Incrementalized Pointer and Escape Analysis

Frédéric Vivien
ICPS/LSIIT
Université Louis Pasteur
Strasbourg, France

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
Start with a program
Lots of allocation sites
Stack Allocation Optimization

void main(i,j)

void compute(d,e)

void multiplyAdd(a,b,c)

void multiply(m)

void add(u,v)

void evaluate(i,j)

void abs(r)

void scale(n,m)
Stack Allocation Optimization

Precise Whole-Program Pointer and Escape Analysis
Drawbacks to Whole-Program Analysis

- Resource Intensive
  - Large analysis times
  - Large memory consumption
- Unsuitable for partial programs
Key Observation
Number One:

Most optimizations require only the analysis of a small part of program surrounding the object allocation site.
Key Observation
Number Two:

Most of the optimization benefit comes from a small percentage of the allocation sites

99% of objects allocated at these two sites
Intuition for Better Analysis

- Locate important allocation sites
- Use demand-driven approach to analyze region surrounding site
- Somehow avoid sinking analysis resources into sites that can’t be optimized

99% of objects allocated at these two sites
What This Talk is About

How we turned this intuition into an algorithm that usually

1) obtains almost all the benefit of the whole program analysis

2) analyzes a small fraction of program

3) consumes a small fraction of whole program analysis time
Structure of Talk

- Motivating Example
- Base whole program analysis (Whaley and Rinard, OOPSLA 99)
- Incrementalized analysis
- Analysis policy
- Experimental results
- Conclusion
Motivating Example
Employee Database Example

- Read in database of employee records
- Extract statistics like max salary
Employee Database Example

- Read in database of employee records
- Extract statistics like max salary

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>$45,000</td>
</tr>
<tr>
<td>Ben Bit</td>
<td>$30,000</td>
</tr>
<tr>
<td>Jane Roe</td>
<td>$55,000</td>
</tr>
</tbody>
</table>
# Computing Max Salary

- Traverse Records to Find Max Salary

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>$45,000</td>
</tr>
<tr>
<td>Ben Bit</td>
<td>$30,000</td>
</tr>
<tr>
<td>Jane Roe</td>
<td>$55,000</td>
</tr>
</tbody>
</table>

```
max = $0
```
Computing Max Salary

- Traverse Records to Find Max Salary

Vector

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>$45,000</td>
</tr>
<tr>
<td>Ben Bit</td>
<td>$30,000</td>
</tr>
<tr>
<td>Jane Roe</td>
<td>$55,000</td>
</tr>
</tbody>
</table>

max = $45,000
who = John Doe
Computing Max Salary

- Traverse Records to Find Max Salary

Vector

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>$45,000</td>
</tr>
<tr>
<td>Ben Bit</td>
<td>$30,000</td>
</tr>
<tr>
<td>Jane Roe</td>
<td>$55,000</td>
</tr>
</tbody>
</table>

max = $45,000
who = John Doe
Computing Max Salary

- Traverse Records to Find Max Salary

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>$45,000</td>
</tr>
<tr>
<td>Ben Bit</td>
<td>$30,000</td>
</tr>
<tr>
<td>Jane Roe</td>
<td>$55,000</td>
</tr>
</tbody>
</table>

Vector

max = $55,000
who = Jane Roe
**Computing Max Salary**

- Traverse Records to Find Max Salary

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>$45,000</td>
</tr>
<tr>
<td>Ben Bit</td>
<td>$30,000</td>
</tr>
<tr>
<td>Jane Roe</td>
<td>$55,000</td>
</tr>
</tbody>
</table>

\[
\text{max salary} = \$55,000 \\
\text{highest paid} = \text{Jane Roe}
\]
class EmployeeDatabase {
    Vector database = new Vector();
    Employee highestPaid;
    void computeMax() {
        int max = 0;
        Enumeration enum = database.elements();
        while (enum.hasMoreElements()) {
            Employee e = enum.nextElement();
            if (max < e.salary()) {
                max = e.salary();
                highestPaid = e;
            }
        }
    }
}
class EmployeeDatabase {
    Vector database = new Vector();
    Employee highestPaid;
    void computeMax() {
        int max = 0;
        Enumeration enum = database.elements();
        while (enum.hasMoreElements()) {
            Employee e = enum.nextElement();
            if (max < e.salary()) {
                max = e.salary(); highestPaid = e;
            }
        }
    }
}
Issues In Implementation

- Enumeration object allocated on heap
  - Increases heap memory usage
  - Increases garbage collection frequency

- Heap allocation is unnecessary
  - Enumeration object allocated inside max
  - Not accessible outside max
  - Should be able to use stack allocation
Basic Idea

Use pointer and escape analysis to recognize captured objects

Transform program to allocate captured objects on stack
Base Analysis
Base Analysis

- Basic Abstraction: Points-to Escape Graph
- Intraprocedural Analysis
  - Flow sensitive abstract interpretation
  - Produces points-to escape graph at each program point
- Interprocedural Analysis
  - Bottom Up and Compositional
  - Analyzes each method once to obtain a single parameterized analysis result
  - Result is specialized for use at each call site
void computeMax() {
    int max = 0;
    Enumeration enum = database.elements();
    while (enum.hasMoreElements()) {
        Employee e = enum.nextElement();
        if (max < e.salary()) { max = e.salary(); highestPaid = e; }
    }
}

Points-to Escape Graph in Example
Edge Types

- **Inside Edges:**
  - created in currently analyzed part of program

- **Outside Edges:**
  - created outside currently analyzed part of program

Diagram:

```
enum vector elementData [ ]
dashed = outside
solid = inside
database
highestPaid
this
e
```
Node Types

- **Inside Nodes:**
  - Represent objects created in currently analyzed part of program

- **Outside Nodes:**
  - Parameter nodes – represent parameters
  - Load nodes - represent objects accessed via pointers created outside analyzed part

```
dashed = outside
solid = inside
```
Escaped Nodes

- **Escaped** nodes
  - parameter nodes
  - thread nodes
  - returned nodes
  - *nodes reachable from other escaped nodes*
- Captured is the opposite of escaped

\[
\text{enum} \rightarrow \text{vector} \rightarrow \text{elementData} \rightarrow [\] \rightarrow \text{database} \rightarrow \text{highestPaid} \rightarrow \text{this} \rightarrow \text{e}
\]

\[\text{green} = \text{escaped}\]
\[\text{white} = \text{captured}\]
Stack Allocation Optimization

- Examine graph from end of method
- If a node is captured in this graph
- Allocate corresponding objects on stack (may need to inline methods to apply optimization)

```
vector elementData[ ]
this
database
highestPaid
```

green = escaped
white = captured

Can allocate enum object on stack
Interprocedural Analysis

```java
void printStatistics(BufferedReader r) {
    EmployeeDatabase e = new EmployeeDatabase(r);
    e.computeMax();
    System.out.println("max salary = " + e.highestPaid);
}
```
void printStatistics(BufferedReader r) {
    EmployeeDatabase e = new EmployeeDatabase(r);
    e.computeMax();
    System.out.println("max salary = " + e.highestPaid);
}
void printStatistics(BufferedReader r) {
    EmployeeDatabase e = new EmployeeDatabase(r);
    e.computeMax();
    System.out.println("max salary = " + e.highestPaid);
}
void printStatistics(BufferedReader r) {
    EmployeeDatabase e = new EmployeeDatabase(r);
    e.computeMax();
    System.out.println("max salary = " + e.highestPaid);
}
void printStatistics(BufferedReader r) {
    EmployeeDatabase e = new EmployeeDatabase(r);
    e.computeMax();
    System.out.println("max salary = " + e.highestPaid);
}
void printStatistics(BufferedReader r) {
    EmployeeDatabase e = new EmployeeDatabase(r);
    e.computeMax();
    System.out.println("max salary = " + e.highestPaid);
}
void printStatistics(BufferedReader r) {
    EmployeeDatabase e = new EmployeeDatabase(r);
    e.computeMax();
    System.out.println("max salary = " + e.highestPaid);
}
void printStatistics(BufferedReader r) {
    EmployeeDatabase e = new EmployeeDatabase(r);
    e.computeMax();
    System.out.println("max salary = " + e.highestPaid);
}

Match corresponding inside and outside edges to complete mapping
void printStatistics(BufferedReader r) {
    EmployeeDatabase e = new EmployeeDatabase(r);
    e.computeMax();
    System.out.println("max salary = " + e.highestPaid);
}
void printStatistics(BufferedReader r) {
    EmployeeDatabase e = new EmployeeDatabase(r);
    e.computeMax();
    System.out.println("max salary = " + e.highestPaid);
}
void computeMax()

Enumeration elements()

Vector elementData()

boolean hasMoreElements()

Employee nextElement()

int salary()

void printStatistics()
Whole Program Analysis

void computeMax()

Enumeration elements()

Vector elementData()

boolean hasMoreElements()

void printStatistics()

Employee nextElement()

int salary()
Whole Program Analysis

void computeMax()

Enumeration elements()

boolean hasMoreElements()

Vector elementData()

void printStatistics()

Employee nextElement()

int salary()
Whole Program Analysis

void computeMax()

Enumeration elements()

Vector elementData()

boolean hasMoreElements()

void printStatistics()

Employee nextElement()

int salary()
Whole Program Analysis

void computeMax()

Enumeration elements()

Vector elementData()

boolean hasMoreElements()

void printStatistics()

Employee nextElement()

int salary()
Whole Program Analysis

void computeMax()

Enumeration elements()

Vector elementData()

boolean hasMoreElements()

Employee nextElement()

int salary()

void printStatistics()
void computeMax()

Enumeration elements()

Vector elementData()

void printStatistics()

boolean hasMoreElements()

Employee nextElement()

int salary()
Incrementalized Analysis
Incrementalized Analysis Requirements

Must be able to

- Analyze method independently of callers
  - Base analysis is compositional
  - Already does this

- Skip analysis of invoked methods

- But later incrementally integrate analysis results if desirable to do so
First Extension to Base Analysis

- Skip the analysis of invoked methods
- Parameters are marked as escaping into skipped call site

Assume analysis skips `enum.nextElement()`

Node 1 escapes into `enum.nextElement()`
First Extension Almost Works

- Can skip analysis of invoked methods
- If allocation site is captured, great!
- If not, escape information tells you what methods you should have analyzed...

Should have analyzed `enum.nextElement()`

Node 1 escapes into `enum.nextElement()`
Second Extension to Base Algorithm

- Record enough information to undo skip and incorporate analysis into existing result
  - Parameter mapping at call site
  - Ordering information for call sites
void compute() {
    foo(x,y);
}

Graph before call site

Graph after call site

Graph at end of method

Generated during analysis

Used to apply stack allocation optimization
void compute() {
    foo(x,y);
}

Graphs from Incrementalized Analysis

Graph before call site

Graph after skipped call site

Graph at end of method

Generated during analysis

Used to apply stack allocation optimization
Incorporating Result from Skipped Call Site

Naive approach: use mapping algorithm directly on graph from end of caller method

Before incorporating result from skipped call site

After incorporating result from skipped call site
Incorporating Result from Skipped Call Site

Naive approach: use mapping algorithm directly on graph from end of caller method

Graph from whole program analysis

After incorporating result from skipped call site
Basic Problem and Solution

- **Problem:** additional edges in graph from end of method make result less precise
- **Solution:** augment abstraction
  - For each call skipped call site, record
    - Edges which were present in the graph before the call site
    - Edges which were present after call site
  - Use this information when incorporating results from skipped call sites
After Augmenting Abstraction

Graph from whole program analysis

After incorporating result from skipped call site
Incrementalized Analysis
Incrementalized Analysis

Attempt to stack allocate Enumeration object from elements
Incrementalized Analysis

Analyze elements (intraprocedurally)
Incrementalized Analysis

Analyze elements (intraprocedurally)
Incrementalized Analysis

Analyze elements (intraprocedurally)

Escapes only into the caller
Incrementalized Analysis

Analyze computeMax
(intraprocedurally)
Incrementalized Analysis

Analyze computeMax
(intraprocedurally)
Incrementalized Analysis

Analyze computeMax (intraprocedurally)

Escapes to
- hasMoreElements
- nextElement
Incrementalized Analysis

Analyze hasMoreElements
Incrementalized Analysis

Analyze hasMoreElements
Incrementalized Analysis

Analyze hasMoreElements

Combine results

void computeMax()

Enumeration elements()

boolean hasMoreElements()

Employee nextElement()

Vector elementData()

void printStatistics()

int salary()
Incrementalized Analysis

Analyze hasMoreElements

Combine results

Still escaping to
- nextElement
Incrementalized Analysis

Analyze nextElement

void computeMax()

Enumeration elements()

boolean hasMoreElements()

Vector elementData()

Employee nextElement()

int salary()

void printStatistics()
Incrementalized Analysis

Analyze nextElement

Enumeration elements()
Incrementalized Analysis

Analyze `nextElement`

Combine results

- `computeMax()`
- `elements()`
- `elementData()`
- `hasMoreElements()`
- `printStatistics()`
- `Employee nextElement()`
- `int salary()`
Incrementalized Analysis

Analyze nextElement

Combine results
Incrementalized Analysis

Enumeration object Captured in computeMax
Incrementalized Analysis

Enumeration object **Captured** in computeMax

Inline elements

Stack allocate enumeration object
Incrementalized Analysis

We skipped the analysis of some methods
Incrementalized Analysis

We skipped the analysis of some methods

We ignored some other methods
Result

- We can incrementally analyze
  - Only what is needed
  - For whatever allocation site we want
  - And even temporarily suspend analysis part of the way through!
New Issue

- We can incrementally analyze
  - Only what is needed
  - For whatever allocation site we want
  - And even temporarily suspend analysis part of the way through!

But...

- Lots of analysis opportunities
  - Not all opportunities are profitable
  - Where to invest analysis resources?
  - How much resources to invest?
Analysis Policy

Formulate policy as solution to an investment problem

Goal
Maximize optimization payoff from invested analysis resources
Analysis Policy Implementation

- For each allocation site, estimate marginal return on invested analysis resources
- Loop
  - Invest a unit of analysis resources (time) in site that offers best return
  - Expand analyzed region surrounding site
- When unit expires, recompute marginal returns (best site may change)
Marginal Return Estimate

\[
\frac{N \cdot P(d)}{C \cdot T}
\]

- \(N\) = Number of objects allocated at the site
- \(P(d)\) = Probability of capturing the site, knowing we explored a region of call depth \(d\)
- \(C\) = Number of skipped call sites the allocation site escapes through
- \(T\) = Average time needed to analyze a call site
Marginal Return Estimate

\[
\frac{N \cdot P(d)}{C \cdot T}
\]

As invest analysis resources

- explore larger regions around allocation sites
- get more information about sites
- marginal return estimates improve
- analysis makes better investment decisions!
Usage Scenarios

- **Ahead of time compiler**
  - Give algorithm an analysis budget
  - Algorithm spends budget
  - Takes whatever optimizations it uncovered

- **Dynamic compiler**
  - Algorithm acquires analysis budget as a percentage of run time
  - Periodically spends budget, delivers additional optimizations
  - Longer program runs, more optimizations
Experimental Results
Methodology

- Implemented analysis in MIT Flex System
- Obtained several benchmarks
  - Scientific computations: barnes, water
  - Our lexical analyzer: jlex
  - Spec benchmarks: db, raytrace, compress
Allocation Sites Analyzed

Percentage of Allocation Sites Touched

barnes  water  jlex  db  raytrace  compress
Analysis Time Payoff

Stack allocated by incrementalized analysis
Decided
Stack allocated by whole-program analysis
Stack Allocation

Percentage of Memory Allocated on Stack

- Incrementalized Analysis
- Whole-Program Analysis
Normalized Execution Times

Reference: execution time without optimization

- barnes
- water
- jlex
- db
- raytrace
- compress

Incrementalized Analysis
Whole-Program Analysis
Experimental Summary

Key Application Properties
Most objects allocated at very few allocation sites
Can capture objects with an incremental analysis of region surrounding allocation sites

Consequence
Most of the benefits of whole-program analysis
Fraction of the cost of whole-program analysis
Related Work

Demand-driven Analyses

- Horwitz, Reps, Sagiv (FSE 1995)
- Duesterwald, Soffa, Gupta (TOPLAS 1997)
- Heintze, Tardieu (PLDI 2001)

Key differences

- Integration of escape information enables incrementalized algorithms to suspend partially completed analyses
- Maintain accurate marginal payoff estimates
- Avoid overly costly analyses
Related Work

Previous Escape Analyses

- Blanchet (OOPSLA 1999)
- Bogda, Hoelzle (OOPSLA 1999)
- Choi, Gupta, Serrano, Sreedhar, Midkiff (OOPSLA 1999)
- Whaley, Rinard (OOPSLA 1999)
- Ruf (PLDI 2000)

Key Differences

- Previous algorithms analyze whole program
- But could incrementalize other analyses
- Get a range of algorithms with varying analysis time/precision tradeoffs
Conclusion

Whole-Program Analysis  Incrementalized Analysis

Properties
- Uses escape information to incrementally analyze relevant regions of program
- Analysis policy driven by estimates of optimization benefits and costs

Results
- Most of benefits of whole program analysis
- Fraction of the cost