

Analyzing Large-Scale Object-Oriented Software to *Find and Remove* Runtime Bloat

Guoqing Xu

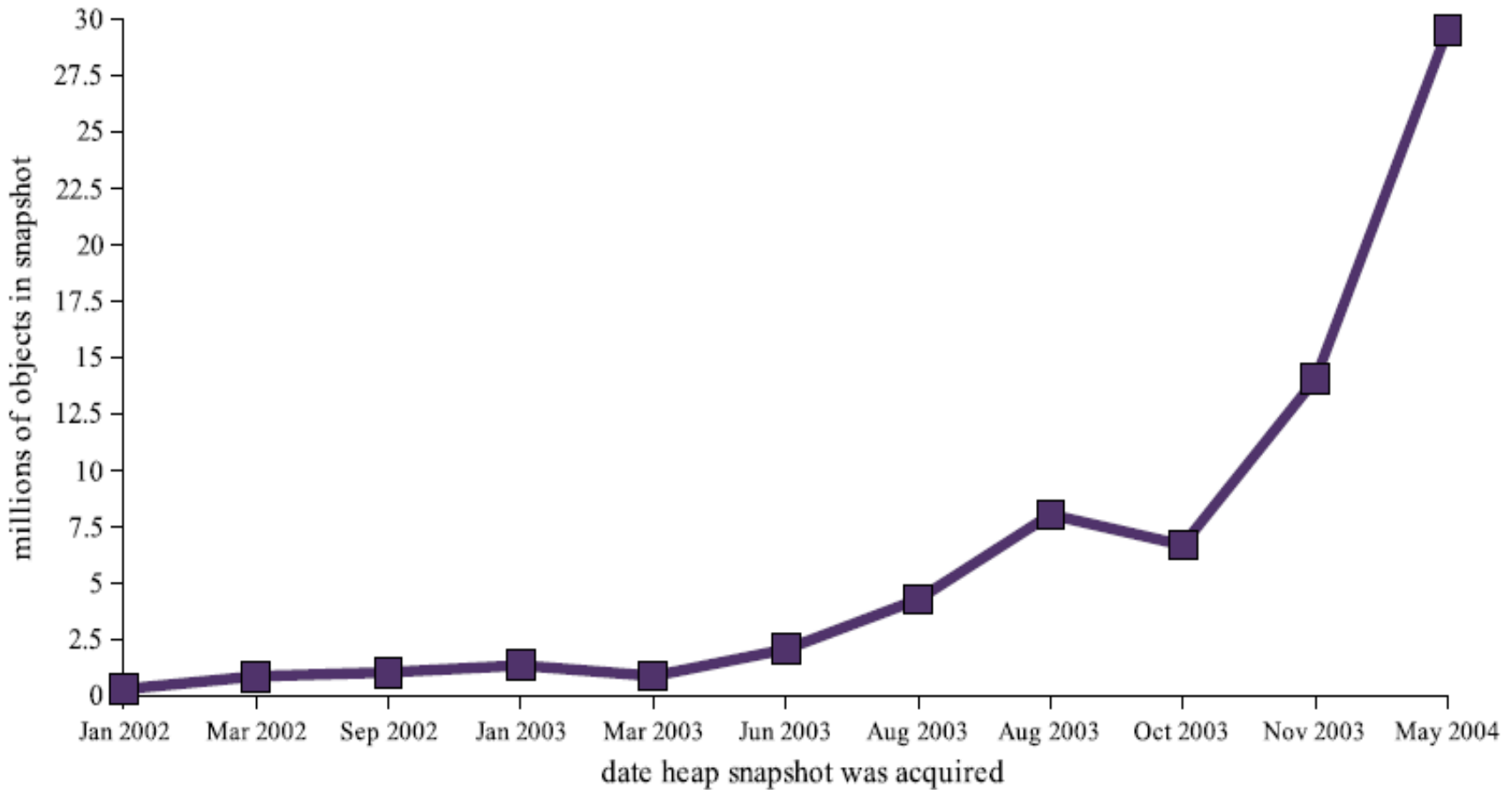
CSE Department
Ohio State University

Ph.D. Thesis Defense
Aug 3, 2011



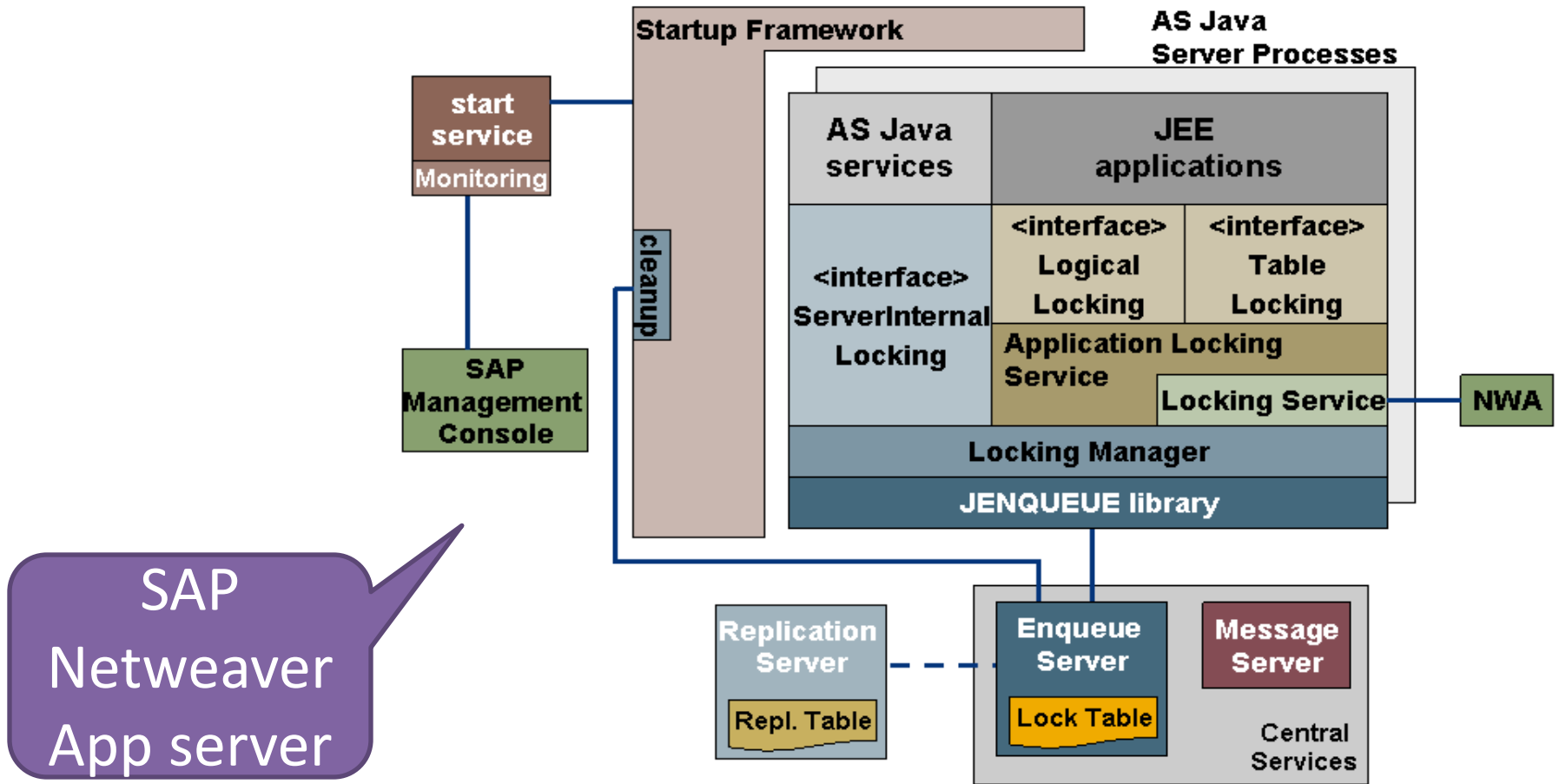
OHIO STATE
UNIVERSITY
PRESTO
RESEARCH
GROUP

Rapid Heap Growth Between 2002 and 2004



Mitchell et al. LCSD'05

The Big Pile-up



* Millions lines of code

* Framework-intensive

* Easy to have performance problems

* Hard to diagnose

The Big Pile-up Causes Runtime Bloat

Heaps are getting bigger

- Grown from **500M** to **2-3G** or more in the past few years
- But not necessarily supporting more users or functions

Surprisingly common (all are from real apps):

- Supporting **thousands** of users (**millions** are expected)
- Saving **500K** session state per user (**2K** is expected)
- Requiring **2M** for a text index per simple document
- Creating **100K** temporary objects per web hit

Consequences for scalability, power usage, and performance

Problems and Insights

- **Problem** with compiler optimizations
 - limited scope
 - lack of developer insight
- **Problem** with human optimizations
 - lack of good tools
- **Our insight:** There is a way to identify larger optimization opportunities with a small amount of developer time
- **Our methodology:** Advocate **compiler-assisted manual tuning**
 - Static and dynamic analysis techniques to **find** and **remove** bloat



An Overview of Bloat Analysis

- Results

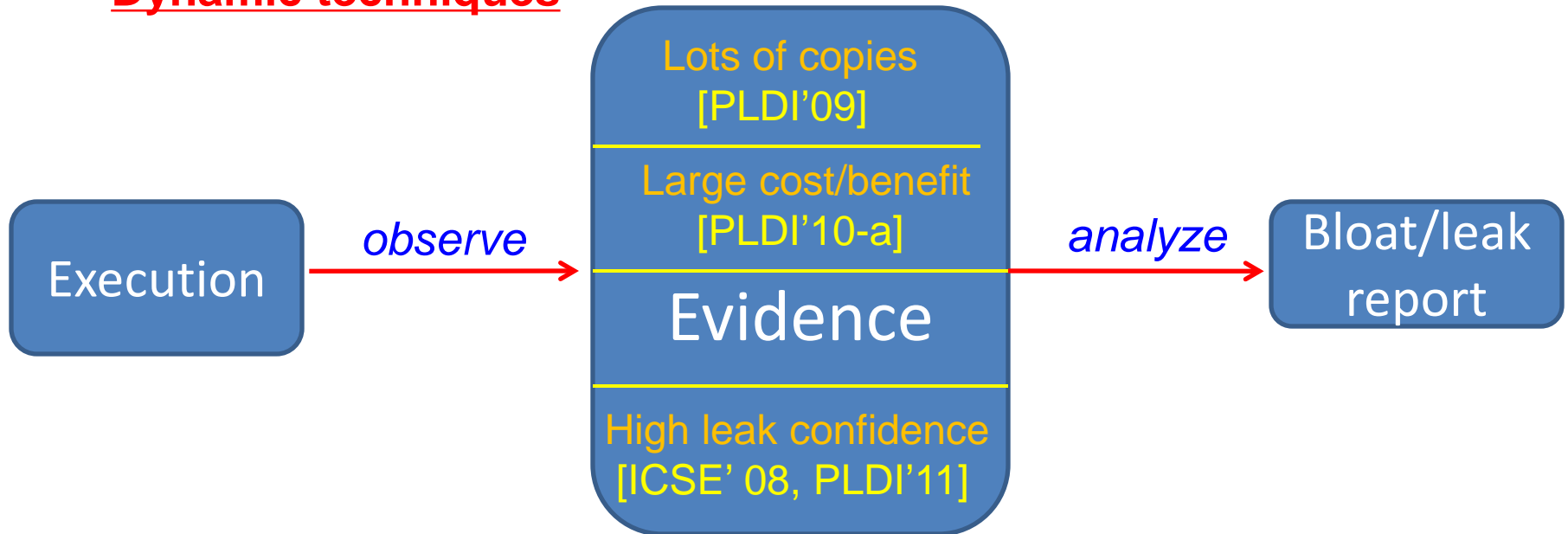
- **Dynamic:** [ICSE'08, PLDI'09, PLDI'10-a, PLDI'11]
- **Static:** [PLDI'10-b, Ongoing]
- **Roadmap:** [FoSER'10]

- Platforms

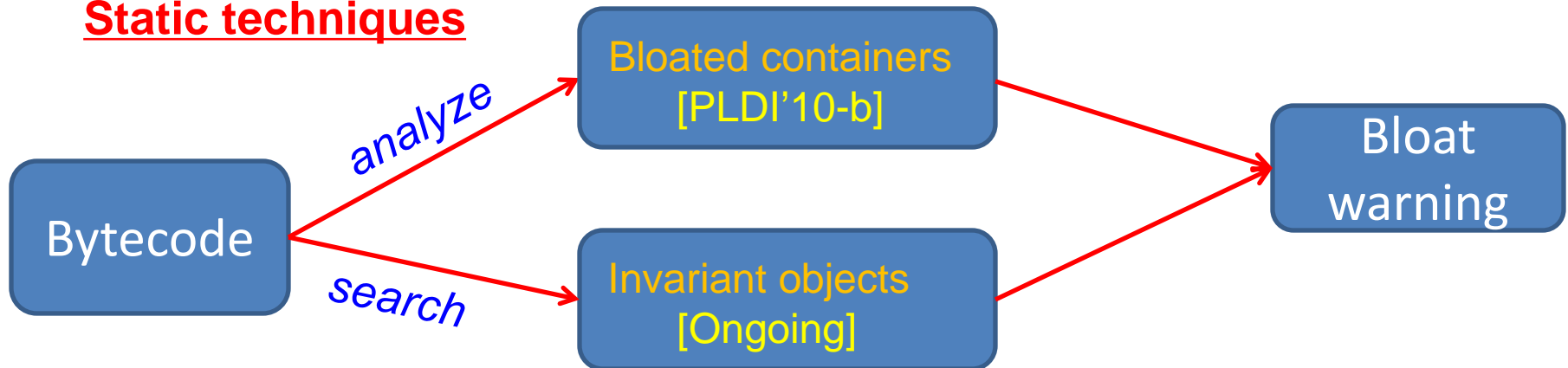
- Java Virtual Machine: IBM J9 [PLDI'09, PLDI'10-a]
JikesRVM [PLDI'11]
- Java Virtual Machine Tool Interface [ICSE'08]
- Soot [PLDI'10-b, Ongoing]

Overview of Techniques

Dynamic techniques



Static techniques



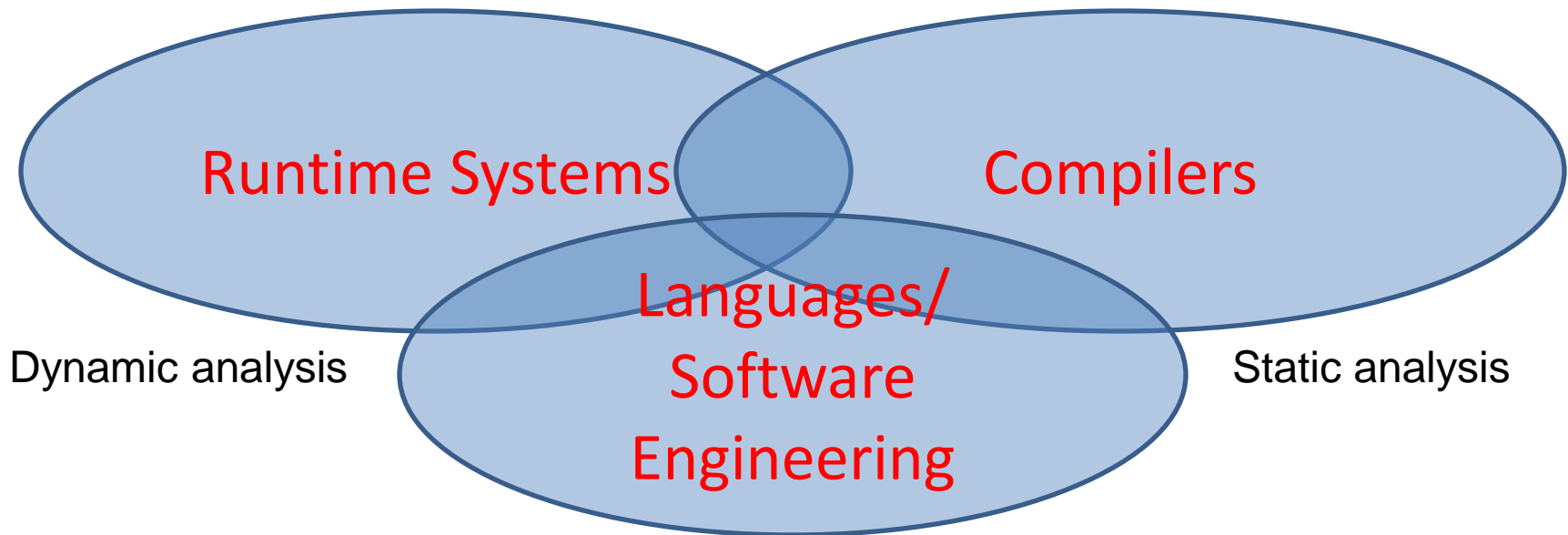
An Overview of My Work

- Software bloat analysis– dissertation work
 - [ICSE'08, PLDI'09, FoSER'10, PLDI'10-a, PLDI'10-b, PLDI'11, Ongoing]
- Scalable points-to and data-flow analysis
 - [CC'08, ISSTA'08, ECOOP'09, ISSTA'11]
- Control- and data-flow analysis for AspectJ
 - [ICSE'07, AOSD'08, ASE'09, TSE'11]
- Language-level checkpointing and replay
 - [FSE'07]

Research Areas

PLDI'09, PLDI'10-a,
PLDI'11

ISSTA'08, CC'08, ECOOP'09,
PLDI'10-b, ISSTA'11, Ongoing



ICSE'07, FSE'07, ICSE'08, AOSD'08,
ASE'09, FoSER'10, TSE'11

Outline

- Introduction
 - Introduction to bloat
 - Overview of my thesis
- I: Finding low-utility data structures [PLDI'10-a]
 - Dynamic analysis to help performance tuning
- II: Finding loop-invariant data structures [Ongoing]
 - Static analysis that targets a specific bloat pattern found by I
- Future work and conclusions

Finding Low-Utility Data Structures

- Identify high-cost-low-benefit data structures that are likely to be performance bottlenecks
- To achieve this, we need to
 - Compute cost and benefit measurements for objects
 - Present to users a list of data structures ranked by their cost/benefit rates

A Systematic Way of Detecting Bloat

Bloat can manifest itself through many *symptoms*

temporary objects [Dufour FSE'08, Shankar OOPSLA'08]

excessive data copies [Xu PLDI'09]

highly-stale objects [Bond ASPLOS'06, Novak PLDI'09]

problematic container behaviors [Xu ICSE'08, Shaham PLDI'09,
Xu PLDI'10]

...

Is there a common way to characterize bloat?

At the heart of all inefficiencies lie computations, with great expense, produce data values that have little impact on the forward progress

What is Common About Bloat

e.g. "java/io"

```
boolean isPackage(String s){
```

```
    List packs = directoryList(s);
```

```
    return packs != null;
```

```
}
```

High cost-benefit rate

```
List directoryList(String pkgName){
```

```
    //find and return all files in directory pkgName
```

```
    // return null if nothing is found
```

```
}
```

--- from Eclipse

8% running time reduction achieved by specializing the method

Cost and Benefit

- Absolute cost of a heap value: **total number of instructions executed to produce it**
 - $a = b$ or $a = b + c$, each has **unit cost**
- Benefit of heap value v :

$v \longrightarrow \bigcirc$ (*output*) : benefit is a very large number

$v \longrightarrow \text{DEAD END}$ (*not used*) : benefit is 0

$v \xrightarrow{M} v'$: benefit is **M** (amount of work)

Cost Computation

b = 10;

a = b + 3;

c = b * 5;

d = a / c;

The cost of d is **4**

An example of a *backward dynamic flow* problem
Requires dynamic slicing: record all mem. accesses

Abstract Dynamic Slicing

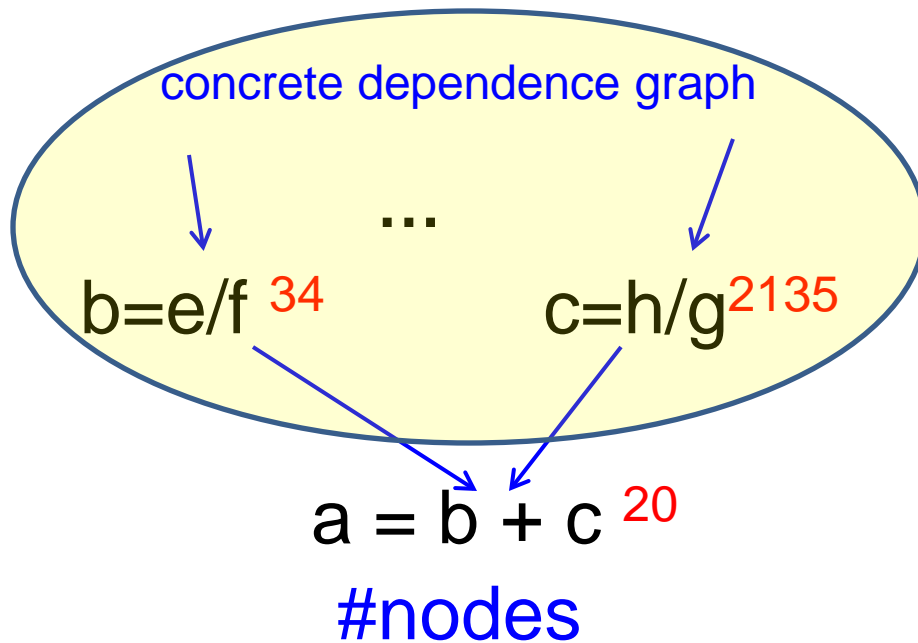
- Trace generated by dynamic slicing is *unbounded*
 - The amount of memory needed depends on the dynamic behavior of a program
- Trace contains many more details than a client would need
 - Is it possible to capture only part of the execution relevant to the client
- For many problems, equivalence classes exist

E1 $a = b.m^1$ $a = b.m^3$ $a = b.m^{67}$ $a = b.m^{23}$ $a = b.m^{1235} \dots$

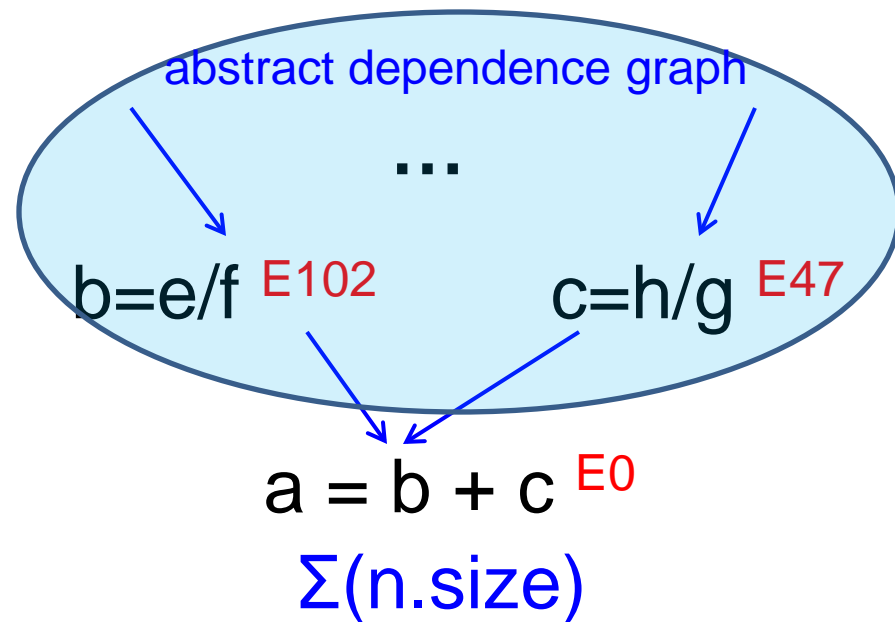
E2 $a = b.m^{45}$ $a = b.m^{217}$ $a = b.m^{29}$ $a = b.m^{35}$ $a = b.m^9 \dots$

Cost Computation

Absolute cost

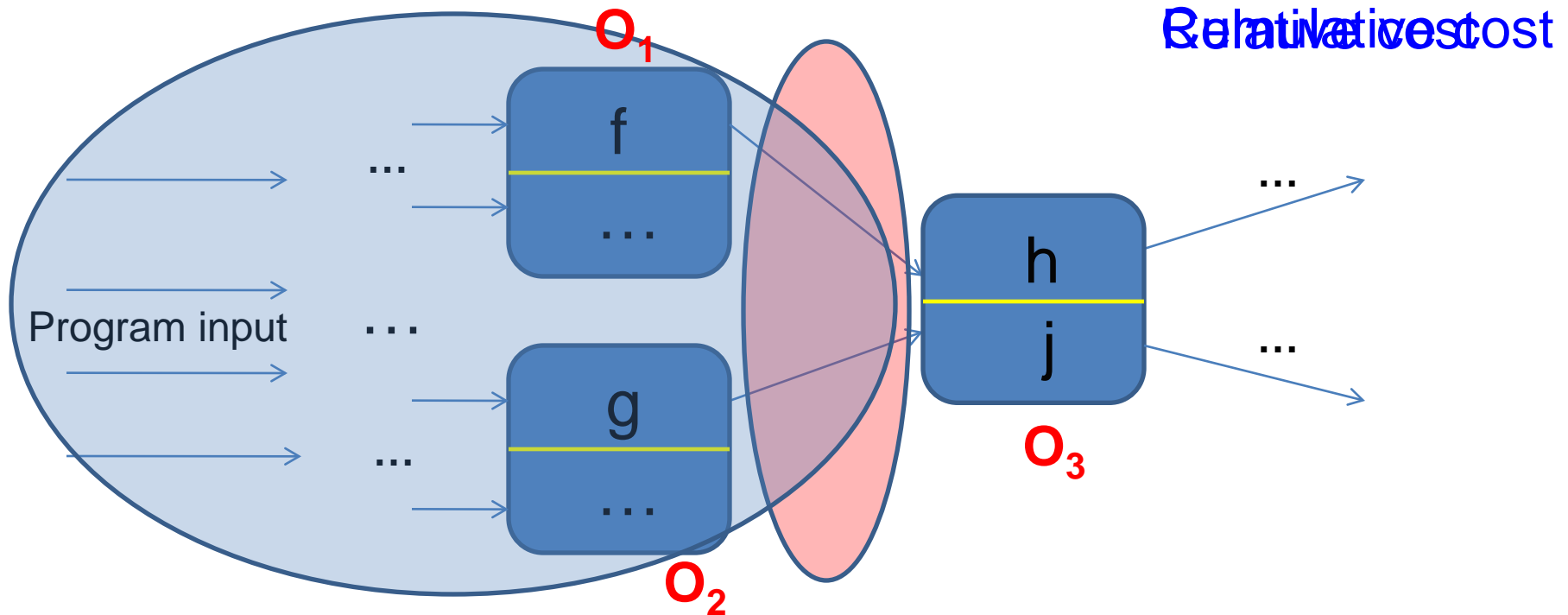


Abstract cost



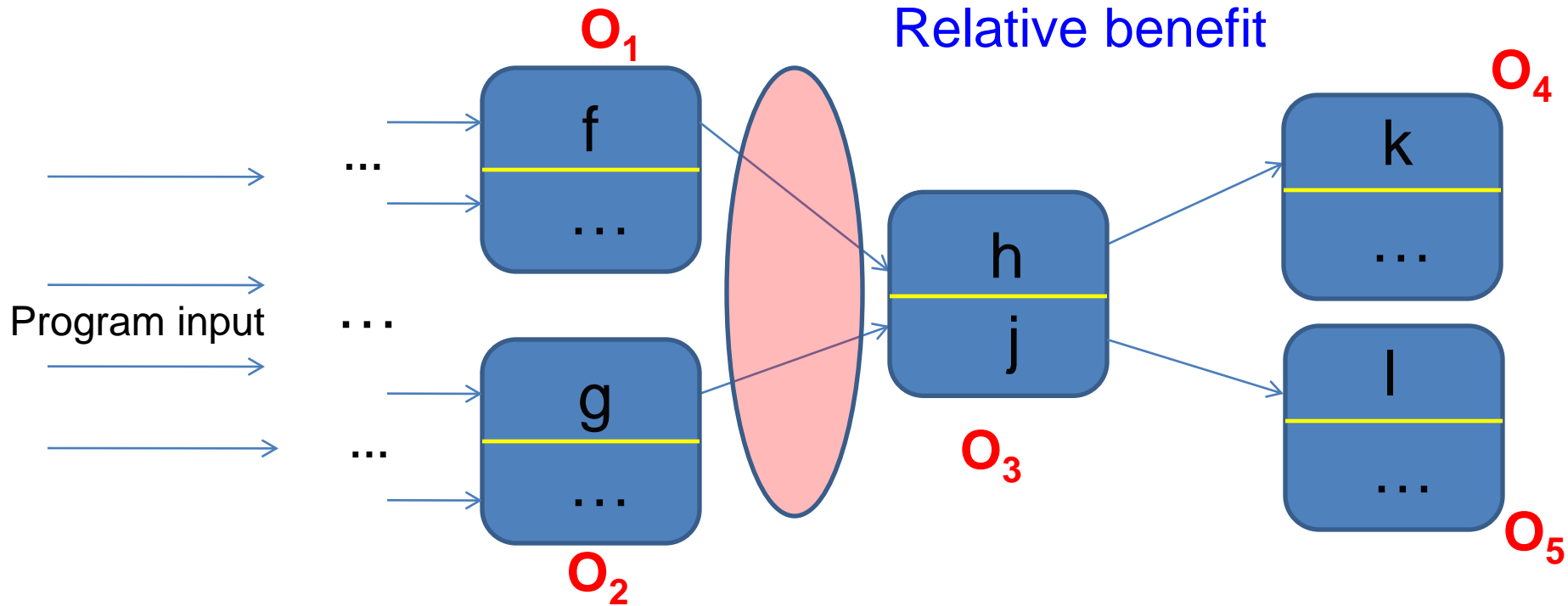
- Use calling contexts to define equivalence classes
- “ $E_{102}, E_{47}, E_0 \dots$ ” are object-sensitivity-based calling contexts (i.e., chains of receiver objects)

Relative Abstract Cost



- **Cumulative cost** measures the effort made *from the beginning of the execution* to produce a value
- It is certain that *the later* a value is produced, *the higher cost* it has

Relative Abstract Benefit



- **Relative benefit:** the effort made to *transform* a value read from a heap loc *l* to the value written into another heap loc *l'*

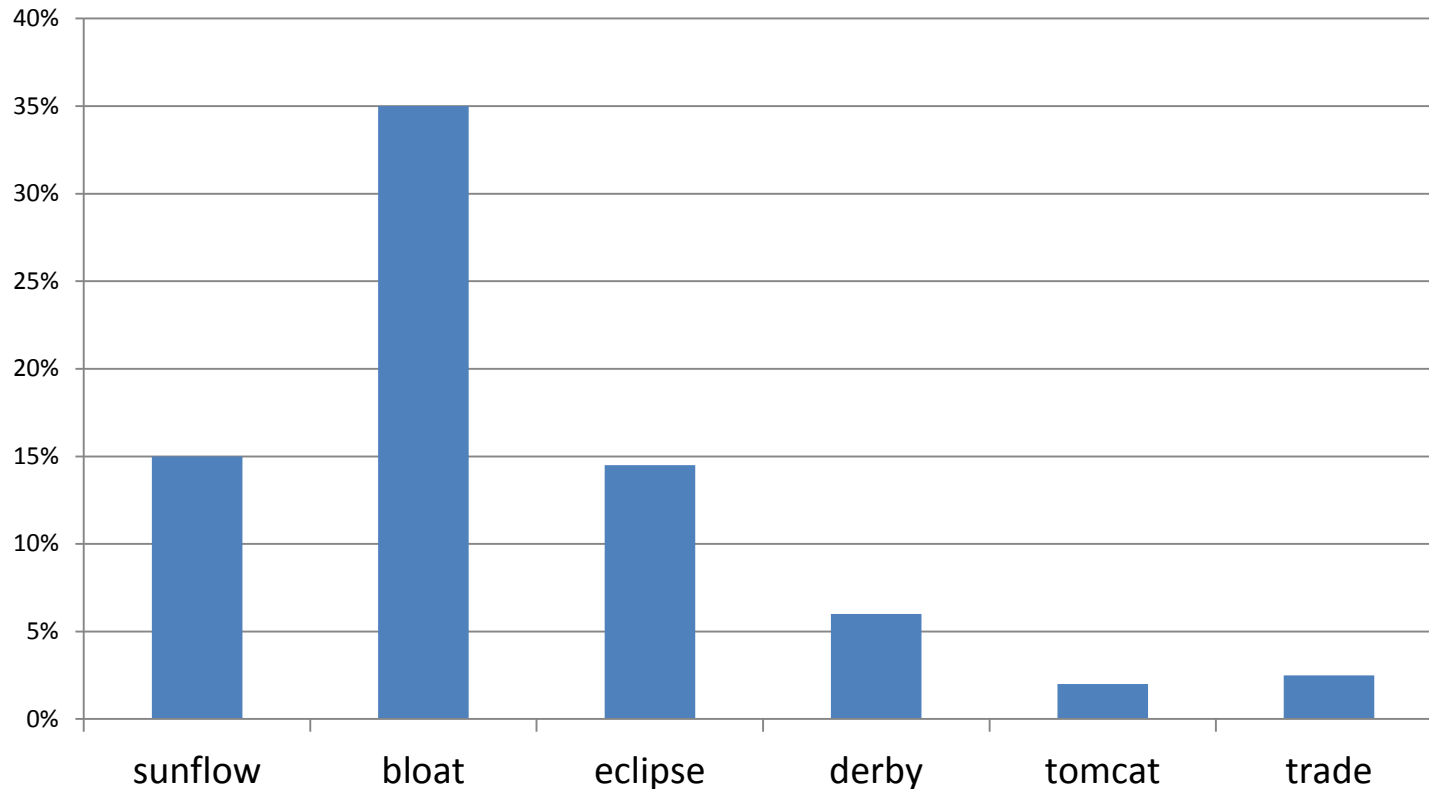
Key Problems and Ideas

- Problem 1: Dynamic slicing is too expensive and unnecessary
 - **Insight 1:** *Abstract slicing*
- Problem 2: How to abstract runtime instances for computing costs for object-oriented data structures
 - **Insight 2:** *(Object-sensitivity-based) calling contexts*
- Problem 3: Cumulative cost is not correlated with performance problems
 - **Insight 3:** *Relative cost*
- Solution: Compute *relative abstract cost*

Performance Improvements

- Implemented in IBM J9 Java Virtual Machine
- Case studies on real-world large applications

Running time reductions



Impact

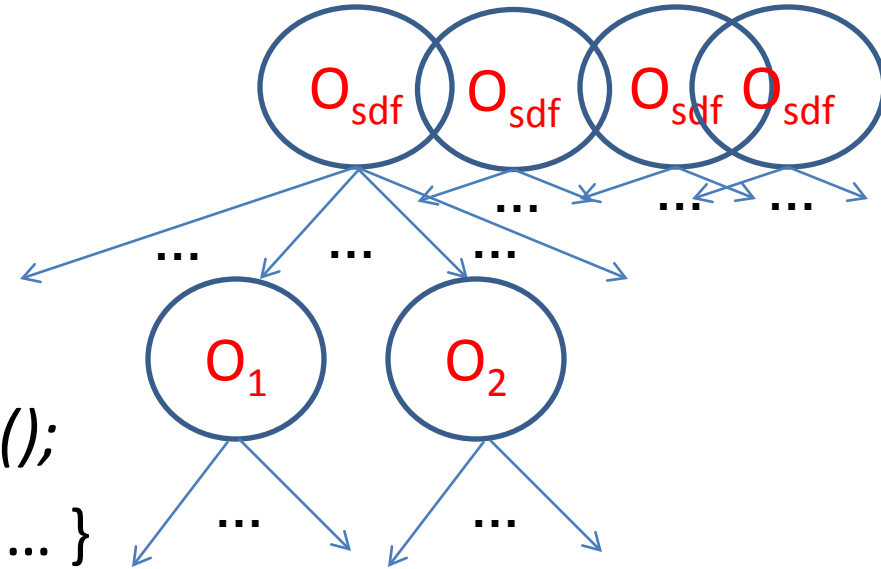
- The first dynamic analysis targeting general bloat
- Identified patterns leading to new techniques
 - Container inefficiencies
[PLDI'10-b,
context-free-language reachability formulation]
 - *Loop-invariant data structures*
[Ongoing, type and effect system]
 - Problematic implementations of certain design patterns
 - Anonymous classes

Outline

- Introduction
 - Introduction to bloat
 - Overview of my thesis
- I: Finding low-utility data structures [PLDI'10-a]
 - Dynamic analysis to help performance tuning
- **II: Finding loop-invariant data structures [Ongoing]**
 - Creating objects with **the same content** many times in loops really hurts performance
 - Pulling them out of loops (i.e., hoisting) would potentially save a lot of computation

Example

```
for(int i = 0; i < N; i++){  
    SimpleDateFormat sdf =  
        new SimpleDateFormat();  
    try{ Date d = sdf.parse(date[i]); ... }  
    catch(...) {...}  
}
```



--- From applications studied in IBM T. J. Watson Research Center

- Hoistable logical data structure
- Computing hoistability measurements

Hoistable Data Structures

- Understand how a data structure is built up
 - *points-to* relationships
- Understand where the data comes from
 - *dependence* relationships
- Understand in which iterations objects are created
 - Compute *iteration count abstraction* (ICA) for each allocation site o : 0, 1, \perp
 - 0: created *outside the loop* (i.e., exists before the loop starts)
 - 1: *inside the loop* and does not escape the current iter
 - \perp : *inside the loop* and escapes to later iterations

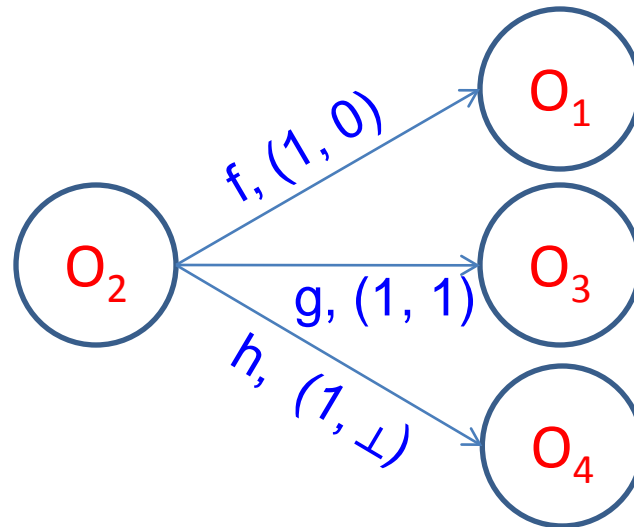
ICA Annotation

- Annotate points-to and dependence relationships with ICAs

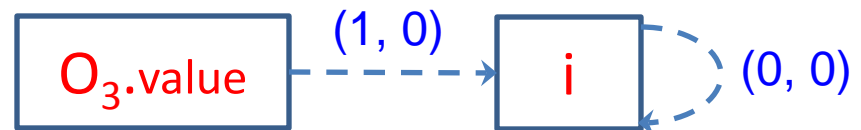
```

A a = new A();           // O1, 0
String s = null;
for (int i = 0; i < 100, i++){
    Triple t = new Triple(); // O2, 1
    Integer p = new Integer(i); // O3, 1
    if(i < 10)
        s = new String("s"); // O4, ⊥

    t.f = a; // connect O2 and O1
    t.g = p; // connect O2 and O3
    t.h = s; // connect O2 and O4
}
    
```



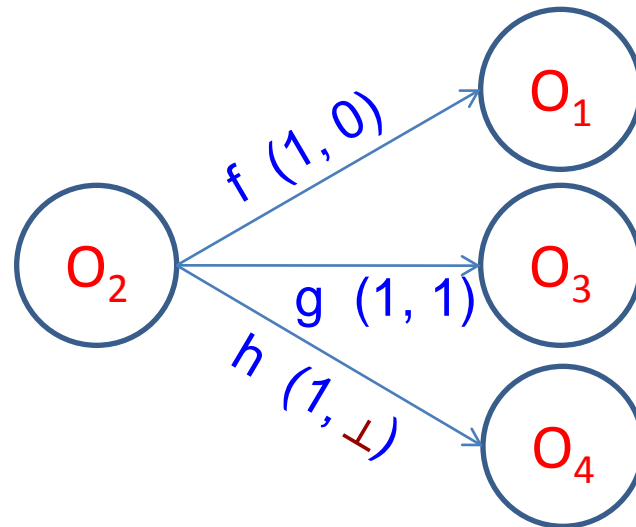
Annotated points-to relationships



Annotated dependence relationships

Detecting Hoistable Data Structures

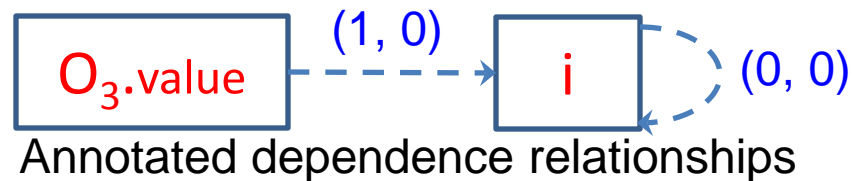
- Disjoint
 - Check: s does not contain objects with \perp ICAs



Annotated points-to relationships

Detecting Hoistable Data Structures (Cond)

- Loop-invariant fields
 - **Check:** No dependence chain starting from a heap location in s is annotated with \perp
 - **Check:** Nodes whose ICA are 0 are not involved in any cycle



Hoistability Measurements

- Instead of doing transformation, we compute hoistability measurements
- Structure-based hoistability (SH)
 - How many objects in the data structure are disjoint
- Data-based hoistability (DH)
 - How many fields in the data structure are loop-invariant

Evaluation

- The technique was implemented using Soot 2.3 and evaluated on a set of 19 large Java programs
- A total 981 loop data structures considered
 - 155 are disjoint data structures(15.8%)

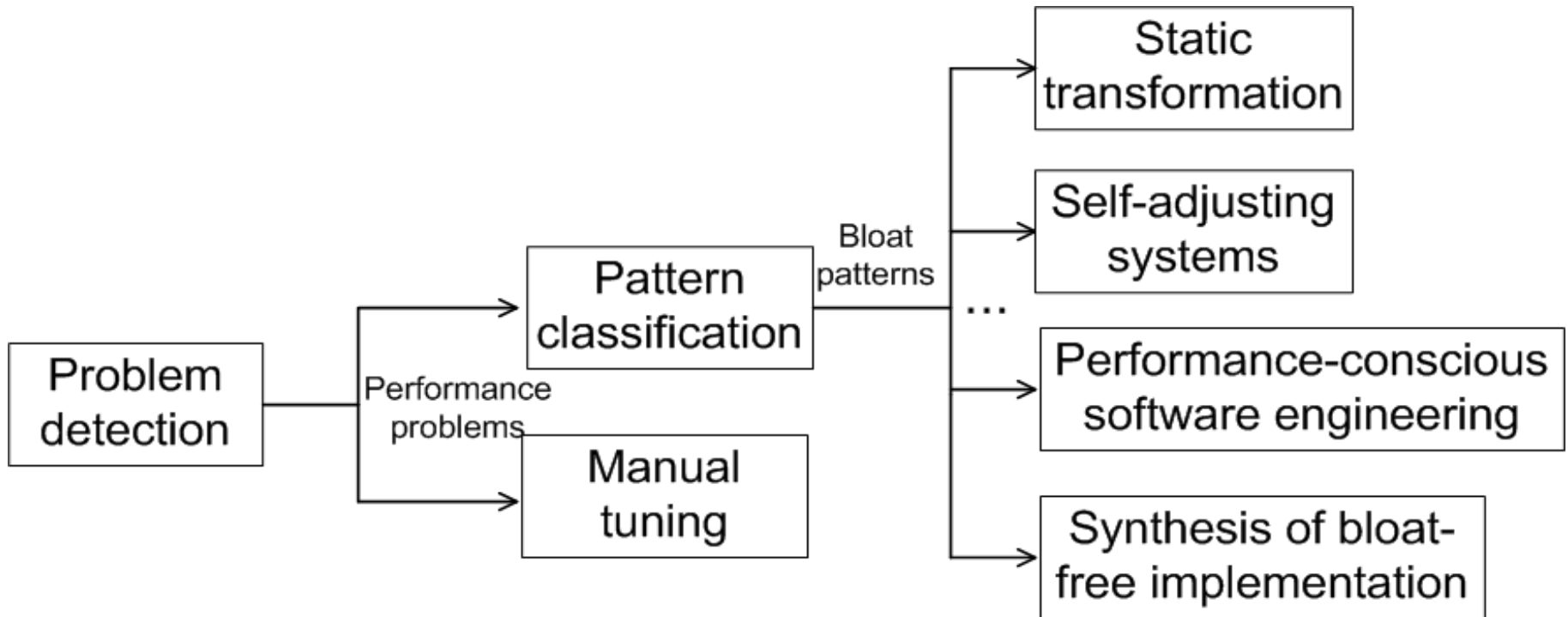
Case Studies

- We studied five large applications by inspecting the top 10 data structures ranked based on hoistability measurements
- Large performance improvements achieved by manual hoisting
 - ps: 82.1%
 - xalan: 10% (confirmed by the DaCapo team)
 - bloat: 11.1%
 - soot-c: 2.5%
 - sablecc-j: 6.7%
- No problems that we found have been reported before

Outline

- Introduction
- I: Finding low-utility data structures [PLDI'10-a]
- II: Finding and hoisting loop-invariant data structures [Ongoing]
- **Future Work and Conclusions**

Future Work--Bloat Analysis



- Look a bit deeper into bloat causes, we will see
 - Object-orientation encourages excess
- Leverage techniques from other fields such as systems and architecture

Conclusions

- Static and dynamic analysis
 - Principles and applications
- My dissertation
 - Software bloat analysis
 - Dynamic analyses that can make more sense of heaps and executions
 - Static analyses that can remove and prevent bloat
- Hopes
 - **Tool support**: Tuning is no longer a daunting task
 - *Education*: Performance is important

Thank You

Q & A