On Slicing a Distributed Computation

Vijay K. Garg*
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

Neeraj Mittal
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188, USA
neerajm@cs.utexas.edu

Abstract

We introduce the notion of a slice of a distributed computation. A slice of a distributed computation with respect to a global predicate is a computation which captures those and only those consistent cuts of the original computation which satisfy the global predicate. We show that a slice exists for a global predicate iff the predicate is a regular predicate. We then give an efficient algorithm for computing the slice and we show applications of slicing to testing and debugging of distributed programs.

1. Introduction

A distributed program is equivalent to the set of all distributed computations it generates. For many applications such as testing and debugging of distributed programs a programmer may be required to analyze a distributed computation. In this paper, we define the notion of a computation slice which we argue is a useful notion for these applications. Informally, given a distributed computation, a programmer may compute its slice with respect to different global predicates. Given a global predicate, the slice represents all possible manners in which the predicate can become true in the original computation.

The notion of a computation slice is analogous to the concept of a program slice [10, 17]. Given a program and a set of variables, a program slice consists of all statements in the program that may affect the value of the variables in the set at some given point. The notion of a slice has been also extended to distributed programs [5, 9]. Program slicing has been shown to be useful in program debugging, testing, program understanding and software maintenance [11, 17]. A slice can significantly narrow the size of the program to be analyzed, thereby making the understanding

of the program behaviour easier. We expect to reap the same benefit from a computation slice.

A computation slice differs from a dynamic program slice in that it is defined for a property rather than a set of variables of a program. Unlike a program slice, which always exists, a computation slice may not always exist. In this paper, we give the necessary and sufficient condition for a computation slice to exist. In particular, we prove that a slice of a computation with respect to a predicate exists iff the set of consistent cuts that satisfy the predicate forms a sublattice of the (original) lattice of consistent cuts. We call such predicates *regular*. The class of regular predicates is closed under conjunction. Some examples of regular predicates are any local predicate such as "process P_i is in *red* state" and many channel predicates [6] such as "there are at most k messages in transit from P_i to P_j". We also present an efficient algorithm to compute the slice for a regular predicate. Computation slicing can be used for monitoring predicates in distributed systems which has applications in testing and debugging, and fault-tolerance of distributed programs. For example, it can be used to observe a predicate under $possibly$ modality, if it is regular, and under $definitely$ modality [3] if its negation is a regular predicate.

This paper makes the following contributions.

- We first prove the set of consistent cuts of a computation forms a finite *distributive* lattice. We then show that this is a complete characterization of the set of consistent cuts, that is, given any arbitrary finite distributive lattice, there exists a distributed computation for which it represents the set of consistent cuts. The fact that the set of consistent cuts of a computation forms a lattice was independently observed by Johnson and Zwaenepoel [8] and Mattern [13]. To our knowledge, this paper is the first to show that the lattice is distributive and that it is a complete characterization.

- We define a class of global predicates called *regular* predicates. The set of consistent cuts that satisfy a reg-

ular predicate forms a sublattice of the lattice of consistent cuts, that is, if two consistent cuts satisfy a regular predicate then their set intersection and set union will also satisfy the predicate. We also show that if two predicates are regular then so is their conjunction.

- We define the notion of a computation slice given a computation E and a predicate B. The computation slice F is a distributed computation such that its consistent cuts are precisely those consistent cuts of E that satisfy B. We show that a computation slice is possible if and only if B is a regular predicate.

- We give an efficient algorithm for computing a computation slice. Our algorithm has the complexity $O(N^2|E|)$ where N is the number of processes and E is the set of events in the system.

- We give applications of computation slice to testing and debugging, and fault-tolerance of distributed programs.

The paper is organized as follows. We present our model of distributed system in Section 2. Section 3 gives the complete characterization of the execution lattice. In Section 4, we define the class of regular predicates. Section 5 introduces the notion of a computation slice. An efficient algorithm to compute the slice is presented in Section 6. Finally, we discuss some applications of our results in Section 7.

2. Model and background

We assume a loosely-coupled message-passing asynchronous system without any shared memory or a global clock. A distributed program consists of N processes denoted by $\{P_1, P_2, \ldots, P_N\}$ communicating via asynchronous messages. In this paper, we will be concerned with a single computation of a distributed program. We do not make any assumptions about FIFO nature of the channels.

The execution of a process in a computation can be viewed as a sequence of events with events across processes ordered by Lamport's happened-before relation [12]. The happened-before relation between two events e and f can be formally stated as: $e \rightarrow f$ iff e occurs before f in the same process, or e is a send of a message and f is a receive of that message, or there exists an event g such that e happened-before g and g happened-before f. We define a distributed computation as the partially ordered set (poset) consisting of the set of events together with the happened before relation and denote it by $(E, \rightarrow)$.

We define a consistent cut of a computation $(E, \rightarrow)$ as a subset $G \subseteq E$ such that

$$f \in G \wedge e \rightarrow f \;\Rightarrow\; e \in G$$

E	set of events
$\rightarrow$	happened-before relation
e, f, g	events
$\sqcup$	join of two elements in a poset
$\sqcap$	meet of two elements in a poset
G, H	consistent cuts
$C(E)$	lattice of consistent cuts of $(E, \rightarrow)$
L	a lattice
$J(L)$	set of join-irreducible elements of L
B	A global predicate
$(I_B, F, \rightarrow_B)$	computation slice of $(E, \rightarrow)$ with respect to B
$\parallel$	concurrency relation
N	number of processes

Figure 1. Notation

A consistent cut captures the notion of a possible global state. The concept of a consistent cut is identical to that of a *down-set (or an order ideal)* used in the lattice theory literature [4].

Given two consistent cuts, G and H, we say that $G \leq H$ iff $G \subseteq H$. It is well known in the lattice theory that the set of all down-sets (order ideals) forms a lattice under $\subseteq$ relation. This is equivalent to the result that the set of all consistent cuts forms a lattice under $\leq$ [8, 13].

For any poset, we use $\sqcup$ and $\sqcap$ to denote join and meet operators.

A *global predicate* (or simply a *predicate*) is a boolean-valued function defined on the set of consistent cuts. We say that $B(G)$ (B holds in the consistent cut G) if the function evaluates to true in the cut G.

Figure 1 summarizes the notation used in this paper.

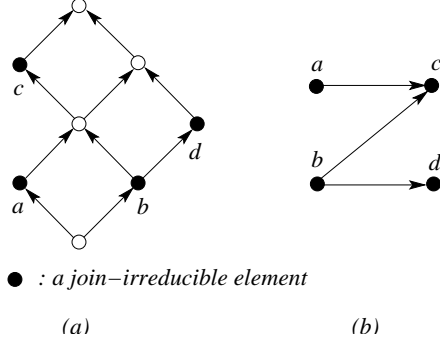## 3. A complete characterization of execution lattice

We first ask the question what additional properties does the lattice of consistent cuts satisfy? The answer to this question comes from a standard result in lattice theory [4].

Lemma 1 *Let $(X, \leq)$ be any poset and $C(X)$ be the set of all down-sets of X. Then, $(C(X), \subseteq)$ is a distributive lattice.*

From the above result we get

Theorem 2 *The set of consistent cuts of any distributed computation $(E, \rightarrow)$ forms a distributive lattice under the relation $\subseteq$.*

We denote the set of consistent cuts of any distributed computation $(E, \rightarrow)$ by $C(E)$ ($\rightarrow$ is implicit). The fact that $C(E)$ forms a lattice has been observed before [8, 13]. In this paper we observe that the lattice is distributive and

● : a join−irreducible element

(a)   (b)

Figure 2. An example of a distributive lattice (a) and its partial order representation (b).

exploit this observation to derive notion of a computation slice.

We next show that Theorem 2 is a complete characterization of $C(E)$ under the relation $\subseteq$. In other words, there is no additional structure property satisfied by this set. To prove this, given any finite distributive lattice L, it is sufficient to construct a distributed computation (a poset P) such that the set of consistent cuts (set of down-sets) is exactly L. To this end, first define *join-irreducible* elements as follows.

Definition 1 (Join-Irreducible Element) *An element* $x \in L$ *is join-irreducible if*

1. $x \neq 0$, and

2. $\forall a, b \in L : x = a \sqcup b \Rightarrow (x = a) \vee (x = b)$.

Here, 0 refers to the zero element of L. Pictorially, in a finite lattice, an element is join-irreducible iff it has exactly one lower cover, that is, it has exactly one incoming edge. Figure 2(a) shows a distributive lattice with its join-irreducible elements. Let $J(L)$ denote the set of join-irreducible elements in L. Now we can state Birkhoff's Theorem for finite distributive lattices.

Theorem 3 [Birkhoff's Representation Theorem for Finite Distributive Lattices] *Let L be a finite distributive lattice. Then the map $f : L \to C(J(L))$ defined by*

$$f(a) = \{x \in J(L) | x \leq a\}$$

is an isomorphism of L onto $C(J(L))$. Dually, let P be a finite poset. Then the map $g : P \to J(C(P))$ defined by

$$g(a) = \{x \in P | x \leq a\}$$

is an isomorphism of P onto $J(C(P))$.

The above theorem implies that there is a one-to-one correspondence between a finite poset and a finite distributive lattice. Given a finite poset, we get the finite distributive

lattice by considering its set of down-sets. Given a finite distributive lattice, we can recover the poset by focusing on its join-irreducible elements. Informally, any element of a lattice can be written as join of a subset of join-irreducible elements of the lattice. For example, Figure 2(b) gives the poset corresponding to the lattice in Figure 2(a).

From the above discussion it is clear that given any finite distributed computation, the structure *finite distributive lattice* completely characterizes its execution graph. We will see implication of this observation in later sections.

Birkhoff's theorem is also useful in computational sense because the set of join-irreducible elements of a lattice is generally exponentially smaller than the size of the lattice itself. In fact, for a finite distributive lattice, the number of join-irreducible elements is exactly equal to the size of the longest chain in the lattice [4]. In our case, the length of the longest chain is bounded by the number of events $|E|$. Hence if some computation on L can instead be done on $J(L)$, then we get a significant computational advantage.

4. Regular predicates

It is useful in a distributed computation to observe and control global predicates. There is a wide body of literature in observing [1, 3, 7] and controlling [15, 16, 14] global predicates. Since the number of consistent cuts is exponential in the number of processes in general, it is computationally hard to observe or control a general predicate [2]. However, when predicates satisfy additional properties, such as stability, observer-independence or linearity, then efficient algorithms can be devised to observe them. In this section we define a new class of predicates called regular predicates. Informally, a global predicate is a regular predicate if the set of consistent cuts satisfying the predicate forms a sublattice of the lattice of consistent cuts, that is, the set of consistent cuts that satisfy the predicate is closed under set intersection and set union. A regular predicate is also *linear* and hence easy to detect. In practice, most linear predicates are also regular predicates; therefore regular predicates form a useful class of predicates. We first define a regular predicate formally.

Definition 2 (Regular Predicate) *Let $C(E)$ be the set of consistent cuts of a computation $(E, \to)$. A predicate B is regular iff*

$$\forall\, G, H : G, H \in C(E) : B(G) \wedge B(H) \Rightarrow B(G \cap H) \wedge B(G \cup H)$$

Some examples of regular predicate are:

- Consider the predicate B as "there is no outstanding message in the channel". We show that this predicate is regular. Observe that B holds on a consistent cut G

iff for all send events in G, the corresponding receive events are also in G. It is easy to see that if $B(G)$ and $B(H)$, then $B(G \cup H)$. To see that it holds for $G \cap H$, let e be any send event in $G \cap H$. Let f be the receive event corresponding to e. From $B(G)$, we get that $f \in G$ and from $B(H)$, we get that $f \in H$. Thus, $f \in G \cap H$. Hence $B(G \cap H)$. Similarly, the following predicates are also regular.

  - There is no token message in transit.
  - There is no token message in transit between processes $P_1, ..., P_5$.
  - Every "request" message has been "acknowledged" in the system.

- Consider any local predicate. Informally, a predicate is local if its truth value depends only on the set of events executed locally.

  It can be shown that any local predicate is regular. Thus the following predicates are regular.

  - The leader has sent all "prepare to commit" messages.
  - Process P_i is in "red" state.

- Many monotonic channel predicates [6] such as "there are at most k messages in transit from P_i to P_j" and "there are at least k messages in transit from P_i to P_j" are in fact regular.

We now show that the class of regular predicates is closed under conjunction.

Lemma 4 *If B_1 and B_2 are regular predicates, then so is $B_1 \wedge B_2$.*

Proof: Given two consistent cuts G and H, let $K = G \cup H$. Since B_1 is a regular predicate $B_1(K)$ holds. Similarly, $B_2(K)$ holds. Hence $B_1 \wedge B_2$ holds for K. A similar proof can be given when $K = G \cap H$. ∎

The closure under conjunction implies that the following predicates are also regular.

- No process has the token and no channel has the token.

- Any conjunction of local predicates.

The closure is not true for disjunction as can be readily verified by taking B_1 and B_2 to be local predicates. Let $C_B(E)$ be the set of consistent cuts satisfying the predicate B. From the definition of regular predicates, it follows that the $C_B(E)$ is a sublattice of $C(E)$. From results in [2], it follows that a regular predicate is always linear. Therefore, the algorithm in [2] can be used to determine the least consistent cut that satisfies B.

5. Slicing a distributed computation

In this section, we define the notion of slice of a distributed computation with respect to a global predicate. Informally, a slice consists of a partial order defined on subsets of events which can be interpreted as the events in the subset must be executed together. In other words, a consistent cut either includes all events corresponding to a subset or none of them implying that to an external observer they *appear* to have been executed *atomically*—he cannot observe the intermediate states. This is similar to the concept of transactions in databases where operations in a transaction appear to have been executed atomically. Further, the set of down-sets (or consistent cuts) of the slice is identical to the set of those consistent cuts of the original computation that satisfy the given predicate. The slice of a computation, therefore, tells us which subsets of events have to be executed atomically, and what is the order in which they must be executed. We now define a slice formally.

Definition 3 (Slice) *Given any distributed computation $(E, \rightarrow)$ and a global predicate B, we call a triplet $(I_B, F, \rightarrow_B)$ (where $F \subseteq 2^E$, and $\rightarrow_B$ is a partial order on F) a* slice *if*

1. *F is a partition of some subset of E, and*

2. *$\forall \, G \in C(E) : B(G) \equiv (G \in C(F))$*

The first requirement on the slice is that its elements are mutually disjoint subsets of events in the computation. The second requirement states it captures all the consistent cuts that satisfy the predicate B. Thus if a consistent cut satisfies B, it is also a consistent cut of F and vice versa. Here, I_B denotes the initial consistent cut of the slice which is the least consistent cut in E that satisfies B.

As an example of a slice, consider the computation in Figure 3(a). The lattice corresponding to the computation is show in Figure 3(b). The label of each element in the lattice is a 3-tuple denoting the index of the maximal event on each process contained in the corresponding consistent cut. For example, the label for the consistent cut G is $(2, 1, 1)$. For the predicate "all channels are empty", the sublattice is shown in Figure 3(c) along with its join-irreducible elements. The slice of the computation with respect to the predicate is shown in Figure 3(d). It can be verified the set of consistent cuts of the slice are exactly those that satisfy the given predicate in the original computation. Further, the length of the longest chain of the sublattice is 4 which is exactly equal to the number of its join-irreducible elements. In this example, $I_B = \emptyset$ which may not be the case in general. Finally, in the slice, as expected, events e_1 and f_1 and events f_2 and g_2 must be executed atomically.

We first show that a slice exists iff the predicate B is a regular predicate.

Figure 3. A computation (a), its lattice (b), a sub-lattice (c) and the corresponding slice (d).
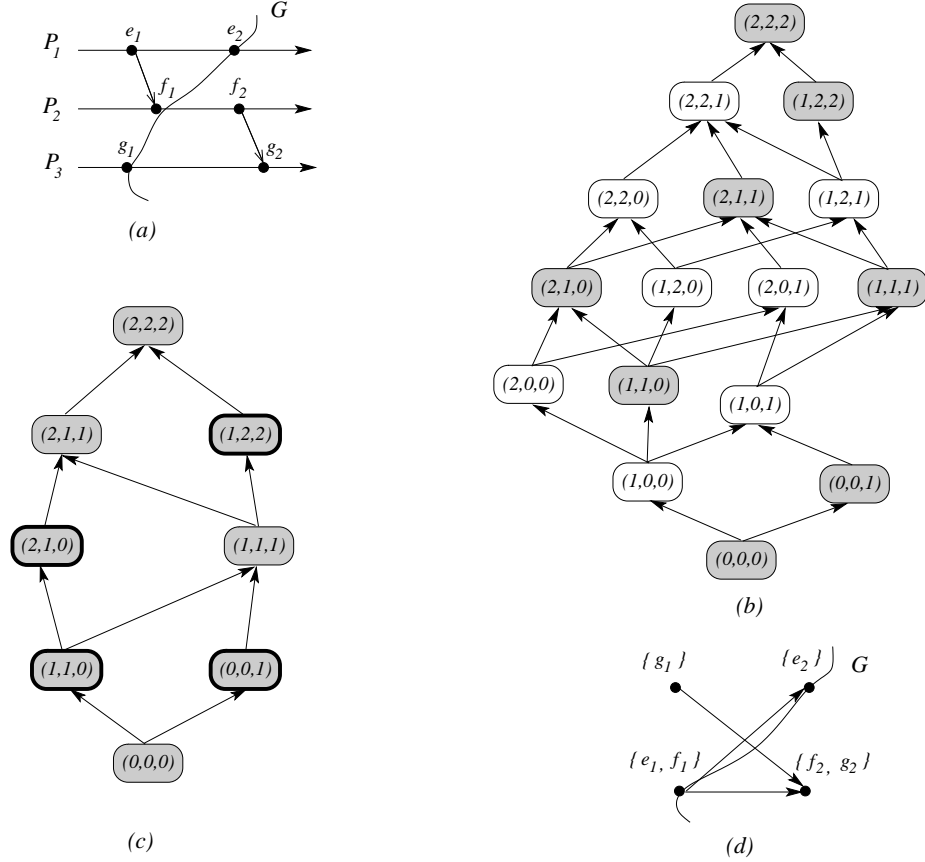
predicate: all channels are empty

⬜ : *consistent cut*   ⬛ : *consistent cut that satisfies the predicate*

⬛ : *join−irreducible element of the sub−lattice induced by the predicate*

Theorem 5 *A computation* $(E, \rightarrow)$ *has a non-empty slice with respect to a predicate B iff B is regular and $C_B(E)$ is non-empty.*

Proof: Assume that B is a regular predicate and $C_B(E)$ is non-empty. Therefore the set of consistent cuts in $C_B(E)$ forms a sublattice of the lattice of consistent cuts in $C(E)$. Any sublattice of a finite distributive lattice is also a finite distributive lattice [4]. Let $J(C_B(E))$ denote the set of join-irreducible elements of $C_B(E)$. Using Birkhoff's representation theorem for finite distributive lattices, we can infer that $C_B(E)$ is isomorphic to $C(J(C_B(E)))$ implying that a non-empty slice exists. The algorithm in Section 6 can be used to compute the triplet $(I_B, F, \rightarrow_B)$ that satisfies the necessary requirements.

Now, assume that $(E, \rightarrow)$ has a non-empty slice, say $(I_B, F, \rightarrow_B)$ with respect to the predicate B. Clearly,

$C_B(E)$ is non-empty. Further, since $C_B(E)$ is isomorphic to $C(F)$ and $C(F)$ is a finite distributive lattice, $C_B(E)$ is a lattice. Thus B is regular. ∎

6. Efficient algorithm

In this section we give an efficient algorithm to construct the slice of a distributed program given a regular predicate. Note that the size of the lattice and the size of the sublattice satisfying predicate B may be exponential in the number of processes; and therefore, a naive algorithm which enumerates all the consistent cuts satisfying B may be prohibitively expensive.

The efficient algorithm is specified formally in Figure 4. It takes as input a distributed computation specified as the happened-before diagram and a regular predicate B. There

---

Input: A computation $(E, \rightarrow)$ and a regular predicate B

Output: A computation slice $(I_B, F, \rightarrow_B)$ such that (1) Every consistent cut of F is also a consistent cut of E, (2) All consistent cuts of F satisfy B, and (3) All consistent cuts of E that satisfy B are also consistent cuts of F.

- Step 1. Let V be the least consistent cut in E that satisfies B. Use the algorithm in [2] to compute V. Since B is a regular predicate it is linear and hence the algorithm in [2] is applicable. If no such cut exists, output "slice does not exist".

- Step 2. Let W be the greatest consistent cut in E that satisfies B. Use the dual of the algorithm in [2] to compute W.

- Step 3. For each event $e \in W - V$, compute the least consistent cut in E that satisfies B and includes e. Call that cut $J(e)$.

- Step 4. Let R be an equivalence relation defined on events in $W - V$ as follows: $(e, f) \in R \equiv J(e) = J(f)$. Let the equivalence classes induced by R be C_i, $1 \leq i \leq m$. Output $(V, F, \rightarrow_B)$, where

$$F := \{C_i \mid 1 \leq i \leq m\}$$

$$\rightarrow_B := \{(C_i, C_j) \mid J(C_i) \subsetneq J(C_j)\}$$

where, for an equivalence class C, $J(C)$ is $J(e)$ for some event e in C.

Figure 4. Algorithm to construct a slice.

are two possible outcomes. There may not be any consistent cut in E that satisfies B. The algorithm will detect this condition and output it in Step 1. Otherwise, the algorithm outputs a computation slice in Step 4. We now describe each step in greater detail.

6.1. Step 1

This step determines if the predicate B is true on any consistent cut in E. Since B is a regular predicate it is also linear. Therefore, the algorithm presented in [2] is applicable. In fact, the algorithm returns the least consistent cut, say V, in E that satisfies B, if it exists. The algorithm in [2] is presented on a state based model, but can easily be transformed into an event based model. The complexity of the algorithm depends upon the complexity of determining an event to advance a given consistent cut G. In this paper, we will assume that the complexity of determining the required event is $O(N)$ and therefore the complexity of this step is $O(N|E|)$. In this step, we have exploited the closure of B

under set intersection.

6.2. Step 2

This step is dual of Step 1. Since B is a regular predicate, it is also closed under set union. Thus, if there is any consistent cut that satisfies B, then there exists the greatest consistent cut, say W, that satisfies B. To determine this cut, one could apply the algorithm in [2] backwards on the computation. Alternatively, one can apply the algorithm [2] to the computation E with all edges reversed.

Observe that if the least cut exists in Step 1, then we are guaranteed that the set of cuts satisfying B is non-empty. This, in turn, guarantees that the greatest cut satisfying B exists.

The computational complexity of this algorithm is the same as Step 1 - $O(N|E|)$.

6.3. Step 3

In this step, we determine the join-irreducible elements for the sublattice. For all events $e \in W - V$, we define a new predicate

$$B_e(X) \equiv B(X) \wedge (e \in X)$$

The following lemma shows that B_e is regular.

Lemma 6 *Given any regular predicate B, B_e as defined above is also a regular predicate.*

Proof: Define a predicate I_e which is true on a cut X iff e is included in X. Given consistent cuts X_1 and X_2 that satisfy I_e, it is clear that $X_1 \cup X_2$ and $X_1 \cap X_2$ are not only consistent but also satisfy I_e. Therefore, I_e is a regular predicate. This implies B_e is a conjunction of two regular predicates and is therefore a regular predicate. ∎

We define $J(e)$ as the least consistent cut that satisfies B_e. Since $e \in W$, we know that W satisfies B_e. Thus the set of cuts satisfying B_e is non-empty. Since B_e is a regular predicate, we conclude that $J(e)$ is well-defined.

We now show that $J(e)$ is a join-irreducible element of the lattice $(C_B(E), \subseteq)$.

Lemma 7 *$J(e)$ is a join-irreducible element of $(C_B(E), \subseteq)$.*

Proof: Let $X, Y \in C_B(E)$ be such that $J(e) = X \sqcup Y = X \cup Y$. Since $e \in J(e)$, we get that $e \in X \vee e \in Y$. Without loss of generality, assume the former. Since $e \in X$ and X satisfies B, we get that X is a consistent cut that includes e and satisfies B. However, $J(e)$ is the least consistent cut with these properties. Therefore $J(e) \subseteq X$. But, we also have $X \subseteq J(e)$. Therefore, $X = J(e)$. ∎

A naive implementation of this step will have $O(N|E|^2)$ complexity since the algorithm in [2] will be invoked exactly $|E|$ times. However, the next lemma enables to reduce the complexity of this step.

Lemma 8 *Let e and f be events of $(E, \rightarrow)$ and B be a regular predicate. Then,*

$$e \rightarrow f \;\Rightarrow\; J(e) \leq J(f)$$

Proof: Since $e \rightarrow f$ and $J(f)$ is a consistent cut, $e \in J(f)$. Since both $J(e)$ and $J(f)$ satisfy B, and $J(e)$ is the least consistent that contains e and satisfies B, $J(e) \subseteq J(f)$. ∎

Lemma 8 allows us to compute $J(e)$ for all events $e \in W - V$ on some process P in a single scan of the computation. Thus the complexity of this step can be reduced to $O(N^2|E|)$.

6.4. Step 4

In this step we enumerate all the elements of the computation slice. We first define an equivalence relation R on events in $W - V$ as follows:

$$(e, f) \in R \;\equiv\; J(e) = J(f)$$

Let C_i, $1 \leq i \leq m$, denote the equivalence classes induced by R. Intuitively, each equivalence class constitutes an element of the slice and all events in an equivalence class must be executed atomically. For a class C, let $J(C)$ be $J(e)$ for some event e in C. The slice $(I_B, F, \rightarrow_B)$ is given by

$$I_B := V$$

$$F := \{ C_i \mid 1 \leq i \leq m \}$$

$$\rightarrow_B := \{ (C_i, C_j)) \mid J(C_i) \subsetneq J(C_j) \}$$

It can be proved that

$$J(C_i) = V \cup C_i \cup \Big(\bigcup_{C_j \rightarrow_B C_i} C_j \Big)$$

It is clear from the construction and Lemma 7 that F contains only the join-irreducible elements of $C_B(E)$. We now show that F contains all join-irreducible elements of $(C_B(E), \subseteq)$. To that end, it suffices to show the following.

Lemma 9 *Every cut X in $C_B(E)$ can be written as a join of elements of F.*

Proof: Since X satisfies B, it satisfies $V \subseteq X \subseteq W$. This implies that X can be written as $V \cup Y$ where $Y \subseteq W - V$. Let

$$Z = \bigcup_{e \in Y} J(e)$$

It is clear that Z is a join of elements in F. We will show that $X = Z$. Since $J(e)$ contains e and at least V, it is clear

that $X \subseteq Z$. We show that $Z \subseteq X$. To prove that, it is sufficient to show that

$$\forall\, e \in Y : J(e) \subseteq X$$

This follows from the fact that $e \in X$, X satisfies B, and $J(e)$ is the least cut that satisfies these properties. ∎

The equivalence classes can be computed by determining the strongly connected components of the directed graph with vertices as events in $W - V$ and an edge from event e to event f iff either

1. e and f occur on the same process and f is the successor of e or

2. f is the earliest event on its process such that $J(e) \subseteq J(f)$.

It is easy to see that the directed graph described above has $O(|E|)$ vertices, $O(N|E|)$ edges and can be constructed in $O(N^2|E|)$ time. Thus the complexity of this step is $O(N^2|E|)$. We are now ready for the main result of this paper.

Theorem 10 *Given any computation $(E, \rightarrow)$ and a regular predicate B, the algorithm outputs the computation slice, if it exists, in $O(N^2|E|)$ time.*

Proof: The fact that triplet $(I_B, F, \rightarrow_B)$ output by the algorithm constitutes the slice of $(E, \rightarrow)$ with respect to B follows from the definition of F, Lemma 7 and Lemma 9. The complexity of each step is $O(N^2|E|)$ giving us an overall complexity of $O(N^2|E|)$. ∎

7. Applications of computation slicing

7.1. Predicate detection

Consider any global predicate B that is expressed as a conjunction of a regular predicate B_1 and another global predicate B_2. Given that the predicate B_2 does not have any structure that can be exploited for efficient detection, the predicate detection algorithm is forced to traverse the execution lattice. However, with the results of this paper we can proceed as follows. Instead of searching the original lattice, we can search the reduced lattice - the one in which all cuts satisfy B_1.

7.2. Predicate control

The predicate control problem states that given a distributed computation and a global predicate, is it possible to add synchronization arrows to the computation such that the predicate always stays true. The predicate control problem provides a useful abstraction for analyzing executions

of distributed programs which can greatly facilitate the detection and localization of bugs [15, 14]. It can also be used to provide software fault-tolerance against synchronization faults.

It can be proved that a predicate can be controlled in a computation iff there exists path in the lattice from the initial to the final consistent cut such that every consistent cut in the path satisfies the predicate. A regular predicate B can therefore be controlled in a computation $(E, \rightarrow)$ iff the length of the longest chain in $C_B(E)$ is $|E|$, that is, $|J(C_B(E))| = |F| = |E|$, where $(I_B, F, \rightarrow_B)$ is the slice of $(E, \rightarrow)$ with respect to B.

The dual of the predicate control problem is the problem of observing a predicate under $definitely$ modality [3]. A predicate is definitely true in a computation iff it eventually becomes true in all runs of the computation. As a corollary, this problem can now be solved efficiently when the negation of the predicate is regular.

8. Conclusion

Analyzing executions of distributed programs is an important problem in asynchronous distributed systems. This problem arises in various contexts such as design, testing and debugging, and fault-tolerance of distributed systems. In this paper, we introduce the notion of a slice of a distributed computation. A slice of a distributed computation with respect to a global predicate is a computation which captures those and only those consistent cuts of the original computation which satisfy the global predicate. A slice can significantly reduce the size of the computation to be analyzed, thereby making the understanding of program behaviour easier.

We give the necessary and sufficient condition for a computation slice to exist. In particular, we prove that a slice of a computation with respect to a predicate exists iff the set of consistent cuts that satisfy the predicate forms a sublattice of the lattice of consistent cuts. This leads us to define the class of regular predicates. An efficient algorithm with $O(N^2|E|)$ complexity to compute a slice is presented, where N is the number of processes and E is the set of events in the system. Slicing can be used to monitor a predicate under $possibly$ modality, if it is regular, and under $definitely$ modality if its negation is a regular predicate.

References

[1] 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.

[2] C. Chase and V. K. Garg. Detection of Global Predicates: Techniques and their Limitations. *Distributed Computing*, 11(4):191–201, 1998.

[3] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.

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

[5] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed Slicing and Partial Re-execution for Distributed Programs. In *Proceedings of the 5th Workshop on Language and Compilers for Parallel Computing*, pages 329–337, 1992.

[6] V. K. Garg, C. Chase, R. Kilgore, and J. R. Mitchell. Efficient Detection of Channel Predicates in Distributed Systems. *Journal of Parallel and Distributed Computing*, 45(2):134–147, Sept. 1997.

[7] 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, Mar. 1994.

[8] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 171–181, Aug. 1988.

[9] B. Korel and R. Ferguson. Dynamic Slicing of Distributed Progrmas. *Applied Mathematics and Computer Science Journal*, 2(2):199–215, 1992.

[10] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[11] B. Korel and J. Rilling. Application of Dynamic Slicing in Program Debugging. In M. Kamkar, editor, *Proceedings of the 3rd International International Workshop on Automated Debugging (AADEBUG)*, pages 43–57, May 1997.

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

[13] 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. Elsevier Science Publishers B. V. (North-Holland), 1989.

[14] N. Mittal and V. K. Garg. Debugging Distributed Programs Using Controlled Re-execution. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 239–248, July 2000.

[15] A. Tarafdar and V. K. Garg. Predicate Control for Active Debugging of Distributed Programs. In *Proceedings of the 9th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, Orlando, 1998.

[16] A. Tarafdar and V. K. Garg. Software Fault Tolerance of Concurrent Programs Using Controlled Re-execution. In *Proceedings of the 13th Symposium on Distributed Computing (DISC)*, pages 210–224, Sept. 1999.

[17] M. Weiser. Programmers Use Slices When Debugging. *Communications of the ACM (CACM)*, 25(7):446–452, 1982.