Write Barrier Removal by Static Analysis

Karen Zee
Martin Rinard

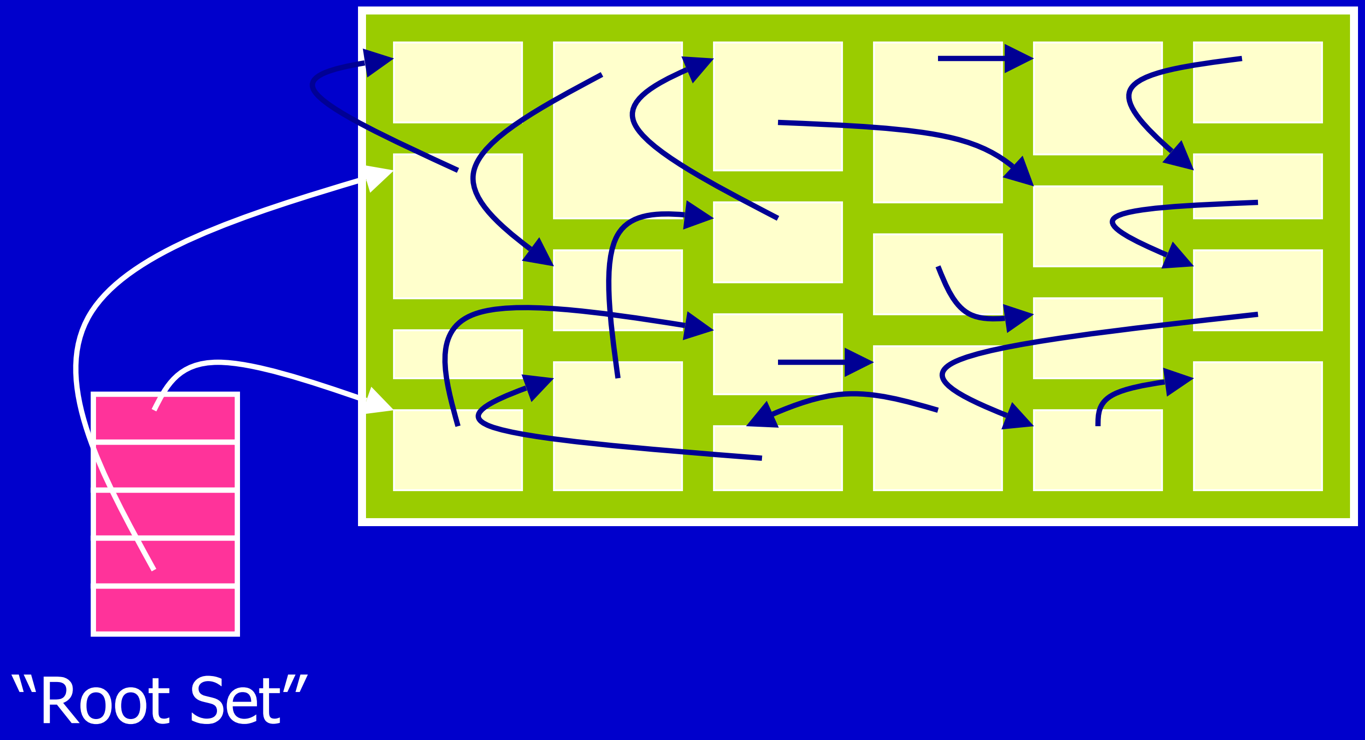
MIT Laboratory for Computer Science

Research Goal

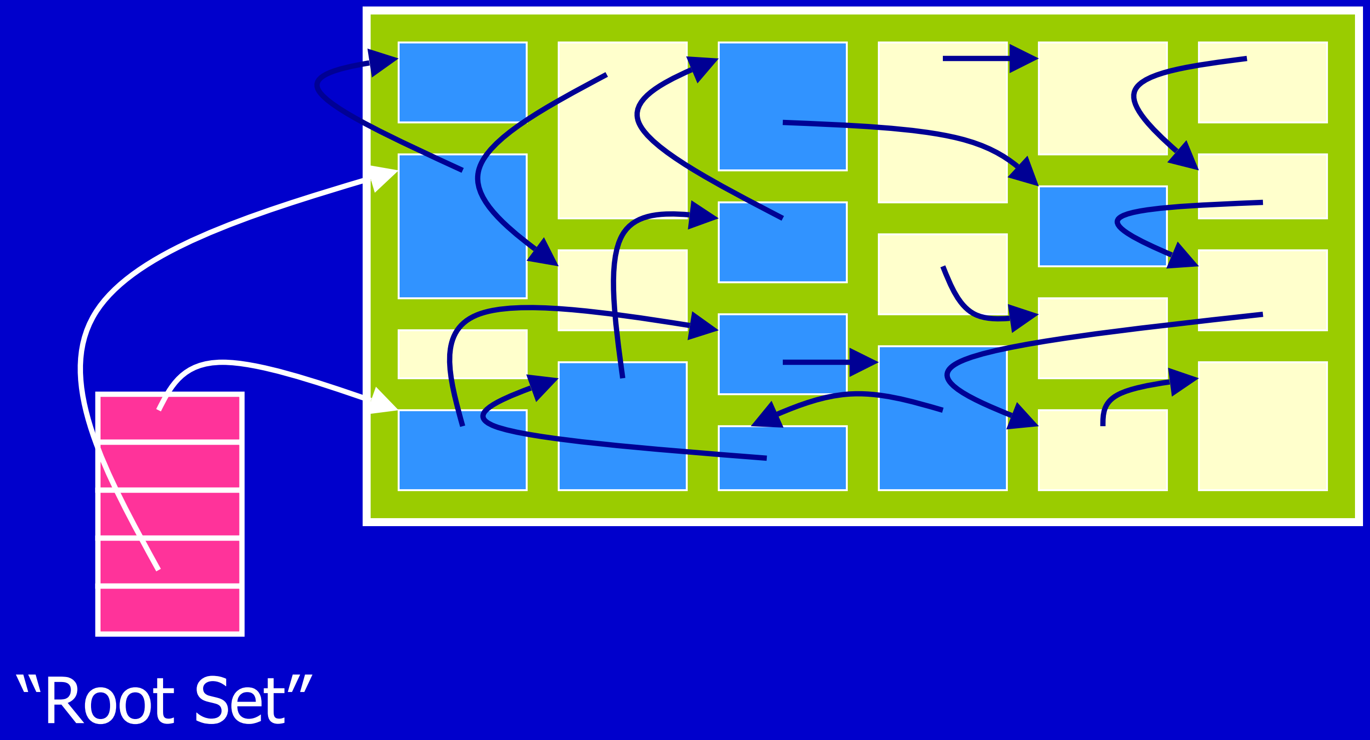
Use static program analysis to identify and remove unnecessary write barriers in programs that use generational garbage collection.

Write Barriers and Generational Garbage Collection

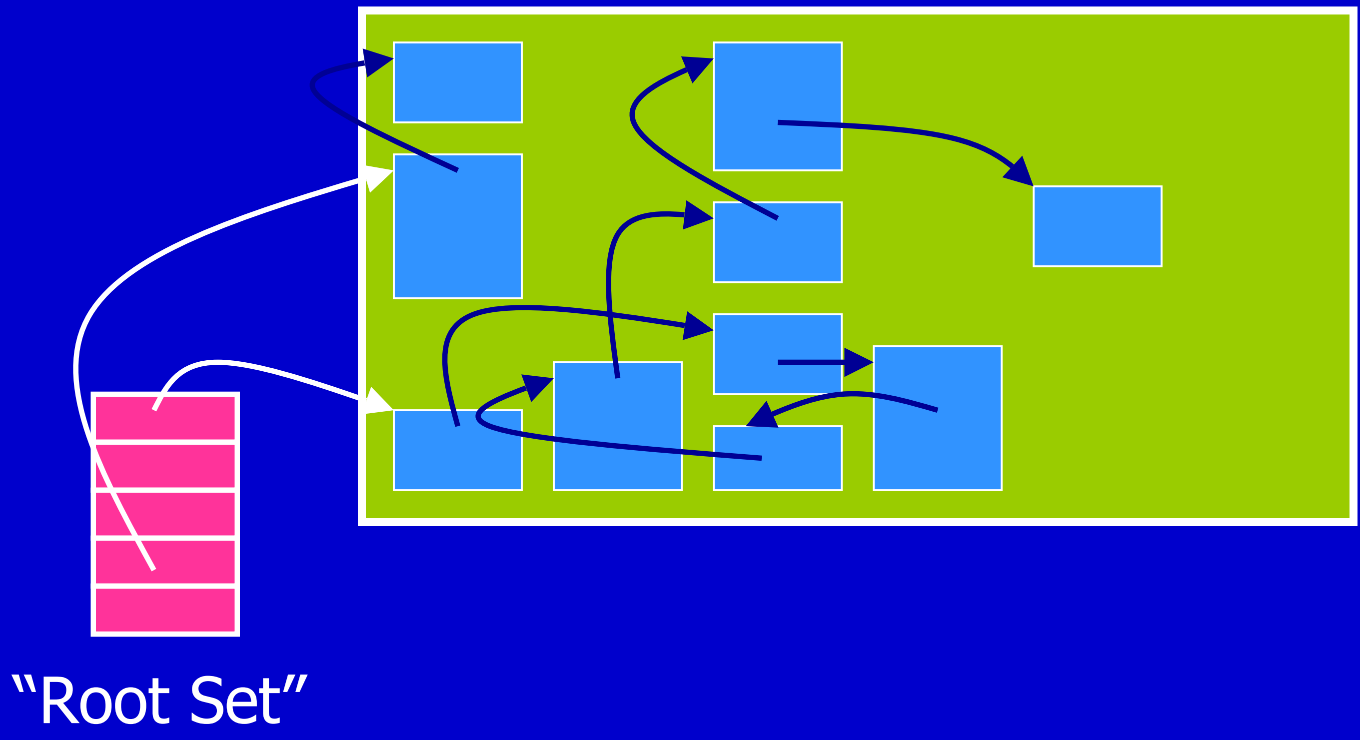
Garbage Collection



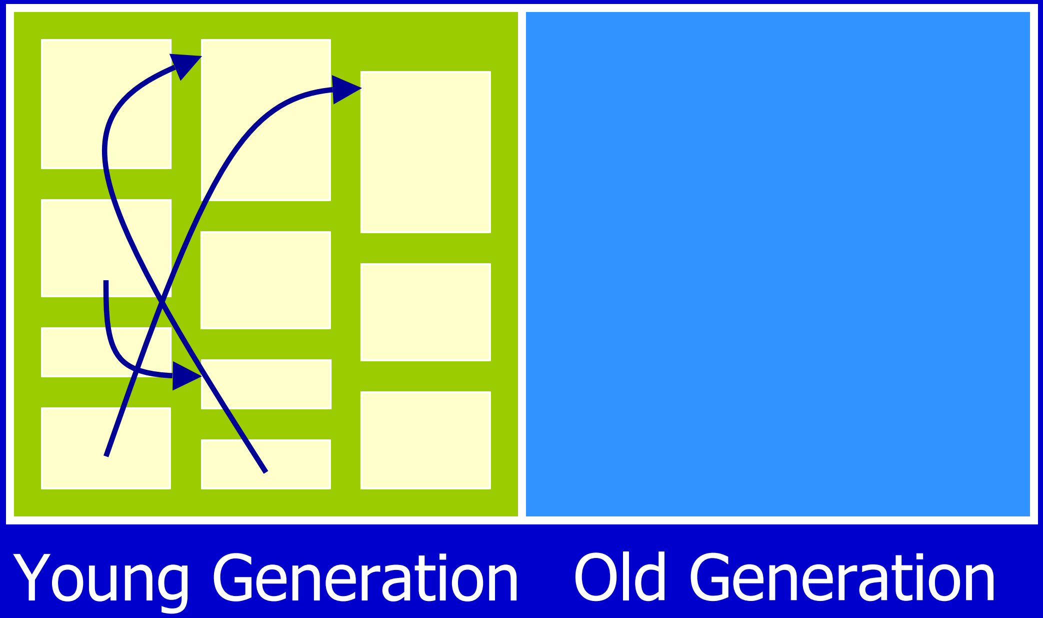
Garbage Collection



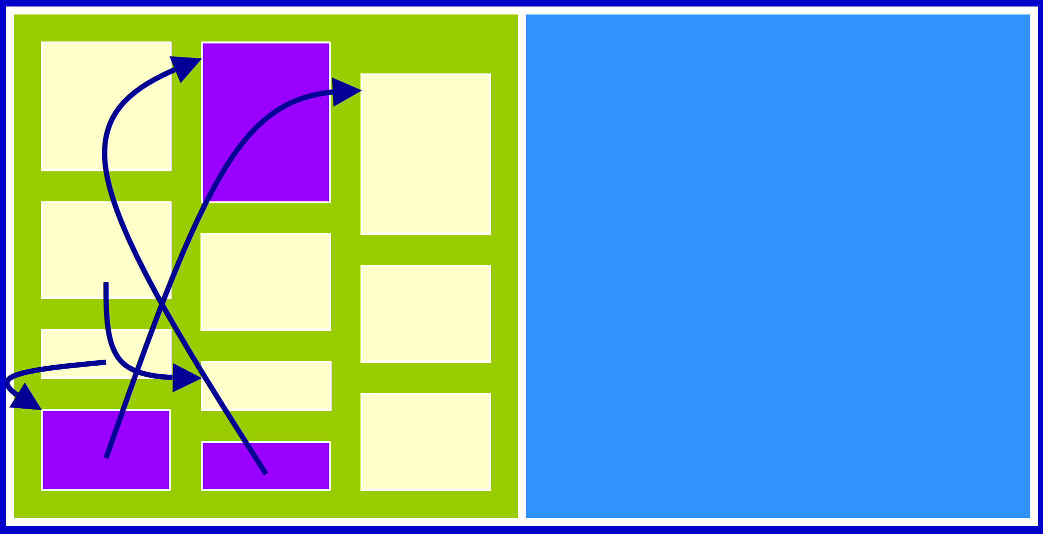
Garbage Collection



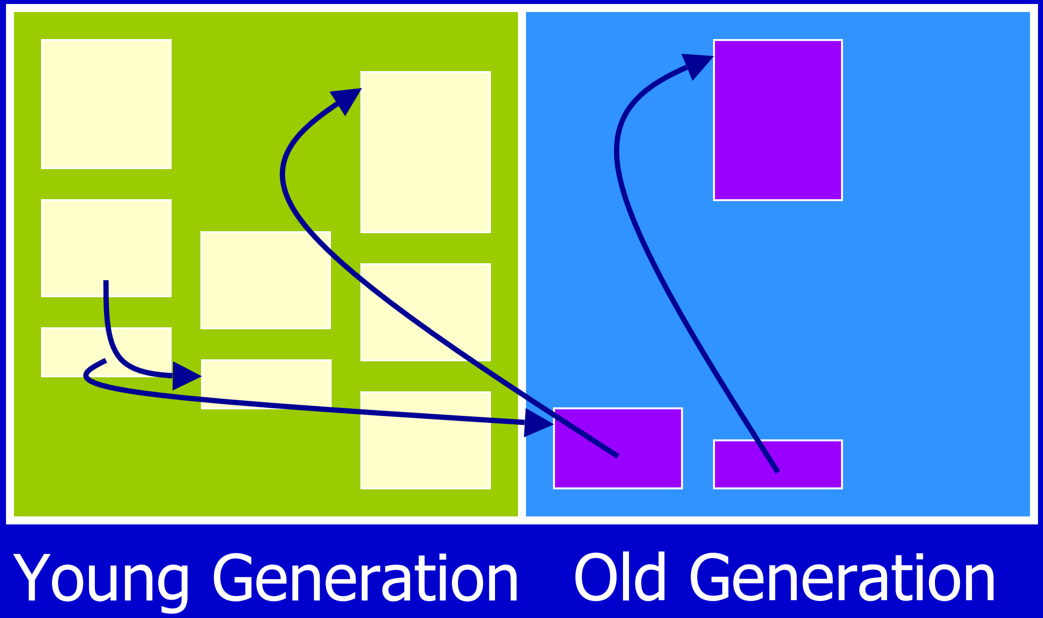
Generational Garbage Collection



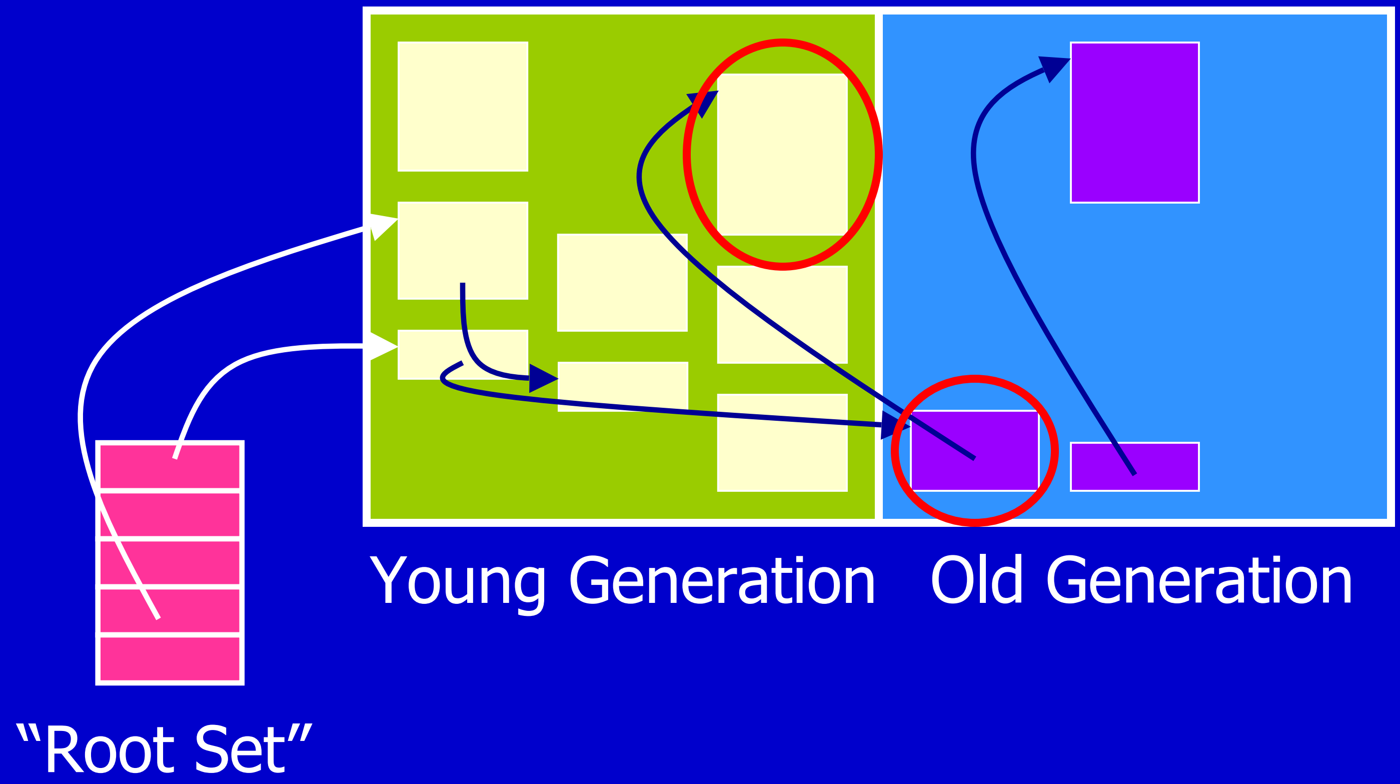
Generational Garbage Collection



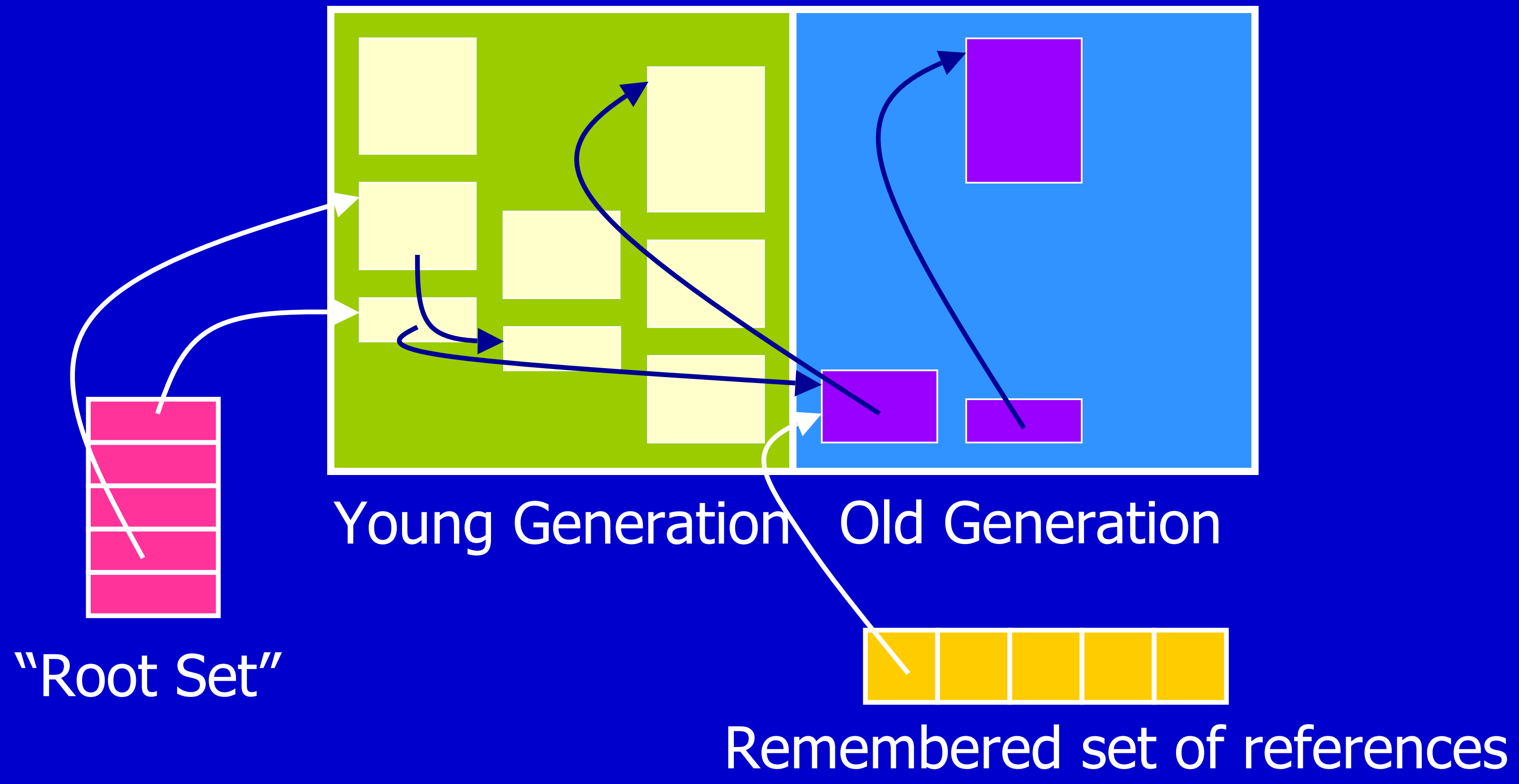
Generational Garbage Collection



Generational Garbage Collection



Generational Garbage Collection



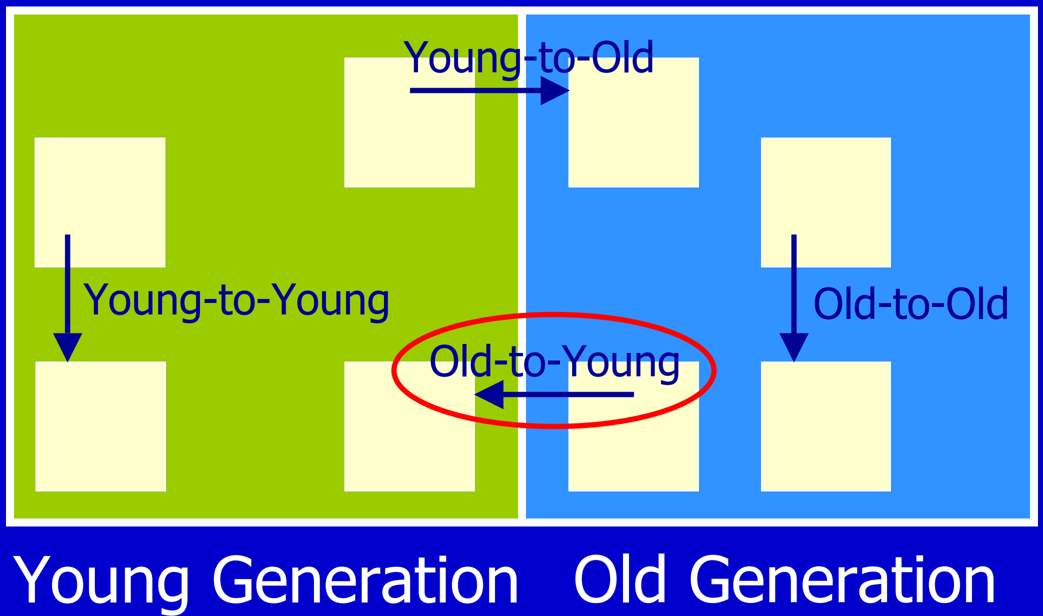
Write Barriers

- Code added by the compiler to every store of a reference into the heap
- Ex. Remembered set of references
  - Array of locations that could contain references into the young generation
  - Maintained at store instructions:

```
v1.f = v2;              // store
rs[i++] = &v1.f;        // write barrier
```

- Undesirable overhead on stores to the heap

Generational Garbage Collection



Write Barrier Removal Principle

- Write barrier unnecessary for stores that create references from younger to older objects
- Analysis approach
  - Identify stores that create references from younger to older objects
  - Specifically, references from the youngest object to other objects
- Then remove the corresponding write barriers

The Analyses

- Track youngest object
- Target Java bytecode
- Intraprocedural
- Interprocedural
  - With callee information
  - With caller information

Intraprocedural Analysis

- Forward dataflow analysis
- Computes must points-to information
- For each point in the program, the analysis generates
  - A set V of local variables
  - All $v \in V$ refer to the youngest object

Using the Analysis Information

- Store statement v1.f = v2
- Write barrier for the store statement can be removed if
  - v1 refers to the youngest object
  - i.e. v1 $\in$ V

Transfer Functions

stm	[stm](V)
lv = NEW C	{ lv }
lv1 = lv2	V $\cup$ { lv1 } if lv2 $\in$ V V – { lv1 } if lv2 $\notin$ V
lv = CALL m(…)	$\emptyset$
lv = …	V – { lv }
…	V

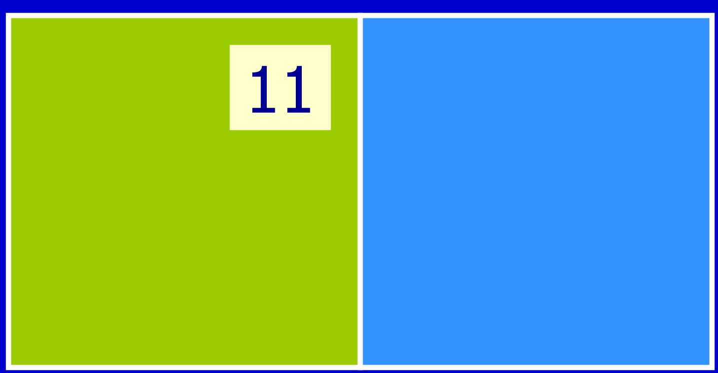
Dataflow Framework

- Join operator is set intersection $\cap$
- Initial value of V
  - $\emptyset$ at method entry
  - All program variables *Vars* at all other program points

Point Example

```
class Point {
    Integer x;
    Integer y;
}
```

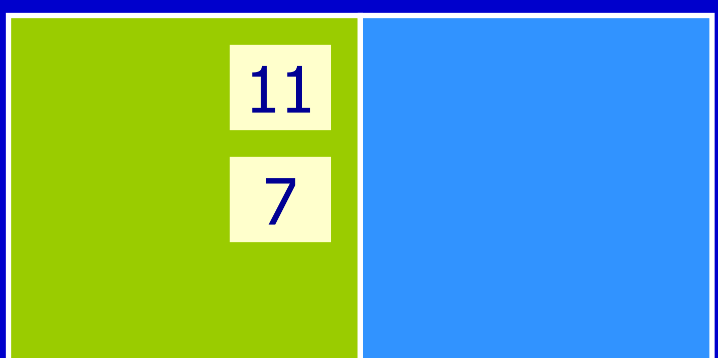
```
…
x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;
p.y = y;
…
```



Point Example

```
class Point {
    Integer x;
    Integer y;
}
```

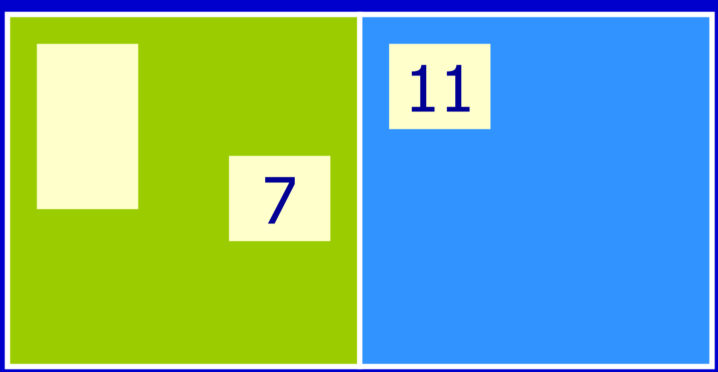
```
…
x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;
p.y = y;
…
```



Point Example

```
class Point {
    Integer x;
    Integer y;
}
```

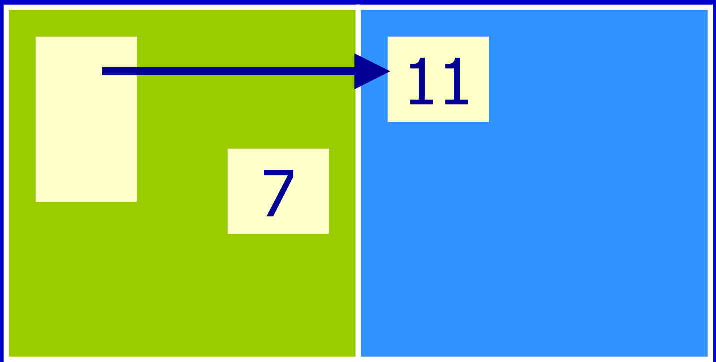
```
…
x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;
p.y = y;
…
```



Point Example

```
class Point {
    Integer x;
    Integer y;
}
```

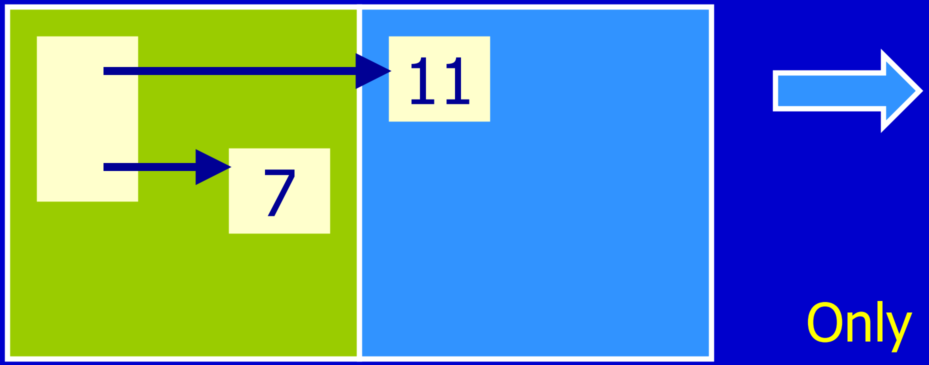
```
...
x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;
p.y = y;
...
```



Point Example

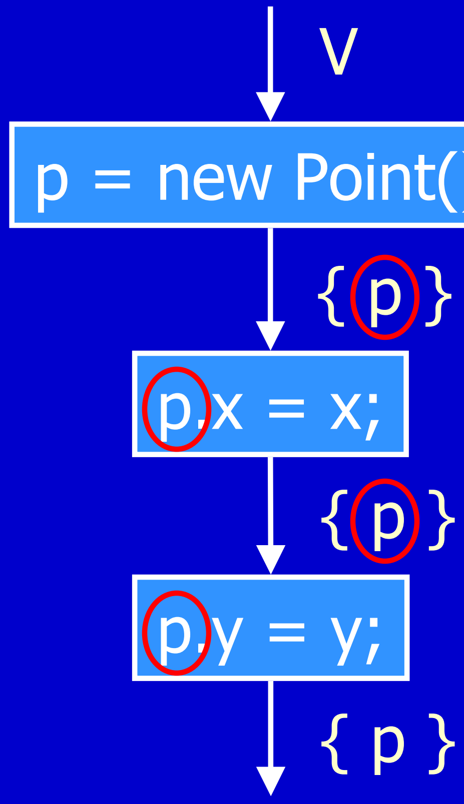
```
class Point {
    Integer x;
    Integer y;
}
```

```
…
x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;
p.y = y;
…
```



Only young-to-old references created.
Write barriers should be unnecessary!

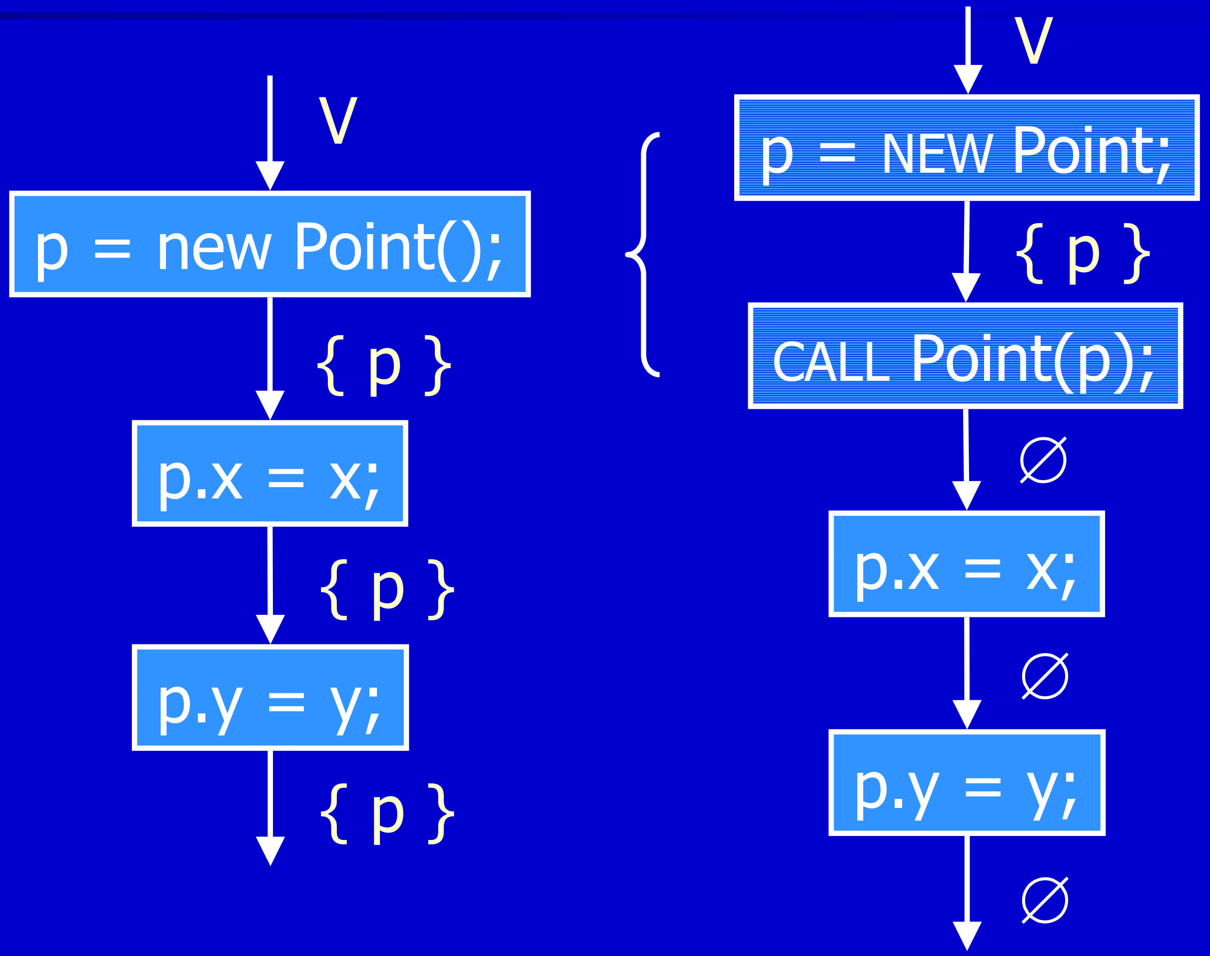
Point Example




```
class Point {
    Integer x;
    Integer y;
}

…
x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;  // write barrier not needed
p.y = y;  // write barrier not needed
…
```

Point Example



Interprocedural Information

Allocated Object Types

- Consider the method-youngest object
- Track the types of objects that invoked methods may allocate
- Use type-based reasoning to remove write barriers associated with stores that
  - Create a reference from the youngest object allocated in the current method
  - Provided that reference does not point to an object allocated by an invoked method

Allocated Type Analysis

- Flow-insensitive interprocedural analysis
- Collects types of objects allocated either
  - Directly by the given method, or
  - Indirectly by (transitively) invoked methods

Information from Callees

- For each point in the program, the analysis generates a pair ⟨ V, T ⟩
  - Set V of local variables
  - All v ∈ V refer to the method-youngest object
  - Set T of types
  - Objects of type t ∈ T may have been allocated by (transitively) invoked methods

Using the Analysis Information

- Store statement v1.f = v2
- Write barrier for the store statement can be removed if
  - v1 refers to the method-youngest object
  - i.e. $v1 \in V$
  - v2 refers to an object older than the method-youngest object
  - i.e. $subtypes(C) \cap T = \emptyset$ where C is the static type of v2

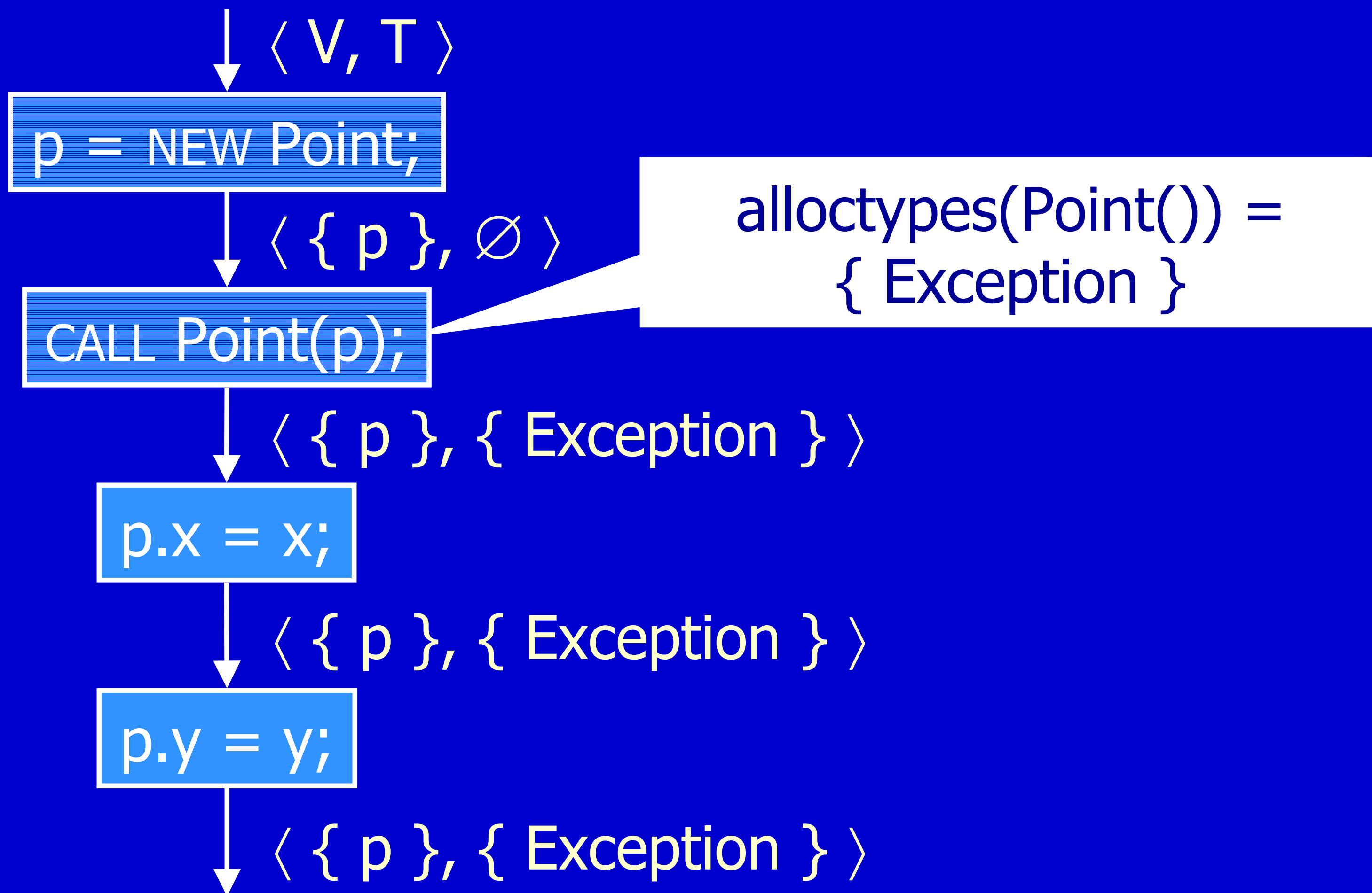
Modified Transfer Functions

st	[st]($\langle$ V, T $\rangle$)
lv = NEW C	$\langle \{ lv \}, \emptyset \rangle$
lv1 = lv2	$\langle V \cup \{ lv1 \}, T \rangle$ if lv2 $\in$ V $\langle V - \{ lv1 \}, T \rangle$ if lv2 $\notin$ V
lv = CALL m(…)	$\langle V', T' \rangle$ where $V' = V - \{ lv1 \}$ and $T' = T \cup (\cup_{m' \in callees(m)} alloctypes(m'))$
lv = …	$\langle V - \{ lv \}, T \rangle$
…	$\langle V, T \rangle$

Dataflow Framework

- Join operator is $\langle \cap, \cup \rangle$
- Initial value of $\langle$ V, T $\rangle$
  - $\langle \emptyset$, *Types* $\rangle$ at method entry
  - $\langle$ *Vars*, $\emptyset \rangle$ at all other program points

Point Example

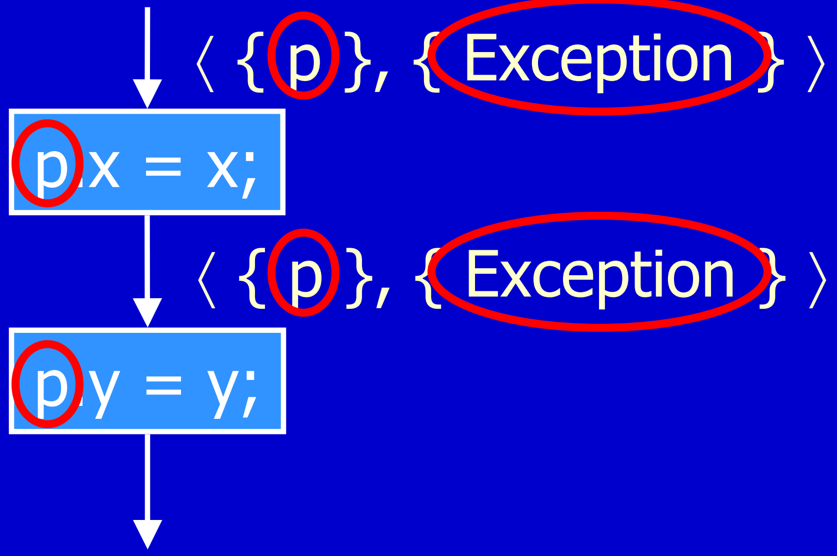


Point Example

Remove write barrier for v1.f = v2 if:

- v1 $\in$ V, and
- subtypes(C) $\cap$ T = $\emptyset$

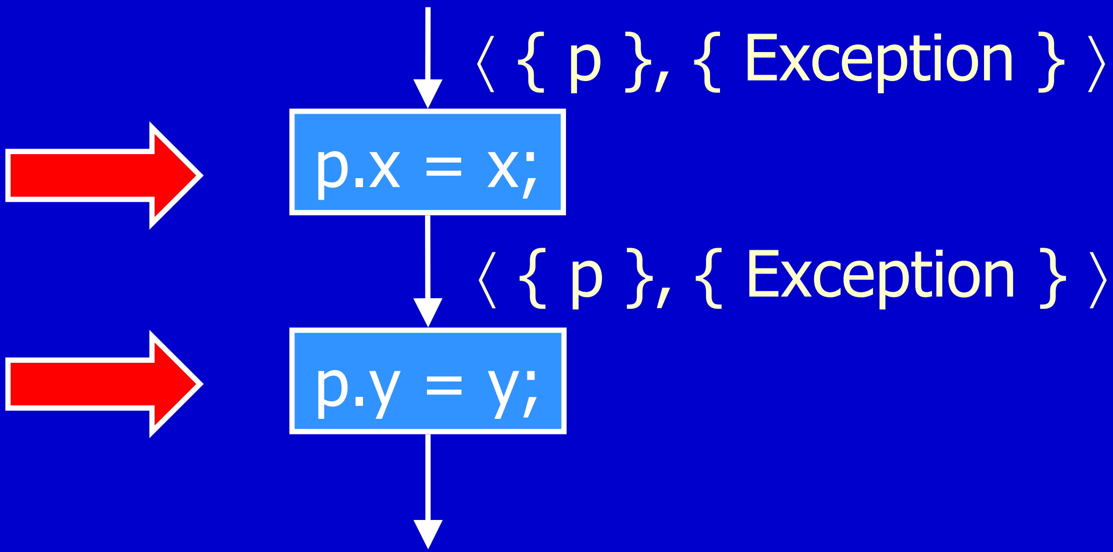
where C is the static type of v2



- Type(x) = Integer
- Type(y) = Integer
- subtypes(Integer) $\cap$ { Exception } = $\emptyset$

Point Example

Can remove write barriers!



- Type(x) = Integer
- Type(y) = Integer
- subtypes(Integer) $\cap$ { Exception } = $\emptyset$

Point Example

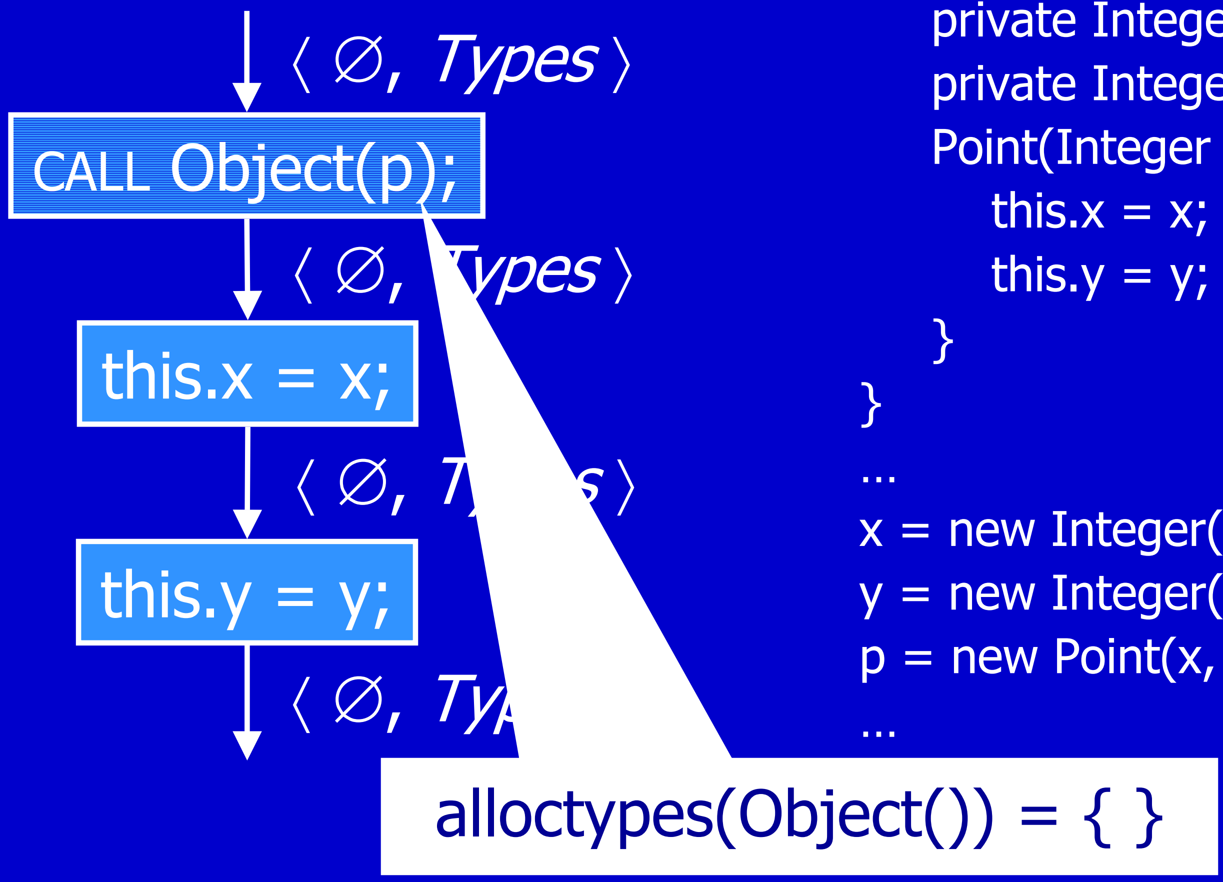
```
class Point {
    Integer x;
    Integer y;
}
…
x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x; // no write barrier
p.y = y; // no write barrier
…
```

Point Example

```
class Point {
    private Integer x;
    private Integer y;
}
…
x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x; // no write barrier
p.y = y; // no write barrier
…
```

```
class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x; // write barrier?
        this.y = y; // write barrier?
    }
}
…
x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);
…
```

Point Constructor Example

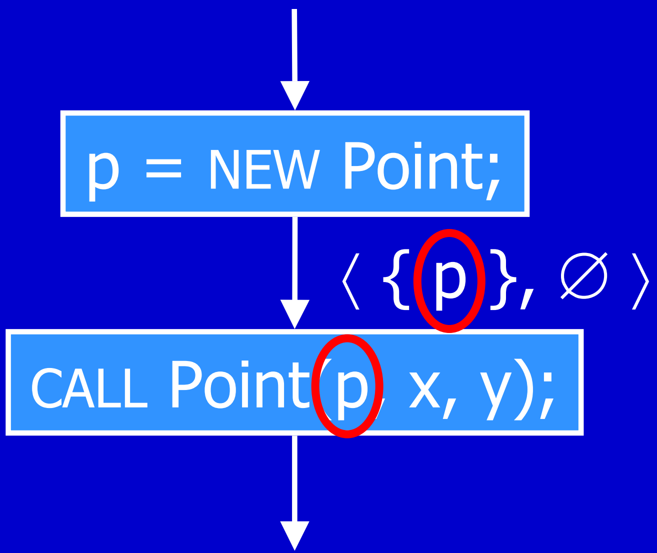



```
class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x; // write barrier?
        this.y = y; // write barrier?
    }
}
…
x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);
…
```

Information from Callers

- Use information from callers
- At call sites where the youngest object is an argument to the call, propagate analysis results to entry point of invoked method

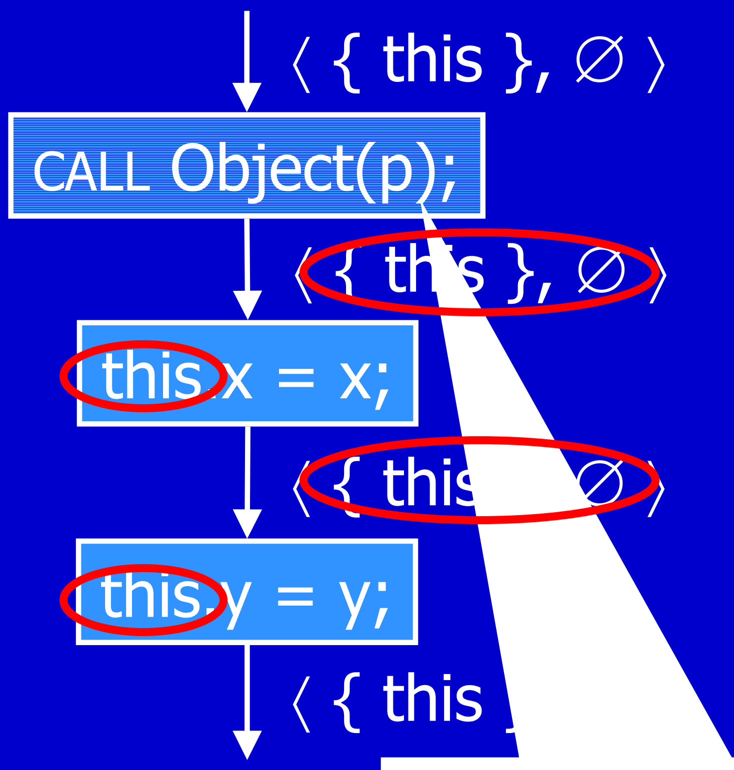
Point Constructor Example



The first argument to the Point constructor is the youngest object.

```
class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x; // write barrier?
        this.y = y; // write barrier?
    }
}
…
x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);
…
```

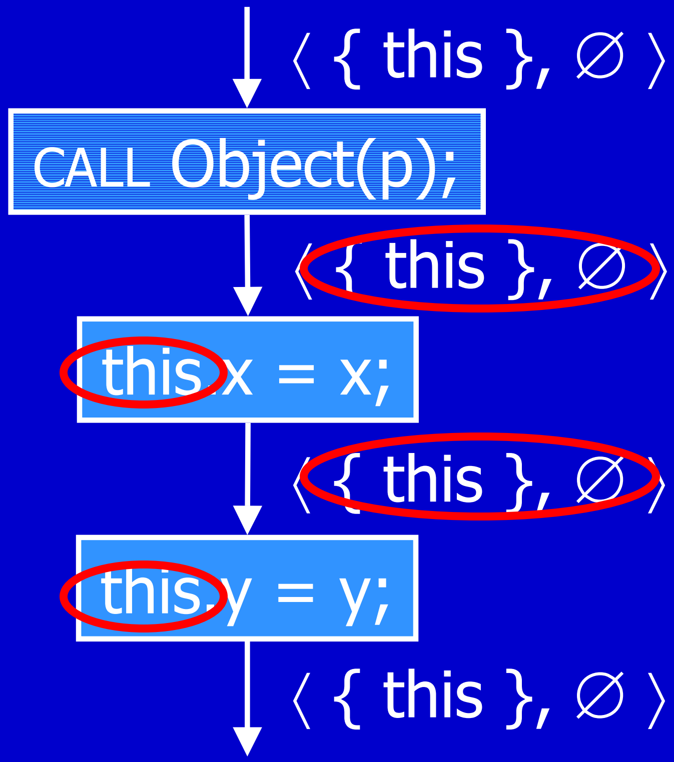
Point Constructor Example



alloctypes(Object()) = ∅

```
class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x; // write barrier?
        this.y = y; // write barrier?
    }
}
…
x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);
…
```

Point Constructor Example




```
class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x; // no write barrier!
        this.y = y; // no write barrier!
    }
}
…
x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);
…
```

Point Constructor Example

```
class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x; // no write barrier
        this.y = y; // no write barrier
    }
}
…
x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);
…
```

```
class Point {
    private Integer x;
    private Integer y;
    Point(int x, int y) {
        this.x = new Integer(x);
        this.y = new Integer(y);
    }
}
…
p = new Point(11, 7);
…
```

Clearly the receiver object is older than the Integer objects!

The write barrier for the field assignment cannot be removed.

Solution: Transform program to reverse the order of the allocations!

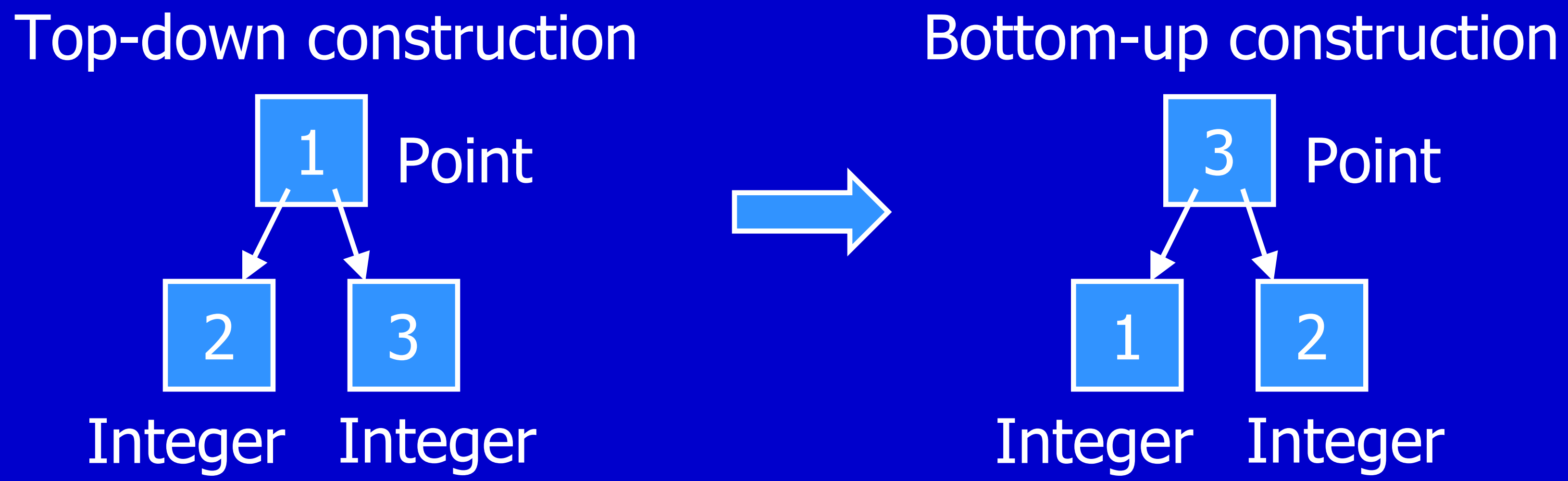
Allocation Order Transformations

```
class Point {
    private Integer x;
    private Integer y;
    Point(int x, int y) {
        tmp0 = NEW Integer;
        CALL Integer(tmp0, x);
        this.x = tmp0;
        tmp1 = NEW Integer;
        CALL Integer(tmp1, y);
        this.y = tmp1;
    }
}
...
p = new Point(11, 7);
...
```

```
class Point {
    private Integer x;
    private Integer y;
    Point(int x, int y, Integer tmp0, Integer tmp1)
    {
        CALL Integer(tmp0, x);
        this.x = tmp0;
        CALL Integer(tmp1, y);
        this.y = tmp1;
    }
}
...
tmp0 = NEW Integer;
tmp1 = NEW Integer;
p = new Point(11, 7, tmp0, tmp1);
...
```

Now the write barriers for the field assignments can be removed!

Allocation Order Transformations



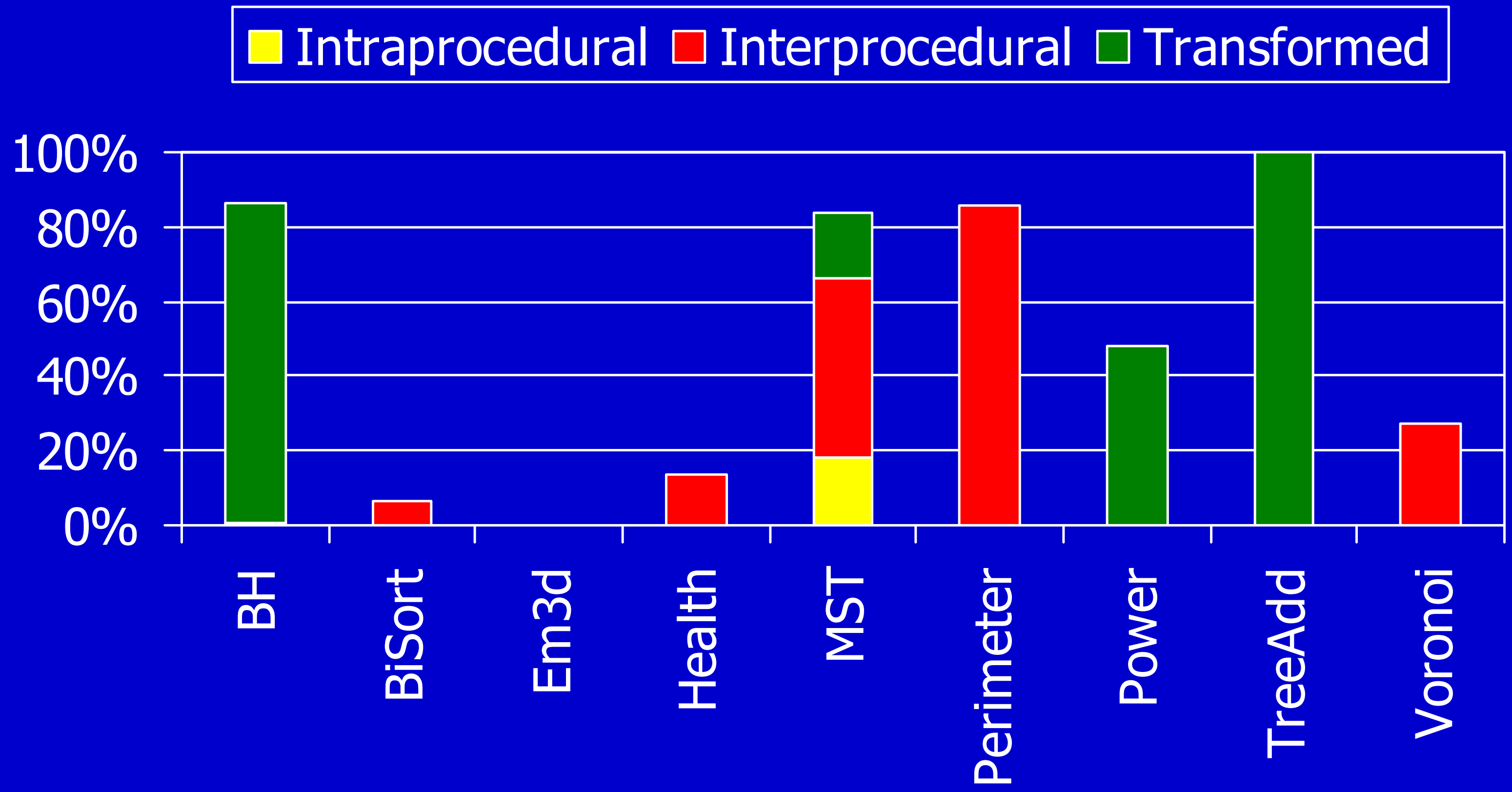
- Make write barriers removable that previously could not be removed
- Handles recursive constructors for data structures such as trees

Experimental Results

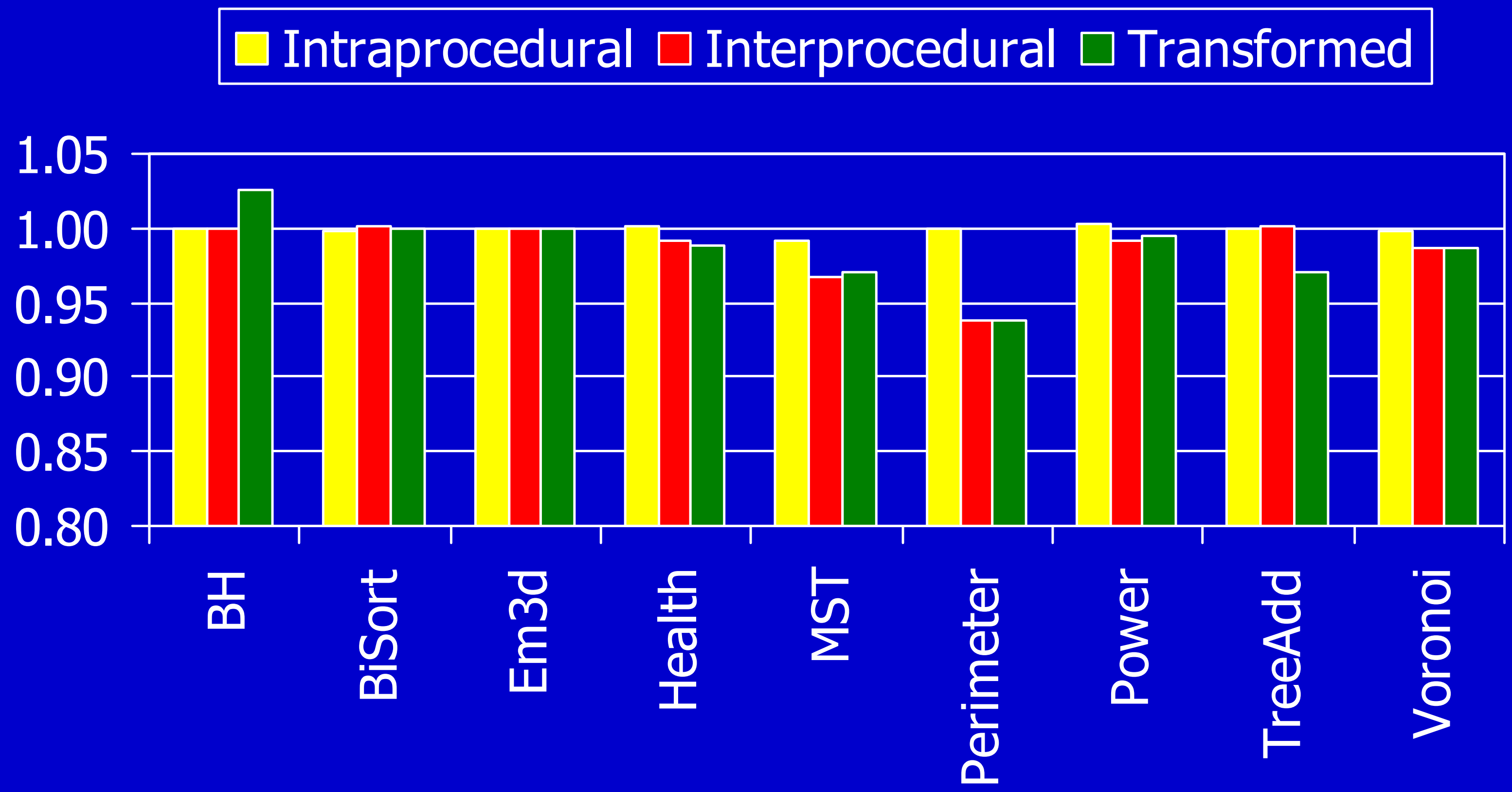
Methodology

- Implemented in Flex compiler infrastructure
  - Ahead-of-time Java bytecode compiler
  - Generates native code or C
- Measured
  - Write barrier counts
  - Execution time
  - Write barrier characterization
  - Analysis time

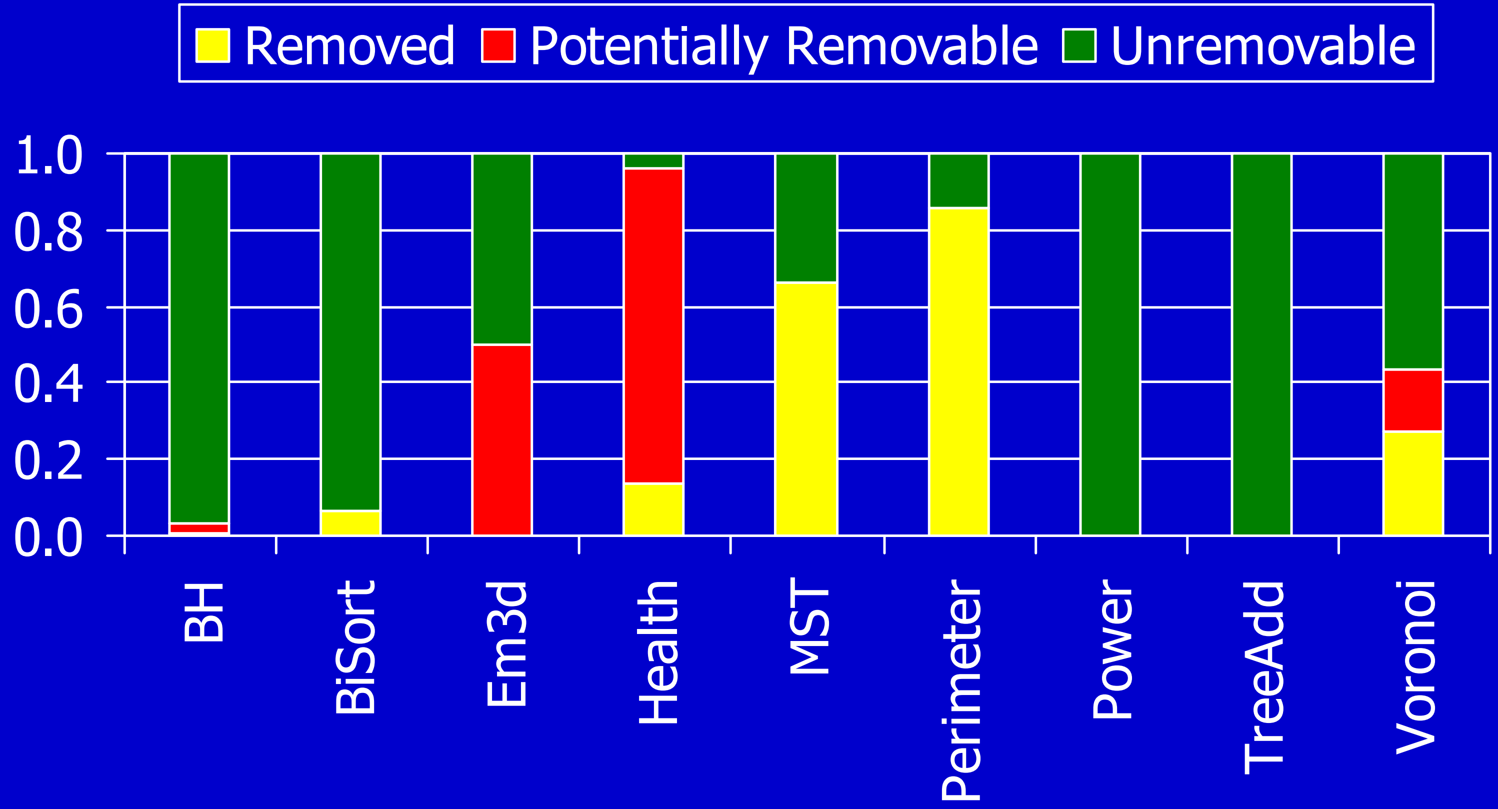
Dynamic Write Barriers Removed



Normalized Execution Time

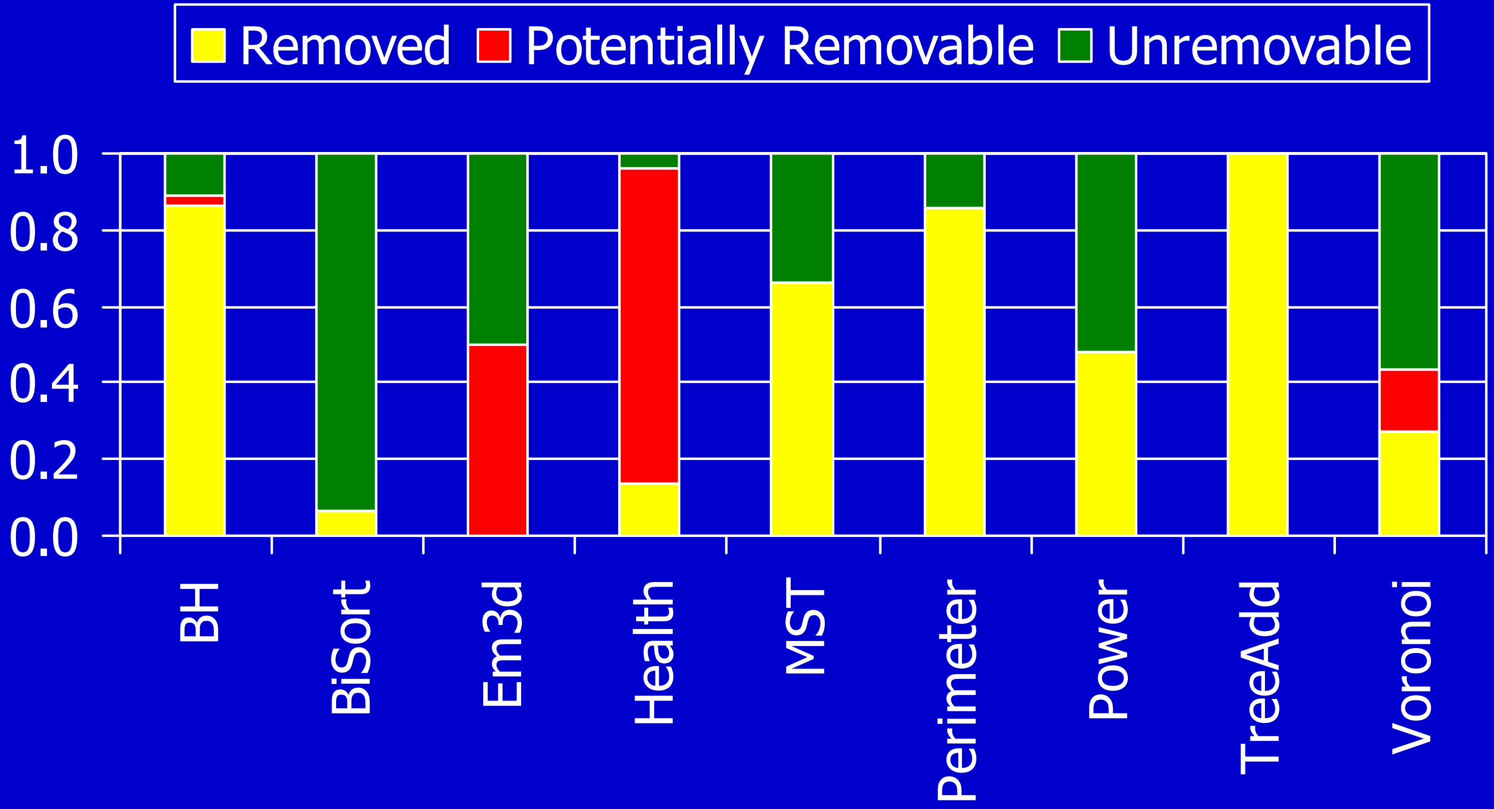


Write Barrier Characterization



Write Barrier Characterization

with Allocation Order Transformations



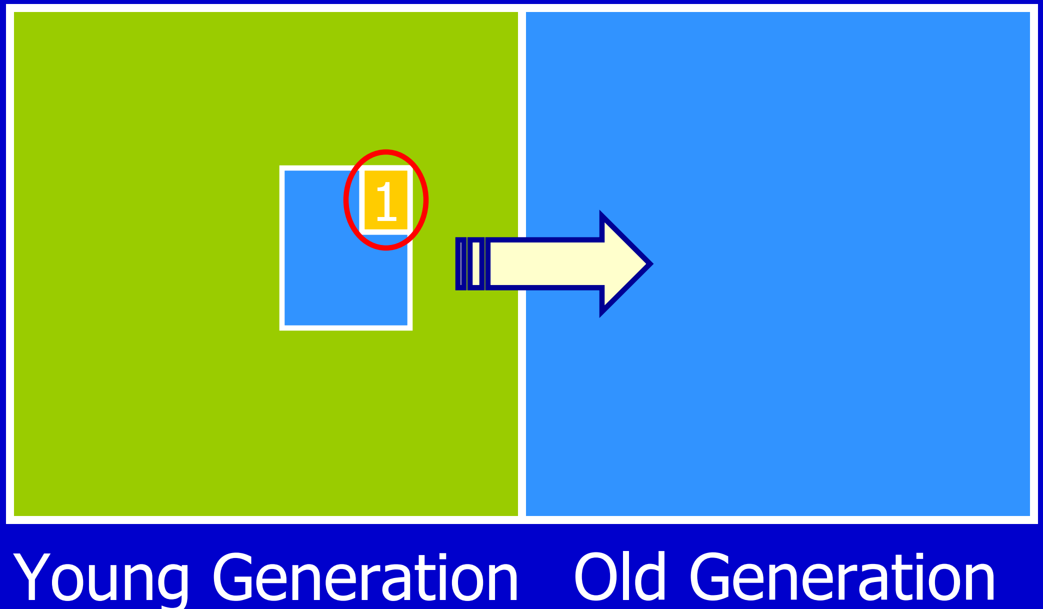
Analysis Time

	Intraprocedural (s)	Interprocedural (s)
BH	15	38
BiSort	16	34
Em3d	16	38
Health	16	38
MST	16	35
Perimeter	15	35
Power	15	38
TreeAdd	12	37
Voronoi	14	42

Other Collectors

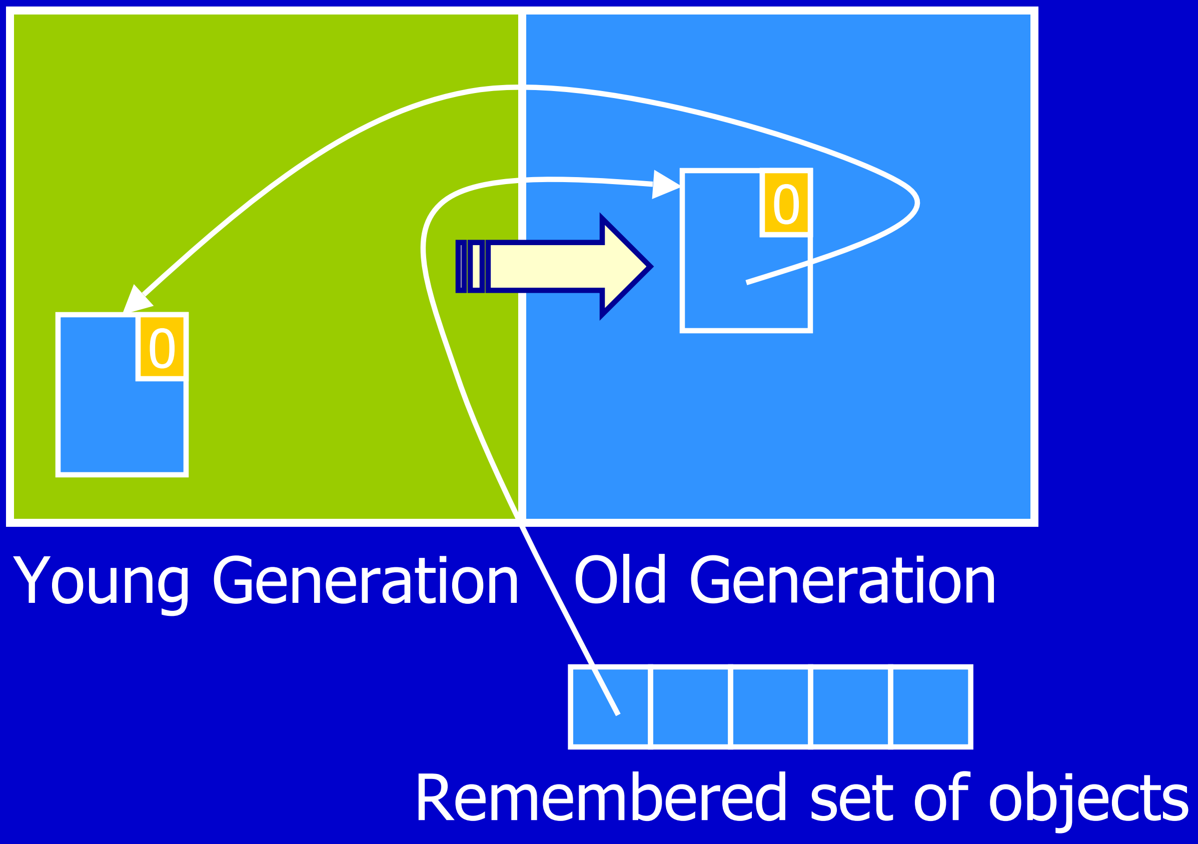
- Analysis assumptions
  - Objects initially allocated in young generation
  - Older objects promoted before younger
- Concurrent collectors may not promote objects in age-order
- Other collectors may not always allocate objects initially in the young generation

An Optimistic Extension

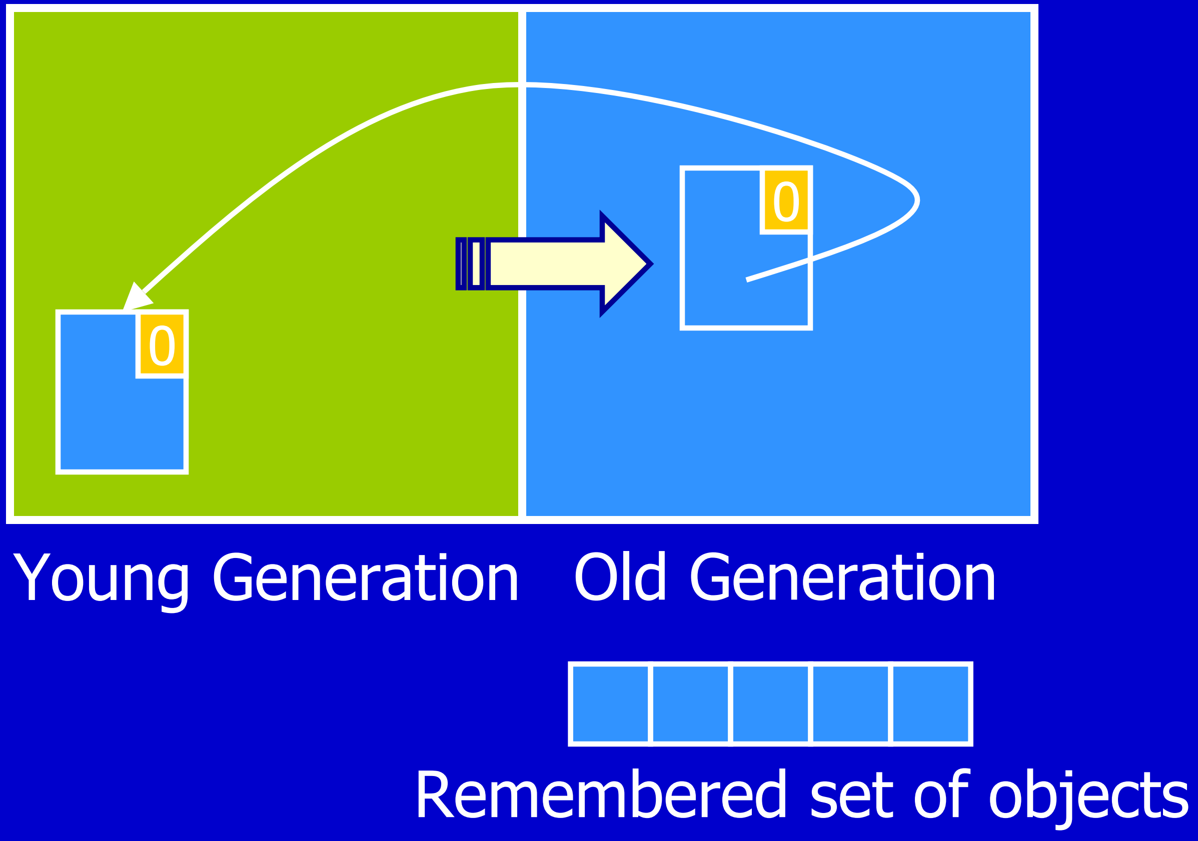


Reserve a bit in the object header

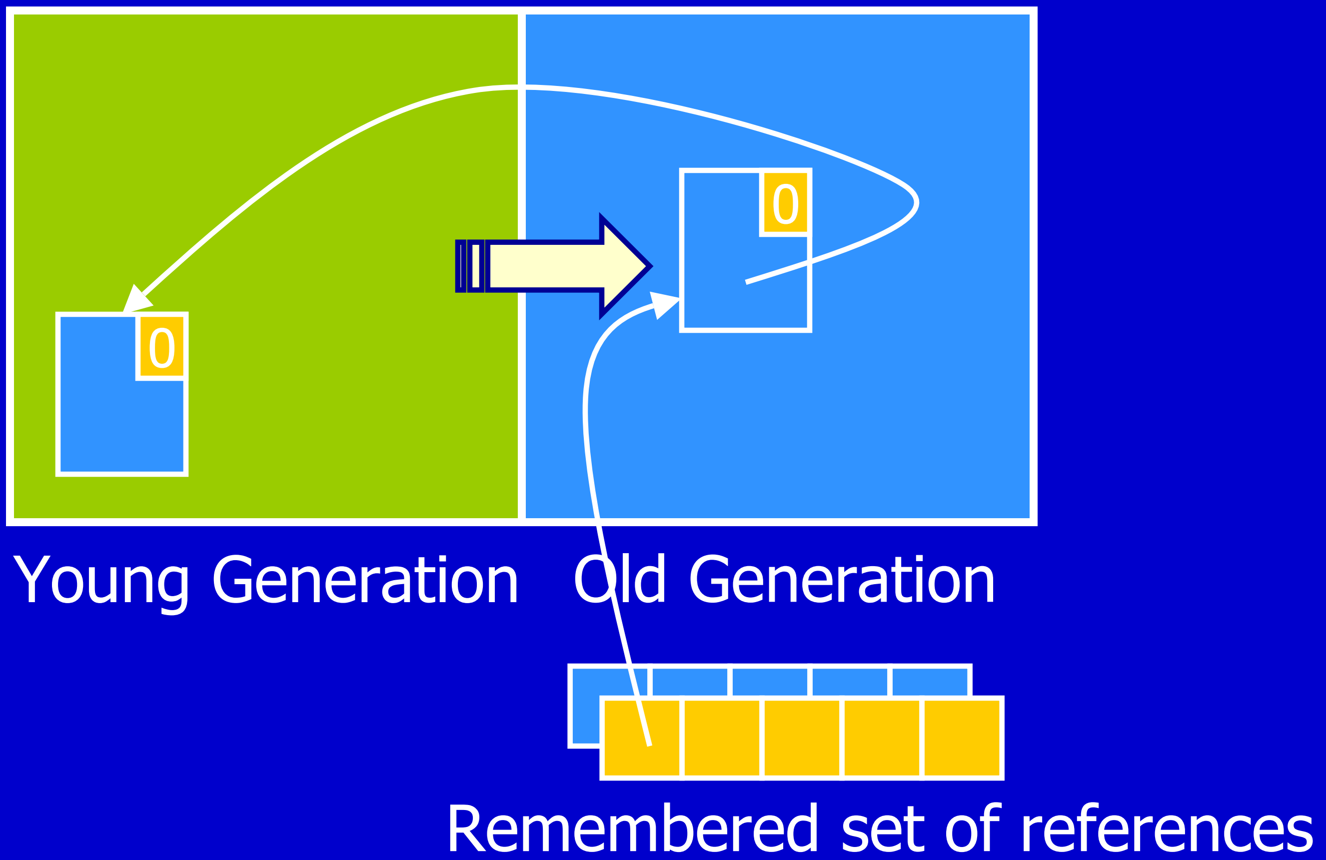
An Optimistic Extension



An Optimistic Extension



An Optimistic Extension



An Optimistic Extension

- Out-of-order promotion
- Allocation in other generations
- Multi-threaded programs
- Low overhead

Overhead of Optimistic Extension

Benchmark	% Increase in Execution Time	
	Without Allocation Order Transformations	With Allocation Order Transformations
BH	0.36%	-0.95%
BiSort	-0.33%	-0.05%
Em3d	0.02%	0.17%
Health	-0.08%	0.37%
MST	0.55%	0.19%
Perimeter	0.37%	0.37%
Power	0.23%	-0.12%
TreeAdd	0.06%	0.22%
Voronoi	0.09%	-0.83%

Related Work

Write Barriers for Generational GC

- Lieberman and Hewitt, 1983 – Entry table
- Ungar, 1984 – Remembered set of objects
- Moon, 1984 – Hardware write barriers
- Shaw, 1988 – Virtual memory write barriers
- Sobalvarro, 1988 – Card and word marking
- Appel, 1989 – Remembered set of addresses
- Wilson and Moher, 1989 – Card marking
- Hudson and Diwan, 1990 – Hashed remembered set
- Zorn, 1990 – Write-protected pages
- Chambers, 1992 – 3 SPARC instructions
- Hölzle, 1993 – 2 SPARC instructions
- Hosking and Hudson, 1993 – Remembered sets + card-marking

Write Barrier Evaluations

- Zorn, 1990
  - Word marking in hardware, software, and page protection
  - Lisp system
- Hosking, Moss, and Stefanović, 1992
  - Remembered sets, card marking, page protection
  - Smalltalk system
- Jones and Lins, 1996
- Blackburn and McKinley, 2002
  - Inlining strategies for remembered sets of references
  - Java system (Jikes RVM)

Type-based Garbage Collection

- Prolific Types
  - Shuf, Gupta, Bordawekar, and Singh, 2002
- Connectivity-Based Garbage Collection
  - Hirzel, Henkel, Diwan, and Hind, 2002

Conclusion

- Static program analysis for removing write barriers used for generational collection
  - Interprocedural information
  - Allocation order transformations
  - Reasonable analysis times
  - In many cases, remove almost all removable write barriers
- Optimistic extension for collectors that may not promote objects in age order

Questions

Scaling

Q: How does this analysis scale?

A: Because we use interprocedural information, it may seem like the analysis would not scale well. However, the main analysis is actually intraprocedural, even though it uses type information that has to be computed using an interprocedural analysis. Now, that analysis, though interprocedural, is flow-insensitive, and context-insensitive. It can actually be computed very efficiently in linear time. So, in a efficient implementation, we expect the analysis to have very reasonable scaling properties.

Dynamic Class Loading

Q: Does the analysis support dynamic class loading?

A: Like many other analyses of this type, dynamic class loading may invalidate some analysis results. However, the optimistic extension will allow the system to operate correctly even in the presence of invalid analysis results. This makes the optimistic extension a good, low-overhead solution for using our analysis in the presence of dynamic class loading.