Thin Slicing

Authored by **Manu Sridharan, Stephen J. Fink, Rastislav Bodík**

University of California, Berkeley & IBM T.J. Watson Research Center

PLDI 2007

Presented by **Jing Pu**

Outline

- Motivation
- Approach
- Definitions & Key Techniques
- Evaluation
- Conclusion

Motivation

- Debugging and Program Understanding tasks
  - Finding buggy statement, diagnose bug (find most relevant statements to the bug)
  - Understanding relevant statements, e.g., aliasing, important conditionals

- Drawbacks of traditional static slicing techniques
  - Overly Broad relevance definition
  - Slice Pollution

Approach

- Thin Slicing
  - Redefine relevance – intuitive semantic definition
  - Hierarchical expansion – providing additional information

- Terminology
  - Seed statement
  - Producer statement
  - Explainer statement
  - Dependences

Definitions – producer statements

- Seed statement

A statement or value of interest, e.g., the position in a program where an error occurs

- Producer statement
  - Statement **s1** is a producer for statement **s2** if **s1** is part of a chain of assignments that computes and copies a value to **s2**
  - **S2 –** seed / other producer
  - Direct uses of memory locations (variables & object fields)

- Explainer statement

None-producer statements:

  - Heap-based value flow
  - Control flow

Definitions – Dependences

- Thin slicing

Producer flow dependences

Ignore:

- Base pointer flow dependences
- Control dependences

- Most Tasks

Very few explainers are needed

- Answer more questions, need more explanation

Expand thin slices, need ignored dependences

Examples

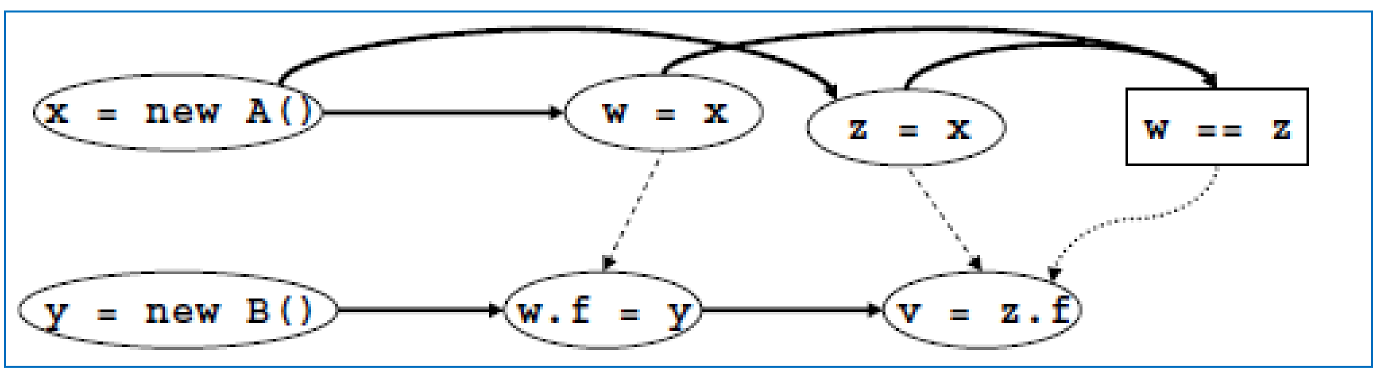
- Example 1: A dependence graph for a program

```
x = new A();
z = x;
y = new B();
w = x;
w.f = y;
if (w == z) {
  v = z.f;   // the seed
}
```

Heap-based value flow

Control flow

Direct uses of memory locations in the thin slicing for the seed



Examples (cont'd)

- Example 2:

thin slices expansion

Diagnose bug:

Which statements cause the 'this' pointers of close() and isOpen() to be aliased

```
class File {
  boolean open;
  File() { ...; this.open = true; }
  isOpen() { return this.open; }
  close() { ...; this.open = false;
  ...
}
readFromFile(File f) {
  boolean open = f.isOpen();
  if (!open)
    throw new ClosedException();
  } ...
}
main() {
  File f = new File();
  Vector files = new Vector();
  files.add(f);
  ...;
  File g = (File)files.get(i);
  g.close();
  ...;
  File h = (File)files.get(i);
  readFromFile(h);
}
```

Bug (→ line `throw new ClosedException();`)

Error (← `g.close();`)

Exception (← `readFromFile(h);`)

Computing thin slices

- Step 1: Do precise pointer analysis:
  - Compute call graph
  - compute may-mod and may-use sets for each method
- Step 2: Build a CFG for each method in the program
- Step 3: Compute each CFG's control and data dependences
- Step 4: Build PDG for each CFG
- **Step 5**: Connect the PDGs to form the SDG:
  - Context-Insensitive/Context-sensitive : Add direct edges for heap access statements

Reference: "Slicing Java Programs that Throw and Catch Exceptions"

Computing thin slices – Modification

- Data dependences for heap access statements

For a statement x.f := e, we add an edge to each statement with an expression w.f on its right-hand side, such that the precomputed points-to analysis indicates x may-alias w.

- Context-Insensitive Thin Slicing
  - Add direct edges across procedure
  - Do not use heap parameters
- Context-sensitive Thin Slicing
  - Add direct edges in the same procedure
  - Use extra parameters and return values to model heap access

Evaluation

- Experimental Setup
  - Implemented thin slicing (context-insensitive/ context-sensitive)& traditional slicing (context-insensitive / context-sensitive) using WALA
  - Used SUN JDK 1.4.2_09 standard library code
- Machine: A Lenovo ThinkPad t60p with dual 2.2GHz Intel T2600 processors and 2GB RAM
- Analyzer: ran on the Sun JDK 1.5_07 using at most 1GB of heap space.
- Benchmarks: SIR, SPECjvm98

Evaluation (cont'd)

Program	Methods	Bytecode Size (KB)	Call Graph Nodes	SDG Statements
Software-Artifact Infrastructure Repository				
nanoxml	541	35	817	22205
jtopas	337	24	397	23766
ant	11147	632	20164	584155
xmlsec	11192	678	17075	525886
SPECjvm98				
mtrt	470	32	514	19699
jess	1061	67	1466	46037
javac	1610	118	2127	71041
jack	592	55	1088	38114

Benchmark characteristics: derived from methods discovered during on-the-fly call graph construction, including Java library methods

Evaluation (cont'd)

- Key points
  - uses injected bugs from the SIR suite
  - Seed: as the point of failure
  - Desired statements: the cause of the bug
  - Control dependence: manually pre-determined
- Scalability
  - context-insensitive (thin/traditional) :good
  - context-sensitive traditional for small codes: good
  - context-sensitive traditional for large codes: bad
- Precision
  - Traditional: Measuring Slice Size
  - New Approach: Use a breadth-first traversal strategy to simulate the statements inspecting process done by the user, **terminated when discover all required statements for original problem**
  - context-sensitive thin: not practical

Experiment 1 – Locating bugs

- Target
  - Check if thin slices include the buggy statement
  - Compare inspected slice size (thin / traditional)

Bug	# Thin	# Trad.	Ratio	# Control	# ThinNoObjSens	# TradNoObjSens
nanoxml-1	12	32	2.67	0	12	32
nanoxml-2	25	113	4.52	0	431	1675
nanoxml-3	29	123	4.24	0	472	1883
nanoxml-4	12	33	2.75	1	17	44
nanoxml-5	35	156	4.46	1	159	45
nanoxml-6	12	52	4.33	0	35	90
jtopas-1	1	1	1	0	1	1
jtopas-2	2	2	1	1	2	2
ant-1	2	2	1	1	2	2
ant-2	4	5	1.25	0	4	5
ant-3	34	55	1.62	15	251	501
ant-4	3	3	1	2	3	3
xml-security-1	2	2	1	1	2	2

aliasing

control dependence

Experiment 2 – Program understanding

- What is a tough cast ?
- Thin Slicing – Understanding the safety of tough cast

```
class Node {
  final int op;
  static int ADD_NODE_OP = 1;
  Node(int op) { this.op = op; }
}
class AddNode extends Node {
  AddNode(...) {
    super(ADD_NODE_OP); ...
  }
}
void simplify(Node n) {
  int op = n.op;
  switch (op) {
  case ADD_NODE_OP:
    AddNode add = (AddNode) n;
    ...
  }
}
```

→ **Tough Cast**

Experiment 2 (cont'd)

- Investigate 10 random tough casts for each SPEC benchmark
- Compared BFS traversal sizes to manually identified required statements size

Cast	# Thin	# Trad.	Ratio	# Control	# ThinNoObjSens	# TradNoObjSens
mtrt-1	22	51	2.32	0	22	51
mtrt-2	23	52	2.26	0	23	52
jess-1	6	7	1.17	2	6	7
jess-2	13	39	3	0	25	93
jess-3	6	6	1	2	6	6
jess-4	6	7	1.17	2	6	7
jess-5	6	7	1.17	2	6	7
jess-6	6	6	1	2	6	6
javac-1	57	910	16	1	57	910
javac-2	43	853	19.8	1	43	853
javac-3	65	2224	34.2	1	65	2267
javac-4	45	855	19	1	45	855
jack-1	18	79	4.39	0	303	758
jack-2	57	151	2.65	0	339	647
jack-3	18	69	3.83	0	304	603
jack-4	18	79	4.39	0	304	759
jack-5	57	151	2.65	0	339	647
jack-6	35	132	3.77	0	338	802
jack-7	35	132	3.77	0	338	802
jack-8	35	132	3.77	0	338	802
jack-9	30	79	2.63	0	304	759
jack-10	57	151	2.65	0	339	647

Threats to validity

- Uses injected bugs from the SIR suite
- Use BFS traversal to simulate user process
- Use of whole-program pointer analysis and call graph construction for the thin slicer may not scale to larger benchmarks

Conclusions

- Thin slices lead the user to desired statements
- Thin slices focus better on desired statements than traditional slices
- A precise pointer analysis is key to effective thin slicing
- Thin slices can be computed efficiently:

  context-insensitive thin slicing algorithm scaled well to large programs

Questions ?

Thanks!