A Distributed Abstraction Algorithm for Online Predicate Detection

Himanshu Chauhan [1] Vijay K. Garg [1] Aravind Natarajan [2] Neeraj Mittal [2]

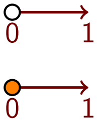
[1]Parallel & Distributed Systems Lab,
Department of Electrical & Computer Engineering
University of Texas at Austin

[2]Department of Computer Science,
University of Texas at Dallas

Outline

Why Online Predicate Detection?

- Large Parallel Computations
  - Non-terminating executions, e.g. server farms
  - Debugging, Runtime validation

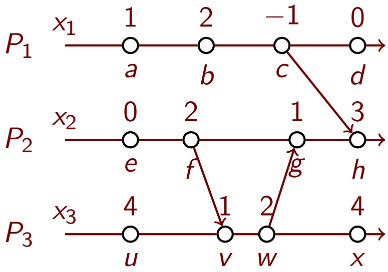


Other Applications

- General predicate detection algorithms, such as Cooper-Marzullo [1991]
  - Perform abstraction with respect to simpler predicate
  - Detect remaining conjunct in the abstracted structure
  - Reduced complexity by using abstraction based detection

Predicate Detection in Distributed Computations

Find all global states in a computation that satisfy a predicate

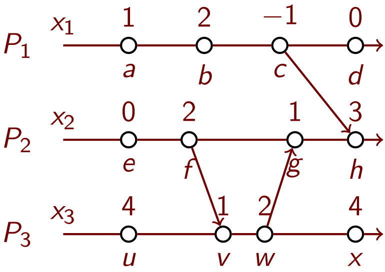


Predicate $(x_1 * x_2 + x_3 < 5) \wedge (x_1 \geq 1) \wedge (x_3 \leq 3)$: $O(k^3)$ steps

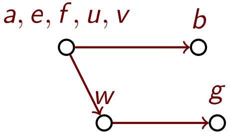
- $O(k^n)$ complexity for n processes, and k events per process
- Compute intensive for large computations

Exploiting Predicate Structure Using Abstractions

Predicate $(x_1 * x_2 + x_3 < 5) \wedge (x_1 \geq 1) \wedge (x_3 \leq 3)$



(a) Original Computation



(b) Slice w.r.t. $(x_1 \geq 1) \wedge (x_3 \leq 3)$

Paper Focus

- *Offline* and *Online* algorithms for abstracting computations for *regular* predicates **exist** [Mittal et al. 01 & Sen et al. 03]

- **This paper**: Efficient **distributed** *online* algorithm to abstract a computation with respect to *regular* predicates.

System Model

- Asynchronous message passing
- n reliable processes
- FIFO, loss-less channels
- Denote a distributed computation with $(E, \rightarrow)$
  - E: Set of all events in the computation
  - $\rightarrow$: *happened-before* relation [Lamport 78]

Consistent Cuts

Consistent Cut: Possible global state of the system during its execution.

Consistent Cuts

Consistent Cut: Possible global state of the system during its execution.

Formally:

Given a distributed computation $(E, \rightarrow)$, a subset of events $C \subseteq E$ is a consistent cut if C contains an event e only if it contains all events that happened-before e.

$$e \in C \wedge f \rightarrow e \Rightarrow f \in C$$

Consistent Cuts

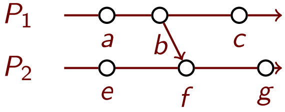
Consistent Cut: Possible global state of the system during its execution.

i.e. if a message receipt event has *happened*, the corresponding message send event must have happened.

Consistent Cuts

Consistent Cut: Possible global state of the system during its execution.

For conciseness, we represent a consistent cut by its maximum elements on each process.



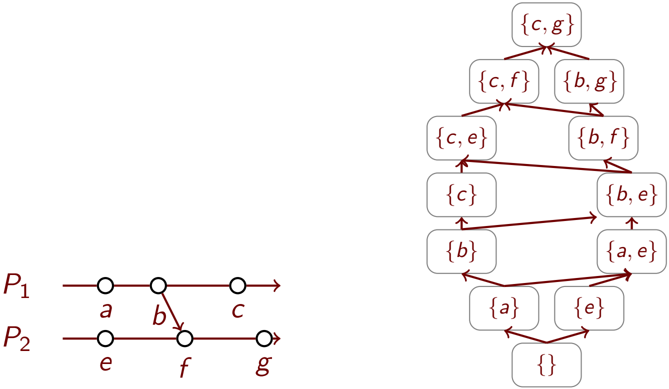
$\{\}$	✓
$\{a\}$	✓
$[b, e]$	✓
$[a, f]$	X

Use vector clocks for checking consistency/finding causal dependency

Lattice of Consistent Cuts

Set of all consistent cuts of a computation $(E, \rightarrow)$, forms a lattice under the relation $\subseteq$. [Mattern 89]

Lattice of Consistent Cuts



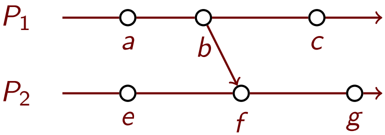
Computation and its Lattice of Consistent Cuts

Regular Predicates

A predicate is *regular* if for any two consistent cuts C and D that satisfy the predicate, the consistent cuts given by $(C \cup D)$ and $(C \cap D)$ also satisfy the predicate.

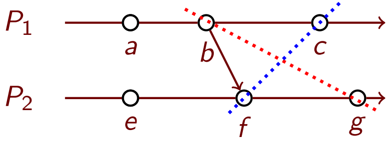
Regular Predicates

A predicate is *regular* if for any two consistent cuts C and D that satisfy the predicate, the consistent cuts given by $(C \cup D)$ and $(C \cap D)$ also satisfy the predicate.



Regular Predicates

A predicate is *regular* if for any two consistent cuts C and D that satisfy the predicate, the consistent cuts given by $(C \cup D)$ and $(C \cap D)$ also satisfy the predicate.



$$\{b, g\} \cap \{c, f\} = \{b, f\},$$
$$\{b, g\} \cup \{c, f\} = \{c, g\}$$

Regular Predicates - Examples

- Local Predicates
- Conjunctive Predicates – conjunctions of local predicates
- Monotonic Channel Predicates
  - All channels are empty/full
  - There are at most m messages in transit from P_i to P_j

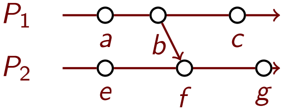
Regular Predicates - Examples

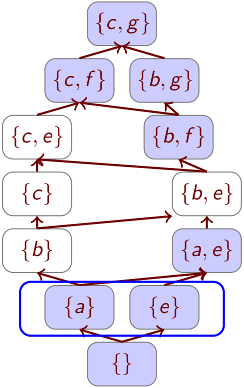
- Local Predicates
- Conjunctive Predicates – conjunctions of local predicates
- Monotonic Channel Predicates
  - All channels are empty/full
  - There are at most m messages in transit from P_i to P_j

Not Regular: There are *even* number of messages in a channel

Regular Predicates

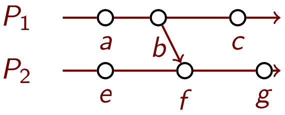
Predicate: "all channels are empty"

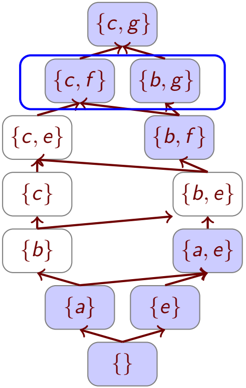




Regular Predicates

Predicate: "all channels are empty"





Why use Abstractions?

Goal: Find all global states that satisfy a given predicate.

Key Benefit of Abstraction

When B is regular: we can "get away" with only enumerating cuts that satisfy B, and are **not** joins of other consistent cuts.

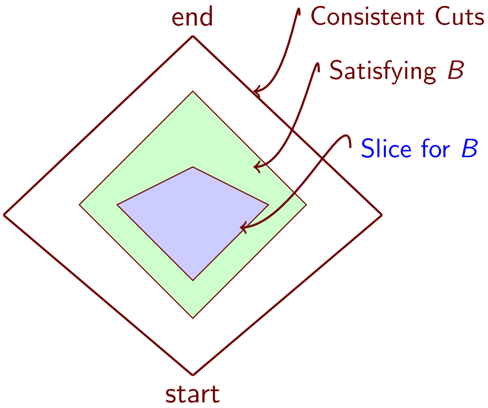
Due to Birkhoff's Representation Theorem for Lattices [Birkhoff 37]

Abstractions for Regular Predicates

Slice: A subset of the set of all global states of a computation that satisfies the predicate.

Abstractions for Regular Predicates

Slice: A subset of the set of all global states of a computation that satisfies the predicate.

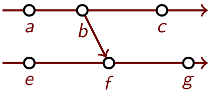


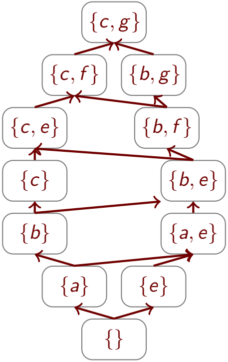
Abstractions for Regular Predicates

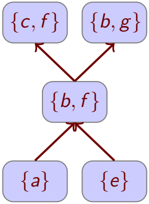
Slice: A subset of the set of all global states of a computation that satisfies the predicate.

Abstractions for Regular Predicates

Slice: A subset of the set of all global states of a computation that satisfies the predicate.







B: "all channels are empty"

How do we do that?

Exploit $J_B(e)$

How do we do that?

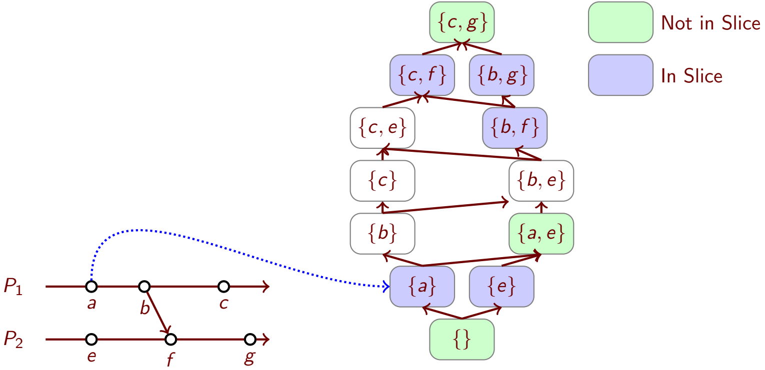
Given a predicate B, and event e in a computation

$J_B(e)$: The least consistent cut that satisfies B and contains e.

How do we do that?

Given a predicate B, and event e in a computation

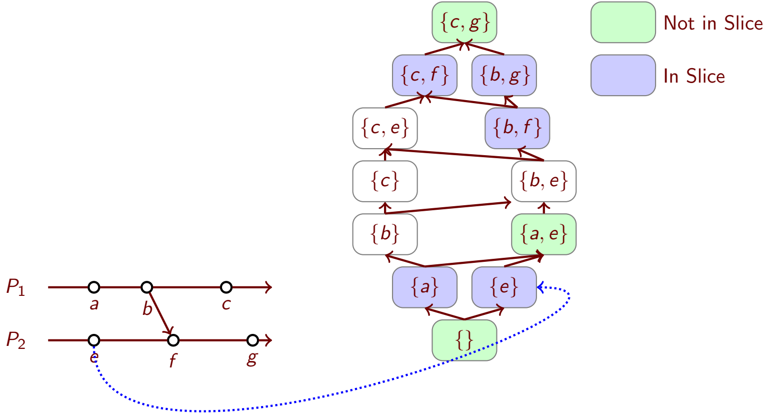
$J_B(e)$: The least consistent cut that satisfies B and contains e.



How do we do that?

Given a predicate B, and event e in a computation

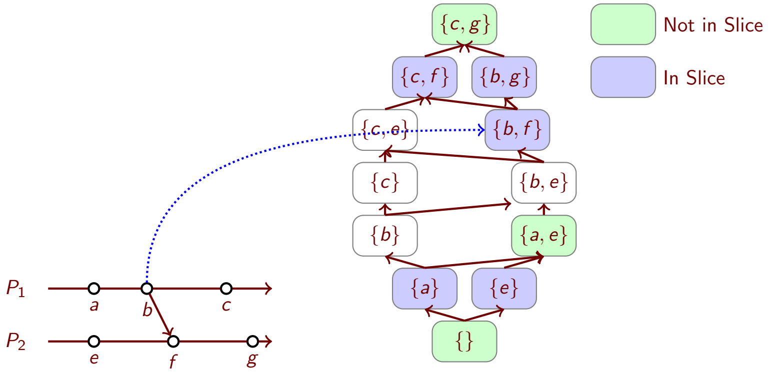
$J_B(e)$: The least consistent cut that satisfies B and contains e.



How do we do that?

Given a predicate B, and event e in a computation

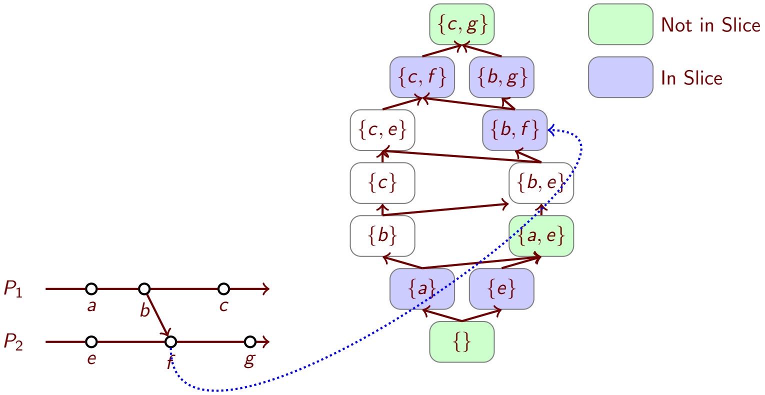
$J_B(e)$: The least consistent cut that satisfies B and contains e.



How do we do that?

Given a predicate B, and event e in a computation

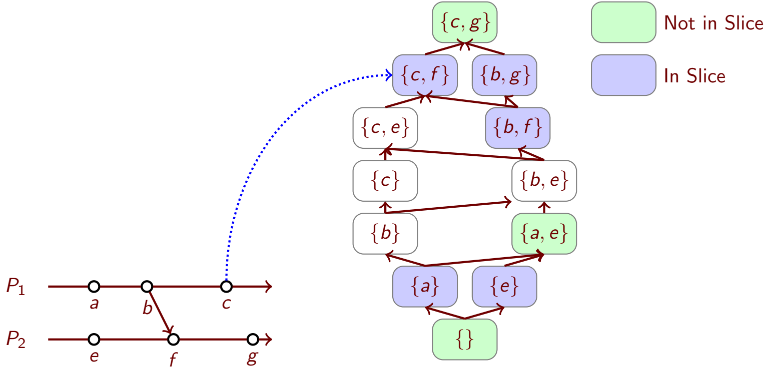
$J_B(e)$: The least consistent cut that satisfies B and contains e.



How do we do that?

Given a predicate B, and event e in a computation

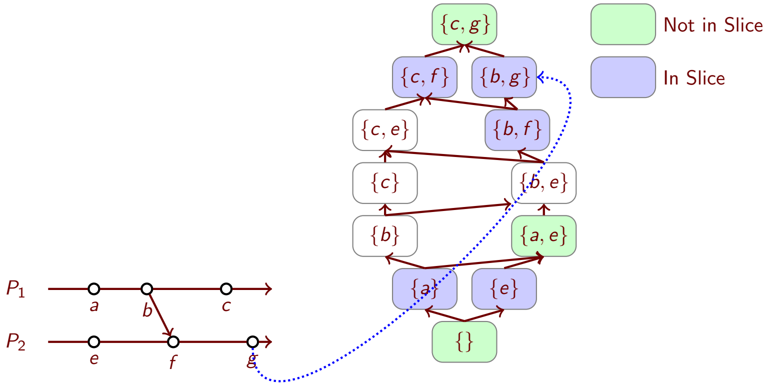
$J_B(e)$: The least consistent cut that satisfies B and contains e.



How do we do that?

Given a predicate B, and event e in a computation

$J_B(e)$: The least consistent cut that satisfies B and contains e.



Slice for Regular Predicates

For a computation $(E, \rightarrow)$, and regular predicate B

Slice for B is defined as:

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

Bored with definitions?

- Enough with the definitions
- Enough with notation
- Just tell us the crux of it

Bored with definitions?

It comes down to a two line pseudo-code

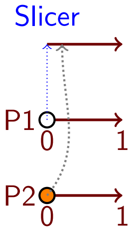
foreach `event` e `in computation:`

`find the least consistent cut that satisfies` B
`and includes` e

Centralized Online Slicing

- One process acts as the central *slicer* - *CS*
- Each process P_i sends details (state/vector clock etc.) of relevant events to *CS*

[Mittal et al. 07]

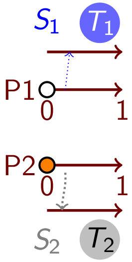


Challenges

- Simple decomposition of *centralized* algorithm into n independent executions is inefficient
- Results in large number of redundant communications
- Multiple computations lead to identical results

Distributed Online Slicing

- Each process P_i has an additional *slicer* thread S_i
- P_i sends details (state/vector clock etc.) of relevant events **locally** to S_i



Distributed Algorithm at S_i

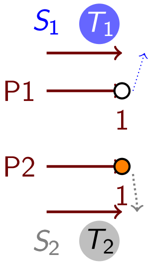
- Each *slicer*, S_i, has a **token**, T_i, that computes $J_B(e)$ where $e \in E_i$
- Tokens are sent to other *slicers* to progress on $J_B(e)$

For each event make use of:

$$e \rightarrow f \Rightarrow J_B(e) \subseteq J_B(f)$$

Distributed Algorithm at S_i

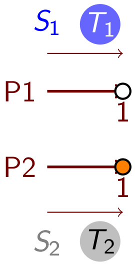
B = "all channels are empty"



	T_1 @ S_1	T_2 @ S_2
e	$P_1.1$	$P_2.1$
cut	$[1, 0]$	$[0, 1]$
dependency	$[1, 0]$	$[0, 1]$
cut consistent?	✓	✓
satisfies B?	✓	✓
output cut?	✓	✓
wait for	$P_1.2$	$P_2.2$

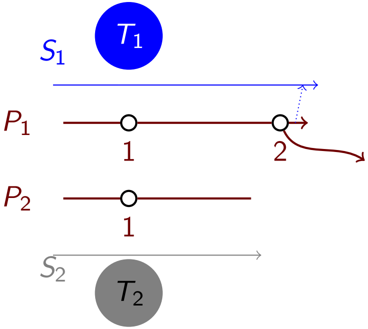
What happens in non-trivial cases?

$B=$ "all channels are empty"



What happens in non-trivial cases?

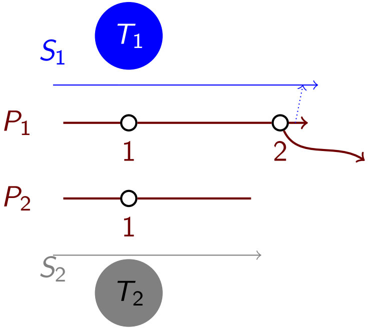
$B=$ "all channels are empty"



Suppose, P_1 just reported its 2^{nd} event to S_1

What happens in non-trivial cases?

$B =$ "all channels are empty"



Suppose, P_1 just reported its 2^{nd} event to S_1

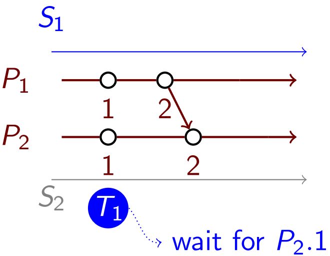
	T_1 @ S_1
e	$P_1.2$
cut	$[2, 0]$
dependency	$[2, 0]$
cut consistent?	✓
satisfies B?	✗
wait for	$P_2.1$

send T_1 to S_2

S_2 receives T_1

Regular predicate structure

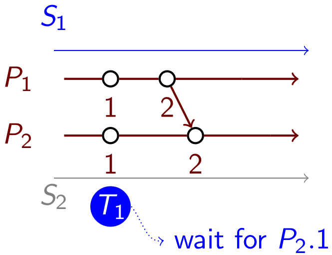
- Exact knowledge of which event to wait for
- Which states to evaluate predicate on



S_2 receives T_1

Regular predicate structure

- Exact knowledge of which event to wait for
- Which states to evaluate predicate on

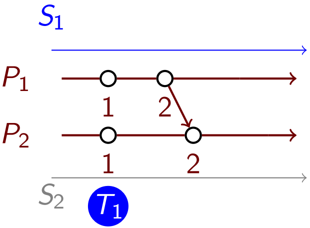


B would not be even evaluated on any state unless S_2 is told about a message ‘receipt’

S_2 receives T_1

Regular predicate structure

- Exact knowledge of which event to wait for
- Which states to evaluate predicate on

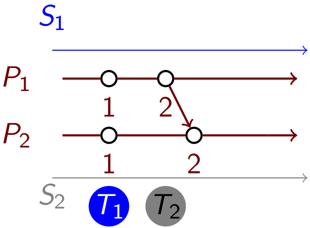


B would not be even evaluated on any state unless S_2 is told about a message 'receipt'

T_1 would wait at S_2 till $P_2.2$ is reported

$P_2.2$ is reported to S_2

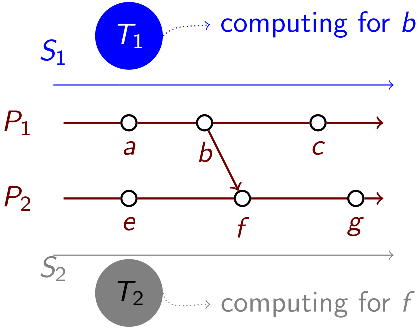
After $P_2.2$ is reported to S_2



	T_1 @ S_2	T_2 @ S_2
e	$P_1.2$	$P_2.2$
cut	[2, 2]	[2, 2]
dependency	[2, 2]	[2, 2]
cut consistent?	✓	✓
satisfies *B*?	✓	✓
output cut?	✓	✓
wait for	$P_1.3$	$P_2.3$

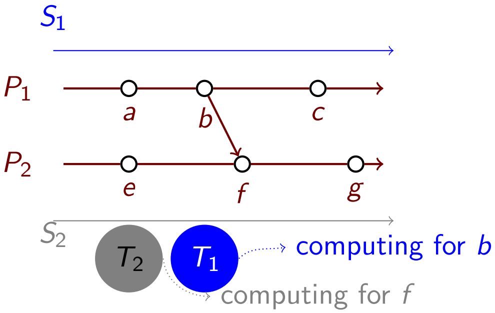
S_2 sends T_1 back to S_1

Optimizations - I



Send only if needed - ie. before sending your token to S_k, check if you have token T_k containing the required information.

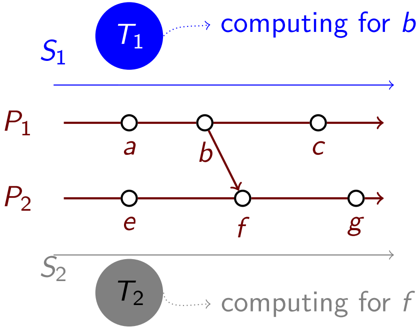
Optimizations - I



Send only if needed - ie. before sending your token to S_k, check if you have token T_k containing the required information.

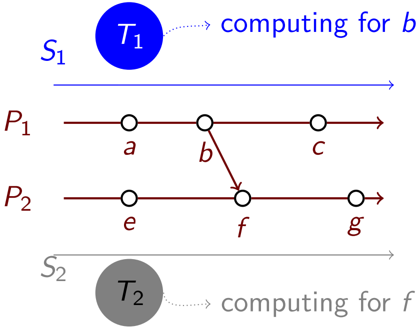
Optimizations - II

Stall computations that would lead to duplicate computations



Optimizations - II

Stall computations that would lead to duplicate computations



Allow only one computation to progress if there is a possibility of duplicates (see paper for details)

Distributed vs Centralized

n: # of processes, $|S|$: # bits required to store state data
$|E|$: # of events in computation $|E_i|$: # of events on process P_i

	Centralized	Distributed
Work/Process	$O(n^2\|E\|)$	$O(n\|E\|)$
Space/Process	$O(\|E\|.\|S\|)$	$O(\|E_i\|.\|S\|)$

$O(n)$ savings in work per process
$O(n)$ savings in storage space per process

For conjunctive predicates:
The optimized version has $O(n)$ savings in message load per process

Questions?

Thanks!

Future Work

- Even with optimizations, there can be degenerate cases with $O(|E|)$ messages on a single process
- Is there a distributed algorithm that guarantees reduced messages (by $O(n)$) per process?
- Total work performed is still $O(n|E|)$
- Is there a distributed algorithm that reduces this bound?