Advanced Program Analyses for Object-oriented Systems

Dr. Barbara G. Ryder
Rutgers University
http://www.cs.rutgers.edu/~ryder
http://prolangs.rutgers.edu/

July 2007

Lecture 5 - Outline

- Uses of analysis in software tools
  - For performance diagnosis, program understanding, slicing and testing
  - **Idea:** choose an appropriate analysis for a problem
  - Projects:
    1. Testing recovery code in Java programs
    2. Accurate interclass test dependence to guide integration testing
    3. Blended (static & dynamic) analysis for performance diagnosis

Thanks to my graduate students: Bruno Dufour, Chen Fu and Weilei Zhang for help with these slides.

Testing Robustness of Java SW

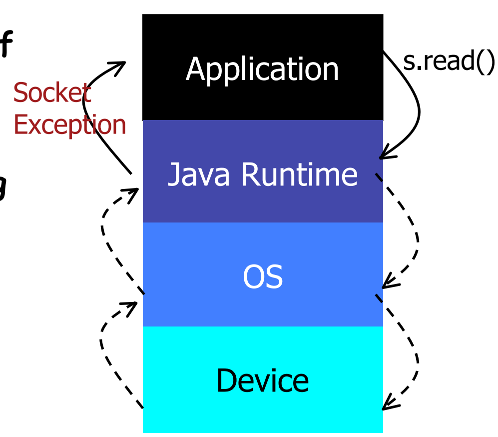
Idea

- Use static analysis to find exception handling paths in Java program fairly accurately;
- Use fault injection engine to test exception handling through communication with instrumented Java program
- Replace current black box testing approach

C. Fu, A. Milanova, B.G. Ryder, D. Wonnacott, "Robustness Testing of Java Server Applications", IEEE TSE, 2005;
C. Fu, B.G. Ryder, "Exception-chain Analysis: Revealing Exception Handling Architecture in Java Server Applications", ICSE'07;

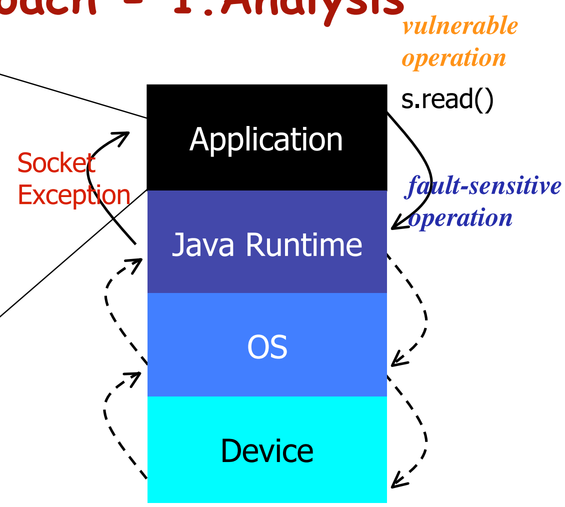
Testing by Fault Injection

The approach, a form of white box testing, uses knowledge of application from the compiler to inject possibly-affecting faults and record their handling



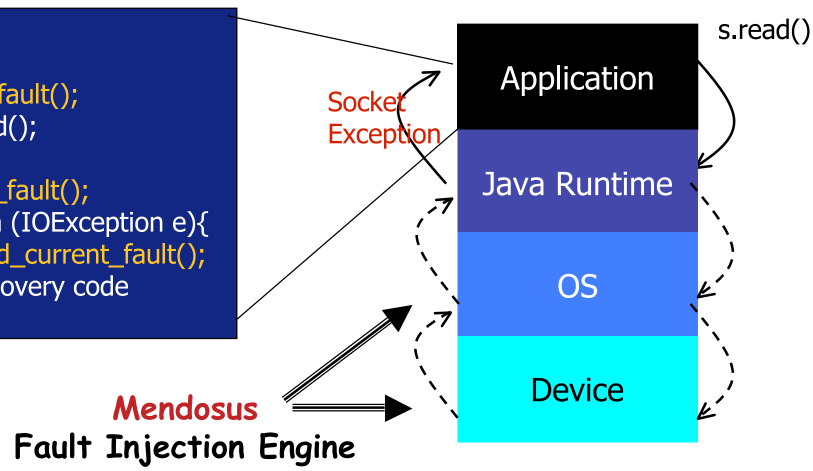
Our Approach - 1.Analysis

```
try{
  …

  s.read();
  …

} catch (IOException e){
  // recovery code
}
```



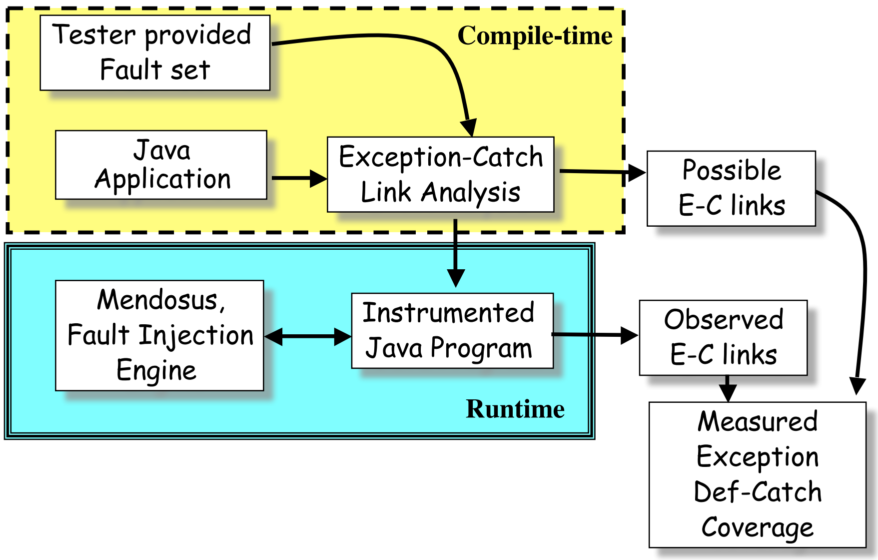
Our Approach - 2.Instrumentation

```
try{
  …
inject_fault();
  s.read();
  …
cancel_fault();
} catch (IOException e){
  record_current_fault();
  // recovery code
}
```



C. Fu et al, TSE'05

Framework Built



Exception-catch Link Analysis

- **Two phase algorithm**
  - Exception-flow analysis - initial estimate of e-c links
  - DataReach analysis - prune away links corresponding to infeasible call paths, using points-to information
- **Need fairly accurate call graph to trace exception handling up the call stack**
  - Techniques - CHA, RTA, FieldSens - determine precision of analysis
  - Whole-program dataflow analysis
    - Backward propagation on call graph from exception throws to `catch` block handlers
    - Intraprocedural propagation maintains ordering between `try` blocks; partially flow-sensitive, context-insensitive

C. Fu et al, TSE'05

DataReach Analysis

```java
void readFile(String s){
 byte[] buffer = new byte[256];
 try{
  InputStream f =new FileInputStream(s);
  InputStream source=new BufferedInputStream(f);
  for (...)
   c = source.read(buffer);
 }catch (IOException e){ ...}
}
```

```java
void readNet(Socket s){
 byte[] buffer = new byte[256];
 try{
  InputStream n =s.getInputStream();
  InputStream source=new BufferedInputStream(n);
  for (...)
   c = source.read(buffer);
 }catch (IOException e){ ...}
}
```

DataReach Analysis

readFile(String s)

readNet(Socket s)

FilterInputStream.read(byte[])

BufferedInputStream.read(byte[],int,int)

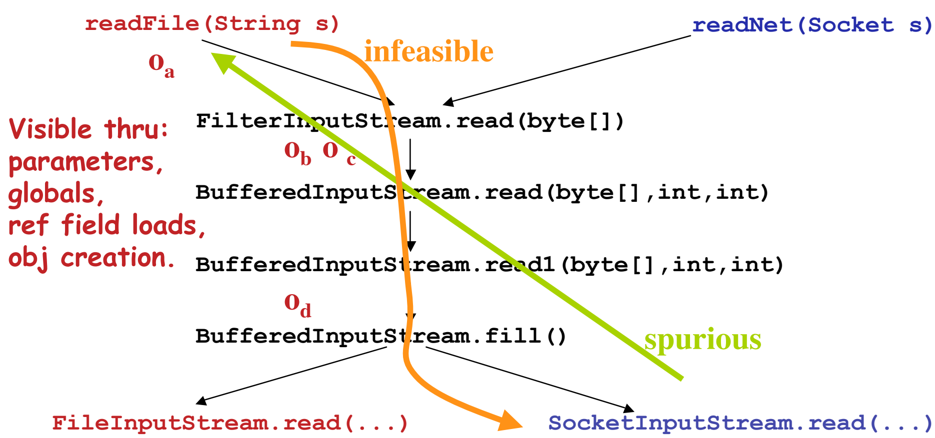
BufferedInputStream.read1(byte[],int,int)

BufferedInputStream.fill()

FileInputStream.read(...)

SocketInputStream.read(...)

DataReach Analysis



Q: what objects are visible in BufferedInputStream.fill() on paths from readFile()?

Experiment Benchmarks

Name	*Classes*	*Methods*	*Try Blocks*	*.class Size*
FTPD	11(1407)	128(7479)	17	39,218
JNFS	56(1664)	447(9603)	36	175,297
Muffin	278(1365)	2080(7677)	270	727,118
Haboob	338(1403)	1323(7432)	134	731,413
HttpClient	252(2210)	1334(4741)	536	1,049,784
SpecJVM	484(2161)	2489(4592)	219	2,817,687
VMark	307(2266)	1565(5029)	502	2,902,947

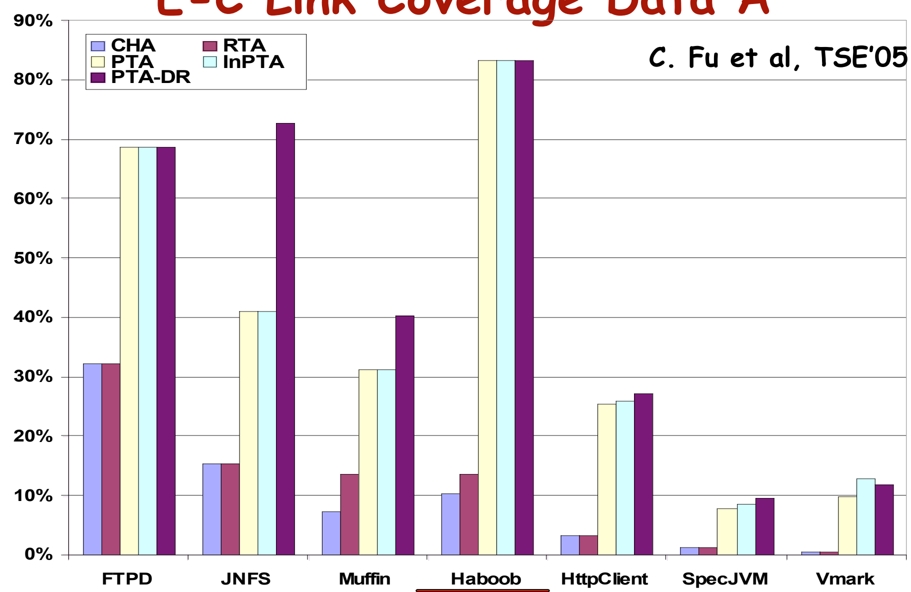
C. Fu et al, TSE'05

Experiment Details

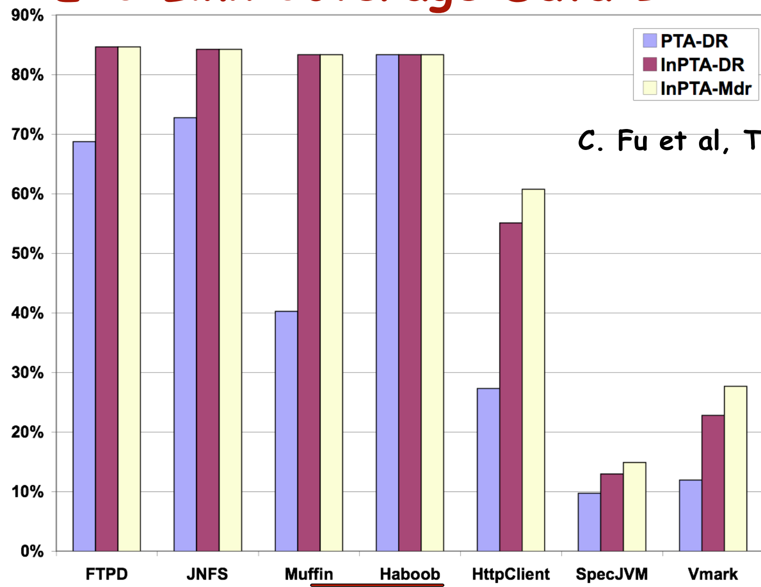
- **Analysis combinations tried:**
  CHA, RTA, FieldSens, In-Points-to, Points-to/DataReach, In-Points-to/DataReach, In-Points-to/MDataReach
- **Measured coverage as ratio of number of executed e-c links to number of possible e-c links**
- **Added some context sensitivity by inlining constructors that assign to reference fields through the `this` parameter**

C. Fu et al, TSE'05

E-C Link Coverage Data A



E-C Link Coverage Data B



C. Fu et al, TSE'05

Uncovered E-C Links

- Categorized uncovered e-c links
  1. Feasible, but uncovered because of insufficient tests or input
  2. Infeasible and difficult to prune for any static analysis
  3. Infeasible, but could be eliminated by a more precise static analysis

	1	2	3	Total
Muffin	1(14%)	3(43%)	3(43%)	7
SpecJVM		4(13%)	26(87%)	30
HTTPClient	10(25)%	24(60%)	6(15%)	40

C. Fu et al, TSE'05

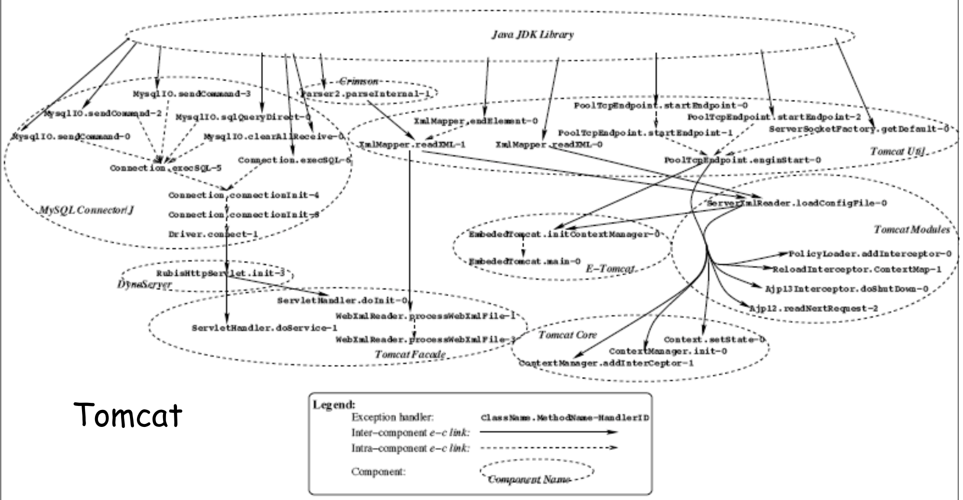
Exception-Catch Chains

- Semantically-related exceptions
  - Preserve state of original exception object in new exception throw, or
  - Re-throw original exception object
  - Called re-thrown exceptions
- E-c chains of re-thrown exceptions
  - Discover through flow-sensitive analysis of each catch clause
  - Link together exception handling paths of re-thrown exceptions to find e-c chains

E-C Chains found

Lengths: Pgms:	1	2	3	4	5	6	Total
Muffin	6						6
SpecJVM	69	46					115
Vmark	300	81	12				393
Tomcat	312	365	31	3	2	10	723
HttpClient	583	547	275				1405

E-C Chains span components



Tomcat

Figure 10: E-c chain Graph of Tomcat

Fu & Ryder, ICSE'07

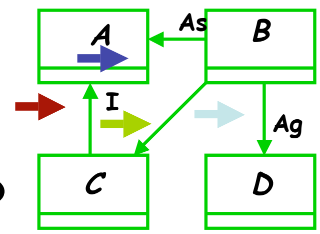
Improving Class-based Testing

- Class testing for OOPLs is unit testing
- Classes can be interdependent imposing an order on how best to integrate and test them
  - Dependence cycles can exist -- solve by stubs or big-bang
- Deriving more accurate Interclass Test Dependence (ICTD) through code analysis can help
  - May allow parallelization of integration test; more testing in same time period
  - ICTD also can be useful for program understanding and software visualization

W. Zhang & B.G. Ryder, "Discovering Accurate Interclass Test Dependences", PASTE'07

ORD-Based Definition

- Causes for ICTD:
  - *Inheritance*
  - *Aggregation (part-of)*
  - *Association (uses)*
  - *Polymorphism*



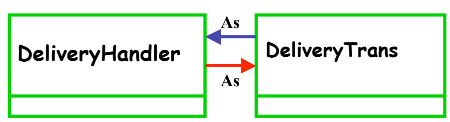
B is test dependent on A, C and D

Transitive closure of the above

Sometimes there are spurious cycles, and associations which are not semantic dependences at runtime

Motivating Example

specjbb benchmark
(class names shortened)




```
class DeliveryHandler{
  public void handleDelivery
            (DeliveryTrans deliveryTrans){
    deliveryTrans.process();
    deliveryTrans.display(outFile);
  }
}
```

Red arrow because handleDelivery calls DeliveryTrans methods `process()` and `display()` but they have no influence on the execution of their caller

Blue arrow because there is a valid dependence of DeliveryTrans on DeliveryHandler

Semantics-based Definition of ICTD

- **There is test dependence from class *A* to class *B*, if there is a statement *s* in a method <u>callable</u> on an *A* object and a statement *t* in a method <u>declared</u> in *B*, such that:**
  - ***s* may have visible side effects (i.e., *s* may either write to the external memory or return a value), and**
  - ***s* is semantically dependent on *t* <u>while testing class *A*</u>**

Zhang & Ryder, PASTE'07

Key Ideas

- **Approximate at method-level granularity for scalability**
- **Three causes for method dependence:**
  - **Caller uses return value of callee**
  - **Callee is control-dependent on caller**
  - **Side effect: one method writes to the same memory region that another method reads**
    - **Requires reference analysis to identify region**
- **Propagate dependences on the call graph**

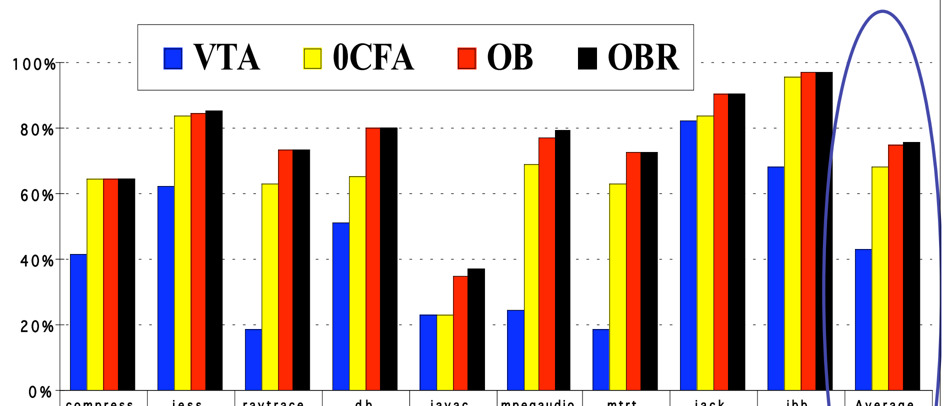
Zhang & Ryder, PASTE'07

Analysis Configurations

- Algorithm parameterized by choice of analyses to calculate the call graph and side effects
- Four analysis configuration applied:
  - VTA: variable type analysis (call graph not constructed on the fly)
  - 0CFA: 0-CFA (call graph constructed on the fly)
  - OB: 1-object-sensitive points-to analysis
  - OBR: OB+R(i.e., DataReach)

Zhang & Ryder, PASTE'07

ICTD Reduction Rate wrt. ORD-based Definition



	compress	jess	raytrace	db	javac	mpegaudio	mtrt	jack	jbb	Average
VTA	41.28%	62.42%	18.62%	50.83%	23.03%	24.51%	18.62%	82.34%	68.08%	43.30%
0CFA	64.53%	83.59%	63.24%	65.00%	23.03%	68.89%	63.24%	83.53%	95.38%	67.83%
OB	64.53%	84.73%	73.19%	80.00%	34.58%	76.98%	72.87%	90.55%	96.75%	74.91%
OBR	64.53%	85.11%	73.19%	80.00%	37.26%	79.51%	72.87%	90.55%	96.83%	75.54%

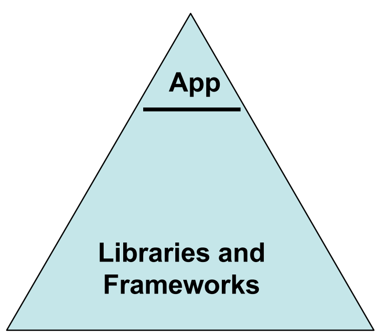
Zhang & Ryder, PASTE'07

Zhang & Ryder, PASTE'07

Size of Dependence Cycles

	ORD	VTA	0CFA	OB	OBR
compress(128)	6,4,4	5,3	*3*	*3*	*3*
jess(163)	147,4	139	96	94	90
raytrace(41)	18,4,2	16	8,4	*7*	*7*
db(20)	10	6	5	*0*	*0*
javac(184)	169	161	161	142	134
mpegaudio(60)	44	37	6,3,2	*0*	*0*
mtrt(41)	18,4,2	16	8,4	*7*	*7*
jack(69)	44	7,6,2,2	7,4,2,2	7	7
jbb(104)	79	49	3,3,2	2,2	*2*

Blended Analysis for Performance Diagnosis



- Framework-intensive applications look like *icebergs* to developers
- Problematic activity usually spans multiple *frameworks*
- Long call chains across multiple frameworks often lead to *object churn*
  - Want to identify these
- Current profiling tools focus on object creation rather than object use (e.g. Jinsight, ArcFlow, HPROF)

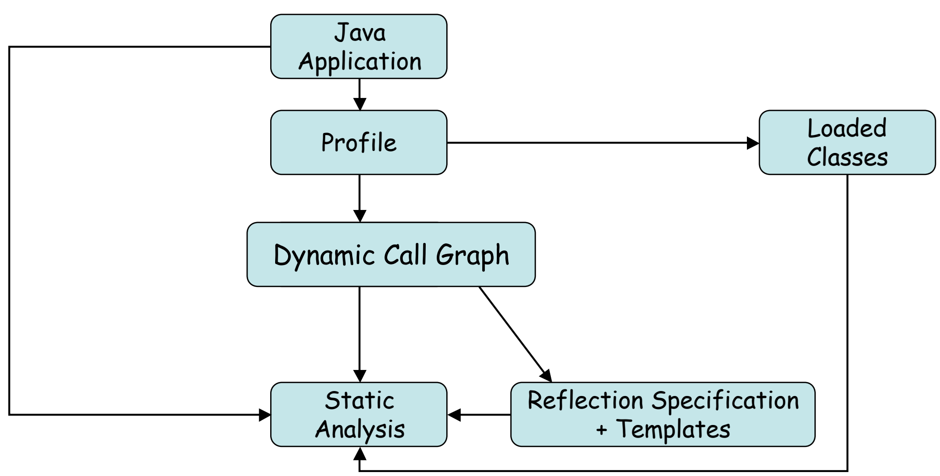
B. Dufour, B.G. Ryder, G. Sevitsky, "Blended Analysis for Performance Understanding of Framework-based Applications", ISSTA'07

Blended Analysis Paradigm

- Need information about a set of program executions
  - Full dynamic analysis is too expensive and too intrusive for production codes
  - Static analysis is too conservative and likely not to scale
- IDEA: Use dynamic analysis to obtain a calling structure of interest to use in a subsequent static analysis
  - Avoids intrusiveness and problems with dynamic loading
  - Dynamic analysis can selectively collect additional useful information (e.g., object creations)

Blended Analysis Paradigm

Dufour et al, ISSTA'07



Definitions

- **Effective lifetime:** period between an object's creation and its last use
- **Allocation context:** method invocations on runtime stack during object allocation
- **Escape:** An object *escapes* method *f()*, if *f()* is in its allocation context and the object can be accessed beyond the lifetime of the invocation of *f()*.
- **Capture:** An object is *captured* by method *g()* if *g()* is in its allocation context and the object cannot be accessed beyond the lifetime of the invocation of *g()*.

Escape Analysis

- Determines escape status of an object at compile time:
  - *Globally escaping*
  - Escaping through arguments and return values (*arg-escaping*)
  - Non-escaping (*captured*)
- Traditional uses in compilation:
  - On-stack allocation
  - Synchronization removal

Global escaping ($\top$)

Arg-escaping

Captured ($\perp$)

Blended Escape Analysis

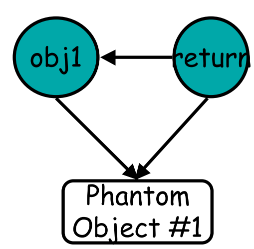
- Based on escape analysis in Choi et. al,TOPLAS'03
- Connection graphs
  - Nodes represent objects, fields and references
  - Edges represent points-to relationships
  - Abstract objects are allocation sites
  - Modified to keep a *distinct escape state* for each object at each node in the calling structure
- Flow-sensitive, context-insensitive, field-sensitive analysis
- Now for an example of static escape analysis…

Example

```
public X identity(X obj1){
  return obj1;
}
```

```
public X escape(X obj2){
  G.global = obj2;
  return obj2;
}

public void f() {
  X inst;
  if (...) inst = identity(new Y())
  else inst = escape(new Z());
}
```

Connection Graph

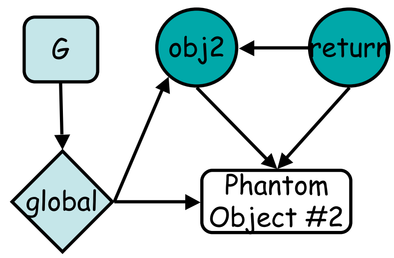


Preserves escape state of actual argument

Example

```
public X identity(X obj1){
  return obj1;
}

public X escape(X obj2){
  G.global = obj2;
  return obj2;
}

public void f() {
  X inst;
  if (...) inst = identity(new Y())
  else inst = escape(new Z());
}
```

Connection Graph

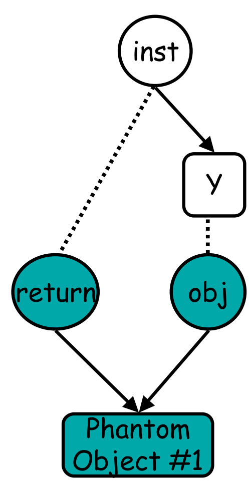


Makes actual argument globally escaping

Example

```
public X identity(X obj){
  return obj;
}

public X escape(X obj){
  G.global = obj;
  return obj;
}

public void f() {
  X inst;
  if (...) inst = identity(new Y())
  else inst = escape(new Z());
}
```

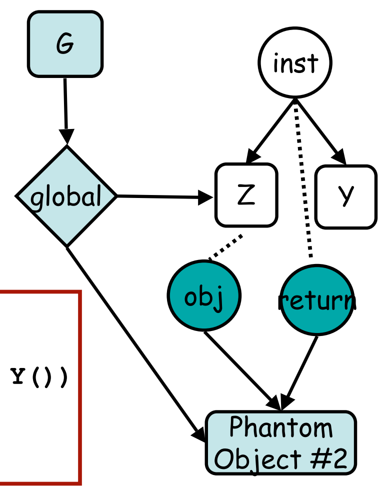
Connection Graph



Example

Connection Graph

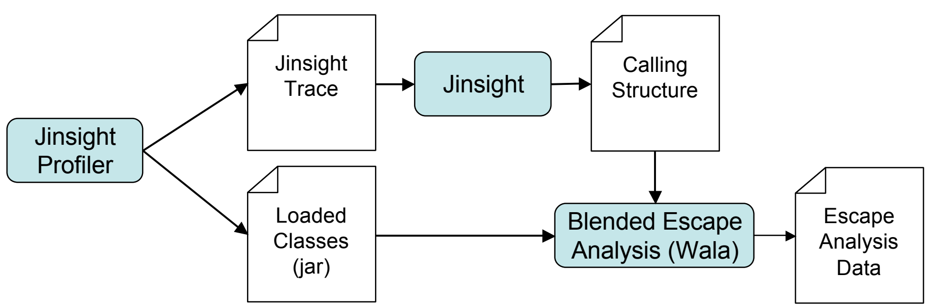
```
public X identity(X obj){
  return obj;
}

public X escape(X obj){
  G.global = obj;
  return obj;
}

public void f() {
  X inst;
  if (...) inst = identity(new Y())
  else inst = escape(new Z());
}
```



z is globally escaping
y is captured in f()

Implementation

Dufour et al, ISSTA'07



Benchmarks

- Software
  - Trade 6.0.1
  - Websphere Application Server 6.0.0.1
  - DB2 v8.2.0
- 4 configurations of Trade6
  - Run-time mode: Direct, EJB
  - Access mode: Standard, WebServices
- Tracing a single login transaction after warm-up

Size Comparison for Configurations

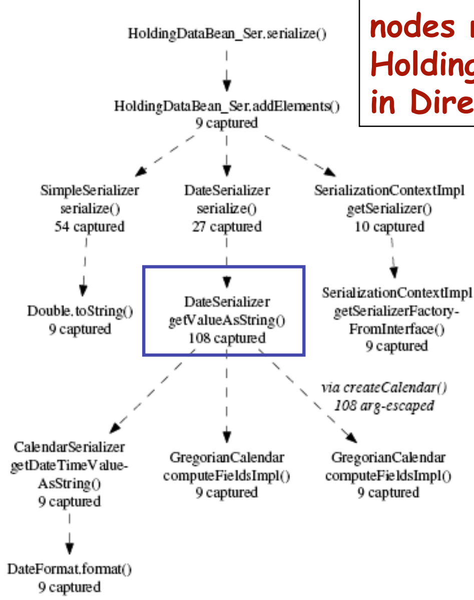
Scenario	Config	Methods	Invocs	Max Stack	Type	Insts	Abs. Objects
getHoldings	Direct-Std	710	4,484	26	30	186	549
	Direct-WS	3,308	127,794	53	166	5,522	2,517
	EJB-Std	1,978	60,936	62	82	1,751	1,834
	EJB-WS	4,479	184,288	72	210	7,088	3,747

getHoldings: Retrieves user's portfolio from a database

Reduced Connection Graphs

- Use additional calling context tree (CCT) data from Jinsight about object instances created, to prune uninteresting object nodes from the connection graphs
- Sum the number of remaining instances in each escape state at method node in the CCT
- Show only CCT nodes that capture object instances, to try to understand use of temporaries
- Used in performance diagnosis

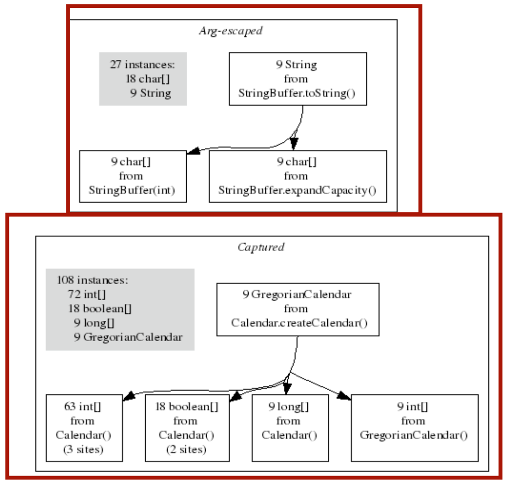
CCT showing only capturing nodes reachable from HoldingDataBean_Ser.serialize in Direct-WS config



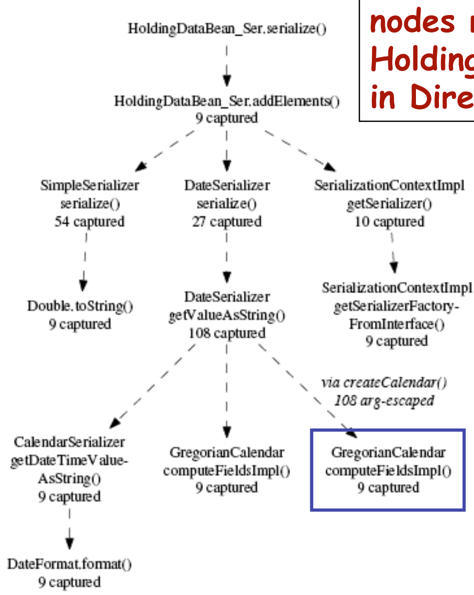
Function of CCT: formatting stock records into response portion of SOAP message

262 instances captured in CCT

Reduced Connection Graph for DateSerializer.getValueAsString()



CCT showing only capturing nodes reachable from HoldingDataBean_Ser.serialize in Direct-WS config

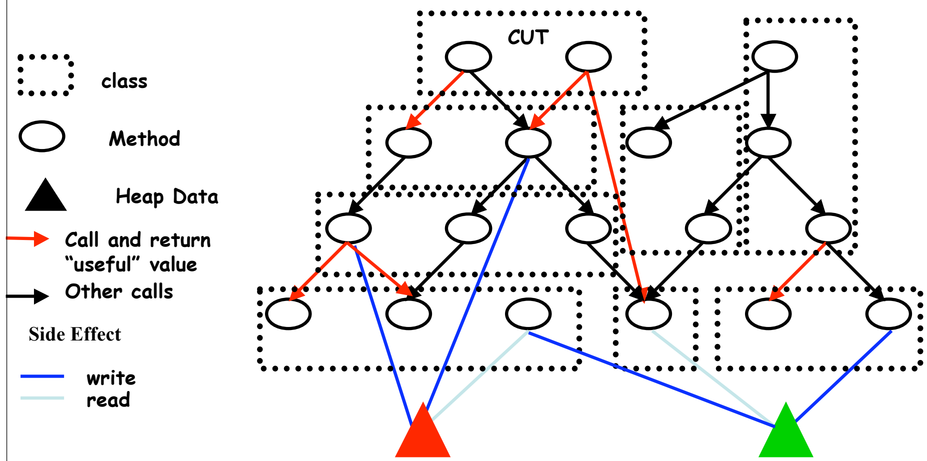


262 instances captured in CCT

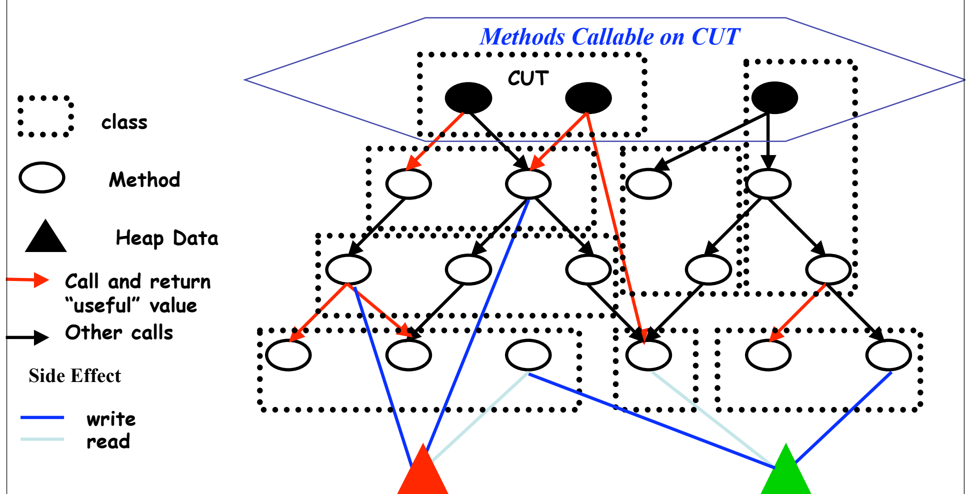
Summary

- **Presented 3 research projects using program analysis for testing, performance diagnosis and program understanding**
  - **Showed different cost/benefit tradeoffs for analyses used**
  - **Demonstrated strength of using static and dynamic analyses together**
  - **Illustrated the need for empirical investigation with accepted benchmarks**

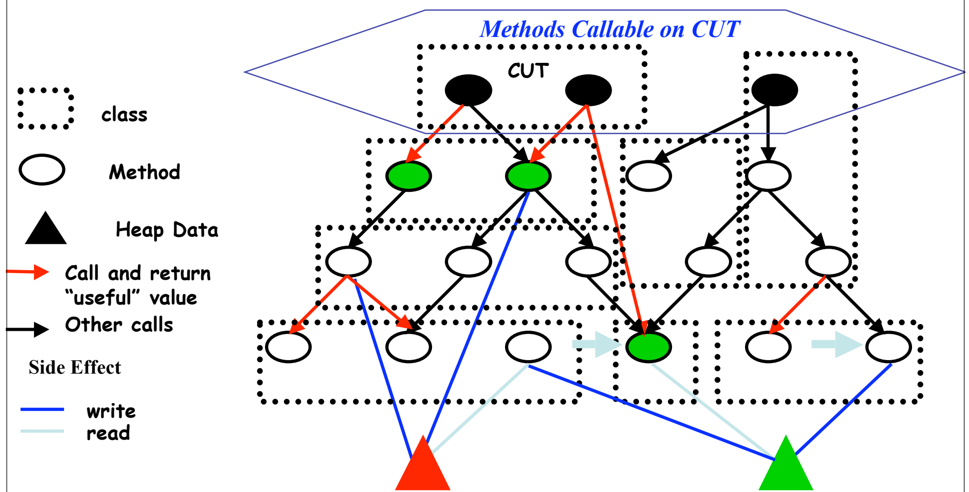
Algorithm Illustration



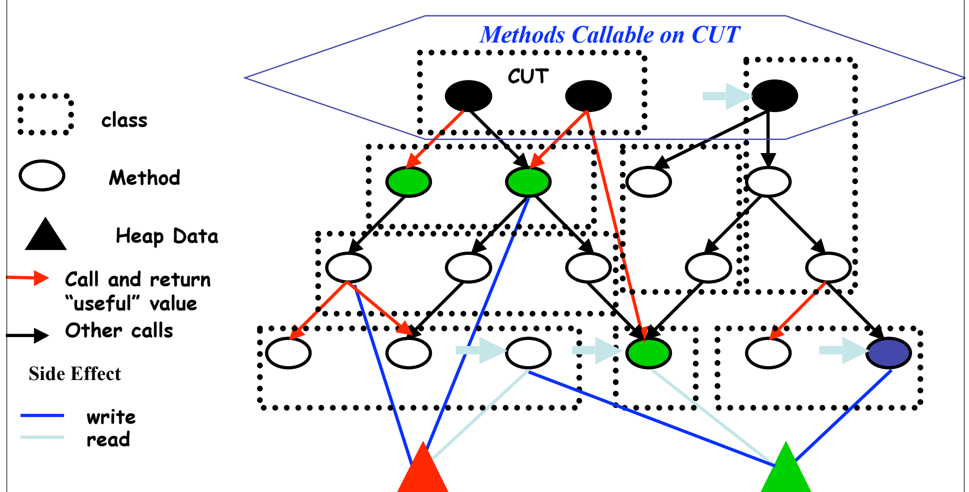
Algorithm Illustration



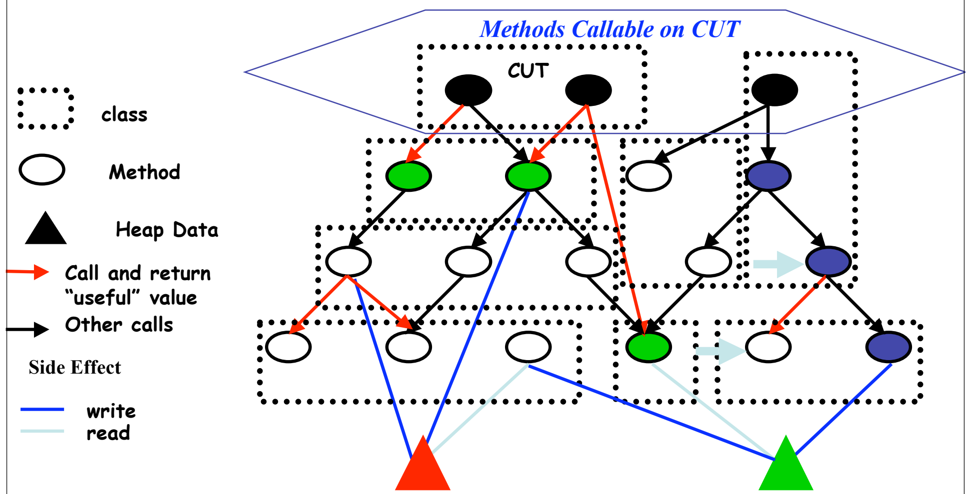
Algorithm Illustration



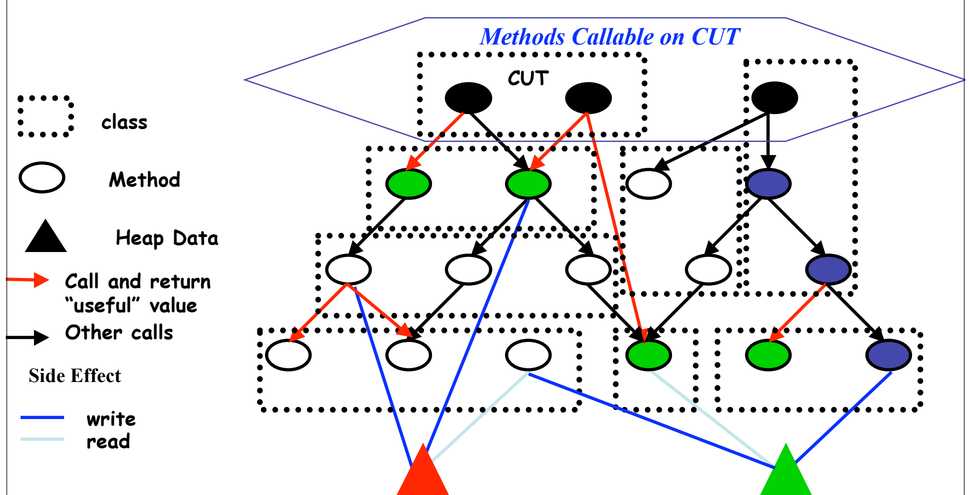
Algorithm Illustration



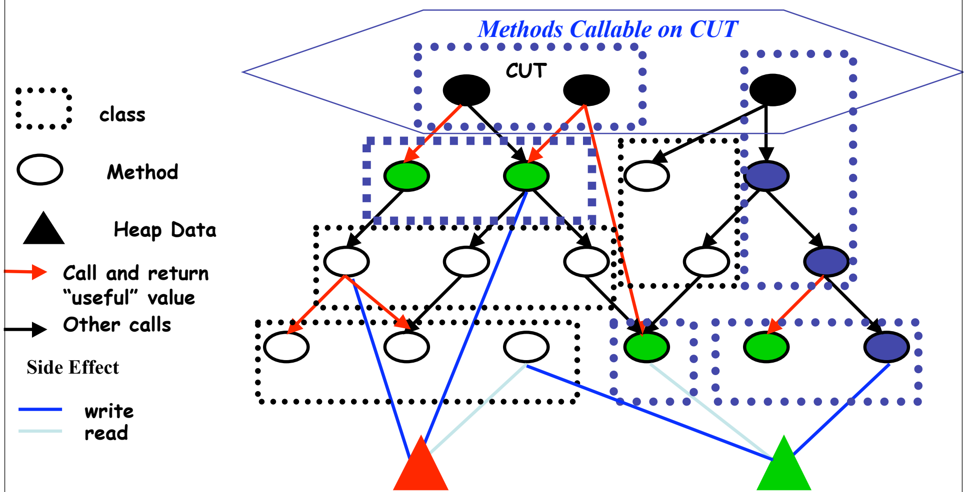
Algorithm Illustration



Algorithm Illustration



Algorithm Illustration



Breakdown of Abstract Objects by Escape State