A Distributed Abstraction Algorithm for Online Predicate Detection

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

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

2Department of Computer Science,
University of Texas at Dallas
Outline

1. Introduction
2. Background
3. Abstraction - Computational Slicing
4. Distributed Online Slicing
5. Conclusion
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

$$P_1$$

$$P_2$$

$$P_3$$

Predicate $$(x_1 \times x_2 + x_3 < 5) \land (x_1 \geq 1) \land (x_3 \leq 3): \mathcal{O}(k^3)$$ steps

- $$\mathcal{O}(k^n)$$ complexity for $$n$$ processes, and $$k$$ events per process
- Compute intensive for large computations
Exploiting Predicate Structure Using Abstractions

Predicate \((x_1 \times x_2 + x_3 < 5) \land (x_1 \geq 1) \land (x_3 \leq 3)\)

\begin{itemize}
  \item (a) Original Computation
  \item (b) Slice w.r.t. \( (x_1 \geq 1) \land (x_3 \leq 3) \)
\end{itemize}
Motivation & Problem Definition

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 \(e\).

\[
e \in C \land f \rightarrow e \implies f \in C
\]
**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.

\[
\begin{align*}
\{\} & \quad \checkmark \\
\{a\} & \quad \checkmark \\
[b, e] & \quad \checkmark \\
[a, f] & \quad \times
\end{align*}
\]

Use vector clocks for checking consistency/finding causal dependency.
Lattice of Consistent Cuts

Set of all consistent cuts of a computation \((E, \to)\), 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.

![Diagram of consistent cuts and lattices]

$P_1$ and $P_2$ represent the consistent cuts and lattices for the regular predicate.
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”

\[ P_1 \quad a \quad b \quad c \]
\[ P_2 \quad e \quad f \quad g \]
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.

\[ \{a\}, \{a, e\}, \{c\}, \{b, f\}, \{c, g\}, \{b, e\}, \{b\}, \{a, e\}, \{a\}, \{e\}, \{\}\]
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 $e$. 

$B$: "all channels are empty"
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 $e$. 

$B$: “all channels are empty"
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 $e$. 

---

**Introduction**

**Background**

**Abstraction - Computational Slicing**

**Distributed Online Slicing**

**Conclusion**

---

Himanshu (UT Austin)
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 $e$. 

$B$: "all channels are empty"
Slice for Regular Predicates

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

\[
J_B = \{ J_B(e) \mid e \in E \}
\]
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 \textit{locally} to $S_i$

\begin{figure}
\centering
\begin{tikzpicture}
\node [draw, circle, fill=blue!20] (S1) at (0,0) {$S_1$};
\node [draw, circle, fill=blue!20] (T1) at (1,0) {$T_1$};
\node [draw, circle, fill=blue!20] (P1) at (0,-1) {$P_1$};
\node [draw, circle, fill=blue!20] (S2) at (1,-1) {$S_2$};
\node [draw, circle, fill=blue!20] (T2) at (2,-1) {$T_2$};
\draw [->, thick] (S1) -- (T1);
\draw [->, thick] (P1) -- (0,0);
\draw [->, thick] (S2) -- (T2);
\draw [->, thick] (P2) -- (1,0);
\end{tikzpicture}
\end{figure}
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 $S_i$

$B = \text{“all channels are empty”}$

<table>
<thead>
<tr>
<th></th>
<th>$T_1 @ S_1$</th>
<th>$T_2 @ S_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>$P_1.1$</td>
<td>$P_2.1$</td>
</tr>
<tr>
<td>cut</td>
<td>$[1, 0]$</td>
<td>$[0, 1]$</td>
</tr>
<tr>
<td>dependency</td>
<td>$[1, 0]$</td>
<td>$[0, 1]$</td>
</tr>
<tr>
<td>cut consistent?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>satisfies $B$?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>output cut?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>wait for</td>
<td>$P_1.2$</td>
<td>$P_2.2$</td>
</tr>
</tbody>
</table>

P1
1

P2
1
What happens in non-trivial cases?

\[ B = \text{“all channels are empty”} \]

\[ S_1 \xrightarrow{T_1} P_1 \]
\[ P_1 \xrightarrow{1} \]
\[ P_2 \xrightarrow{1} \]
\[ S_2 \xrightarrow{T_2} \]
What happens in non-trivial cases?

\[ B = \text{“all channels are empty”} \]

Suppose, \( P_1 \) just reported its 2\(^{nd} \) event to \( S_1 \).
What happens in non-trivial cases?

$B = \text{"all channels are empty"}$

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

<table>
<thead>
<tr>
<th></th>
<th>$T_1 @ S_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
<td>$P_{1.2}$</td>
</tr>
<tr>
<td>$cut$</td>
<td>$[2, 0]$</td>
</tr>
<tr>
<td>$dependency$</td>
<td>$[2, 0]$</td>
</tr>
<tr>
<td>$cut$ consistent?</td>
<td>✓</td>
</tr>
<tr>
<td>satisfies $B$?</td>
<td>X</td>
</tr>
<tr>
<td>wait for</td>
<td>$P_{2.1}$</td>
</tr>
</tbody>
</table>

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

\[
\begin{aligned}
S_1 & \\
P_1 & \quad \quad 1 \quad 2 \\
P_2 & \quad \quad 1 \quad 2 \\
S_2 & \quad T_1 \\
\end{aligned}
\]

wait for $P_{2.1}$
**S₂ receives T₁**

Regular predicate structure

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

\[ S₂ \text{ receives } T₁ \]

\[ B \text{ would not be even evaluated on any state unless } S₂ \text{ is told about a message ‘receipt’} \]
Basic Algorithm

\( S_2 \) receives \( T_1 \)

Regular predicate structure

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

\[
\begin{align*}
S_1 & \\
P_1 & (1, 2) \\
P_2 & (1, 2) \\
S_2 & \quad T_1
\end{align*}
\]

\( B \) 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

Himanshu (UT Austin)
$P_{2.2}$ is reported to $S_2$

After $P_{2.2}$ is reported to $S_2$

<table>
<thead>
<tr>
<th></th>
<th>$T_1@S_2$</th>
<th>$T_2@S_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
<td>$P_{1.2}$</td>
<td>$P_{2.2}$</td>
</tr>
<tr>
<td>cut</td>
<td>[2, 2]</td>
<td>[2, 2]</td>
</tr>
<tr>
<td>dependency</td>
<td>[2, 2]</td>
<td>[2, 2]</td>
</tr>
<tr>
<td>cut consistent?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>satisfies $B$?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>output cut?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>wait for</td>
<td>$P_{1.3}$</td>
<td>$P_{2.3}$</td>
</tr>
</tbody>
</table>

$S_2$ sends $T_1$ back to $S_1$
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

![Diagram showing stall computations with nodes and labels: S₁, T₁, S₂, T₂, P₁, P₂, a, b, c, e, f, g. T₁ computes for b and T₂ computes for f.](image)
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,  
$|E|$: # of events in computation  
$|S|$: # bits required to store state data  
$|E_i|$: # of events on process $P_i$

<table>
<thead>
<tr>
<th></th>
<th>Centralized</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work/Process</td>
<td>$O(n^2</td>
<td>E</td>
</tr>
<tr>
<td>Space/Process</td>
<td>$O(</td>
<td>E</td>
</tr>
</tbody>
</table>

$O(n)$ 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?