Field-Sensitive Program Dependence Analysis

Nurit Dor, Panaya Inc.

Joint work with:
Shay Litvak, Panaya Inc & Tel-Aviv Univ. Mooly Sagiv, Tel-Aviv Univ.
Rastislav Bodik, UC Berkley. Noam Rinetzky, Queen Mary Univ. of London.
Agenda

• ERP systems
  • The need for tools
  • Challenges

• Field sensitive analysis
  • Algorithm
  • Empirical study

• Architecture
  • Usability
  • Analyzing large programs
ERP Systems

- Critical processes of the organization
- Cross-organization
- Highly integrated
- Database centric
- A few vendors
  - SAP
  - Oracle
  - ...
ERP Users

- Higher Education and Research
- Travel and Transportation
- Automotive and defense
- Telecommunications
- Manufacturing
- Public Sector
- Healthcare
- Banking
Business Specific Needs

- Packaged Application
  - Standard code base
  - Cannot use as-is
- Recommended Customization
  - Setting properties
  - Interaction with the system
  - Minimal

As long as the new system does exactly what our current system does we’re ready to move in.

We have a problem.
Minimal Customization?

- Customization
  - Adding new features
  - Changing existing features
- Up to hundreds in ERP group:
  - Business analysts
  - Programmers
  - IT + QA
- Go Live
  - Can be few times a month
Typical ERP Installation

- **Custom code**: 12K files, 3K programs
- **API**: 30K - 385K APIs
- **DataBase**: 2M files, 250M LOCs
- **Customization**: 30K settings
- **Standard Code Base**: 350K tables, 110 columns
- **Setting**: 30K programs
Problems in ERP Maintenance

- Regression testing
  - Replay is not possible
  - Dependent on the database
- System upgrade
  - Massive code change
    - Standard code base
    - API
    - Database schema changes
  - Clone finder
- Customization changes
Changing Customization

- **Process**
  - Need to accommodate some business requirements
  - Change the behavior of programs by external setting
  - What’s the impact?
  - What to test?
Changing Customization

• Impact analysis problem
  • A record in the database is changed
  • Find programs that reads this table
  • Is the specific column change impacting the program behavior?
Field Sensitive Dependence

Setting

- Large programs
- Very large aggregate structures

Purpose

- Impact analysis
  - Start point - select from customization table
  - What is impacted?
- Optimizations for other algorithms
  - Flow-insensitive field-sensitive data dependences
  - Constant propagation
  - Control flow analysis
Toy Example

- Field sensitive
  - Infer that res.f is dependent on seed.g
  - Infer that res.g is not dependent on seed.g

\[
\begin{align*}
  l_1: & \quad \text{seed} := \text{exp} \\
  l_2: & \quad i := \text{seed.f} \\
  l_3: & \quad j := \text{seed.g} \\
  l_4: & \quad \text{res.g} := i \\
  l_5: & \quad \text{res.f} := j \\
  l_7: & \quad \text{return res}
\end{align*}
\]
Current approaches

- Whole structure
  - No field sensitive
  - Easy implementation
  - Scalable but not precise
Current approaches

- **Whole structure**
  - No field sensitive
  - Easy implementation
  - Scalable but not precise

- **Atomization**
  - Treat structures as many simple variables
  - Field sensitive
  - Precise but not scalable
Transitive Dependences

• Partial kill operations
• Big L-value operation

l₁: s := exp1  
l₂: s.f2 := kill  
l₃: x := s  
l₄: t := x.f3  
l₅: a.f1 := t  
l₆: c := a  
l₇: b := c.f3  
l₈: d := c.f1
Algorithm Outline

• Compute interval-based reaching definition
  • Track for each substructure the last definition point
  • Substructure represented by interval (start - end offset)

• Build an interval-labeled Program Dependence Graph (PDG)
  • Records immediate control and data dependencies

• Compute transitive dependences
Interval Based Reaching Definition

1: $s := \text{expl}$
2: $s.f2 := \text{kill}$
3: $x := s$
4: $y.g2 := x$
5: $z := y$
6: $a := z.g2$
7: $b := z.g1$
Interval Based Reaching Definition

l₁: s := expl
l₂: s.f2 := kill
l₃: x := s
l₄: y.g2 := x
l₅: z := y
l₆: a := z.g2
l₇: b := z.g1

```
s f1 f2 f3
l₁ l₂ l₁
```

```
x f1 f2 f3
```

```
y g1 g2.f1 g2.f2 g2.f3
```

```
z g1 g2.f1 g2.f2 g2.f3
```
Interval Based Reaching Definition

l₁: s := exp1
l₂: s.f2 := kill
l₃: x := s
l₄: y.g2 := x
l₅: z := y
l₆: a := z.g2
l₇: b := z.g1
Interval Based Reaching Definition

1: s := exp1
2: s.f2 := kill
3: x := s
4: y.g2 := x
5: z := y
6: a := z.g2
7: b := z.g1
Interval Based Reaching Definition

\[ l_1: s := \text{exp1} \]
\[ l_2: s.f2 := \text{kill} \]
\[ l_3: x := s \]
\[ l_4: y.g2 := x \]
\[ l_5: z := y \]
\[ l_6: a := z.g2 \]
\[ l_7: b := z.g1 \]
Interval Based Reaching Definition

\[ l_1: s := \text{exp1} \]
\[ l_2: s.f2 := \text{kill} \]
\[ l_3: x := s \]
\[ l_4: y.g2 := x \]
\[ l_5: z := y \]
\[ l_6: a := z.g2 \]
\[ l_7: b := z.g1 \]
Interval Based Reaching Definition

\[ l_1: \quad s := \text{exp1} \]
\[ l_2: \quad s.f2 := \text{kill} \]
\[ l_3: \quad x := s \]
\[ l_4: \quad y.g2 := x \]
\[ l_5: \quad z := y \]
\[ l_6: \quad a := z.g2 \]
\[ l_7: \quad b := z.g1 \]
Transitive Dependence

\[ l_1: s := \text{exp1} \]
\[ l_2: s.f2 := \text{kill} \]
\[ l_3: x := s \]
\[ l_4: y.g2 := x \]
\[ l_5: z := y \]
\[ l_6: a := z.g2 \]
\[ l_7: b := z.g1 \]
Transitive Dependence

l₁: s := exp1
l₂: s.f2 := kill
l₃: x := s
l₄: y.g2 := x
l₅: z := y
l₆: a := z.g2
l₇: b := z.g1
Transitive Dependence

\[ l_1: s := \text{expl} \]
\[ l_2: s.f2 := \text{kill} \]
\[ l_3: x := s \]
\[ l_4: y.g2 := x \]
\[ l_5: z := y \]
\[ l_6: a := z.g2 \]
\[ l_7: b := z.g1 \]
Transitive Dependence

\[ l_1: s := \text{exp1} \]
\[ l_2: s.f2 := \text{kill} \]
\[ l_3: x := s \]
\[ l_4: y.g2 := x \]
\[ l_5: z := y \]
\[ l_6: a := z.g2 \]
\[ l_7: b := z.g1 \]
Transitive Dependence

1. \( s := \exp1 \)
2. \( s.f2 := \text{kill} \)
3. \( x := s \)
4. \( y.g2 := x \)
5. \( z := y \)
6. \( a := z.g2 \)
7. \( b := z.g1 \)
Avoiding Spurious Dependences

l_1: s := exp1
l_2: s.f2 := kill
l_3: x := s
l_4: y.g2 := x
l_5: z := y
l_6: a := z.g2
l_7: b := z.g1
Avoiding Spurious Dependences

1: \( s := \text{exp1} \)
2: \( s.f2 := \text{kill} \)
3: \( x := s \)
4: \( y.g2 := x \)
5: \( z := y \)
6: \( a := z.g2 \)
7: \( b := z.g1 \)
l₁: s := exp1
l₂: s.f2 := kill
l₃: x := s
l₄: t := x.f3
l₅: a.f1 := t
l₆: c := a
l₇: b := c.f3
l₈: d := c.f1
Compute Transitive Dependences

- Which segments are dependent on the (partial) definition of seed
- Iterative algorithm over the PDG
- Tracks also the seed interval dependence
  - Which segment of the used variable is dependent on which segment of the seed
- Efficient representation
  - Need to record only according to statement
Compute Transitive Dependences

1. \( l_1: s := \text{exp1} \)  
   \( \{<s, s>\} \)

2. \( l_2: s.f2 := \text{kill} \)

3. \( l_3: x := s \)
   \( \{<s.f1, s.f1>,<s.f3,s.f3>\} \)

4. \( l_4: t := x.f3 \)
   \( \{<x.f3,s.f3>\} \)

5. \( l_5: a.f1 := t \)
   \( \{<t,s.f3>\} \)

6. \( l_6: c := a \)
   \( \{<a.f1,s.f3>\} \)

7. \( l_7: b := c.f3 \)
8. \( l_8: d := c.f1 \)
   \( \{<c.f1,s.f3>\} \)
Empirical Study

- Compare three algorithms:
  - ATOM - full atomization (code level atomization)
  - WS - whole structure (field insensitive)
  - FSD - field sensitive dependences
- Analyzing 12 Programs (3K - 419K LOCs)
  - ATOM failed on largest one
- Computer grid of 5 Intel Servers
  - Two dual core CPUs, 16GB RAM
  - Window XP 64-bit, Java 5.0
Precision Results

- ATOM and FSD produce the same results
- WS has a false positive rate of 62%
Performance - Running Time

- WS averages at 30 min
- ATOM more than double at 60 min
- FSD shows small increase at 35 min
Performance - Memory

- ATOM footprint 50% larger than WS
  - Failed on the largest program (419K LOCs)
- FSD footprint only 10% larger
Related work

• Aggregate Structure Identification
  • POPL99: Ramalingam, Field, Tip
    • Avoid atomization for unused parts
  • Only 22% of the fields are unused
  • Almost full atomization
  • Large programs < 15% unused fields

• Procedure summary
  • TOPLAS90: Horwitz, Reps, Binkley
  • FSE94: Horwitz, Reps, Sagiv, Rosay
Architecture

- Software-As-A-Service (SaaS) Application
- Wide and efficient use of cloud computing
  - Multi-process
- Deep static analysis techniques
Analysis Challenges: Control Flow

- Interdependency and sharing of code
  - Modular analysis (assembly - link)
  - Eliminating dead code - saves 30%
- Call by name which is resolved at runtime
  - Find possible targets using constant propagation
- Boundaries of programs are unclear
  - Using heuristics to put logical boundaries
- Programs have large cyclic parts
  - Using procedure cloning for specialization
  - Using SCCs for optimizing chaotic iterations
Analysis Challenges: Data Flow

- Extensive declaration of global variables
  - Localization algorithm eliminates 80%
- Very large aggregate types
  - Field sensitive dependences
Conclusion

- Field sensitive algorithm
  - Importance in the presence of large aggregate structures
- ERP systems
  - Large critical systems
  - In need for solutions
    - Various impact analysis
    - Clone finder
    - Regression testing
Questions?

How to Contact Me:
Nurit Dor, nurit@panayainc.com
Upgrade

• Massive code change
  • Standard code base
  • API
  • Database
• Requirements
  • What needs to be fixed
  • What is used
  • Testing plan

• Example SAP upgrade:
  • 46% of the code changed
  • 5K-7K: API changes
Control dependences

• Using control dependency edges on the PDG
• Non-value dependences
• “Freeze” the seed

\[
\begin{align*}
\text{l}_1 & : s := \text{exp1} && \{<s, s, \text{true}>\} \\
\text{l}_2 & : t := s.f3 && \{s.f3^*, s.f3, \text{true}\} \\
\text{l}_3 & : \text{if } (t > 0) \\
\text{l}_4 & : c := a && \{a, s.f3, \text{false}\} \\
\text{l}_5 & : \text{fi} \\
\text{l}_8 & : d := c.f1 && \{c.f1^*, s.f3, \text{false}\}
\end{align*}
\]
procedure bar(a, b)
    t := a.f3
    b.f2 := t
return

procedure foo(s, t)
    x := s
    y := exp
    call bar(x, y)
    t := y
return
Inter-Procedural Analysis

\[
\begin{align*}
s &= s_{in} \\
x &= s \\
a_{in} &= x \\
y &= \exp \\
b_{in} &= y \\
b_{out}.f2 &= a_{in}.f3 \\
x &= a_{out} \\
y &= b_{out} \\
t &= y \\
t_{out} &= t
\end{align*}
\]
Programming Language

- **SAP - ABAP**
  - Constantly Developed
  - Started as a report generator in the 70’s

- **Oracle - PL/SQL**
  - Many frameworks for developing programs
    - Reports
    - Forms
    - …
  - Satellites
Clone

```
FORM TDNAME_SETZEN USING I_TDOBJECT
    CHANGING I_TDNAME.
    CLEAR I_TDNAME.
    CASE I_TDOBJECT.
        WHEN 'EINA'.
            IF FC_CALL EQ SPACE OR EINA-INFNR NE SPACE.
                I_TDNAME(10) = EINA-INFNR.
            ELSE.
                I_TDNAME(1) = '$'.
                I_TDNAME+1(10) = EINA-LIFNR.
                I_TDNAME+11(18) = EINA-MATNR.
            ENDIF.
        WHEN 'EINE'.
            IF FC_CALL EQ SPACE OR EINA-INFNR NE SPACE.
                I_TDNAME(10) = EINA-INFNR.
                I_TDNAME+10(4) = EINE-EKORG.
                I_TDNAME+14(1) = EINE-ESOKZ.
                I_TDNAME+15(4) = EINE-WERKS.
            ELSE.
                I_TDNAME(1) = '$'.
                I_TDNAME+1(10) = EINA-LIFNR.
                I_TDNAME+11(18) = EINA-MATNR.
                I_TDNAME+29(4) = EINE-EKORG.
                I_TDNAME+33(1) = EINE-ESOKZ.
                I_TDNAME+34(4) = EINE-WERKS.
            ENDIF.
    ENDCASE.
ENDFORM.
```

THEADTAB-TDOBJECT = 'EINE'.
THEADTAB-TDID = 'BT'.
PERFORM TDNAME_SETZEN USING THEADTAB-TDOBJECT
    THEADTAB TDNAME.
READ TABLE THEADTAB.
    .
    .
    .
FORM konditionen USING i_kschl
...
IF kond_exit NE space AND fc_call
  EQ space.
  ok-code = endb.
  PERFORM okcode.
ENDIF.
...
ENDFORM

FORM okcode.
...
CASE ok-code
...
WHEN endb.
  IF sy-dyngr EQ 'ANFO'.
    IF sy-calld NE space OR sy-binpt NE space.
      LEAVE.
    ELSE.
      PERFORM return_to_menu.
    ENDIF.
  ELSE.
    PERFORM info_ende.
  ENDIF.
ELSE.
  PERFORM konditionen USING space.
ENDFORM

WHEN kond.
  PERFORM konditionen USING space.

ENDCASE
..
ENDFORM
Program Boundaries

- Spaghetti code
  - No clear boundaries
  - One activates another
  - Lots of shared code with multiple behaviors
- We studied 160 programs, with 94,000 procedures
  - On average each procedure is potentially called in 30 programs
  - Half the procedures are only called by a single program
  - Half are called by more than 1/3 of the programs
- Heuristics based on components to decide what to include in which program
Modular Analysis

- Each file is analyzed once independently of calling context
- Whole program is “linked” and analyzed