Demand-Driven Points-To Analysis For Java

Manu Sridharan, Ras Bodik
UC Berkeley

Lexin Shan
Microsoft

Denis Gopan
UW Madison

OOPSLA 2005

Who needs better pointer analysis?

IDEs:

- for refactoring, program understanding
- but program edits invalidate analysis
- current analyses too slow to re-analyze
- incremental analyses hard to engineer

JIT compilers:

- for virtual call resolution, register allocation
- but current analyses too slow for runtime
- plus, class loading invalidates, needs re-analysis

What we provide

Analysis so <u>fast</u> that you can

- run it in the JIT compiler
  - 16x speedup compared to Andersen's analysis
- rerun it after the code changes
  - 2ms per query about a pointer variable

Analysis with <u>low memory overhead</u>

- < 50 KB, eases engineering effort

Analysis with <u>tunable precision</u>

- adjustable to different time constraints

Contributions

1) Demand analysis with early termination:

- return conservative result after a time out

Problem we had to solve:

- how to approximate to make early termination rare?

2) Refining the approximation

Problems we had to solve:

- Mechanism: how to refine?
- Policy: where to focus the refinement budget?

Outline

- Points-to analysis background
- Our approach
  - Demand analysis
  - Early termination
- Our algorithms
  - CFL-reachability formulation
  - Approximation
  - Refinement (undoing the approximation)
- Experiments

Points-To Analysis

- Compute objects each variable can point to
  - For each var x, points-to set pt(x)
- Andersen's Analysis: our reference point
  - Want similar precision for our analysis
  - One abstract location for each allocation site

    x = new Foo() yields pt(x) = { $o1_{Foo}$ }
  - Context- and flow-insensitive
- Current implementations not suitable for us
  - Too costly for JIT, IDE
    - 30 s / 30 MB (Berndl et. al. PLDI03) on `jedit`
  - Code changes require re-analysis

Demand-Driven Analysis

Protocol:

- Client asks a <u>query</u>: what's the points-to set of variable x?
- Analysis computes <u>only</u> the points-to set of x

Works well when typically <u>few queries</u>:

- JIT compiler: variables in hot code
- IDE: variables in code being edited by developer

Visits <u>theoretically minimal</u> set of statements

Problem:

- worst-case time same as exhaustive
- Happens in practice for standard Andersen's

Lesson: Need to <u>approximate</u> for scalability

- Ideally, maintain nearly all precision

Approx: Early Termination

Terminate queries when budget exhausted

Return a sound result to client

- early result: pt(x) = { all abstract locs }
- complete result: pt(x) = { $o1_{Foo}$, $o2_{Bar}$ }

No precision loss if complete result does not satisfy client

Hypothesis: long-running) unsatisfying

- Suggested previously (Heintze / Tardieu PLDI01)
- For standard Andersen's, large precision loss

<u>Challenge</u>: how to approximate further?

Key Ideas

Formulate analysis in CFL-reachability

- Natural for demand-driven analysis
- Andersen's for Java is balanced parens

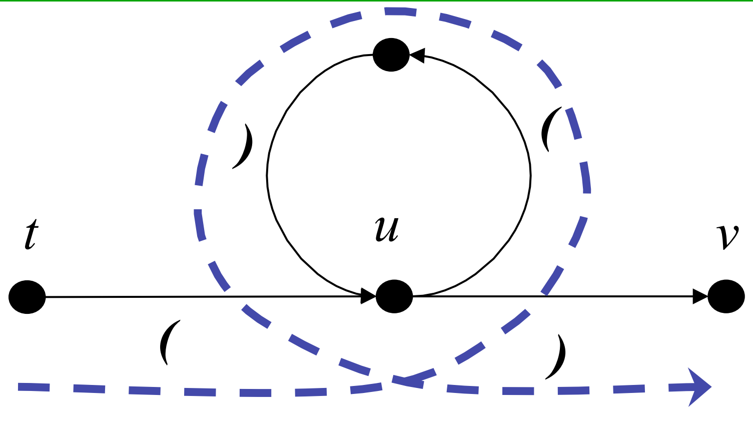
Approximate through regularization

- Solvable by linear DFS algorithm

Iterative refinement to de-approximate

- Simple recursive queries
- Client-driven

CFL-Reachability



S ! SS | (S) | ε

(( ))

Points-to analysis graph:

- Nodes represent variables / locs
- Edges represent statements

Points-to analysis paths:

- o 2 pt(x) , ***flowsTo***-path from o to x
- pt(x) Å pt(y) ≠ ; , ***alias***-path from x to y

Andersen's Analysis in CFL-Reachability

x = new Obj(); // o_1

z = new Obj(); // o_2

w = x;

y = x;

y.f = z;

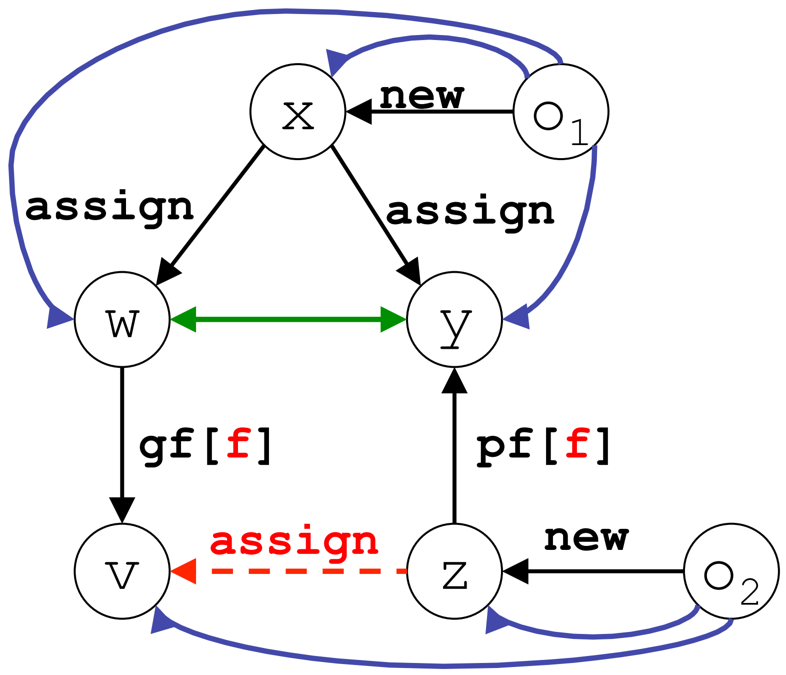
v = w.f;

Edge types

statement

flowsTo

alias



flowsTo ! **new** (pf[f] alias gf[f] | assign)*

balanced parens

Field-sensitive formulation: standard for Java

See paper for alias grammar

Approx: Regularization

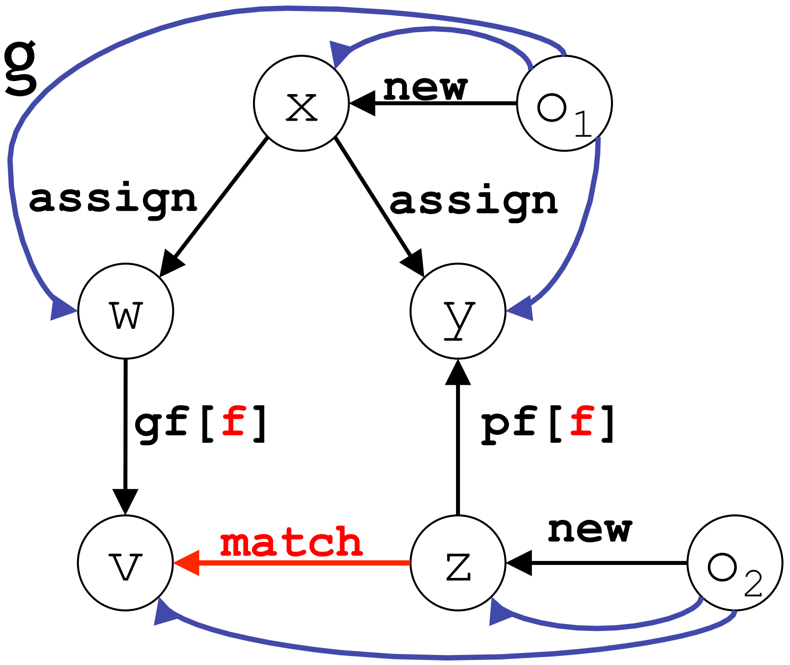
Add match edges for matching field read/write pairs

- From source of putfield
- To sink of getfield

Regular grammar

- Yields DFS algorithm

Field-based precision



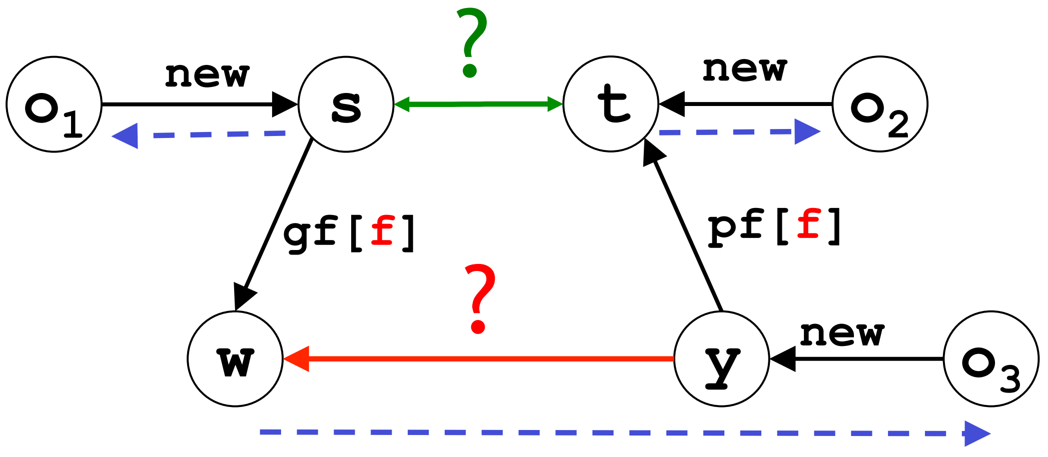
```
flowsToReg ! new (match | assign)*
! new (pf[f] alias gf[f] | assign)* assign)*
pf[f] alias gf[f] ) match
o flowsTo x ) o flowsToReg x
```

RegularPT

```
marked, worklist: Set of Node
procedure query(source: Node)
  add source to marked and worklist
  while (worklist is non-empty) do
    remove w from worklist
    foreach NEW edge o -> w do
      add o to points-to set of source
    end
    foreach ASSIGN and MATCH edge y -> w do
      if y unmarked, add y to marked and worklist
    end
  end
end
```

- No caching, so very low memory usage
- Early termination through traversal budget

Refining Match Edges



match ¼ pf[f] alias gf[f]

Most approximation can be refined

- Imprecise for recursive fields

Client-Driven Refinement Policy

Not clear when / where to refine

- Extra queries may be costly
- Refining match edge may not affect result

Client-driven: only refine when client affected

- E.g, multiple targets for virtual call
- Guyer and Lin SAS03

RefinedRegularPT:

- Refine edges traversed by RegularPT
- Iterate until client satisfied or budget exhausted

Experimental Hypotheses

1) Algorithms <u>precise with early termination</u>
   - Regular approximation reasonable
   - Refinement yields improved precision
2) Algorithms <u>meet performance goals</u>
   - Fast running time
   - Low memory

Evaluation Framework

Implemented in Soot / SPARK framework

Benchmarks: SPEC, Ashes, **`jedit`**

Clients

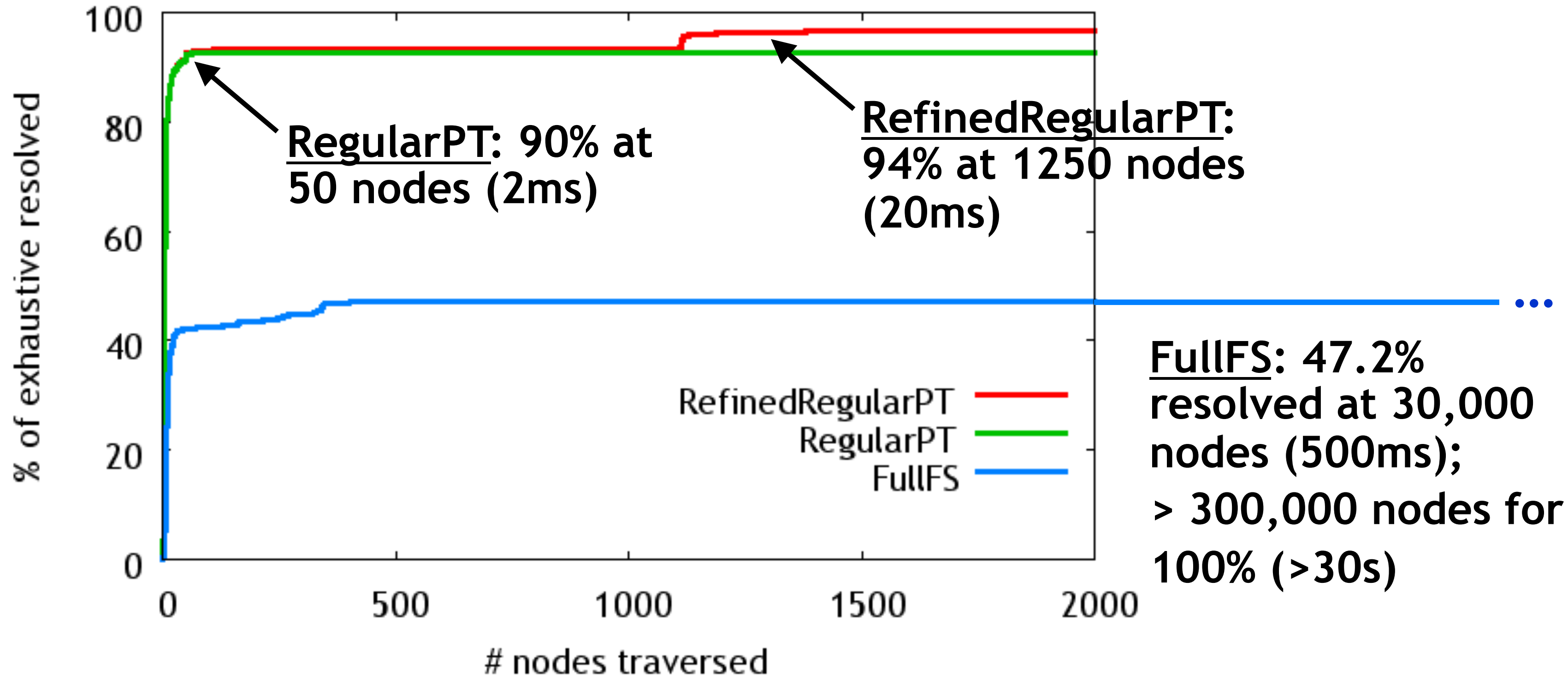
- IDE: Virtual call resolution
  - For program understanding
- JIT: queries from hot code
  - Virtual call resolution (for inlining)
  - Local aliasing (for load/store elimination)

Algorithms

Name	Field-based / sensitive	Demand / Exhaustive	Notes
RegularPT	Field-based	Demand	DFS Traversal
RefinedRegularPT	Variable up to partially field-sensitive	Demand	Client-driven refinement
FullFS	Field-sensitive	Demand	Heintze and Tardieu [PLDI01] adapted to Java
ExhaustiveFB	Field-based	Exhaustive	from SPARK
ExhaustiveFS	Field-sensitive	Exhaustive	from SPARK

1) Evaluation: Precision

jedit virtual calls



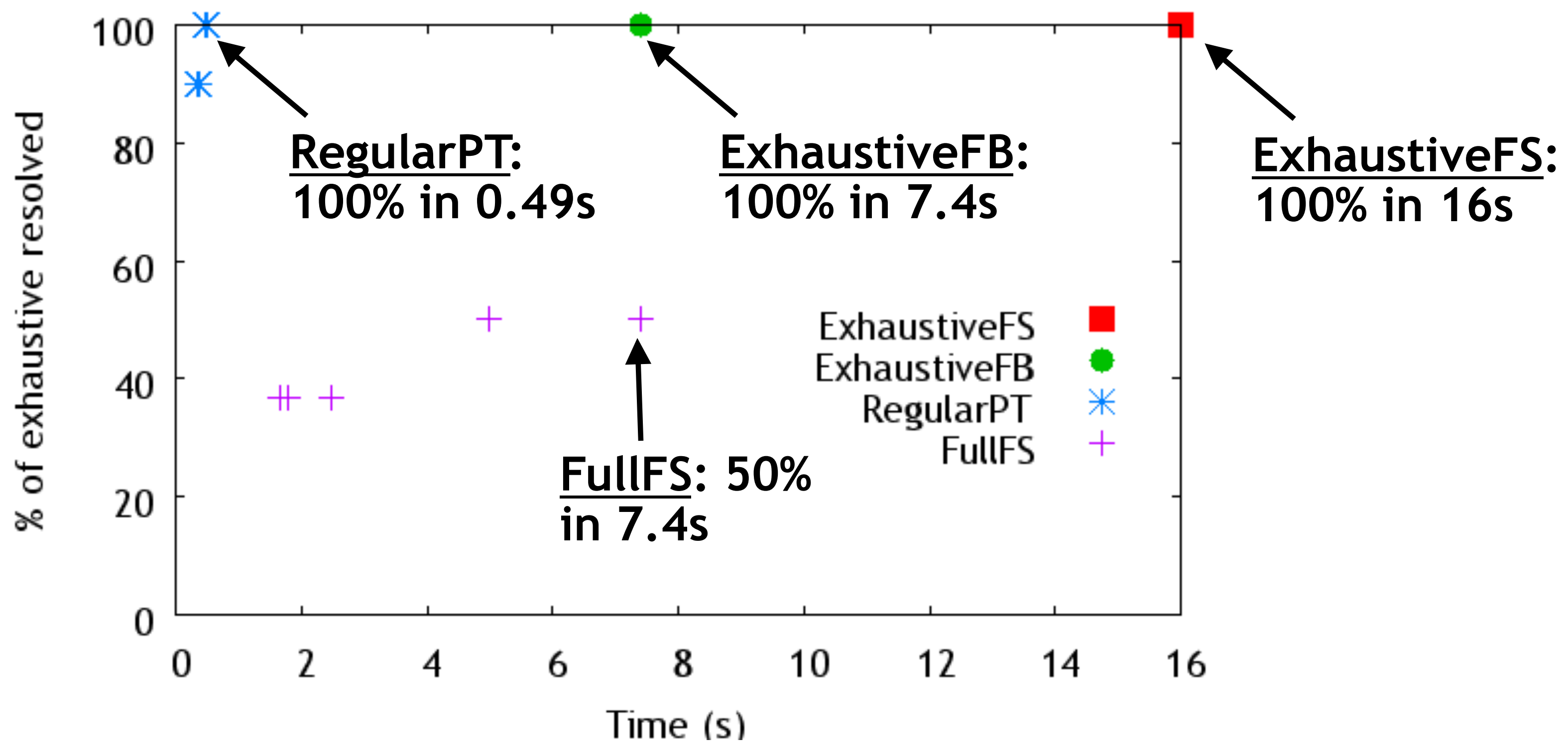
<u>RegularPT</u>: 90% at 50 nodes (2ms)

<u>RefinedRegularPT</u>: 94% at 1250 nodes (20ms)

<u>FullFS</u>: 47.2% resolved at 30,000 nodes (500ms); > 300,000 nodes for 100% (>30s)

2) Evaluation: Performance

`javac` virtual calls in hot code



Memory:

- < 50 KB for (Refined)RegularPT
- 28MB for FullFS using BDDs

Conclusions

New demand points-to analysis

- Speed through <u>two approximations</u>
  - Early termination
  - Regularization
- Refinement <u>driven by client</u>

Provide <u>high precision</u> in <u>tight budget</u>

Suitable for JITs, IDEs; and elsewhere?