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.

Why space optimizations?

Why space optimizations?

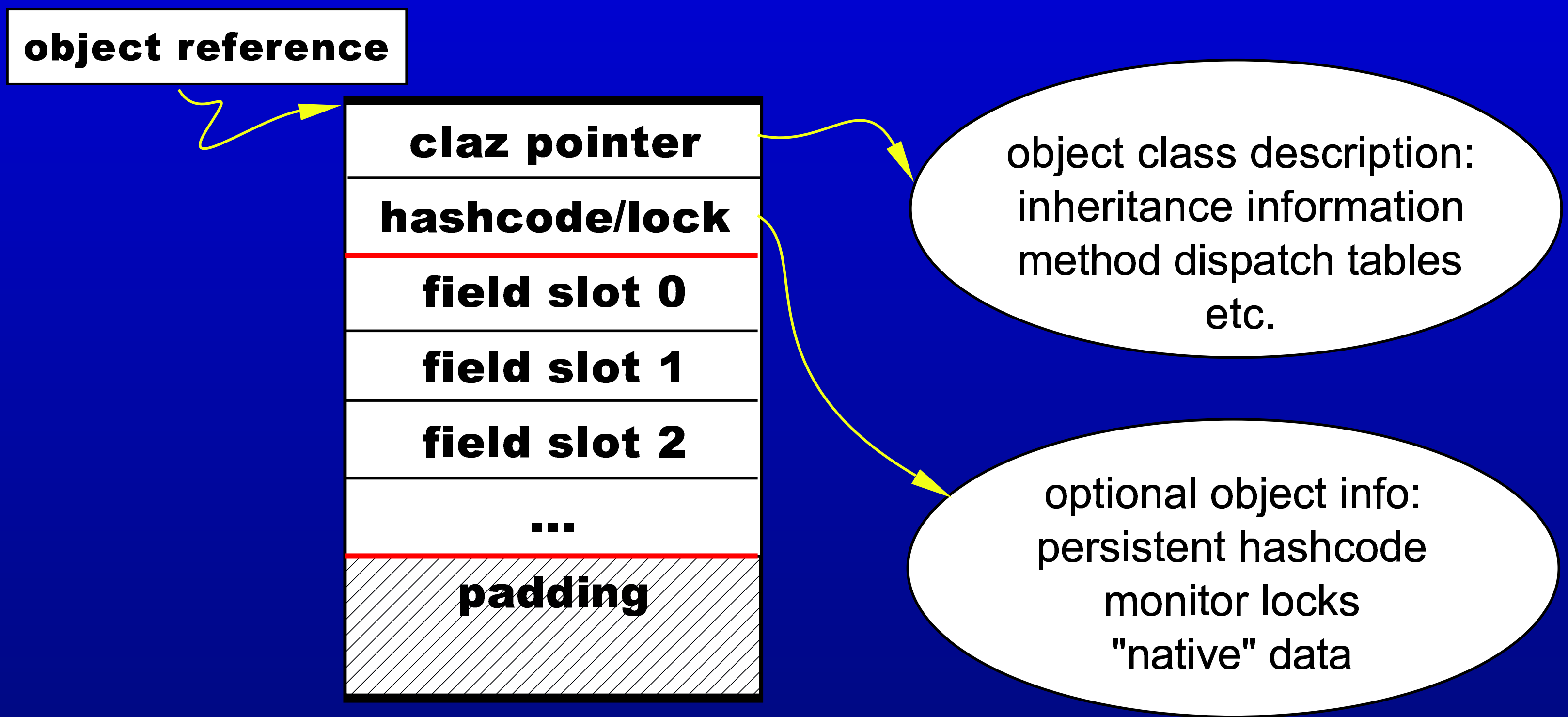
- Embedded applications:
  - Better use of existing fixed memory resources.
  - Reduce memory costs of new devices.

Why space optimizations?

- Embedded applications:
  - Better use of existing fixed memory resources.
  - Reduce memory costs of new devices.
- Performance:
  - "Memory wall" getting higher.
  - Space optimizations increase the effective cache size, improving performance.
  - Added ALU ops getting comparatively cheaper.

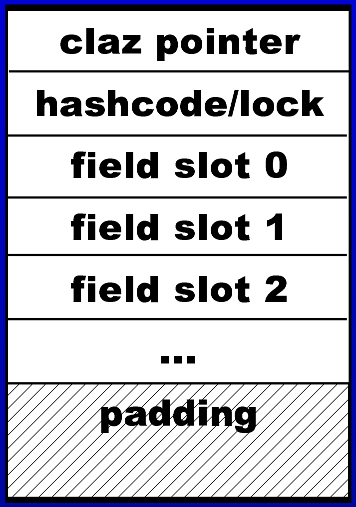
Structure of a Java Object

- Typical of many O-O languages.



How to compress objects

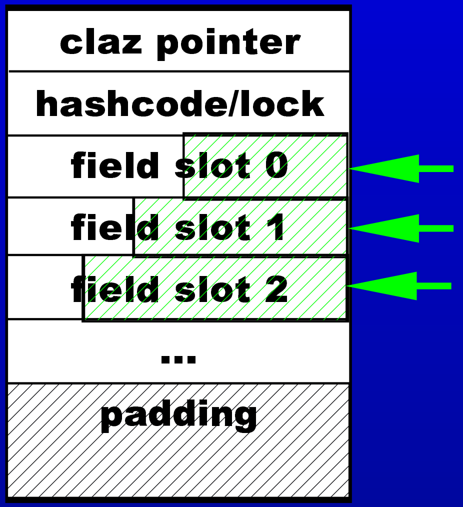
Three broad techniques:



How to compress objects

Three broad techniques:

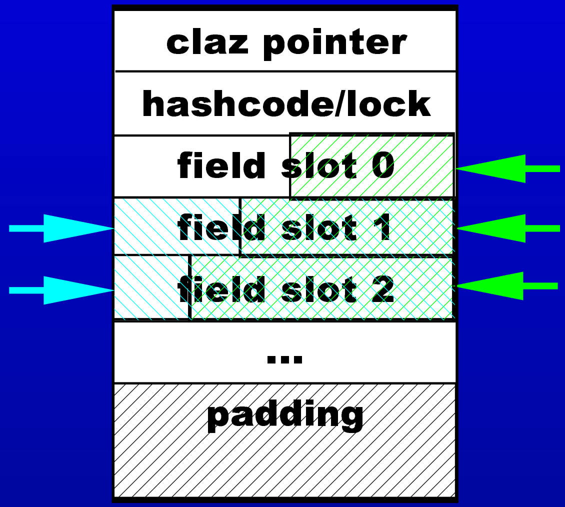
- Field compression



How to compress objects

Three broad techniques:

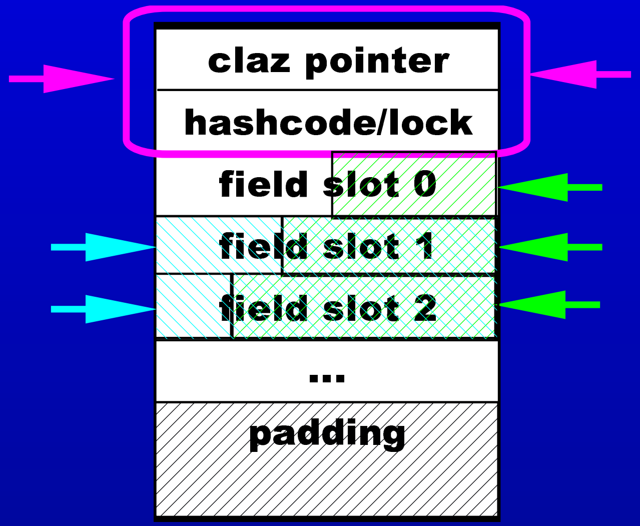
- Field compression
- Mostly-constant field elimination



How to compress objects

Three broad techniques:

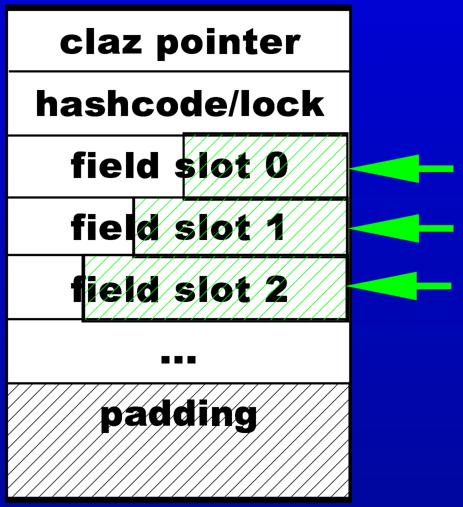
- Field compression
- Mostly-constant field elimination
- Header optimizations



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;
    ...
}
```

Field Compression

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

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

```
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.

```
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.

```
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.

```
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 overview

Intraprocedural overview

- Combined Sparse Predicated Typed Constant and Bitwidth analysis

Intraprocedural overview

- Combined Sparse Predicated Typed Constant and Bitwidth analysis
- Forward (sparse) dataflow algorithm

Intraprocedural overview

- Combined Sparse Predicated Typed Constant and Bitwidth analysis
- Forward (sparse) dataflow algorithm
- Discovers:
    - Executability of each control-flow edge (SPTC)
    - Program constants (SPTC)
    - Bitwidth specifications for all abstract values

Intraprocedural overview

- Combined Sparse Predicated Typed Constant and Bitwidth analysis
- Forward (sparse) dataflow algorithm
- Discovers:
  - Executability of each control-flow edge (SPTC)
  - Program constants (SPTC)
  - Bitwidth specifications for all abstract values
    - Number of bits in smallest negative number
    - Number of bits in largest positive number

Bitwidth analysis domains

Domains:

- $\mathcal{C} : \mathbb{Z}$, integer constants
  - $c \in \mathcal{C}$
- $\mathcal{T} : \mathbb{N}_0 \times \mathbb{N}_0$, bitwidth specifications ($\mathbb{N}_0 = \{0, 1, 2, \ldots\}$)
  - $\langle m, p \rangle \in \mathcal{T}$
- $\mathcal{L} : (\mathcal{C} \cup \mathcal{T})_\perp$, abstract value lattice

Bitwidth analysis domains

Domains:

- $\mathcal{C} : \mathbb{Z}$, integer constants
  - $c \in \mathcal{C}$
- $\mathcal{T} : \mathbb{N}_0 \times \mathbb{N}_0$, bitwidth specifications ($\mathbb{N}_0 = \{0, 1, 2, \ldots\}$)
  - $\langle m, p \rangle \in \mathcal{T}$
- $\mathcal{L} : (\mathcal{C} \cup \mathcal{T})_\perp$, abstract value lattice

Concretization: $\mathcal{L} \to 2^{\mathcal{Z}}$

$$\begin{aligned} \mathbb{C}\,[\perp] &= \emptyset \\ \mathbb{C}\,[\,c\,] &= \{c\} \\ \mathbb{C}\,[\langle m, p \rangle] &= \{n \,|{-2^m} < n < 2^p\} \end{aligned}$$

Ordering relationships in $\mathcal{L}$

For all $c \in \mathcal{C}$, $\langle m, p \rangle \in \mathcal{T}$:

$$\bot \sqsubseteq c \quad \text{and} \quad \bot \sqsubseteq \langle m, p \rangle$$
$$\langle m_1, p_1 \rangle \sqsubseteq \langle m_2, p_2 \rangle \quad \text{iff} \quad m_1 \leq m_2 \wedge p_1 \leq p_2$$
$$c \sqsubseteq \langle m, p \rangle \quad \text{iff} \quad \textbf{bw}(c) \sqsubseteq \langle m, p \rangle$$

Ordering relationships in $\mathcal{L}$

For all $c \in \mathcal{C}$, $\langle m, p \rangle \in \mathcal{T}$:

$$\begin{array}{rcl} \bot \sqsubseteq c & \text{and} & \bot \sqsubseteq \langle m, p \rangle \\ \langle m_1, p_1 \rangle \sqsubseteq \langle m_2, p_2 \rangle & \text{iff} & m_1 \leq m_2 \wedge p_1 \leq p_2 \\ c \sqsubseteq \langle m, p \rangle & \text{iff} & \textbf{bw}(c) \sqsubseteq \langle m, p \rangle \end{array}$$

where:

$$\textbf{bw}(c) : \mathcal{C} \rightarrow \mathcal{T} = \begin{cases} \langle 0, 0 \rangle & c = 0 \\ \langle 0, 1 + \lfloor \ln |c| \rfloor \rangle & c > 0 \\ \langle 1 + \lfloor \ln |c| \rfloor, 0 \rangle & c < 0 \end{cases}$$

Ordering relationships in $\mathcal{L}$

For all $c \in \mathcal{C}$, $\langle m, p \rangle \in \mathcal{T}$:

$$
\begin{array}{rcl}
\bot \sqsubseteq c & \text{and} & \bot \sqsubseteq \langle m, p \rangle \\
\langle m_1, p_1 \rangle \sqsubseteq \langle m_2, p_2 \rangle & \text{iff} & m_1 \leq m_2 \wedge p_1 \leq p_2 \\
c \sqsubseteq \langle m, p \rangle & \text{iff} & \textbf{bw}(c) \sqsubseteq \langle m, p \rangle
\end{array}
$$

where:

$$
\textbf{bw}(c) : \mathcal{C} \rightarrow \mathcal{T} = \begin{cases} \langle 0, 0 \rangle & c = 0 \\ \langle 0, 1 + \lfloor \ln|c| \rfloor \rangle & c > 0 \\ \langle 1 + \lfloor \ln|c| \rfloor, 0 \rangle & c < 0 \end{cases}
$$

$\mathtt{0...0}\overbrace{\mathtt{1X...XXX}}^{1+\lfloor \ln|c| \rfloor}$ $\qquad$ $\mathtt{1...1}\overbrace{\mathtt{0X...XXXX}}^{1+\lfloor \ln(|c|-1) \rfloor}$

Positive $\qquad$ Negative

Ordering relationships in $\mathcal{L}$

For all $c \in \mathcal{C}$, $\langle m, p \rangle \in \mathcal{T}$:

$$\begin{array}{rcl} \bot \sqsubseteq c & \text{and} & \bot \sqsubseteq \langle m, p \rangle \\ \langle m_1, p_1 \rangle \sqsubseteq \langle m_2, p_2 \rangle & \text{iff} & m_1 \leq m_2 \wedge p_1 \leq p_2 \\ c \sqsubseteq \langle m, p \rangle & \text{iff} & \mathbf{bw}(c) \sqsubseteq \langle m, p \rangle \end{array}$$

where:

$$\mathbf{bw}(c) : \mathcal{C} \to \mathcal{T} = \begin{cases} \langle 0, 0 \rangle & c = 0 \\ \langle 0, 1 + \lfloor \ln |c| \rfloor \rangle & c > 0 \\ \langle 1 + \lfloor \ln |c| \rfloor, 0 \rangle & c < 0 \end{cases}$$

$$\texttt{0...0}\,\overbrace{\texttt{1X...XXX}}^{1+\lfloor \ln|c| \rfloor = p} \qquad \texttt{1...1}\,\overbrace{\texttt{0X...XXXX}}^{1+\lfloor \ln(|c|-1) \rfloor}$$

Positive Negative

Ordering relationships in $\mathcal{L}$

For all $c \in \mathcal{C}$, $\langle m, p\rangle \in \mathcal{T}$:

$$\begin{array}{rcl} \bot \sqsubseteq c & \text{and} & \bot \sqsubseteq \langle m, p\rangle \\ \langle m_1, p_1\rangle \sqsubseteq \langle m_2, p_2\rangle & \text{iff} & m_1 \leq m_2 \wedge p_1 \leq p_2 \\ c \sqsubseteq \langle m, p\rangle & \text{iff} & \textbf{bw}(c) \sqsubseteq \langle m, p\rangle \end{array}$$

where:

$$\textbf{bw}(c) : \mathcal{C} \rightarrow \mathcal{T} = \begin{cases} \langle 0, 0\rangle & c = 0 \\ \langle 0, 1 + \lfloor \ln|c| \rfloor\rangle & c > 0 \\ \langle 1 + \lfloor \ln|c| \rfloor, 0\rangle & c < 0 \end{cases}$$

$$\texttt{0...0}\overbrace{\texttt{1X...XXX}}^{1+\lfloor \ln|c| \rfloor = p} \qquad \texttt{1...1}\overbrace{\texttt{0X...XXXX}}^{1+\lfloor \ln(|c|-1) \rfloor \leq m}$$

Positive Negative

Some abstract evaluation rules

Negation:

$$\text{-}\ \langle m, p \rangle = \langle p, m \rangle$$

Addition:

$$\langle m_l, p_l \rangle + \langle m_r, p_r \rangle = \langle 1 + \max(m_l, m_r), 1 + \max(p_l, p_r) \rangle$$

Multiplication:

$$\langle m_l, p_l \rangle * \langle m_r, p_r \rangle = \left\langle \begin{array}{l} \max(m_l + p_r, p_l + m_r), \\ \max(m_l + m_r, p_l + p_r) \end{array} \right\rangle$$

Bitwise-AND:

$$\langle 0, p_l \rangle \ \& \ \langle 0, p_r \rangle = \langle 0, \min(p_l, p_r) \rangle$$

$$\langle m_l, p_l \rangle \ \& \ \langle m_r, p_r \rangle = \langle \max(m_l, m_r), \max(p_l, p_r) \rangle$$

Intraprocedural bitwidth analysis

- Given domains:
  - $\mathcal{E}$, CFG edges
  - $\mathcal{V}$, (SSI form) variables
- The intraprocedural analysis discovers:
  - a set **e** $: 2^{\mathcal{E}}$ of executable edges
  - a map **val** $: \mathcal{V} \to \mathcal{L}$ giving abstract values valid for all possible executions

SSI form

Allows us to discover facts about `i` at points *A* and *B*:

```
i = ...;
...
i = ...;
while (0<i  && i <50) {
    ...=i ; // A
}
...=i ; // B
```

SSI form

Allows us to discover facts about `i` at points *A* and *B*:

```
i₀= ...;
...
i₁= ...;
while (0<i₁ && i₁<50) {
    ...=i₂; // A
}
...=i₃; // B
```

$$\textbf{val}(\mathtt{i}_2) \sqsubseteq \langle 0, \underbrace{6}_{1+\lfloor \ln 50 \rfloor} \rangle$$

Interprocedural overview

- Field-based extension from intra- to interprocedural analysis.

Interprocedural overview

- Field-based extension from intra- to interprocedural analysis.
  - Ignore the left component of expression `o.f`
  - Single analysis value for each declared field

Interprocedural overview

- Field-based extension from intra- to interprocedural analysis.
  - Ignore the left component of expression `o.f`
  - Single analysis value for each declared field
- Context-sensitive implementation.
  - Discriminate between variables (but not fields) in distinct calling contexts.

Interprocedural overview

- Field-based extension from intra- to interprocedural analysis.
  - Ignore the left component of expression `o.f`
  - Single analysis value for each declared field
- Context-sensitive implementation.
  - Discriminate between variables (but not fields) in distinct calling contexts.
- All results in this talk use zero-length context (context-insensitive).

Interprocedural analysis

We use a field-based interprocedural analysis.

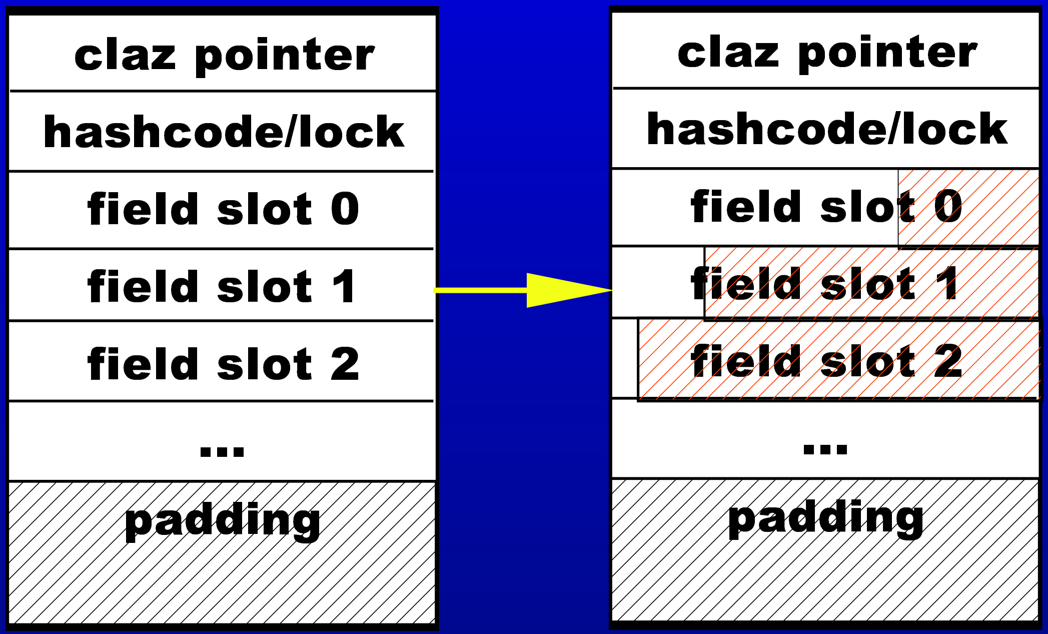
- Given domains:
  - $\mathcal{E}$, CFG edges
  - $\mathcal{V}$, (SSI form) variables
  - $\mathcal{M}$, call sites in the program
  - $\mathcal{F}$, declared fields in the program
- The interprocedural analysis discovers:
  - a set **e** $: 2^{\mathcal{E}\times\mathcal{M}^*}$ of executable edges
  - a map **val** $: ((\mathcal{V} \times \mathcal{M}^*) + \mathcal{F}) \to \mathcal{L}$ giving abstract values valid for all possible executions
  - a set **Read** $: 2^{\mathcal{F}}$ of readable fields

All cars are black

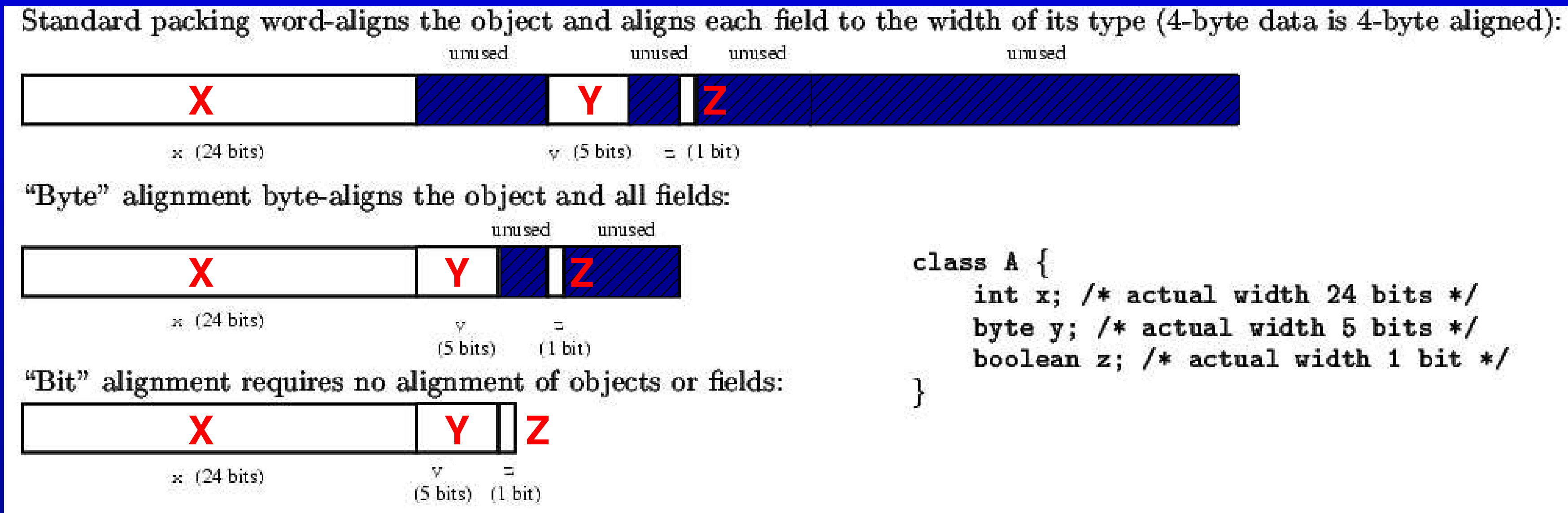
```
void paint(int color) {
    if (this.model == FORD)
        color = BLACK;
    this.color = color;
}
```

Using the analysis:

Field compression using bitwidths



Field packing

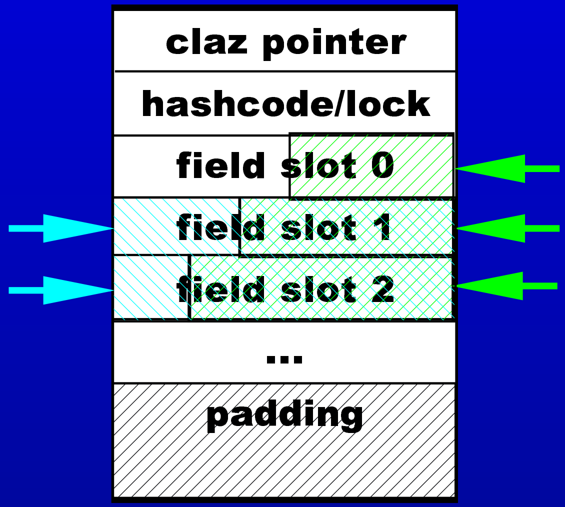


Object header omitted.

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

```
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-occuring 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

```
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);
    }
}
```

Specialization example:
java.lang.String

```
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; }
}
```

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.

```
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);
```

Transforming allocation sites

Case 3: assignment to field is unknown.

```
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.

```
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 cannot be mutated after the constructor completes except by subclasses.

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-occuring 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.
- To achieve our storage savings, we interpret a nonexistent entry as the frequent "mostly-constant" value.

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.
- To achieve our storage savings, we interpret a nonexistent entry as the frequent "mostly-constant" value.
- If a field is set to the "mostly-constant" value, remove its entry from the map.

Externalization example:

java.lang.String

```
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);
    }
}
```

Externalization example:

java.lang.String

```
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);
    }
}
```

Externalization example:

java.lang.String

```
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

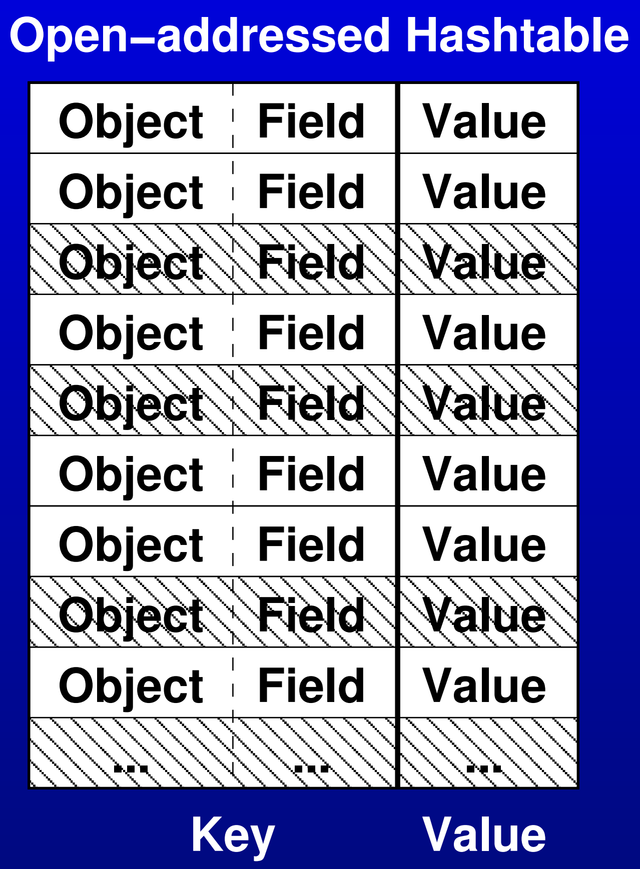
Open–addressed Hashtable

Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
...	...	...

Key **Value**

- “open addressed” for low overhead.

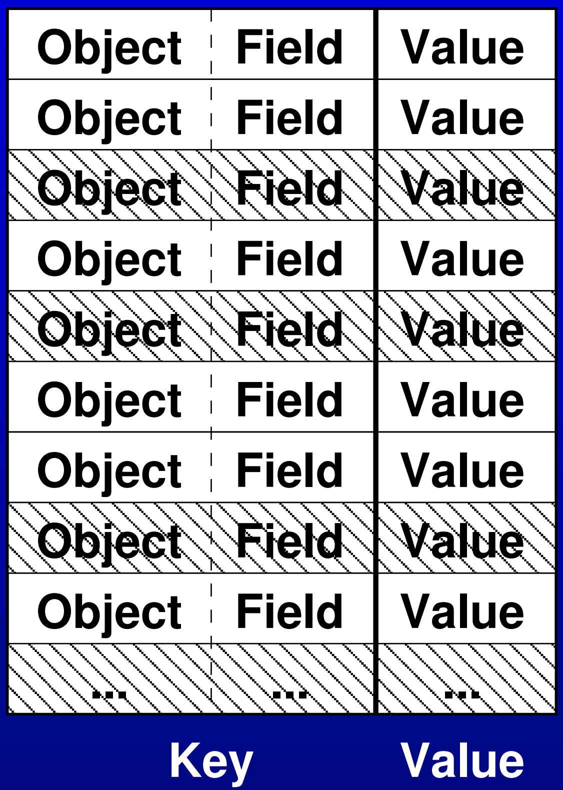
External map implementation



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

External map implementation

Open–addressed Hashtable



- “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 Hashtable

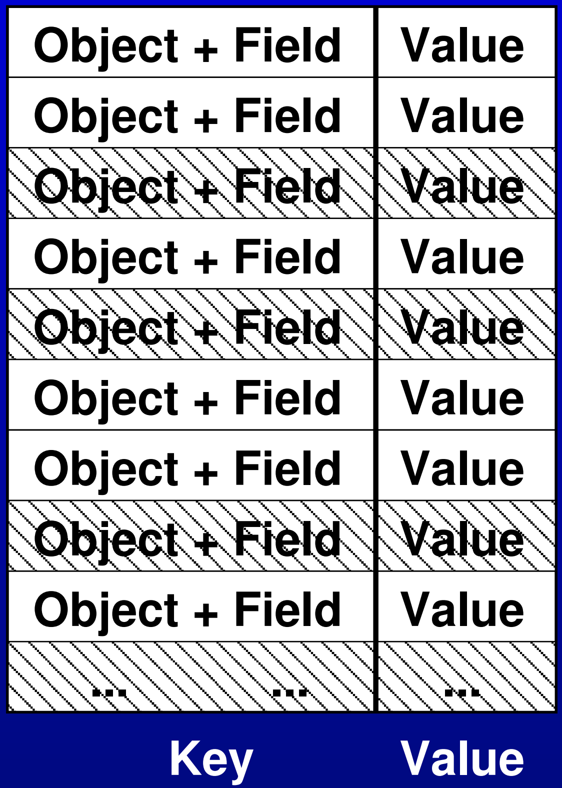
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
Object	Field	Value
...	...	...
Key		Value

- “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!

Open–addressed Hashtable



- Use small integers to enumerate fields.

We can do better!

Open–addressed Hashtable

Key	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
... ...	...

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

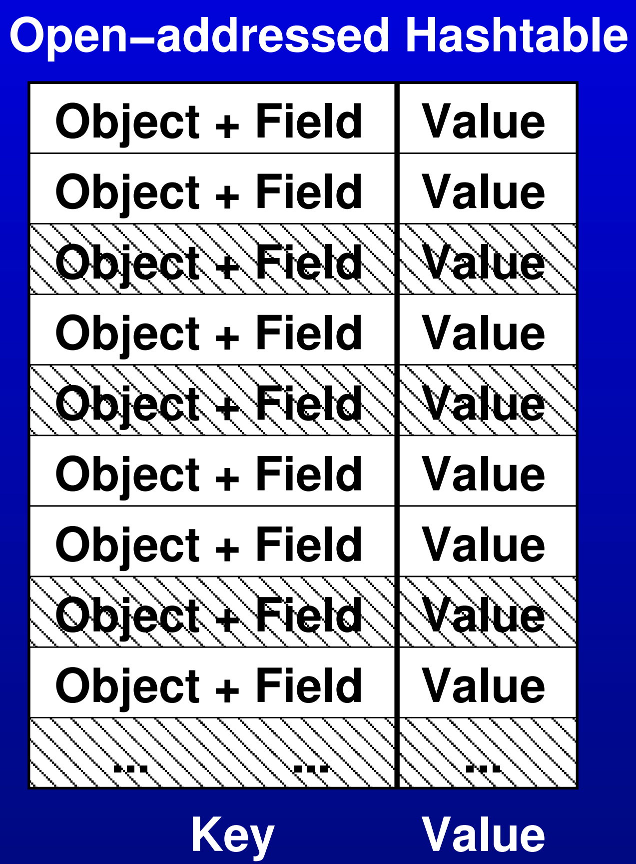
We can do better!

Open–addressed Hashtable

Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
Object + Field	Value
... ...	...
Key	**Value**

- 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%.

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.

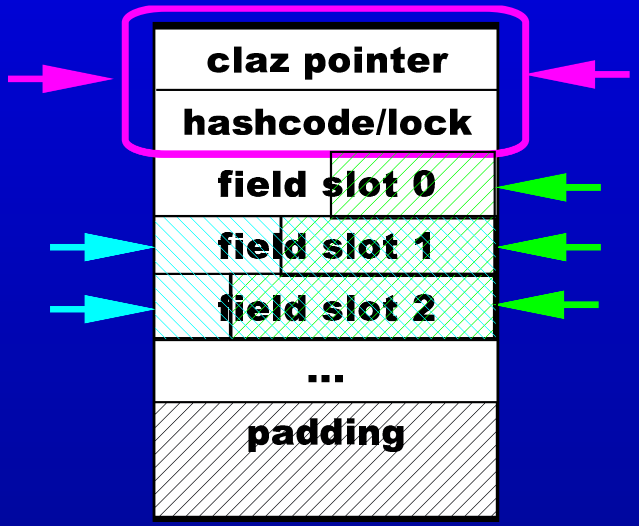
Other details

- 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

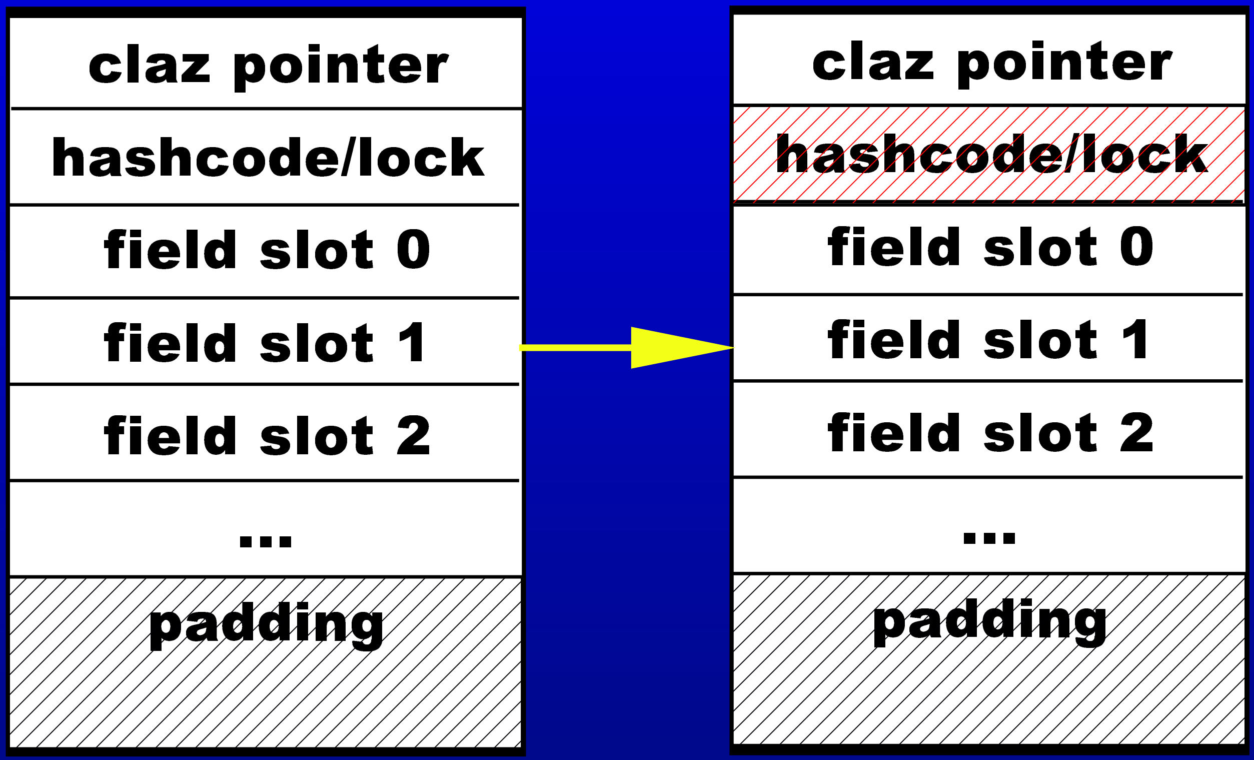
Three broad techniques:

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



Header optimizations:

Hashcode/Lock compression



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.

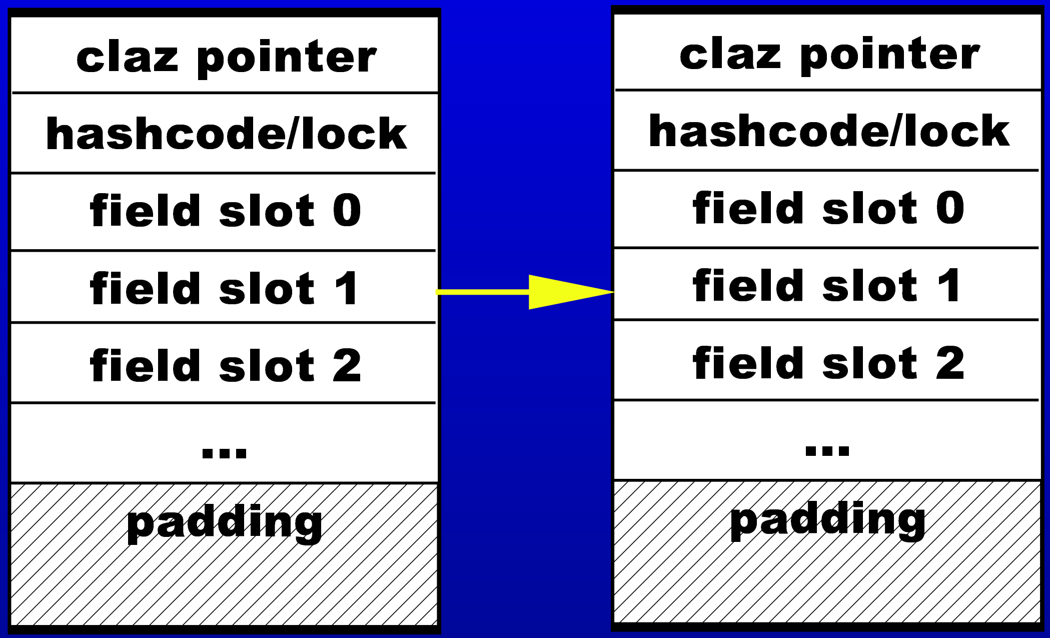
Header optimizations:

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.

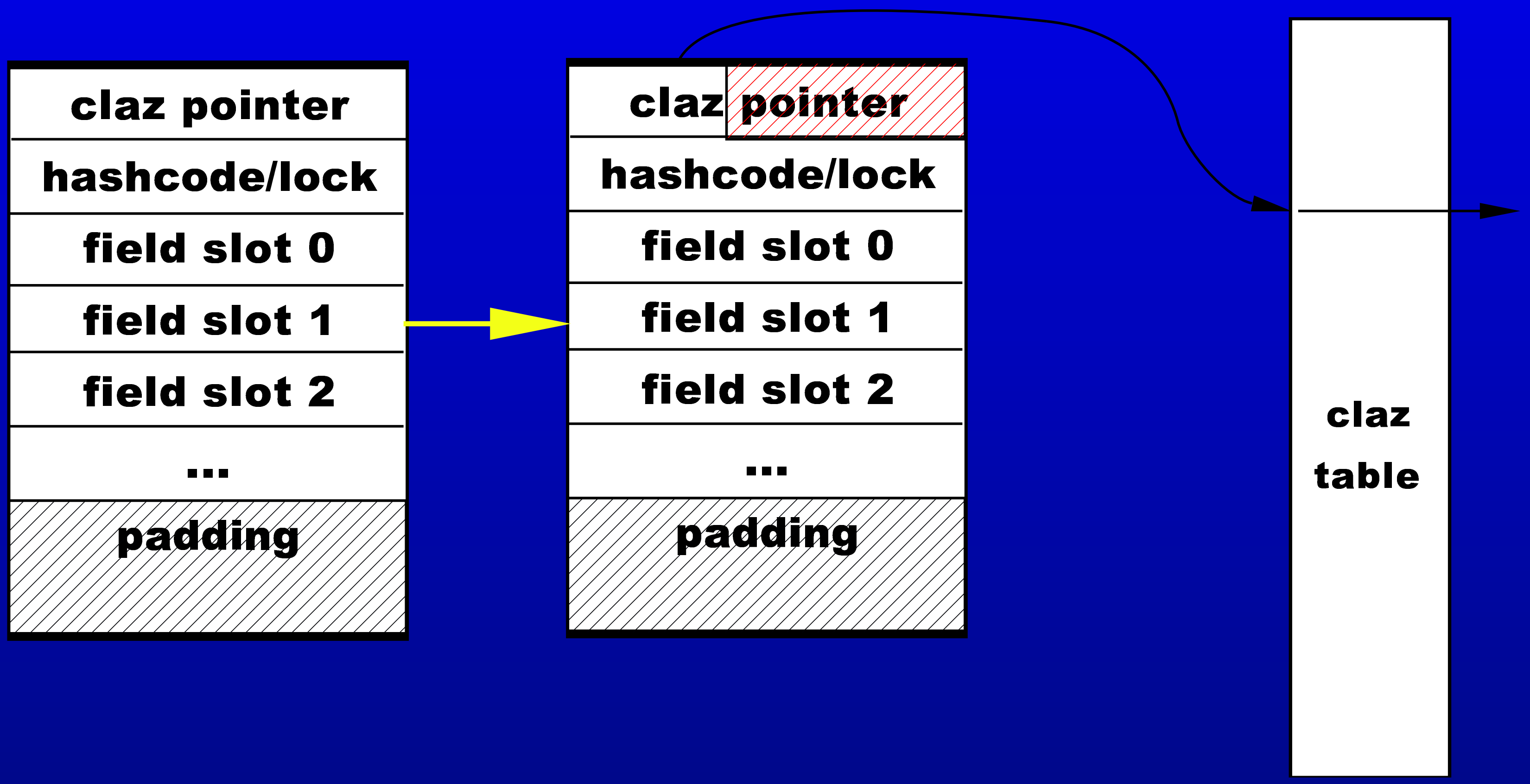
Header optimizations:

claz compression



Header optimizations:

claz compression



Header optimizations:
claz compression

- Replace `claz` pointer with a (smaller) table index. Only instantiated types need be indexed.

Header optimizations:

claz compression

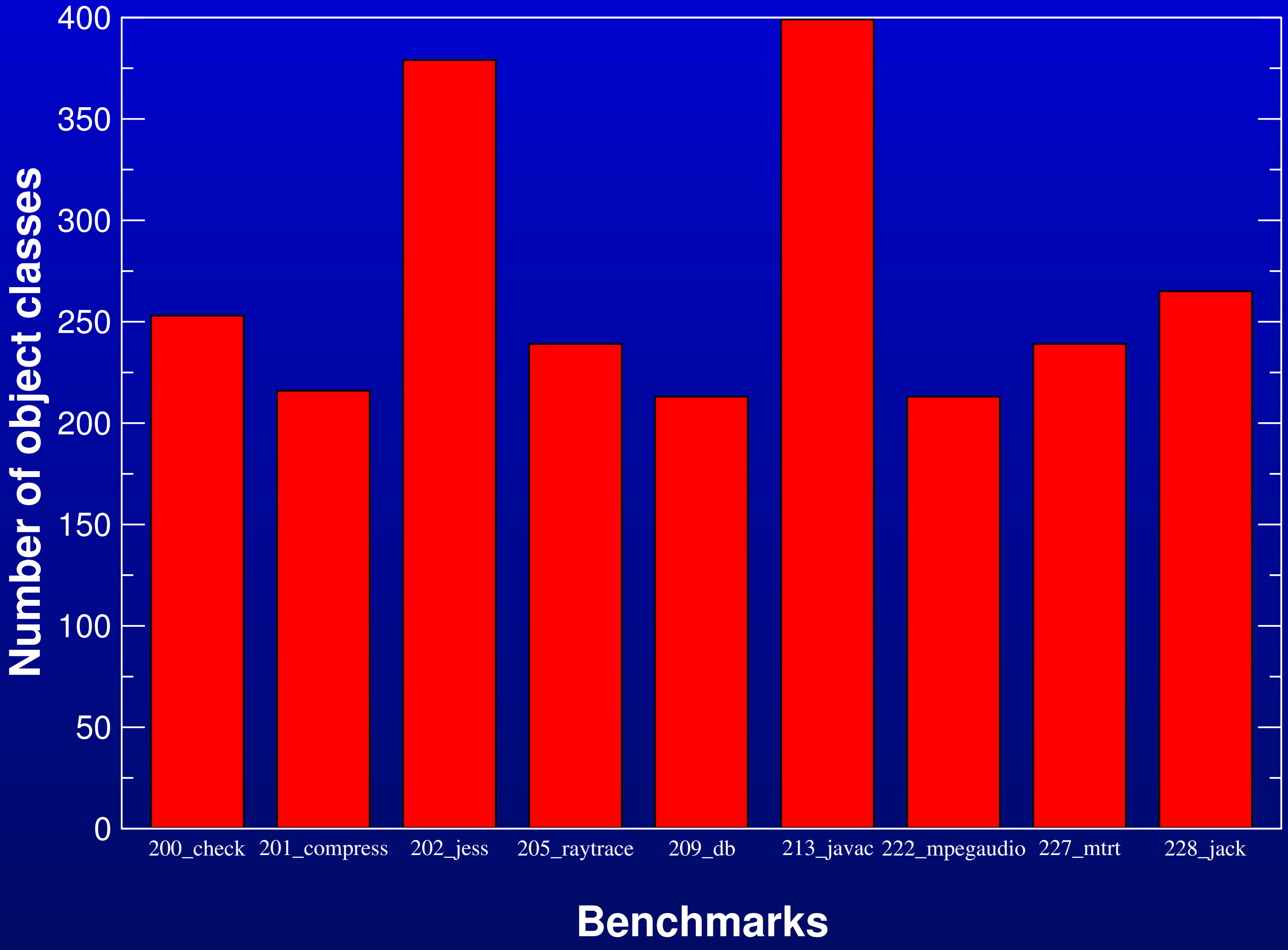
- Replace `claz` pointer with a (smaller) table index. Only instantiated types need be indexed.
- With co-operation of GC, works in dynamic environments.

Header optimizations:
claz compression

- Replace `claz` pointer with a (smaller) table index. Only instantiated types need be indexed.
- With co-operation of GC, works in dynamic environments.
- Many applications instantiate less than 256 object types.

Class statistics

Class statistics for applications in SPECjvm98 benchmark suite:

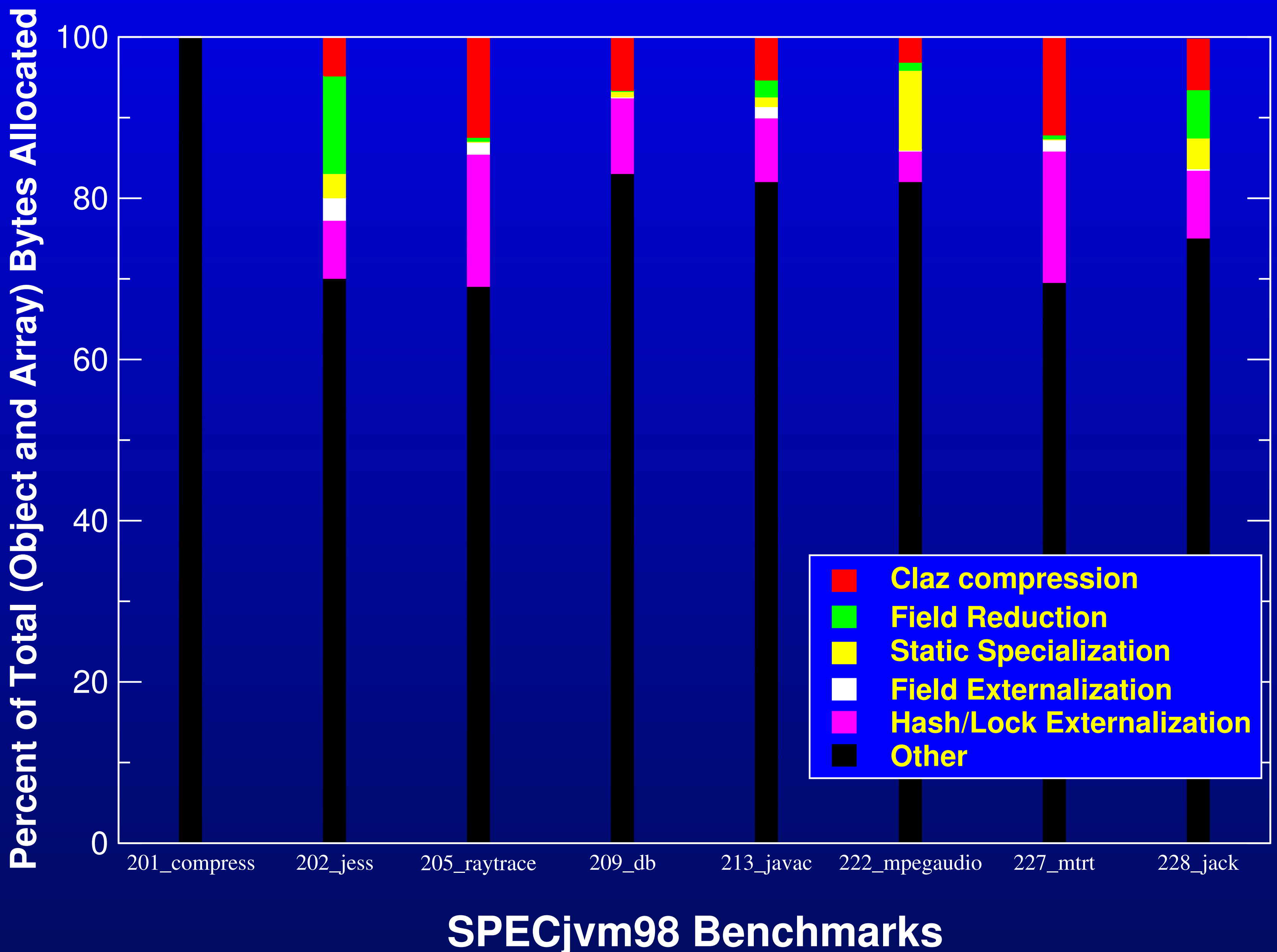


How well does it work?

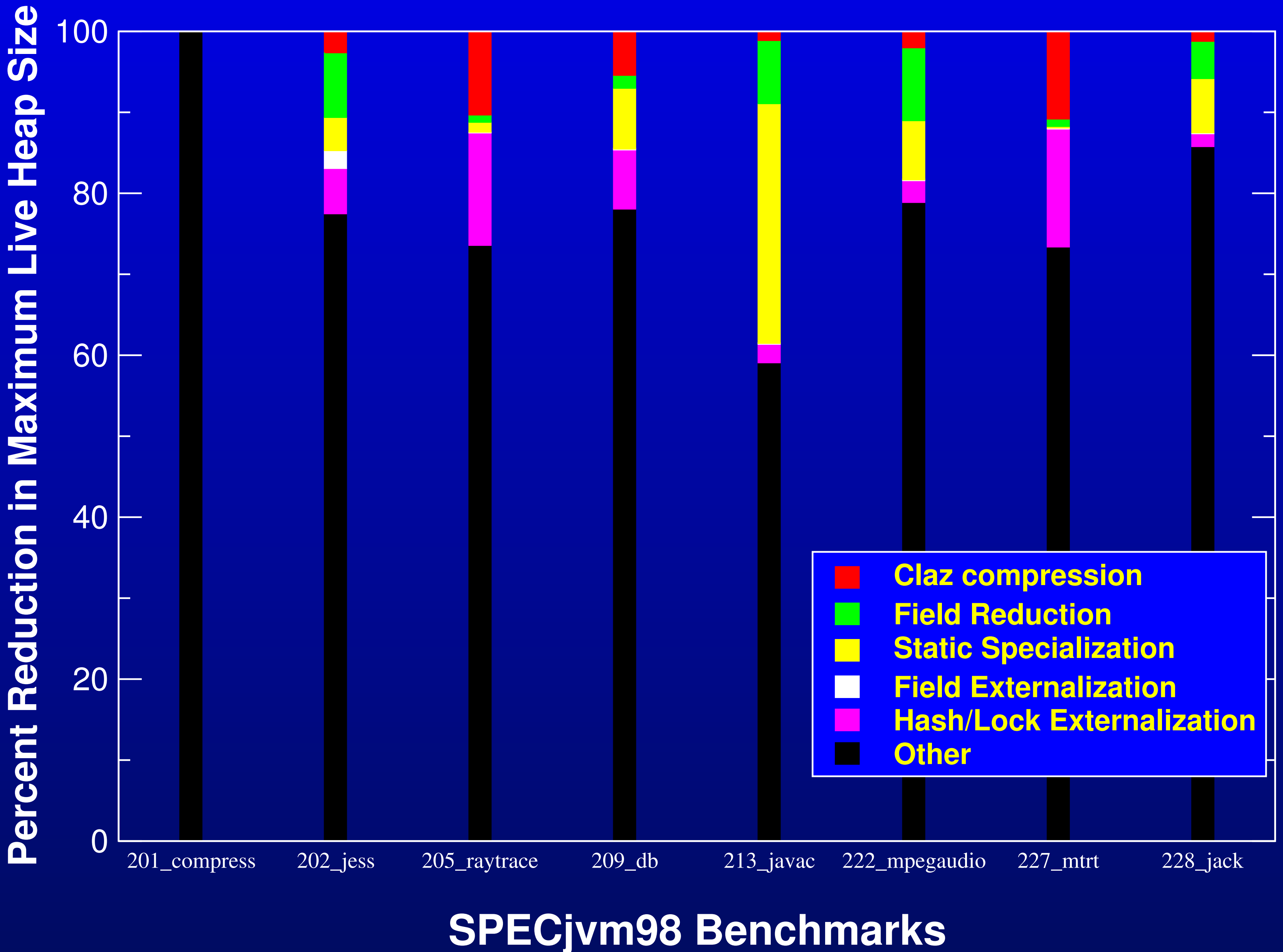
Experimental setup

- Implemented all analyses and transformations with MIT FLEX Java compiler infrastructure.
  - Whole-program static compiler.
  - Generates C or native code for ARM/MIPS/Sparc.
- SPECjvm98 benchmark suite with full input size.
  - JDK 1.1.8 class libraries.
- Benchmarks run on dual-processor 900 MHz Pentium III running Debian Linux.
  - C backend, GCC 2.95, -O9

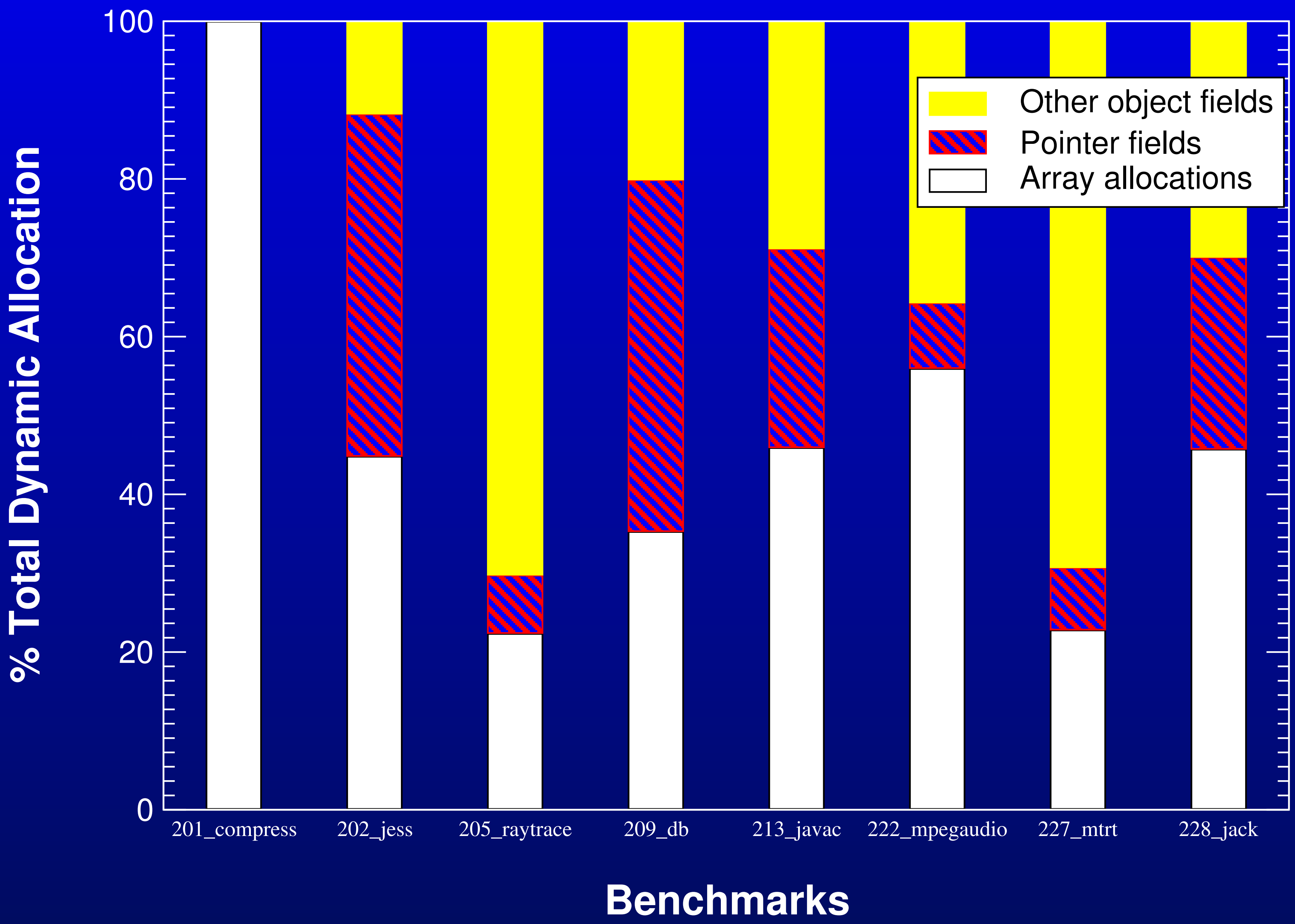
Reduction in total allocations



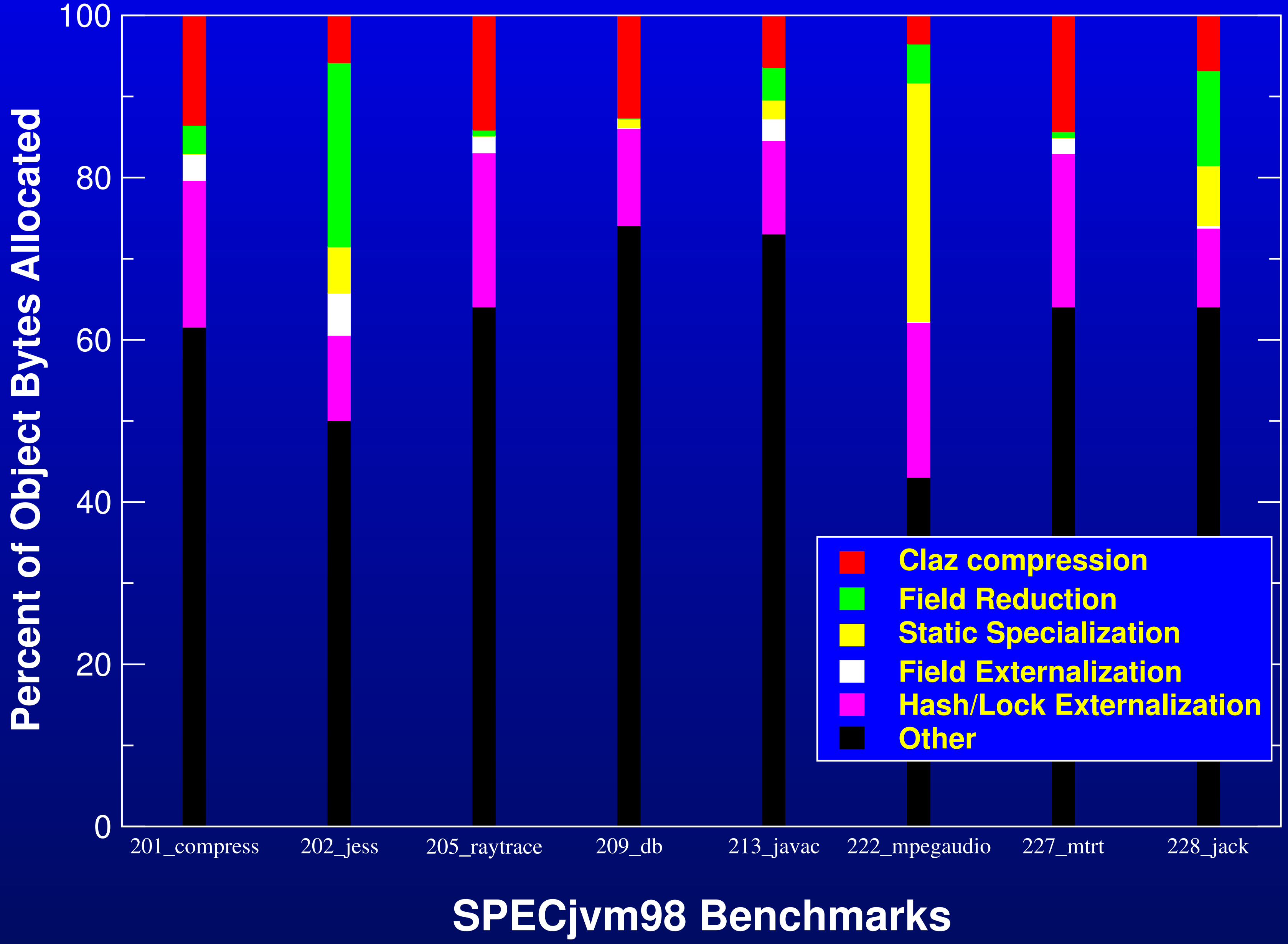
Reduction in total live data



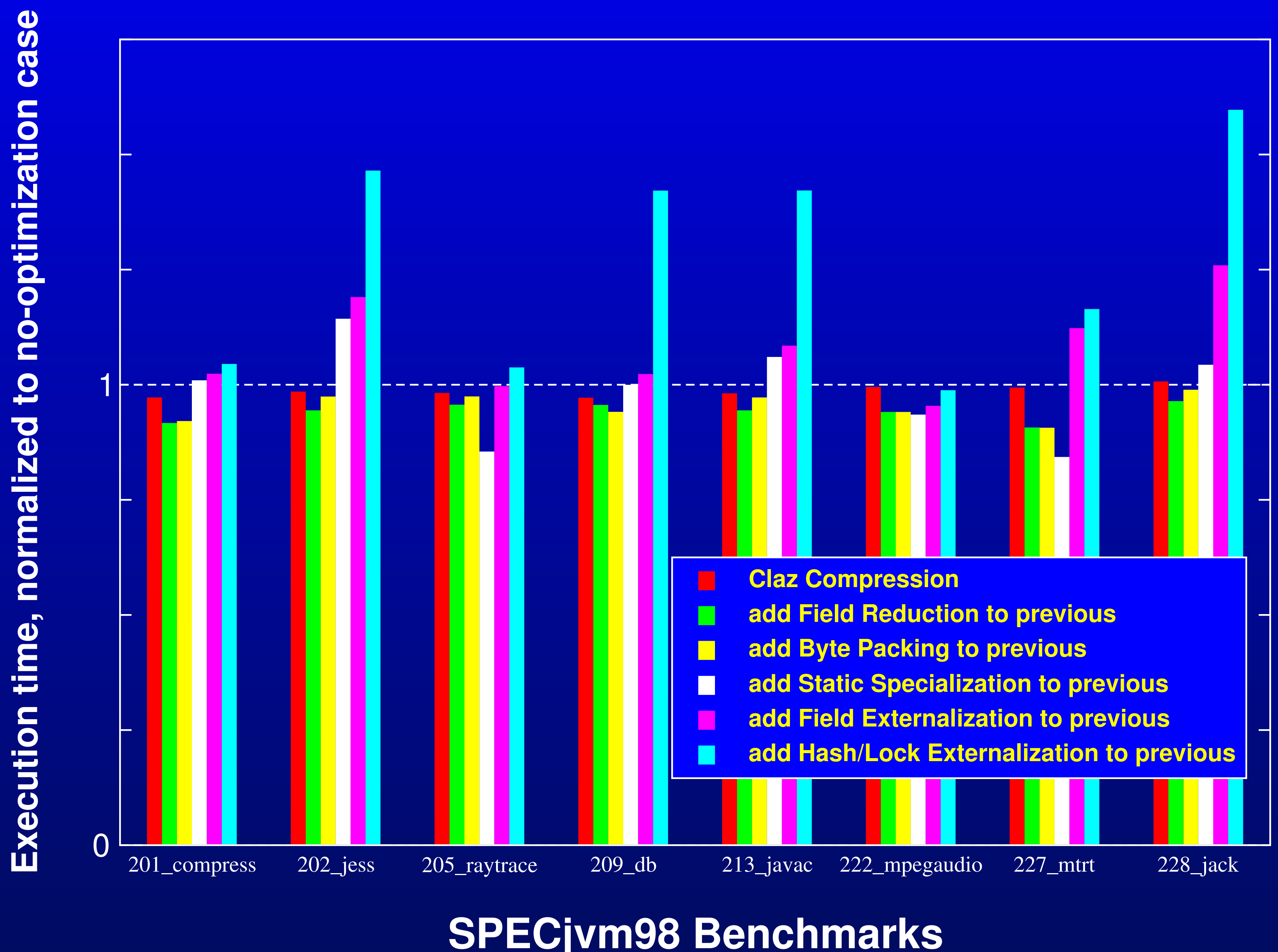
Available reduction opportunities



Reduction in object allocations



Moderate performance impact



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 by 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 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.

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, Konuru, Murthy, Serrano (PLDI ’98)
  - 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)
  - Sălcianu, 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)
  - Rugină, 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.

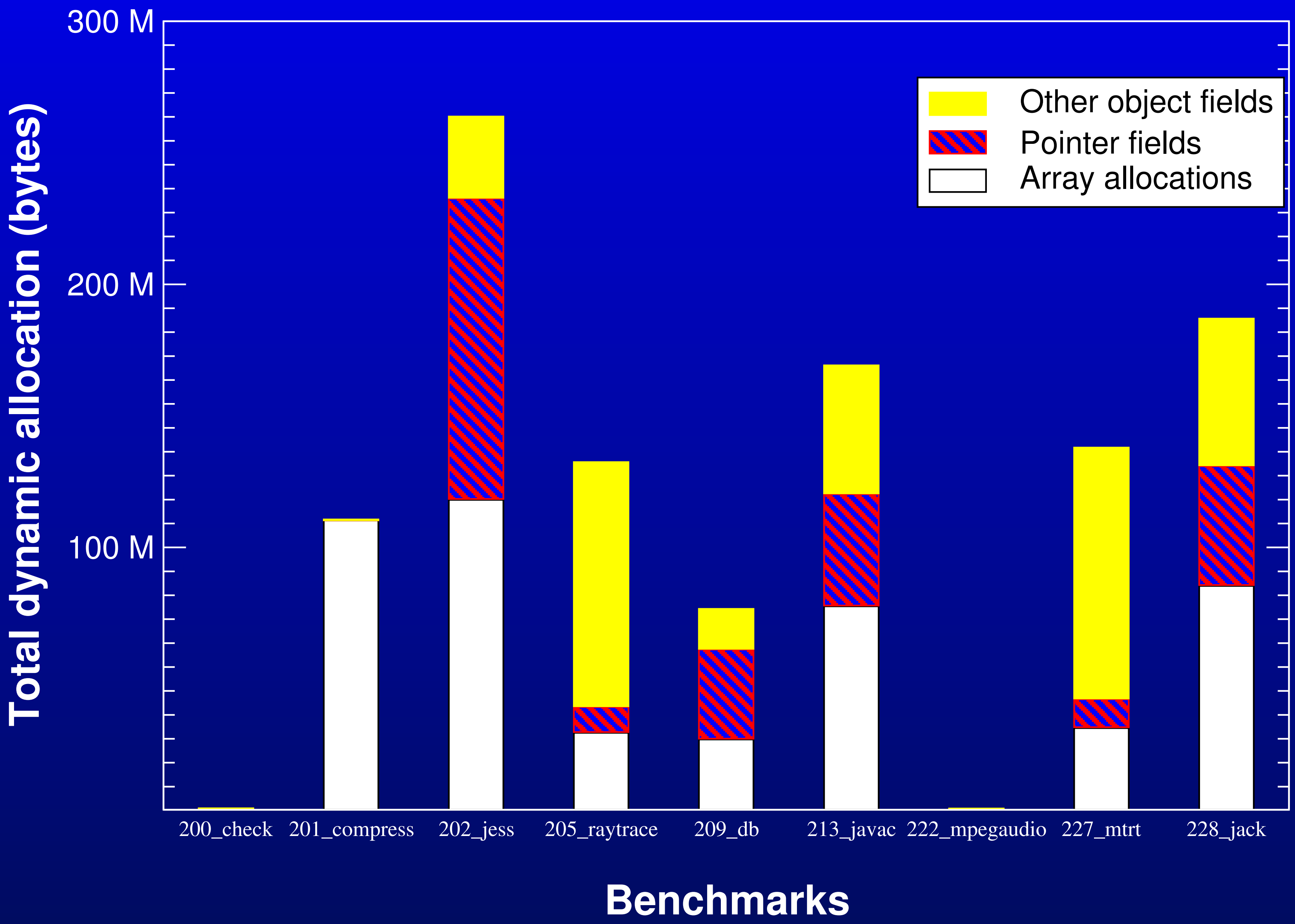
Size Optimizations for Java Programs

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.

Available reduction opportunities



Bitwidth analysis

Motivation:

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

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

Bitwidth analysis

Motivation:

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

```
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.

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

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

```
void foo() {
    int x, y, z;
    ...
}
```

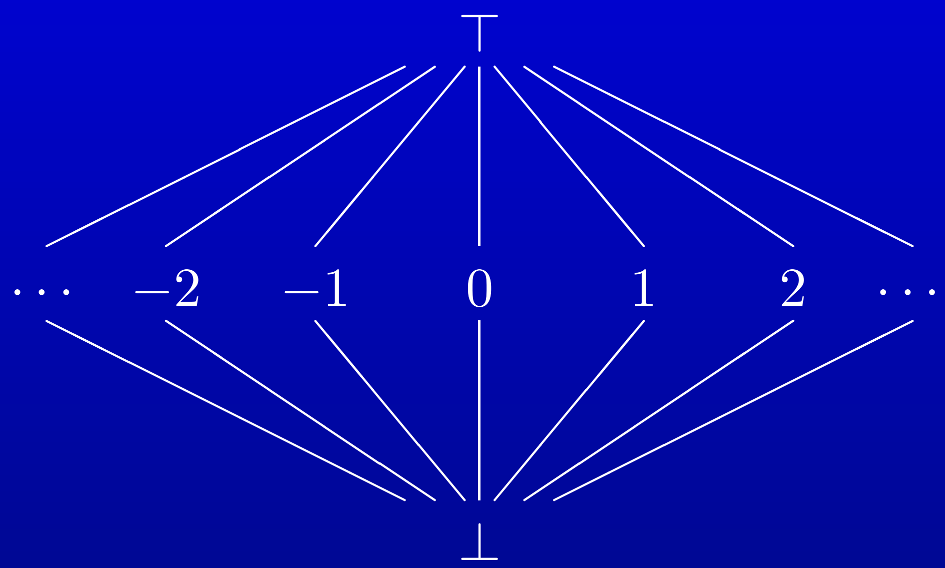
- The compiler can do it for us!

Intraprocedural Analysis

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

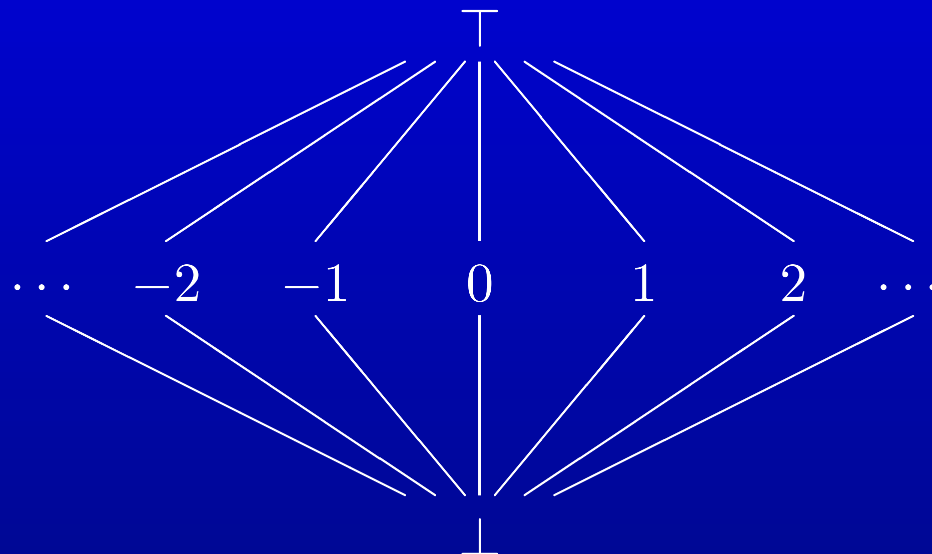
Intraprocedural Analysis

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



Intraprocedural Analysis

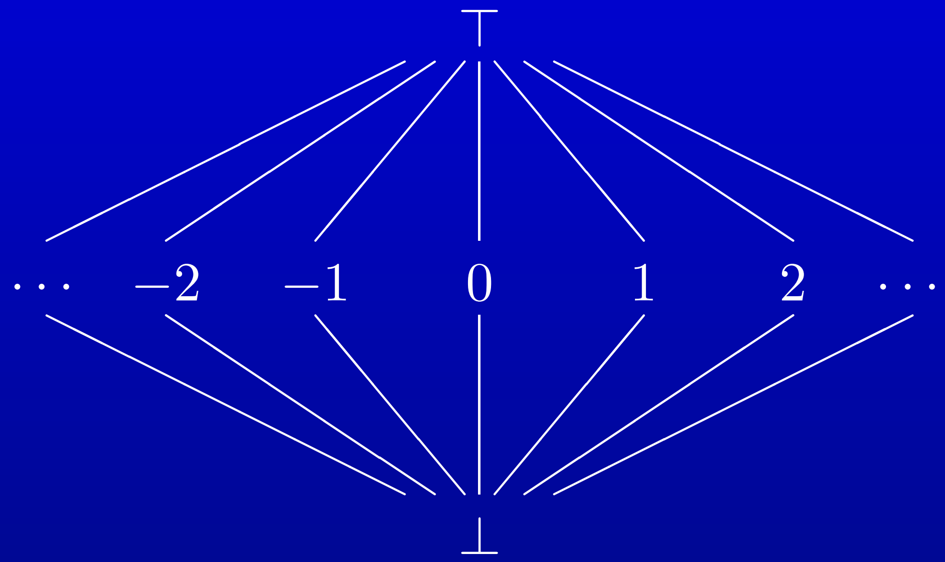
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\boxed{\texttt{i} = \perp}$

Intraprocedural Analysis

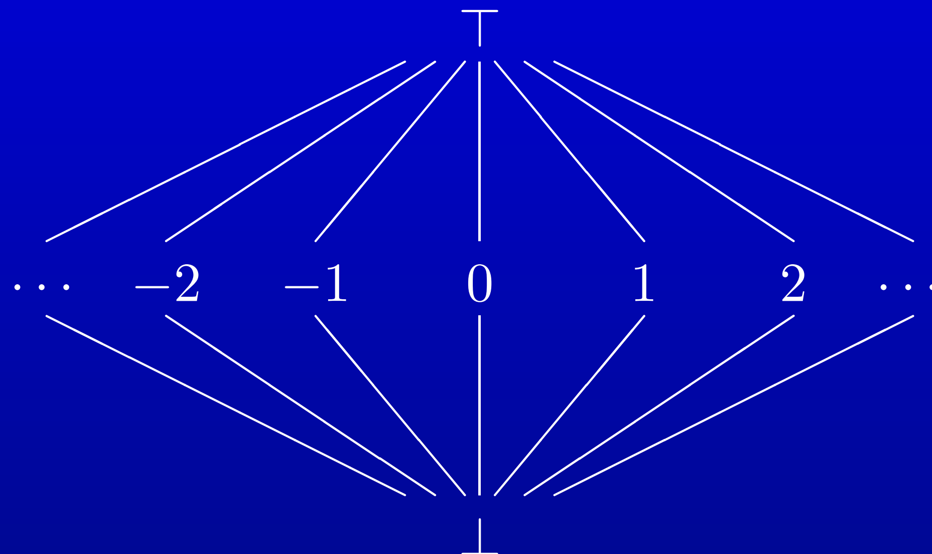
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\boxed{\mathtt{i} = \perp}$

Intraprocedural Analysis

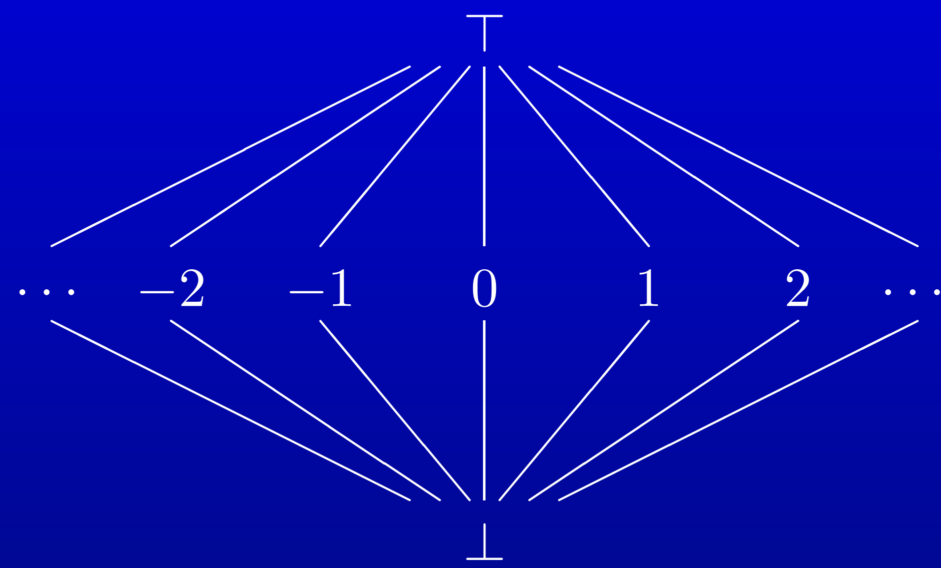
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\boxed{\mathtt{i} = \perp \sqcap 1 \sqcap 2}$$

Intraprocedural Analysis

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

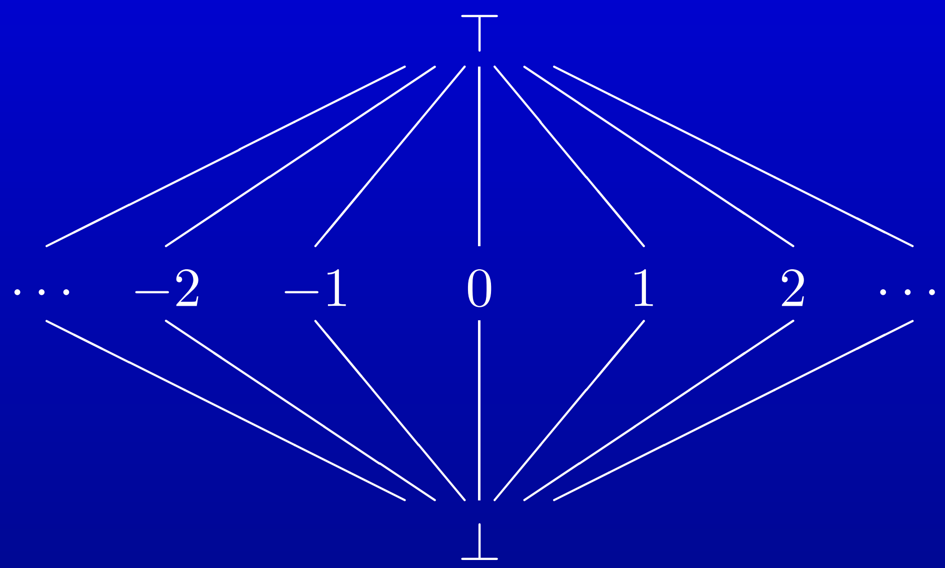


$$\boxed{\mathbf{i} = \bot \sqcap 1 \sqcap 2 = \top}$$

[Because $1 \sqsubseteq \top$ and $2 \sqsubseteq \top$]

Intraprocedural Analysis

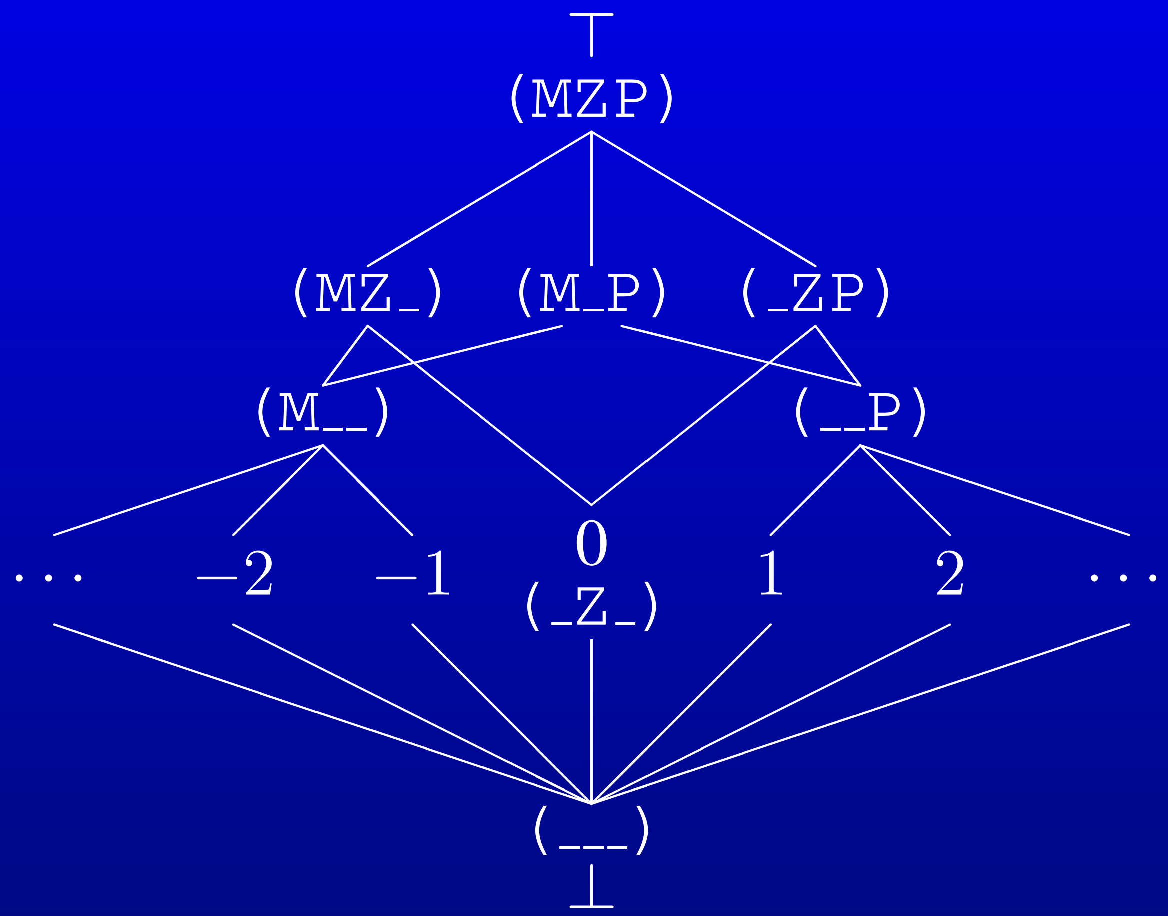
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\boxed{\mathbf{i} = \bot \sqcap 1 \sqcap 2 = \top}$$

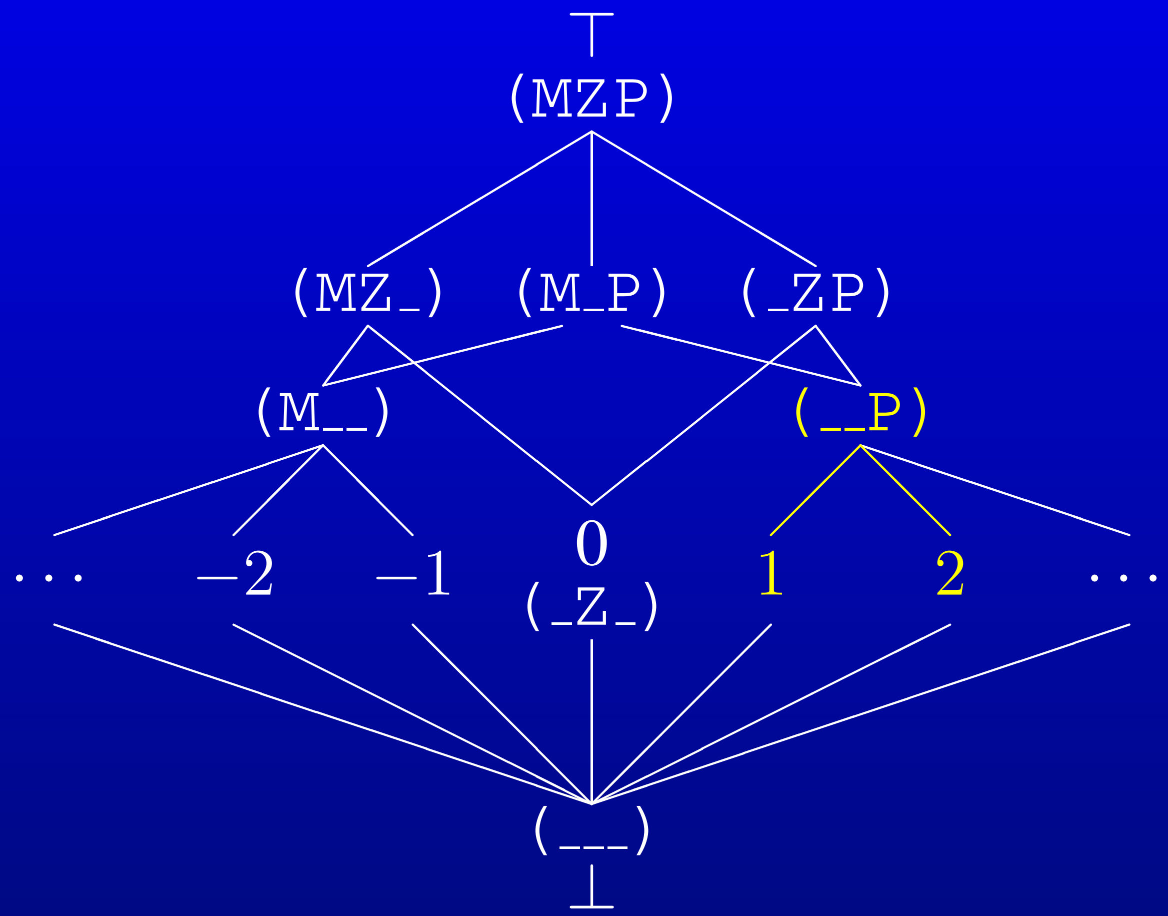
[Because $1 \sqsubseteq \top$ and $2 \sqsubseteq \top$]

A signed integer lattice



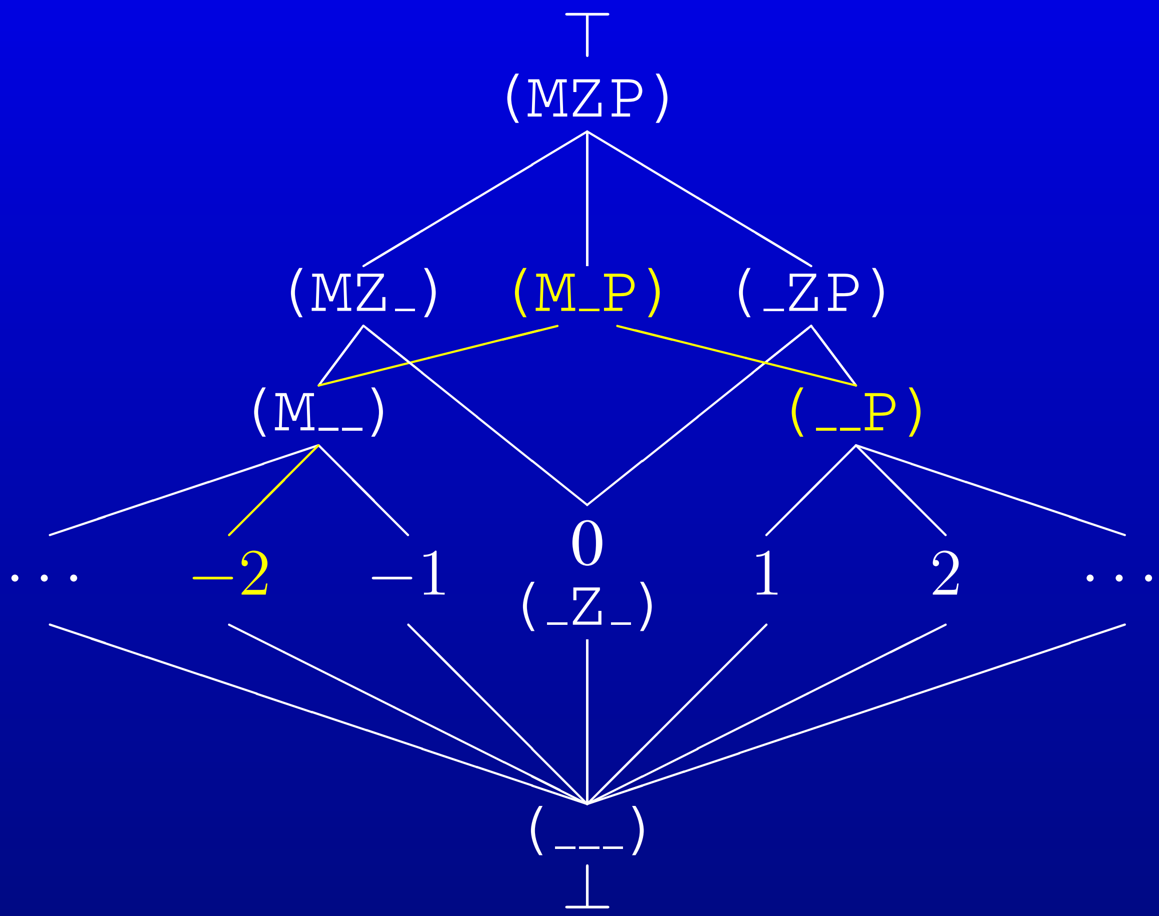
An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain.

A signed integer lattice



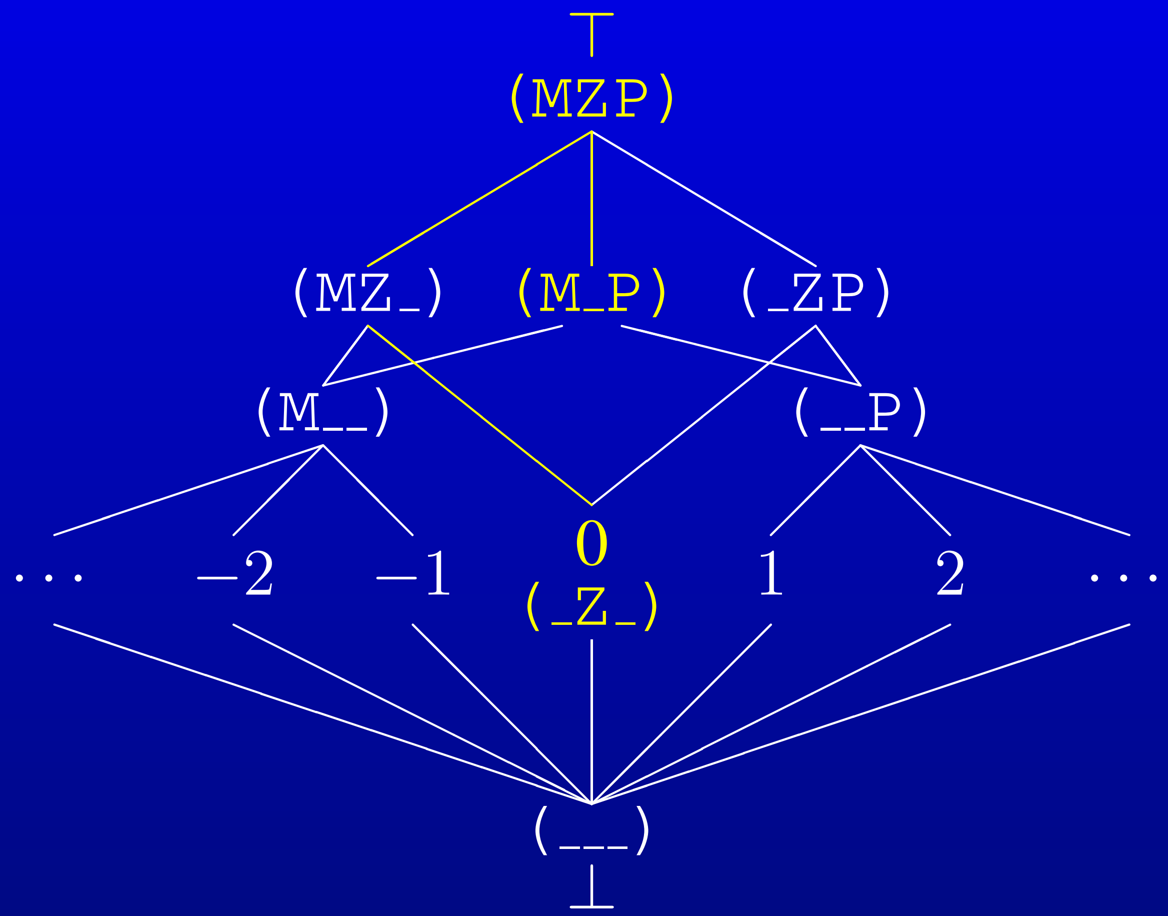
An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain.

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.

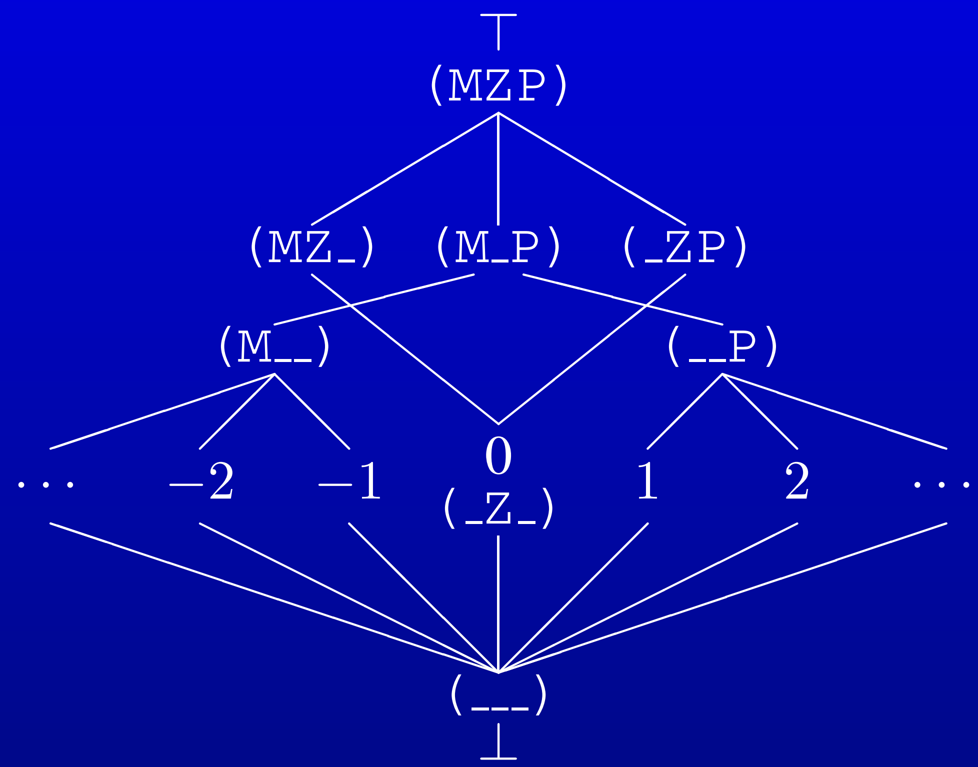
A signed integer lattice



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

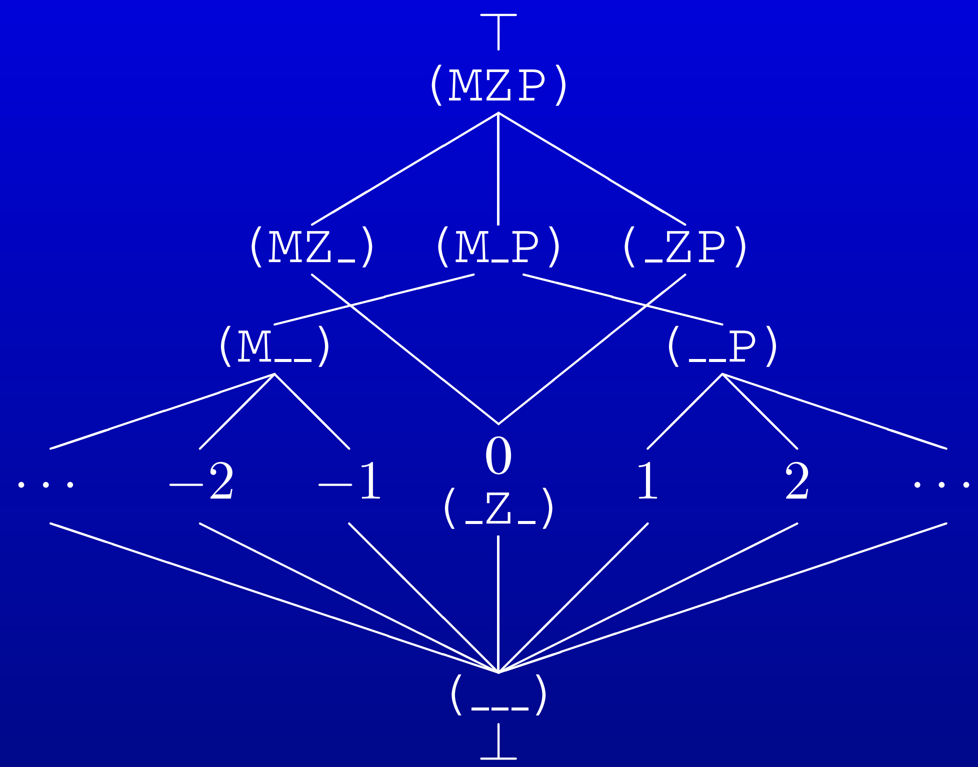
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\boxed{\mathbf{i} = \perp}$

Example, redux

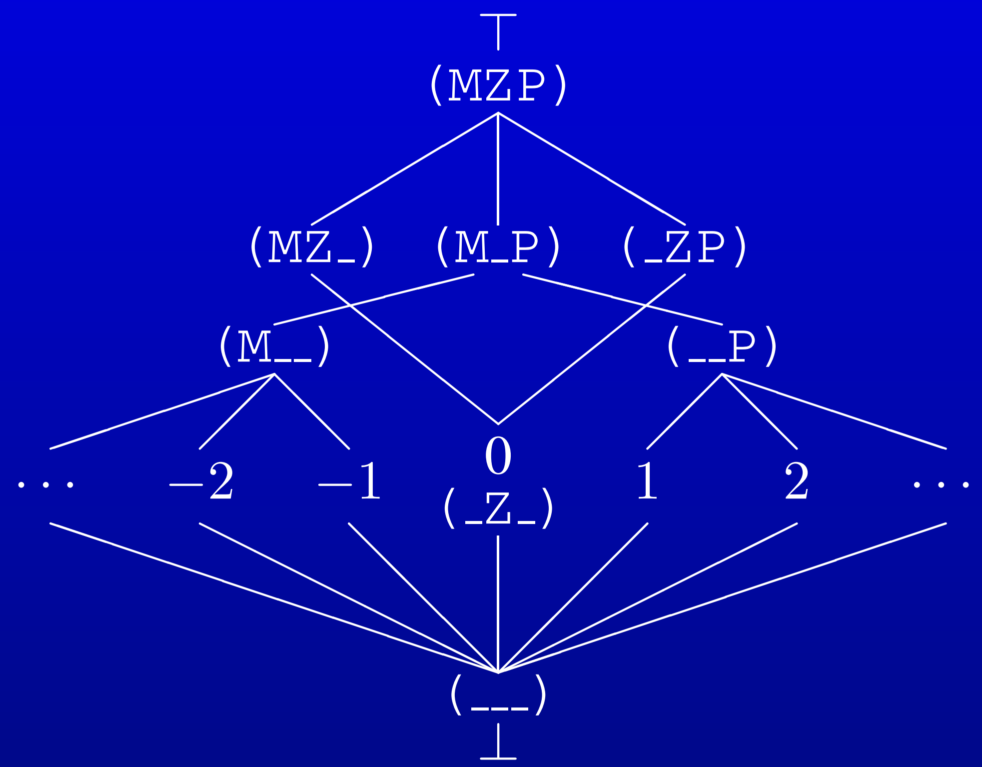
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\boxed{\mathtt{i} = \perp}$

Example, redux

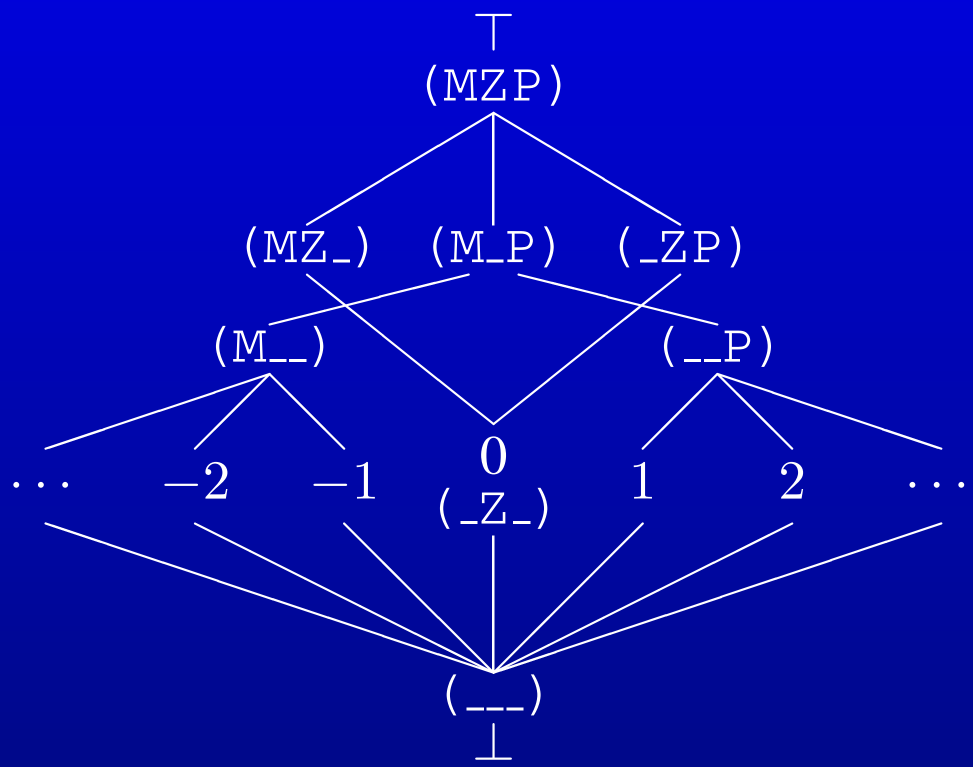
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\boxed{\mathtt{i} = \perp \sqcap 1 \sqcap 2}$$

Example, redux

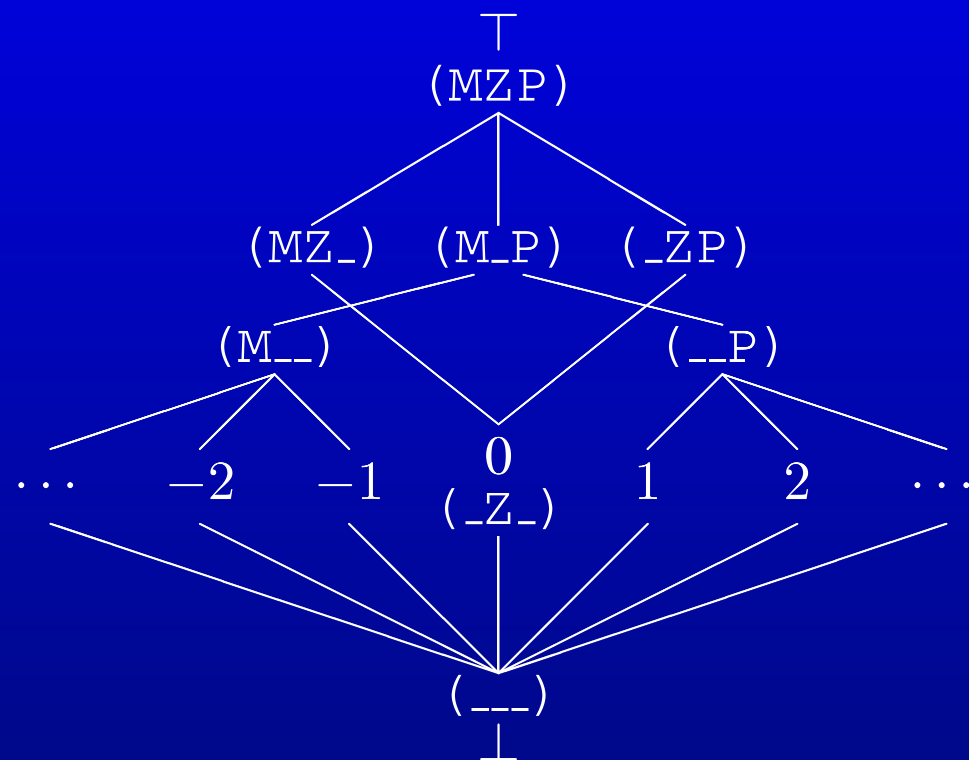
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\boxed{\mathtt{i} = \perp \sqcap 1 \sqcap 2 = \mathtt{(\_\_P)}}$

Example, redux

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



$$\boxed{\mathbf{i} = \bot \sqcap 1 \sqcap 2 = \mathbf{(\_\_P)}}$$

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 $m = p = 0$.

$$\texttt{(\_\_P)} \Rightarrow \langle 0, p \rangle$$
$$\texttt{(M\_\_)} \Rightarrow \langle m, 0 \rangle$$
$$\texttt{(\_Z\_)} \Rightarrow \langle 0, 0 \rangle$$

Extending the lattice

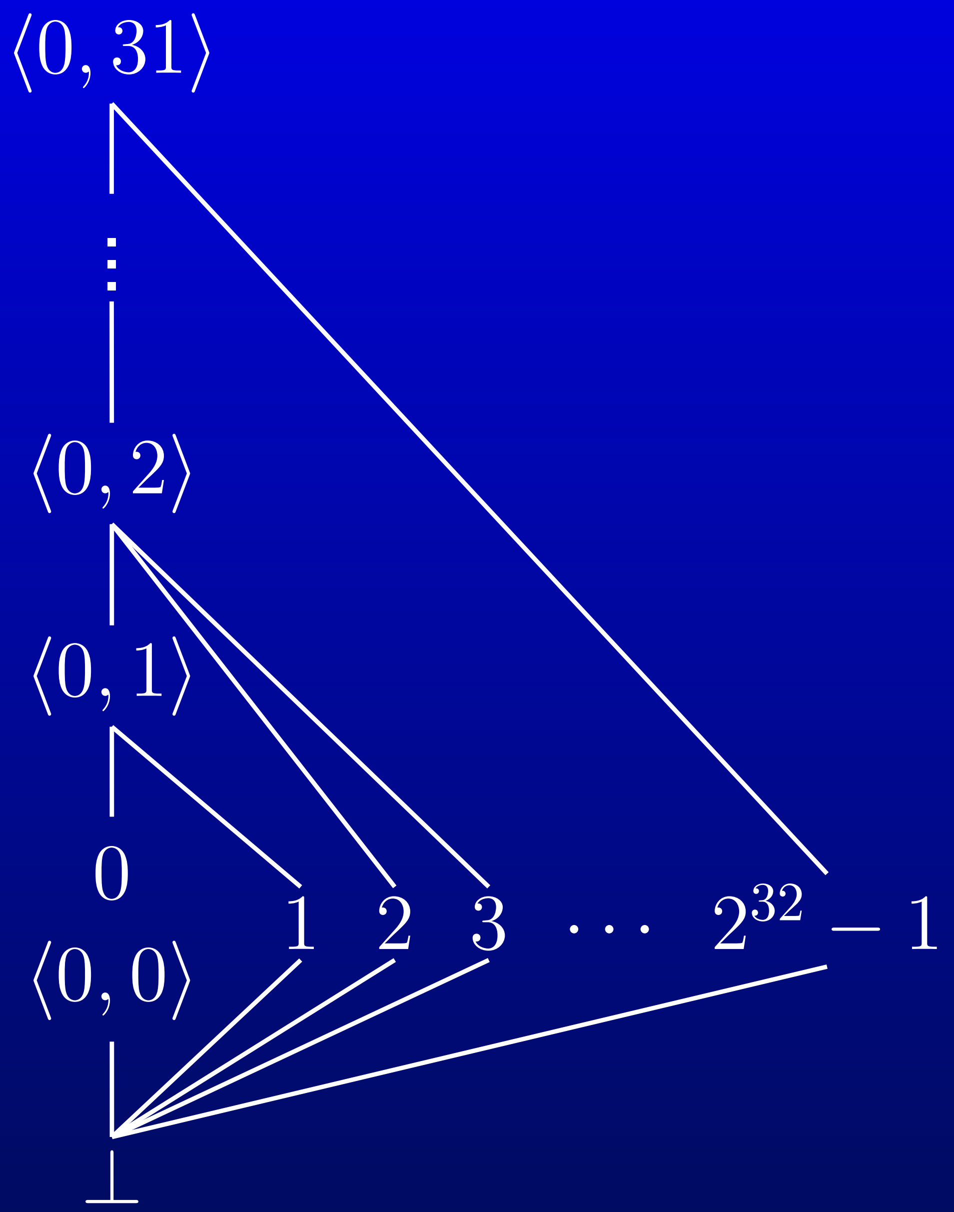
Replace `M` and `P` in previous lattice entries with positive integers m and p. Encode zero as $m = p = 0$.

$$
\begin{aligned}
\texttt{(\_\_P)} &\Rightarrow \langle 0, p \rangle \\
\texttt{(M\_\_)} &\Rightarrow \langle m, 0 \rangle \\
\texttt{(\_Z\_)} &\Rightarrow \langle 0, 0 \rangle
\end{aligned}
$$

In lattice context: $\texttt{(\_\_P)} \Rightarrow$

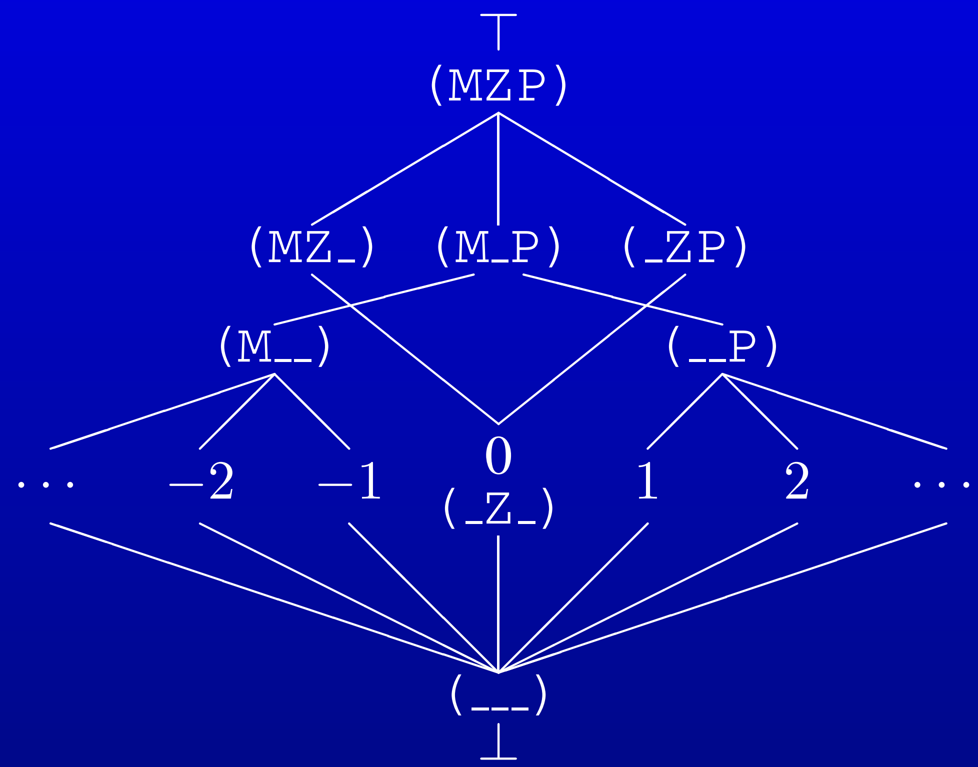
$$
\begin{array}{c}
\langle 0, 31 \rangle \\
| \\
\vdots \\
| \\
\langle 0, 3 \rangle \\
| \\
\langle 0, 2 \rangle \\
| \\
\langle 0, 1 \rangle
\end{array}
$$

Bitwidth lattice detail



Example redux, redux

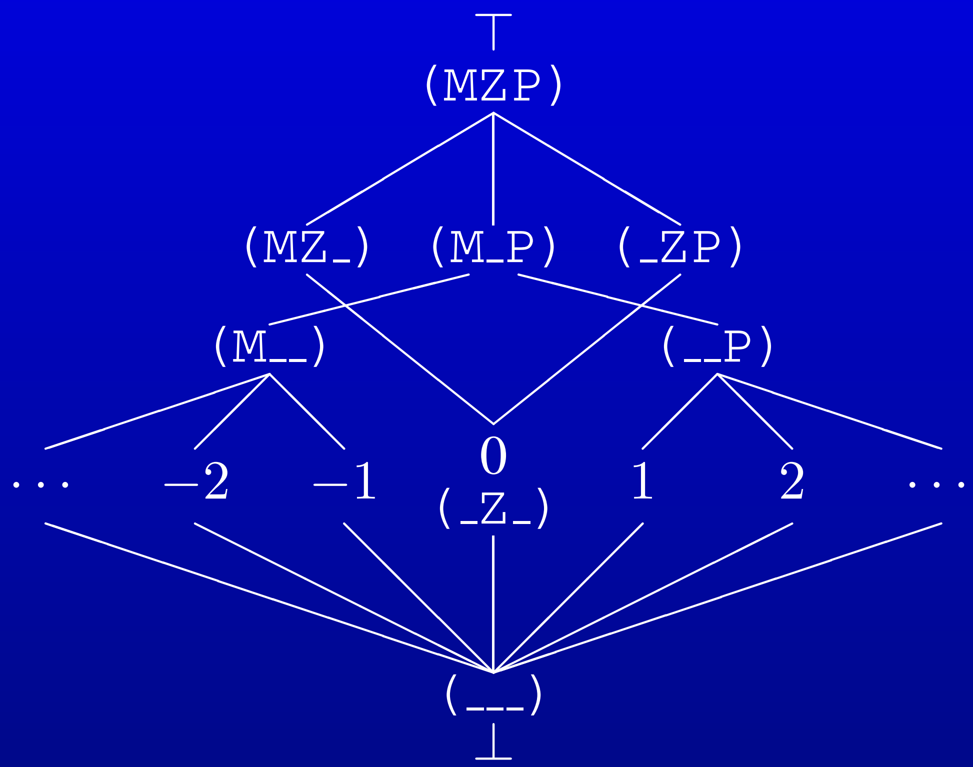
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\boxed{\mathbf{i} = \perp}$

Example redux, redux

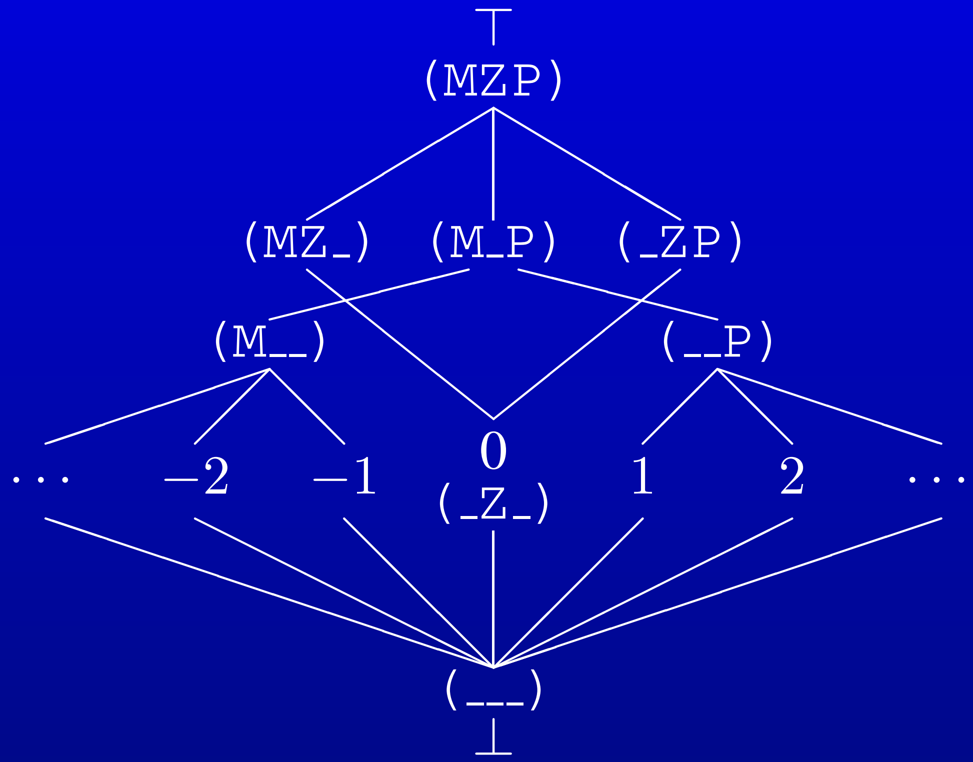
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\boxed{\mathtt{i} = \perp}$

Example redux, redux

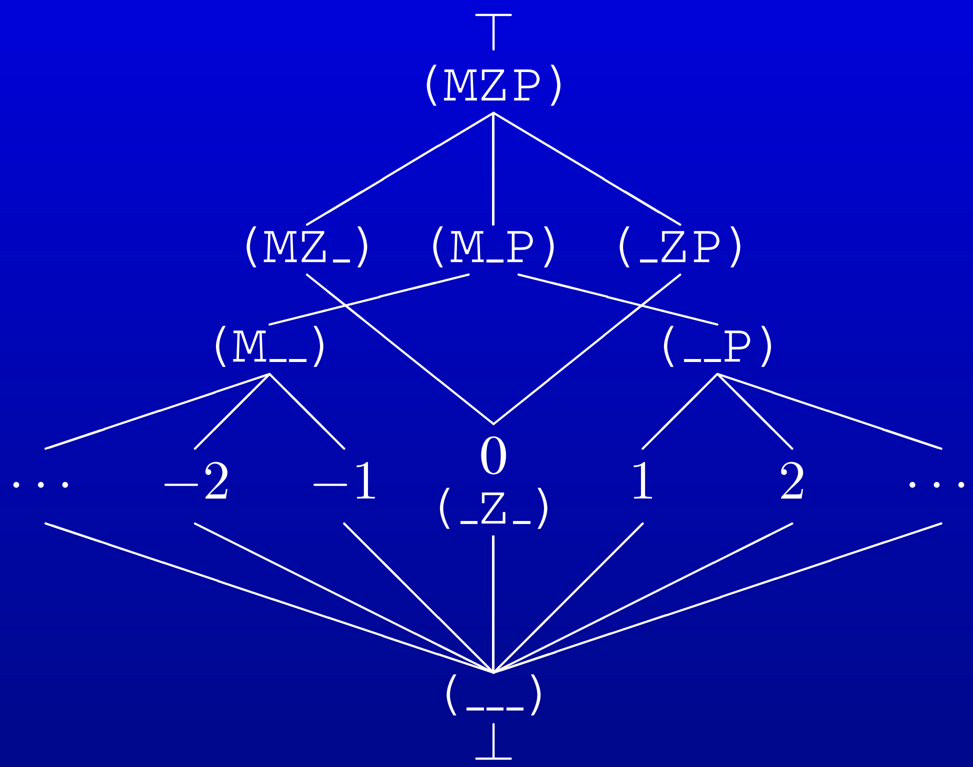
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\boxed{\mathtt{i} = \bot \sqcap 1 \sqcap 2}$$

Example redux, redux

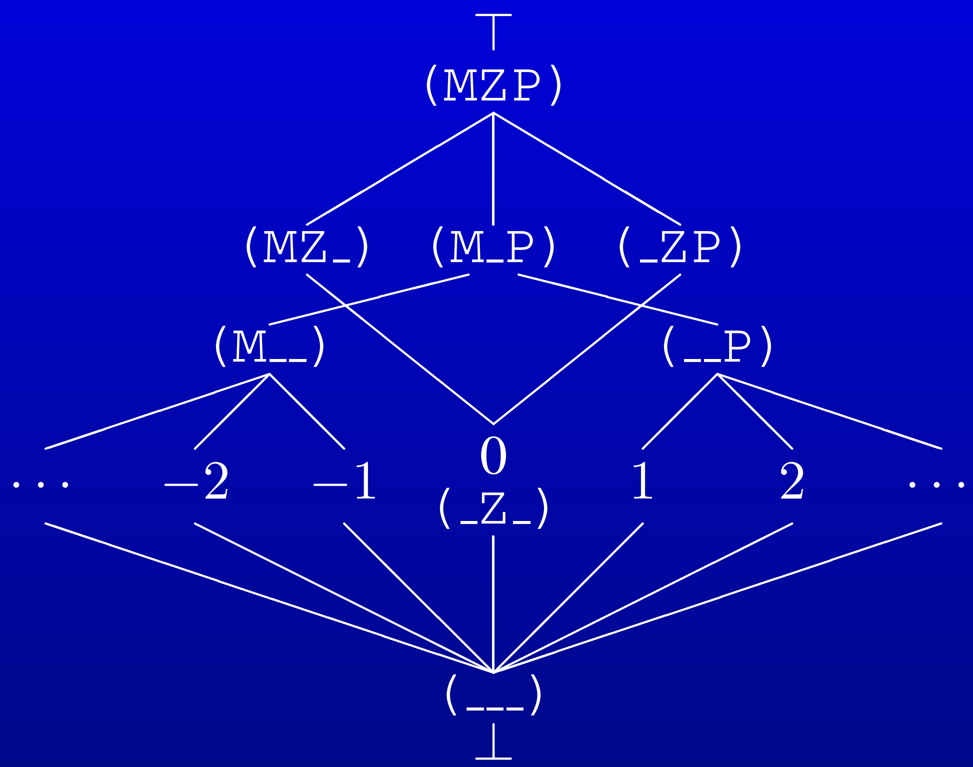
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\boxed{\mathtt{i} = \perp \sqcap 1 \sqcap 2 = \langle 0, 2 \rangle}$$

Example redux, redux

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



$$\boxed{\mathtt{i} = \bot \sqcap 1 \sqcap 2 = \langle 0, 2 \rangle}$$

Bitwidth combination rules

Bitwidth combination rules

- $\langle m, p \rangle \quad = \quad \langle p, m \rangle$

Bitwidth combination rules

$$
\begin{aligned}
-\langle m, p\rangle &= \langle p, m\rangle \\
\langle m_l, p_l\rangle + \langle m_r, p_r\rangle &= \langle 1+\max(m_l, m_r), 1+\max(p_l, p_r)\rangle
\end{aligned}
$$

Bitwidth combination rules

$$
\begin{aligned}
\text{-}\ \langle m, p \rangle &= \langle p, m \rangle \\
\langle m_l, p_l \rangle + \langle m_r, p_r \rangle &= \langle 1 + \max(m_l, m_r), 1 + \max(p_l, p_r) \rangle \\
\langle m_l, p_l \rangle * \langle m_r, p_r \rangle &= \left\langle \begin{array}{l} \max(m_l + p_r, p_l + m_r), \\ \max(m_l + m_r, p_l + p_r) \end{array} \right\rangle
\end{aligned}
$$

Bitwidth combination rules

$$
\begin{array}{rcl}
\text{-}\ \langle m, p\rangle & = & \langle p, m\rangle \\
\langle m_l, p_l\rangle + \langle m_r, p_r\rangle & = & \langle 1+\max(m_l, m_r), 1+\max(p_l, p_r)\rangle \\
\langle m_l, p_l\rangle\ \text{*}\ \langle m_r, p_r\rangle & = & \left\langle \begin{array}{l} \max(m_l+p_r, p_l+m_r), \\ \max(m_l+m_r, p_l+p_r) \end{array} \right\rangle \\
\langle 0, p_l\rangle\ \&\ \langle 0, p_r\rangle & = & \langle 0, \min(p_l, p_r)\rangle \\
\langle m_l, p_l\rangle\ \&\ \langle m_r, p_r\rangle & = & \langle \max(m_l, m_r), \max(p_l, p_r)\rangle
\end{array}
$$

Bitwise-AND, continued

$$\textbf{bw}(n) = \begin{cases} \langle 0, 0 \rangle & n = 0 \\ \langle 0, 1 + \lfloor \ln |n| \rfloor \rangle & n > 0 \\ \langle 1 + \lfloor \ln |n| \rfloor, 0 \rangle & n < 0 \end{cases}$$

Bitwise-AND, continued

$$\textbf{bw}(n) = \begin{cases} \langle 0, 0 \rangle & n = 0 \\ \langle 0, 1 + \lfloor \ln |n| \rfloor \rangle & n > 0 \\ \langle 1 + \lfloor \ln |n| \rfloor, 0 \rangle & n < 0 \end{cases}$$

$\texttt{0...0}\overbrace{\texttt{1X...XXX}}^{1+\lfloor \ln|n| \rfloor}$ Positive

$\texttt{1...1}\overbrace{\texttt{0X...XXXX}}^{1+\lfloor \ln(|n|-1) \rfloor}$ Negative

Bitwise-AND, continued

$$\textbf{bw}(n) = \begin{cases} \langle 0, 0 \rangle & n = 0 \\ \langle 0, 1 + \lfloor \ln |n| \rfloor \rangle & n > 0 \\ \langle 1 + \lfloor \ln |n| \rfloor, 0 \rangle & n < 0 \end{cases}$$

$$\texttt{0...0}\,\overbrace{\texttt{1X...XXX}}^{1+\lfloor \ln |n| \rfloor = p}$$

Positive

$$\texttt{1...1}\,\overbrace{\texttt{0X...XXXX}}^{1+\lfloor \ln (|n|-1) \rfloor}$$

Negative

Bitwise-AND, continued

$$\textbf{bw}(n) = \begin{cases} \langle 0, 0 \rangle & n = 0 \\ \langle 0, 1 + \lfloor \ln|n| \rfloor \rangle & n > 0 \\ \langle 1 + \lfloor \ln|n| \rfloor, 0 \rangle & n < 0 \end{cases}$$

$$\texttt{0...0}\,\overbrace{\texttt{1X...XXX}}^{1+\lfloor \ln|n| \rfloor = p}$$

Positive

$$\texttt{1...1}\,\overbrace{\texttt{0X...XXXX}}^{1+\lfloor \ln(|n|-1) \rfloor \leq m}$$

Negative

Bitwise-AND, continued

$$\textbf{bw}(n) = \begin{cases} \langle 0, 0\rangle & n = 0 \\ \langle 0, 1 + \lfloor \ln|n| \rfloor\rangle & n > 0 \\ \langle 1 + \lfloor \ln|n| \rfloor, 0\rangle & n < 0 \end{cases}$$

$$\texttt{0...0}\,\overbrace{\texttt{1X...XXX}}^{1+\lfloor \ln|n| \rfloor = p} \qquad\qquad \texttt{1...1}\,\overbrace{\texttt{0X...XXXX}}^{1+\lfloor \ln(|n|-1) \rfloor \leq m}$$

Positive Negative

$$\begin{aligned} \langle 0, p_l\rangle \mathbin{\&} \langle 0, p_r\rangle &= \langle 0, \min(p_l, p_r)\rangle \\ \langle m_l, p_l\rangle \mathbin{\&} \langle m_r, p_r\rangle &= \langle \max(m_l, m_r), \max(p_l, p_r)\rangle \end{aligned}$$

Interprocedural analysis

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

Interprocedural analysis

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

Interprocedural analysis

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

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