

Fast and Effective Optimization of Statically Typed Object-Oriented Programs

David F. Bacon

University of California, Berkeley
and
IBM T.J. Watson Research Center

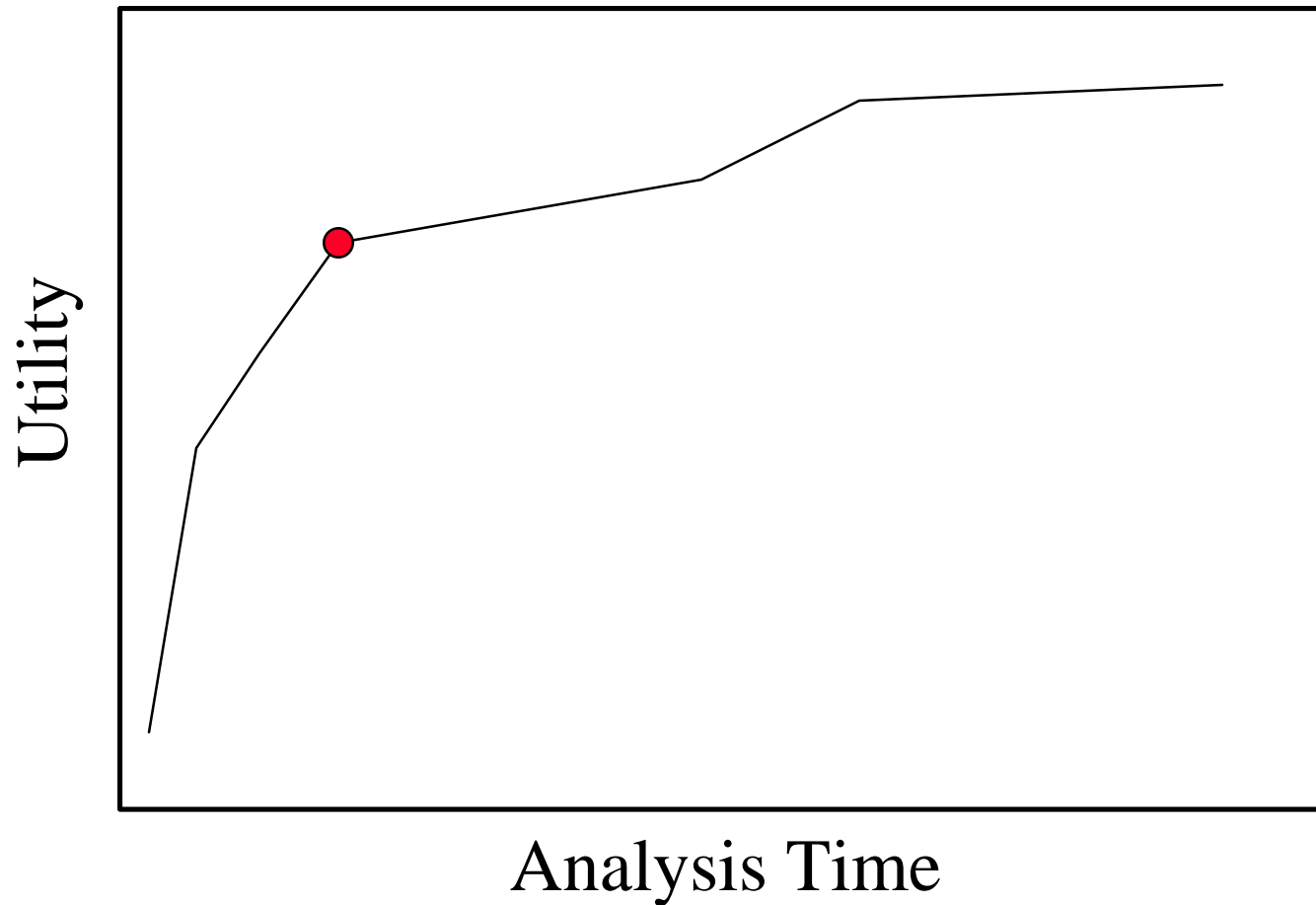
Introduction

- ◆ OO languages have powerful features
 - Virtual Function Calls
 - Virtual Base Classes
 - Dynamic Casts
- ◆ But they cost
 - Time (virtual calls up to 35% for C++)
 - Space (virtual inheritance up to 50% for C++)

My Thesis

- ◆ For statically typed OO languages
 - C++
 - Java
- ◆ powerful and expensive features can be optimized by an algorithm that is
 - relatively simple
 - very fast

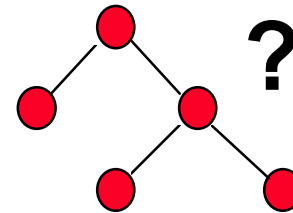
Utility/Time Hypothesis



Outline

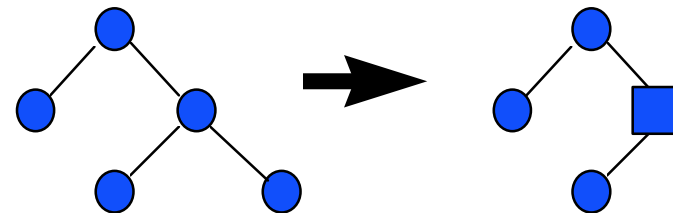
- ◆ **Analysis**

- present 2 fast algorithms



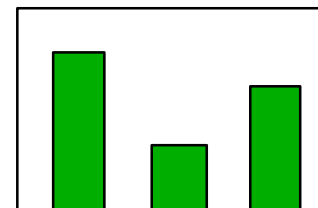
- ◆ **Transformation**

- applications of analysis

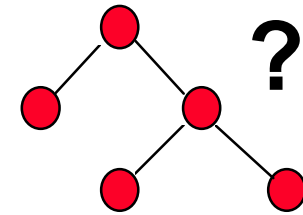


- ◆ **Evaluation**

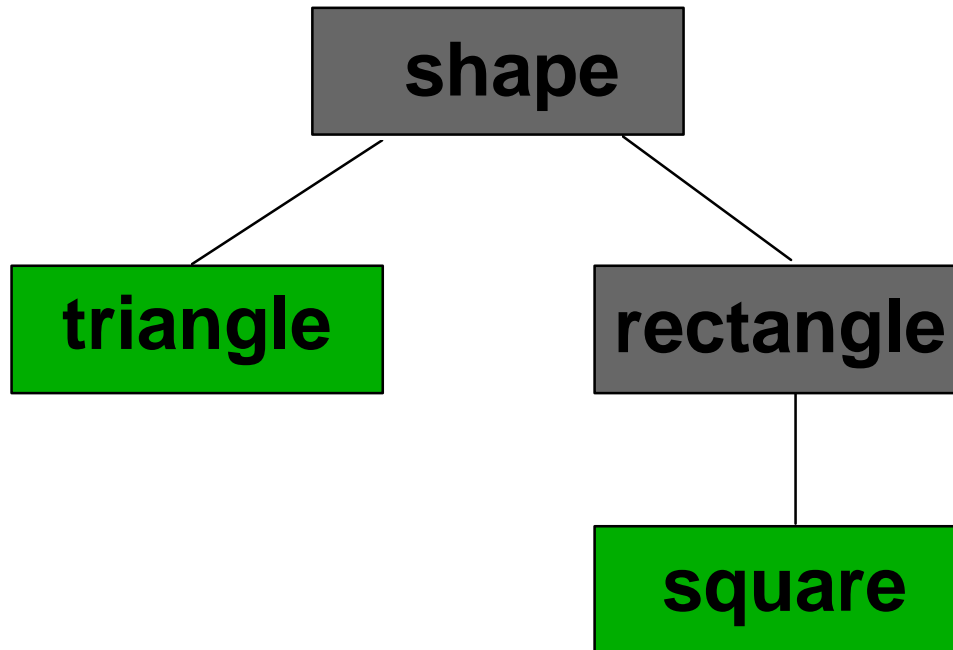
- compare 3 algorithms



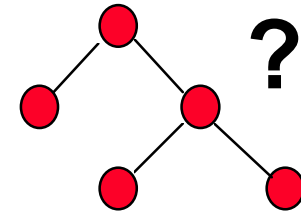
Analysis Framework



- ◆ What classes are created?



Analysis Framework

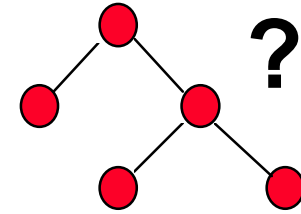


- ◆ What are the classes of each value?

`object->draw() ;` { **triangle** }

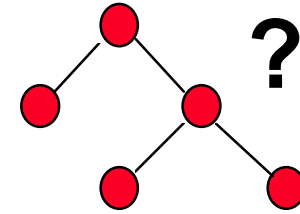
`sign.reshape() ;` { **rectangle, square** }

Class Hierarchy Analysis

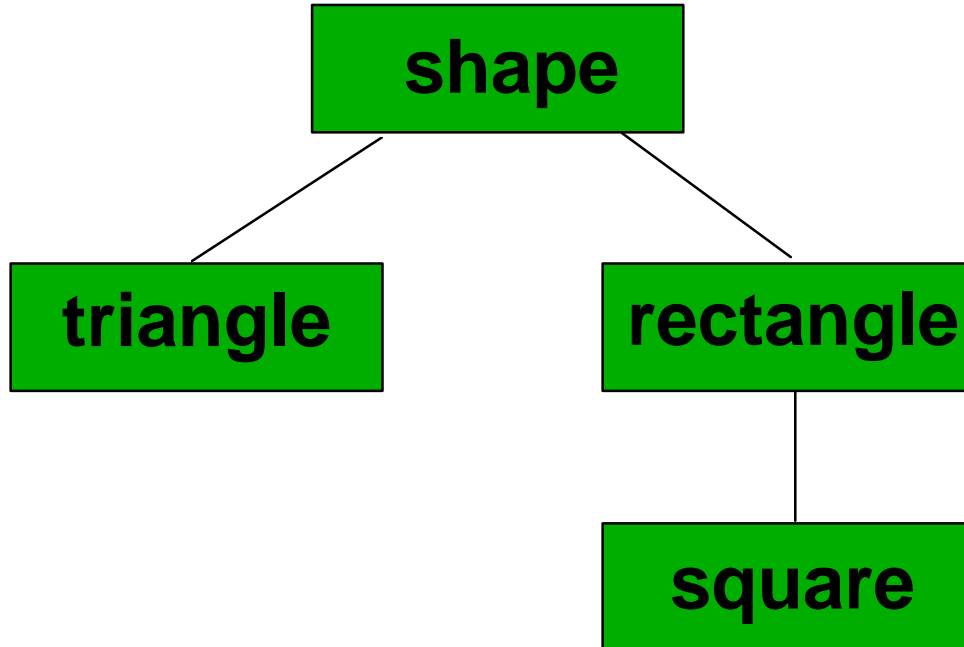


- ◆ What classes are created?
 - all classes
- ◆ What are the classes of each value?
 - the classes derived from the declared type

Class Hierarchy Analysis



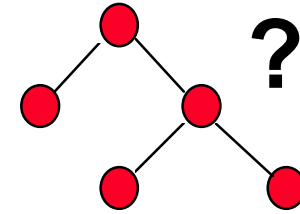
- ◆ the classes derived from the declared type



```
rectangle* r;  
{rectangle,square}
```

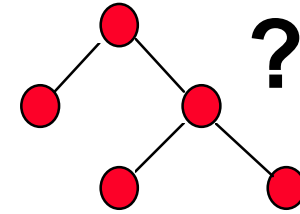
```
square * s;  
{square}
```

Rapid Type Analysis

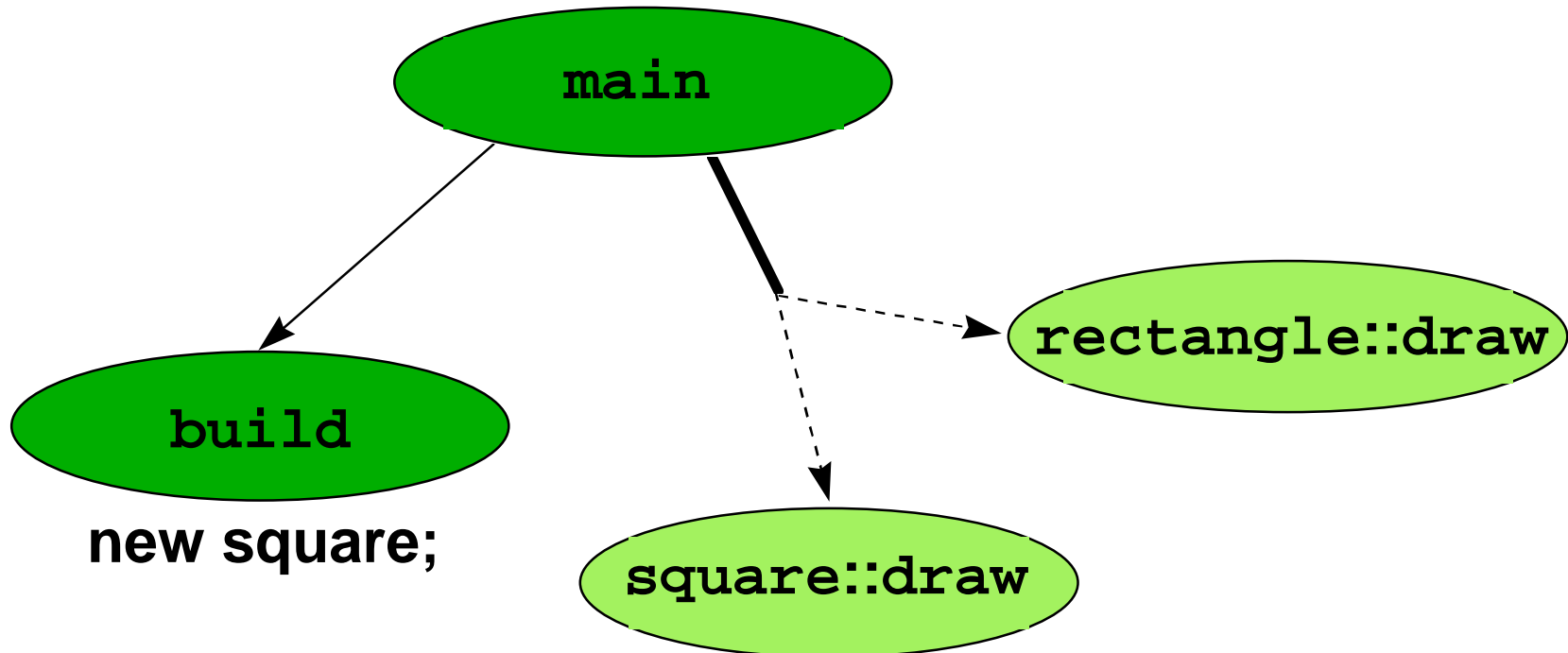


- ◆ What classes are created?
 - the classes created in the reachable call graph
- ◆ What are the classes of each value?
 - the classes derived from the declared type
 - that are created

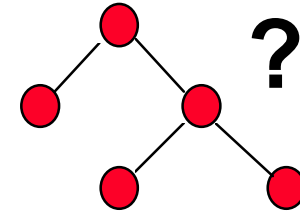
Rapid Type Analysis



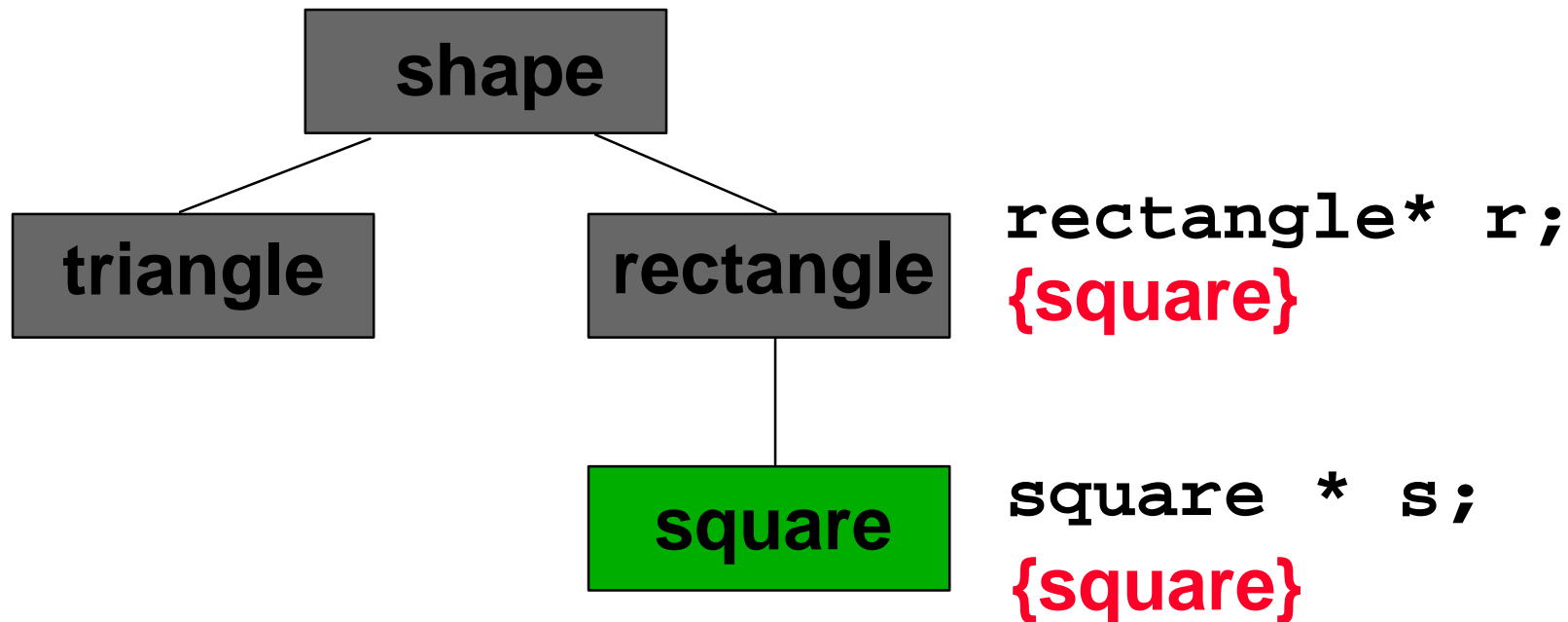
- ◆ the classes in the call graph



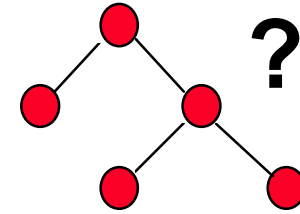
Rapid Type Analysis



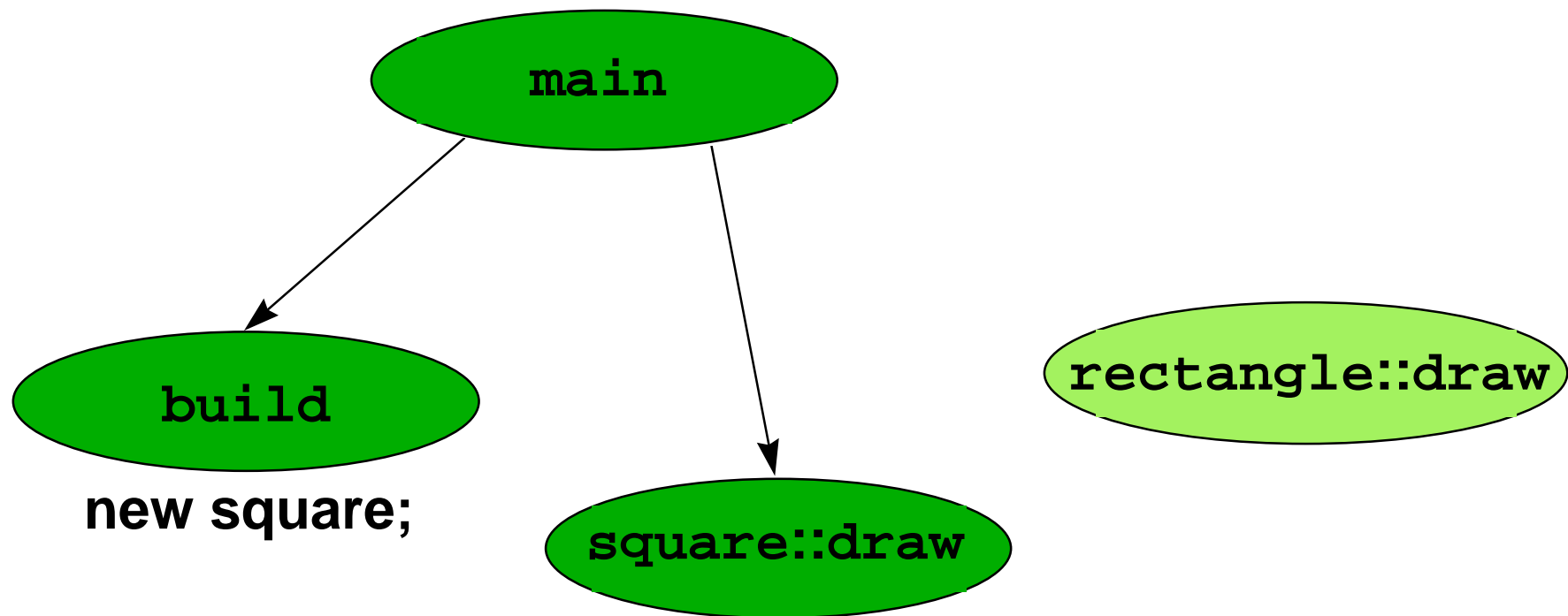
- ◆ What are the classes of each value?
 - the classes derived from the declared type
 - that have been created



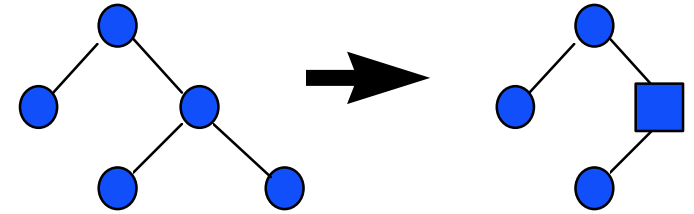
Rapid Type Analysis



- ◆ the classes in the call graph: **{square}**

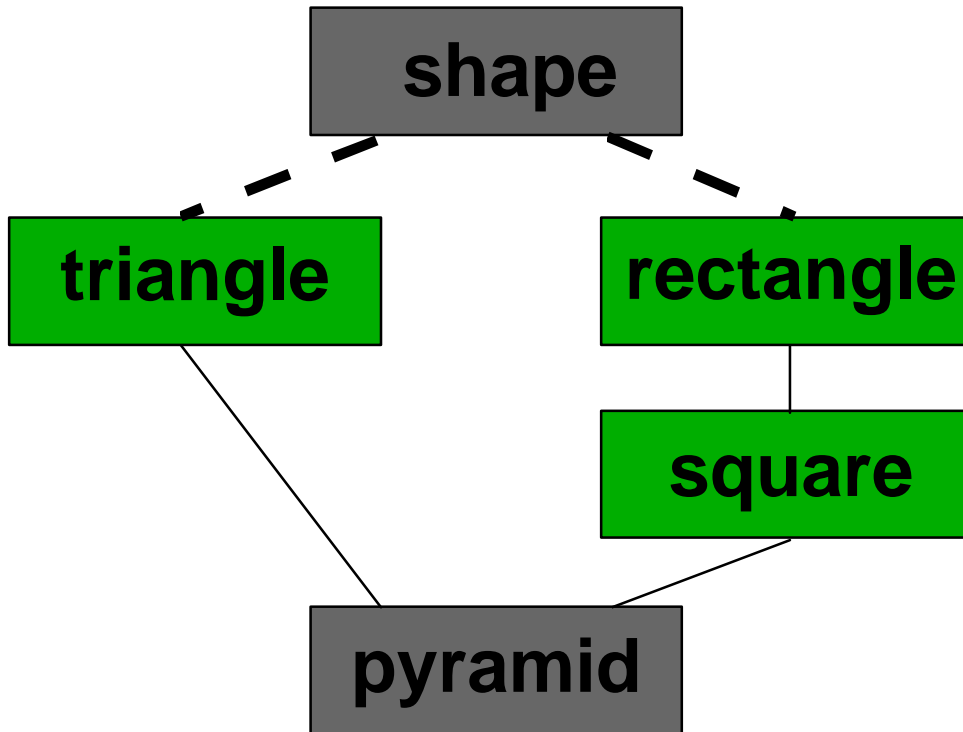
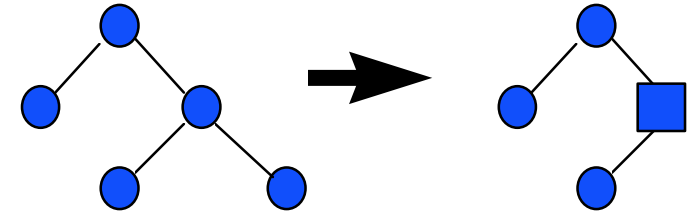


Transformation



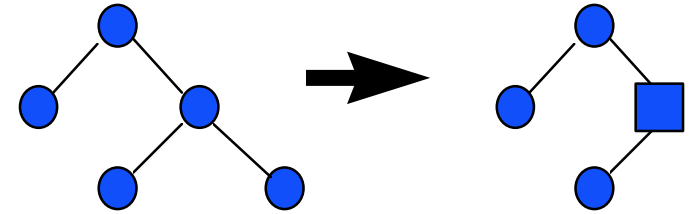
- ◆ virtual base classes to non-virtual bases
- ◆ dynamic casts to static casts
- ◆ virtual calls to direct calls
- ◆ removal of dead code

Virtual Bases



- ◆ Is **shape** multiply inherited?
- ◆ Is that class live?
- ◆ If not, make **shape** a non-virtual base

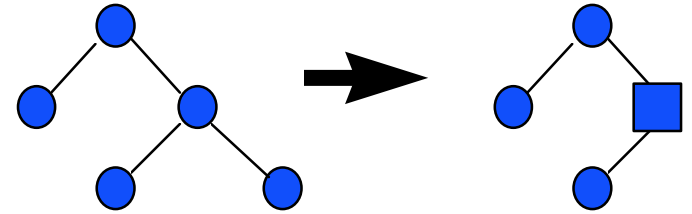
Dynamic Casts



```
foo(shape* p) {  
    square* s = dynamic_cast<square> p;  
    ...  
}
```

- ◆ What classes could **p** point to?
- ◆ Is **square** the only possible class type?
- ◆ If so, do a compile-time cast

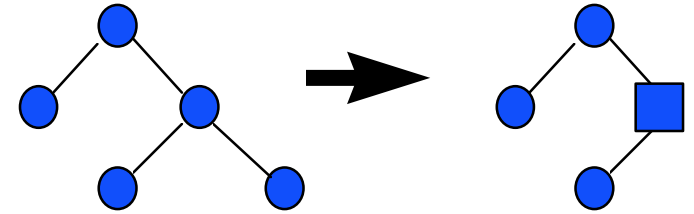
Virtual Calls



p->draw();

- ◆ What classes could **p** point to?
- ◆ What **draw()** functions are defined?
- ◆ If only 1 **draw()** function
 - resolve the call

Benefits of VFR



◆ Time

- direct calls are faster
- in-lining is possible

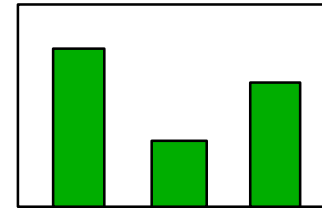
◆ Space

- unused functions can be eliminated

◆ Understandability

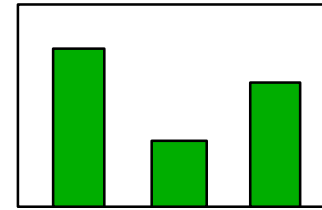
- programmer doesn't see unused code

Evaluation



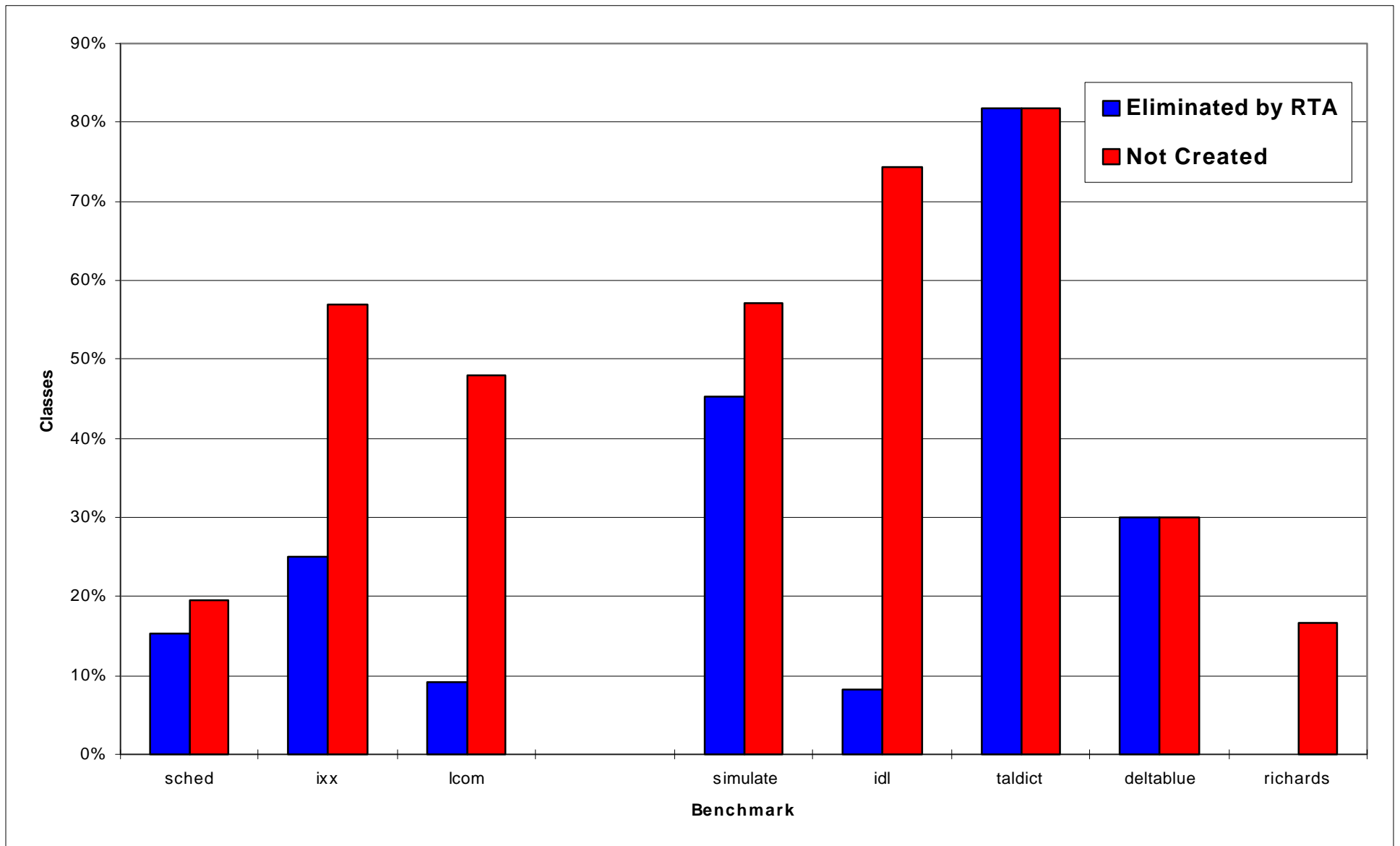
- ◆ Compare 3 fast analysis algorithms
 - Rapid Type Analysis
 - Class Hierarchy Analysis
 - Unique Name [Calder & Grunwald 1994]
- ◆ Virtual function resolution
- ◆ Dead code elimination

Methodology

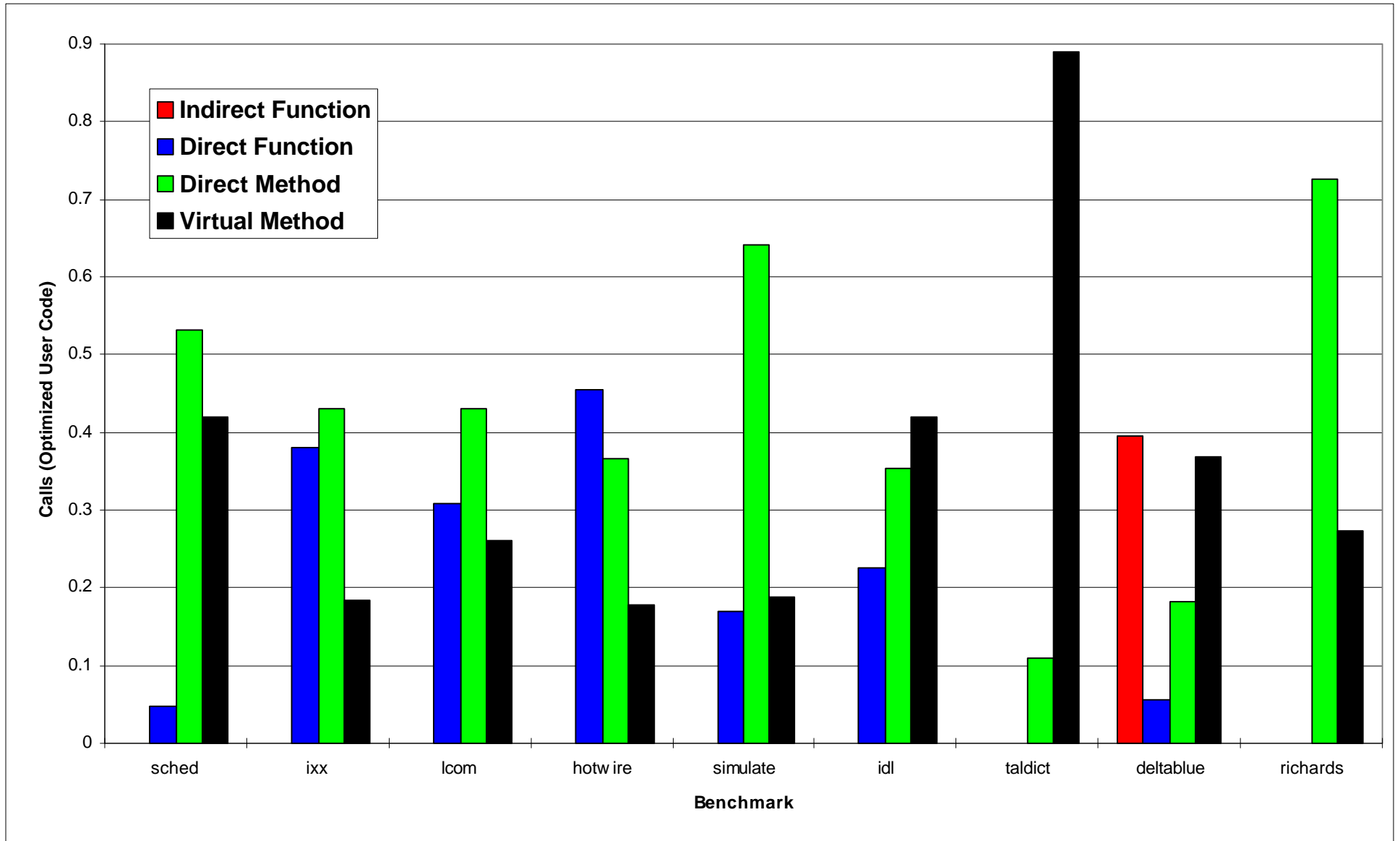


- ◆ Use real programs as benchmarks
 - 7 medium-sized programs (5000-20000 lines)
 - 2 commonly used mini-benchmarks
- ◆ Evaluate against best possible algorithm
 - use dynamic traces for loose upper bounds

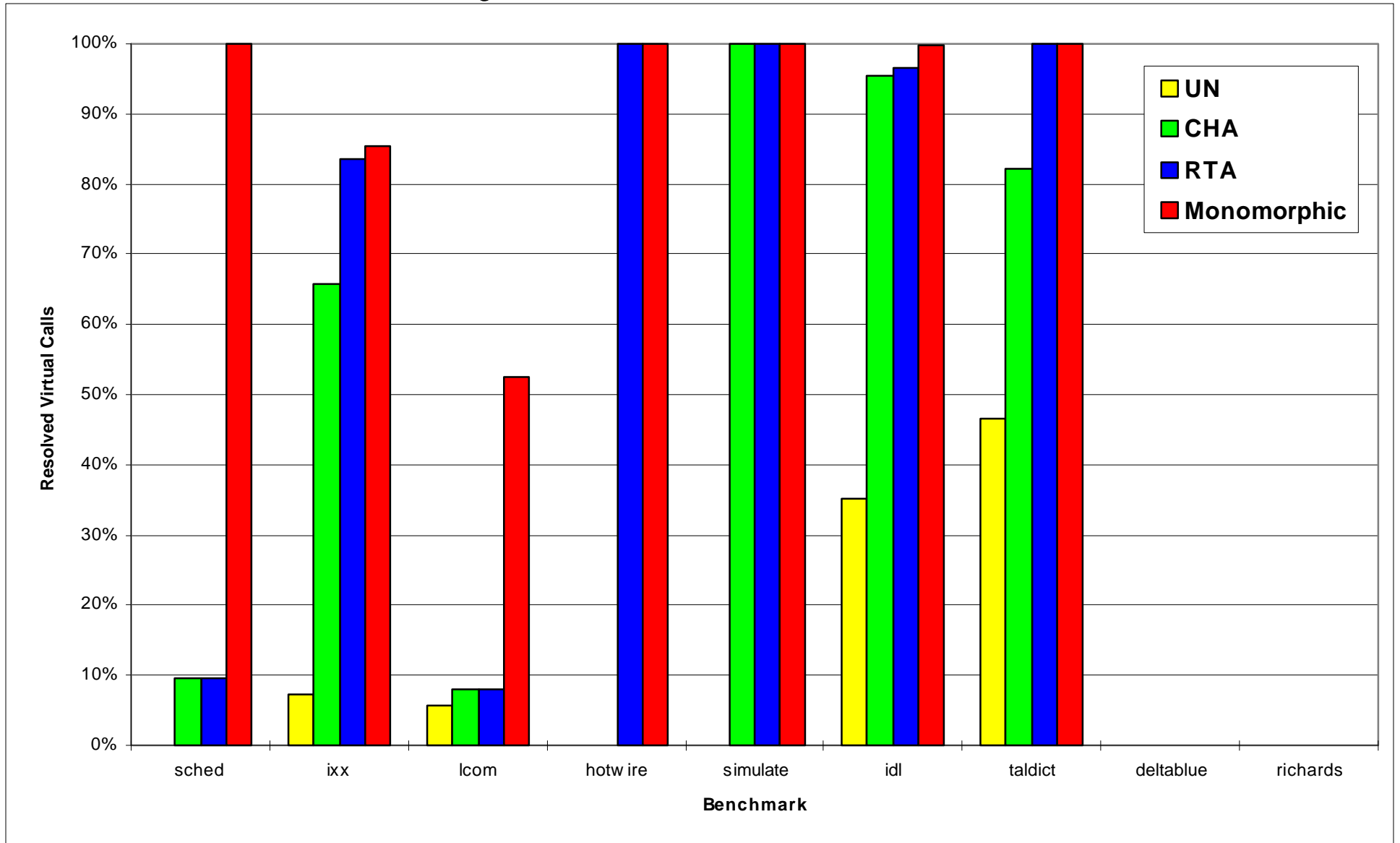
Elimination of Dead Classes



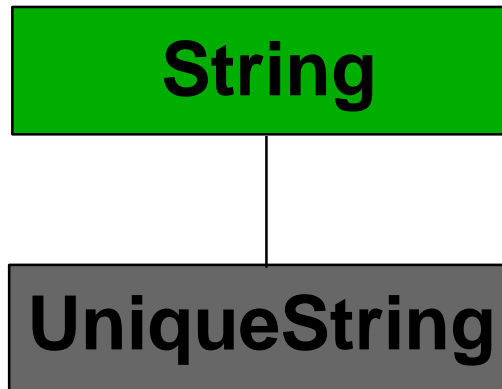
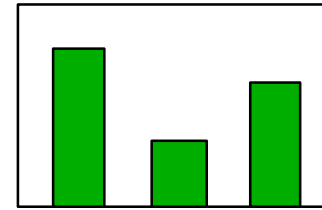
Dynamic Call Type Distribution



Resolved Dynamic Calls

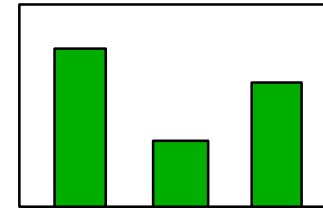


Why RTA Wins: **ixx**

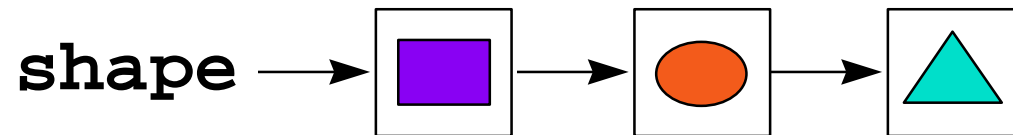


- ◆ RTA finds unused classes
- ◆ String ops are in inner loops
- ◆ Similar win for
 - `taldict`
 - `hotwire`

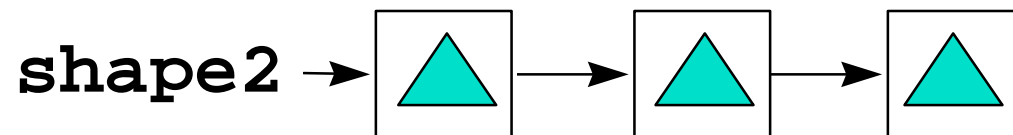
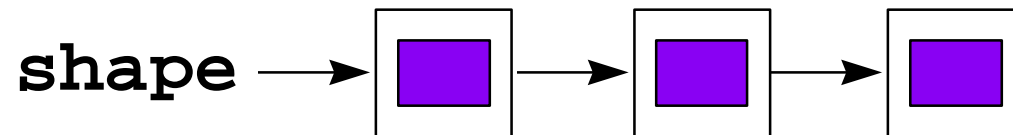
Why RTA Loses: **sched**



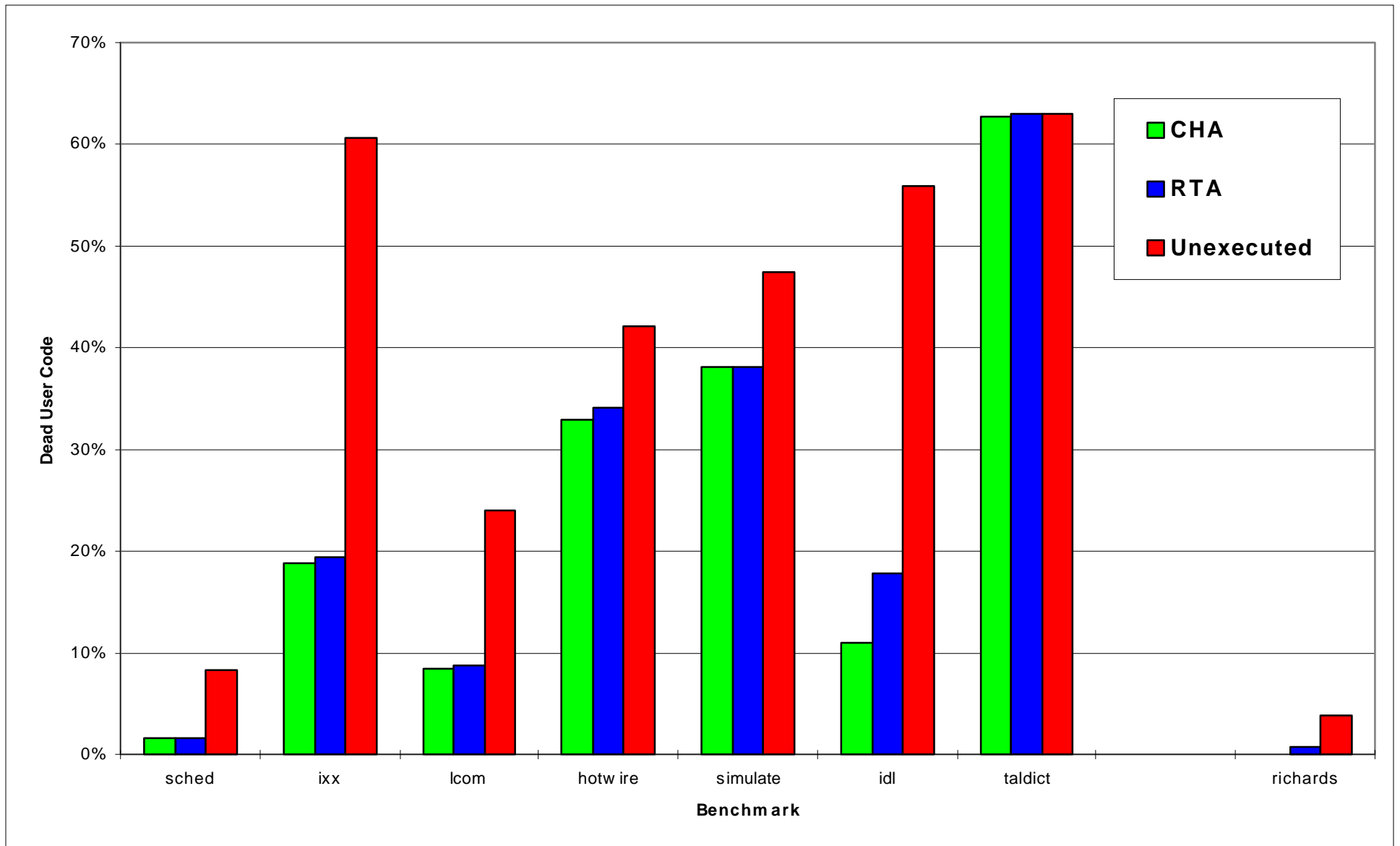
◆ True Polymorphism



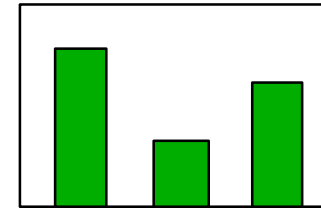
◆ Parametric Polymorphism



Elimination of Dead Code

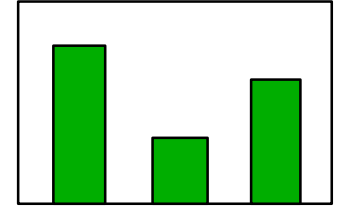


Speed of Analysis



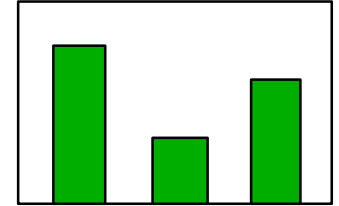
Benchmark	Size (lines)	Analysis Time (s)	Overhead (%)
sched	5,712	1.94	0.1%
ixx	11,157	5.22	1.4%
lcom	17,278	6.50	3.0%
hotwire	5,335	2.06	1.3%
simulate	6,672	2.75	5.6%
idl	30,288	6.42	1.4%
taldict	11,854	1.78	4.0%
deltablue	1,250	0.44	2.4%
richards	606	0.32	3.6%

Comparison: Alias Analysis

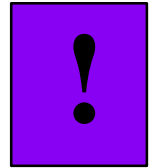


- ◆ Best precision from a static analysis
- ◆ Complex algorithm--expensive to implement
- ◆ Pande & Ryder [1996] C++ implementation
 - between 0.4 and 55 source lines analyzed/second
 - RTA is 45 to 8250 times faster
 - RTA=AA on half of their micro-benchmarks

Comparison



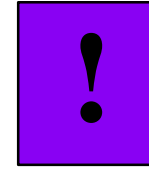
- ◆ Grove et al [1997] Java/Cecil Implementation
 - Compared CHA, RTA, more powerful algorithms
 - RTA significantly more precise than CHA
 - RTA significantly faster than other algorithms
 - for Java, RTA speedup equivalent in 4/5 programs
 - RTA usually *reduced* compile time



Impact: Industry & Academia

- ◆ RTA currently being implemented for inclusion in IBM's Visual Age C++ compiler
- ◆ Dead data members [Sweeney & Tip 1997]
 - up to 27% of data members removed
 - up to 12% dynamic space reduction
- ◆ RTA implementations for Java
 - U. Washington (Cecil group)
 - IBM Santa Teresa Laboratory
 - IBM Haifa Scientific Center

Rapid Type Analysis



- ◆ RTA is *effective*:
 - resolves 71% of virtual function calls
 - reduces code size by 25%
 - often as good as much more powerful alg.'s
- ◆ RTA is *fast*:
 - analyzes 3300 lines/second on 80 MHz PPC
- ◆ RTA is *practical*
 - not hard to implement, often reduces compile time