Software Fault Tolerance of Distributed Programs Using Computation Slicing

Neeraj Mittal
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083-0688, USA
neerajm@utdallas.edu

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

Abstract

Writing correct distributed programs is hard. In spite of extensive testing and debugging, software faults persist even in commercial grade software. Many distributed systems, especially those employed in safety-critical environments, should be able to operate properly even in the presence of software faults. Monitoring the execution of a distributed system, and, on detecting a fault, initiating the appropriate corrective action is an important way to tolerate such faults. This gives rise to the predicate detection problem which involves finding a consistent cut of a distributed computation, if it exists, that satisfies the given global predicate.

Detecting a predicate in a computation is, however, an NP-complete problem. To ameliorate the associated combinatorial explosion problem, we introduce the notion of computation slice in our earlier papers [5, 10]. Intuitively, slice is a concise *representation of those consistent cuts that satisfy a certain condition. To detect a predicate, rather than searching the state-space of the computation, it is much more efficient to search the state-space of the slice. In this paper, we provide efficient algorithms to compute the slice for several classes of predicates. Our experimental results demonstrate that slicing can lead to an* exponential *improvement over existing techniques in terms of time and space.*

Keywords: *predicate detection, testing and debugging, software-fault tolerance, pruning search-space, partial-order methods*

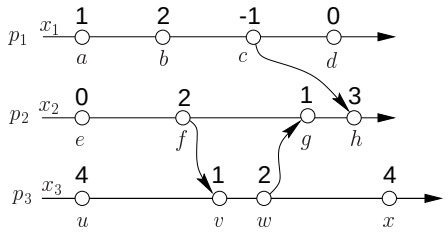
1. Introduction

Writing distributed programs is an error prone activity; it is hard to reason about them because they suffer from the combinatorial explosion problem. Software faults (bugs), in particular global faults, are caused by subtle interactions between various components of the system. As such, they may occur only for specific combinations of inputs and certain interleavings of events. This makes it difficult to eliminate them entirely using testing and debugging. In fact, in spite of extensive testing and debugging, software faults persist even in commercial grade software. Many distributed systems, especially those employed in safety-critical environments, should be able to operate properly even in the presence of software faults. Monitoring the execution of a distributed system, and, on detecting a fault, initiating the appropriate corrective action is an important way to tolerate such bugs.

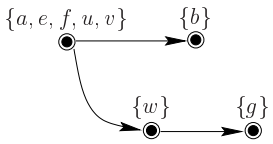
In this paper, we focus on detecting those faults that can be expressed as predicates on variables of processes. For example, "no process has the token" can be written as $no\_token_1 \wedge no\_token_2 \wedge \cdots \wedge no\_token_n$, where no_token_i denotes the absence of token on process p_i. This gives rise to the *predicate detection problem* which involves finding a consistent cut of a distributed computation, if it exists, that satisfies the given global predicate. (This problem is also referred to as detecting a predicate under *possibly* modality in the literature.) Predicate detection problem also arises in other areas in distributed systems such as testing and debugging where it can be used to set conditional breakpoints.

Detecting a predicate in a computation is a hard problem in general [4, 12, 11]. The reason is the combinatorial explosion in the number of possible consistent cuts. Given a computation consisting of n processes each with at most k local states, the number of possible consistent cuts of the computation could be as large as $O(k^n)$. Finding a consistent cut that satisfies the given predicate may, therefore, require looking at a large number of consistent cuts. In fact, we prove in [11] that detecting a predicate in 2-CNF (conjunctive normal form), even when no two clauses contain variables from the same process, is NP-complete, in general. Example of such a predicate is: $(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge \cdots \wedge (x_{n-1} \vee x_n)$, where each x_i

*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.



(a)



◉: meta-event

(b)

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

is a boolean variable on process p_i.

In our earlier papers [5, 10], we introduce the notion of *computation slice*. Intuitively, slice is a succinct representation of those consistent cuts that satisfy a certain condition. The slice of a computation with respect to a predicate is a directed graph with the least number of consistent cuts that contains all consistent cuts of the computation for which the predicate evaluates to true. Since we expect global faults to be relatively rare, their slice will be much smaller than the computation itself. Therefore, in order to detect a global fault, rather than searching the state-space of the computation, it is much more efficient to search the state-space of the slice. We also identify a class of predicates, called *regular predicates*, for which the slice is *exact* [5, 10]. That is, the slice for a regular predicate contains precisely those consistent cuts for which the predicate evaluates to true. In [5], we also present an $O(n^2|E|)$ algorithm to compute the slice for a regular predicate, where n is the number of processes and E is the set of events.

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 Figure 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 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 Figure 1(b). A slice is modeled by a directed graph. Each vertex of the graph corresponds to a *meta-event*, which is 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 prove in [10] that it is, in general, intractable to compute the slice for an arbitrary predicate. Nonetheless, it is still useful to be able to compute an *approximate slice* for such a predicate efficiently. An approximate slice may be bigger than the actual slice but will be much smaller than the computation itself. We also provide a polynomial-time algorithm in [10] to compute an approximate slice for a predicate composed from regular predicates using $\neg$, $\wedge$ and $\vee$ operators.

Our contributions in this paper are as follows. First, we provide a faster algorithm for computing the slice for a *decomposable regular predicate*, which can be expressed as conjunction of clauses where each clause is again a regular predicate but depends on variables of only a small number of processes. For example, consider the regular predicate "counters of all processes are approximately synchronized". It can be expressed as: for all processes p_i and p_j, $|counter_i - counter_j| \leqslant \triangle_{ij}$, where each $counter_i$ is a monotonically non-decreasing variable on p_i. Here, each clause depends on variables of at most two processes. For this predicate, the slicing algorithm given in this paper is n times faster than the one in [5], where n is the number of processes. In general, the time-complexity of the algorithm is given by $O((n + k^2 s)|E|)$, where E is the set of events, k is the maximum number of processes spanned by a clause, and s is the maximum number of clauses any single process appears in. For example, for the aforementioned predicate, $k = 2$ and $s = n$. Second, we give an efficient algorithm to compute the slice for a *k-local predicate* for constant k. A k-local predicate depends on variables of at most k processes [12]. The time-complexity of the algorithm is $O(nm^{k-1}|E|)$, where m is the maximum number of events on a process. Third, we provide an efficient algorithm to compute the slice for a *linear predicate* and its dual *post-linear predicate* [4]. The class of linear

predicates is a generalization of the class of regular predicates. The algorithm has the time-complexity of $O(n^2|E|)$. As a corollary, it is now possible to compute an approximate slice—in polynomial-time—for a much more general class of predicates than described in [10]. Finally, we discuss our experimental results in evaluating the effectiveness of slicing in pruning the search-space for predicate detection. This is the first experimental study of slicing for any application. As such, our objective in this paper is not to measure the efficacy of the slicing algorithms described in this paper only, but to evaluate the effectiveness of slicing algorithms in general including those in our earlier papers [5, 10]. Our results indicate that slicing can lead to an *exponential* improvement over existing techniques in terms of time and space. Furthermore, other techniques for reducing the time-complexity [13] and/or the space-complexity [1] are orthogonal to slicing, and as such can actually be used in conjunction with slicing. For instance, Alagar and Venkatesan's polynomial space algorithm [1] for searching the state-space of a computation can also be used for searching the state-space of a slice.

Although, in this paper, we focus on application of slicing to predicate detection, slicing can be employed to reduce the search-space when monitoring a predicate under other modalities as well including *definitely* [2, 4], *invariant* [10] and *controllable* [4].

The paper is organized as follows. In Section 2, we describe our model and notation. In Section 3, we discuss the background on computation slicing necessary to understand the rest of the paper. Section 4 describes efficient algorithms to compute the slice for three classes of predicates. In Section 5, we discuss our experimental results on slicing. Finally, Section 6 concludes the paper.

2. Model and Notation

Traditionally, a distributed computation is modeled as a partial order on a set of events [8]. In this paper, we relax the restriction that the order on events must be a partial order. Instead we use directed graphs to model distributed computations as well as slices. Directed graphs allow us to handle both of them in a uniform and convenient 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 forms a *consistent cut* if the subset contains a vertex only if it also 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 strongly connected component or none of them. 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. Also, 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 every vertex has a path to itself.

A *distributed computation* (or simply a *computation*) $\langle E, \rightarrow \rangle$ is 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 $\mathcal{P}(\langle E, \rightarrow \rangle)$ contains at least the Lamport's happened-before relation [8]. A distributed computation in our model can contain cycles. This is because, whereas a computation in the traditional or happened-before model captures the *observable* order of execution of events, a computation in our model captures the set of possible consistent cuts. Intuitively, each strongly connected component of a computation can be viewed as a *meta-event*; all events in a meta-event should be executed atomically.

We assume the presence of fictitious initial and final events on each process. The initial event on process p_i, denoted by $\perp_i$, occurs before any other event on p_i. Likewise, the final event on process p_i, denoted by $\top_i$, occurs after all other events on p_i. We assume that all initial events belong to the same strongly connected component. Similarly, 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 the traditional model is a non-trivial consistent cut of the corresponding computation in our model and vice versa. Only non-trivial consistent cuts are of real interest to us.

A *global predicate* (or simply a *predicate*) is defined as a boolean-valued function on variables of processes. Given a consistent cut, a predicate is evaluated with respect to the values of variables resulting after executing all events in the cut. If a predicate b evaluates to true for a consistent cut C, we say that "C satisfies b". We leave the predicate undefined for the trivial consistent cuts. A global predicate is *local* if it depends on variables of a single process. It is said to be k-*local* if it depends on variables of at most k processes [12].

3. Background

The notion of computation slice is based on the Birkhoff's Representation Theorem for Finite Distributive Lattices [3] which we describe next.

3.1. Birkhoff's Theorem

We first describe some concepts needed to understand the theorem. A lattice is said to be *distributive* if its meet operator distributes over its join operator [3]. It can be proved

that meet distributes over join if and only if join distributes over meet. An element of a lattice is called *join-irreducible* if it cannot be expressed as join of two distinct elements (of the lattice), both different from itself [3].

Let L be a lattice and $\mathcal{JI}(L)$ be the set of its join-irreducible elements. In case L is a distributive lattice, it satisfies an important property. Specifically, every element in L can be expressed as the join of some subset of elements in $\mathcal{JI}(L)$ and vice versa [3, Birkhoff's Theorem]. In other words, $\mathcal{JI}(L)$ completely characterizes L. This is significant because $|\mathcal{JI}(L)|$ is generally much smaller—exponentially in many cases—than $|L|$. Hence if some computation on L can instead be performed on $\mathcal{JI}(L)$, we obtain a significant computational advantage.

Consider a computation $\langle E, \rightarrow \rangle$ and let $\mathcal{C}(E)$ denote the set of its consistent cuts. In [5], we show that $\mathcal{C}(E)$ forms a distributive lattice under the relation $\subseteq$; its join and meet operators correspond to set union ($\cup$) and set intersection ($\cap$), respectively. Furthermore, no additional structural property is satisfied by $\mathcal{C}(E)$. There is a one-to-one correspondence between the set of join-irreducible elements of $\mathcal{C}(E)$ and the set of strongly connected components of $\langle E, \rightarrow \rangle$

Now, consider a subset $\mathcal{D} \subseteq \mathcal{C}(E)$. We say that $\mathcal{D}$ forms a *sublattice* of $\mathcal{C}(E)$ if $\mathcal{D}$ is closed under set union and set intersection. That is, given two consistent cuts of $\mathcal{D}$, the consistent cuts obtained by their set union and set intersection also belong to $\mathcal{D}$. It can be proved that a sublattice of a distributive lattice is also a distributive lattice [3]. Thus if $\mathcal{D}$ is a sublattice of $\mathcal{C}(E)$, then, using Birkhoff's Theorem, $\mathcal{JI}(\mathcal{D})$ completely characterizes $\mathcal{D}$. This forms the basis for the notion of computation slice.

3.2. Computation Slice

Roughly speaking, a computation slice (or simply slice) is a succinct representation of all those consistent cuts of the computation that satisfy a certain predicate. More precisely,

Definition 1 (slice [10]) *A* slice *of a computation with respect to a predicate is the smallest directed graph (with the least number of consistent cuts) that contains all consistent cuts of the given computation for which the predicate evaluates to true.*

We denote the slice of a computation $\langle E, \rightarrow \rangle$ with respect to a predicate b by $\langle E, \rightarrow \rangle_b$. We prove in [10] that the slice exists and is uniquely defined for all predicates. The main idea behind the proof is as follows. Consider a computation $\langle E, \rightarrow \rangle$ and a predicate b. Let $\mathcal{C}(E)$ denote the set of consistent cuts of $\langle E, \rightarrow \rangle$ and, further, let $\mathcal{C}_b(E) \subseteq \mathcal{C}(E)$ be the subset of those consistent cuts that satisfy b. We show that there exists a subset $\mathcal{D} \subseteq \mathcal{C}(E)$ satisfying the following conditions. First, $\mathcal{D}$ contains $\mathcal{C}_b(E)$, that is, $\mathcal{C}_b(E) \subseteq \mathcal{D}$. Second, D forms a sublattice of $\mathcal{C}(E)$. Last, among all sublattices that fulfill the first two conditions, it is the *smallest* one. From Birkhoff's Theorem, $\mathcal{JI}(\mathcal{D})$, the set of join-irreducible elements of $\mathcal{D}$, completely characterizes $\mathcal{D}$. We call the poset (partially ordered set) induced on the consistent cuts of $\mathcal{JI}(\mathcal{D})$ by the relation $\subseteq$ as the slice $\langle E, \rightarrow \rangle_b$. Alternatively, the slice can also be represented by a directed graph drawn on the set of events E such that its set of consistent cuts is given by $\mathcal{D}$. Whereas the *poset representation* of a slice is better for presentation purposes, the *graph representation* is more suited for slicing algorithms.

Every slice derived from the computation $\langle E, \rightarrow \rangle$ has the trivial consistent cuts ($\emptyset$ and E) among its set of consistent cuts. A slice is *empty* if it has no non-trivial consistent cuts [10]. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. In general, a slice will contain consistent cuts that do not satisfy the predicate (besides trivial consistent cuts). In case a slice does not contain any such cut, it is called *lean* [10]. We now describe the class of predicates for which the slice is lean.

3.3. Regular Predicate

Informally, if a predicate is regular, then the set of consistent cuts satisfying the predicate is closed under set intersection and set union [5]. Equivalently,

Definition 2 (regular predicate [5]) *A predicate is said to be* regular *if, given two consistent cuts that satisfy the predicate, the consistent cuts obtained by their set intersection and set union also satisfy the predicate. Formally, given a regular predicate b, and consistent cuts C and D,*

$$(C \textit{ satisfies } b) \wedge (D \textit{ satisfies } b) \Rightarrow (C \cap D \textit{ satisfies } b) \wedge (C \cup D \textit{ satisfies } b)$$

Some examples of regular predicates are: conjunction of local predicates such as "all processes are in *red* state", and certain channel predicates including "at most k messages are in transit from process p_i to process p_j" and "at least k messages are in transit from process p_i to process p_j". We prove in [5] that the conjunction of two regular predicates is also a regular predicate.

In [5, 10], we provide efficient algorithms to compute the slice for a regular predicate and its complement—referred to as a *co-regular predicate*. Time-complexities of the two algorithms are $O(n^2|E|)$ and $O(n^2|E|^2)$, respectively, where n is the number of processes and E is the set of events. We also give optimal $O(|E|)$ algorithms to compute the slice for special cases of regular predicates in [10].

3.4. Grafting Two Slices

Grafting is used to compose two slices [10]. Specifically, given two slices, grafting involves computing either (1) the

smallest slice that contains all consistent cuts common to both the slices, or (2) the smallest slice that contains all consistent cuts that belong to at least one of the slices. The former is referred to as grafting with respect to conjunction, and the latter as grafting with respect to disjunction. Intuitively, given slices for two regular predicates b_1 and b_2, the former can be used to compute the slice for $b_1 \wedge b_2$ and the latter for $b_1 \vee b_2$. In [10], we give $O(n|E|)$ algorithms to graft two slices.

4. Computing the Slice

In this section, we provide efficient algorithms for computing the slice for three classes of predicates. The first class contains those predicates which can be expressed as conjunction of k-local regular predicates for a small k; they are referred to as *decomposable regular predicates*. Note that although each clause of a decomposable regular predicate depends on variables of only a small number of processes, the predicate itself may span a much larger set of processes, possibly even the entire set. The second class consists of *k-local predicates* for constant k. This class may contain predicates that are not regular. Finally, the third class consists of *linear predicates* and their dual.

4.1. Fast Algorithm for Computing the Slice for Decomposable Regular Predicate

In [5], we provide an $O(n^2|E|)$ algorithm to compute the slice for a regular predicate, where n is the number of processes and E is the set of events. We now explore the possibility of a faster algorithm for the special case when a regular predicate can be expressed as a conjunction of clauses where each clause is again a regular predicate but spans a small number of processes. An example of such a predicate is "counters of all processes are approximately synchronized", which can be expressed as:

$$\bigwedge_{1 \leqslant i,j \leqslant n} (|counter_i - counter_j| \leqslant \triangle_{ij})$$

where each $counter_i$ is a monotonically non-decreasing variable on process p_i. In this case, each clause depends on variables of at most two processes and is therefore 2-local. For this example, our slicing algorithm has $O(n|E|)$ time-complexity, which is a factor of n less than that of the algorithm in [5]. We describe the algorithm in two steps. First, we present a fast algorithm to compute the slice for each clause. Second, we show how to combine these slices together in an efficient manner to obtain the slice for the given regular predicate. Due to lack of space, we only provide a brief description of each step. For details of the algorithm and its proof of correctness, please refer to [9].

In the first step, for each clause, we first take the *projection* of the computation on those processes whose variables the clause depends on. We then compute the slice of the projected computation using the algorithm described in [5]. We show that *combined* time-complexity of computing the slice of the projected computation for all clauses is given by $O((n+k^2s)|E|)$, where s is the maximum number of clauses any single process appears in. For instance, for the example in the previous paragraph, s is n. (The time-complexity includes the time it takes to compute the projection as well.) In the second step, we combine these slices together efficiently in $O((n + ks)|E|)$ time to obtain the desired slice. The overall time-complexity of the two steps is given by $O((n + k^2s)|E|)$. In case k is $O(1)$ and s is $O(n)$, as is the case with our example, the time-complexity is $O(n|E|)$, which is a factor of n less than computing the slice directly using the algorithm in [11].

4.2. Computing the Slice for k-Local Predicate, for Constant k

In case the predicate is regular, we can simply use the approach described in the previous section to compute the slice in $O(n|E|)$ time. However, if the predicate is not regular, then the slice produced will only be an approximate one [9]. To compute the (exact) slice for a k-local predicate, which is not regular, we use the technique developed by Stoller and Schneider [12]. For a given computation, their technique can be used to transform a k-local predicate into a predicate in k-DNF (disjunctive normal form) with at most m^{k-1} clauses, where m is the maximum number of events on a process. For example, consider the predicate $x_1 \neq x_2$. Let V denote the set of values that x_1 can take in the given computation. Then $x_1 \neq x_2$ can be rewritten as:

$$x_1 \neq x_2 \;\equiv\; \bigvee_{v \in V} \Big((x_1 = v) \wedge (x_2 \neq v) \Big)$$

Note that $|V| \leqslant m$. Thus the resultant predicate, in the above case, consists of m clauses; each clause is a conjunction of local predicates, also referred to as a *conjunctive predicate* [4]. In general, the resultant k-DNF predicate will consist of m^{k-1} clauses. To compute the slice for each clause, we use the optimal $O(|E|)$ algorithm given in [10]. We then graft these slices together with respect to disjunction to obtain the slice for the given predicate. The overall time-complexity of the algorithm is given by $O(nm^{k-1}|E|)$.

4.3. Computing the Slice for Linear Predicate

A predicate is said to be *linear* if the set of consistent cuts that satisfy the predicate is closed with respect to set intersection [4]. A *post-linear predicate* can be defined dually. Evidently, the class of linear predicates includes the class of regular predicates. But the converse does not hold.

An example of a linear predicate that, in general, is not regular is: "at most k messages destined for process p_i have not been received yet". The class of linear predicates is also closed under conjunction [4].

We identify and prove that the algorithm to compute the slice for a regular predicate, given in [5], is applicable even for a linear predicate. The main idea behind the algorithm is as follows. Consider a linear predicate b. For each event e, we define $J_b(e)$ as the *least consistent cut* that contains e and satisfies b [5]. In case no consistent cut containing e that also satisfies b exists or when $e \in \top$, $J_b(e)$ is set to E. It can be proved that J_b is uniquely defined for all events [9]. In [9], we give an $O(n^2|E|)$ algorithm to compute $J_b(e)$ for each event e. We also establish in [9] that the slice for b can be obtained by constructing a graph with vertices as the set of events, and an edge from an event e to an event f if $J_b(e) \subseteq J_b(f)$. Further, we show a way to restrict the number of edges in the graph to $O(n|E|)$ without adversely affecting the time-complexity.

5. Computing an Approximate Slice

It is, in general, intractable to compute the slice for a predicate [10]. In fact, computing the slice for a predicate in 2-CNF, even when no two clauses contain variables from the same process, is an NP-complete problem. Nonetheless, it is still useful to be able to compute an *approximate slice* for such a predicate efficiently. An approximate slice may be bigger than the actual slice but may be much smaller than the computation itself.

In [10], we give an efficient polynomial-time algorithm to compute an approximate slice for a predicate composed from regular predicates using $\neg$, $\wedge$ and $\vee$ operators. Utilizing results of this paper, it is now possible to compute an approximate slice, in polynomial-time, for a much larger class of predicates. Specifically, we can now compute an approximate slice for a predicate composed from co-regular predicates, linear predicates, post-linear predicates, and k-local predicates for constant k, using $\wedge$ and $\vee$ operators.

To compute an approximate slice for such a predicate, we first construct the parse tree for the corresponding boolean expression; all predicates occupy leaf nodes whereas all operators occupy non-leaf nodes. We then recursively compute the slice by starting with leaf nodes and moving up, level by level, until we reach the root. For a leaf node, we use the slicing algorithm appropriate for the predicate contained in the node. For example, if the leaf node contains a linear predicate, we use the algorithm described in Section 4.3. For a non-leaf node, we use the suitable grafting algorithm depending on the operator.

As an illustration, suppose we wish to compute an approximate slice of a computation with respect to the predicate $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$, where each x_i is a linear predicate. We first compute slices for the linear predicates x_1, x_2, x_3 and x_4. We then graft the first two and the last two slices together with respect to disjunction to obtain (approximate) 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 conjunction to obtain an approximate slice for $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$.

The next section describes our experimental results in evaluating the efficacy of slicing in pruning the search-space for predicate detection.

5.1. Experimental Evaluation

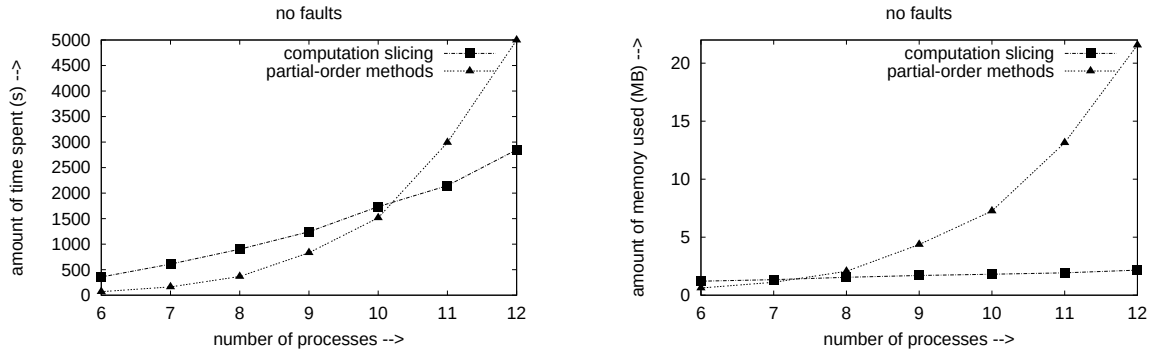
This is the first experimental study of slicing for any application. Therefore our aim is to evaluate the effectiveness of slicing algorithms in general including those in our earlier papers [5, 10] and not just the ones described in this paper.

We compare our approach with Stoller, Unnikrishnan and Liu's approach [13] which is based on *partial-order methods* [6]. Intuitively, when searching the state-space, at each consistent cut, partial-order methods allow only a small subset of enabled transitions to be explored. In particular, we use partial-order methods employing both persistent and sleep sets for comparison. We consider two examples that are also used by Stoller, Unnikrishnan and Liu to evaluate their approach [13].

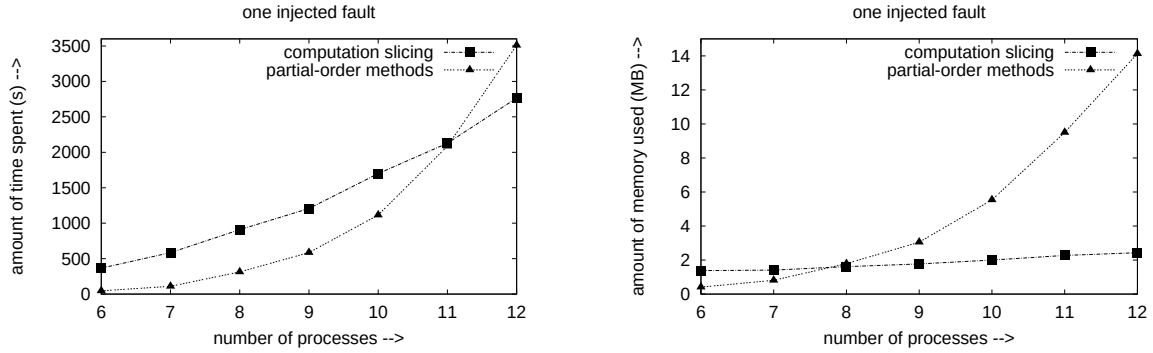
The first example, called *primary-secondary*, concerns an algorithm designed to ensure that the system always contains a pair of processes acting together as primary and secondary. The invariant for the algorithm requires that there be a pair of processes p_i and p_j such that (1) p_i is acting as a primary and correctly thinks that p_j is its secondary, and (2) p_j is acting as a secondary and correctly thinks that p_i is its primary. Both the primary and the secondary may choose new processes as their successor at any time; the algorithm must ensure that the invariant is never falsified. A global fault, therefore, corresponds to the complement of the invariant which can be expressed as:

$$\neg I_{ps} = \bigwedge_{i,j \in [1 \ldots n],\ i \neq j} \Big(\neg isPrimary_i \vee \neg isSecondary_j \vee (secondary_i \neq p_j) \vee (primary_j \neq p_i) \Big)$$

Note that $\neg I_{ps}$ is a predicate in CNF where each clause is a disjunction of two local predicates. An approximate slice for $\neg I_{ps}$ can be computed in $O(n^3|E|)$ time. In the second example, called *database partitioning*, a database is partitioned among processes p_2 through p_n, while process p_1 assigns tasks to these processes based on the current partition. A process p_i, $i \in [2 \ldots n]$, can suggest a new partition at any time by setting variable $change_i$ to true and then broadcasting a message containing the proposed partition. An invariant that should be maintained is: if no process is



(a)



(b)

Figure 2. Primary-Secondary example with the number of events on a process upper-bounded by 90 for (a) no faults and (b) one injected fault.

changing the partition, then all processes agree on the partition. Its complement, corresponding to a global fault, can be expressed as:

$$\neg I_{db} = \neg change_2 \wedge \neg change_3 \wedge \cdots \wedge \neg change_n \wedge \Big(\bigvee_{i,j \in [1 \ldots n],\, i \neq j} (partition_i \neq partition_j) \Big)$$

Note that the first $n-1$ clauses of $\neg I_{db}$ are local predicates and the last clause, say LC, is a disjunction of 2-local predicates. Thus, using the technique described in Section 4.2, LC can be rewritten as a predicate in DNF with $O(n|E|)$ clauses. To reduce the number of clauses, we proceed as follows. Let V denote the set of values that $partition_1$ assumes in the given computation. Then it can be verified that LC is logically equivalent to:

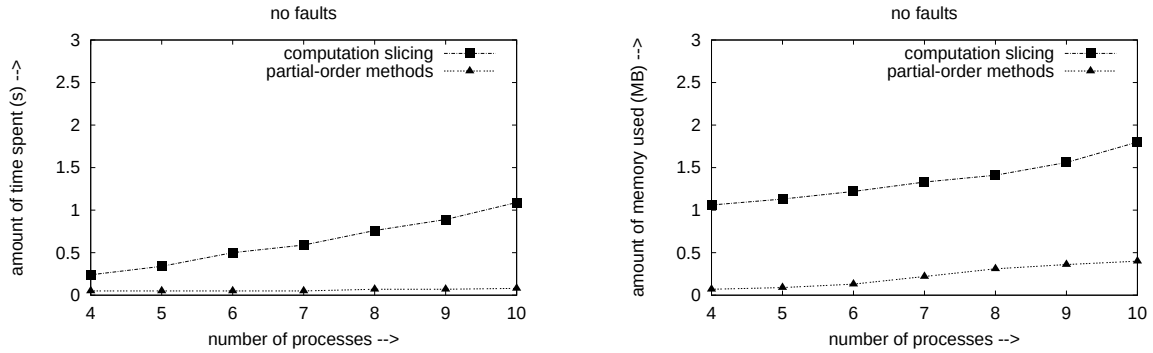
$$\bigvee_{v \in V} \Big((partition_1 = v) \wedge \Big(\bigvee_{2 \leqslant i \leqslant n} (partition_i \neq v) \Big) \Big)$$

This decreases the number of clauses, when LC is rewritten in a form suitable for slice computation, to $O(n|V|)$. Note that $|V|$ is bounded by the number of events on the first process, and therefore we expect $n|V|$ to be $O(|E|)$. As a result, the number of clauses reduces by a factor of n.

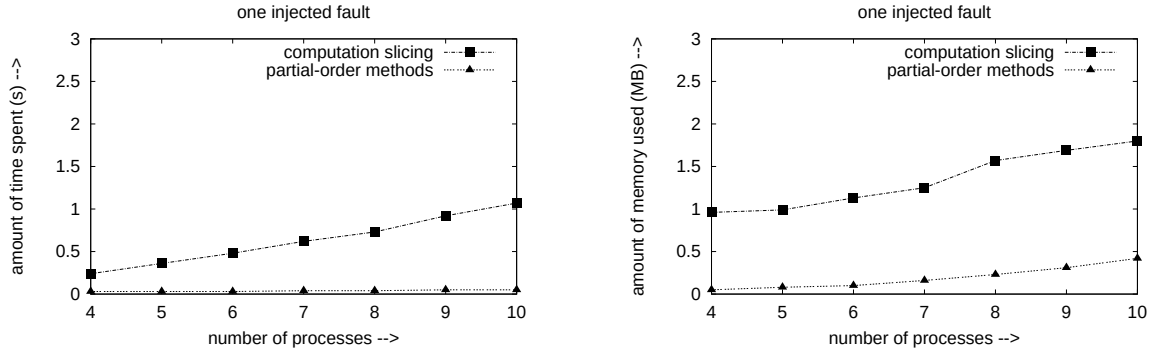
We use the simulator implemented in Java by Stoller, Unnikrishnan and Liu to generate computations of these protocols. Further details of the two protocols and the simulator can be found elsewhere [13]. We consider two different scenarios: *fault-free* and *faulty*. The simulator always produces fault-free computations. A faulty computation is generated by randomly injecting faults into a fault-free computation. Note that in the first (fault-free) scenario, we know *a priori* that the computation does not contain a faulty consistent cut. We cannot, however, assume the availability of such knowledge in general. Thus it is important to study the behaviour of the two predicate detection techniques in the fault-free scenario as well.

Algorithms for slicing a computation are implemented in Java. We compare the two predicate detection techniques with respect to the following metrics: amount of time spent and amount of memory used. In case of the former technique, both metrics also include the overhead of computing the slice. We run our experiments on a machine with Pentium 4 processor operating at 1.8GHz clock frequency and 512MB of physical memory.

For primary-secondary example, the simulator is run un-



(a)



(b)

Figure 3. Database partitioning example with the number of events on a process upper-bounded by 80 for (c) no faults and (d) one injected fault.

til the number of events on some process reaches 90. With computation slicing, for fault-free computations, the slice is always empty. As the number of processes is increased from 6 to 12, the amount of time spent increases from 356s to 2,849s, whereas the amount of memory used increases from 1.21M to 2.16M. On the other hand, with partial-order methods, they increase, almost exponentially, from 69s to 4,997s and 0.62M to 21.56M, respectively. Even on injecting a fault, the slice stays quite small. After computing the slice, in our experiments, we only need to examine at the most 13 consistent cuts to locate a faulty consistent cut, if any. The amount of time spent and the amount of memory used, with computation slicing, increase from 366s to 2,765s and 1.38M to 2.43M, respectively, as the number of processes is increased from 6 to 12. However, with partial-order methods, they again increase almost exponentially from 46s to 3,510s and 0.41M to 14.13M, respectively. Clearly, with slicing, both time and space complexities for detecting a global fault, if it exists, in primary-secondary example are polynomial in input size for the specified range of parameters. In contrast, with partial-order methods, they are exponential in input size. Figure 2(a) and Figure 2(b) plot the variation in the two metrics with the number of processes for the two approaches.

The worst-case performance of the partial-order methods approach is quite bad. With 12 processes in the system and the limit on the memory set to 100MB, the approach runs out of memory in approximately 6% of the cases. In around two-thirds of such cases, the computation actually contains a consistent cut that does not satisfy the invariant. It may be noted that we do not include the above-mentioned cases in computing the average amount of time spent and memory used. Including them will only make the average performance of the partial-order methods approach worse. Further, performance of the partial-order methods approach appears to be very sensitive to the location of the fault, in particular, whether it occurs earlier during the search or much later or perhaps does not occur at all. Consequently, the variation or standard deviation in the two metrics is very large. This has implications when predicate detection is employed for achieving software fault tolerance. Specifically, it becomes hard to provision resources (in our case, memory) when using partial-order methods approach. If too little memory is reserved, then, in many cases, the predicate detection algorithm will not be able to run successfully to completion. On the other hand, if too much memory is re-

served, the memory utilization will be sub-optimal.

For database partitioning example, the simulator is run until the number of events on some process reaches 80. Figure 3(c) and Figure 3(d) plot the variation in the two metrics with the number of processes for the two approaches. As it can be seen, the average performance of partial-order methods is much better than computation slicing. This is because substantial overhead is incurred in computing the slice. The slice itself is quite small. Specifically, for the fault-free scenario, the slice is always empty. On the other hand, for the faulty scenario, only at most 4 transitions need to be explored after computing the slice to locate a faulty consistent cut, if any.

Even for database partitioning example, for 10 processes, the partial-order methods approach runs out of memory in a small fraction—approximately 1%—of the cases. Therefore the worst-case performance of computation slicing is better than partial-order methods. To get the best of both worlds, predicate detection can be first done using the partial-order methods approach. In case it turns out that the approach is using too much memory, say more than $cn|E|$ for some small constant c, and still has not terminated, it can be aborted and the computation slicing approach can then be used for predicate detection.

6. Conclusion and Future Work

In this paper, we present efficient algorithms for computing the slice for several classes of predicates. In addition, we also provide heuristics to compute an approximate slice for predicates for which it is otherwise intractable to compute the actual slice. We experimentally demonstrate that slicing can indeed be used to prune a large fraction of the set of consistent cuts in an efficient manner.

At present, our algorithms for computing a slice and therefore for detecting a predicate work in an off-line fashion. In the future, we plan to develop slicing algorithms that are incremental in nature. As the execution of the system progresses and more and more events become available, the current slice is updated to reflect the newly generated events.

References

[1] S. Alagar and S. Venkatesan. Techniques to Tackle State Explosion in Global Predicate Detection. *IEEE Transactions on Software Engineering*, 27(8):704–714, Aug. 2001.

[2] 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.

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

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

[5] 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, Apr. 2001.

[6] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[7] R. Jegou, R. Medina, and L. Nourine. Linear Space Algorithm for On-line Detection of Global Predicates. In J. Desel, editor, *Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT)*, pages 175–189. Springer-Verlag, 1995.

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

[9] N. Mittal. *Techniques for Analyzing Distributed Computations*. PhD thesis, The University of Texas at Austin, May 2002. Available at "http://www.utdallas.edu/~neerajm".

[10] N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 78–92, Lisbon, Portugal, Oct. 2001.

[11] 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, Apr. 2001.

[12] S. D. Stoller and F. Schneider. Faster Possibility Detection by Combining Two Approaches. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, volume 972 of *Lecture Notes in Computer Science (LNCS)*, pages 318–332, France, Sept. 1995.

[13] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science (LNCS)*, pages 264–279. Springer-Verlag, July 2000.