Size Optimizations for Java Programs

C. Scott Ananian

cananian@lcs.mit.edu

Laboratory for Computer Science
Massachusetts Institute of Technology
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.

<table>
<thead>
<tr>
<th>object reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>claz pointer</td>
</tr>
<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>

- 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.

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

```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;
    ...
}
```
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.

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

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

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

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

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

![Diagram of a control flow graph with nodes labeled 0, 1, 2, and an edge to a sink node labeled i = \( \bot \).]
Intraprocedural Analysis

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

\[ i = \bot \sqcap 1 \]
Intraprocedural Analysis

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

[Because $\perp \sqsubseteq 1$ and $1 \sqsubseteq 1$]
Intraprocedural Analysis

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

Diagram:
```
          /
         /
        /  /
       /    
      /      
     /        
    /         
   /          
  /           
 /            
| i = 1      |
```

Size Optimizations for Java Programs – p.9
Intraprocedural Analysis

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

\[
i = \begin{cases}
1 & \text{if } i > 0 \\
2 & \text{otherwise}
\end{cases}
\]
```
Intraprocedural Analysis

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

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

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

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

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

[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.
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

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

Size Optimizations for Java Programs – p.1
```java
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        
    }
}
```
int foo() {
    if (...) {
        i = 1;
    } else {
        i = 2;
    }
    if (i > 0) {
    ::= 
    }
}

\[ i = \bot \sqcup 1 \]
Example, redux

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

![Diagram showing a tree structure with nodes labeled ZP, MP, MZ, and M. The root node is labeled ZP with children labeled Z, M, and P. The node labeled i=1 is highlighted.]
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        :
    }

\[
i = 1
\]
Example, redux

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

\[ i = 1 \boxplus 2 \]
```
int foo() {
    if (...) {
        i=1;
    } else {
        i=2;
    }
    if (i>0) {
        ...
    }
}

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

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

\[
i = 1 \land 2 = (\_P)
\]
Extending the lattice

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

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

\[
\begin{align*}
\text{(__P__)} & \Rightarrow \langle 0, p \rangle \\
\text{(M__)} & \Rightarrow \langle m, 0 \rangle \\
\text{(__Z__)} & \Rightarrow \langle 0, 0 \rangle
\end{align*}
\]
Extending the lattice

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

\[
\begin{align*}
(\_	exttt{P}) & \Rightarrow \langle 0, p \rangle \\
(\texttt{M}\_\_) & \Rightarrow \langle m, 0 \rangle \\
(\_\texttt{Z}\_) & \Rightarrow \langle 0, 0 \rangle
\end{align*}
\]

In lattice context: \( (\_	exttt{P}) \Rightarrow \langle 0, 3 \rangle \)

\[
\begin{align*}
\langle 0, 31 \rangle \\
\vdots \\
\langle 0, 2 \rangle \\
\langle 0, 1 \rangle
\end{align*}
\]
Bitwidth lattice detail

\[
\begin{array}{c}
\langle 0, 31 \rangle \\
\vdots \\
\langle 0, 2 \rangle \\
\langle 0, 1 \rangle \\
0 \\
\langle 0, 0 \rangle \\
\end{array}
\]

\[
\begin{array}{ccccccc}
& 1 & 2 & 3 & \cdots & 2^{32} - 1 \\
\langle 0, 0 \rangle & \downarrow & \downarrow & \downarrow & \cdots & \downarrow & \downarrow \\
\end{array}
\]
Example, redux redux

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

Size Optimizations for Java Programs – p.14
int foo() {
    if (...) {
        i=1;
    }
    else {
        i=2;
    }
    if (i>0) {
        
    }
}
Example, redux redux

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

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

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

Size Optimizations for Java Programs – p.14
int foo() {
    if (...) 
        i=1;
    else 
        i=2;
    if (i>0)
        : 
}
Example, redux redux

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

```
i = 1 \U00002227 2
```
Example, redux redux

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

\[ i \cap 2 = \langle 0, 2 \rangle \]
Example, redux redux

```java
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 (...) {
        this.f=1;
    } else {
        this.f=2;
    }
    if (this.f>0) {
        ;
    }
}
Interprocedural analysis

```java
int foo() {
    int f = 1;
    if (...) {
        this.f = 1;
    } else {
        this.f = 2;
    }
    if (this.f > 0) ...;
}

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

int bar() {
    if (this.f > 0) ...;
}
```

Size Optimizations for Java Programs – p.10
All cars are black

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

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
</table>

x (24 bits)  
y (5 bits) = (1 bit)

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

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
</table>

x (24 bits)  
y (5 bits) = (1 bit)

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

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
</table>

x (24 bits)  
y (5 bits) = (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.
Specialization example:
java.lang.String

```java
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.
Specialization example:
java.lang.String

```java
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);
    }
}
```
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

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; }
}

Size Optimizations for Java Programs – p.2
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;
}
Transforming allocation sites

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.

```java
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);
```
Transforming allocation sites

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.
- 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.
- Can be done recursively on multiple fields.
  - Profiling guides splitting order if there are multiple candidates.
Key properties (revisited)

To use static specialization we need:

- A field with a frequently-occuring 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.
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 \( \perp \).
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 $\bot$.

- 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 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>
</tbody>
</table>

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

- "open addressed" for low overhead.
- load-factor of 2/3

<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>

Key | Value
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%

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

Open-addressed Hashtable

Key | Value
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%
  (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.

<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>Object + Field</td>
<td>Value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
We can do better!

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

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>...</td>
<td>...</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.
- One-word key and one-word value lowers break-even point to 66%.

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

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

![Bar chart showing number of object classes for various benchmarks](chart.png)
How well does it work?
Reduction in total live data

SPECjvm98 Benchmarks
Available reduction opportunities

![Graph showing percentage of total dynamic allocation by category for various benchmarks.](image-url)

- **Available reduction opportunities**
- **Benchmarks**
- **% Total Dynamic Allocation**
  - Other object fields
  - Pointer fields
  - Array allocations

**Benchmarks**
- 201_compress
- 202_jess
- 205_raytrace
- 209_db
- 213_javac
- 222_mpegaudio
- 227_mtrt
- 228_jack

Size Optimizations for Java Programs – p.4
Reduction in object allocations

SPECjvm98 Benchmarks

Percent of Object Bytes Allocated

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

Size Optimizations for Java Programs – p.4
Moderate performance impact

SPECjvm98 Benchmarks

Execution time, normalized to no-optimization case

- 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

Size Optimizations for Java Programs – p.49
How can we make this even better?
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 from a certain allocation-site; optimize each alloc site.
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 from a certain allocation-site; optimize each alloc site.
- We don’t 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 from a certain allocation-site; optimize each alloc site.
- We don’t compress pointers at all.
  - Investigate region-based/enumerated approaches.
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 from a certain allocation-site; optimize each alloc site.
- We don’t compress pointers at all.
  - Investigate region-based/enumerated approaches.
- 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 from a certain allocation-site; optimize each alloc site.
- We don’t compress pointers at all.
  - Investigate region-based/enumerated approaches.
- 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 from a certain allocation-site; optimize each alloc site.
- We don’t compress pointers at all.
  - Investigate region-based/enumerated approaches.
- The mostly-constant analysis requires profiling.
  - Investigate heuristic methods.
- 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 from a certain allocation-site; optimize each alloc site.
- We don’t compress pointers at all.
  - Investigate region-based/enumerated approaches.
- The mostly-constant analysis requires profiling.
  - Investigate heuristic methods.
- We know nothing about “field-like” maps.
  - Enable internalization.
Conclusions
Conclusions

- We achieved substantial space savings on typical object-oriented applications.
Conclusions

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

- 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 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.
The Graveyard Of Unused Slides follows this point.
Available reduction opportunities

Total dynamic allocation (bytes)

Benchmarks

- 200_check
- 201_compress
- 202_jess
- 205_raytrace
- 209_db
- 213_javac
- 222_mpegaudio
- 227_mtrt
- 228_jack

Other object fields
Pointer fields
Array allocations
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.

```java
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.

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

- The compiler can do it for us!