Computation Slicing: Techniques and Theory

Neeraj Mittal[1] and Vijay K. Garg[2]*

[1] Department of Computer Sciences
The University of Texas at Austin, Austin, TX 78712, USA
neerajm@cs.utexas.edu http://www.cs.utexas.edu/users/neerajm
[2] Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, TX 78712, USA
garg@ece.utexas.edu http://www.ece.utexas.edu/~garg

Abstract. We generalize the notion of slice introduced in our earlier paper [6]. A slice of a distributed computation with respect to a global predicate is the smallest computation that contains all consistent cuts of the original computation that satisfy the predicate. We prove that slice exists for all global predicates. We also establish that it is, in general, NP-complete to compute the slice. An optimal algorithm to compute slices for special cases of predicates is provided. Further, we present an efficient algorithm to graft two slices, that is, given two slices, either compute the smallest slice that contains all consistent cuts that are common to both slices or compute the smallest slice that contains all consistent cuts that belong to at least one of the slices. We give application of slicing in general and grafting in particular to global property evaluation of distributed programs. Finally, we show that the results pertaining to consistent global checkpoints [14, 18] can be derived as special cases of computation slicing.

1 Introduction

Writing distributed programs is an error prone activity; it is hard to reason about them because they suffer from the combinatorial explosion problem. Testing and debugging, and software fault-tolerance is an important way to ensure the reliability of distributed systems. Thus it becomes necessary to develop techniques that facilitate the analysis of distributed computations. Various abstractions such as predicate detection (e.g., [1, 3, 7]) and predicate control [16, 17, 11] have been defined to carry out such analysis.

In our earlier paper [6], we propose another abstraction, called *computation slice*, which was defined as: a slice of a distributed computation with respect to a global predicate is another computation that contains *those and only those* consistent cuts (or snapshots) of the original computation that satisfy the predicate. In [6], we also introduce a class of global predicates called *regular predicates*: a global predicate is regular iff whenever two consistent cuts satisfy the predicate

* supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

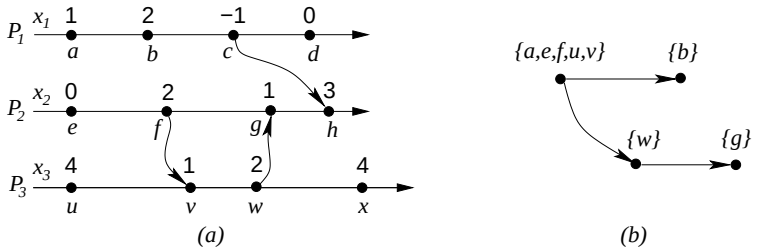

Fig. 1. (a) A computation and (b) its slice with respect to $(x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$.

then the cuts given by their set intersection and set union also satisfy the predicate. We show that slice exists only for regular predicates and present an efficient algorithm to compute the slice. The class of regular predicates is closed under conjunction.

A limitation of the definition of slice in [6] is that slice exists only for a specific class of predicates. This prompted us to weaken the definition of slice to the *smallest* computation that contains all consistent cuts of the original computation that satisfy the predicate. In this paper, we show that slice exists for all global predicates.

The notion of computation slice is analogous to the concept of program slice [19]. 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. A slice could be static [19] or dynamic (for a specific program input) [9]. The notion of a slice has been also extended to distributed programs [8]. Program slicing has been shown to be useful in program debugging, testing, program understanding and software maintenance [9, 19]. 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.

Computation slicing is also useful for reducing search space for NP-complete problems such as predicate detection [3, 7, 15, 13]. Given a distributed computation and a global predicate, predicate detection requires finding a consistent cut of the computation, if it exists, that satisfies the predicate. It is a fundamental problem in distributed system and arises in contexts such as software fault tolerance, and testing and debugging.

As an illustration, suppose we want to detect the predicate $(x_1 * x_2 + x_3 < 5) \wedge (x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$ in the computation shown in Fig. 1(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. The events are represented by solid circles. Each event is labeled with the value of the respective variable immediately after the event is executed. For example, the value of variable x_1 immediately after executing the event c is -1. The first event on each process initializes the state of the process and every consistent cut contains these initial events. Without computation slicing,

we are forced to examine all consistent cuts of the computation, twenty eight in total, to ascertain whether some consistent cut satisfies the predicate. Alternatively, we can compute a slice of the computation with respect to the predicate $(x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$ as portrayed in Fig. 1(b). The slice is modeled by a directed graph. Each vertex of the graph corresponds to a subset of events. If a vertex is contained in a consistent cut, the interpretation is that all events corresponding to the vertex are contained in the cut. Moreover, a vertex belongs to a consistent cut only if all its incoming neighbours are also present in the cut. We can now restrict our search to the consistent cuts 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 consistent cuts than the computation itself—exponentially smaller in many cases—resulting in substantial savings.

We also show that the results pertaining to consistent global checkpoints [14, 18] can be derived as special cases of computation slicing. In particular, we furnish an alternate characterization of the condition under which individual local checkpoints can be combined with others to form a consistent global checkpoint (consistency theorem by Netzer and Xu [14]): a set of local checkpoints can belong to the same consistent global snapshot iff the local checkpoints in the set are mutually consistent (including with itself) in the slice. Moreover, the R-graph (rollback-dependency graph) defined by Wang [18] is a special case of the slice. The minimum and maximum consistent global checkpoints that contain a set of local checkpoints [18] can also be easily obtained using the slice.

In summary, this paper makes the following contributions:

- In Section 3, we generalize the notion of computation slice introduced in our earlier paper [6]. We show that slice exists for all global predicates in Section 4.
- We establish that it is, in general, NP-complete to determine whether a global predicate has a non-empty slice in Section 4.
- In Section 4, an application of computation slicing to monitoring global properties in distributed systems is provided. Specifically, we give an algorithm to determine whether a global predicate satisfying certain properties is possibly true, invariant or controllable in a distributed computation using slicing.
- We present an efficient representation of slice in Section 5 that we use later to devise an efficient algorithm to *graft* two slices in Section 6. Grafting can be done in two ways. Given two slices, we can either compute the smallest slice that contains all consistent cuts that are common to both slices or compute the smallest slice that contains all consistent cuts that belong to at least one of the slices. An efficient algorithm using grafting to compute slice for complement of a regular predicate, called *co-regular* predicate, is provided. We also show how grafting can be used to avoid examining many consistent cuts when detecting a predicate.
- We provide an optimal algorithm to compute slices for special cases of regular predicates in Section 7. In our earlier paper [6], the algorithm to compute slices has $O(N^2|E|)$ time complexity, where N is the number of processes

and E is the set of events in the distributed system. The algorithm presented in this paper has $O(|E|)$ complexity which is optimal.
- Finally, in Section 7, we show that the results pertaining to consistent global checkpoints [14, 18] can be derived as special cases of computation slicing.

Due to lack of space, the proofs of lemmas, theorems and corollaries, and other details have been omitted. Interested reader can find them in the technical report [12].

2 Model and Notation

2.1 Lattices

Given a lattice, we use $\sqcap$ and $\sqcup$ to denote its meet (infimum) and join (supremum) operators, respectively. A lattice is *distributive* iff meet distributes over join. Formally, $a \sqcap (b \sqcup c) \equiv (a \sqcap b) \sqcup (a \sqcap c)$.

2.2 Directed Graphs: Path- and Cut-Equivalence

Traditionally, a distributed computation is modeled by a partial order on a set of events. We use directed graphs to model both distributed computation and slice. Directed graphs allow us to handle both of them in a convenient and uniform manner.

Given a directed graph G, let $\mathsf{V}(G)$ and $\mathsf{E}(G)$ denote its set of vertices and edges, respectively. A subset of vertices of a directed graph form a *consistent cut* iff the subset contains a vertex only if it contains all its incoming neighbours. Formally,

$$C \text{ is a consistent cut of } G \triangleq \langle \forall e, f \in \mathsf{V}(G) : (e, f) \in \mathsf{E}(G) : f \in C \Rightarrow e \in C \rangle$$

Observe that a consistent cut either contains all vertices in a cycle or none of them. This observation can be generalized to a strongly connected component. Traditionally, the notion of consistent cut (*down-set* or *order ideal*) is defined for partially ordered sets [5]. Here, we extend the notion to sets with arbitrary orders. Let $\mathcal{C}(G)$ denote the set of consistent cuts of a directed graph G. Observe that the empty set $\emptyset$ and the set of vertices $\mathsf{V}(G)$ trivially belong to $\mathcal{C}(G)$. We call them *trivial* consistent cuts. The following theorem is a slight generalization of the result in lattice theory that the set of down-sets of a partially ordered set forms a distributive lattice [5].

Theorem 1. *Given a directed graph G, $\langle \mathcal{C}(G); \subseteq \rangle$ forms a distributive lattice.*

The theorem follows from the fact that, given two consistent cuts of a graph, the cuts given by their set intersection and set union are also consistent.

A directed graph G is *cut-equivalent* to a directed graph H iff they have the same set of consistent cuts, that is, $\mathcal{C}(G) = \mathcal{C}(H)$. Let $\mathcal{P}(G)$ denote the set of pairs of vertices (u, v) such that there is a path from u to v in G. We assume

that each vertex has a path to itself. A directed graph G is *path-equivalent* to a directed graph H iff a path from vertex u to vertex v in G implies a path from vertex u to vertex v in H and vice versa, that is, $\mathcal{P}(G) = \mathcal{P}(H)$.

Lemma 1. *Let G and H be directed graphs on the same set of vertices. Then,*

$$\mathcal{P}(G) \subseteq \mathcal{P}(H) \;\equiv\; \mathcal{C}(G) \supseteq \mathcal{C}(H)$$

Lemma 1 implies that two directed graphs are cut-equivalent iff they are path-equivalent. This is significant because path-equivalence can be verified in polynomial-time ($|\mathcal{P}(G)| = O(|\mathsf{V}(G)|^2)$) as compared to cut-equivalence which is computationally expensive to ascertain in general ($|\mathcal{C}(G)| = O(2^{|\mathsf{V}(G)|})$).

2.3 Distributed Computations as Directed Graphs

We assume an asynchronous distributed system [12] with the set of processes $P = \{P_1, P_2, \ldots, P_N\}$. Processes communicate and synchronize with each other by sending messages over a set of reliable channels.

A *local computation* of a process is described by a sequence of events that transforms the *initial state* of the process into the *final state*. At each step, the *local state* of a process is captured by the initial state and the sequence of events that have been executed up to that step. Each event is a *send event*, a *receive event*, or an *internal event*. An event causes the local state of a process to be updated. Additionally, a send event causes a message to be sent and a receive event causes a message to be received. We assume the presence of fictitious *initial* and *final events* on each process P_i, denoted by $\perp_i$ and $\top_i$, respectively. The initial event occurs before any other event on the process and initializes the state of the process. The final events occurs after all other events on the process.

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. We denote the order of events on process P_i by $\rightsquigarrow_{P_i}$. Let $\rightsquigarrow_P$ be the union of all $\rightsquigarrow_{P_i}$s, $1 \leqslant i \leqslant N$, and $\underline{\rightsquigarrow}_P$ denote the reflexive closure of $\rightsquigarrow_P$.

We model a *distributed computation* (or simply a *computation*), denoted by $\langle E, \rightarrow \rangle$, as a directed graph with vertices as the set of events E and edges as $\rightarrow$. To limit our attention to only those consistent cuts that can actually occur during an execution, we assume that, for any computation $\langle E, \rightarrow \rangle$, $\mathcal{P}(\langle E, \rightarrow \rangle)$ contains at least the Lamport's happened-before relation [10]. We assume that the set of all initial events belong to the same strongly connected component. Similarly, the set of all final events belong to the same strongly connected component. This ensures that any non-trivial consistent cut will contain all initial events and none of the final events. As a result, every consistent cut of a computation in traditional model is a non-trivial consistent cut of the computation in our model and vice versa. Only non-trivial consistent cuts are of real interest to us. We will see later that our model allows us to capture empty slices in a very convenient fashion.

A distributed computation in our model can contain cycles. This is because whereas a computation in the happened-before model captures the observable

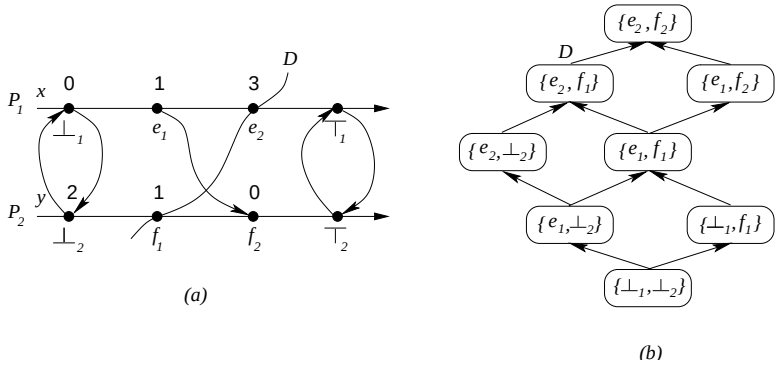

Fig. 2. (a) A computation and (b) the lattice corresponding to its consistent cuts.

order of execution of events, a computation in our model captures the set of possible consistent cuts.

A *frontier* of a consistent cut is the set of those events of the cut whose successors, if they exist, are not contained in the cut. Formally,

$$frontier(C) \triangleq \{e \in C \mid succ(e) \text{ exists } \Rightarrow succ(e) \notin C\}$$

A consistent cut is uniquely characterized by its frontier and vice versa. Thus sometimes, especially in figures, we specify a consistent cut by simply listing the events in its frontier instead of enumerating all its events. Two events are said to be *consistent* iff they are contained in the frontier of some consistent cut, otherwise they are *inconsistent*. It can be verified that events e and f are consistent iff there is no path in the computation from $succ(e)$, if it exists, to f and from $succ(f)$, if it exists, to e. Also, note that, in our model, an event can be inconsistent with itself. Fig. 2 depicts a computation and the lattice of its (non-trivial) consistent cuts. A consistent cut in the figure is represented by its frontier. For example, the consistent cut D is represented by $\{e_2, f_1\}$.

2.4 Global Predicates

A *global predicate* (or simply a *predicate*) is a boolean-valued function defined on variables of processes. It is evaluated on events in the frontier of a consistent cut. Some examples are mutual exclusion and "at least one philosopher does not have any fork". We leave the predicate undefined for the trivial consistent cuts. A global predicate is *local* iff it depends on variables of at most one process. For example, "P_i is in red state" and "P_i does not have the token".

3 Slicing a Distributed Computation

In this section, we define the notion of slice of a computation with respect to a predicate. The definition given here is weaker than the definition given in our

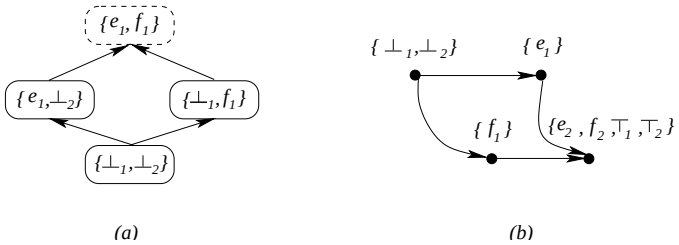

Fig. 3. (a) The sublattice of the lattice in Fig. 2(b) with respect to the predicate $((x < 2) \wedge (y > 1)) \bigvee (x < 1)$, and (b) the corresponding slice.

earlier paper [6]. However, slice now exists with respect to every predicate (not just specific predicates).

Definition 1 (Slice). *A* slice *of a computation with respect to a predicate is the smallest directed graph (with minimum number of consistent cuts) that contains all consistent cuts of the original computation that satisfy the predicate.*

We will later show that the smallest computation is well-defined for every predicate. A slice of computation $\langle E, \rightarrow \rangle$ with respect to a predicate b is denoted by $\langle E, \rightarrow \rangle_b$. Note that $\langle E, \rightarrow \rangle = \langle E, \rightarrow \rangle_{\mathsf{true}}$. In the rest of the paper, we use the terms "computation", "slice" and "directed graph" interchangeably.

Fig. 3(a) depicts the set of consistent cuts of the computation in Fig. 2(a) that satisfy the predicate $((x < 2) \wedge (y > 1)) \bigvee (x < 1)$. The cut shown with dashed outline does not actually satisfy the predicate but has to be included to complete the sublattice. Fig. 3(b) depicts the slice of the computation with respect to the predicate. In the figure, all events in a subset belong to the same strongly connected component.

In our model, every slice derived from the computation $\langle E, \rightarrow \rangle$ will have the trivial consistent cuts ($\emptyset$ and E) among its set of consistent cuts. Consequently, a slice is *empty* iff it has no non-trivial consistent cuts. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut.

A slice of a computation with respect to a predicate is *lean* iff every consistent cut of the slice satisfies the predicate.

4 Regular Predicates

A global predicate is *regular* iff the set of consistent cuts that satisfy the predicate forms a *sublattice* of the lattice of consistent cuts [6]. Equivalently, if two consistent cuts satisfy a regular predicate then the cuts given by their set intersection and set union will also satisfy the predicate. Some examples of regular predicates are any local predicate and channel predicates such as 'there are at most k messages in transit from P_i to P_j". The class of regular predicates is closed under conjunction [6]. We prove elsewhere [6] that the slice of a computation with respect to a predicate is lean iff the predicate is regular. We next show how

slicing can be used to monitor predicates in distributed systems. Later, we use the notion of regular predicates to prove that the slice exists and is well-defined with respect to every predicate.

4.1 Using Slices to Monitor Regular Predicates

A predicate can be monitored under four modalities, namely *possibly*, *definitely*, *invariant* and *controllable* [3, 7, 17, 11]. A predicate is *possibly* true in a computation iff there is a consistent cut of the computation that satisfies the predicate. On the other hand, a predicate *definitely* holds in a computation iff it eventually becomes true in all runs of the computation (a *run* is a path in the lattice of consistent cuts). The predicates $invariant{:}b$ and $controllable{:}b$ are duals of predicates $possibly: b$ and $controllable: b$, respectively. Predicate detection normally involves detecting a predicate under *possibly* modality whereas predicate control involves monitoring a predicate under *controllable* modality. Monitoring has applications in the areas of testing and debugging and software fault-tolerance of distributed programs.

The next theorem describes how $possibly: b$, $invariant: b$ and $controllable: b$ can be computed using the notion of slice when b is a regular predicate. We do not yet know the complexity of computing $definitely: b$ when b is regular.

Theorem 2. *A regular predicate is*

1. **possibly** *true in a computation iff the slice of the computation with respect to the predicate has at least one non-trivial consistent cut, that is, it has at least two strongly connected components.*
2. **invariant** *in a computation iff the slice of the computation with respect to the predicate is cut-equivalent to the computation.*
3. **controllable** *in a computation iff the slice of the computation with respect to the predicate has the same number of strongly connected components as the computation.*

Observe that the first proposition holds for any arbitrary predicate. Since detecting whether a predicate possibly holds in a computation is NP-complete in general [2, 15, 13], it is, in general, NP-complete to determine whether a predicate has a non-empty slice.

4.2 Regularizing a Non-Regular Predicate

In this section, we show that slice exists and is well-defined with respect to every predicate. We know that it is true for at least regular predicates [6]. In addition, the slice with respect to a regular predicate is lean. We exploit these facts and define a closure operator, denoted by *reg*, which, given a computation, converts an arbitrary predicate into a regular predicate satisfying certain properties. Given a computation, let $\mathcal{R}$ denote the set of predicates that are regular with respect to the computation.

Definition 2 (reg). *Given a predicate b, we define $reg\,(b)$ as the predicate that satisfies the following conditions:*

1. *it is regular, that is, $reg\,(b) \in \mathcal{R}$,*
2. *it is weaker than b, that is, $b \Rightarrow reg\,(b)$, and*
3. *it is stronger than any other predicate that satisfies 1 and 2, that is, $\langle \forall\, b' : b' \in \mathcal{R} : (b \Rightarrow b') \Rightarrow (reg\,(b) \Rightarrow b') \rangle$*

Informally, $reg\,(b)$ is the *strongest regular predicate weaker* than b. In general, $reg\,(b)$ not only depends on the predicate b but also on the computation under consideration. We assume the dependence on computation to be implicit and make it explicit only when necessary. The next theorem establishes that $reg\,(b)$ exists for every predicate. Observe that the slice for b is given by the slice for $reg\,(b)$. Thus slice exists and is well-defined for all predicates.

Theorem 3. *Given a predicate b, $reg\,(b)$ exists and is well-defined.*

Thus, given a computation $\langle E, \rightarrow \rangle$ and a predicate b, the slice of $\langle E, \rightarrow \rangle$ with respect to b can be obtained by first applying reg operator to b to get $reg\,(b)$ and then computing the slice of $\langle E, \rightarrow \rangle$ with respect to $reg\,(b)$.

Theorem 4. *reg is a closure operator. Formally,*

1. *$reg\,(b)$ is weaker than b, that is, $b \;\Rightarrow\; reg\,(b)$,*
2. *reg is monotonic, that is, $(b \Rightarrow b') \;\Rightarrow\; (reg\,(b) \Rightarrow reg\,(b'))$, and*
3. *reg is idempotent, that is, $reg\,(reg\,(b)) \;\equiv\; reg\,(b)$.*

From the above theorem it follows that [5, Theorem 2.21],

Corollary 1. *$\langle \mathcal{R}; \Rightarrow \rangle$ forms a lattice.*

The meet and join of two regular predicates b_1 and b_2 is given by

$$b_1 \sqcap b_2 \;\triangleq\; b_1 \wedge b_2$$
$$b_1 \sqcup b_2 \;\triangleq\; reg\,(b_1 \vee b_2)$$

The dual notion of $reg\,(b)$, the weakest regular predicate stronger than b, is conceivable. However, such a predicate may not always be unique [12].

5 Representing a Slice

Observe that any directed graph that is cut-equivalent or path-equivalent to a slice constitutes its valid representation. However, for computational purposes, it is preferable to select those graphs to represent a slice that have fewer edges and can be constructed cheaply. In this section, we show that every slice can represented by a directed graph with $O(|E|)$ vertices and $O(N|E|)$ edges. Furthermore, the graph can be built in $O(N^2|E|)$ time.

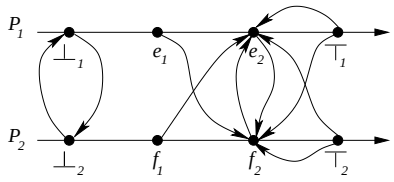

Fig. 4. The skeletal representation of the slice in Fig. 3(b) (without self-loops).

Given a computation $\langle E, \rightarrow \rangle$, a regular predicate b and an event e, let $J_b(e)$ denote the least consistent cut of $\langle E, \rightarrow \rangle$ that contains e and satisfies b. If $J_b(e)$ does not exist then it is set to the trivial consistent cut E. Here, we use E as a *sentinel* cut. Fig. 4 depicts a directed graph that represents the slice shown in Fig. 3(b). In the figure, $J_b(e_1) = \{\perp_1, e_1, \perp_2\}$ and $J_b(f_2) = \{\perp_1, e_1, e_2, \top_1, \perp_2, f_1, f_2, \top_2\}$.

The cut $J_b(e)$ can also be viewed as the least consistent cut of the slice $\langle E, \rightarrow \rangle_b$ that contains the event e. The results in [6] establish that it is sufficient to know $J_b(e)$ for each event e in order to recover the slice. In particular, a directed graph with E as the set of vertices and an edge from an event e to an event f iff $J_b(e) \subseteq J_b(f)$ is cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$. We also present an $O(N^2|E|)$ algorithm to compute $J_b(e)$ for each event e. However, the graph so obtained can have as many as $\Omega(|E|^2)$ edges.

Let $F_b(e, i)$ denote the earliest event f on P_i such that $J_b(e) \subseteq J_b(f)$. Informally, $F_b(e, i)$ is the earliest event on P_i that is reachable from e in the slice $\langle E, \rightarrow \rangle_b$. For example, in Fig. 4, $F_b(e_1, 1) = e_1$ and $F_b(e_1, 2) = f_2$. Given $J_b(e)$ for each event e, $F_b(e, i)$ for each event e and process P_i can be computed in $O(N|E|)$ time [12]. We now construct a directed graph that we call the *skeletal representation* of the slice with respect to b and denote it by G_b. The graph G_b has E as the set of vertices and the following edges: (1) for each event e, that is not a final event, there is an edge from e to $succ(e)$, and (2) for each event e and process P_i, there is an edge from e to $F_b(e, i)$.

The skeletal representation of the slice depicted in Fig. 3(b) is shown in Fig. 4. To prove that the graph G_b is actually cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$, it suffices to show the following:

Theorem 5. *For events e and f, $J_b(e) \subseteq J_b(f) \equiv (e, f) \in \mathcal{P}(G_b)$.*

Besides having computational benefits, the skeletal representation of a slice can be used to devise a simple and efficient algorithm to graft two slices.

6 Grafting Two Slices

In this section, we present algorithm to graft two slices which can be done with respect to meet or join. Informally, the former case corresponds to the smallest slice that contains all consistent cuts common to both slices whereas the latter case corresponds to the smallest slice that contains consistent cuts of both slices.

In other words, given slices $\langle E, \rightarrow \rangle_{b_1}$ and $\langle E, \rightarrow \rangle_{b_2}$, where b_1 and b_2 are regular predicates, we provide algorithm to compute the slice $\langle E, \rightarrow \rangle_b$, where b is either $b_1 \sqcap b_2 = b_1 \wedge b_2$ or $b_1 \sqcup b_2 = reg\,(b_1 \vee b_2)$. Grafting enables us to compute the slice for an arbitrary boolean expression of local predicates—by rewriting it in DNF—although it may require exponential time in the worst case. Later, in this section, we present an efficient algorithm based on grafting to compute slice for a *co-regular* predicate (complement of a regular predicate). We also show how grafting can be used to avoid examining many consistent cuts when detecting a predicate under *possibly* modality.

6.1 Grafting with respect to Meet: $\mathbf{b \equiv b_1 \sqcap b_2 \equiv b_1 \wedge b_2}$

In this case, the slice $\langle E, \rightarrow \rangle_b$ contains a consistent cut of $\langle E, \rightarrow \rangle$ iff the cut satisfies b_1 as well as b_2. Let $F_{\min}(e,i)$ denote the earlier of events $F_{b_1}(e,i)$ and $F_{b_2}(e,i)$, that is, $F_{\min}(e,i) = \min\{F_{b_1}(e,i), F_{b_2}(e,i)\}$. The following lemma establishes that, for each event e and process P_i, $F_{\min}(e,i)$ cannot occur before $F_b(e,i)$.

Lemma 2. *For each event e and process P_i, $F_b(e,i) \underline{\leadsto}_P F_{\min}(e,i)$.*

We now construct a directed graph $G_{\min}$ which is similar to G_b, the skeletal representation for $\langle E, \rightarrow \rangle_b$, except that we use $F_{\min}(e,i)$ instead of $F_b(e,i)$ in its construction. The next theorem proves that $G_{\min}$ is cut-equivalent to G_b.

Theorem 6. *$G_{\min}$ is cut-equivalent to G_b.*

Roughly speaking, the aforementioned algorithm computes the union of the sets of edges of each slice. Note that, in general, $F_b(e,i)$ need not be same as $F_{\min}(e,i)$ [12]. This algorithm can be generalized to conjunction of an arbitrary number of regular predicates.

6.2 Grafting with respect to Join: $\mathbf{b \equiv b_1 \sqcup b_2 \equiv reg\,(b_1 \vee b_2)}$

In this case, the slice $\langle E, \rightarrow \rangle_b$ contains a consistent cut of $\langle E, \rightarrow \rangle$ if the cut satisfies either b_1 or b_2. The dual of the graph $G_{\min}$—min replaced by max—denoted by $G_{\max}$ (surprisingly) turns out to be cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$. As before, let $F_{\max}(e,i)$ denote the later of events $F_{b_1}(e,i)$ and $F_{b_2}(e,i)$, that is, $F_{\max}(e,i) = \max\{F_{b_1}(e,i), F_{b_2}(e,i)\}$. The following lemma establishes that, for each event e and process P_i, $F_b(e,i)$ cannot occur before $F_{\max}(e,i)$.

Lemma 3. *For each event e and process P_i, $F_{\max}(e,i) \underline{\leadsto}_P F_b(e,i)$.*

We now construct a directed graph $G_{\max}$ that is similar to G_b, the skeletal representation for $\langle E, \rightarrow \rangle_b$, except that we use $F_{\max}(e,i)$ instead of $F_b(e,i)$ in its construction. The next theorem proves that $G_{\max}$ is cut-equivalent to G_b.

Theorem 7. *$G_{\max}$ is cut-equivalent to G_b.*

Intuitively, the above-mentioned algorithm computes the intersection of the sets of edges of each slice. In this case, in contrast to the former case, $F_b(e,i)$ is actually identical to $F_{\max}(e,i)$ [12]. This algorithm can be generalized to disjunction of an arbitrary number of regular predicates.

6.3 Applications of Grafting

Computing Slice for a Co-Regular Predicate. Given a regular predicate, we give an algorithm to compute the slice of a computation with respect to its negation—a co-regular predicate. In particular, we express the negation as disjunction of polynomial number of regular predicates. The slice can then be computed by grafting together slices for each disjunct.

Let $\langle E, \rightarrow \rangle$ be a computation and $\langle E, \rightarrow \rangle_b$ be its slice with respect to a regular predicate b. For convenience, let $\rightarrow_b$ be the edge relation for the slice. *We assume that both $\rightarrow$ and $\rightarrow_b$ are transitive relations.* Our objective is to find a property that *distinguishes* the consistent cuts that belong to the slice from the consistent cuts that do not. Consider events e and f such that $e \not\rightarrow f$ but $e \rightarrow_b f$. Then, clearly, a consistent cut that contains f but does not contain e cannot belong to the slice. On the other hand, every consistent cut of the slice that contains f also contains e. This motivates us to define a predicate $prevents(f, e)$ as follows:

$$C \text{ satisfies } prevents(f, e) \triangleq (f \in C) \wedge (e \notin C)$$

It can be proved that $prevents(f, e)$ is a regular predicate [12]. It turns out that every consistent cut that does not belong to the slice satisfies $prevents(f, e)$ for some events e and f such that $(e \not\rightarrow f) \wedge (e \rightarrow_b f)$ holds. Formally,

Theorem 8. *Let C be a consistent cut of $\langle E, \rightarrow \rangle$. Then,*

$$C \text{ satisfies } \neg b \equiv \langle \exists\, e, f : (e \rightarrow_b f) \wedge (e \not\rightarrow f) : C \text{ satisfies } prevents(f, e) \rangle$$

Theorem 8 implies that $\neg b$ can be expressed as disjunction of *prevents*'.

Pruning State Space for Predicate Detection. Detecting a predicate under *possibly* modality is NP-complete in general [2, 15, 13]. Using grafting, we can reduce the search space for predicates composed from local predicates using $\neg$, $\wedge$ and $\vee$ operators. We first transform the predicate into an equivalent predicate in which $\neg$ is applied directly to the local predicates and never to more complex expressions. Observe that the negation of a local predicate is also a local predicate. We start by computing slices with respect to these local predicates. This can be done because a local predicate is regular and hence the algorithm given in [6] can be used to compute the slice. We then recursively graft slices together, with respect to the appropriate operator, working our way out from the local predicates until we reach the whole predicate. This will give us a slice of the computation—not necessarily the smallest—which contains all consistent cuts of the computation that satisfy the predicate. In many cases, the slice obtained will be much smaller than the computation itself enabling us to ignore many consistent cuts in our search.

For example, suppose we wish to compute the slice of a computation with respect to the predicate $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$, where x_i is a boolean variable on process p_i. As explained, we first compute slices for the local predicates x_1, x_2, x_3 and x_4. We then graft the first two and the last two slices together with

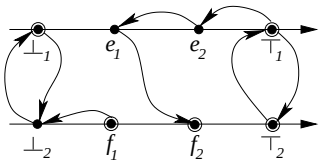

Fig. 5. An optimal algorithm to compute the slice for a conjunctive predicate.

respect to join to obtain slices for the clauses $x_1 \vee x_2$ and $x_3 \vee x_4$, respectively. Finally, we graft the slices for both clauses together with respect to meet to get the slice for the predicate $reg\ (x_1 \vee x_2) \wedge reg\ (x_3 \vee x_4)$ which, in general, is larger than the slice for the predicate $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$ but much smaller than the computation itself.

The result of Section 6.3 allows us to generalize this approach to predicates composed from arbitrary regular predicates using $\neg$, $\wedge$ and $\vee$ operators. We plan to conduct experiments to quantitatively evaluate the effectiveness of our approach. Although our focus is on detecting predicates under *possibly* modality, slicing can be used to prune search space for monitoring predicates under other modalities too.

7 Optimal Algorithm for Slicing

The algorithm we presented in [6] to compute slices for regular predicates has $O(N^2|E|)$ time complexity, where N is the number of processes and E is the set of events. In this section we present an *optimal* algorithm for computing slices for special cases of regular predicates. Our algorithm will have $O(|E|)$ time complexity. Due to lack of space, only the optimal algorithm for conjunctive predicates is presented. The optimal algorithm for other regular predicates such as channel predicates can be found elsewhere [12].

A *conjunctive predicate* is a conjunction of local predicates. For example, "P_1 is in red state" $\wedge$ "P_2 is in green state" $\wedge$ "P_3 is in blue state". Given a set of local predicates, one for each process, we can categorize events on each process into *true events* and *false events*. An event is a true event iff the corresponding local predicate evaluates to true, otherwise it is a false event.

To compute the slice of a computation for a conjunctive predicate, we construct a directed graph with vertices as events in the computation and the following edges: (1) from an event, that is not a final event, to its successor, (2) from a send event to the corresponding receive event, and (3) from the successor of a false event to the false event.

For the purpose of building the graph, we assume that all final events are true events. Thus every false event has a successor. The first two kinds of edges ensure that the Lamport's happened-before relation is captured in the graph. The algorithm is illustrated Fig. 5. In the figure, all true events have been encircled.

It can be proved that the directed graph obtained is cut-equivalent to the slice of the computation with respect to the given conjunctive predicate [4]. It is easy

to see that the graph has $O(|E|)$ vertices, $O(|E|)$ edges (at most three edges per event assuming that an event that is not local either sends at most one message or receives at most one message but not both) and can be built in $O(|E|)$ time. The slice can be computed by finding out the strongly connected components of the graph [4]. Thus the algorithm has $O(|E|)$ overall time complexity. It also gives us an $O(|E|)$ algorithm to evaluate $possibly: b$ when b is a conjunctive predicate (see Theorem 2).

By defining a local predicate (evaluated on an event) to be true iff the event corresponds to a local checkpoint, it can be verified that there is a *zigzag path* [14, 18] from a local checkpoint c to a local checkpoint c' in a computation iff there is a path from $succ(c)$, if it exists, to c' in the corresponding slice—which can be ascertained by comparing $J_b(succ(c))$ and $J_b(c')$. An alternative formulation of the consistency theorem in [14] can thus be obtained as follows:

Theorem 9. *A set of local checkpoints can belong to the same consistent global snapshot iff the local checkpoints in the set are mutually consistent (including with itself) in the corresponding slice.*

Moreover, the R-graph (rollback-dependency graph) [18] is path-equivalent to the slice when each contiguous sequence of false events on a process is merged with the nearest true event that occurs later on the process. The minimum consistent global checkpoint that contains a set of local checkpoints [18] can be computed by taking the set union of J_b's for each local checkpoint in the set. The maximum consistent global checkpoint can be similarly obtained by using the dual of J_b.

8 Conclusion and Future Work

In this paper, the notion of slice introduced in our earlier paper [6] is generalized and its existence for all global predicates is established. The intractability of computing the slice, in general, is also proved. An optimal algorithm to compute slices for special cases of predicates is provided. Moreover, an efficient algorithm to graft two slices is also given. Application of slicing in general and grafting in particular to global property evaluation of distributed programs is discussed. Finally, the results pertaining to consistent global checkpoints [14, 18] are shown to be special cases of computation slicing.

As future work, we plan to study grafting in greater detail. Specifically, we plan to conduct experiments to quantitatively evaluate its effectiveness in weeding out unnecessary consistent cuts from examination during state space search for predicate detection. Another direction for future research is to extend the notion of slicing to include temporal predicates.

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, February 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. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.
5. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.
6. 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, April 2001.
7. V. K. Garg and B. Waldecker. Detection of Unstable Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, May 1991.
8. B. Korel and R. Ferguson. Dynamic Slicing of Distributed Programs. *Applied Mathematics and Computer Science Journal*, 2(2):199–215, 1992.
9. B. Korel and J. Rilling. Application of Dynamic Slicing in Program Debugging. In Mariam Kamkar, editor, *Proceedings of the 3rd International Workshop on Automated Debugging (AADEBUG)*, pages 43–57, Linköping, Sweden, May 1997.
10. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.
11. 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, Portland, Oregon, July 2000.
12. N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. Technical Report TR-PDS-2001-002, The Parallel and Distributed Systems Laboratory, Department of Electrical and Computer Engineering, The University of Texas at Austin, April 2001. Available at http://www.cs.utexas.edu/users/neerajm.
13. 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, April 2001.
14. R. H. B. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.
15. S. D. Stoller and F. Schnieder. Faster Possibility Detection by Combining Two Approaches. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, France, September 1995.
16. 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.
17. 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, Bratislava, Slovak Republic, September 1999.
18. Yi-Min Wang. Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, April 1997.
19. M. Weiser. Programmers Use Slices when Debugging. *Communications of the ACM (CACM)*, 25(7):446–452, 1982.