More Pointer Analysis

Last time

- Flow-Insensitive Pointer Analysis
  - Inclusion-based analysis (Andersen)

Today

- Class projects
- Context-Sensitive analysis

Cloning-Based Context-Sensitive Pointer Alias Analysis using BDDs

John Whaley
Monica Lam
Stanford University

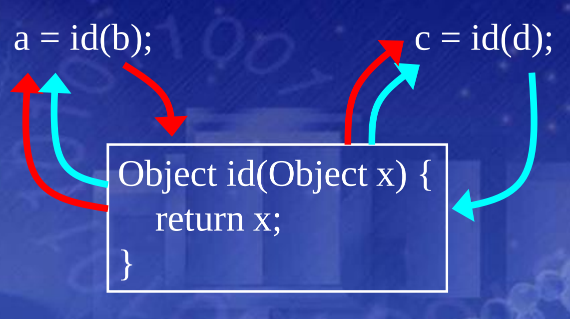
June 10, 2004

Unification vs. Inclusion

- Earlier scalable pointer analysis was context-insensitive unification-based [Steensgaard '96]
  - Pointers are either unaliased or point to the same set of objects.
  - Near-linear, but VERY imprecise.
- Inclusion-based pointer analysis
  - Can point to overlapping sets of objects.
  - Closure calculation is $O(n^3)$
  - Various optimizations [Fahndrich,Su,Heintze,…]
  - BDD formulation, simple, scalable [Berndl,Zhu]

Context Sensitivity

- Context sensitivity is important for precision.
  - Unrealizable paths.



Context Sensitivity

- Context sensitivity is important for precision.
  - Unrealizable paths.

a = id(b);

c = id(d);

```
Object id(Object x) {
    return x;
}
```

June 10, 2004

Cloning-Based Context-Sensitive
Pointer Alias Analysis using BDDs

4

Context Sensitivity

- Context sensitivity is important for precision.
  - Unrealizable paths.
  - Conceptually give each caller its own copy.

a = id(b);

c = id(d);

```
Object id(Object x) {
    return x;
}
```

```
Object id(Object x) {
    return x;
}
```

June 10, 2004

Cloning-Based Context-Sensitive
Pointer Alias Analysis using BDDs

5

Summary-Based Analysis

- Popular method for context sensitivity.
- Two phases:
  - Bottom-up: Summarize effects of methods.
  - Top-down: Propagate information down.
- Problems:
  - Difficult to summarize pointer analysis.
  - Summary-based analysis using BDD: not shown to scale [Zhu'02]
  - Queries (e.g. which context points to x) require expanding an exponential number of contexts.

Cloning-Based Analysis

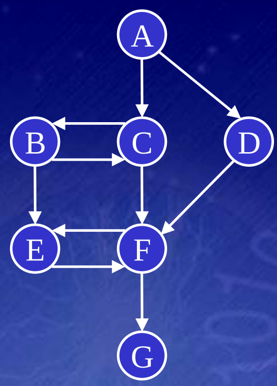
- Simple brute force technique.
  - Clone every path through the call graph.
  - Run context-insensitive algorithm on expanded call graph.
- The catch: exponential blowup

Cloning is exponential!

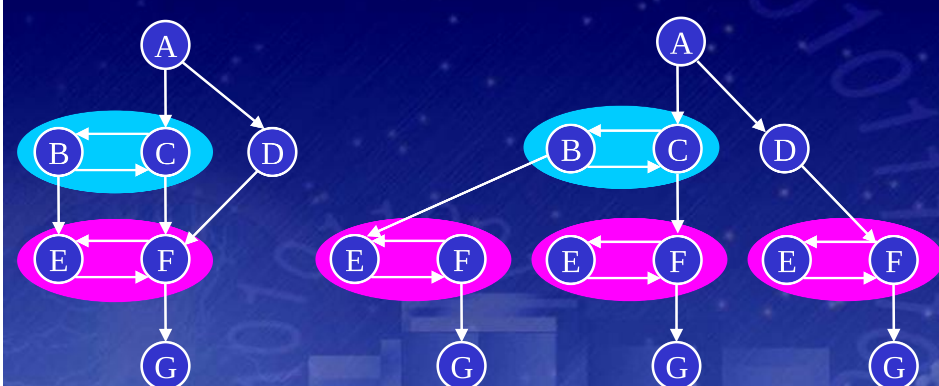
Recursion

- Actually, cloning is unbounded in the presence of recursive cycles.
- Technique: We treat all methods within a strongly-connected component as a single node.

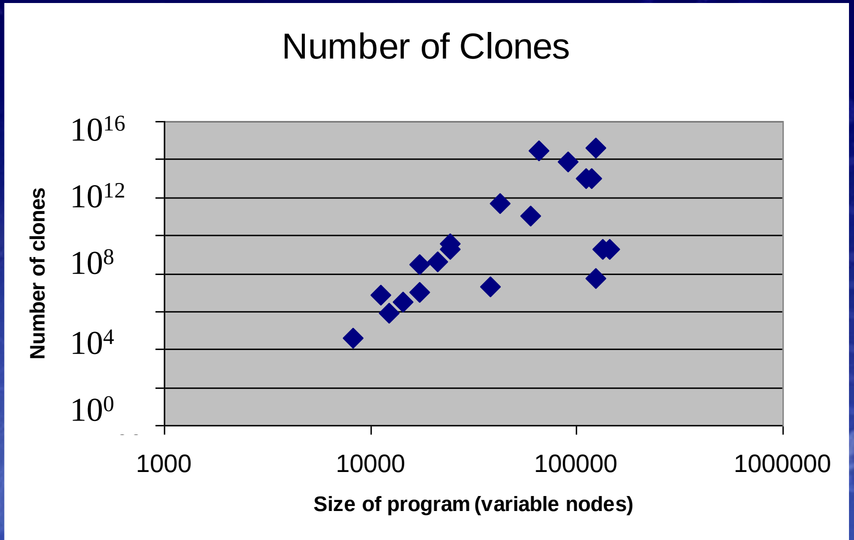
Recursion



Recursion



Top 20 Sourceforge Java Apps



Cloning is infeasible (?)

- Typical large program has ~10^{14} paths
  - If you need 1 byte to represent a clone:
    - Would require 256 terabytes of storage
      - Registered ECC 1GB DIMMs: $98.6 million
        - » Power: 96.4 kilowatts = Power for 128 homes
      - 300 GB hard disks: 939 x $250 = $234,750
        - » Time to read (sequential): 70.8 days
- Seems unreasonable!

BDD comes to the rescue

- There are many similarities across contexts.
  - Many copies of nearly-identical results.
- BDDs can represent large sets of redundant data efficiently.
  - Need a BDD encoding that exploits the similarities.

Contribution (1)

- Can represent context-sensitive call graph efficiently with BDDs and a clever context numbering scheme
  - Inclusion-based pointer analysis
    - 10^{14} contexts, 19 minutes
  - Generates all answers

Contribution (2)

BDD hacking is complicated →

bddbddb
(BDD-based deductive database)

- Pointer algorithm in 6 lines of Datalog
- Automatic translate into efficient BDD implementation
- 10x performance over hand-tuned solver (2164 lines of Java)

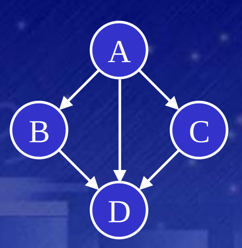
Contribution (3)

- bddbddb: General Datalog solver
  - Supports simple declarative queries
  - Easy use of context-sensitive pointer results
- Simple context-sensitive analyses:
  - Escape analysis
  - Type refinement
  - Side effect analysis
  - Many more presented in the paper

Context-sensitive call graphs in BDD

Call graph relation

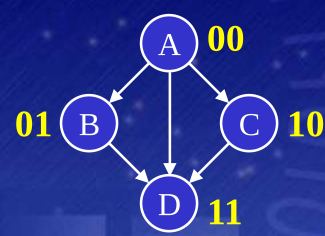
- Call graph expressed as a relation.
  - Five edges:
    - Calls(A,B)
    - Calls(A,C)
    - Calls(A,D)
    - Calls(B,D)
    - Calls(C,D)



Call graph relation

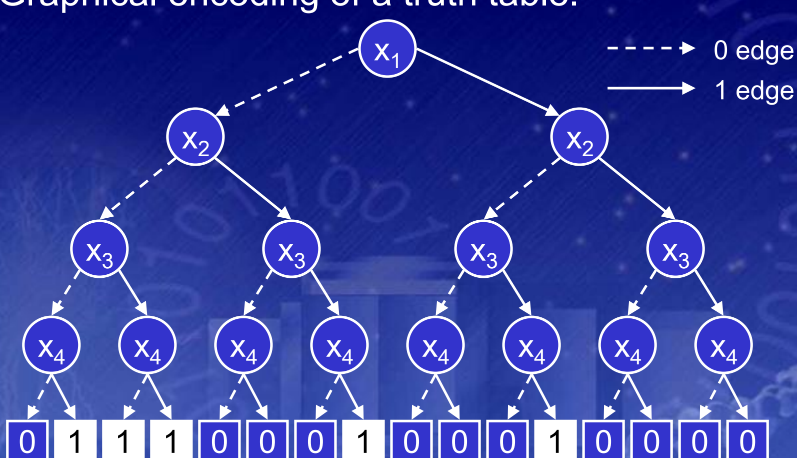
x_1	x_2	x_3	x_4	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

- Relation expressed as a binary function.
  - A=00, B=01, C=10, D=11



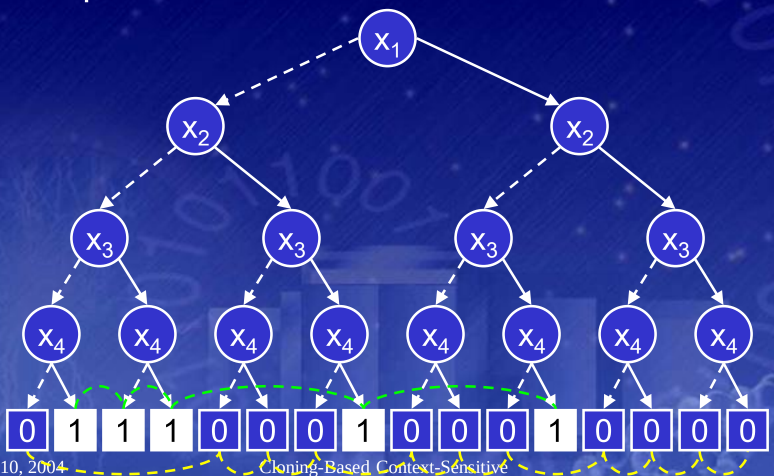
Binary Decision Diagrams

- Graphical encoding of a truth table.



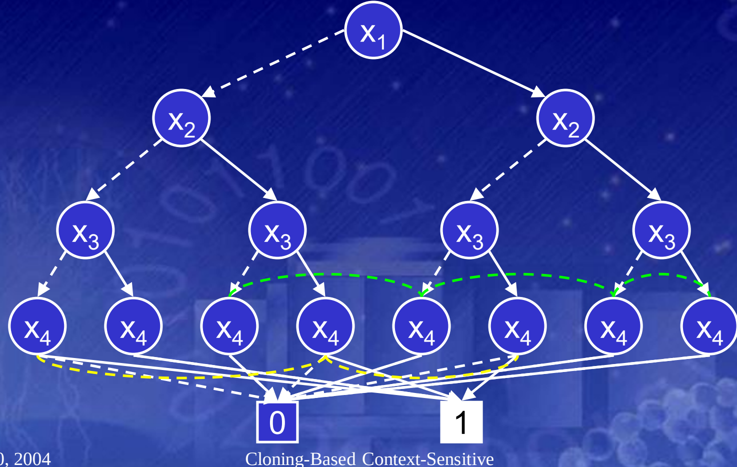
Binary Decision Diagrams

- Collapse redundant nodes.



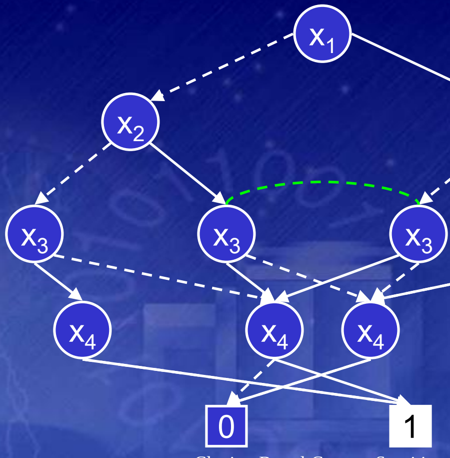
Binary Decision Diagrams

- Collapse redundant nodes.



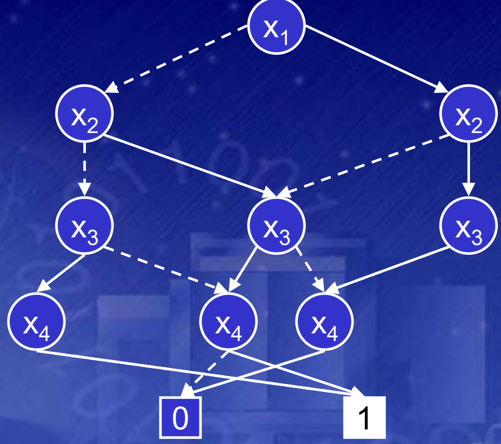
Binary Decision Diagrams

- Collapse redundant nodes.



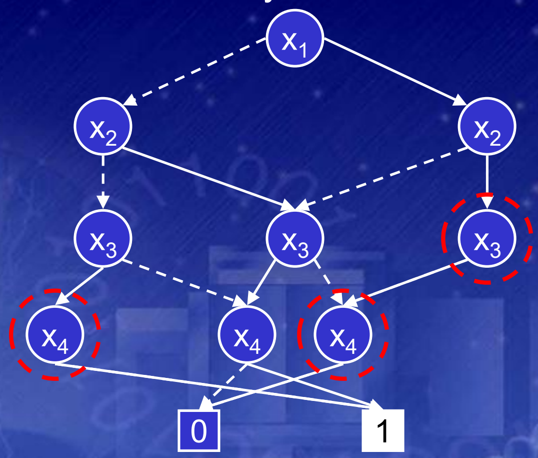
Binary Decision Diagrams

- Collapse redundant nodes.



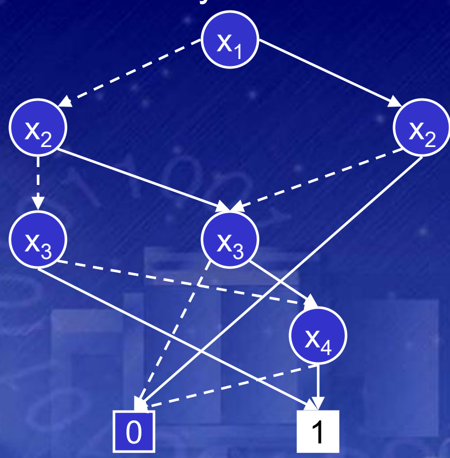
Binary Decision Diagrams

- Eliminate unnecessary nodes.



Binary Decision Diagrams

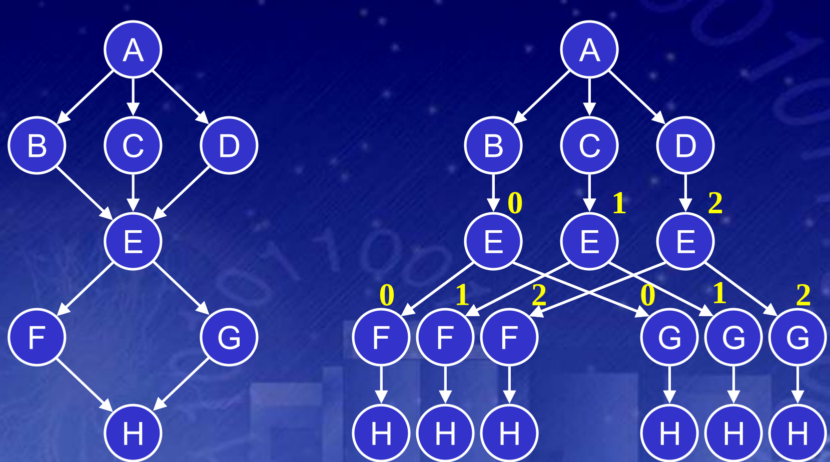
- Eliminate unnecessary nodes.



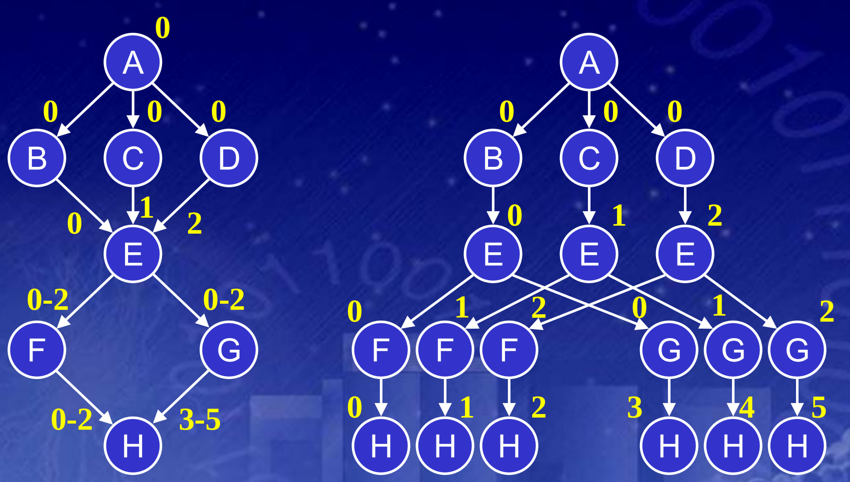
Binary Decision Diagrams

- Size is correlated to amount of redundancy, NOT size of relation.
  - As the set gets larger, the number of don't-care bits *increases*, leading to fewer necessary nodes.

Expanded Call Graph



Numbering Clones

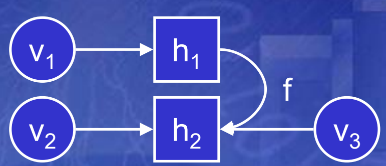


Pointer Analysis

Pointer Analysis Example

h_1: v_1 = new Object();
h_2: v_2 = new Object();
$v_1.f = v_2$;
$v_3 = v_1.f$;

<u>Input Relations</u>
vPointsTo(v_1,h_1)
vPointsTo(v_2,h_2)
Store(v_1,f,v_2)
Load(v_1,f,v_3)



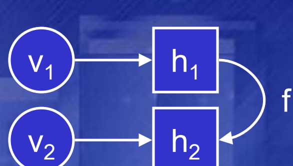
<u>Output Relations</u>
hPointsTo(h_1,f,h_2)
vPointsTo(v_3,h_2)

Inference Rule in Datalog

Stores:

hPointsTo(h_1, f, h_2) :- Store(v_1, f, v_2),
vPointsTo(v_1, h_1),
vPointsTo(v_2, h_2).

$v_1.f = v_2$;



Context-sensitive pointer analysis

- Compute call graph with context-insensitive pointer analysis.
  - Datalog rules for:
    - assignments, loads, stores
    - discover call targets, bind parameters
    - type filtering
  - Apply rules until fix-point reached.
- Compute expanded call graph relation.
- Apply context-insensitive algorithm to expanded call graph.

bddbddb:
BDD-Based Deductive DataBase

Datalog

- Declarative logic programming language designed for databases
  - Horn clauses
  - Operates on relations
- Datalog is expressive
  - Relational algebra:
    - Explicitly specify relational join, project, rename.
  - Relational calculus:
    - Specify relations between variables; operations are implicit.
  - Datalog:
    - Allows recursively-defined relations.

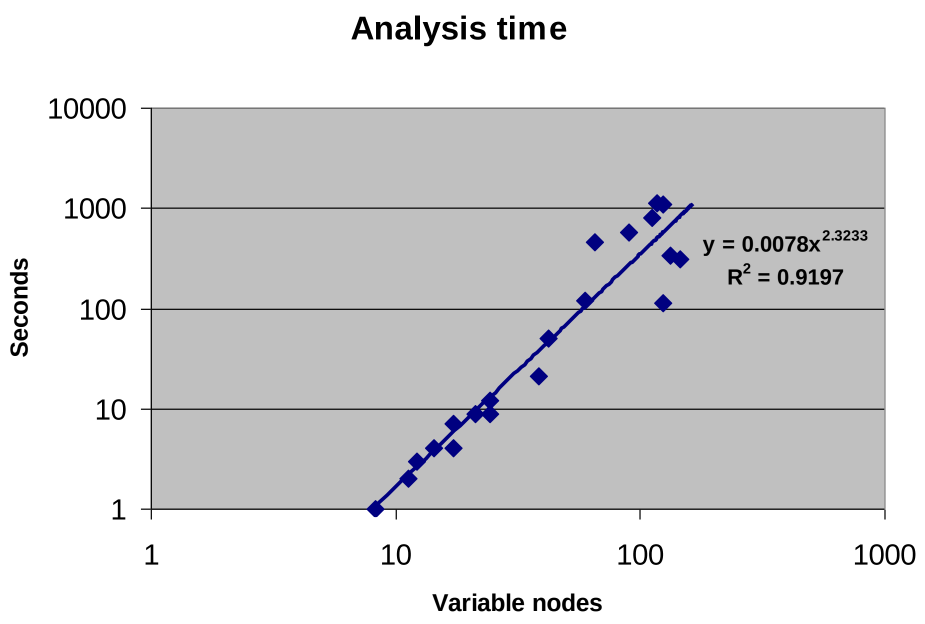
Datalog → BDD

- Join, project, rename are directly mapped to built-in BDD operations
- Automatically optimizes:
  - Rule application order
  - Incrementalization
  - Variable ordering
  - BDD parameter tuning
  - Many more…

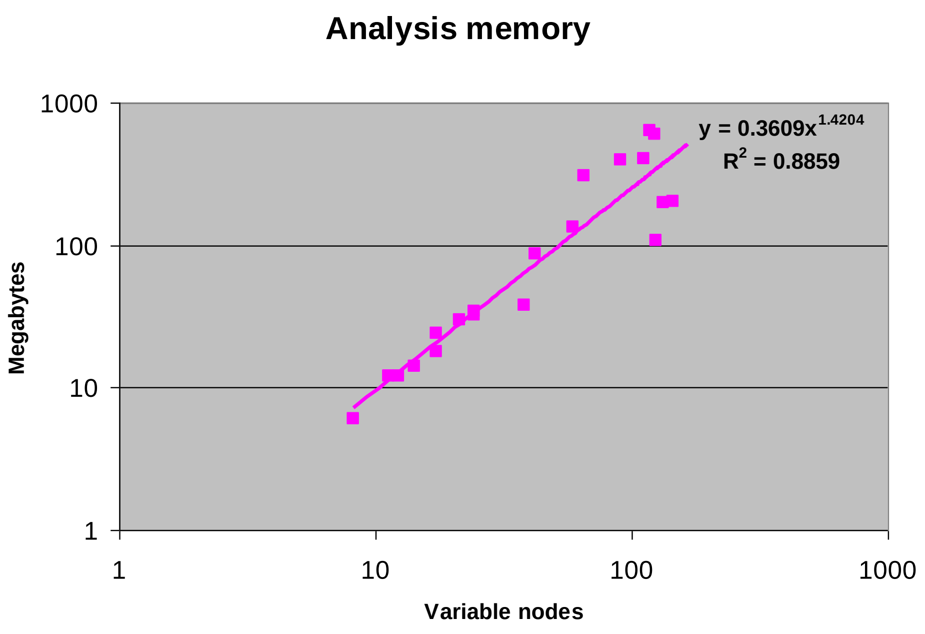
Experimental Results

Experimental Results

- Top 20 Java projects on SourceForge
  – Real programs with 100K+ users each
- Using automatic bddbddb solver
  – Each analysis only a few lines of code
  – Easy to try new algorithms, new queries
- Test system:
  – Pentium 4 2.2GHz, 1GB RAM
  – RedHat Fedora Core 1, JDK 1.4.2_04, javabdd library, Joeq compiler



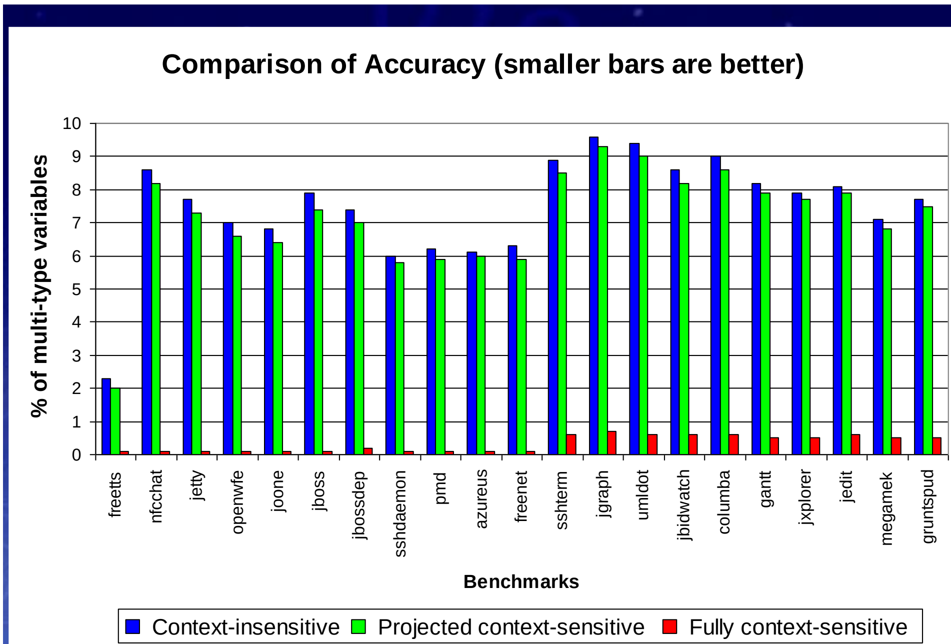
Analysis time
Seconds
10000
1000
100
10
1
1
10
100
1000
Variable nodes
y = 0.0078x^2.3233
R^2 = 0.9197



Analysis memory
Megabytes
1000
100
10
1
1
10
100
1000
Variable nodes
y = 0.3609x^1.4204
R^2 = 0.8859

Multi-type variables

- A variable is multi-type if it can point to objects of different types.
  - Measure of analysis precision
  - One line in Datalog
- Two ways of handling context sensitivity:
  - Projected: Merge all contexts together
  - Full: Keep each context separate



Related Work

- Context-insensitive pointer analysis
  - Steensgaard: Unification-based (POPL'96)
  - Andersen: Inclusion-based ('94)
    - Optimizations: too many to list
    - Berndl: formulate in BDD (PLDI'03)
  - Das: one-level-flow (PLDI'00)
    - Hybrid unification/inclusion

Related Work

- Scalable context-sensitive pointer analysis
  - Fähndrich etal, instantiation constraints (PLDI'00)
    - CFL-reachability
    - Unification-based: Imprecise.
    - Handles recursion well.
    - Computes on-demand.
  - GOLF: Das etal. (SAS'01)
    - One level of context sensitivity.
  - Foster, Fahndrich, Aiken (SAS'00)
    - Throws away information.
  - Wilson & Lam: PTF (PLDI'95)
    - Doesn't really scale (especially complexity)

Related Work

- Whaley & Rinard: Escape analysis (OOPSLA'99)
  - Compositional summaries: only weak updates.
  - Achieves scalability by collapsing escaped nodes.
- Emami & Hendren: Invocation graphs (PLDI'94)
  - Only shown to scale to 8K lines.
- Zhu & Calman: (PLDI'04)
  - To be presented next in this session.
- More complete coverage in the paper.

Conclusion

- The first scalable context-sensitive inclusion-based pointer analysis.
  - Achieves context sensitivity by cloning.
- bddbddb: Datalog → efficient BDD
- Easy to query results, develop new analyses.
- Very efficient!
  - <19 minutes, <600mb on largest benchmark.
- Complete system is publically available at: http://suif.stanford.edu/bddbddb

Epilogue

Impact

- Best Paper Award, PLDI 2004
- High-level specification is successful
  - Datalog now used as specification language for Java pointer analyses (Doop)

Reflection

Scalability

- Whaley and Lam's algorithm scales to 700K LOC
- Shows the benefits of abstraction
  - Represent the call graph as a binary function
  - Represent the binary function as a BDD

Is this a solved problem?

- LOC measured in bytecodes not source lines
- Only top-level variables are context-sensitive
- This strategy works well for Java but not C
  - For C, this analysis only scales to 30K LOC