *some* $0 \leq i \leq k$. We now present an algorithm to compute the slice for $\mathsf{EF}(B)$ when predicate B is not regular. Algorithm 19 shows the steps of our algorithm. Note that the slice is efficiently computable only if B is efficiently detectable. Let W denote the greatest (final) consistent cut of the input slice $\langle E, \mapsto\rangle_B$. In the algorithm, we construct a graph H with vertices as the vertices in the original computation, G, and the following edges:

1. all the edges in G, and
2. from $\top$ to the successors of events in *frontier*(W).

The first type of edges ensure that the consistent cuts of H are a subset of the consistent cuts of G. The second type of edges ensure that the final consistent cut of H is W, therefore all consistent cuts of G that can reach W are consistent cuts of H. Note that when B is not regular, the slice $\langle E, \mapsto\rangle_B$ may not be lean — some of its consistent cuts may not satisfy B. Hence, it is possible that W may not satisfy the predicate B. From the definition of $\mathsf{EF}(B)$, all consistent cuts of the computation that can reach some consistent cut that satisfies B will also satisfy $\mathsf{EF}(B)$ and furthermore these are the only cuts that satisfy $\mathsf{EF}(B)$. However, the definition of slice requires that it is a lattice, and hence W must be the join-closure of all the consistent cuts of $\langle E, \mapsto\rangle_B$ that satisfy B. It can be shown that any consistent cut C less than equal to W can be written as join of $C_1 \ldots C_m$ such that C_i satisfies $\mathsf{EF}(B)$. Hence W is the largest consistent cut that is reachable from every consistent cut that satisfies B in the computation. We can find the cut W using slice $\langle E, \mapsto\rangle_B$ when it is nonempty. We construct the slice for $\mathsf{EF}(B)$ from the computation so that the slice contains all consistent cuts of the computation that can reach W. To ensure that all cuts that cannot reach W do not belong to the slice, we add edges from $\top$ to the successors of events in the frontier of W in the computation. Note that adding an edge from $\top$ to an event makes any cut that contains the event trivial.

Algorithm 19 Slicing algorithm for $\mathsf{EF}(B)$ when B is not regular.

Input:
(1) computation $G = \langle E, \mapsto\rangle$, (2) predicate B such that B is efficiently detectable
(3) slice of $\langle E, \mapsto\rangle$ with respect to B, denoted by $\langle E, \mapsto\rangle_B$

Output: slice of $\langle E, \mapsto\rangle$ with respect to $\mathsf{EF}(B)$

```
1: $H = G$
2: if slice $\langle E, \mapsto\rangle_B$ is non-empty then
3:     $W$ = the final consistent cut of slice $\langle E, \mapsto\rangle_B$
4:     $\forall e \in \mathit{frontier}(W)$: add an edge from the vertex $\top$ to $\mathit{succ}(e)$ in $H$
5: else // $H$ becomes an empty slice
6:     add an edge from $\top$ to $\bot$ in $H$
7: return $H$
```

6.2.1 Proof of Correctness

Lemma 24. *Every consistent cut of $G = \langle E, \mapsto\rangle$ that satisfies $\mathsf{EF}(B)$ is a consistent cut of H.*

Proof. Consider a consistent cut C of $G = \langle E, \mapsto\rangle$ that satisfies $\mathsf{EF}(B)$. In this case, slice $\langle E, \mapsto\rangle_B$ is nonempty. Observe that when $\langle E, \mapsto\rangle_B$ is non-empty, by construction, H contains only those consistent cuts of the computation that can reach W. Since C satisfies $\mathsf{EF}(B)$, there exist a cut $D \supseteq C$ such that D satisfies B. Since W is join-closure of all the maximal cuts that satisfy B in G, $D \subseteq W$. This implies that $C \subseteq W$, and based on the earlier observation C must be a consistent cut of H. $\square$

Lemma 25. *Every consistent cut of H either satisfies $\mathsf{EF}(B)$ or is a join of some set of cuts such that all of them satisfy $\mathsf{EF}(B)$.*

Proof. Let C be a consistent cut of H. Then, either $C \models B$, or C does not satisfy B. $C \models B \Rightarrow C \models \mathsf{EF}(B)$. We now show that if C does not satisfy B, then it is join-closure of some consistent cuts that satisfy $\mathsf{EF}(B)$. Note that W is the largest consistent cut of slice $\langle E, \mapsto\rangle_B$, and by our algorithm W is included in H. Hence, $C \subseteq W$. Since W is the largest consistent cut in slice with respect to B, W can be written as: $W = W_1 \cup W_2 \cup \ldots W_m$ where $W_i \models B$ as every slice is a join-closed lattice. As $C \subseteq W$, we can re-write C as:

$$\begin{aligned} & C = C \cap W \\ \equiv\ & C = C \cap (W_1 \cup W_2 ... \cup W_m) \\ \equiv\ & C = (C \cap W_1) \cup (C \cup W_2) ... \cup (C \cup W_m). \end{aligned}$$

Let us define $C_i = C \cap W_i$. Then, we have: $C_i \subseteq W_i$, and $W_i \models B$. Therefore, $C_i \models \mathsf{EF}(B)$. Since all C_i's are in H, then C must also be in H because it is union of C_is. $\square$

6.2.2 Complexity Analysis

Note that the slice with respect to B is required as an input to our algorithm. When B is not regular, computation of this slice may not be efficient itself. We now analyze the complexity of the algorithm in Algorithm 19 once this slice has been computed. The graph H produced by our algorithm has $\mathcal{O}(|E|)$ vertices, and $\mathcal{O}(|E| + n)$ edges, and can be built in $\mathcal{O}(n|E|)$ time. The slice with respect to a predicate contains $\mathcal{O}(n|E|)$ edges using the skeletal representation. The non-emptiness check at line 2 can be done by checking whether the number of strongly connected components of the input slice is greater than one, which takes $\mathcal{O}(n|E|)$ time. We can compute the final consistent cut of this slice, that is W, by proceeding backwards from vertex $\top$ as follows: first, we compute the strongly connected component of the slice that contains $\top$, in $\mathcal{O}(n|E|)$ time. Second, for each process P_i, starting from the final event on P_i, we find the predecessors of events until we reach events on P_i that do not belong to the strongly connected component. This step takes $\mathcal{O}(|E_i|)$ time. Hence, we can compute the frontier of W in $\mathcal{O}(|E|)$ time across all the processes. There are n successor events to the events in frontier of W, requiring $\mathcal{O}(n)$ time to add edges from $\top$ to these successor events. Thus the algorithm has $\mathcal{O}(n|E|)$ overall time-complexity.

This chapter ends the presentation of algorithmic contributions of this dissertation. In the next chapter, we present concluding remarks and future work.

Chapter 7

Conclusion and Future Work

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, breadth-first-search based traversal of global states of parallel programs is a crucial routine. We have reduced the space complexity of this routine exponentially. This reduction in space complexity allows us to analyze computation with high degree of parallelism with relatively small memory footprint. Moreover, our BFS based enumeration algorithm (Algorithm 2) lends itself well to parallel implementations with minimal effort. This is because it traverses cuts of rank $r+1$ independently of those of rank r. We can perform a parallel traversal easily using a parallel-for loop at line 3 of Algorithm 2. It is an interesting future problem to implement this parallel approach and compare its performance against parallel traversal algorithms such as Paramount [12].

Our algorithm for detecting counting 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.

Our algorithm for enumerating all consistent cuts that satisfy a stable predicate has applications not only in predicate detection, but also in analysis of parallel computations. 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 9) 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 counting predicate and returns the set of consistent cuts satisfying it. Our algorithm also applies to solving instance of stable marriage [30, 37] problem with constraints on the minimum or maximum regret. Optimization of our algorithm for efficiently solving such stable marriage instances is another future direction of research.

Our algorithms on computation slicing are useful for online global predicate detection. Suppose the predicate B is of the form $B_1 \wedge B_2$, where B_1 is regular but B_2 is not, and we are interested in monitoring the system online to check if any possible global state satisfies B during its execution. Cooper and Marzullo's widely used online algorithm [23] traverses the lattice of global states while remaining oblivious to the nature of the predicate. It will check all possible states of the computation, and can be quite expensive in terms of both time and space. In contrast, instead of searching for the global state that satisfies B in the original computation, with our distributed slicing algorithm we can search the global states in the slice for B_1. Thus, running our algorithm together with Cooper and Marzullo's algorithm, the space and time complexity of predicate detection is reduced significantly (possibly exponentially) for predicates in the above mentioned form. Our distributed online slicing algorithm has been adopted by others for detecting *general* temporal logic formulas for runtime verification [52, 7]. However, detection performed by these works is sound but not *complete*. An important future problem is to extend our slicing algorithms to develop techniques that guarantee both soundness and completion.

Our distributed slicing algorithm is also useful for recovery of distributed programs based on checkpointing. For fault-tolerance, we may want to restore a distributed computation to a checkpoint which satisfies the required properties such as "all channels are empty", and "all processes are in some states that have been saved on storage". If we compute the slice of the computation in an online fashion, then on a fault, processes can restore the global state that corresponds to the maximum of the last vector of the slice at each surviving process. This global state is consistent as well as recoverable from the storage.

To conclude, we have presented multiple algorithms that provide exponential savings in either space or time — in many cases both — for the task of detecting predicates in parallel computations. These algorithms are not only limited to the field of predicate detection, and can also be applied to solve problems in the fields of performance analysis, check-pointing, stable marriage, and lattice theory.

Bibliography

[1] R. Alagappan, A. Ganesan, Y. Patel, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Correlated crash vulnerabilities. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 151–167, GA, 2016. USENIX Association.

[2] S. Alagar and S. Venkatesan. Hierarchy in Testing Distributed Programs. In *Proceedings of the International Workshop on Automated Debugging (AADEBUG)*, pages 101–116, 1993.

[3] S. Alagar and S. Venkatesan. Techniques to Tackle State Explosion in Global Predicate Detection. In *IEEE Transactions on Software Engineering*, pages 412–417, Dec. 1994.

[4] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer. Preemption sealing for efficient concurrency testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 420–434. Springer, 2010.

[5] A. Bauer and Y. Falcone. Decentralised LTL Monitoring. In *Proceedings of the 18th International Symposium on Formal Methods*, pages 85–100, Paris, France, Aug. 2012.

[6] L. Bianco, P. Dell Olmo, and S. Giordani. An optimal algorithm to find the jump number of partially ordered sets. *Computational Optimization and Applications*, 8(2):197–210, 1997.

[7] B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, D. A. Rosenblueth, and C. Travers. Decentralized asynchronous crash-resilient runtime verification. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 59. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[8] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. J. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *OPODIS*, pages 83–97, 2013.

[9] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[10] Y. Chang and V. 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.

[11] Y.-J. Chang. *Predicate Detection for Parallel Computations.* PhD thesis, UT Austin, Austin, TX, 2016.

[12] Y.-J. Chang and V. K. Garg. A parallel algorithm for global states enumeration in concurrent systems. In *Proceedings of the 20th ACM SIG-*

PLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pages 140–149. ACM, 2015.

[13] H. Chauhan and V. K. Garg. Detecting stable and counting predicates in parallel computations. *Under review*, 2017.

[14] H. Chauhan and V. K. Garg. Space efficient breadth-first and level traversals of consistent global states of parallel programs. *To appear in Proceedings of the 17th International Conference on Runtime Verification (RV 2017)*, 2017.

[15] H. Chauhan, V. K. Garg, A. Natarajan, and N. 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.

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

[17] F. Chen, T. F. Serbanuta, and G. Roşu. jPredictor: a predictive runtime analysis tool for java. In *Proceedings of the International Conference on Software Engineering*, pages 221–230, 2008.

[18] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proceedings of the Workshop on Logics of Programs*, Yorktown Heights, New York, May 1981.

[19] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986.

[20] E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 294–303, New York, NY, USA, 1987. ACM.

[21] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2000.

[22] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

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

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

[25] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, volume 47, pages 257–266, 2012.

[26] 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, Feb. 1988.

[27] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[28] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[29] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 328–343, New York, NY, USA, 2017. ACM.

[30] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

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

[32] V. K. Garg. *Elements of Distributed Computing.* John Wiley and Sons, Incorporated, New York, NY, 2002.

[33] V. 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.

[34] V. K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.

[35] V. K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, Phoenix, Arizona, USA, Apr. 2001.

[36] V. 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.

[37] D. Gusfield and R. W. Irving. *The stable marriage problem: structure and algorithms*. MIT press, 1989.

[38] M. Habib, R. Medina, L. Nourine, and G. Steiner. Efficient algorithms on distributive lattices. *Discrete Appl. Math.*, 110(2-3):169–187, 2001.

[39] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[40] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

[41] W.-L. Hung, H. Chauhan, and V. K. Garg. Brief announcement: Non-blocking monitor executions for increased parallelism. In *28th International Symposium on Distributed Computing (DISC)*, pages 553–554, 2014.

[42] W.-L. Hung, H. Chauhan, and V. K. Garg. Activemonitor: Asynchronous monitor framework for scalability and multi-object synchronization. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 46. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[43] R. Jegou, R. Medina, and L. 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.

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

[45] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[46] Y. Lei and R. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6):382–403, 2006.

[47] S. Lu, J. Tucek, F. Qin, and Y. 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.

[48] 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.

[49] N. Mittal and V. K. Garg. On Detecting Global Predicates in Distributed Computations. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, Phoenix, Arizona, USA, Apr. 2001.

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

[51] 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, Dec. 2007.

[52] M. Mostafa and B. Bonakdarpour. Decentralized runtime verification of ltl specifications in distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 494–503. IEEE, 2015.

[53] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of Conference on Programming language design and implementation*, pages 446–455, 2007.

[54] A. Natarajan, H. Chauhan, N. Mittal, and V. K. Garg. Efficient abstraction algorithms for predicate detection. *Theoretical Computer Science*, 688:24 – 48, 2017. Distributed Computing and Networking.

[55] V. A. Ogale and V. K. Garg. Detecting temporal logic predicates on distributed computations. In *Proceedings of International Symposium in Distributed Computing*, pages 420–434, 2007.

[56] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks e ciently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA'99). World Scientific*, 1999.

[57] G. Pruesse and F. Ruskey. Gray codes from antimatroids. *Order 10*, pages 239–252, 1993.

[58] A. Sen and V. K. Garg. Detecting temporal logic predicates on the happened-before model. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[59] A. Sen and V. K. Garg. Automatic generation of computation slices for detecting temporal logic predicates. Technical Report TR-PDS-2003-001,

Department of Electrical and Computer Engineering, The University of Texas at Austin, 2003.

[60] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates in Distributed Programs using Computation Slicing. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 171–183, Dec. 2003.

[61] A. Sen and V. K. Garg. Formal Verification of Simulation Traces Using Computation Slicing. *IEEE Transactions on Computers*, 56(4):511–527, Apr. 2007.

[62] K. Sen, A. Vardhan, G. Agha, and G. Roşu. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 418–427, 2004.

[63] W. Song, T. Gkountouvas, K. Birman, Q. Chen, and Z. Xiao. The freeze-frame file system. In *ACM Symposium on Cloud Computing (SOCC)*, 2016.

[64] M. B. Squire. Enumerating the ideals of a poset. In *PhD Dissertation, Department of Computer Science, North Carolina State University*, 1995.

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

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

[67] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[68] C. von Praun and T. R. Gross. Object race detection. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

Vita

Himanshu Chauhan was born to Beena Chauhan and Devendra Singh Chauhan in Kanpur, India on 27 October 1984. He received the Bachelor of Technology degree in Chemical Engineering from the Indian Institute of Technology, Kanpur in 2005. He then worked as a software engineer for PricewaterhouseCoopers, and IBM Research Labs. He started graduate school at the University of Texas at Austin in August, 2011.

Permanent address: 463/6 Shastri Nagar
Kanpur, Utter Pradesh
India

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.