1 Introduction & Motivation (2 Minutes)

Today I will be talking about **Program Slicing**.

Let's start with a practical scenario. Imagine you have a program that computes the Sum, Product, Minimum, and Maximum of a set of numbers. You run it, and the "Product" output is wrong.

Write on Whiteboard

- `Product = ...  // Bug!`

To fix this, you don't need to look at the code calculating the Sum or the Max. You only care about the lines of code that actually affect the "Product" variable.

Definition: Informally, a **Program Slice** is a subset of a program that affects the values computed at a specific statement. If we execute the slice, it must yield the same result for that specific statement as the original program.

Why is this useful?

1. **Debugging:** It helps developers focus only on relevant code.
2. **Parallelization:** Disjoint slices can run on different cores.
3. **Regression Testing:** If we change one line, we only need to re-test the tests that cover that slice.

---

2 Formal Definition (2 Minutes)

Let's formalize this. To define a slice, we need a **Slicing Criterion**.

Write on Whiteboard

$$C = (n, V)$$

- n: A statement in the program.
- V: A set of variables accessed at n.

For example, if we have a statement on line 6: `output(x)`, our criterion is $(6, \{x\})$.

We determine if a program P' is a valid slice of P by looking at their **Execution Traces**. If P executes statement n, P' must execute n the same number of times, and the values of the variables V must be identical at each execution.

Important Note: It is formally **undecidable** to compute the *minimal* slice of a program (due to Rice's Theorem/Halting problem). Therefore, in practice, we focus on approximate, safe solutions that might include a few extra statements but never miss a necessary one.

---

3 Intraprocedural Slicing: Data Flow Analysis (4 Minutes)

How do we compute this automatically? We start with **Intraprocedural Slicing** (slicing within a single function without calls). We use a fixpoint algorithm based on the Control Flow Graph (CFG).

First, we need to categorize how variables are used in each statement s.

[Action: Write on Whiteboard]

- $VAR(s)$: The set of all variables appearing in s.
- $DEF(s)$: Variables defined (assigned to) in s.
- $REF(s)$: Variables referenced (read) in s.

With these definitions, we can compute two things: the **Relevant Variables** ($\mathcal{R}_C^0$) and the **Relevant Statements** ($\mathcal{S}_C^0$).

First, we calculate $\mathcal{R}_C^0(d)$, which tells us which variables are relevant immediately *before* statement d executes.

[Action: Write the Data Flow Equation on Whiteboard]

$$\mathcal{R}_C^0(d) = \mathcal{R}_C^0(d) \cup \{v | v \in \mathcal{R}_C^0(f) \setminus DEF(d)\} \cup \{v \in REF(d) \text{ and } DEF(d) \cap \mathcal{R}_C^0(f) \neq \emptyset\}$$

This equation works backwards from a successor node f:

1. **Preservation:** If a variable is relevant at f and is *not* redefined in d, it remains relevant (it flows through).
2. **Generation:** If d defines a variable that is relevant at f ($DEF(d) \cap \mathcal{R}_C^0(f) \neq \emptyset$), then the variables used to calculate that definition ($REF(d)$) become relevant.

Once we have the variables, we can determine the actual slice. We call this set $\mathcal{S}_C^0$—the set of **Relevant Statements**.

[Action: Write the Statement Equation on Whiteboard]

$$\mathcal{S}_C^0 = \{d | d \xrightarrow{CFG} f \text{ and } DEF(d) \cap \mathcal{R}_C^0(f) \neq \emptyset\}$$

Simply put, a statement d is included in the slice $\mathcal{S}_C^0$ if it defines a variable that is relevant at its successor f.

Once we establish this data flow, we must account for **Control Flow**. If a statement d is inside a loop or an `if` block, the condition of that block controls whether d executes. If d is in the slice, the block controlling it must also be added to the slice.

4 Program Dependence Graphs (PDG) (4 Minutes)

The data flow approach works, but it's iterative and can be slow. A more structural approach uses the **Program Dependence Graph (PDG)**.

The PDG is a graph where edges represent dependencies, not just control flow order. Slicing in a PDG is simple: It is just **Backwards Reachability**. If you want the slice for node n, you just find all nodes that can reach n in the PDG.

There are two types of edges in a PDG:

1. Flow Dependence

Write on Whiteboard

$$u \to_f v$$

This exists if u defines a variable x, v reads x, and there is a path from u to v where x is not re-defined.

2. Control Dependence

Write on Whiteboard

$$u \to_c v$$

This captures the logic of "Node u decides whether node v executes." Formally, we use the concept of **Postdominance**. A node v *postdominates* u if every path from u to the `END` goes through v.

We say v is control dependent on u ($u \to_c v$) if:

1. v postdominates one branch of u (e.g., the True branch).
2. v does **not** postdominate u itself.

Handling Jumps: Unstructured code (like `break`, `continue`) is tricky. We handle this by modeling jumps as "Pseudo control nodes" with pseudo-edges in the PDG to ensure the slice includes the jump if it affects the execution flow of relevant statements.

---

5 Interprocedural Slicing (3 Minutes)

Finally, real programs have functions. This brings us to **Interprocedural Slicing**. We assume a "Call by Value-Result" model (inputs copied in, outputs copied back).

We extend the PDG to a **System Dependence Graph (SDG)**. For every procedure call, we add:

- **Call nodes** connecting to the procedure entry.
- **Actual-in/out** nodes connecting to **Formal-in/out** nodes.

The Challenge: If we just do standard graph reachability on the SDG, we get imprecise results. *Example:* If `Main` calls `Add`, and `Foo` calls `Add`, a standard graph traversal might enter `Add` from `Main` but exit back to `Foo`. This path is invalid—a function must return to the site that called it.

The Solution (Two-Phase Slicing): We compute **Summary Edges**. A summary edge connects an input parameter directly to an output parameter if there is a path through the function.

The slicing algorithm then runs in two phases:

1. **Phase 1:** Traverse backwards following flow, control, and call edges (but NOT return edges). This finds everything that happens *before* or *inside* the current call stack.
2. **Phase 2:** From the nodes found in Phase 1, traverse backwards following flow, control, and return edges (but NOT call edges). This finds everything in the procedures that called us.

---

6 Conclusion (< 1 Minute)

To summarize:

1. **Slicing** isolates the code relevant to a specific computation ($C = (n, V)$).
2. We can compute it using **Data Flow Equations** (finding fixpoints of $\mathcal{R}_C$ and $\mathcal{S}_C$).
3. **PDGs** allow us to solve the problem using graph reachability via Flow and Control dependencies.
4. **Interprocedural Slicing** requires SDGs and a two-phase algorithm to respect the calling context.

Thank you. I am happy to take any questions.