Program Slicing

Xiangyu Zhang
Outline

- What is slicing.
- Why slicing.
- Static slicing.
- Dynamic slicing.
  - Data dependence detection
  - Control dependence detection
  - Slicing algorithms (forward vs. backward)
  - Chopping
What is a slice?

S: .... = f (v)

Slice of v at S is the set of statements involved in computing v’s value at S.

[Mark Weiser, 1982]

```
Void main () {
  int I=0;
  int sum=0;
  while (I<N) {
    sum=add(sum,I);
    I=add(I,1);
  }
  printf ("sum=\%d\n",sum);
  printf("I=\%d\n",I);
}
```

---

```
int sum=0;
while (I<N) {
  sum=add(sum,I);
  I=add(I,1);
}
printf ("sum=\%d\n",sum);
printf("I=\%d\n",I);
```
Why Slicing?

- **Debugging:** that's why slicing was introduced.
- **Data Flow Testing:** Reduce cost of regression testing after modifications to the program.
- **Code Reuse:** Extracting modules for reuse.
- **Version Integration:** Safely integrate non-interfering extensions of an application.
- **Partial Execution replay:** Replay only part of the execution that is relevant to a failure.
- **Partial roll back:** partially roll back a transaction.
- **Information flow:** prevent confidential information from being sent out to untrusted environment.
- **Others.**
How to Compute Slices?

- **Dependence Graph**
  - Data dep.
  - Control dep.

- X is data dependent on Y if (1) there is a variable v that is defined at Y and used at X and (2) there exists a path of nonzero length from Y to X along which v is not re-defined.
How to Compute Slices? (continued)

**Dependence Graph**
- Data dep.
- Control dep.

**Y is control-dependent on X**
iff X directly determines whether Y executes
- X is not strictly post-dominated by Y
- there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y

```
I=0

I< \ N

T

F

I=I+1

print (sum);

print(I)
```
How to Compute Slices? (continued)

Given a slicing criterion, i.e., the starting point, a slice is computed as the set of reachable nodes in the dependence graph.

Slice(1@7) = \{1, 3, 5, 7\}

Slice(6) = ?
Static Slices are Imprecise

- Don’t have dynamic control flow information

```plaintext
1: if (P)
2:    x=f(...);
3: else
4:    x=g(...);
5:    ...=x;
```

- Use of Pointers – static alias analysis is very imprecise

```plaintext
1: int a,b,c;
2: a=...;
3: b=...;
4: p=&a;
5: ...=p[i];
```

- Use of function pointers
Dynamic Slicing

Korel and Laski, 1988

The set of executed statement instances that did contribute to the value of the criterion.

Dynamic slicing makes use of all information about a particular execution of a program.

Dynamic slices are often computed by constructing a dynamic program dependence graph (DPDG).

- Each node is an executed statement (instruction).
- An edge is present between two nodes if there exists a data/control dependence.
- A dynamic slice criterion is a triple \(<\text{Var}, \text{Execution Point}, \text{Input}>\)
- The set of statements reachable in the DPDG from a criterion constitute the slice.

Dynamic slices are smaller, more precise, more helpful to the user.
An Example

1: \( I = 0 \)

2: \( \text{sum} = 0 \)

3: \( I < N \)

4: \( \text{sum} = \text{sum} + I \)

5: \( I = I + 1 \)

6: \( \text{print (sum);} \)

7: \( \text{print(I)} \)

\( \text{Slice}(I@7) = \{1, 3, 5, 7\} \)

Trace (N=0)
1: \( I = 0 \)
2: \( \text{sum} = 0 \)
3: \( I < N \)
4: \( \text{sum} = \text{sum} + I \)
5: \( I = I + 1 \)
6: \( \text{print (sum)}; \)
7: \( \text{print(I);} \)

\( \text{DSlice}(I@7_1, N=0) = \{1, 7\} \)
Another Example

Slice($I@7$) = \{1, 3, 5, 7\}

Trace (N=1)
1₁: $I=0$
2₁: sum=0
3₁: $I< N$
4₁: sum=sum+1
5₁: $I=I+1$
6₁: print(sum)
7₁: print(I);

DSlice($I@7₁, N=1$) = \{1, 3, 5, 7\}
Effectiveness of Dynamic Slicing

<table>
<thead>
<tr>
<th>Program</th>
<th>Static / Dynamic (25 slices)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AVG</td>
</tr>
<tr>
<td>126.gcc</td>
<td>5448</td>
</tr>
<tr>
<td>099.go</td>
<td>1258</td>
</tr>
<tr>
<td>134.perl</td>
<td>66</td>
</tr>
<tr>
<td>130.li</td>
<td>149</td>
</tr>
<tr>
<td>008.espresso</td>
<td>49</td>
</tr>
</tbody>
</table>

- Sometimes, static and dynamic get the same answers.
- Sometimes, static gets explosions
- On average, static slices can be many times larger
Computing Dynamic Slices

- **Data dependence:** - do we still care about aliasing?
  - No.
  - A backwards linear scan over the trace is able to recover all data dependences.

- **Control dependence**
  - In order to find the predicate instance that a statement execution $X$ is control dependent on, can I simply traverse backwards and find the closest predicate?
    - No.
  - There exist inefficient backwards algorithms

```
Trace:
1_1: if (P)
2_1: ...
3_1: i=...
```
Instrument the program to generate the control flow and memory access trace

```c
Void main ( ) {
    int I=0;
    int sum=0;
    while (I<N) {
        sum=add(sum,I);
        I=add(I,1);
    }
    printf ("sum=%d\n",sum);
    printf("I=%d\n",I);
}
```
**Offline Algorithms – Data Dep**

- Instrument the program to generate the control flow and memory access trace

```c
Void main ( ) {
    int I=0; trace("1 W " + &I);
    int sum=0; trace("2 W " + &sum);
    while (trace("3 R " + &I + &N), I < N) {
        sum = add(sum, I);
        trace("4 R " + &I + &sum + " W " + &sum);
        I = add(I, 1);
    }
    printf("sum=%d\n", sum);
    printf("I=%d\n", I);
}
```

Trace (N=0)
- 1 W &I
- 2 W &sum
- 3 R &I &N
- 4 R &I &sum W &sum
- 5 R &I W &I
- 3 R &I &N
- 7 R &sum
- 8 R &I
Offline Algorithms – Data Dep

Instrument the program to generate the control flow and memory access trace

For a “R, addr”, traverse backward to find the closest “W,addr”, introduce a DD edge, traverse further to find the corresponding writes of the reads on the identified write.

• “8, R &I” -> “5, W &I”-> “5, R &I”-> “1, R&I”
Offline Algorithms – Control Dep

Assume there are no recursive functions and CD(i) is the set of static control dependence of i, traverse backward, find the closest x, s.t. x is in CD(i), introduce a dynamic CD from i to x.

Problematic in the presence of recursion.
Efficiently Computing Dynamic Dependences

- The previous mentioned graph construction algorithm implies offline traversals of long memory reference and control flow traces

Efficient online algorithms
- Online data dependence detection.
- Online control dependence detection.
# Efficient Data Dependence Detection

**Basic idea**

- $i$: $x = \ldots \Rightarrow \text{hashmap}[x] = i$
- $j$: $\ldots = x \ldots \Rightarrow$ dependence detected $j \rightarrow \text{hashmap}[x]$, which is $j \rightarrow i$

<table>
<thead>
<tr>
<th>Trace ($N=1$)</th>
<th>HashMap</th>
<th>Data Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $I=0$</td>
<td>I: $1_1$</td>
<td></td>
</tr>
<tr>
<td>2: sum=0</td>
<td>I: $1_1$</td>
<td>sum: $2_1$</td>
</tr>
<tr>
<td>3: $I&lt;N$</td>
<td>I: $1_1$</td>
<td></td>
</tr>
<tr>
<td>4: sum=sum+I</td>
<td>I: $1_1$</td>
<td>sum: $4_1$</td>
</tr>
<tr>
<td>5: $I=I+1$</td>
<td>I: $5_1$</td>
<td>sum: $4_1$</td>
</tr>
<tr>
<td>3: $I&lt;N$</td>
<td>I:</td>
<td></td>
</tr>
<tr>
<td>6: print(sum)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: print(I);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$3_1 \rightarrow \text{hashmap}[I]=1_1$

$4_1 \rightarrow \text{hashmap}[\text{sum}]=2_1$

$5_1 \rightarrow \text{hashmap}[I]=1_1$

$3_2 \rightarrow \text{hashmap}[I]=5_1$

$6_1 \rightarrow \text{hashmap}[\text{sum}]=4_1$

$7_1 \rightarrow \text{hashmap}[I]=5_1$
Efficient Dynamic Control Dependence (DCD) Detection

**Def**: \( y_j \text{ DCD on } x_i \) iff there exists a path from \( x_i \) to Exit that does not pass \( y_j \) and no such paths for nodes in the executed path from \( x_i \) to \( y_j \).

**Region**: executed statements between a predicate instance and its immediate post-dominator form a region.
Region Examples

1. for(i=0; i<N, i++) {
2.     if(i%2 == 0) {
3.         p = &a[i];
4.     } foo(p);
5. }
6. a = a+1;

A statement instance xᵢ DCD on the predicate instance leading xᵢ ‘s enclosing region.
Regions are either nested or disjoint. Never overlap.
**DCD Properties**

- **Def:** $y_j$ DCD on $x_i$ iff there exists a path from $x_i$ to Exit that does not pass $y_j$ and no such paths for nodes in the executed path from $x_i$ to $y_j$.

- **Region:** executed statements between a predicate instance and its immediate post-dominator form a region.

- **Property One:** A statement instance $x_i$ DCD on the predicate instance leading $x_i$'s enclosing region.
Property One

A statement instance $x_i$ DCD on the predicate instance leading $x_i$'s enclosing region.

Proof: Let the predicate instance be $p_j$ and assume $x_i$ does not DCD $p_j$. Therefore,

either there is not a path from $p_j$ to exit that does not pass $x_i$, which indicates $x_i$ is a post-dominator of $p_j$, contradicting the condition that $x_i$ is in the region delimited by $p_j$ and its immediate post-dominator;

or there is a $y_k$ in between $p_j$ and $x_i$ so that $y_k$ has a path to exit that does not pass $x_i$. Since $p_j$'s immediate post-dominator is also a post dominator of $y_k$, $y_k$ and $p_j$'s post-dominator form a smaller region that include $x_i$, contradicting that $p_j$ leads the enclosing region of $x_i$. 


DCD Properties

**Def:** $y_j \text{ DCD on } x_i$ iff there exists a path from $x_i$ to Exit that does not pass $y_j$ and no such paths for nodes in the executed path from $x_i$ to $y_j$.

**Region:** executed statements between a predicate instance and its immediate post-dominator form a region.

Property Two: regions are disjoint or nested, never overlap.
Property Two

Regions are either nested or disjoint, never overlap.

Proof: Assume there are two regions \((x, y)\) and \((m, n)\) that overlap. Let \(m\) reside in \((x, y)\). Thus, \(y\) resides in \((m, n)\), which implies there is a path from \(m\) to exit without passing \(y\). Let the path be \(P\). Therefore, the path from \(x\) to \(m\) and \(P\) constitute a path from \(x\) to exit without passing \(y\), contradicting the condition that \(y\) is a post-dominator of \(x\).
Efficient DCD Detection

**Observation:** regions have the LIFO characteristic.
- Otherwise, some regions must overlap.

**Implication:** the sequence of nested active regions for the current execution point can be maintained by a stack, called control dependence stack (CDS).
- A region is nested in the region right below it in the stack.
- An entry is pushed onto CDS if a branching point (predicates, switch statements, etc.) executes.
- The current entry is popped if the immediate post-dominator of the branching point executes, denoting the end of the current region.
Algorithm

**Predicate** \( (x_i) \)

\[
\{ \\
\quad \text{CDS.push}(<x_i, \text{IPD}(x) >); \\
\}
\]

**Merge** \( (t_j) \)

\[
\{ \\
\quad \text{while (CDS.top().second==t)} \\
\quad \quad \text{CDS.pop();} \\
\}
\]

**GetCurrentCD** \( () \)

\[
\{ \\
\quad \text{return CDS.top().first;} \\
\}
\]
An Example

1. if (p1 || p2) {
2.     s1;
3.     s2;
4. }
5. if (p3) {
6.     while (p4) {
7.         s3;
8.     }
9. } else {
10.    if (p5) {
11.        return;
12.    }
13. }
14. s4;

control flow edge

Branching ()

Merging ()
Interprocedural Control Dependence

<table>
<thead>
<tr>
<th>code</th>
<th>trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(n) {</td>
<td></td>
</tr>
<tr>
<td>1: if (n&gt;0)</td>
<td></td>
</tr>
<tr>
<td>2: f(n-1);</td>
<td></td>
</tr>
<tr>
<td>3: s1;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>4: s2;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

- Annotate CDS entries with calling context.

| 1            | cc1  | 4 |
| 1            | cc2  | 4 |
| 1            | cc3  | 4 |
Wrap Up

- We have introduced the concept of slicing and dynamic slicing
- Offline dynamic slicing algorithms based on backwards traversal over traces is not efficient
- Online algorithms that detect data and control dependences are discussed.
Forward Dynamic Slice Computation

- The approaches we have discussed so far are backwards.
  - Dependence graphs are traversed backwards from a slicing criterion.
  - The space complexity is $O$ (execution length).
- Forward computation
  - A slice is represented as a set of statements that are involved in computing the value of the slicing criterion.
  - A slice is always maintained for a variable.
The Algorithm

An assignment statement execution is formulated as

\[ s_i: \ x = p_j ? \text{op}(\text{src1}, \text{src2}, \ldots); \]
- That is to say, the statement execution instance \( s_i \) is control dependent on \( p_j \) and operates on variables of src1, src2, etc.

Upon the execution of \( s_i \), the slice of \( x \) is updated to

\[ \text{Slice}(x) = \{s\} \cup \text{Slice(src1)} \cup \text{Slice(src2)} \cup \ldots \cup \text{Slice}(p_j) \]
- The slice of variable \( x \) is the union of the current statement, the slices of all variables that are used and the slice of the predicate instance that \( s_i \) is control dependent on. Because they are all contributing to the value of \( x \).
- Such slices are equivalent to slices computed by backwards algorithms.
  - Proof is omitted.
- Slices are stored in a hashmap with variables being the keys.
- The computation of \( \text{Slice}(p_j) \) is in the next slide. Note that \( p_j \) is not a variable.
A predicate is formulated as
\[ s_i: p_j? \text{ op (src1, src2, \ldots)} \]
That is to say, the predicate itself is control dependent on another predicate instance \( p_j \) and the branch outcome is computed from variables of src1, src2, etc.

Upon the execution of \( s_i \)
- A triple is pushed to CDS with the format of
  - \( <s_i, \text{IPD}(s), s \text{ U Slice (src1) U Slice (src2) U \ldots U Slice}(p_j)> \)
- The entry is popped at its immediate post dominator

\( \text{Slice}(p_j) \) can be retrieved from the top element of CDS.
Example

Statements Executed

<table>
<thead>
<tr>
<th>Statement</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: a=1</td>
<td></td>
</tr>
<tr>
<td>2: b=2</td>
<td></td>
</tr>
<tr>
<td>3: c=a+b</td>
<td></td>
</tr>
<tr>
<td>4: if a&lt;b then</td>
<td></td>
</tr>
<tr>
<td>5: d=b*c</td>
<td></td>
</tr>
</tbody>
</table>

Dynamic Slices

- Slice(a) = {1}
- Slice(b) = {2}
- Slice(c) = {1,2,3}
- push(<4, 6, {1,2,4}>)
- Slice(d) = {1,2,3,4,5}
Properties

- The slices are equivalent to those computed by backwards algorithms
  - The proof is omitted.
- The space complexity is bounded
  - $O\left(\left(\# \text{ of variables} + \text{MAX}_CDS\_DEPTH\right) \times \# \text{ of statements}\right)$
- Efficiency relies on the hash map implementation and set operations.
  - A cost-effective implementation will be discussed later in cs510.
Extending Slicing

Essentially, slicing is an orthogonal approach to isolate part of a program (execution) giving certain criterion.

Mutations of slicing
- Event slicing – intrusion detection, execution fast forwarding, understanding network protocol, malware replayer.
- Forward slicing.
- Chopping.
- Probabilistic slicing.