Dynamic Assignment of Scoped Memory Regions in the Translation of Java to Real-Time Java

Morgan Deters

Advisor: Dr. Ron K. Cytron

Distributed Object Computing Laboratory
Washington University
St. Louis, MO

Research funded by DARPA under contract F33615-00-C-1697

M.S. Thesis Defense

10 March 2003

Copyright © 2003 Morgan Deters

Overview

Java

Real-Time

Dynamic Analysis

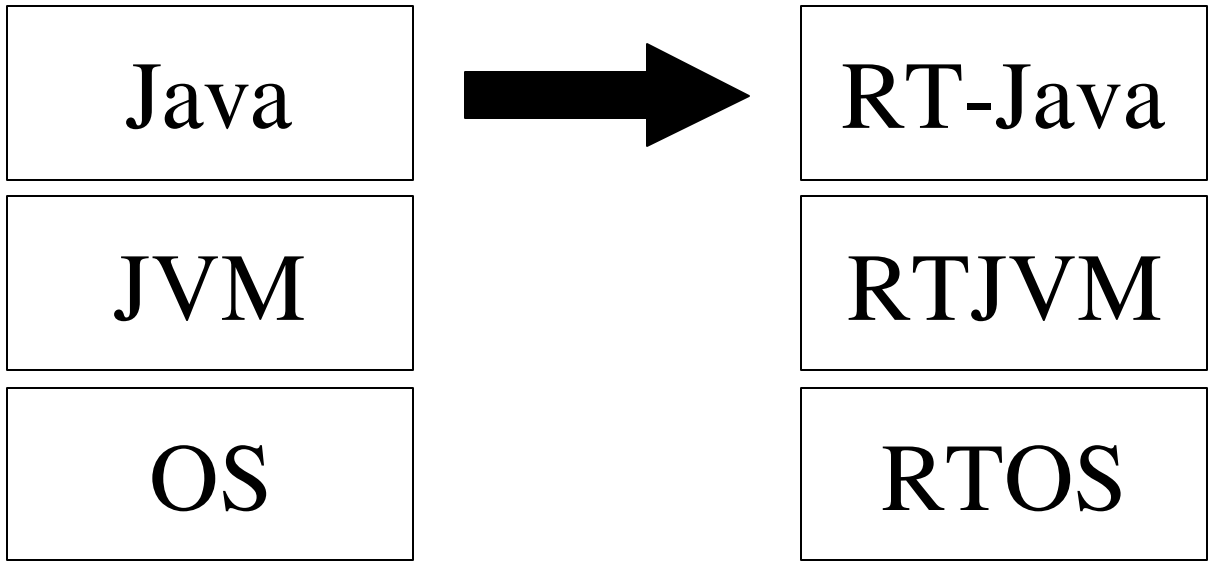
Memory Regions

Aspect-Oriented Programming

Scoped Memory

Garbage Collection Avoidance

Big Picture



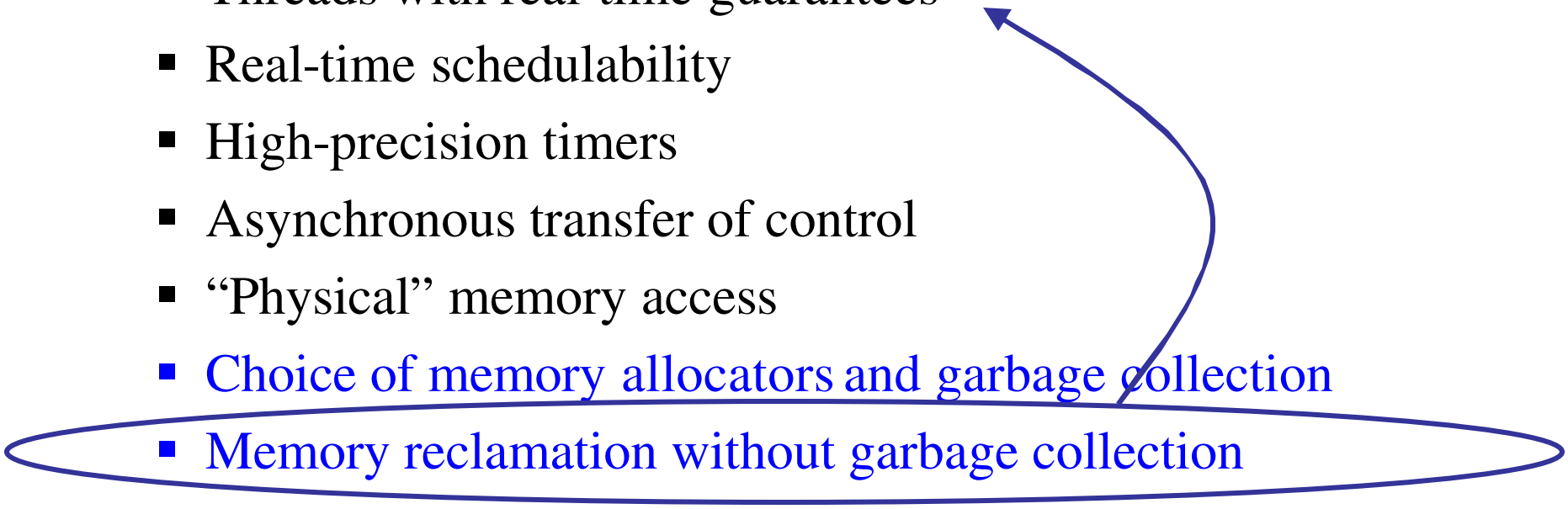
Java vs. Real-Time

- High-level
- Popular
- Object-oriented
- Garbage collection
- Reusable components
- Flexible, adaptable
- Scalable
- Portable
- Optimized average-case
- Memory use varies
- Coarse control of concurrency

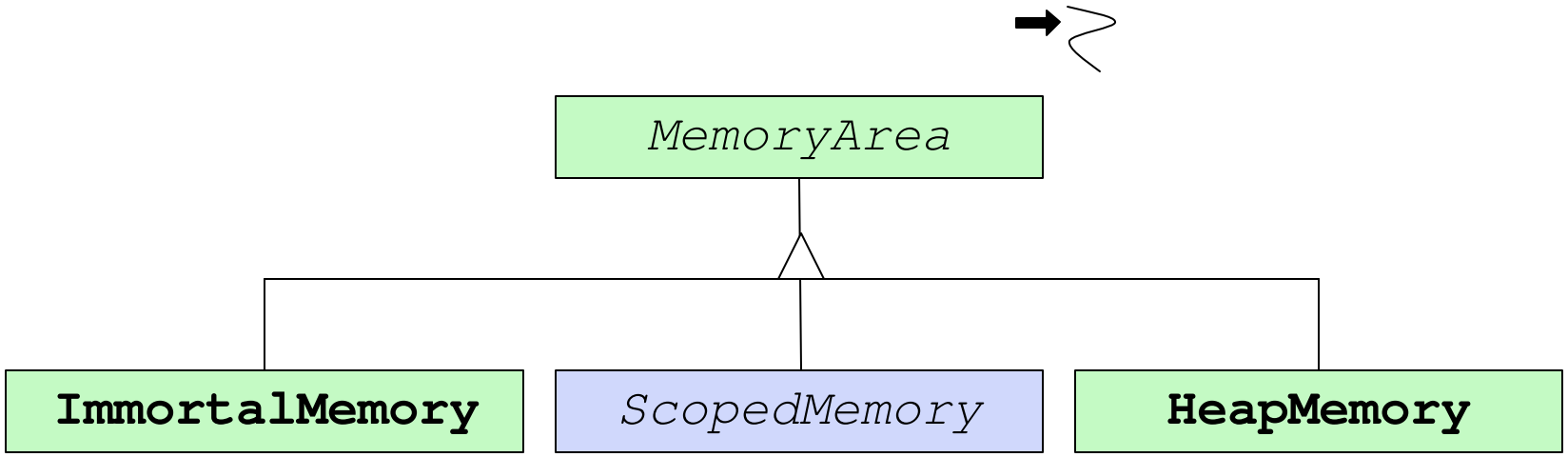
- Low-level
- Specialized
- Manually-optimized
- Specialized allocators
- Tailored components
- Hard-coded behavior
- May not scale
- Hardware-dependent
- Reasonable worst-case
- Footprint, latency critical
- Fine control over concurrency

Real-Time Java ?

- Real-Time Specification for Java (RTSJ, JSR001) brings infrastructure for real-time accountability to Java
  - Threads with real-time guarantees
  - Real-time schedulability
  - High-precision timers
  - Asynchronous transfer of control
  - "Physical" memory access
  - Choice of memory allocators and garbage collection
  - Memory reclamation without garbage collection



Real-Time Java Memory Hierarchy



Rules for Scope References

14. Strict assignment rules placed on assignments to or from memory areas prevent the creation of dangling pointers, and thus maintain the pointer safety of Java.

Thou shalt not reference any object whose lifetime could be shorter than thine own.

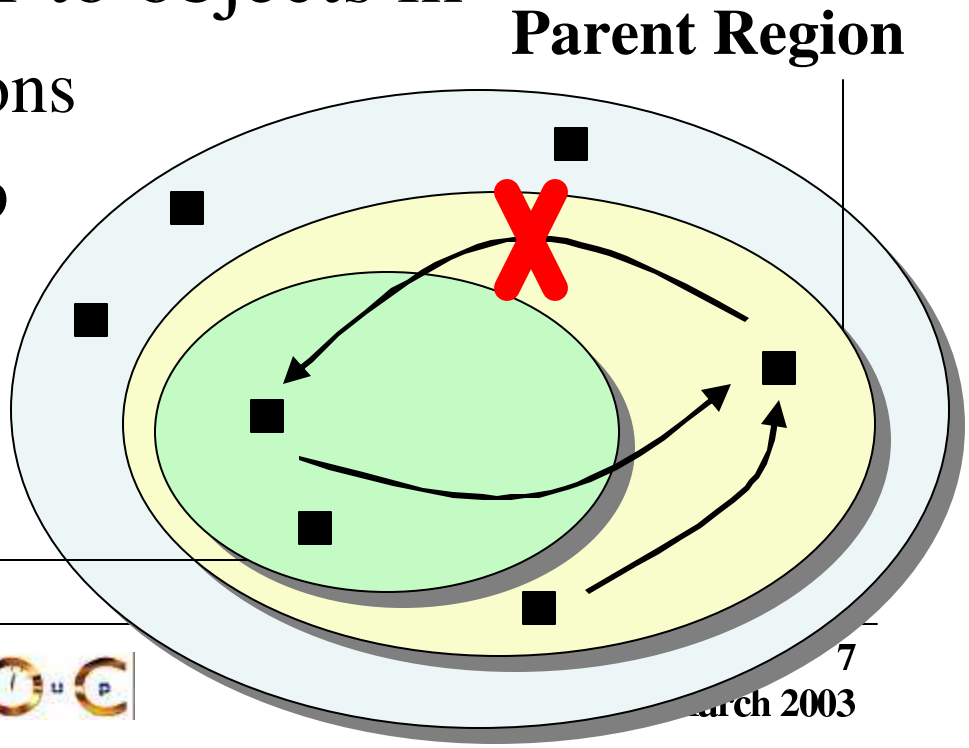
			or shared scope
Local Variable	Yes	Yes	Yes, if same, outer, or shared scope

From the Real-Time Specification for Java v1.0

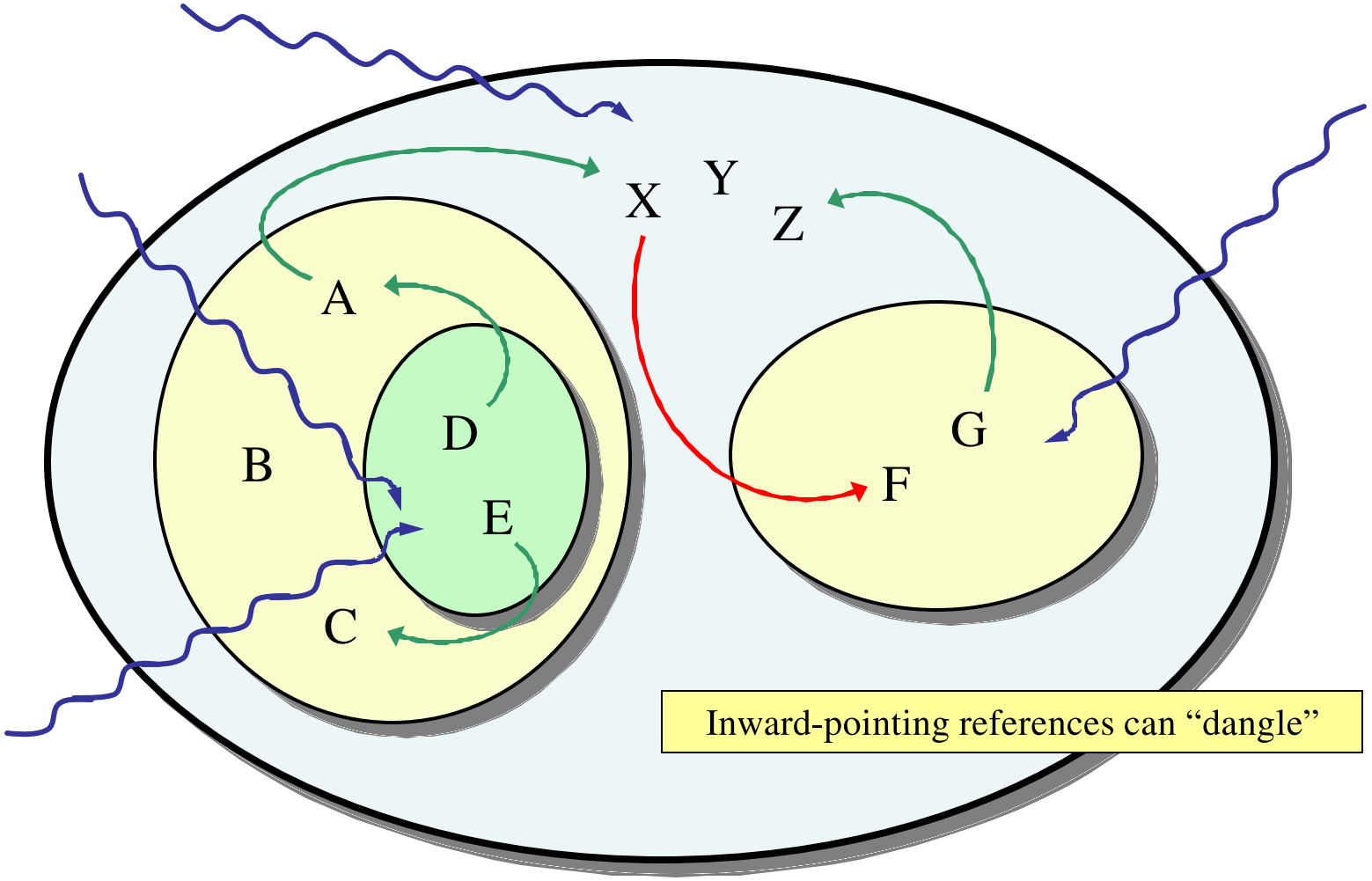
Inter-Object Reference Rules

An object cannot refer to an object from a shorter-lived memory region!

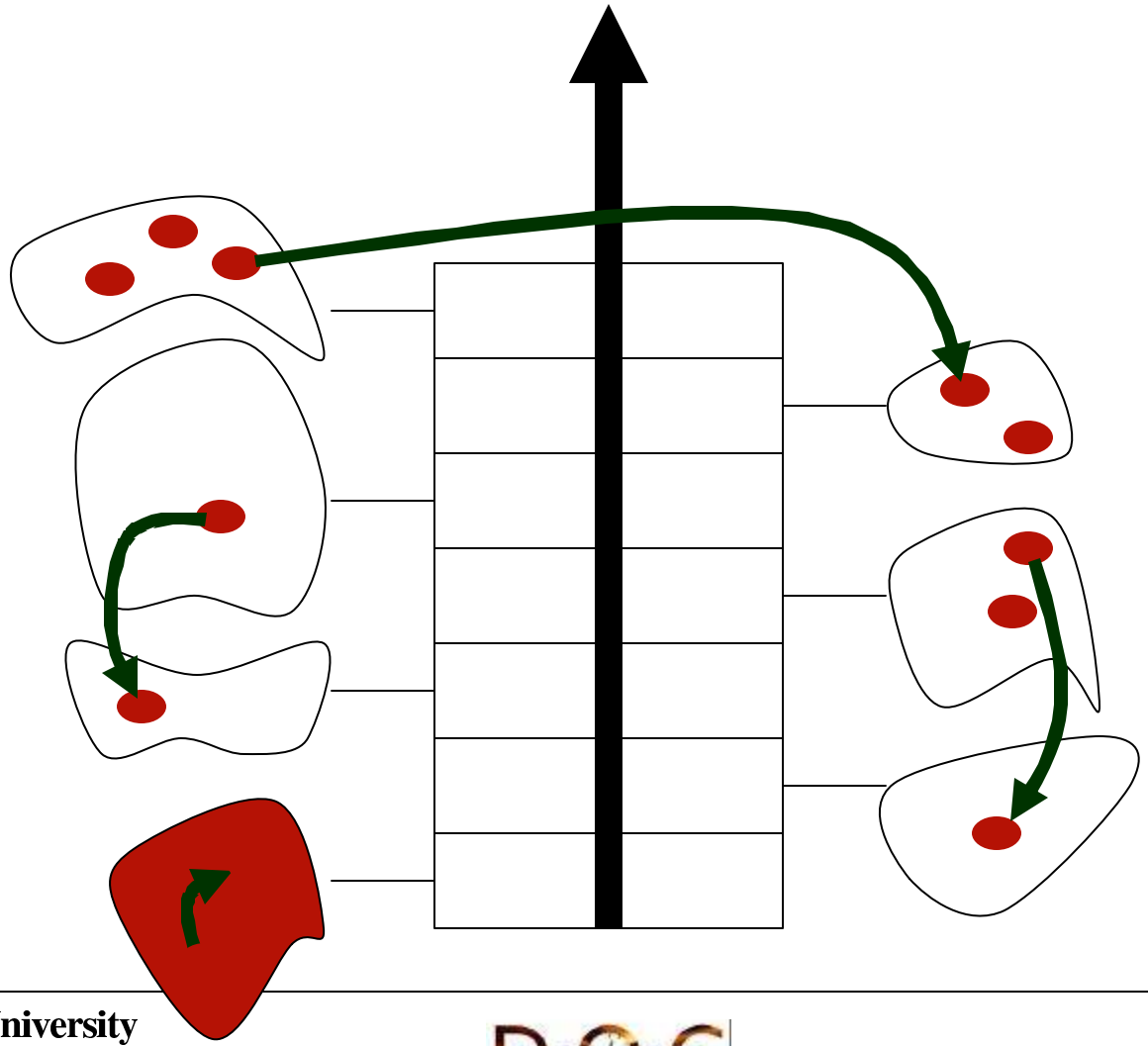
- Objects may only refer to objects in
  - ancestor memory regions
  - garbage-collected heap



Nested ScopeMemory References



A Stack of Regions



Using Real-Time Java Scoped Memory

- Consider class *C*:

```
class C {
  void foo() {
    bar();
    // ...
  }
  void bar() {
    // generate lots of garbage
  }
}
```

Using Real-Time Java Scoped Memory

- `foo()` calls `bar()` in a separate region

```
class C {
  void foo() {
    ScopedMemory sc = new ScopedMemory(1024,1024);
    sc.enter(new Runnable() {
        public void run() {
          bar();
        }
      });
  }
  void bar() {
    // generate lots of garbage
  }
}
```

Using Real-Time Java Scoped Memory

```
class C {
  Object barResult; // assume single-threaded
  void foo() {
    ScopedMemory sc = new ScopedMemory(1024,1024);
    sc.enter(new Runnable() {
        public void run() {
          barResult = bar();
        }
      });
    // now use the value in barResult...
  }
  Object bar() {
    // generate lots of garbage
    // ... new MyObject() ...
  }
}
```

Illegal reference

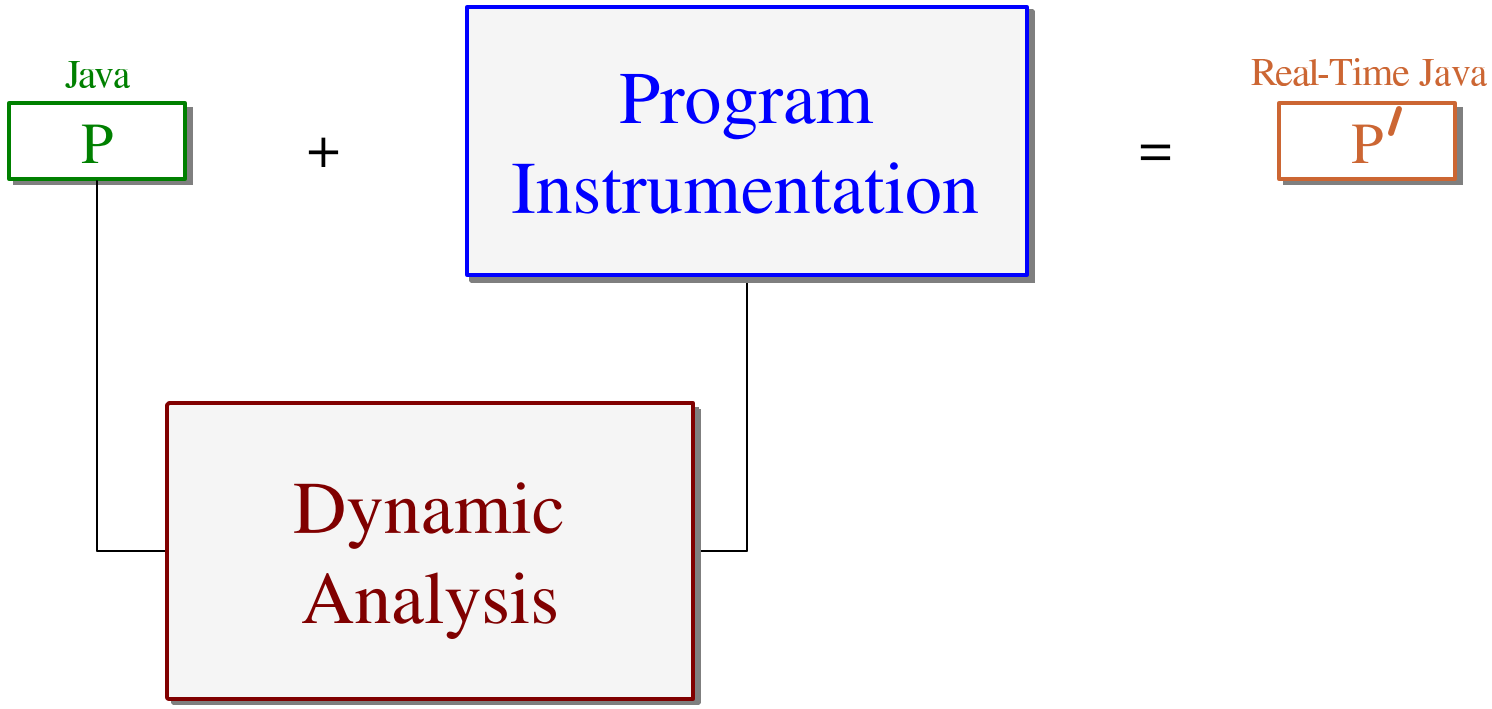
Allocated in scoped memory lt

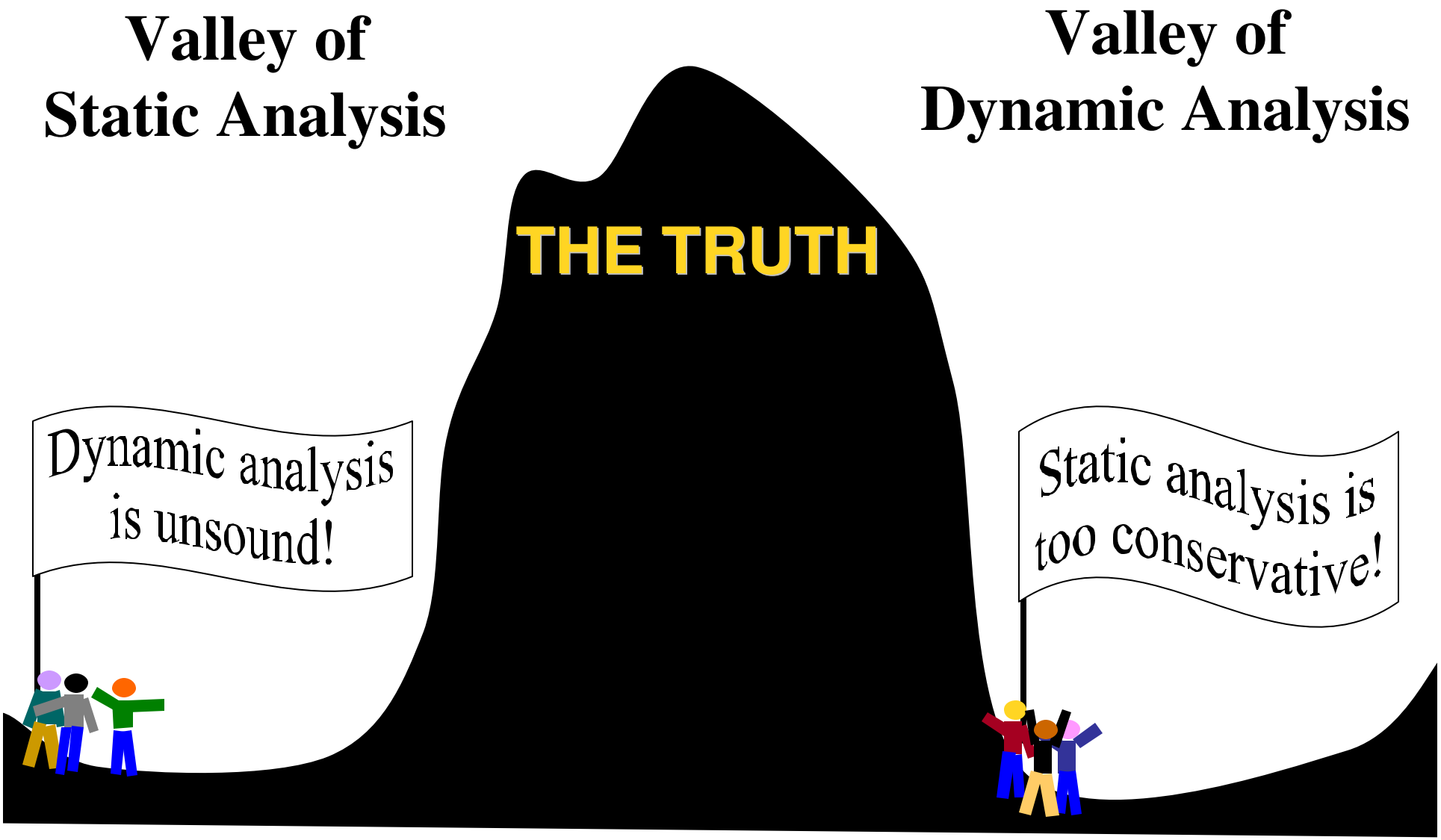
RT-Java Memory Summary

- Assign memory regions to execution scopes
- `new` operations allocate from current region

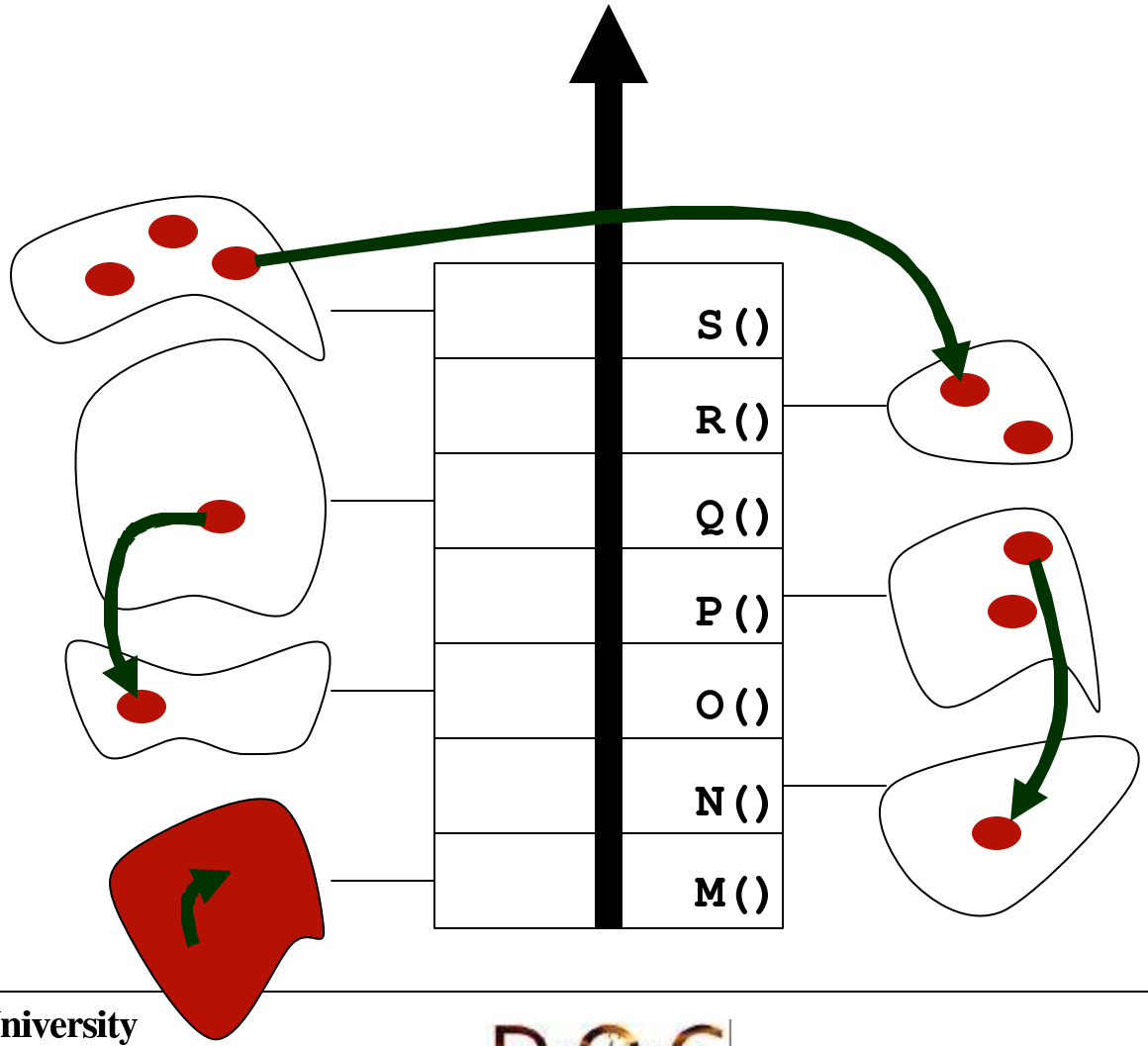
- Introduces memory management dependencies between caller and target
- Hurts reusability of code
- *Integrates* concerns of program
- Introduces interdependence of libraries

High-Level View




Valley of
Static Analysis
Valley of
Dynamic Analysis
THE TRUTH
Dynamic analysis
is unsound!
Static analysis is
too conservative!

Stack of Regions == Call Stack



Given program *P* …

Goal:

$$A_P : O_P \; ? \; F_P$$

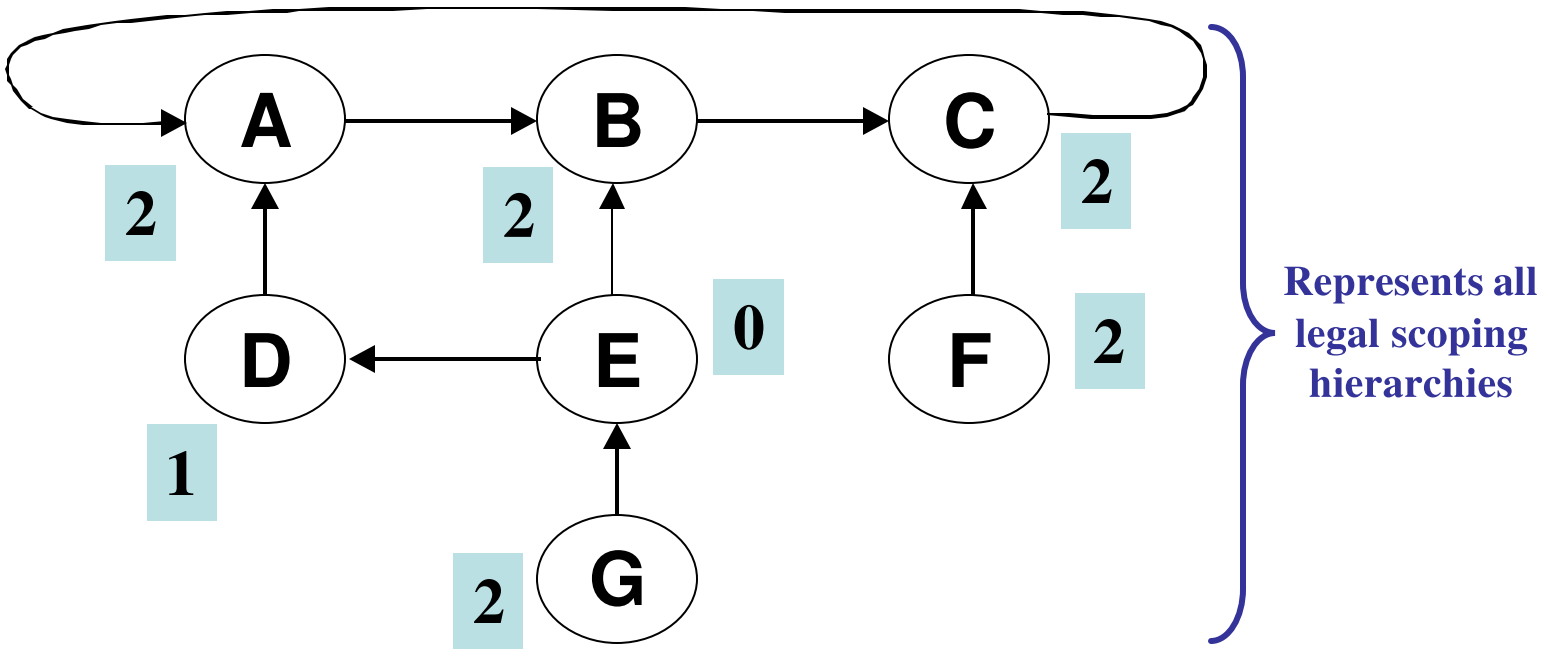
O_P: objects of program *P*
F_P: call-stack frames of program *P*

Approach

- Run program *P*…
- Observe referencing behavior
- Observe object lifetimes
- Come up with scoping hierarchy that does this

Determine Referencing and Object Lifetimes

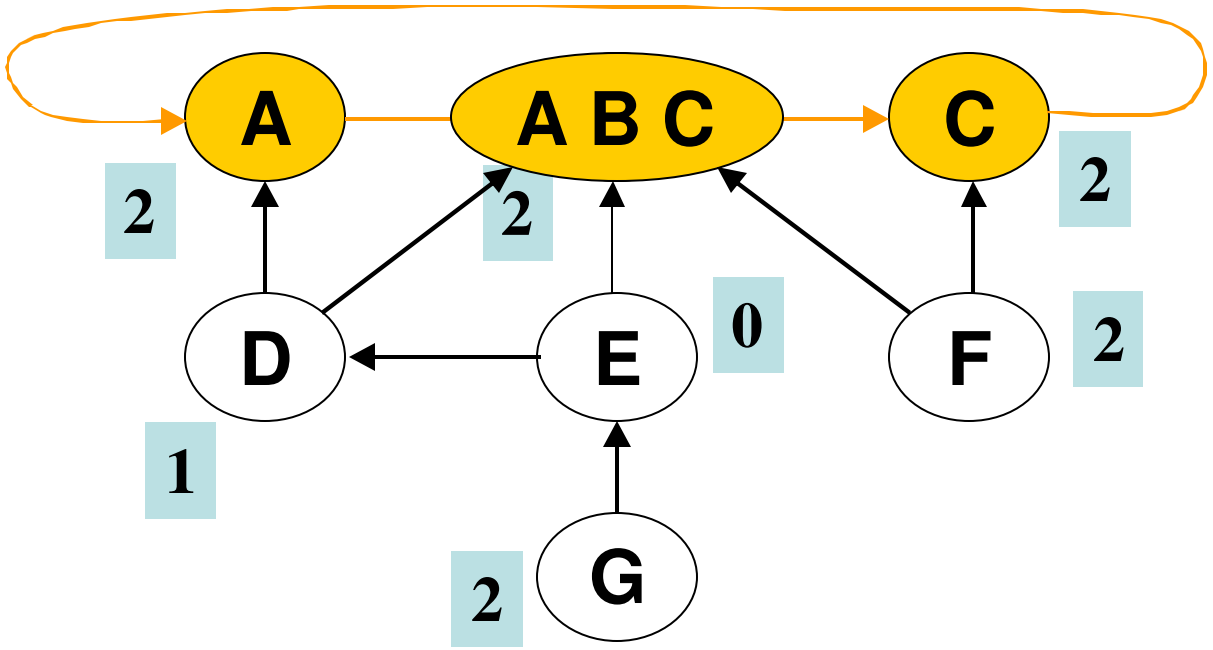
- Keep a *doesReference* graph



- Annotate it with object death frames

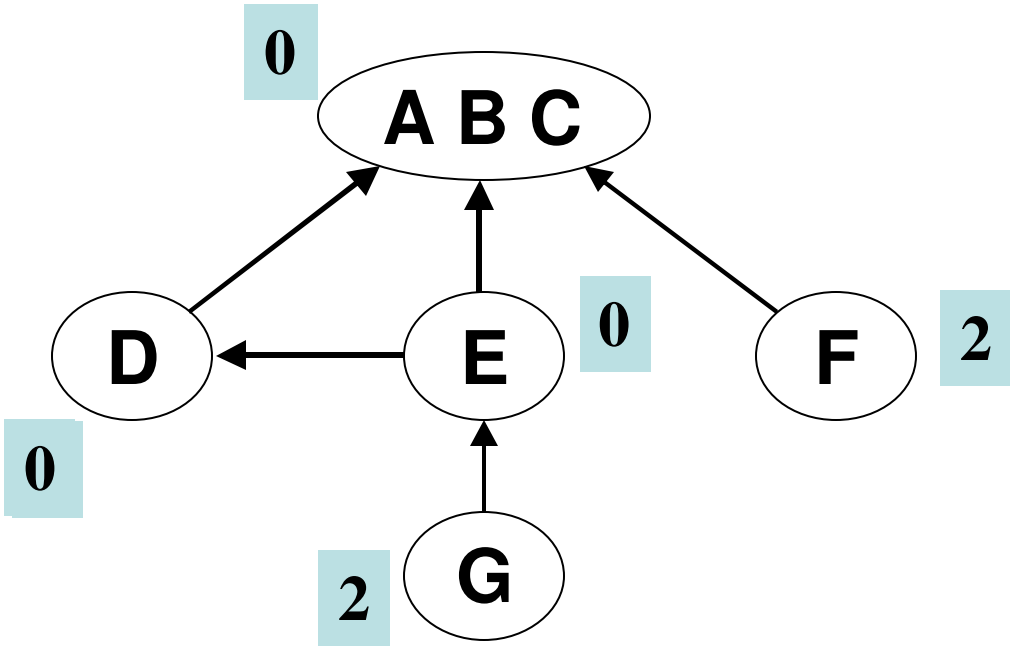
Solving the *doesReference* graph

- Decompose into strongly-connected components



Solution to *doesReference* graph

- Propagate smaller numbers through the graph



Q: How to recognize objects from one run to another?

A: Use an object's *allocation site* as a discriminator

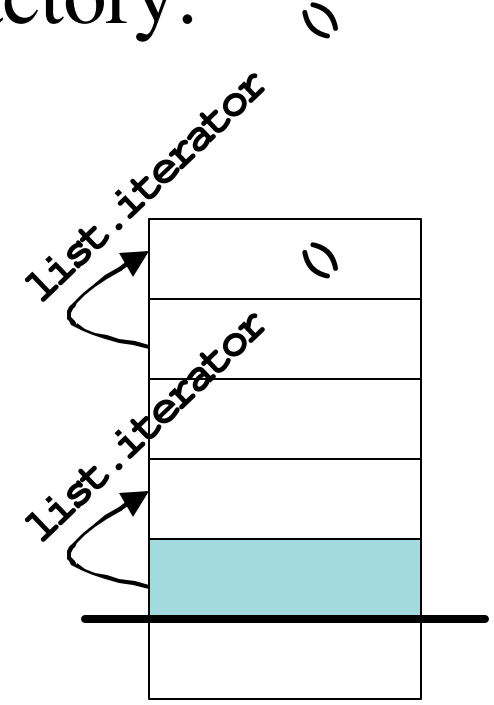
Object Behavior Categories

- How to map objects observed in previous analyses to objects observed in current program execution?
  - *Class-based* behavior
    - All instances of a class behave similarly
  - *"New site"-based* behavior
    - All instances of a class instantiated at the same source code location behave similarly
  - *Instance-based* behavior
    - Instances of a class behave differently

Impact of Simplification

- Consider a method that acts as a Factory:

```
class LinkedList {
  public Iterator iterator() {
    return new LLIterator();
  }
}

Iterator i = list.iterator();
while(i.hasNext()) {
  // ...
}
```



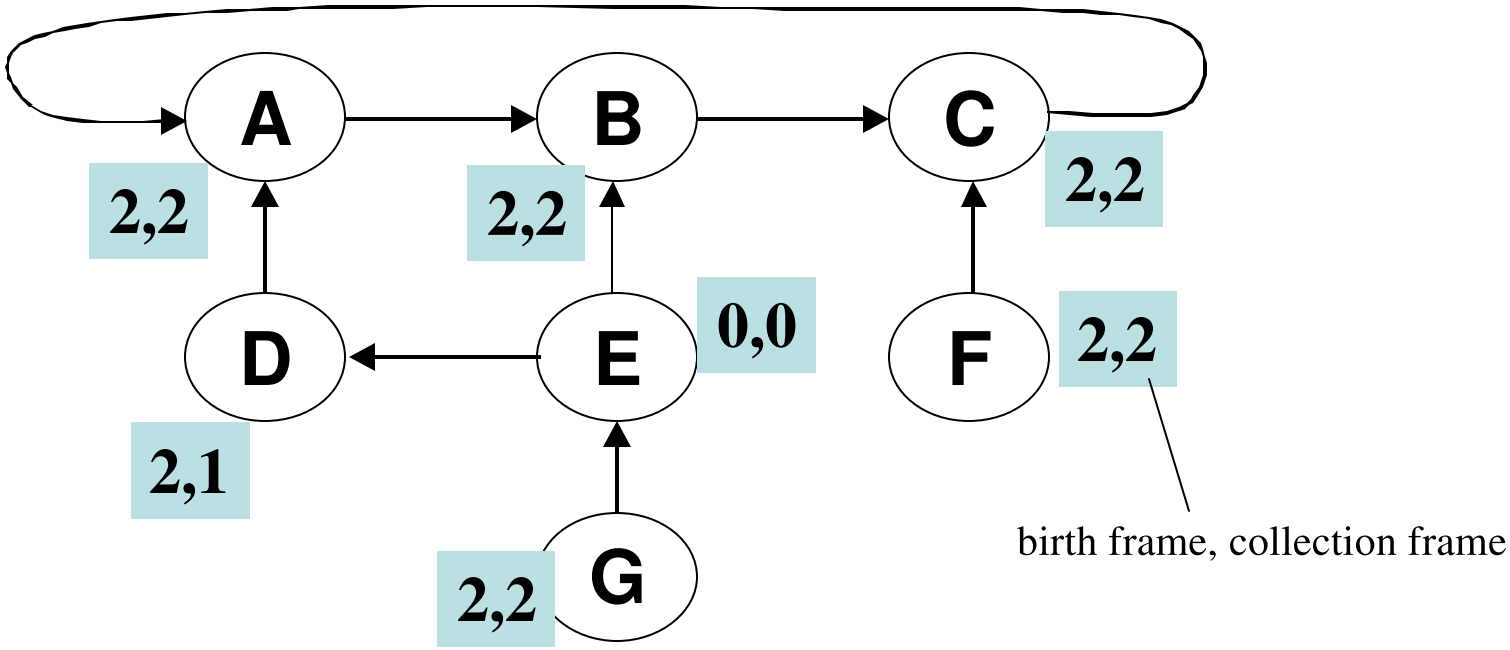
Given program *P* …

Revised Goal:

$$A_P : S_P \; ? \quad ? F_P$$

S_P: allocation sites of program *P*
F_P: call-stack frames of program *P*
$?F_P$: object lifetimes in # of frames

Determine Referencing and Object Lifetimes



Solving the Graph

- Liveness constraints:

  For each vertex v

  $$Scope(v) = \max_{o \in Objects(v)} o_i - o_c$$

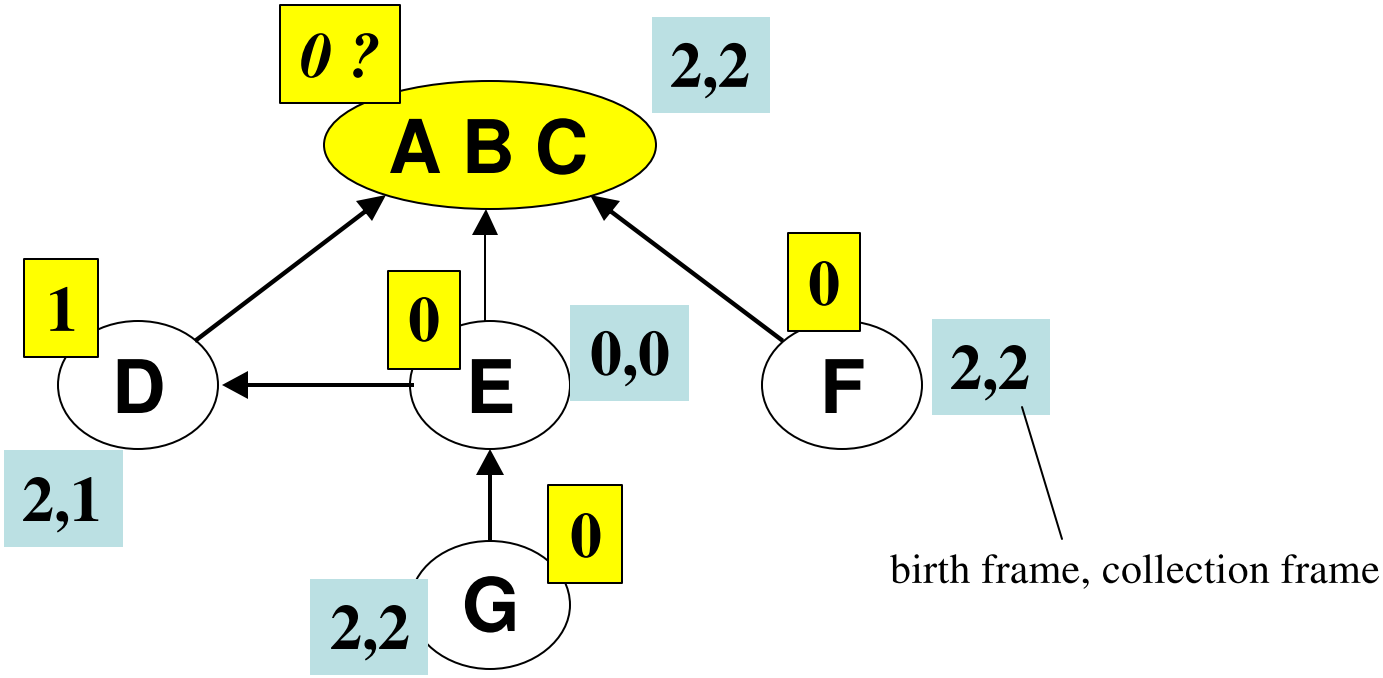
  o_i = object allocation frame
  o_c = object collection frame

- Reference constraints:

  For each edge (u,v)

  $$Scope(v) = Scope(u) + \max_{o \in Objects(v)} o_i - \min_{o \in Objects(u)} o_i$$

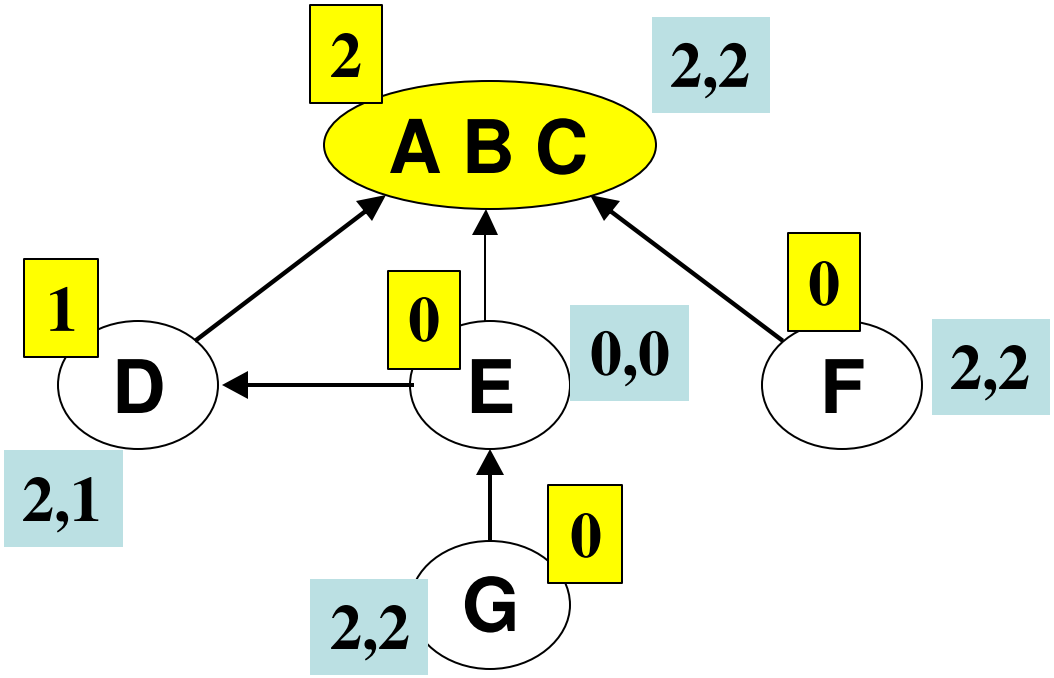
Solving the Graph



Reference Constraint:

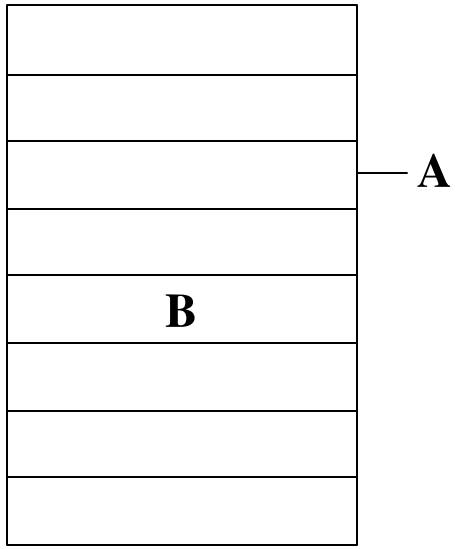
$$\text{Scope}(ABC) = \text{Scope}(E) + \max_{ABC} o_i - \min_{E} o_i$$

Solution to *doesReference* graph



Deriving the Reference Constraint

- $Scope(v) = Scope(u) + \max_{o \in Objects(v)} o_i - \min_{o \in Objects(u)} o_i$

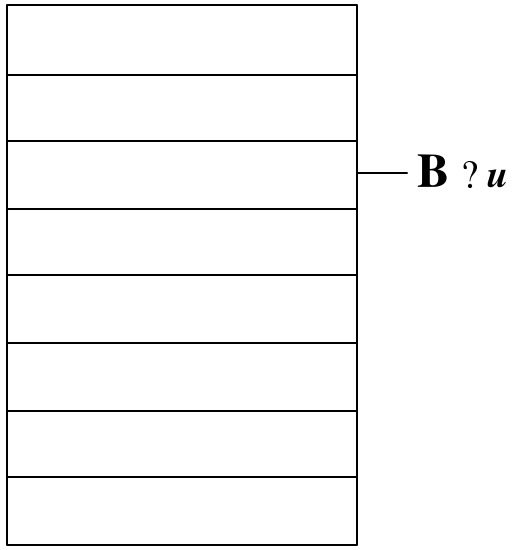


In *absolute depth* model using *individual objects*:
Suppose we know that object A references object B
We want to instantiate A – *where does it go* ?

Scope(B) = Scope(A)

Deriving the Reference Constraint

- $Scope(v) = Scope(u) + \max_{o \in Objects(v)} o_i - \min_{o \in Objects(u)} o_i$



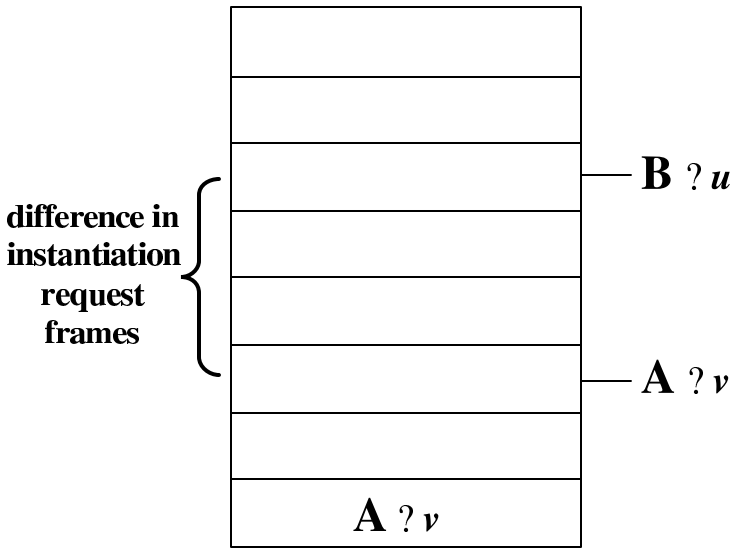
In *relative depth* model using *instantiation sites*:
Suppose we know that *u* references *v*
We want to create something at *u* – *where does it go* ?

Scope(*v*) = Scope(*u*)

but there's another case…

Deriving the Reference Constraint

- $Scope(v) = Scope(u) + \max_{o \in Objects(v)} o_i - \min_{o \in Objects(u)} o_i$



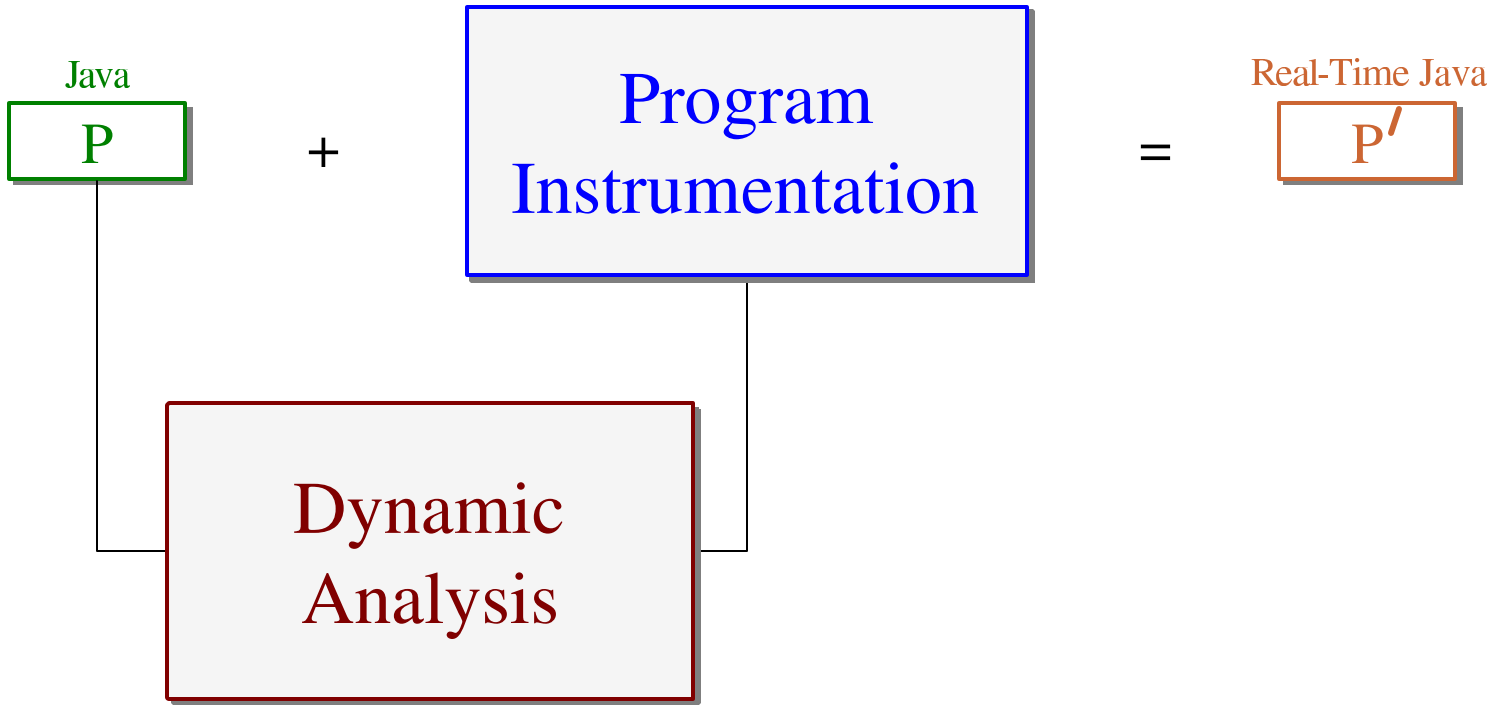
In *relative depth* model using *instantiation sites*:
Suppose we know that *u* references *v*
We want to create something at *u* – *where does it go* ?

Scope(*v*) = Scope(*u*)

but there's another case…

Scope(*v*) = Scope(*u*) + *max difference*

High-Level View



Aspect-Oriented Programming

- Advanced Separation of Concerns (ASoC)
- *quantification*
  - Our object analysis and program instrumentation can be expressed in modular pieces of code
- *obliviousness*
  - Target code is *oblivious* to this extra processing
    - it need not be changed at all

Reference Probe

1. Determine references between objects
   a. Track objects' references to other objects
   b. Build a *doesReference* graph

2. Discover legal scoping hierarchies
   a. Collapse strongly connected components
   b. The resulting DAG is a representation of legal scoping hierarchies

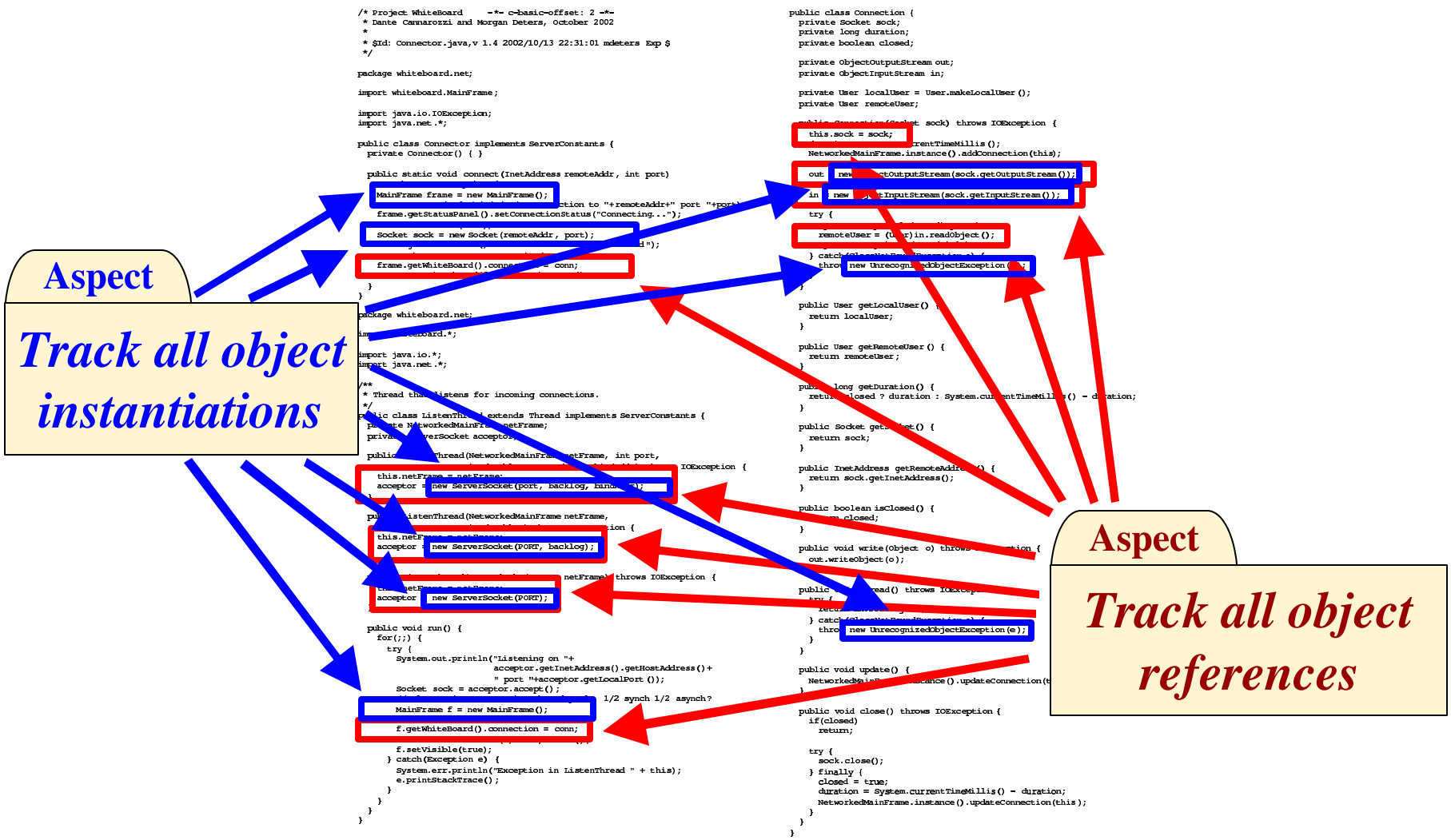
Effect of Reference Probe

```
class C {
  MyObject anObject;

  void foo() {


    anObject = new MyObject();


    // do other stuff...
  }

}
```

Effect of Reference Probe

```
class C {
  MyObject anObject;

  void foo() {
    MyObject o = new MyObject();
    ReferenceProbe.newObject(o, getSourceLocation());

    anObject = o;
    ReferenceProbe.registerReference(this, o);

    // do other stuff...
  }

}
```

Aspect "Weaving"



Object Liveness Probe

1. Determine the points at which objects become collectible
   a. Track object birth times
   b. After each execution join point, run a garbage collection cycle and determine which objects were collected
2. Express object liveness behavior in terms of:
   a. Enclosing execution join points
   b. Distance from birth to death

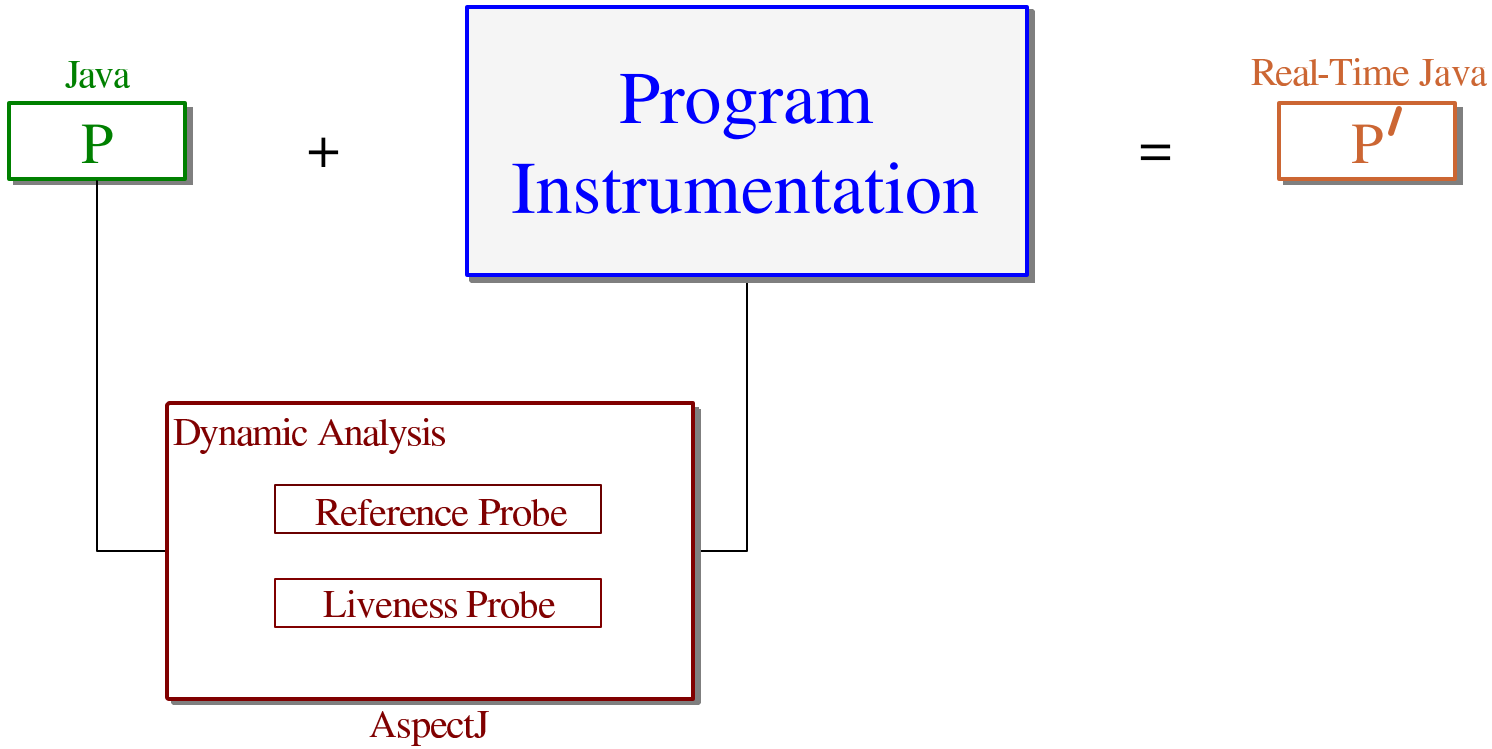
Effect of Liveness Probe

```
class C {
  MyObject anObject;

  void foo() {

    anObject = new MyObject();


    // do stuff...


  }

}
```

Effect of Liveness Probe

```
class C {
  MyObject anObject;

  void foo() {

    anObject = new MyObject();
    LivenessProbe.registerObject(anObject);

    // do stuff...

    LivenessProbe.gcCycle();
  }

}
```

High-Level View



Adding an Enforcement Aspect

- We have:
  - A set of all legal ScopedMemory hierarchies
  - Object lifetime behavior (join points)
- From this information, we:
  - Select a desirable, legal scope memory hierarchy
  - Direct a dispatch unit to implement this hierarchy

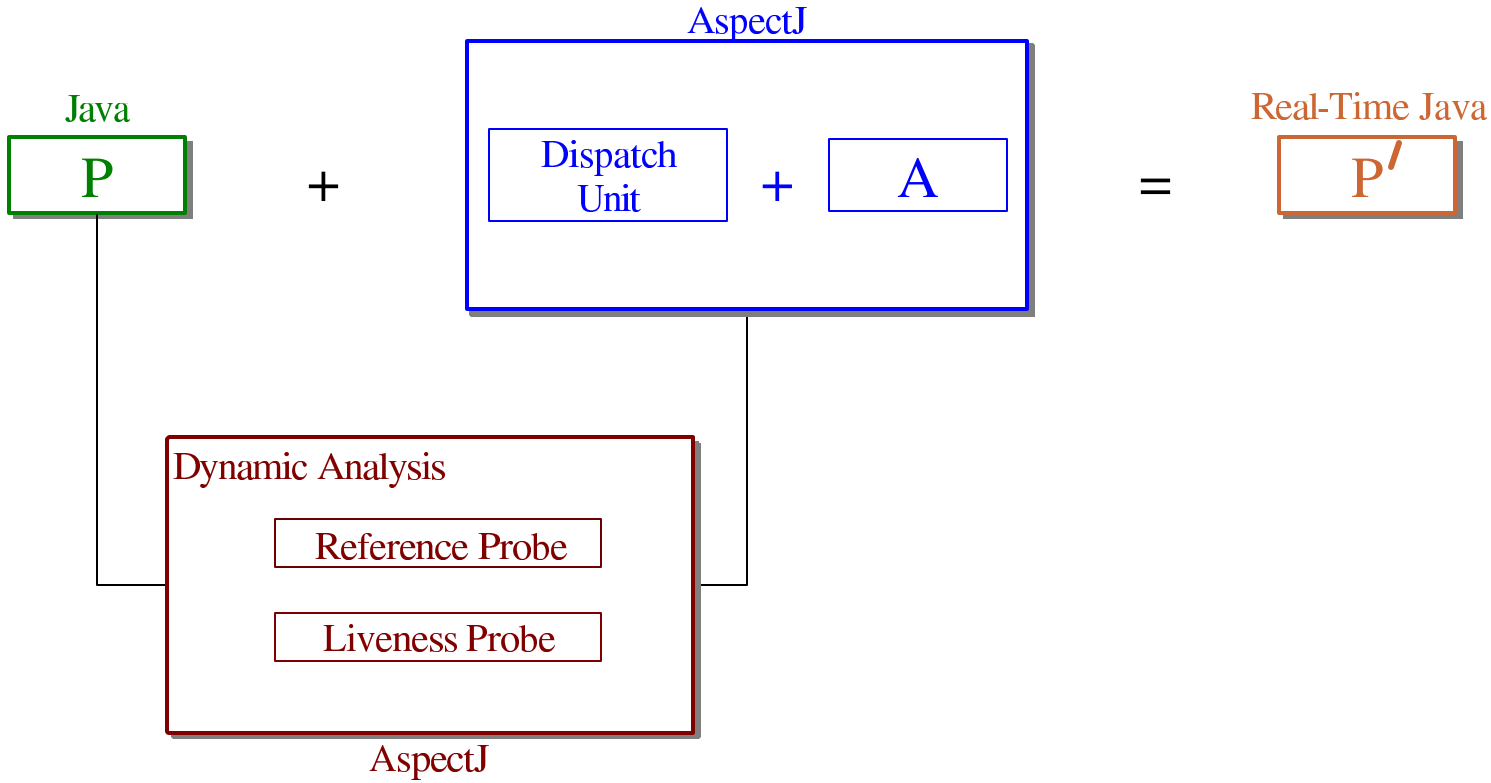
Effect of Scope Enforcement

```
class C {
  void foo() {


    bar();

    // do other stuff...
  }

  void bar() {
    // generate lots of garbage
  }

}
```

Effect of Scope Enforcement

```
class C {
  void foo() {
    ScopedMemory sc = new ScopedMemory(1024,1024);
    sc.enter(new Runnable() {
      bar();
    });
    // do other stuff...
  }

  void bar() {
    // generate lots of garbage
  }

}
```

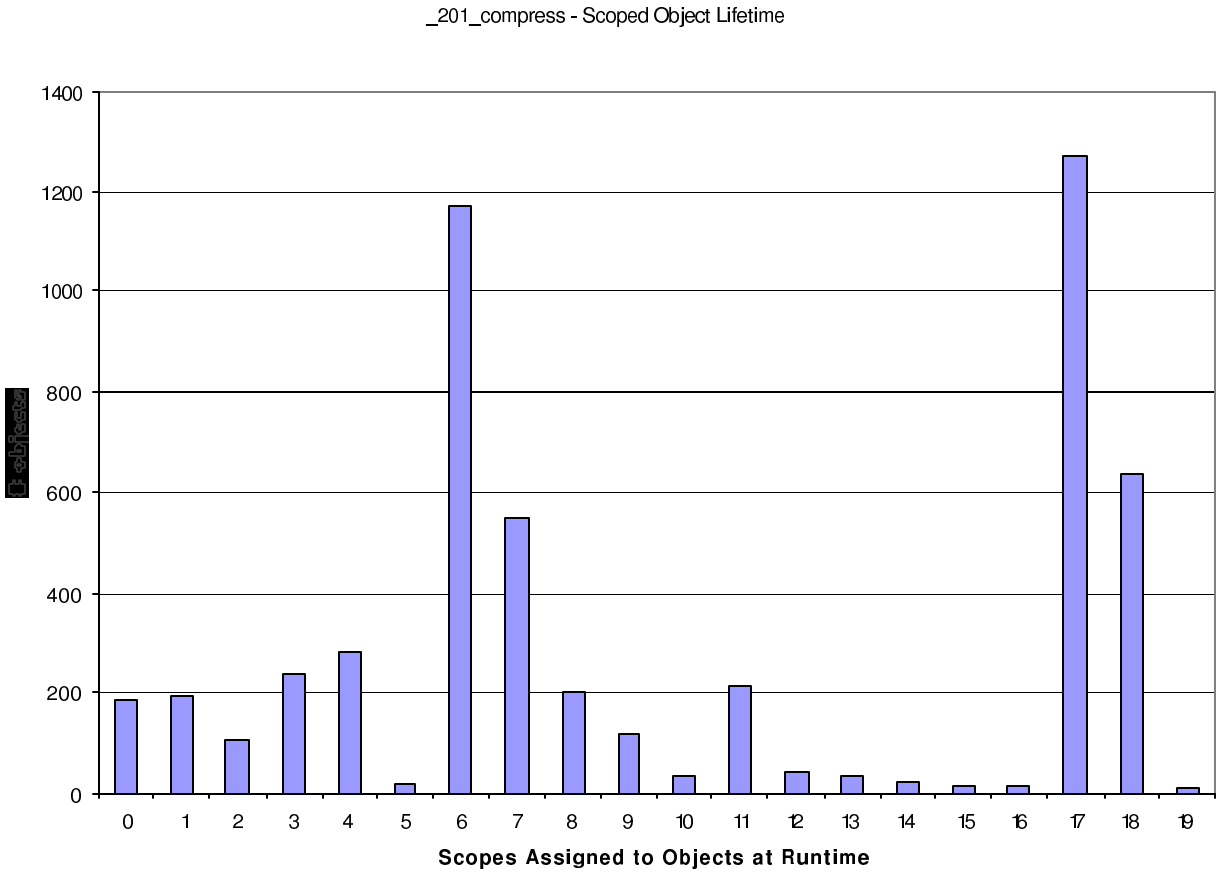
High-Level View



Experimentation

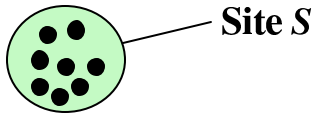
- Instrumented JVM 1.1.8 to emit
  - object allocations
  - object collections
  - intra-heap references
- SPECjvm98 benchmarks – size 10

Compress assigned scopes

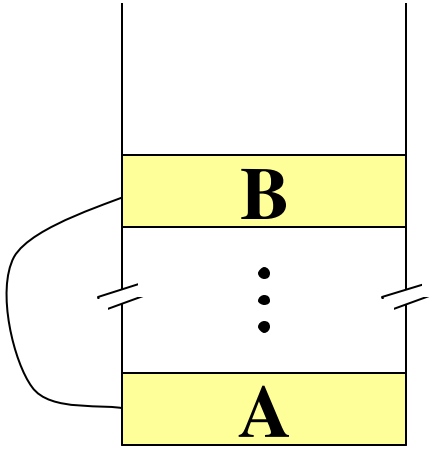


Understanding Object Lifetimes

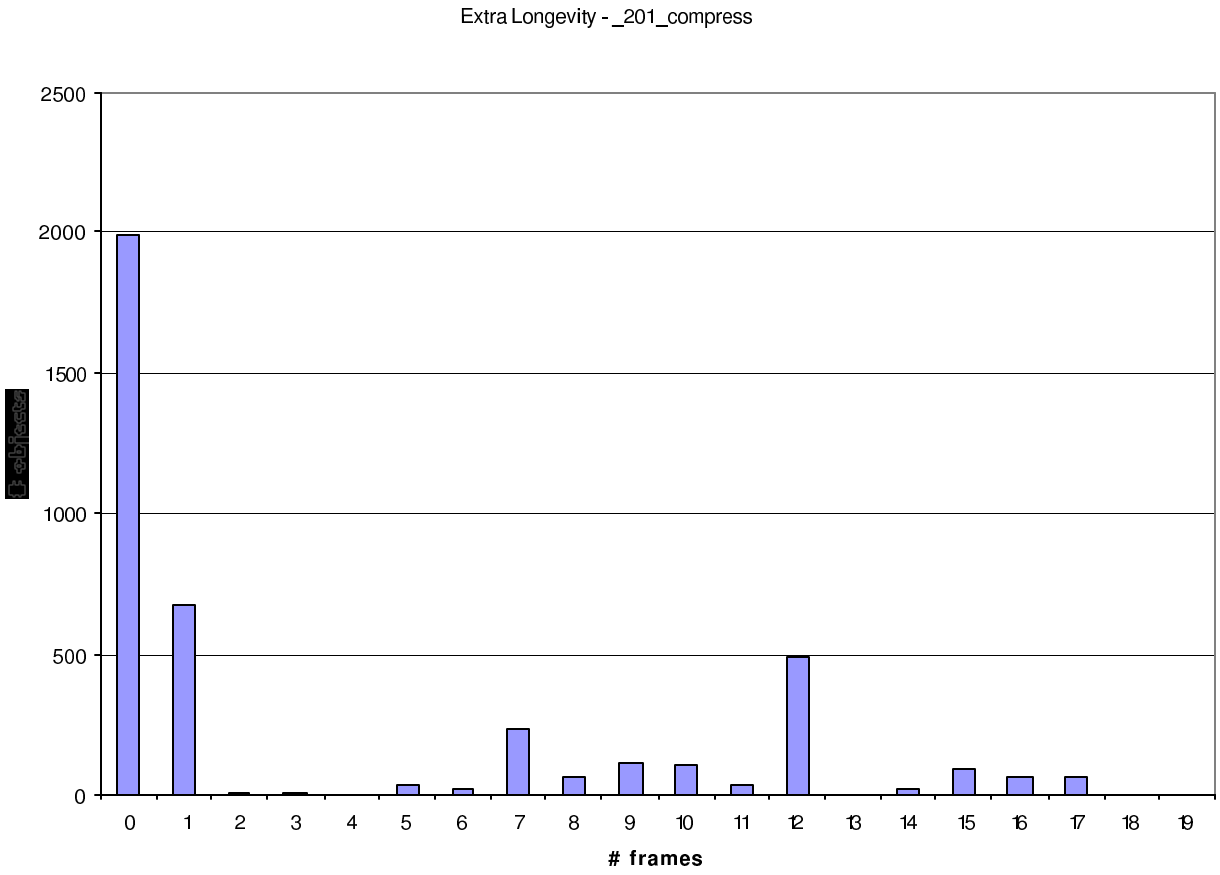
- Two reasons objects live longer than necessary
  - *Allocation site* assumption



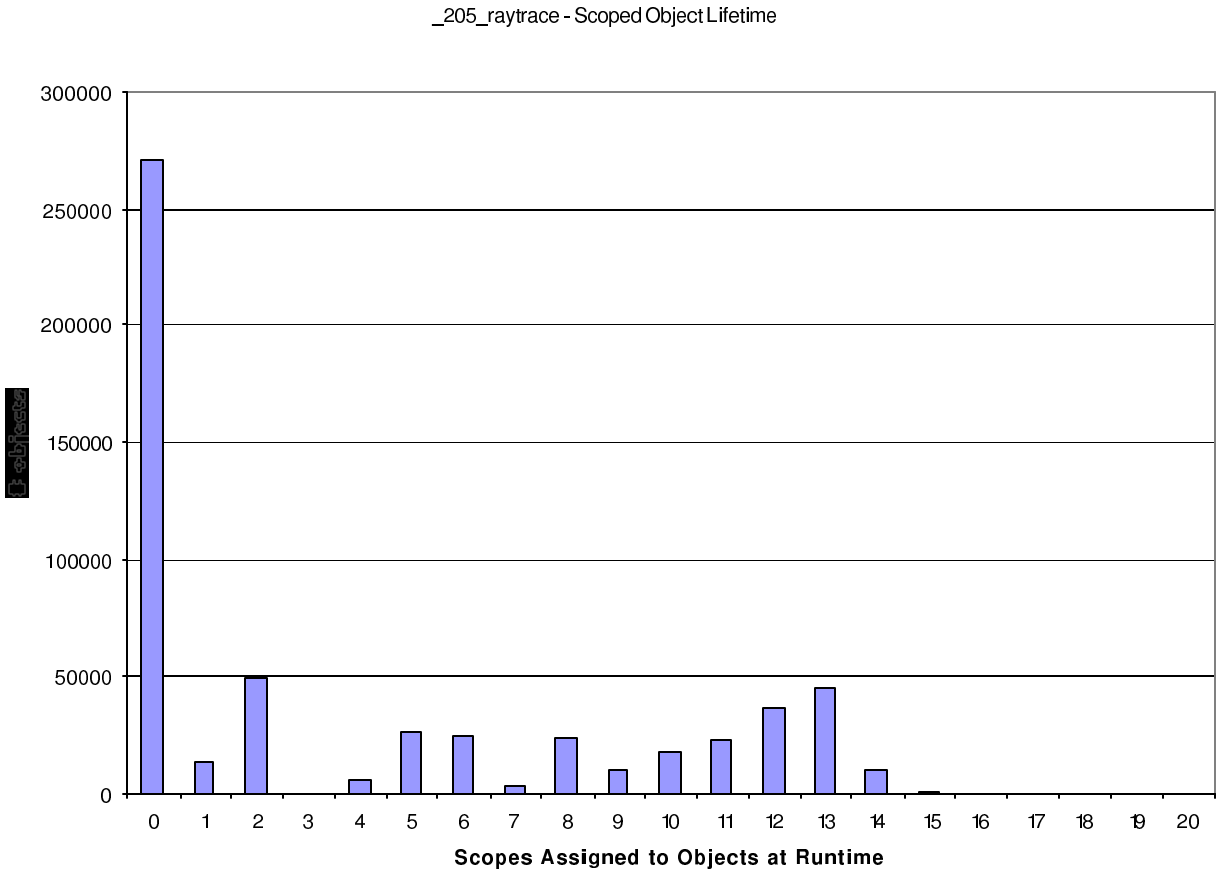
  - Referencing requirements



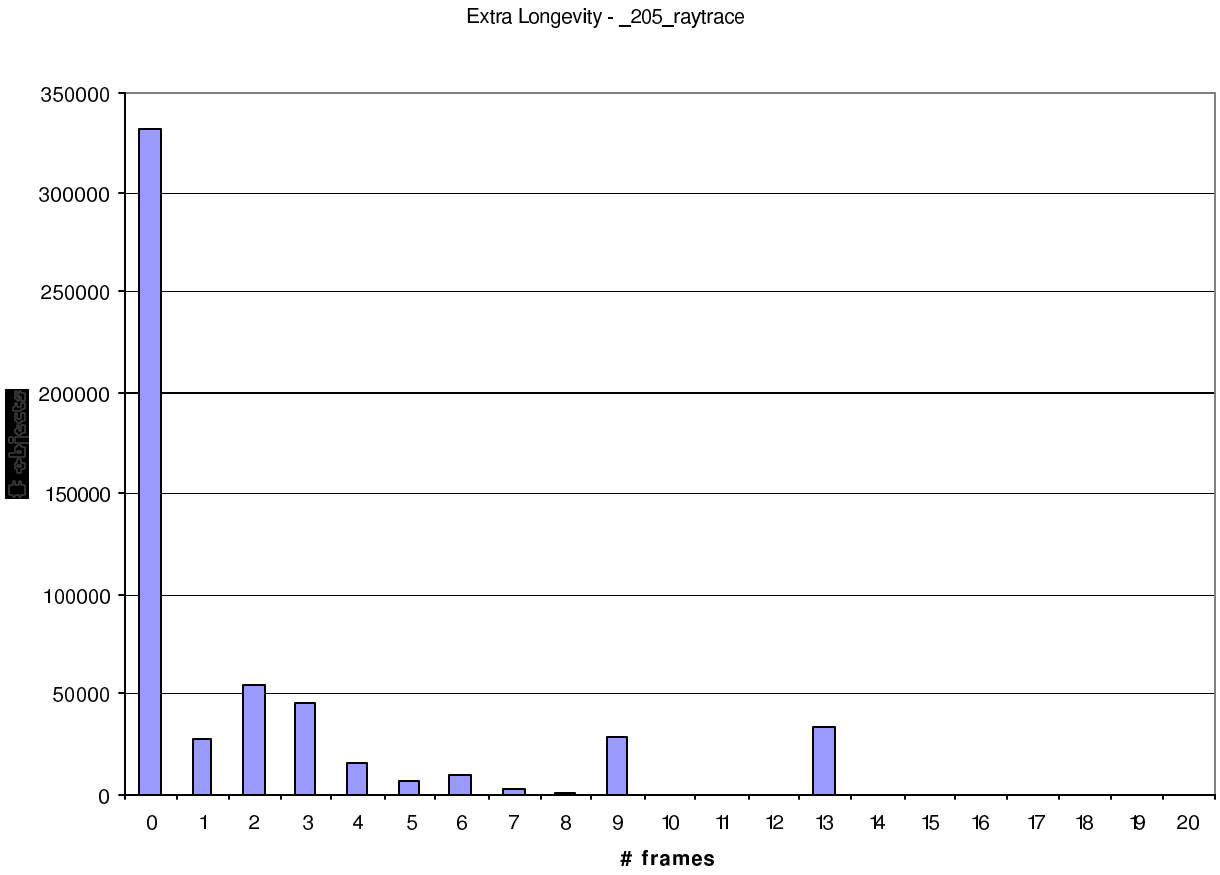
Compress extra longevity



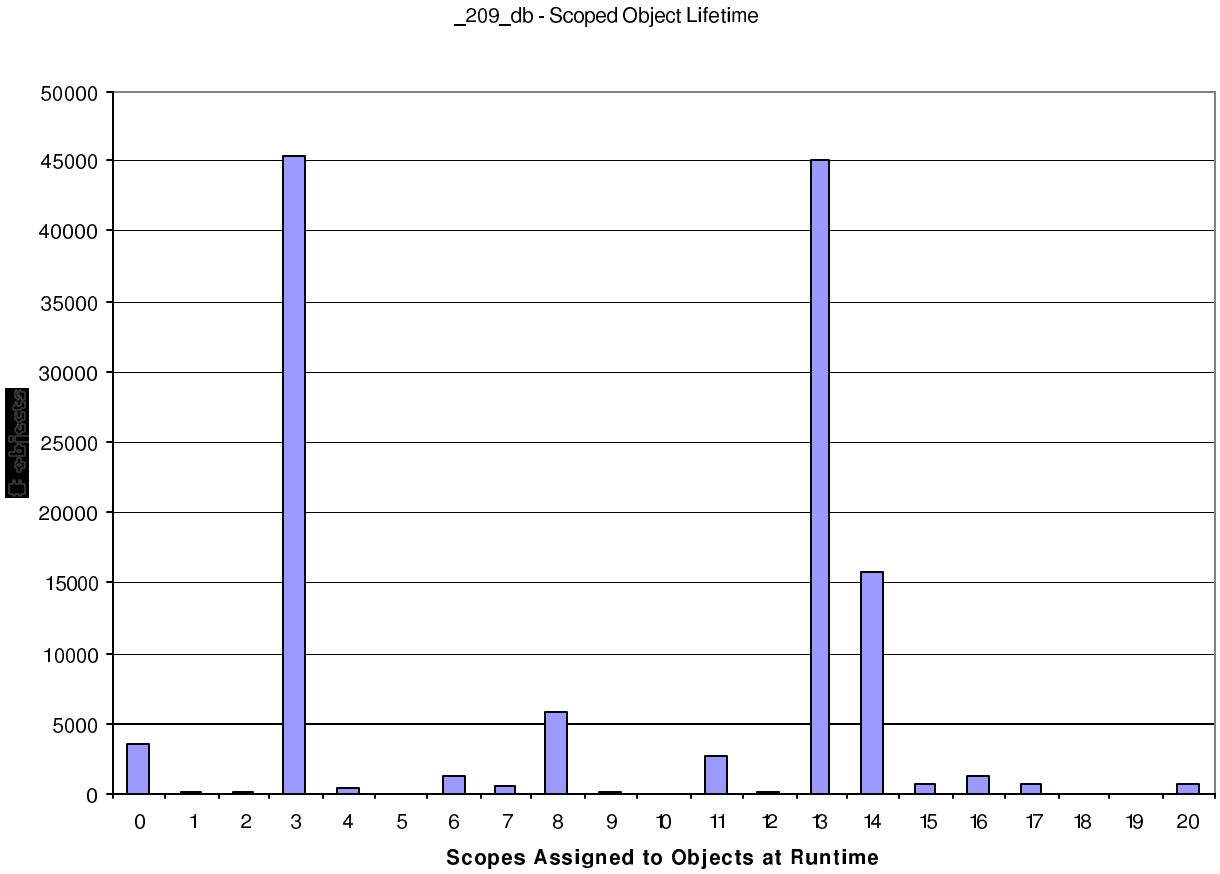
Raytrace assigned scopes



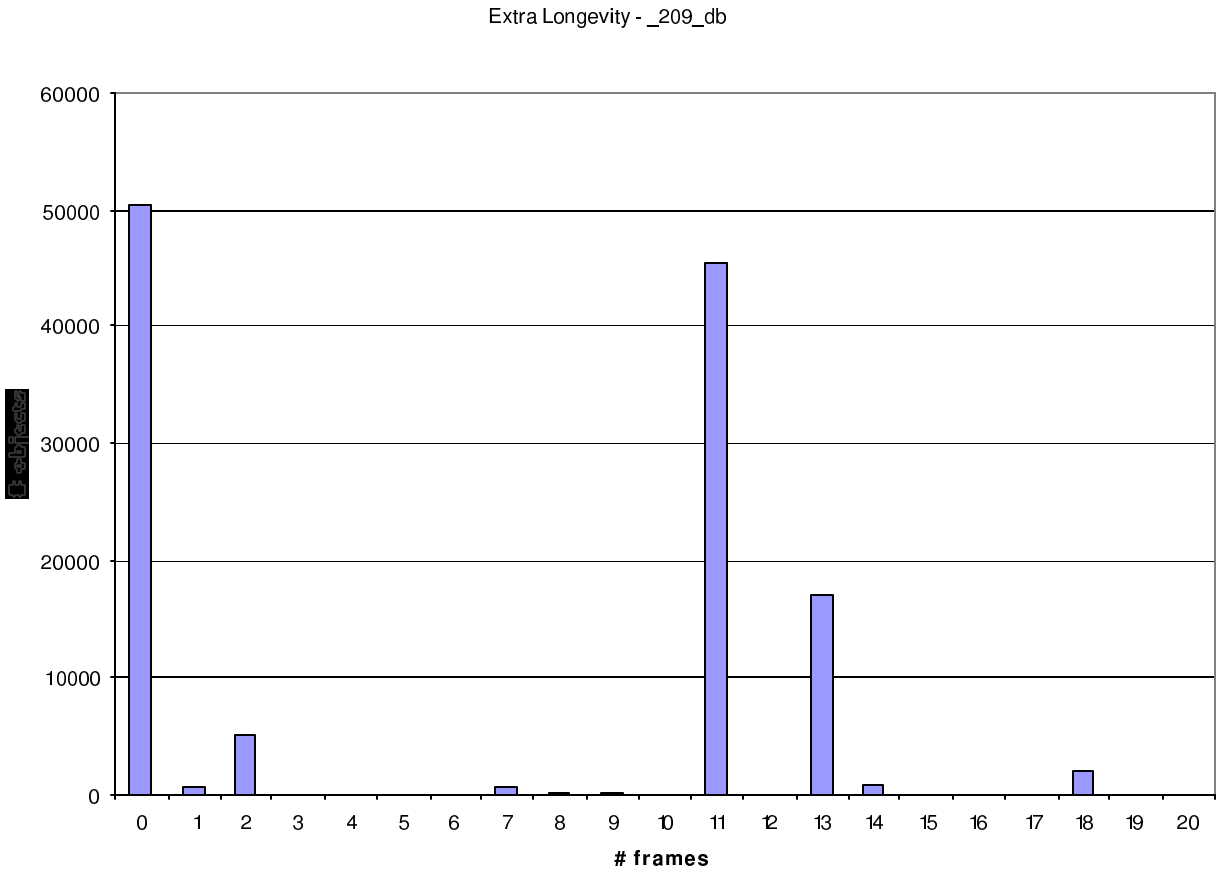
Raytrace extra longevity



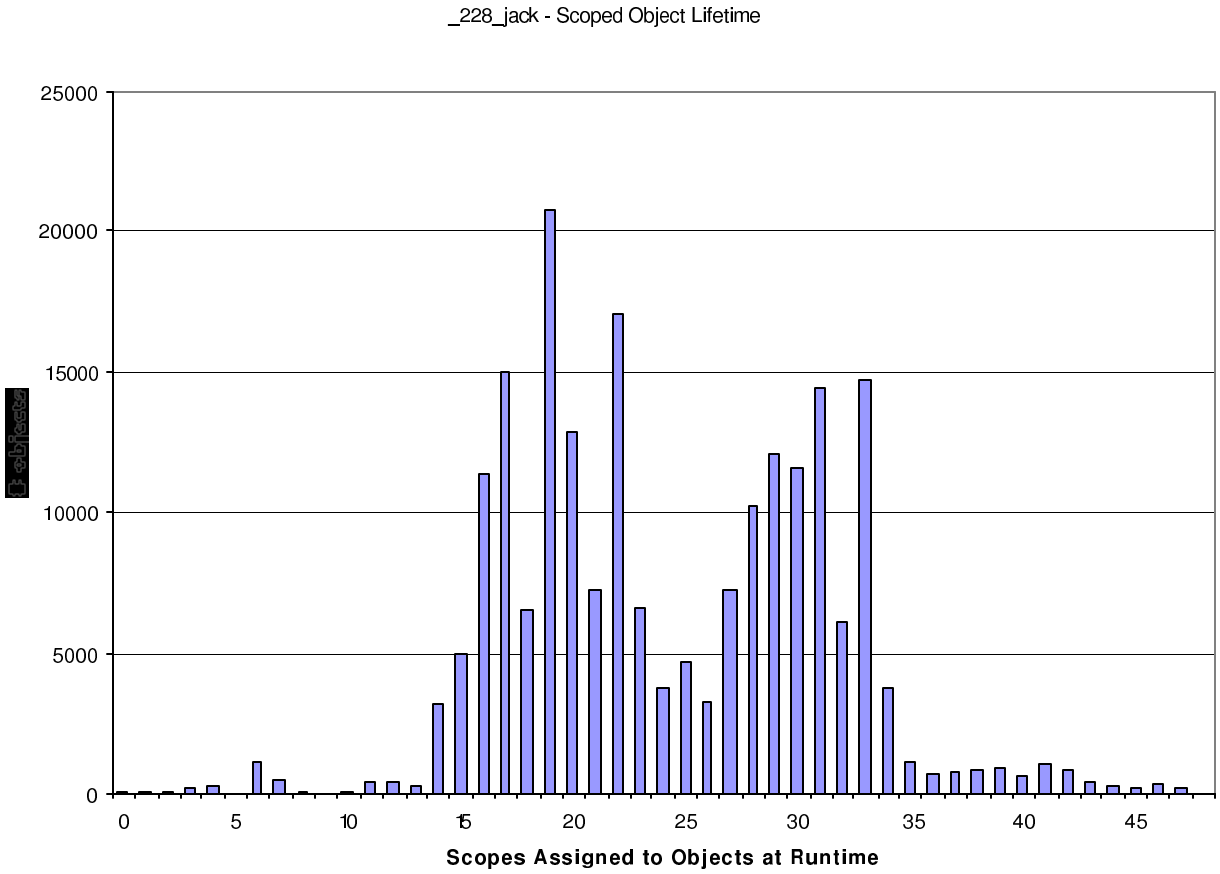
Db scopes assigned



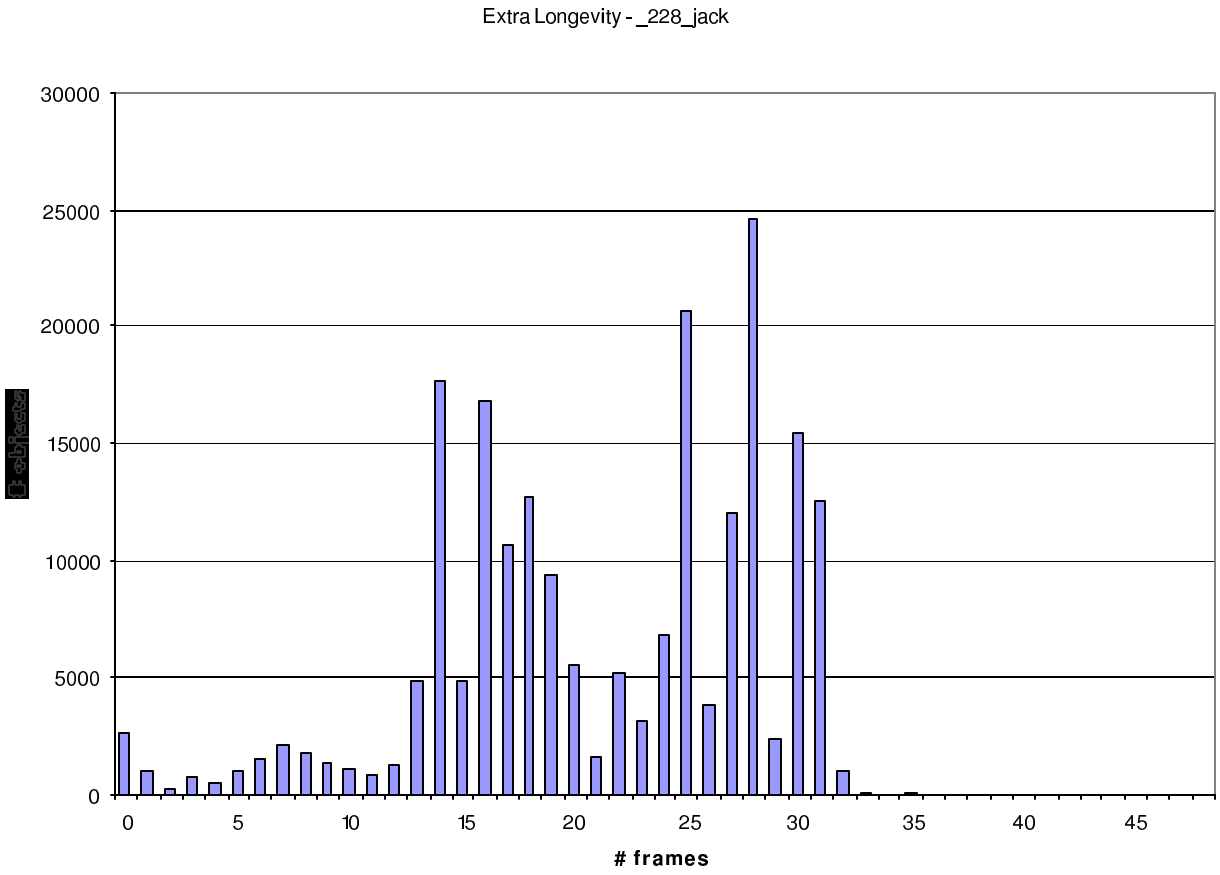
Db extra longevity

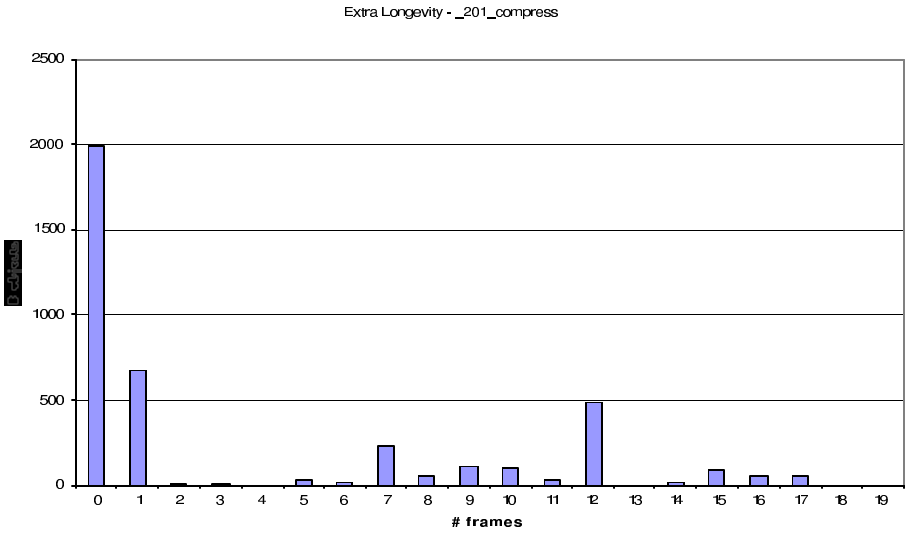


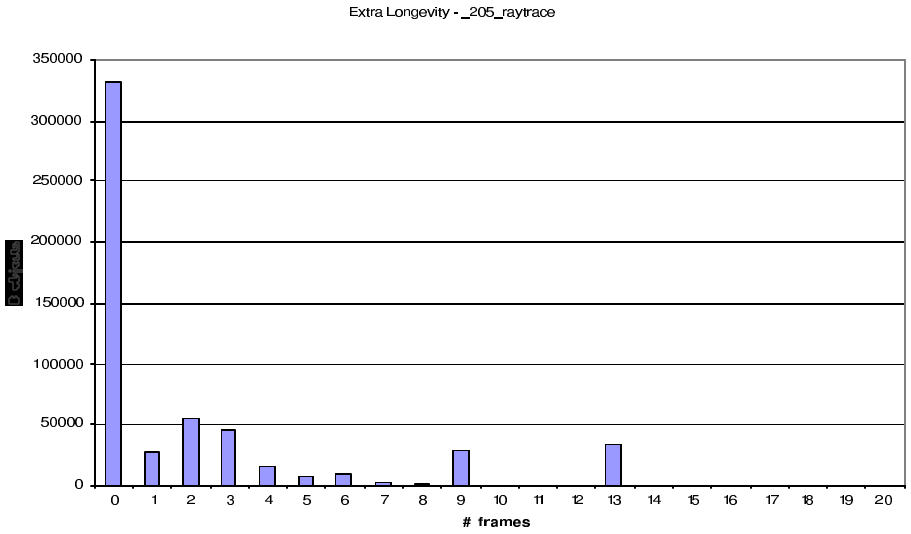
Jack scopes assigned

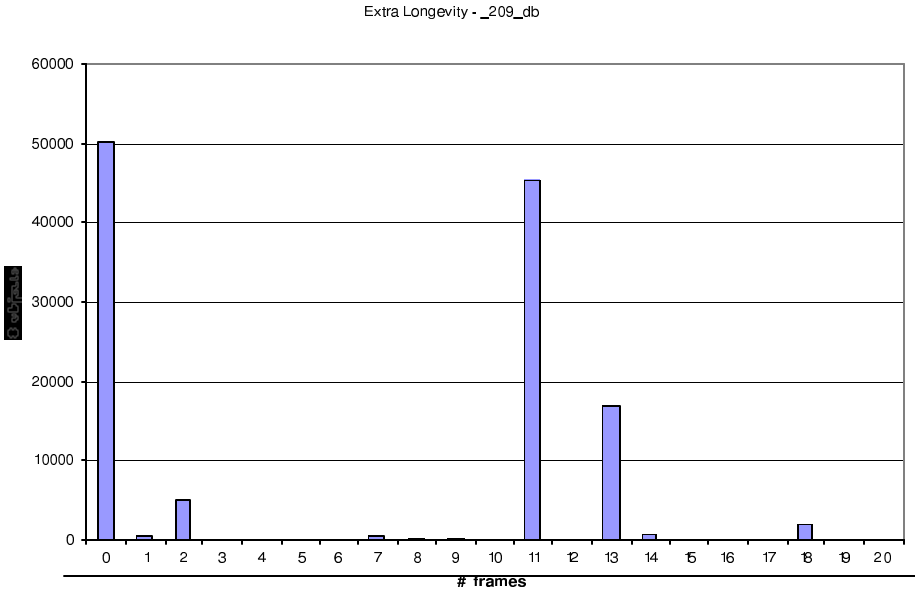


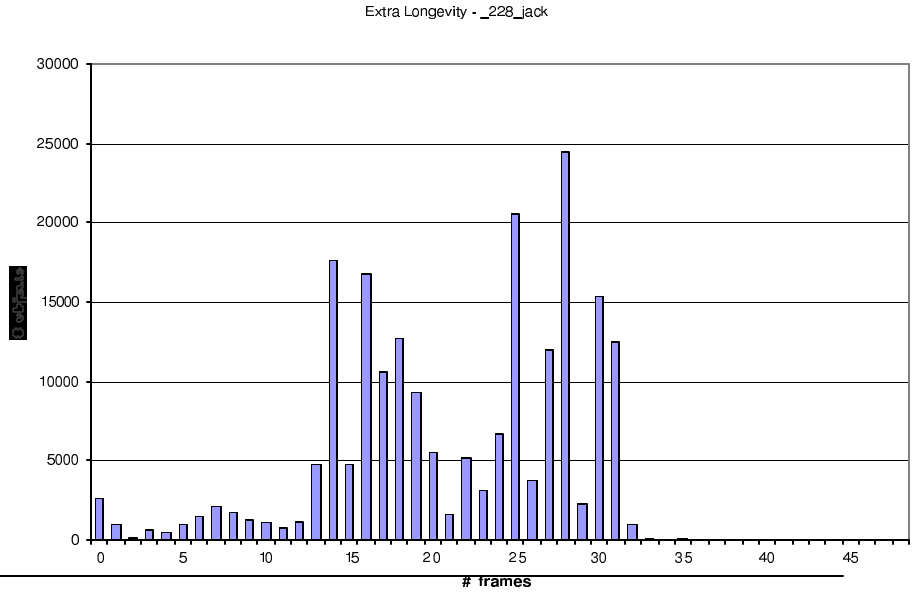
Jack extra longevity




Extra Longevity - _201_compress
2500
2000
1500
1000
500
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
frames


Extra Longevity - _205_raytrace
350000
300000
250000
200000
150000
100000
50000
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
frames


Extra Longevity - _209_db
60000
50000
40000
30000
20000
10000
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
frames


Extra Longevity - _228_jack
30000
25000
20000
15000
10000
5000
0
0 5 10 15 20 25 30 35 40 45
frames

Combining Static and Dynamic Analyses

- Static analysis used for
  - Eliminating RTSJ-mandated reference checks
  - Verifying scope assignments, flagging others for programmer input
- Dynamic analysis used for
  - Initial scope assignment
  - Debugging information

Related Work

- Regions
  - Tofte & Talpin
  - Gay & Aiken
- Real-Time Java analysis
  - Pointer and escape analysis for Java
  - Salcianu & Rinard

Remarks

- Garbage collection is great, but…
- Stack-based region-allocated objects
- RTSJ scoped memory breaks modularity
- Compose with results of static analysis
- Extension to non-RTSJ systems

Contributions

- Use of aspects in dynamic analysis
  - Deters, Leidenfrost, and Cytron. *Translation of Java to Real-Time Java*, International Workshop on Aspect-Oriented Programming and Separation of Concerns, August 2001
- A Real-Time Java translation system, implemented in AspectJ, that requires no program annotation and transforms Java programs into ScopeMemory-aware RT-Java programs modularly
  - Deters and Cytron. *Introduction of Program Instrumentation using Aspects*, OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems, October 2001
- Automation of scope detection and selection
  - Deters and Cytron. *Automated Discovery of Scoped Memory Regions for Real-Time Java*, International Symposium on Memory Management, June 2002

Acknowledgments

- Dr. Ron K. Cytron
- Committee
  - Dr. Chris Gill
  - Dr. Aaron Stump
- JVM hacking
  - Nick Leidenfrost
  - Dante Cannarozzi
  - Matt Hampton
- Members of the DOC Group, past & present

Thanks !

Morgan Deters
mdeters@cs.wustl.edu

www.cs.wustl.edu/~doc

Department of Computer Science
Washington University Box #1045
St. Louis, MO 63130 USA

Copyright © 2003 Morgan Deters