

## Field Analysis

---

### Last time

- Exploit encapsulation to improve memory system performance

### This time

- Exploit encapsulation to simplify analysis
- Two uses of field analysis
  - Escape analysis
  - Object inlining
- Lesson: You don't always have to solve the general problem!

## Motivation

---

### “Problems” with Modern High Level Languages

- Bounds checks for type safety
- Virtual method calls to support object-oriented semantics
- Heap allocation to provide uniform view of objects

### Solutions

- Prove facts about array bounds and about types to tighten assumptions  
*e.g.* To devirtualize a call, prove that the call has exactly one target class
- Such analysis typically requires interprocedural analysis
  - Costly
  - Sometimes impossible: dynamic class loading, unavailable source code

## Field Analysis

---

### A Cheap Form of Interprocedural Analysis

- Exploits encapsulation to limit the scope of analysis  
*e.g.*, If an array is indexed by a private variable that is only set by one method, then only that one method needs to be analyzed to determine the index's value
- Deduce properties about fields based on the properties of all accesses to that field

### Benefits

- Efficient
- Does not require access to the entire program
- Works well with dynamic class loading
- Can be applied to any language that supports encapsulation
  - Java, C++, Modula-3, *etc.*

## Field Analysis for Java

---

### Today: A specific solution [Ghemawat, Randall, & Scales, PLDI'00]

- Implemented in the context of Compaq's **Swift** optimizing Java compiler
- Swift translates bytecode to native Alpha code
- Swift performs a number of aggressive optimizations
- This implementation focuses on **reference types**
  - Ignores scalar fields

## Field Modifiers Dictate Scope of Analysis

---

### Java field modifiers

Class	Field modifier	Where can the field be assigned?
public	private	containing class
public	<i>package</i>	containing package
public	protected	containing package and subclasses
non-public	private	containing class
non-public	non-private	containing package
public	public	entire program

## Example

---

```
public class Plane {
    private Point[] points;

    public Plane() {
        points = new Point[3];
    }

    public int GetAverageColor() {
        return (points[0].GetColor() +
                points[1].GetColor() +
                points[2].GetColor())/3;
    }
}
```

Since **points** is private

- Its properties can be determined by analyzing only the **Plane** class
- We can determine the exact type of **points**
- So we can inline the **GetColor()** method

## Idea: Create an Enhanced Type System

---

### Introduce special types

- A value is an object of exactly class T, and not a subclass of T
- A value is an array of some constant size
- The value is known to be non-null
- . . .

### Type analysis begins by determining types of

- Method arguments
- Loads of fields of objects
- Loads of global variables
- Non-null exact types assigned to newly allocated objects

### Use type propagation to determine types of other nodes in the SSA graph

## Basic Approach

---

### 1. Initialize

- Build SSA graph and gather type information  
SSA provides flow-sensitivity

### 2. Incrementally update properties

- Consider all loads and stores and update properties associated with each field

#### Load of a field:

Analyze all uses of the load

```
x = y.f;  
. . .  
x.z ();
```

#### Store of a field:

Analyze the value stored into the field  
and all other uses of the value

```
x = new T;  
. . .  
y.f = x;
```

## Example of Useful Properties

---

**exact\_type**(*field*)

-The field is always assigned a value of the specified type

**always\_init**(*field*)

-The field is always initialized

**only\_init**(*field*)

-The field is only modified by constructors

## Example Analysis

---

```
public class Plane {
    private Point[] points;

    public Plane() {
        points = new Point[3];
    }

    public void SetPoint(Point p, int i) {
        points[i] = p;
    }

    public Point GetPoint(int i) {
        return points[i];
    }
}
```

**points** is private, so its properties can be determined by only scanning the Plane class

**exact\_types**(*points*) indicates a non-null array with base type **Point** and a constant size of 3

**only\_init**(*points*) is true

**always\_init**(*points*) is true

## Example Optimizations

---

### `exact_type(field)`

- If the type is precisely known, we can convert a virtual method call to a static method call
- Precise type information can be used to statically evaluate type-inclusion tests such as `instanceof` or `array store checks`
- If the type is an array of constant size, some bounds checks can be eliminated and expressions that use the array length (e.g., `a.length()`) can be statically evaluated

## Example Optimizations (cont)

---

```
public class Plane {
    private Point[] points;

    public Plane() {
        points = new Point[3];
    }

    public void SetPoint(Point p, int i) {
        points[i] = p;
    }

    public Point GetPoint(int i) {
        return points[i];
    }
}
```

What optimizations are possible in this example?

Can eliminate null checks on `points`

Can use the constant 3 in bounds checks on `points`

Can eliminate the array store check for `points`

## Example Optimizations (cont)

---

These properties can enhance other optimizations

```
x = y.f;   CSE?   x = y.f;
x.foo();  →      x.foo();
z = y.f;           z = x;
```

CSE is possible if `x.foo` does not modify `y.f`.

We know that `y.f` is only modified by a constructor if  
`only_init(f) = true`

## Escape Analysis

---

### Idea

- Does an object **escape** the method in which it is allocated?
- *E.g.*, return, assign to global/heap, pass to another method

```
f() {
    Point p = new Point();
    Stack s = new Stack(100);
    s.push(p);      /* p escapes */
    . . .
    return p;      /* p escapes */
}
```

## Escape Analysis

---

### Uses

- Objects that do not escape can be allocated on the stack

```
f() {  
    Point p = new Point();  
    return;      /* Allocate p on the stack */  
}
```

- Why is this desirable?
  - Less overhead than heap allocation
  - Less work for garbage collector
  - Usually has better cache behavior
- Synchronization elimination
  - Escape from a **thread**: Can another thread access the object?
  - If an object cannot escape a thread, it need not be synchronized

## Escape Analysis (cont)

---

### Heavyweight escape analysis

- Many proposed variations [Aldrich'99, Blanchet'99, Bogda'99, Choi'99, Whaley'99]
- Typically expensive interprocedural data-flow analysis
- Large flow values
  - **Connection Graphs** represent “points-to” relationship among objects

### Simple escape analysis

- Simplifying assumption: Any object that is assigned into the heap or returned from a method escapes that method

## Evaluation of Simple Escape Analysis

---

### Pros

- Very simple
- Inexpensive to compute (linear in code size)

### Cons

- Inaccurate
- Assignment to heap does not necessarily imply escape

## Limitations of Simple Escape Analysis

---

### Consider the following code

```
class Pair {
    private Object first;
    private Object second;
}

Pair p = new Pair();
Integer x = new Integer(5);
p.first = x;
```

### Questions

- Is **x** assigned to the heap?
- Does **x** escape?
  - Only if **p** escapes, since **x** is only assigned to an encapsulated field of **p**

## Escape Analysis with Field Analysis

---

### Idea

- Identify encapsulated fields
- If an object does not escape, then the contents of its encapsulated fields do not escape
- Escape from a thread can be handled similarly by focusing on thread creation routines

### Identifying encapsulated fields

- (1) The value of the field does not escape through a method that accesses the field, and
- (2) Any value assigned to the field has not already escaped
  - This is trivially true for newly-allocated objects

## Field Properties for Escape Analysis

---

### Field Property: `may_leak(field)`

- Indicates whether the object in the field might escape the containing object

### Field Property: `source_type(field)`

- Indicates the kind of values assigned to the field:
  - new *only assigned newly allocated objects*
  - new/null *... or null*
  - new/null/param *... or method parameters*
  - other

### A field, *f*, is encapsulated when

- `may_leak(f)` = false, and
- `source_type(f)` = new/null

## Limitations of Simple Escape Analysis (reprise)

Consider the following code

```
class Pair {
    private Object first;
    private Object second;
}

Pair p = new Pair();
Integer x = new Integer(5);
p.first = x;
```

### Questions

- Is **x** assigned to the heap? Yes
- Does **x** escape?
  - Only if **p** escapes, since **x** is only assigned to an encapsulated field of **p**
  - Check `may_leak(p)`, `source_type(p)`

## Object Inlining

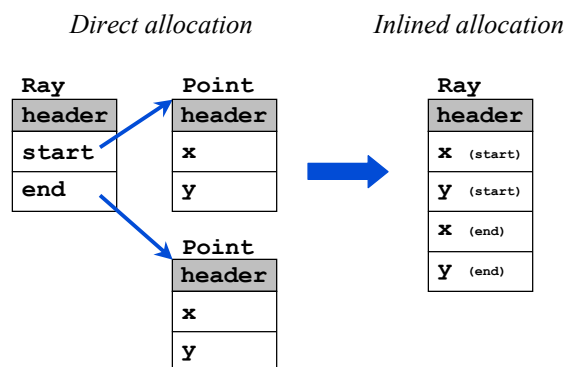
### Idea

- Allocate storage for an object **inside** its containing object

### Example

```
class Point {
    int x,y;
    ...
}

class Ray {
    Point start;
    Point end;
    ...
}
```



## Object Inlining (cont)

---

### Benefits

- Allows inlined objects to be accessed directly (*i.e.*, without following pointers)
- Reduces allocation/garbage-collection overheads
- May improve data cache performance (Inlined objects are likely to be accessed together)

### Bottom line

- Object inlining produces code much like hand-tuned C

## Object Representation and Inlining

---

### Objects contain headers

- Type of object
  - Method table
  - Synchronization state
- } Needed for type checking, virtual method calls, synchronization

**Question:** Does the header need to be preserved for inlined objects?

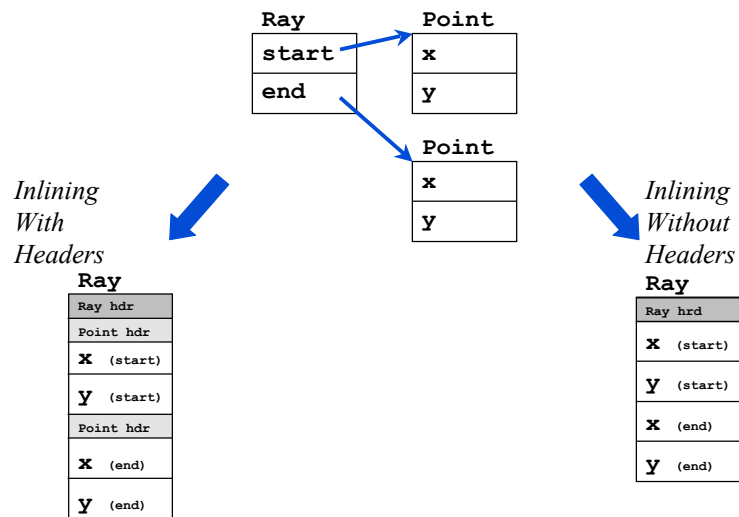
**Answer:** No, if the following hold:

- There are no virtual method invocations, no synchronization, and no type inclusion checks on the object (*i.e.*, we don't need it), and
- The object does not escape (*i.e.*, no one else will need it)
- Otherwise, `uses_header(field) = true`

**Question:** Can a compiler do this type of inlining in C++?

**Answer:** No

## Object Representation and Inlining (cont)



## Object Inlining and Garbage Collection

**Question:** What if an inlined object escapes and its enclosing object does not?

**Answer**

- Problem: the garbage collector might reclaim the enclosing object, which would also implicitly reclaim the inlined object
- Garbage collector must be aware of this

**Approaches**

- In most systems, objects that may escape cannot be inlined
- Or...
  - Inlined objects may be tagged as such (in header), and
  - The garbage collector agrees not to collect enclosing object if inlined object is live

## Object Inlining with Field Analysis

---

### Recall Field Property: `source_type(field)`

- Indicates the kind of values assigned to the field:
  - `new` *only assigned newly allocated objects*
  - `new/null` *... or null*
  - `new/null/param` *... or method parameters*
  - `other`

### For inlining we are interested in the first case

- We have static information about the candidate inlined object

## Object Inlining with Field Analysis (cont)

---

### Field Property: `uses_header(field)`

- Indicates whether the header for the object in the field might ever be used

### Recall Field Property: `may_leak(field)`

- Indicates whether the object in the field might escape the containing object

### Idea

- These properties are used to determine whether the header for inlined objects must be preserved

## Exploiting Field Analysis Properties

---

A field  $f$  can be inlined with a header when

- `always_init(f) = true`,
  - `only_init(f) = true`,
  - `source_type(f) = new`, and
  - `exact_type(f) = static_type(f)`
- } The field is always initialized exactly once by a newly allocated object

The final condition is a simplification

- It makes object layout easier for the JVM
- One layout for all inlined objects of the same static type

## Exploiting Field Analysis (cont)

---

A field  $f$  can be inlined without a header when

- It can be inlined **with** a header,
- `uses_header(f) = false`, and
- `may_leak(f) = false`

Can also inline arrays when

- The array satisfies the above constraints, and
- The array has a constant size


## Object Inlining Transformation

---

### References

`x = y.f;`            `x = y + offset(y, f);`

### Initializations

`x = new T;`  
`y.f = x;`            if required, initialize header of `y.f`  
`x = y + offset(y, f);`  
deleted

## Limitations of Field Analysis

---

### Native methods

- Cannot analyze native methods
- Conservative assumption: Assume the native methods read and write all fields that they can access

### Weak consistency

- Some optimizations are not legal under weak consistency models on multiprocessors
- Race conditions may allow a thread to see a null value even if the `always_init(field)` is true and `source_type(field) = new`

### Reflection

- Field properties can be modified through reflection (`setAccessible()`)
- Disable field analysis on such fields

## Impact on Performance

---

### Run-time check elimination

- Many null-checks eliminated (0-50%)
- Many array bounds checks eliminated (0-60%)
- Not many cast checks eliminated (0-1%)

### Virtual method calls

- Significantly reduced (0-90%)

### Object inlining

- 0-11% performance improvement

### Stack allocation

- Escape information does not significantly assist stack allocation (for the benchmarks considered)

## Impact on Performance (cont)

---

### Synchronization removal

- 0-90% reduction in dynamic synchronization
- Either helps a lot or helps very little

### Bottom line

- 0-27% performance improvement
- Average improvement of 7%
- Cheap to compute

## Concepts

---

### Escape analysis

- Useful for optimizing the allocation of objects
- Useful for removing unnecessary synchronization

### Object inlining

- Remove object overhead
- Improve data locality

### Field analysis

- Exploit encapsulation to simplify analysis
- Many uses
  - De-virtualization
  - Remove runtime checks
  - Perform escape analysis
  - Perform object inlining

## Next Time

---

### Lecture

- Traditional uses of compilers
  - Register allocation