Compositional Pointer and Escape Analysis for Java Programs

based on the paper by John Whaley and Martin Rinard
presented by Natalia Prytkova

28.06.2010

Universität des Saarlandes
### Motivation

**The aim:**
- Optimization

**Two critical moments:**
1. Memory allocation
2. Synchronization

**The main question:**
How to eliminate the overhead related with synchronization and allocation of objects in the heap?
Memory
• Register
• The stack
• The heap
• Constant storage

Solution
Objects that would otherwise be allocated in the heap and then reclaimed using garbage collection are allocated in an activation record on the stack and automatically (and cheaply) reclaimed when the activation record is popped off the stack. (Young Gil Park and Benjamin Goldberg Escape Analysis on Lists)
Storage in Java

- Register
- The stack
- The heap
- Constant storage

Solution

Objects that would otherwise be allocated in the heap and then reclaimed using garbage collection are allocated in an activation record on the stack and automatically (and cheaply) reclaimed when the activation record is popped off the stack. (Young Gil Park and Benjamin Goldberg Escape Analysis on Lists)
Storage in Java

- Register
- The stack
- The heap
- Constant storage

Solution

Objects that would otherwise be allocated in the heap and then reclaimed using garbage collection are allocated in an activation record on the stack and automatically (and cheaply) reclaimed when the activation record is popped off the stack. (Young Gil Park and Benjamin Goldberg Escape Analysis on Lists)
Storage in Java

- Register
- The stack
- The heap
- Constant storage

Solution

Objects that would otherwise be allocated in the heap and then reclaimed using garbage collection are allocated in an activation record on the stack and automatically (and cheaply) reclaimed when the activation record is popped off the stack. (Young Gil Park and Benjamin Goldberg Escape Analysis on Lists)
Objects that would otherwise be allocated in the heap and then reclaimed using garbage collection are allocated in an activation record on the stack and automatically (and cheaply) reclaimed when the activation record is popped off the stack. (Young Gil Park and Benjamin Goldberg Escape Analysis on Lists)
class complex
{
  double x, y;
  complex (double a, double b) { x = a; y = b; }

  complex multiply(complex a)
  {
    complex product =
      new complex(a*a.x - y*a.y, x*a.y + y*a.x);
    return product;
  }

  complex add(complex a)
  {
    complex sum =
      new complex(x + a.x, y + a.y);
    return sum;
  }

  complex multiplyAdd(complex a, complex b)
  {
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
  }
}
class complex
{
    double x, y;
    complex (double a, double b){x = a; y = b;}

    complex multiply(complex a)
    {
        complex product =
            new complex(a*a.x - y*a.y, x*a.y + y*a.x);
        return product;
    }

    complex add(complex a)
    {
        complex sum =
            new complex(x + a.x, y + a.y);
        return sum;
    }

    complex multiplyAdd(complex a, complex b)
    {
        complex product = a.multiply(b);
        complex sum = this.add(product);
        return sum;
    }
}
Example: Return Values

class complex
{
    double x, y;
    complex (double a, double b){x = a; y = b;}

    complex multiply(complex a)
    {
        complex product =
            new complex(a.a.x - y.a.y, x.a.y + y.a.x);
        return product;
    }

    complex add(complex a)
    {
        complex sum =
            new complex(x + a.x, y + a.y);
        return sum;
    }

    complex multiplyAdd(complex a, complex b)
    {
        complex product = a.multiply(b);
        complex sum = this.add(product);
        return sum;
    }
}

Example: Analysis Result for multiplyAdd

```java
complex multiplyAdd(complex a, complex b)
{
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```

Diagram:
```
  a
   v
   b
   v
  this
   v
  product
   v
  sum
```
The node that `product` points to is accessible only via the local variable `product`. Lifetime of the object is equal to the lifetime of the method invocation $\Rightarrow$ `product` can be allocated in the activation record of the `multiplyAdd` method.
The node that \textit{product} points to is accessible only via the local variable \textit{product}. Lifetime of the object is equal to the lifetime of the method invocation $\Rightarrow$ \textit{product} can be allocated in the activation record of the \textit{multiplyAdd} method.
Escaped objects

When we can not allocate an object in the stack?

- A reference to the object was passed as a parameter into the current method
- A reference to the object was written into a static class variable
- A reference to the object was passed as a parameter to an invoked method, and there is no information about what the invoked method did with the object.
- The object is a thread object

In these situations we say, that an object escapes the current method.
Escaped objects

When we can not allocate an object in the stack?

- A reference to the object was passed as a parameter into the current method
- A reference to the object was written into a static class variable
- A reference to the object was passed as a parameter to an invoked method, and there is no information about what the invoked method did with the object.
- The object is a thread object

In these situations we say, that an object escapes the current method.
When we can not allocate an object in the stack?

- A reference to the object was passed as a parameter into the current method
- A reference to the object was written into a static class variable
- A reference to the object was passed as a parameter to an invoked method, and there is no information about what the invoked method did with the object.
- The object is a thread object

In these situations we say, that an object escapes the current method.
When we can not allocate an object in the stack?

- A reference to the object was passed as a parameter into the current method
- A reference to the object was written into a static class variable
- A reference to the object was passed as a parameter to an invoked method, and there is no information about what the invoked method did with the object.
- The object is a thread object

In these situations we say, that an object escapes the current method.
The Second Problem

Synchronization

http://openbook.galileodesign.de/javainsel5/javainsel09_005.htm
Why and when do we need synchronization?

To prevent the problem of thread collision.
If you are writing a variable that might next be read by another thread or reading the variable that might have last been written by another thread, you must use synchronization. (Bruce Eckel, Thinking in Java)
Why and when do we need synchronization?

To prevent the problem of thread collision. If you are writing a variable that might next be read by another thread or reading the variable that might have last been written by another thread, you must use synchronization. (Bruce Eckel, Thinking in Java)
Why and when do we need synchronization?

To prevent the problem of thread collision. If you are writing a variable that might next be read by another thread or reading the variable that might have last been written by another thread, you must use synchronization. (Bruce Eckel, Thinking in Java)

Solution

If an object does not escape a thread, it is legal to remove the lock acquire and release operations from synchronized methods that execute on that object.
Example: Synchronization

```java
public class Server extends Thread {

    ServerSocket serverSocket;
    int duplicateConnections;

    Server(ServerSocket s) {
        serverSocket = s;
        duplicateConnections = 0;
    }

    public void run() {
        try {
            Vector connectionList = new Vector();
            while(true) {
                Socket clientSocket = serverSocket.accept();
                new ServerHelper(clientSocket).start();
                InetAddress addr = clientSocket.getInetAddress();
                if (connectionList.indexOf(addr) < 0)
                    connectionList.addElement(addr);
                else
duplicateConnections++;
            }
        } catch (IOException e) {} 
    }
}
```
```java
public class Server extends Thread {

    ServerSocket serverSocket;
    int duplicateConnections;

    Server(ServerSocket s) {
        serverSocket = s;
        duplicateConnections = 0;
    }

    public void run() {
        try {
            Vector connectionList = new Vector();
            while (true) {
                Socket clientSocket = serverSocket.accept();
                new ServerHelper(clientSocket).start();
                InetAddress addr = clientSocket.getInetAddress();
                if (connectionList.indexOf(addr) < 0)
                    connectionList.addElement(addr);
                else duplicateConnections++;
            }
        }
        catch (IOException e) {} 
    }
}
```
## Analysis Algorithm

### What is to analyse?

The algorithm analyses the program at the granularity of methods.

### How to analyse?

- Represent a program (interprocedural level)
- Represent an object (intraprocedural level)
What is to analyse?
The algorithm analyses the program at the granularity of methods.

How to analyse?
- Represent a program (interprocedural level)
- Represent an object (intraprocedural level)
class complex
{
    double x, y;
    complex (double a, double b){x = a; y = b;}

    complex multiply(complex a)
    {
        complex product =
            new complex(a.a.x - y*a.y, x*a.y + y*a.x);
        return product;
    }

    complex add(complex a)
    {
        complex sum =
            new complex(x + a.x, y + a.y);
        return sum;
    }

    complex multiplyAdd(complex a, complex b)
    {
        complex product = a.multiply(b);
        complex sum = this.add(product);
        return sum;
    }
}
class complex
{
    double x, y;
    complex (double a, double b){x = a; y = b;}

    complex multiply(complex a)
    {
        complex product =
            new complex(a*a.x - y*a.y, x*a.y + y*a.x);
        return product;
    }

    complex add(complex a)
    {
        complex sum =
            new complex(x + a.x, y + a.y);
        return sum;
    }

    complex multiplyAdd(complex a, complex b)
    {
        complex product = a.multiply(b);
        complex sum = this.add(product);
        return sum;
    }
}
class complex
{
  double x, y;
  complex (double a, double b){x = a; y = b;}

  complex multiply(complex a)
  {
    complex product =
    new complex(a*x - y*a.y, x*a.y + y*a.x);
    return product;
  }

  complex add(complex a)
  {
    complex sum =
    new complex(x + a.x, y + a.y);
    return sum;
  }

  complex multiplyAdd(complex a, complex b)
  {
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
  }
}
class complex
{
    double x, y;
    complex (double a, double b){ x = a; y = b; }

    complex multiply(complex a)
    {
        complex product =
            new complex(a*a.x - y*a.y, x*a.y + y*a.x);
        return product;
    }

    complex add(complex a)
    {
        complex sum =
            new complex(x + a.x, y + a.y);
        return sum;
    }

    complex multiplyAdd(complex a, complex b)
    {
        complex product = a.multiply(b);
        complex sum = this.add(product);
        return sum;
    }
}
Program representation

Analysis objects

- Set of classes \( cl \in Cl \)
- Set of object fields \( f \in F \)
- Set of local variables \( l \in L \)
- Set of formal parameters \( p \in P \)
Program representation

Statements that affect points-to and escape information

- A copy statement \( l = v \)
- A load statement \( l_1 = l_2.f \)
- A store statement \( l_1.f = l_2 \)
- A global load statement \( l = cl.f \)
- A global store statement \( cl.f = l \)
- A return statement \( \text{return } l \)
- An object creation \( l = \text{new } cl \)
- A method invocation \( l = l_0.op(l_1, \ldots l_k) \)
class complex
{
    double x, y;
    complex (double a, double b){x = a; y = b;}

    complex multiply(complex a)
    {
        complex product =
            new complex(a*x - y*a.y, x*a.y + y*a.x);
        return product;
    }

    complex add(complex a)
    {
        complex sum =
            new complex(x + a.x, y + a.y);
        return sum;
    }

    complex multiplyAdd(complex a, complex b)
    {
        complex product = a.multiply(b);
        complex sum = this.add(product);
        return sum;
    }
}
## Types of nodes

- Set of inside nodes $N_I$
- Set of thread nodes $N_T$, $N_T \subseteq N_I$
- Set of outside nodes $N_O$
- Set of parameter nodes $N_P$, $N_P \subseteq N_O$
- Set of global nodes $N_G$, $N_G \subseteq N_O$
- Set of load nodes $N_L$, $N_L \subseteq N_O$
- Set of return nodes $N_R$, $N_R \subseteq N_O$
Points-to Escape Graph

- Is a quadruple \( \langle O, I, e, r \rangle \)
- \( O \) is a set of outside edges \( O \subseteq (N \times F) \times (N_L \cup N_G) \)
- \( I \) is a set of inside edges \( I \subseteq (V \times (N \times F)) \times N \)
- \( e : N \rightarrow 2^M \) is an escape function
- \( r \subseteq N \) is a return set
Points-to Escape Graph

```java
complex multiplyAdd(complex a, complex b) {
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```

- $O = \emptyset$
- $I = \{ \langle \textit{this}, N_{\textit{this}} \rangle, \langle a, N_a \rangle, \langle b, N_b \rangle, \langle \textit{product}, N_{\textit{product}} \rangle, \langle \textit{sum}, N_{\textit{sum}} \rangle \}$
- $e = \{ (b \rightarrow a\text{.multiply}(b)), (\textit{product} \rightarrow \textit{this}\text{.add}(\textit{product})) \}$
- $r = \{ N_{\textit{sum}} \}$
Intraprocedural Analysis: Example

```java
complex multiplyAdd(complex a, complex b)
{
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```

**Line #20**

- $O = \emptyset$
- $I = \{ \langle this, N_{this} \rangle, \langle a, N_a \rangle, \langle b, N_b \rangle \}$
- $\forall n \ e_0(n) = \emptyset$
- $r = \emptyset$
Intraprocedural Analysis: Example

```java
complex multiplyAdd(complex a, complex b) {
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```

**Line #22**

- $Kill_I = \text{edges}(I, product)$
- $Gen_I = \{product\} \times I(a\text{.multiply}(b))$
- $I = (I - Kill_I) \cup Gen_I = I + \{\langle product, a\text{.multiply}(b)\rangle\}$
Intraprocedural Analysis: Example

```java
complex multiplyAdd(complex a, complex b)
{
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```

Line #23

- \( \text{Kill}_I = \text{edges}(I, \text{sum}) \)
- \( \text{Gen}_I = \{\text{sum}\} \times I(\text{this.add(product)}) \)
- \( I = (I - \text{Kill}_I) \cup \text{Gen}_I = I + \{\langle \text{sum}, \text{this.add(product)} \rangle\} \)
Intraprocedural Analysis: Example

```java
complex multiplyAdd(complex a, complex b)
{
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```

**Line #24**

- $r = I(sum) = \{N_{sum}\}$
Intraprocedural Analysis: Copy Statement $l = v$

\[ Kill_I = edges(I, l) \]
\[ Gen_I = \{l\} \times I(v) \]
\[ I = (I - \text{Kill}_I) \cup \text{Gen}_I \]
Intraprocedural Analysis: Load Statement \( l_1 = l_2.f \), 1st Case

\[
Kill_I = \text{edges}(I, l_1)
\]

\[
Gen_I = \{l_1\} \times \{I(l_2) \times \{f\}\}
\]

\[
I = (I - Kill_I) \cup Gen_I
\]
Intraprocedural Analysis: Load Statement $l_1 = l_2.f$, 2nd Case

$Kill_I = edges(I, l_1)$

$Gen_I = \{l_1\} \times \{I(l_2) \times \{f\}\}$

$I = (I - Kill_I) \cup Gen_I$

$Gen_O = \{I(l_2) \times \{f\}\} \times n$

$O = O \cup Gen_O$
Intraprocedural Analysis: Store Statement $l_1.f = l_2$

\[
Gen_I = \{I(l_1) \times \{f\}\} \times I(l_2)
\]

\[
I = I \cup Gen_I
\]
Intraprocedural Analysis: Global Load Statement \( l = \text{cl}.f \)

\[
\begin{align*}
\text{Kill}_I &= \text{edges}(I, l) \\
\text{Gen}_I &= \{l\} \times (O \cup I)(\text{cl}, f) \\
I &= (I - \text{Kill}_I) \cup \text{Gen}_I
\end{align*}
\]
Intraprocedural Analysis: Global Store Statement $cl.f = l$

\[ Gen_I = \{<cl, f>\} \times I(l) \]
\[ I = I \cup Gen_I \]
Intraprocedural Analysis: Object Creation Sites \( l = \text{new } cl \)

\[ \begin{align*}
    Kill_I &= \text{edges}(I, l) \\
    Gen_I &= \{ \langle l, n \rangle \} \\
    I &= (I - Kill_I) \cup Gen_I
\end{align*} \]

1 \( \rightarrow \) 3  is the inside node for \( l = \text{new } cl \)
Intraprocedural Analysis: Return Statement \textit{return} \( l \)

\[ r = I(l) \]
Intraprocedural Analysis

We analyse a method line by line, statement by statement

<table>
<thead>
<tr>
<th>Analysis Result Before Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha(\bullet st) = \sqcup { \alpha(st'\bullet) : st' \in \text{pred}(st) }$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis Result After Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha(st\bullet) = [st](\alpha(\bullet st))$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis Result Before Method Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha(\bullet enter_{op}) = \langle O_0, I_0, e_0, r_0 \rangle$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis Result After Method Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha(exit_{op}\bullet) = \langle O', I', e', r' \rangle = \text{remove}(S, \langle O, I, e, r \rangle)$</td>
</tr>
<tr>
<td>where $S = { n \in N : \text{not reachable} \ (O \cup I, CL \cup P \cup r, n) }$</td>
</tr>
</tbody>
</table>
We analyse a method line by line, statement by statement

**Analysis Result Before Statement**

\[ \alpha(\bullet st) = \sqcup \{ \alpha(st'\bullet) : st' \in pred(st) \} \]

**Analysis Result After Statement**

\[ \alpha(st\bullet) = [st](\alpha(\bullet st)) \]

**Analysis Result Before Method Invocation**

\[ \alpha(\bullet enter_{op}) = \langle O_0, I_0, e_0, r_0 \rangle \]

**Analysis Result After Method Invocation**

\[ \alpha(exit_{op}\bullet) = \langle O', I', e', r' \rangle = \text{remove}(S, \langle O, I, e, r \rangle) \]

where \( S = \{ n \in N : \text{not reachable} \ (O \cup I, CL \cup P \cup r, n) \} \)
Intraprocedural Analysis

We analyse a method line by line, statement by statement

<table>
<thead>
<tr>
<th>Analysis Result Before Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha(\bullet st) = \sqcup{\alpha(st'\bullet) : st' \in \text{pred}(st)}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis Result After Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha(st\bullet) = [st](\alpha(\bullet st))$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis Result Before Method Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha(\bullet enter_{op}) = \langle O_0, I_0, e_0, r_0 \rangle$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis Result After Method Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha(exit_{op}\bullet) = \langle O', I', e', r' \rangle = \text{remove}(S, \langle O, I, e, r \rangle)$</td>
</tr>
<tr>
<td>where $S = {n \in N : \text{not reachable } (O \cup I, CL \cup P \cup r, n)}$</td>
</tr>
</tbody>
</table>
Intraprocedural Analysis

We analyse a method line by line, statement by statement

**Analysis Result Before Statement**
\[
\alpha(\bullet st) = \bigcup \{ \alpha(st'\bullet) : st' \in pred(st) \}
\]

**Analysis Result After Statement**
\[
\alpha(st\bullet) = [st](\alpha(\bullet st))
\]

**Analysis Result Before Method Invocation**
\[
\alpha(\bullet enter_{op}) = \langle O_0, I_0, e_0, r_0 \rangle
\]

**Analysis Result After Method Invocation**
\[
\alpha(exit_{op}\bullet) = \langle O', I', e', r' \rangle = remove(S, \langle O, I, e, r \rangle)
\]
where \( S = \{ n \in N : \text{not reachable } (O \cup I, CL \cup P \cup r, n) \} \)
At each method invocation site, the analysis has the option of either skipping the site or analysing the site.

**Skipped Method Invocation Sites**

- $l = l_0\.op(l_1 \ldots l_k)$, return node $n_R$
- $\langle O, I', e, r' \rangle = [m](\langle O, I, e, r \rangle)$
- $I' = (I - edges(I, l)) \cup \{\langle l, n_R \rangle\}$
- $e'(n) = \begin{cases} 
  e(n) \cup m & \text{if } \exists 0 \leq i \leq k : n \in I(l_i) \\
  e(n) & \text{otherwise}
\end{cases}$

**Analysed Method Invocation Sites**

- $l = l_0\.op(l_1 \ldots l_k)$
- $\langle O, I', e, r' \rangle = [m](\langle O, I, e, r \rangle)$
- $\langle O, I', e, r' \rangle = \sqcup \{\text{map}(m, \langle O, I, e, r \rangle, op) : op \in callees(m)\}$
Mapping Algorithm

We have 3 different points-to escape graphs:

```java
void function()
{
    complex a1 = new Complex(1.0, 2.0);
    complex b1 = new Complex(3.0, 4.0);
    complex c1 = new Complex(5.0, 6.0);

    complex d = c.multiplyAdd(a1, b1);
}
```

The old graph (before method invocation)
We have 3 different points-to escape graphs:

```java
void function()
{
    complex a1 = new Complex(1.0, 2.0);
    complex b1 = new Complex(3.0, 4.0);
    complex c1 = new Complex(5.0, 6.0);

    complex d = c.multiplyAdd(a1, b1);
}
```

The incoming graph (the analysis result for the invoked method)
Mapping Algorithm

We have 3 different points-to escape graphs:

```java
void function() {
    complex a1 = new Complex(1.0, 2.0);
    complex b1 = new Complex(3.0, 4.0);
    complex c1 = new Complex(5.0, 6.0);
    complex d = c.multiplyAdd(a1, b1);
}
```

The incoming graph (the analysis result for the invoked method)
We have 3 different points-to escape graphs:

```java
void function()
{
    complex a1 = new Complex(1.0, 2.0);
    complex b1 = new Complex(3.0, 4.0);
    complex c1 = new Complex(5.0, 6.0);
    complex d = c.multiplyAdd(a1, b1);
}
```

The new graph (after method invocation)
Mapping Algorithm: Rule 1

The new graph (after the method invocation) extends the old:

```java
void function()
{
    complex a1 = new Complex(1.0, 2.0);
    complex b1 = new Complex(3.0, 4.0);
    complex c1 = new Complex(5.0, 6.0);
    complex d = c.multiplyAdd(a1, b1);
}
```

where

```java
complex multiplyAdd(complex a, complex b)
{
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```
Mapping Algorithm: Rule 2

Map each parameter node to the node in the old graph:

```java
void function()
{
    complex a1 = new Complex(1.0, 2.0);
    complex b1 = new Complex(3.0, 4.0);
    complex c1 = new Complex(5.0, 6.0);
    complex d = c.multiplyAdd(a1, b1);
}
```

where

```java
complex multiplyAdd(complex a, complex b)
{
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```
Mapping Algorithm: Rule 2

Map each parameter node to the node in the old graph:

```java
void function()
{
    complex a1 = new Complex(1.0, 2.0);
    complex b1 = new Complex(3.0, 4.0);
    complex c1 = new Complex(5.0, 6.0);

    complex d = c.multiplyAdd(a1, b1);
}
```

where

```java
complex multiplyAdd(complex a, complex b)
{
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```
Mapping Algorithm: Rule 2

Map each parameter node to the node in the old graph:

```java
void function()
{
    complex a1 = new Complex(1.0, 2.0);
    complex b1 = new Complex(3.0, 4.0);
    complex c1 = new Complex(5.0, 6.0);

    complex d = c.multiplyAdd(a1, b1);
}
```

where

```java
complex multiplyAdd(complex a, complex b)
{
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```
Mapping Algorithm: Rule 3

Map the return nodes from incoming to the new graph:

```java
void function()
{
    complex a1 = new Complex(1.0, 2.0);
    complex b1 = new Complex(3.0, 4.0);
    complex c1 = new Complex(5.0, 6.0);

    complex d = c.multiplyAdd(a1, b1);
}
```

where

```java
complex multiplyAdd(complex a, complex b)
{
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```
Mapping Algorithm: Rule 3

Map the return nodes from incoming to the new graph:

```java
void function()
{
    complex a1 = new Complex(1.0, 2.0);
    complex b1 = new Complex(3.0, 4.0);
    complex c1 = new Complex(5.0, 6.0);

    complex d = c.multiplyAdd(a1, b1);
}
```

where

```java
complex multiplyAdd(complex a, complex b)
{
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return sum;
}
```
Mapping Algorithm: Rule 4

Map each outside edge in the incoming graph to the corresponding inside edge in the old graph:

```java
function function1(Obj n1)
{
    Obj n2 = new Obj();
    n2 = n1.f;
    ...
}

...
public static void main(String[] args)
{
    Obj n3 = new Obj();
    Obj n4 = new Obj();

    n3.f = n4;
    function1(n3);
    ...
    n3 → f → n4
}
```
Mapping Algorithm: Rule 4

Map each outside edge in the incoming graph to the corresponding inside edge in the old graph:

```
function function1(Obj n1)
{
    Obj n2 = new Obj();
    n2 = n1.f;
    ...
}

... public static void main(String[] args)
{
    Obj n3 = new Obj();
    Obj n4 = new Obj();
    n3.f = n4;
    function1(n3);
    ...
}
```
Mapping Algorithm: Rule 4

Map each outside edge in the incoming graph to the corresponding inside edge in the old graph:

```java
function function1(Obj n1)
{
    Obj n2 = new Obj();
    n2 = n1.f;
    ...
}

... public static void main(String[] args)
{
    Obj n3 = new Obj();
    Obj n4 = new Obj();

    n3.f = n4;
    function1(n3);
    ...
}
```
Map each outside edge in the incoming graph to the corresponding inside edge in the old graph:

```
function function1(Obj n1)
{
    Obj n2 = new Obj();
    n2 = n1.f;
    ...
}
...
public static void main(String[] args)
{
    Obj n3 = new Obj();
    Obj n4 = new Obj();
    n3.f = n4;
    function1(n3);
    ...
}
```
Mapping Algorithm: Rule 5

Map all inside nodes that are reachable from some node in the new graph:

```java
function function3()
{
    Obj n1 = new Obj();
    Obj n2 = new Obj();

    n1.f = n2;
    ...
    return n1;
}

...

public static void main(String[] args)
{
    Obj n = function3();

    Obj n2 = n.f;
    ...
    ...
}
Mapping Algorithm: Rule 5

Map all inside nodes that are reachable from some node in the new graph:

```
function function3()
{
    Obj n1 = new Obj();
    Obj n2 = new Obj();
    n1.f = n2;
    ...
    return n1;
}

...

public static void main(String[] args)
{
    Obj n = function3();
    Obj n2 = n.f;
    ...
}
```
Mapping Algorithm: Rule 5

Map all inside nodes that are reachable from some node in the new graph:

```java
function function3()
{
    Obj n1 = new Obj();
    Obj n2 = new Obj();
    n1.f = n2;
    ...
    return n1;
}

...
public static void main(String[] args)
{
    Obj n = function3();
    Obj n2 = n.f;
    ...
}
```
Mapping Algorithm: Rule 6

All nodes passed into unanalysed methods are marked in the escape function:

```java
function function4(Obj n1)
{
    ...
    function5(n1);
    ...
}

... public static void main(String[] args)
{
    Obj n = new Obj();
    function4(n);
    ...
}
```
Mapping Algorithm: Rule 6

All nodes passed into unanalysed methods are marked in the escape function:

```
function function4(Obj n1)
{
    ...
    function5(n1);
    ...
}
...
public static void main(String[] args)
{
    Obj n = new Obj();
    function4(n);
    ...
}
```

\[ e = \{(n1 \rightarrow function5(n1))\} \]
Results: Synchronization Elimination
Results: Stack Allocation

![Bar chart showing allocation results for javac, javacup, javalex, and pbob. javalex has the highest allocation, followed by javac, javacup, and pbob.]
Analysis Properties

- Interprocedural
- Compositional
- Partial
Related Works

- Pointer analysis
- Escape Analysis
- Synchronization Optimization
Now we are able:

- To analyse an object-oriented program with help of points-to escape graphs
- To analyse program parts independently
- To allocate up to 95% of objects in the stack
- To eliminate unnecessary synchronization up to 67%