Size Optimizations for Java Programs

C. Scott Ananian and Martin Rinard
{cananian, rinard}@lcs.mit.edu

Laboratory for Computer Science
Massachusetts Institute of Technology

LCTES’03, June 2003
Our Goal

Reduce the memory consumption of object-oriented programs

By

Using program analysis to identify opportunities to reduce the space required to store objects,

Then

Applying transformations to reduce the memory consumption of the program.
Structure of a Java Object

- Typical of many O-O languages.

```
object reference
```

```
claz pointer
hashcode/lock
field slot 0
field slot 1
field slot 2
...
padding
```

- object class description: inheritance information, method dispatch tables, etc.
- optional object info: persistent hashcode, monitor locks, "native" data
Strategy

Push hard on all the bits.
Strategy

Push hard on all the bits.
How to compress objects

Three broad techniques:

<table>
<thead>
<tr>
<th>claz pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashcode/lock</td>
</tr>
<tr>
<td>field slot 0</td>
</tr>
<tr>
<td>field slot 1</td>
</tr>
<tr>
<td>field slot 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>padding</td>
</tr>
</tbody>
</table>
How to compress objects

Three broad techniques:

- Field compression
How to compress objects

Three broad techniques:

- Field compression
- Mostly-constant field elimination
How to compress objects

Three broad techniques:

- Field compression
- Mostly-constant field elimination
- Header optimizations
How to compress objects

Three broad techniques:

- **Field compression**
- Mostly-constant field elimination
- Header optimizations
Field Compression

Reduce the space taken up by an object’s fields.

class Car {
    int color;
    ...
}

BLACK = 0
RED = 1
BLUE = 2
Field Compression

Reduce the space taken up by an object’s fields.

- **Sparse Predicated Typed Constant analysis** to discover unread/unused/constant fields.

```java
class Car {
    int color;
    ...
}
```
Field Compression

Reduce the space taken up by an object’s fields.

- **Sparse Predicated Typed Constant analysis** to discover unread/unused/constant fields.

```java
class Car {
    int color;
    ...
}
BLACK = 0
```
Field Compression

Reduce the space taken up by an object’s fields.

- Sparse Predicated Typed Constant analysis to discover unread/unused/constant fields.
- Bitwidth analysis to discover tight upper bounds on field size.

```java
class Car {
    int color;
    ...
}
BLACK=0
```
Field Compression

Reduce the space taken up by an object’s fields.

- **Sparse Predicated Typed Constant analysis** to discover unread/unused/constant fields.
- **Bitwidth analysis** to discover tight upper bounds on field size.

```java
class Car {
    int color;
    ...
}

BLACK=0  RED=1  BLUE=2
```
Field Compression

Reduce the space taken up by an object’s fields.

- **Sparse Predicated Typed Constant analysis** to discover unread/unused/constant fields.
- **Bitwidth analysis** to discover tight upper bounds on field size.
- **Field packing** into bytes or bits.

```java
class Car {
    int color;
    ...
}
```

BLACK=0  RED=1  BLUE=2
How are these analyses performed?
Intraprocedural Analysis

```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        //
        //
    }
}
```
int foo() {  
    if (...)  
        i=1;  
    else  
        i=2;  
    if (i>0)  
        ...  
}
Intraprocedural Analysis

```c
int foo() {
    if (...) 
        i=1;
    else 
        i=2;
    if (i>0) 
        ::
}
```

![Diagram showing the analysis of the function with possible paths]

\[ i = \bot \]
Intraprocedural Analysis

```c
int foo() {
    if (...) {
        i = 1;
    } else {
        i = 2;
    }
    if (i > 0) {
        ...
    }
}
```

The diagram represents a control flow graph (CFG) for the given function. The CFG shows the possible paths through the function, including conditional branches and the assignment of `i`. The value of `i` is inferred to be `⊥` (undefined) under certain conditions based on the control flow and the assignments made within the function.
Intraprocedural Analysis

```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        ...
    }
}
```

```
i = ⊥ \land 1
```
Intraprocedural Analysis

```c
int foo() {
    if (...
        i=1;
    else
        i=2;
    if (i>0)
        ...
    
}

[Because ⊥ ⊆ 1 and 1 ⊆ 1]
```
int foo() {
    if (...
        i=1;
    else
        i=2;
    if (i>0)
        ...
    ...
}
Intraprocedural Analysis

```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        ...
    }
}
```

\[
i = \{1 \land 2\}
\]
Intraprocedural Analysis

```c
int foo() {
    if (...) i=1;
else
    i=2;
    if (i>0) ...
    :
}
```

\[ i = 1 \sqcap 2 = \top \]

[Because \( 1 \sqsubseteq \top \) and \( 2 \sqsubseteq \top \)]
Intraprocedural Analysis

```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        ...
    }
}
```

**Diagram:**

```
     0
    / \
   1   2
  / \ / \ \
 ...-1-2...
```

**Equation:**

\[
i = 1 \sqcap 2 = \top
\]

**Explanation:**

[Because \(1 \subseteq \top\) and \(2 \subseteq \top\)]
A signed integer lattice

An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain.
An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain.
An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain.
A signed integer lattice

An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain.
Example, redux

```c
int foo() {
    if (...) {
        i = 1;
    }
    else {
        i = 2;
    }
    if (i > 0) {
        ...
    }
}
```

```
i = \bot
```
Example, redux

int foo() {
    if (...) 
        i=1;
    else 
        i=2;
    if (i>0) 
        ;
}

i = \bot
Example, redux

```c
int foo() {
    if (...) 
        i=1;
    else 
        i=2;
    if (i>0)
        : ;
}
```

```
i = ⊥ □ 1
```
```c
int foo() {
    if (...) 
        i=1;
    else 
        i=2;
    if (i>0) 
        : 
    }

```

Example, redux

![Diagram](image)

```
\textbf{i=1}
```
Example, redux

```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        ...
    }
}
```

```plaintext
i = 1
```
int foo() {
    if (...)  
        i=1;
    else
        i=2;
    if (i>0)  
        ...;
    }

\[
i = 1 \cap 2
\]
Example, redux

```c
int foo() {
    if (...) {
        i = 1;
    } else {
        i = 2;
    }
    if (i > 0) {
        ...
    }
}
```

\[
i = 1 \sqcap 2 = (\_\_P)
\]
Example, redux

```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        ...
    }
}
```

\[ i = 1 \sqcap 2 = (__P) \]
Extending the lattice

Replace $\mathfrak{m}$ and $\mathfrak{p}$ in previous lattice entries with positive integers $m$ and $p$. Encode zero as $m = p = 0$. 
Extending the lattice

Replace $\mathcal{M}$ and $\mathcal{P}$ in previous lattice entries with positive integers $m$ and $p$. Encode zero as $m = p = 0$.

$$(\mathcal{P}) \Rightarrow \langle 0, p \rangle$$
$$(\mathcal{M}_\varepsilon) \Rightarrow \langle m, 0 \rangle$$
$$(\mathcal{Z}_\varepsilon) \Rightarrow \langle 0, 0 \rangle$$
Extending the lattice

Replace $M$ and $P$ in previous lattice entries with positive integers $m$ and $p$. Encode zero as $m = p = 0$.

\[
\begin{align*}
   (^{__}P) & \Rightarrow \langle 0, p \rangle \\
   (^M__) & \Rightarrow \langle m, 0 \rangle \\
   (^Z__) & \Rightarrow \langle 0, 0 \rangle
\end{align*}
\]

In lattice context:

\[
\begin{align*}
   (^{__}P) & \Rightarrow \langle 0, 3 \rangle \\
   & \Rightarrow \langle 0, 2 \rangle \\
   & \Rightarrow \langle 0, 1 \rangle
\end{align*}
\]
Bitwidth lattice detail

\[ \langle 0, 31 \rangle \]
\[ \langle 0, 2 \rangle \]
\[ \langle 0, 1 \rangle \]
\[ 0 \]
\[ \langle 0, 0 \rangle \]

\[ \cdots \]

\[ 2^{32} - 1 \]
Example redux, redux

```c
int foo() {
    if (...) { i=1;
    } else { i=2;
    }
    if (i>0) {
    ...
    } } i=⊥
```

![Diagram of i=⊥ and a tree structure with nodes labeled MZP, MZ, MP, ZP, etc.](Ananian/LCTES’03 – p. 14)
Example redux, redux

```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        ...
    }
}
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        ...
    }
}
```

Diagram:

```
                 ⊤
                 |
                (MZP)
                /    |
           (MZ_)  (M_P)  (ZP)
              |
           (M_)   (P)
          /  |
   ...
   -2 -1 0 1 2 ...
```

```
i = \bot \sqcap 1
```
Example redux, redux

```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        ...
    }
}
```

```
        MZP
       /   \
(MZ_)  (M_P)  (ZP)
      /     \
(M__)  (__P)
    /   \
  (-2) -1 0 1 2 ...
 /   / \ \
(_Z_) (_Z_) ...
```

```
i = 1
```
Example redux, redux

```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        ...
    }
}
```

```
... –2 –1 0 1 2 ...
```

Ananian/LCTES’03 – p. 14
```
int foo() {
  if (...) {
    i=1;
  } else {
    i=2;
  }
  if (i>0) {
    ...
  }
}
```

```
i = 1 \in \{2\}
```
int foo() {
    if (...) 
        i=1;
    else 
        i=2;
    if (i>0) 
        ...
    }
Example redux, redux

```c
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0)
    :
}  

\[
i = 1 \sqcap 2 = \langle 0, 2 \rangle
\]
```
Bitwidth combination rules

\[- \langle m, p \rangle = \langle p, m \rangle\]
\[\langle m_l, p_l \rangle + \langle m_r, p_r \rangle = \langle 1 + \max(m_l, m_r), 1 + \max(p_l, p_r) \rangle\]
\[\langle m_l, p_l \rangle \times \langle m_r, p_r \rangle = \left\langle \begin{array}{c}
\max(m_l + p_r, p_l + m_r), \\
\max(m_l + m_r, p_l + p_r)
\end{array} \right\rangle\]
\[\langle 0, p_l \rangle \land \langle 0, p_r \rangle = \langle 0, \min(p_l, p_r) \rangle\]
\[\langle m_l, p_l \rangle \land \langle m_r, p_r \rangle = \langle \max(m_l, m_r), \max(p_l, p_r) \rangle\]

Some combination rules for bit-width analysis.
int foo() {
    if (...) 
        i=1;
    else
        i=2;
    if (i>0) 
        ...
    ...
}
int foo() {
    if (...
        a.f=1;
    else
        b.f=2;
    if (c.f>0)
        ;
    }
}

int bar() {
    this.f=2;
}

if (this.f>0)
int foo() {
    if (...) {
        a.f=1;
    } else {
        b.f=2;
    }
    if (c.f>0) {
        ...
    }
}

int bar() {
    this.f=1;
}

int bar() {
    this.f=2;
}

int bar() {
    if (this.f>0) {
        ...
    }
}
All cars are black

void paint(int color) {
    if (this.model == FORD)
        color = BLACK;
    this.color = color;
}

Ananian/LCTES ’03 – p. 17
Field compression using bitwidths
Field packing

Standard packing word-aligns the object and aligns each field to the width of its type (4-byte data is 4-byte aligned):

```
X | Y | Z
```

- X (24 bits)
- Y (5 bits)
- Z (1 bit)

“Byte” alignment byte-aligns the object and all fields:

```
X | Y | Z
```

- X (24 bits)
- Y (5 bits)
- Z (1 bit)

“Bit” alignment requires no alignment of objects or fields:

```
X | Y | Z
```

- X (24 bits)
- Y (5 bits)
- Z (1 bit)

Object header omitted.
How to compress objects

Three broad techniques:

- Field compression
- Mostly-constant field elimination
- Header optimizations
Mostly-constant field elimination

• It’s easy to remove constant fields.
Mostly-constant field elimination

- It’s easy to remove constant fields.
- Key idea: remove mostly constant fields.
Mostly-constant field elimination

- It’s easy to remove constant fields.
- Key idea: remove mostly constant fields.
  - Identify fields which have a certain value “most of the time.”
 Mostly-constant field elimination

- It’s easy to remove constant fields.
- Key idea: remove mostly constant fields.
  - Identify fields which have a certain value “most of the time.”
  - Static analysis/profiling.
Mostly-constant field elimination

- It’s easy to remove constant fields.
- Key idea: remove mostly constant fields.
  - Identify fields which have a certain value “most of the time.”
  - Static analysis/profiling.
- Transform objects to remove fields w/ the common value.
Mostly-constant field elimination

- It’s easy to remove constant fields.
- Key idea: remove *mostly* constant fields.
  - Identify fields which have a certain value “most of the time.”
    - Static analysis/profiling.
  - Transform objects to remove fields w/ the common value.
    - Static specialization/externalization.
public final class String {
    private final char value[];
    private final int offset;
    private final int count;
    ...
    public char charAt(int i) {
        return value[offset+1];
    }
    public String substring(int start) {
        int noff = offset + start;
        int ncnt = count - start;
        return new String(value, noff, ncnt);
    }
}
Key properties

To use static specialization we need:

- A field with a frequently-occurring value.
  - `String.offset` is almost always zero.
- The value of the field must never be modified after the object is created.
Transforming the class

We will split String into two classes:

- `SmallString` without the field.
- `BigString` with the field.

We will use `SmallString` for all instances where the offset field is zero (our “mostly-constant” value).
Transforming the class

We will split String into two classes:

- **SmallString** without the field.
- **BigString** with the field.

We will use **SmallString** for all instances where the offset field is zero (our “mostly-constant” value).

Problems:
Transforming the class

We will split String into two classes:

- SmallString without the field.
- BigString with the field.

We will use SmallString for all instances where the offset field is zero (our “mostly-constant” value).

Problems:

- The code could directly access the to-be-removed field.
Transforming the class

We will split String into two classes:

- `SmallString` without the field.
- `BigString` with the field.

We will use `SmallString` for all instances where the offset field is zero (our “mostly-constant” value).

Problems:

- The code could directly access the to-be-removed field.
- Allocation sites directly instantiate the old class.
public final class String {
    private final char value[];
    private final int offset;
    private final int count;

    ...

    public char charAt(int i) {
        return value[offset+i];
    }

    public String substring(int start) {
        int noff = offset + start;
        int ncnt = count - start;
        return new String(value, noff, ncnt);
    }
}
Specialization example: java.lang.String

```java
public final class SmallString {
    private final char value[];
    private final int offset;
    private final int count;

    public char charAt(int i) {
        return value[offset+1];
    }

    public String substring(int start) {
        int noff = offset + start;
        int ncnt = count - start;
        return new String(value, noff, ncnt);
    }
}
```
Specialization example: java.lang.String

public final class SmallString {
    private final char value[];
    private final int offset;
    private final int count;
    protected int getOffset() { return 0; }
    ...
    public char charAt(int i) {
        return value[getOffset() + 1];
    }
    public String substring(int start) {
        int noff = getOffset() + start;
        int ncnt = count - start;
        return new String(value, noff, ncnt);
    }
}
Specialization example: java.lang.String

```java
public final class SmallString {
    private final char value[];
    private final int count;
    protected int getOffset() { return 0; }
    ...
    public char charAt(int i) {
        return value[getOffset()+i];
    }
    ...
}
public final class BigString extends SmallString {
    private final int offset;
    protected int getOffset() { return offset; }
}
```
Transforming allocation sites

Case 1: field is constant in constructor.

String s = new String (new char[] {'a', 'b', 'c'});

String (char[] val) {
    this.value = (char[]) val.clone();
    this.offset = 0;
    this.count = val.length;
}
Transforming allocation sites

Case 1: field is constant in constructor.

SmallString s = new SmallString(new char[] {'a', 'b', 'c'});

SmallString(char[] val) {
    this.value = (char[]) val.clone();
    this.offset = 0;
    this.count = val.length;
}
Case 2: field is simple function of constructor parameter.

```java
String s = new String(new char[] {'a', 'b', 'c'},
    x, 1);

String(char[] val, int offset, int length) {
    this.value = (char[]) val.clone();
    this.offset = offset;
    this.count = length;
}
```
Transforming allocation sites

Case 2: field is simple function of constructor parameter.

SmallString s;

if (x==0)
    s = new SmallString(new char[] {'a', 'b', 'c'}, x, 1);
else
    s = new BigString(new char[] {'a', 'b', 'c'}, x, 1);
Case 3: assignment to field is unknown.

```java
String s = new String (s, o, l);

String (char[] val, int offset, int length) {
    this.value = (char[]) val.clone();
    while (length>0 && value[offset]==' ') {
        offset++; length-;
    }
    this.offset = offset;
    this.count = length;
}
```
Transforming allocation sites

Case 3: assignment to field is unknown.

```java
BigString s = new BigString(s, o, l);

BigString(char[] val, int offset, int length) {
    this.value = (char[]) val.clone();
    while (length>0 && value[offset]==' ') {
        offset++; length-;
    }
    this.offset = offset;
    this.count = length;
}
```
Static specialization

- Split class implementations into “field-less” and “field-ful” versions.
- Use virtual accessor functions to hide this split from users of the class.
- Can be done recursively on multiple fields.
  - Profiling guides splitting order if there are multiple candidates.
- Done at compile time, on fields which can be shown to be compile-time constants, thus “static.”
  - Fields can not be mutated after the constructor completes.
Key properties (revisited)

To use static specialization we need:

- A field with a frequently-occurring value.
  - `String.offset` is almost always zero.
- The value of the field must never be modified after the object is created.
Key properties (revisited)

To use static specialization we need:

- A field with a frequently-occurring value.
  - `String.offset` is almost always zero.
- The value of the field must never be modified after the object is created.
Creating external fields

• Sometimes fields are *run-time* constants (or nearly so) but not *compile-time* constants.
Creating external fields

- Sometimes fields are *run-time* constants (or nearly so) but not *compile-time* constants.
- Examples: sparse matrices, “two-input nodes” in Jess expert system, the “next” field in short linked lists.
Creating external fields

- Sometimes fields are *run-time* constants (or nearly so) but not *compile-time* constants.
  - Examples: sparse matrices, “two-input nodes” in Jess expert system, the “next” field in short linked lists.
- **Exploit field→map duality** to reduce memory overhead in the common case.
Fields and Maps

- Accessing an object field \( a.b \) (where \( a \) is the object reference and \( b \) is the field name) is equivalent to evaluating a map from \( \langle a, b \rangle \) to the value type.

Ananian/LCTES’03 – p. 33
Fields and Maps

- Accessing an object field $a . b$ (where $a$ is the object reference and $b$ is the field name) is equivalent to evaluating a map from $\langle a, b \rangle$ to the value type.

- The mapping we will implement will be *incomplete*. We define the result of accessing a non-existing mapping to be $\bot$. 

Ananian/LCTES'03 – p. 33
Fields and Maps

- Accessing an object field $a . b$ (where $a$ is the object reference and $b$ is the field name) is equivalent to evaluating a map from $\{a, b\}$ to the value type.

- The mapping we will implement will be incomplete. We define the result of accessing a non-existing mapping to be $\bot$.

- To achieve our storage savings, we map $\bot$ to the frequent “mostly-constant” value.
public final class String {
    private final char value[];
    private final int offset;
    private final int count;

    public char charAt(int i) {
        return value[offset + 1];
    }

    public String substring(int start) {
        int noff = offset + start;
        int ncnt = count - start;

        return new String(value, noff, ncnt);
    }
}

public final class String {
    private final char value[];
    private final int offset;
    private final int count;
    public char charAt(int i) {
        return value[offset+1];
    }
    public String substring(int start) {
        int noff = offset + start;
        int ncnt = count - start;
        return new String(value, noff, ncnt);
    }
}
public final class String {
    private final char value[];
    private final int offset;
    private final int count;
    public char charAt(int i) {
        return value[getOffset()+1];
    }
    public String substring(int start) {
        int noff = getOffset() + start;
        int ncnt = count - start;
        return new String(value, noff, ncnt);
    }
    protected int getOffset() {
        Integer i = External.map.get(this, "offset");
        if (i==null) return 0;
        else return i.intValue();
    }
}
External map implementation

- “open addressed” for low overhead.
External map implementation

Open–addressed Hashtable

<table>
<thead>
<tr>
<th>Object</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Key | Value

- “open addressed” for low overhead.
- load-factor of 2/3
External map implementation

Open–addressed Hashtable

<table>
<thead>
<tr>
<th>Object</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object</td>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- “open addressed” for low overhead.
- load-factor of 2/3
- two-word key and one-word values means break-even point is 82%
External map implementation

- "open addressed" for low overhead.
- load-factor of 2/3
- two-word key and one-word values means break-even point is 82%
  (i.e. field may not differ from the "mostly-constant" value in more than 18% of objects.)
We can do better!

- Use small integers to enumerate fields.

Open–addressed Hashtable

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Ananian/LCTES’03 – p. 36
We can do better!

- Use small integers to enumerate fields.
- Offset the object pointer by the field index to get a 1-word key.

Open–addressed Hashtable

<table>
<thead>
<tr>
<th>Object + Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
</tbody>
</table>

Key | Value
---|---
We can do better!

- Use small integers to enumerate fields.
- Offset the object pointer by the field index to get a 1-word key.
- Limits the number of fields which may be externalized, based on the size of the object.
We can do better!

- Use small integers to enumerate fields.
- Offset the object pointer by the field index to get a 1-word key.
- Limits the number of fields which may be externalized, based on the size of the object.
- One-word key and one-word value lowers break-even point to 66%.

Open–addressed Hashtable

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Other details

- Use value profiling to identify classes where field externalization will be worthwhile.
Other details

- Use value profiling to identify classes where field externalization will be worthwhile.
- In our experiments, looked for integer “mostly-constant” values in the range \([-5, 5]\) for numeric types. Only looked at `null` as a candidate for pointer types.
Other details

- Use value profiling to identify classes where field externalization will be worthwhile.
- In our experiments, looked for integer “mostly-constant” values in the range $[-5, 5]$ for numeric types. Only looked at null as a candidate for pointer types.
- 0 and 1 by far the most common.
How to compress objects

Three broad techniques:

- Field compression
- Mostly-constant field elimination
- Header optimizations
Header optimizations:
Hashcode/Lock compression

<table>
<thead>
<tr>
<th>claz pointer</th>
<th>claz pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashcode/lock</td>
<td>hashcode/lock</td>
</tr>
<tr>
<td>field slot 0</td>
<td>field slot 0</td>
</tr>
<tr>
<td>field slot 1</td>
<td>field slot 1</td>
</tr>
<tr>
<td>field slot 2</td>
<td>field slot 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>padding</td>
<td>padding</td>
</tr>
</tbody>
</table>

Ananian/LCTES’03 – p. 39
Header optimizations:
Hashcode/Lock compression

- Implemented as a special case of field externalization.
Header optimizations: Hashcode/Lock compression

- Implemented as a special case of field externalization.
- The hashcode/lock field often unused because:
  - Most objects do not use their built-in hashcode.
  - Most objects are not synchronization targets.
Header optimizations:
Hashcode/Lock compression

- Implemented as a special case of field externalization.
- The hashcode/lock field often unused because:
  - Most objects do not use their built-in hashcode.
  - Most objects are not synchronization targets.
- Could also use a static pointer analysis.
Header optimizations:
claz compression

<table>
<thead>
<tr>
<th>claz pointer</th>
<th>claz pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashcode/lock</td>
<td>hashcode/lock</td>
</tr>
<tr>
<td>field slot 0</td>
<td>field slot 0</td>
</tr>
<tr>
<td>field slot 1</td>
<td>field slot 1</td>
</tr>
<tr>
<td>field slot 2</td>
<td>field slot 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>padding</td>
<td>padding</td>
</tr>
</tbody>
</table>
Header optimizations:
claz compression

claz pointer
hashcode/lock
field slot 0
field slot 1
field slot 2
...
padding

claz pointer
hashcode/lock
field slot 0
field slot 1
field slot 2
...
padding

claz table
Header optimizations:
claz compression

- replace claz pointer with a (smaller) table index.
Header optimizations:

claz compression

- replace claz pointer with a (smaller) table index.
- With co-operation of GC, works in dynamic environments.
Header optimizations:
claz compression

- replace claz pointer with a (smaller) table index.
- With co-operation of GC, works in dynamic environments.
- Many applications use less than 256 object types.
Class statistics

Class statistics for applications in SPECjvm98 benchmark suite:
How well does it work?
Reduction in total allocations

SPECjvm98 Benchmarks

Percent of Total (Object and Array) Bytes Allocated

- Claz compression
- Field Reduction
- Static Specialization
- Field Externalization
- Hash/Lock Externalization
- Other

Ananian/LCTES’03 – p. 45
Reduction in total live data

SPECjvm98 Benchmarks

- Claz compression
- Field Reduction
- Static Specialization
- Field Externalization
- Hash/Lock Externalization
- Other
Available reduction opportunities

![Bar chart showing available reduction opportunities for various benchmarks. The chart indicates the percentage of total dynamic allocation contributed by different categories: other object fields, pointer fields, and array allocations.]
Reduction in object allocations

SPECjvm98 Benchmarks

- Claz compression
- Field Reduction
- Static Specialization
- Field Externalization
- Hash/Lock Externalization
- Other

Percent of Object Bytes Allocated

201_compress 202_jess 205_raytrace 209_db 213_javac 222_mpegaudio 227_mtrt 228_jack
Moderate performance impact

Execution time, normalized to no-optimization case

SPECjvm98 Benchmarks

Claz Compression
- add Field Reduction to previous
- add Byte Packing to previous
- add Static Specialization to previous
- add Field Externalization to previous
- add Hash/Lock Externalization to previous

Ananian/LCTES’03 – p. 49
How can we make this even better?

Currently no array analysis/can't distinguish between different uses of a class.

Use pointer analysis to discriminate among objects by allocation site; optimize each alloc site.

We hardly compress pointers at all.

Investigate region-based/enumerated approaches.

Zhang, Gupta (ICCC '02)

The mostly-constant analysis requires profiling.

Investigate heuristic methods.

Leverage dynamic profiling; identify cold fields.

We know nothing about "eld-like" maps.

Enable internalization.

Ananian/LCTES'03 – p. 50
How can we make this even better?

- Currently no array analysis/can’t distinguish between different uses of a class.
How can we make this even better?

- Currently no array analysis/can’t distinguish between different uses of a class.
  - Use pointer analysis to discriminate among objects by allocation site; optimize each alloc site.

- Investigate region-based/enumerated approaches.
- Zhang, Gupta (ICCC '02)
- The mostly-constant analysis requires profiling.
  - Investigate heuristic methods.
  - Leverage dynamic profiling; identify cold fields.
- We know nothing about “eld-like” maps.
  - Enable internalization.
How can we make this even better?

- Currently no array analysis/can’t distinguish between different uses of a class.
  - Use pointer analysis to discriminate among objects by allocation site; optimize each alloc site.
- We hardly compress pointers at all.
How can we make this even better?

- Currently no array analysis/can’t distinguish between different uses of a class.
  - Use pointer analysis to discriminate among objects by allocation site; optimize each alloc site.
- We hardly compress pointers at all.
  - Investigate region-based/enumerated approaches.

Zhang, Gupta (ICCC ’02)

The mostly-constant analysis requires profiling.

Investigate heuristic methods.

Leverage dynamic profiling; identify cold fields.

We know nothing about “field-like” maps.

Enable internalization.

Ananian/LCTES’03 – p. 50
How can we make this even better?

- Currently no array analysis/can’t distinguish between different uses of a class.
  - Use pointer analysis to discriminate among objects by allocation site; optimize each alloc site.
- We hardly compress pointers at all.
  - Investigate region-based/enumerated approaches.
  - Zhang, Gupta (ICCC ’02)
How can we make this even better?

- Currently no array analysis/can’t distinguish between different uses of a class.
  - Use pointer analysis to discriminate among objects by allocation site; optimize each alloc site.
- We hardly compress pointers at all.
  - Investigate region-based/enumerated approaches.
  - Zhang, Gupta (ICCC ’02)
- The mostly-constant analysis requires profiling.
How can we make this even better?

- Currently no array analysis/can’t distinguish between different uses of a class.
  - Use pointer analysis to discriminate among objects by allocation site; optimize each alloc site.
- We hardly compress pointers at all.
  - Investigate region-based/enumerated approaches.
  - Zhang, Gupta (ICCC ’02)
- The mostly-constant analysis requires profiling.
  - Investigate heuristic methods.
How can we make this even better?

- Currently no array analysis/can’t distinguish between different uses of a class.
  - Use pointer analysis to discriminate among objects by allocation site; optimize each alloc site.
- We hardly compress pointers at all.
  - Investigate region-based/enumerated approaches.
  - Zhang, Gupta (ICCC ’02)
- The mostly-constant analysis requires profiling.
  - Investigate heuristic methods.
  - Leverage dynamic profiling; identify cold fields.
How can we make this even better?

- Currently no array analysis/can’t distinguish between different uses of a class.
  - Use pointer analysis to discriminate among objects by allocation site; optimize each alloc site.
- We hardly compress pointers at all.
  - Investigate region-based/enumerated approaches.
  - Zhang, Gupta (ICCC ’02)
- The mostly-constant analysis requires profiling.
  - Investigate heuristic methods.
  - Leverage dynamic profiling; identify cold fields.
- We know nothing about “field-like” maps.
How can we make this even better?

- Currently no array analysis/can’t distinguish between different uses of a class.
  - Use pointer analysis to discriminate among objects by allocation site; optimize each alloc site.
- We hardly compress pointers at all.
  - Investigate region-based/enumerated approaches.
  - Zhang, Gupta (ICCC ’02)
- The mostly-constant analysis requires profiling.
  - Investigate heuristic methods.
  - Leverage dynamic profiling; identify cold fields.
- We know nothing about “field-like” maps.
  - Enable internalization.
Related Work

- Reducing lock overhead.
  - Bacon, Sweeney (OOPSLA ’96)
  - Onodera, Kawachiya (OOPSLA ’99)
  - Agesen, Detlefs, Garthwaite, Knippel, Ramakrishna, White (OOPSLA ’99)

- Escape analysis.
  - Aldrich, Chambers, Sirer, Eggers (SAS ’99)
  - Bogda, Hözle (OOPSLA ’99)
  - Whaley, Rinard (OOPSLA ’99)
  - Choi, Gupta, Serrano, Sreedhar, Midkiff (OOPSLA ’99)
  - Ruf (PLDI ’00)
  - Sǎlcianu, Rinard (PPoPP ’01)
Related Work II

- Space and time usage of Java programs.
  - Dieckmann, Hölzle (ECOOP ’99)
  - Bacon, Fink, Grove (ECOOP ’02)

- Bitwidth Analyses
  - Ananian (MIT ’99)
  - Rugina, Rinard (PLDI ’00)
  - Stephenson, Babb, Amarasinghe (PLDI ’00)
  - Budiu, Sakr, Walker, Goldstein (Europar ’00)

- Dead members in C++
  - Sweeney, Tip (PLDI ’98)
Conclusions

We identified a variety of opportunities for space reductions in object-oriented programs. We described analyses and transformations to exploit these opportunities. We achieved substantial space savings on typical object-oriented applications. In one case, over 40% reduction in total live data. Even more space reduction is possible! Performance impact was acceptable and tunable.
Conclusions

• We identified a variety of opportunities for space reductions in object-oriented programs.
Conclusions

• We identified a variety of opportunities for space reductions in object-oriented programs.
• We described analyses and transformations to exploit these opportunities.
Conclusions

• We identified a variety of opportunities for space reductions in object-oriented programs.
• We described analyses and transformations to exploit these opportunities.
• We achieved substantial space savings on typical object-oriented applications.
Conclusions

- We identified a variety of opportunities for space reductions in object-oriented programs.
- We described analyses and transformations to exploit these opportunities.
- We achieved substantial space savings on typical object-oriented applications.
  - In one case, over 40% reduction in total live data.
Conclusions

- We identified a variety of opportunities for space reductions in object-oriented programs.
- We described analyses and transformations to exploit these opportunities.
- We achieved substantial space savings on typical object-oriented applications.
  - In one case, over 40% reduction in total live data.
- Even more space reduction is possible!
Conclusions

• We identified a variety of opportunities for space reductions in object-oriented programs.

• We described analyses and transformations to exploit these opportunities.

• We achieved substantial space savings on typical object-oriented applications.
  • In one case, over 40% reduction in total live data.

• Even more space reduction is possible!

• Performance impact was acceptable and tunable.
Size Optimizations for Java Programs

FLEX homepage
http://flex-compiler.lcs.mit.edu

This talk:
http://flex-compiler.lcs.mit.edu/Harpoon/papers.html
The Graveyard Of Unused Slides follows this point.
Bitwidth analysis

Motivation:
  • Tedious and error-prone for programmer to manually specify widths.

```c
struct foo {
    int x:24;
    int y:5;
    int z:1;
};
```
Bitwidth analysis

Motivation:

- Tedious and error-prone for programmer to manually specify widths.

```c
struct foo { 
    void foo() {
        int x:24; int x:24;
        int y:5; int y:5;
        int z:1; int z:1;
    };
    ...
}
```
Bitwidth analysis

Motivation:

• Tedious and error-prone for programmer to manually specify widths.

```c
struct foo {
    void foo() {
        int x:24;
        int y:5;
        int z:1;
    }
}
```

• The compiler can do it for us!