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

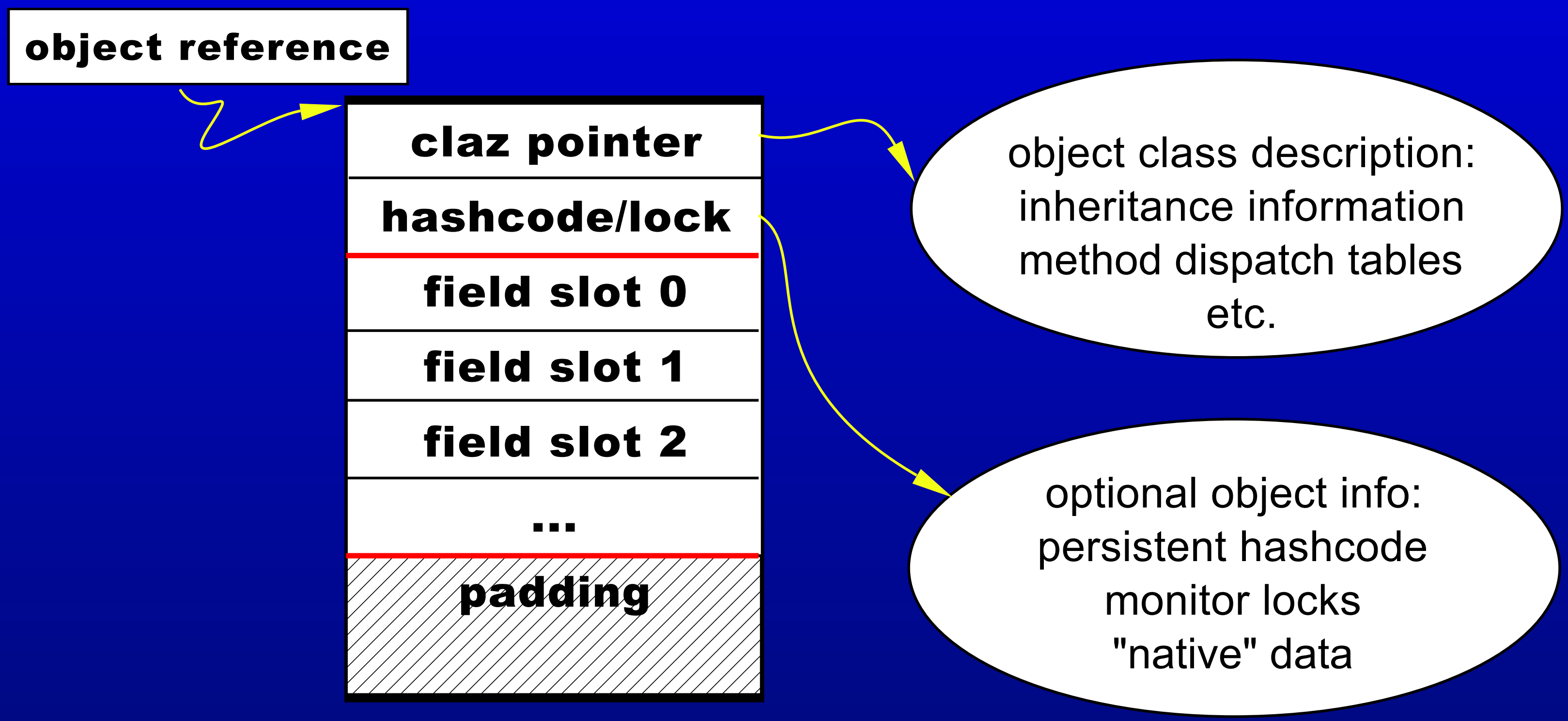
Using program analysis to identify opportunities to reduce the space required to store objects,

Then

Applying transformations to reduce the memory consumption of the program.

Structure of a Java Object

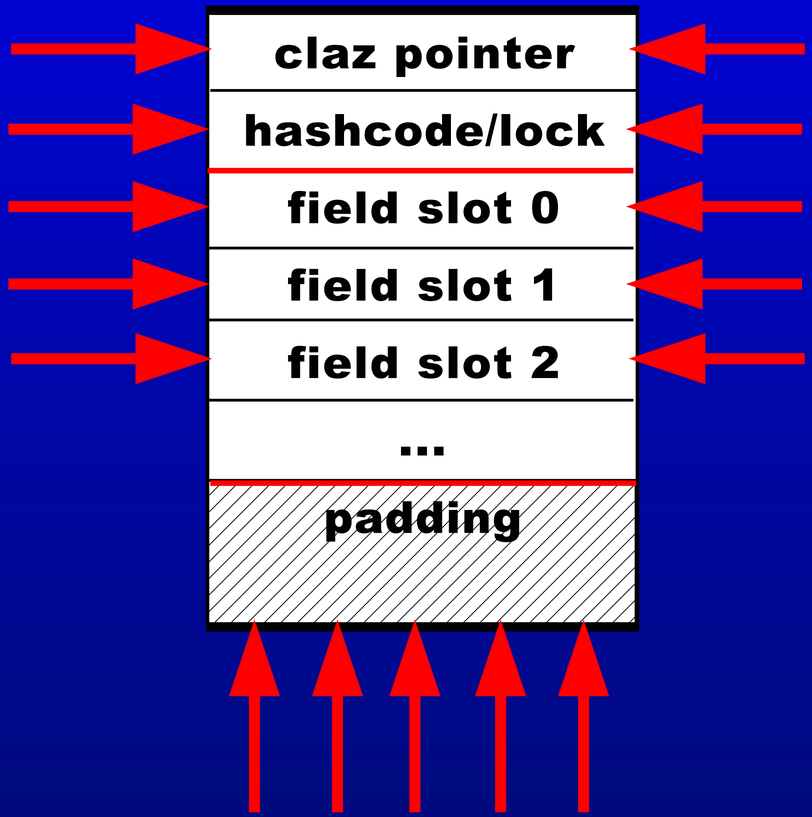
- Typical of many O-O languages.



Strategy

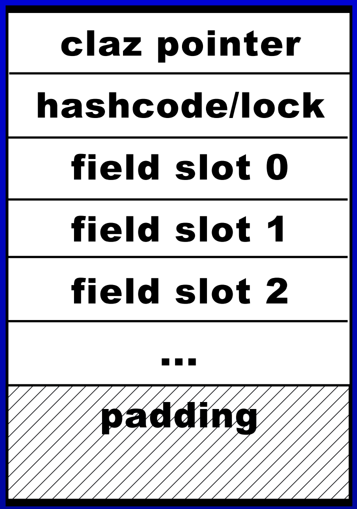
Strategy

Push hard on all the bits.



How to compress objects

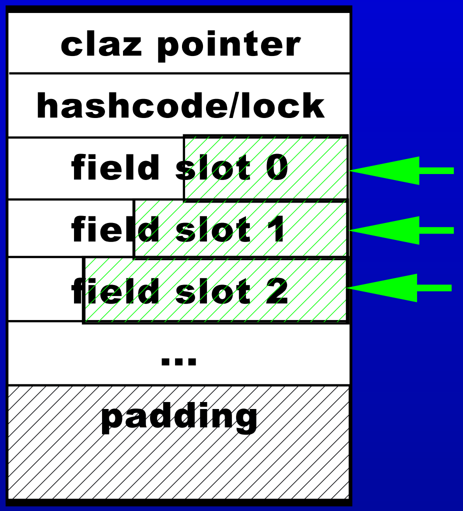
Three broad techniques:



How to compress objects

Three broad techniques:

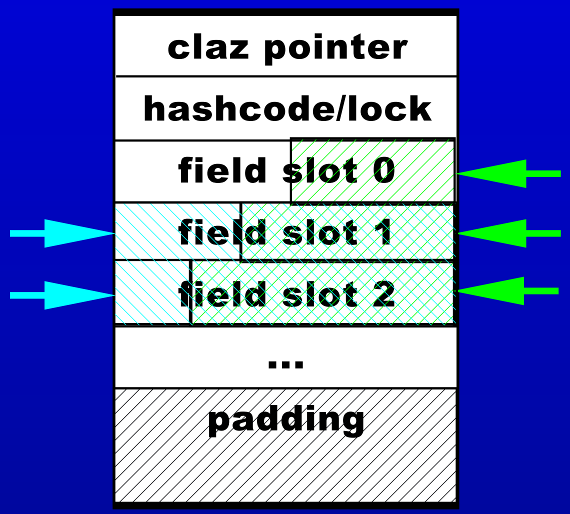
- Field compression



How to compress objects

Three broad techniques:

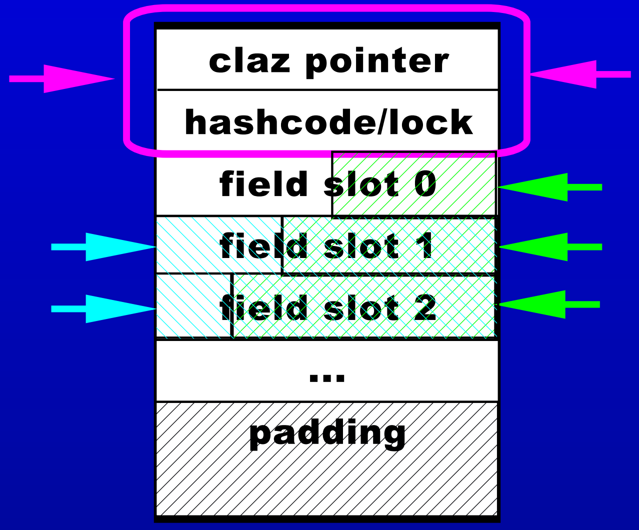
- Field compression
- Mostly-constant field elimination



How to compress objects

Three broad techniques:

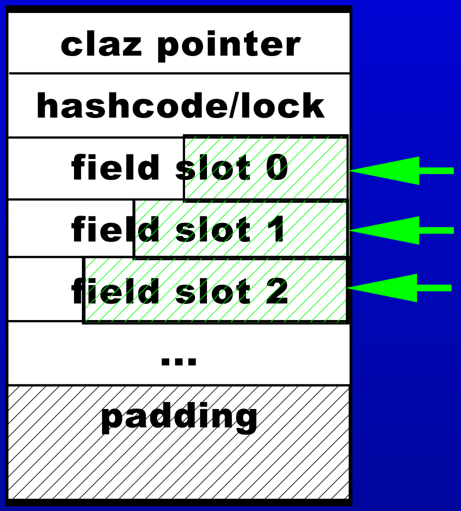
- Field compression
- Mostly-constant field elimination
- Header optimizations



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

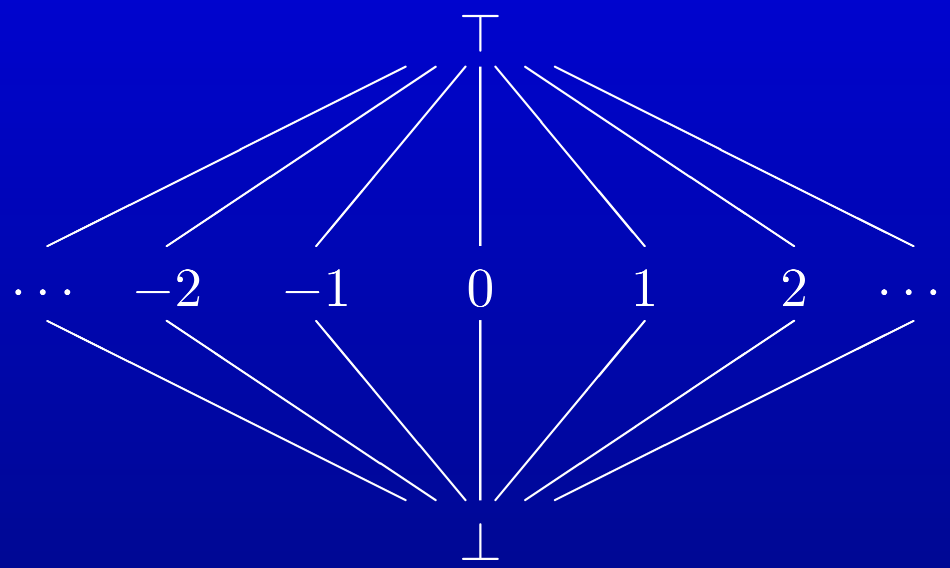
How are these analyses performed?

Intraprocedural Analysis

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

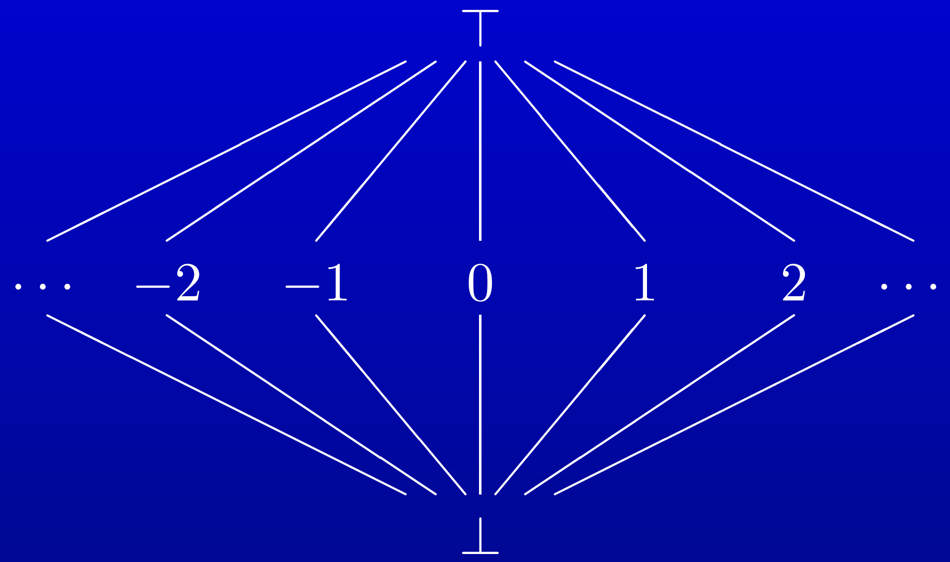
Intraprocedural Analysis

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



Intraprocedural Analysis

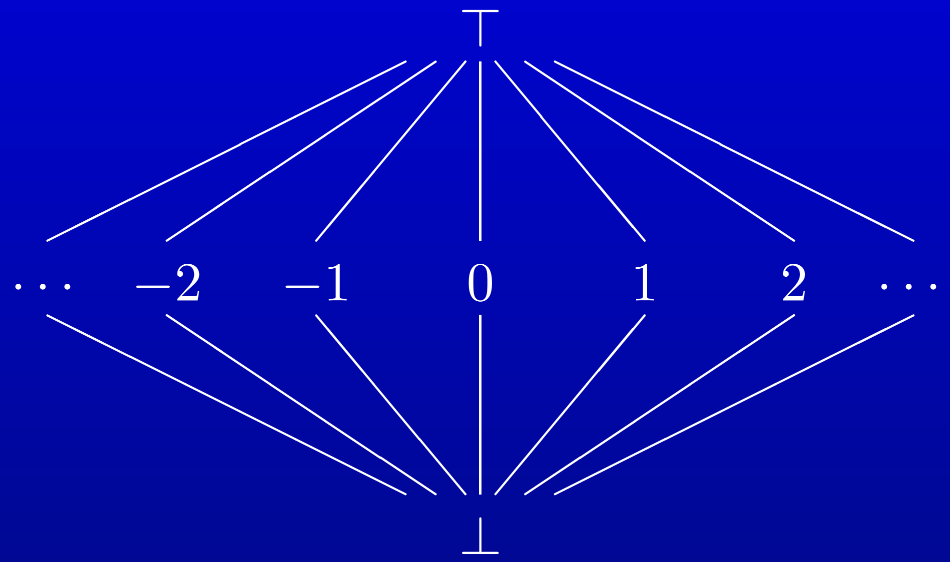
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\mathtt{i} = \bot$

Intraprocedural Analysis

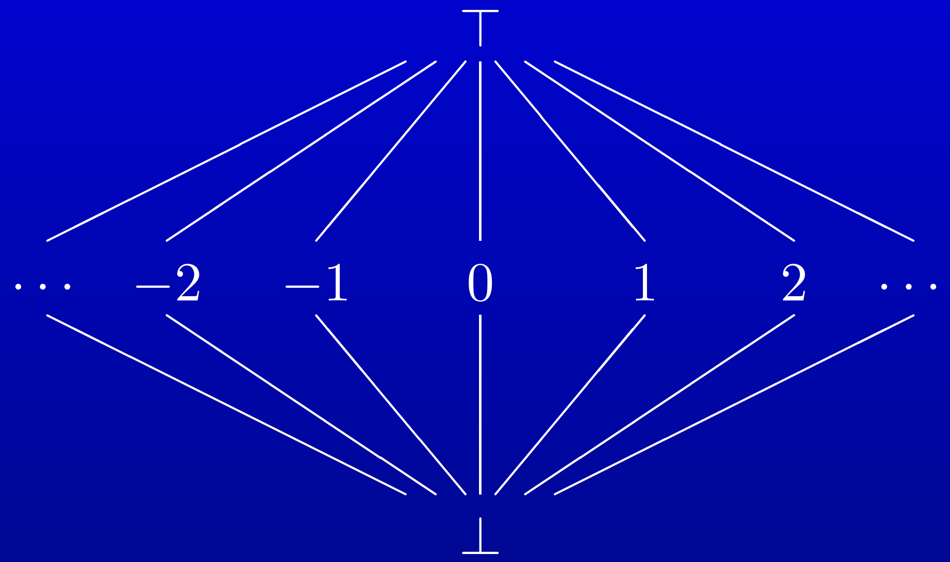
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\mathtt{i} = \bot$$

Intraprocedural Analysis

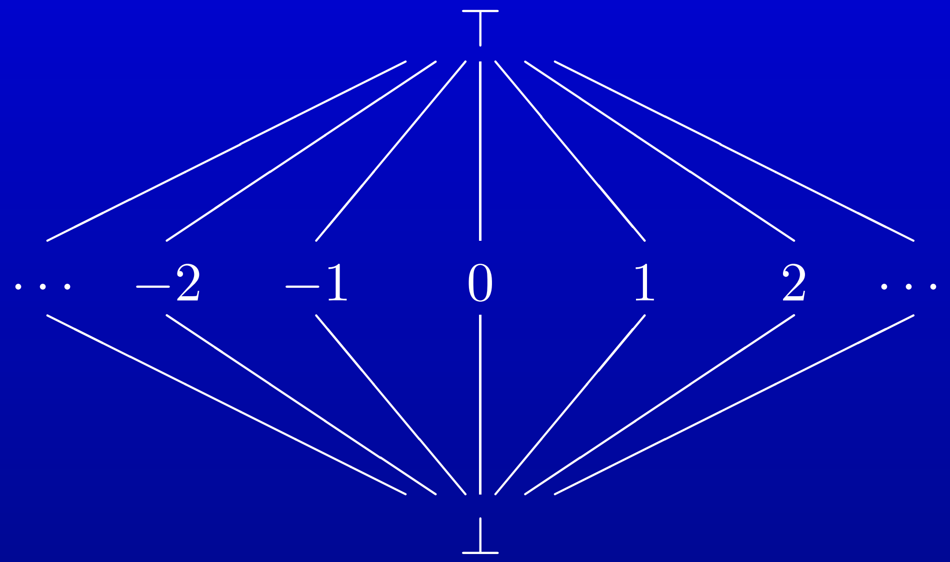
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



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

Intraprocedural Analysis

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

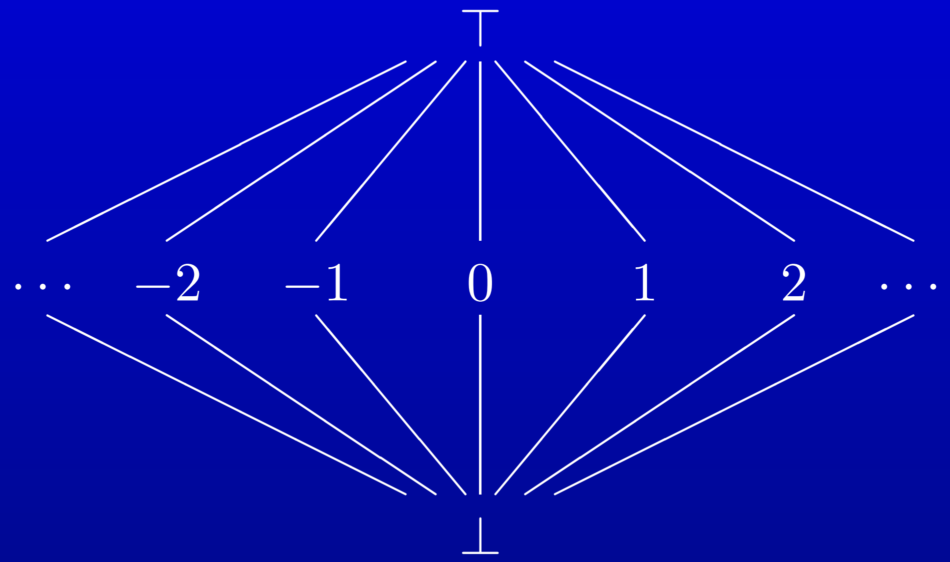


$\mathbf{i} = 1$

[Because $\bot \sqsubseteq 1$ and $1 \sqsubseteq 1$]

Intraprocedural Analysis

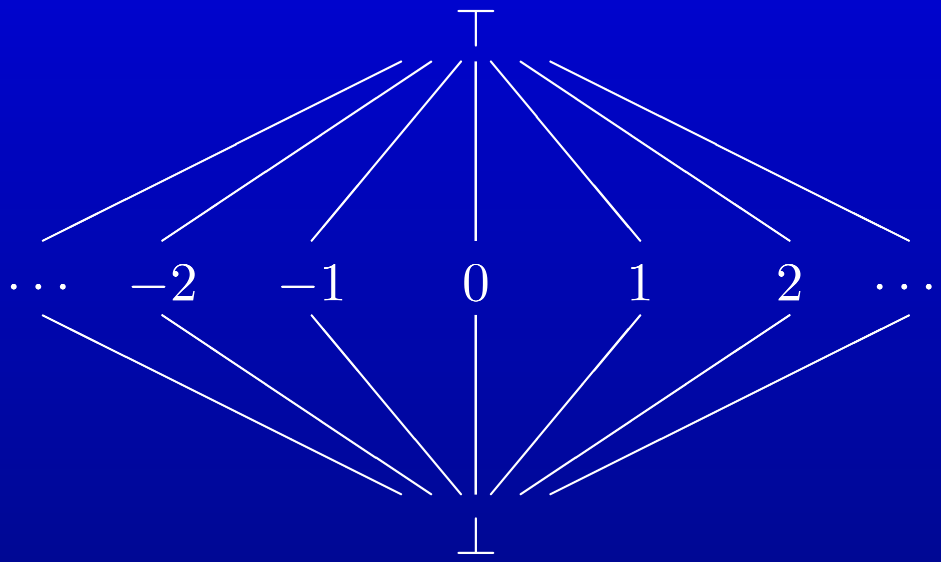
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\boxed{\mathtt{i} = 1}$

Intraprocedural Analysis

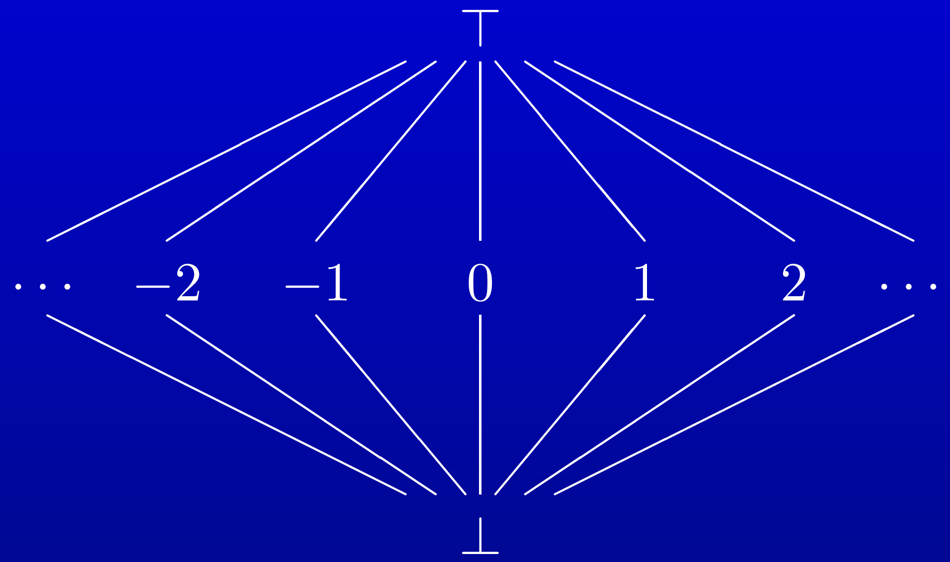
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



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

Intraprocedural Analysis

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

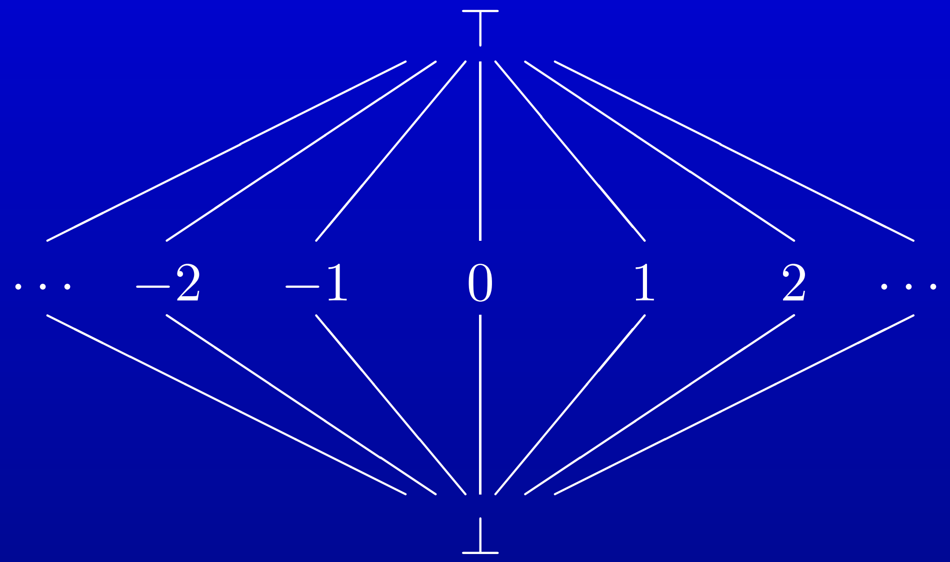


$$\mathbf{i} = 1 \sqcap 2 = \top$$

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

Intraprocedural Analysis

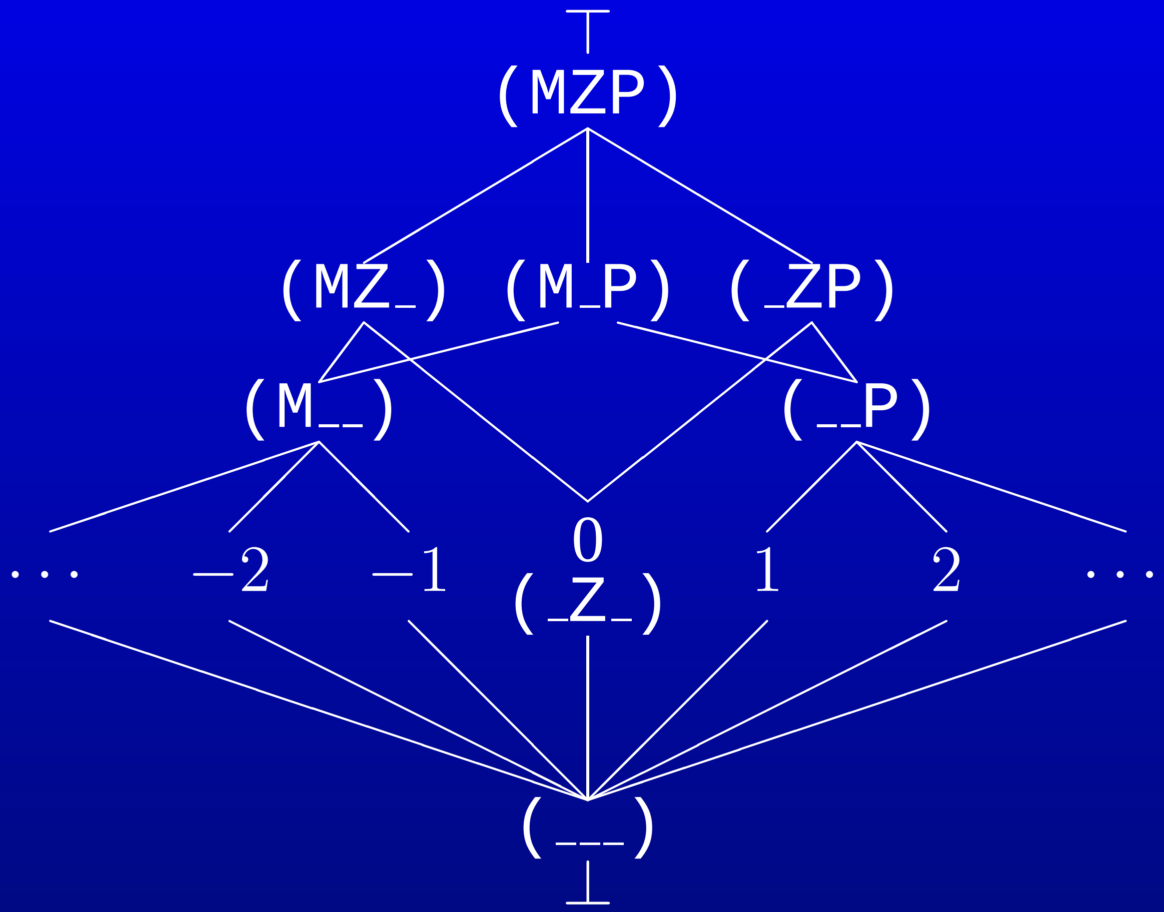
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



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

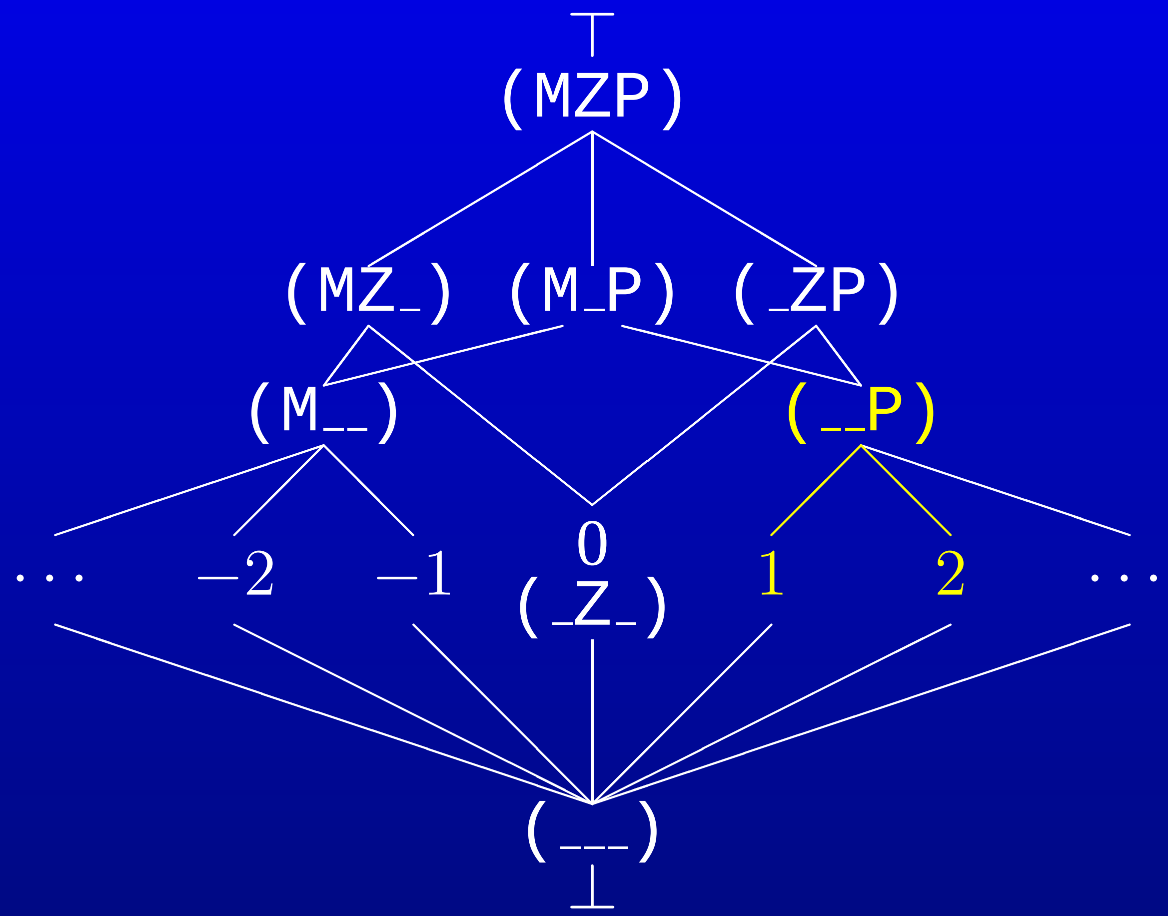
[Because $1 \sqsubseteq \top$ and $2 \sqsubseteq \top$]

A signed integer lattice



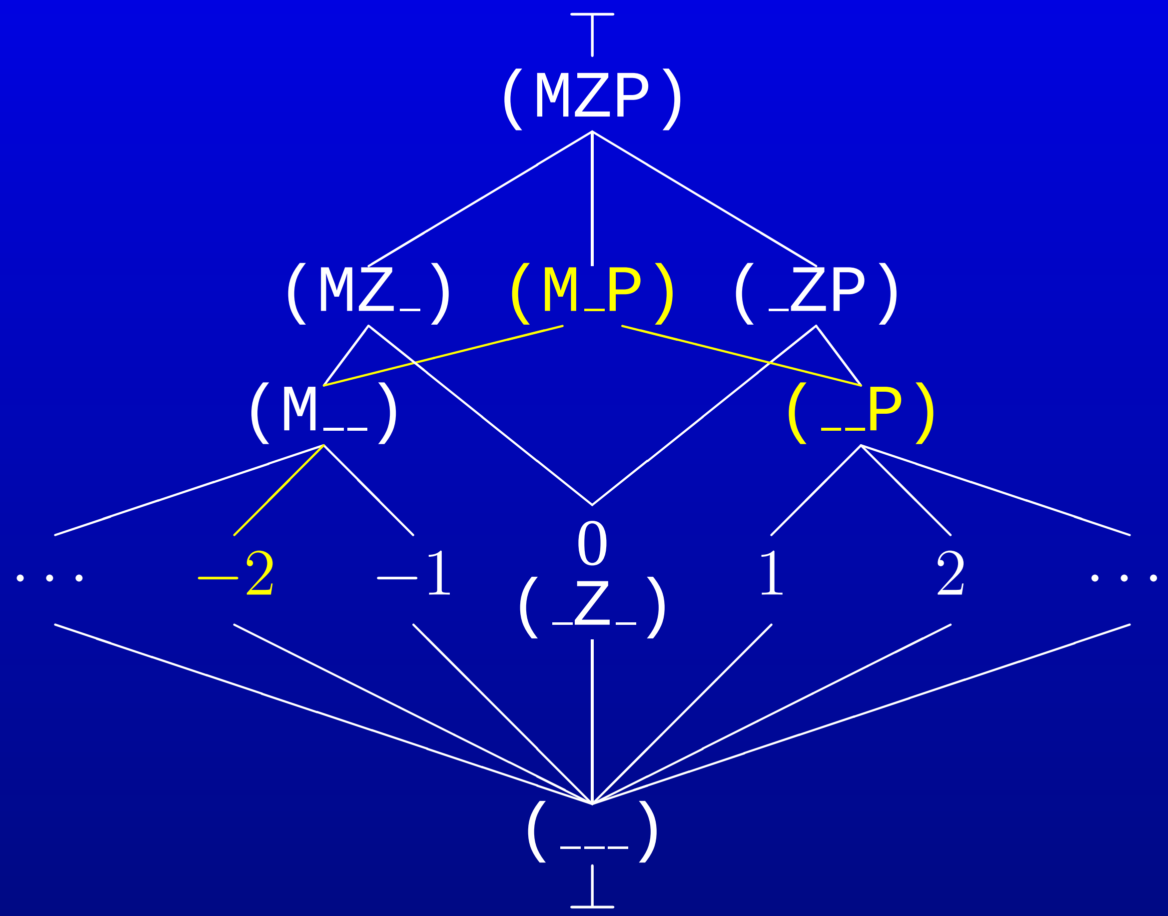
An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain.

A signed integer lattice



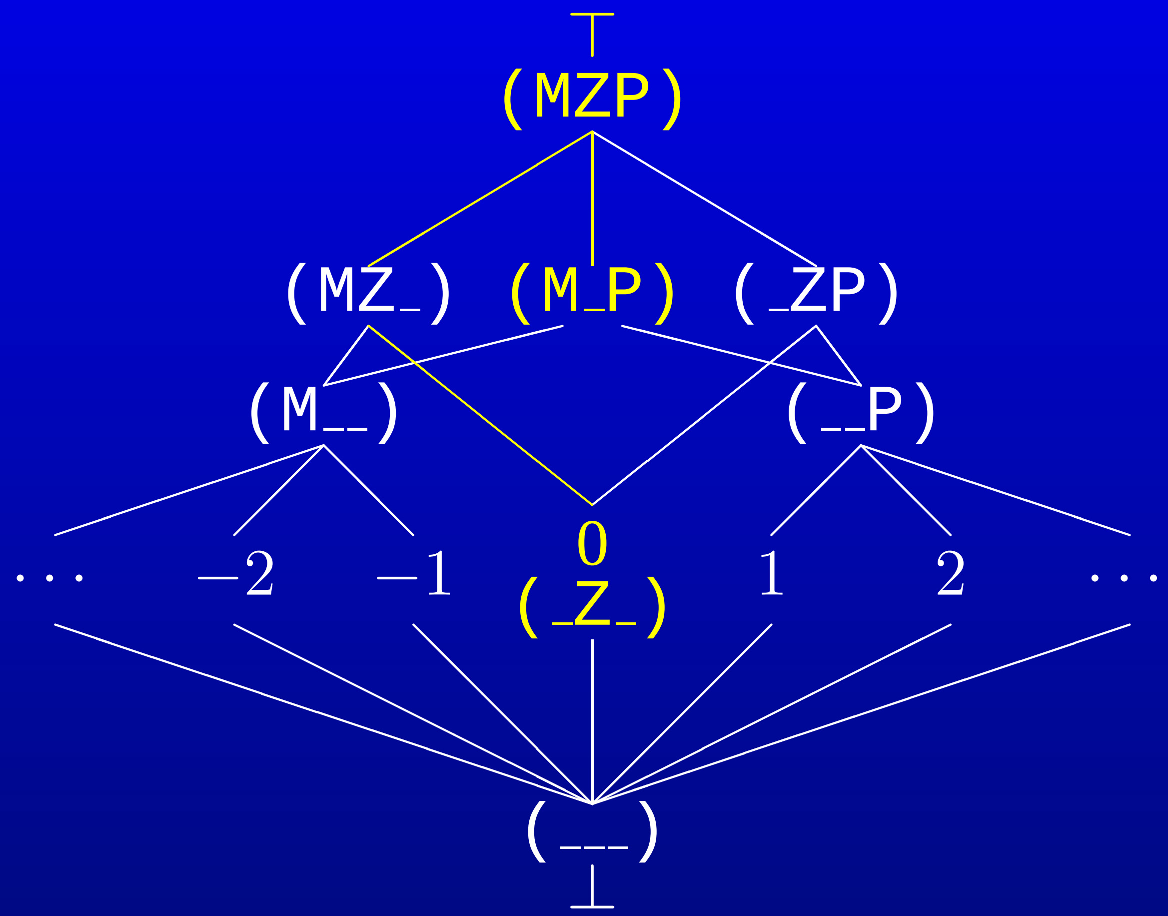
An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain.

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.

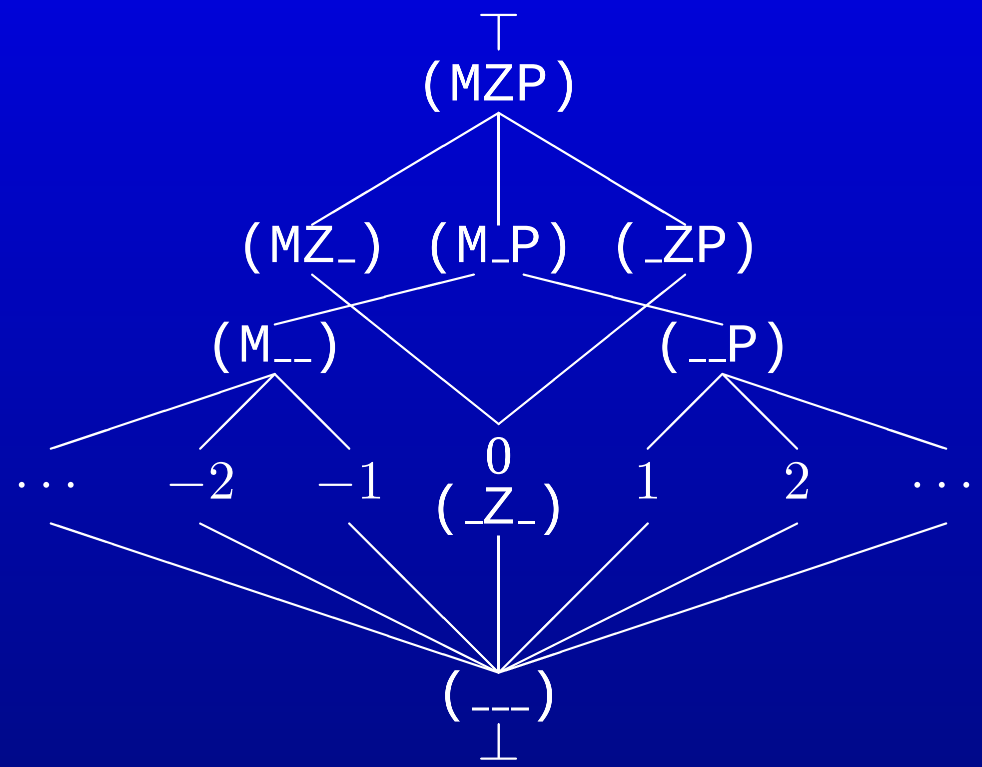
A signed integer lattice



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

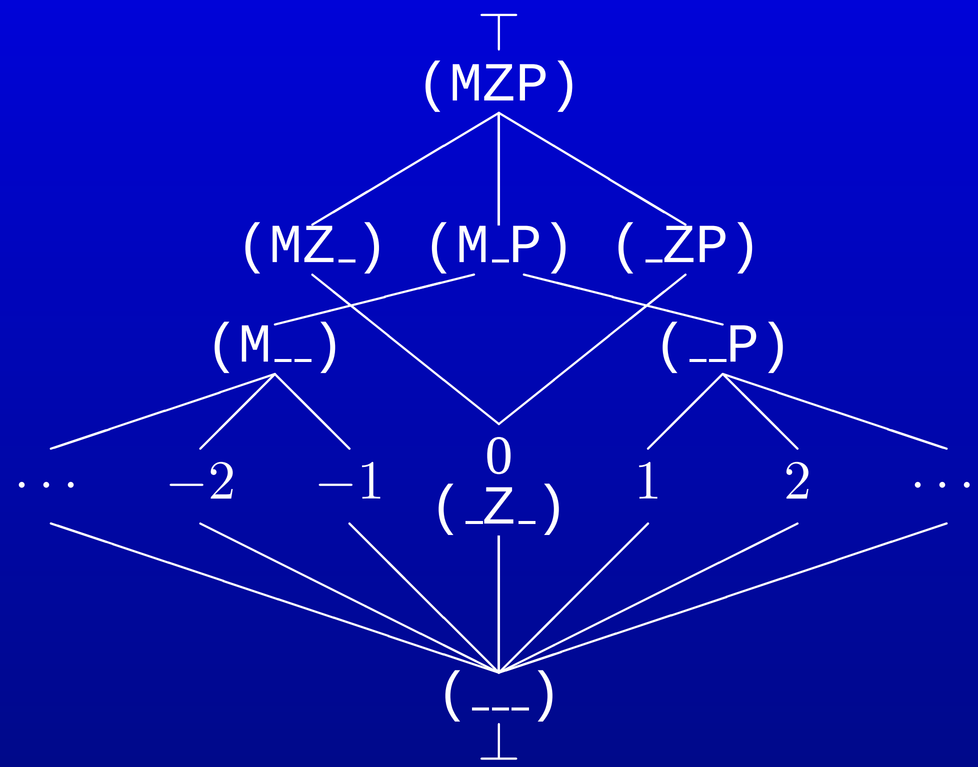
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\mathtt{i} = \bot$

Example, redux

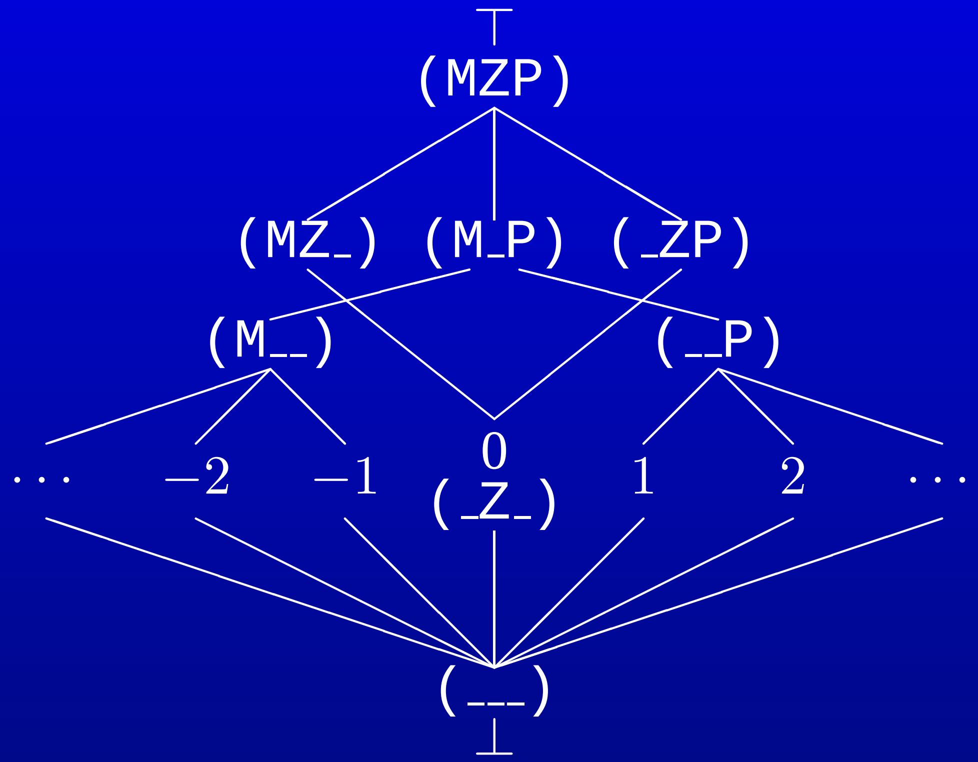
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$i = \bot$$

Example, redux

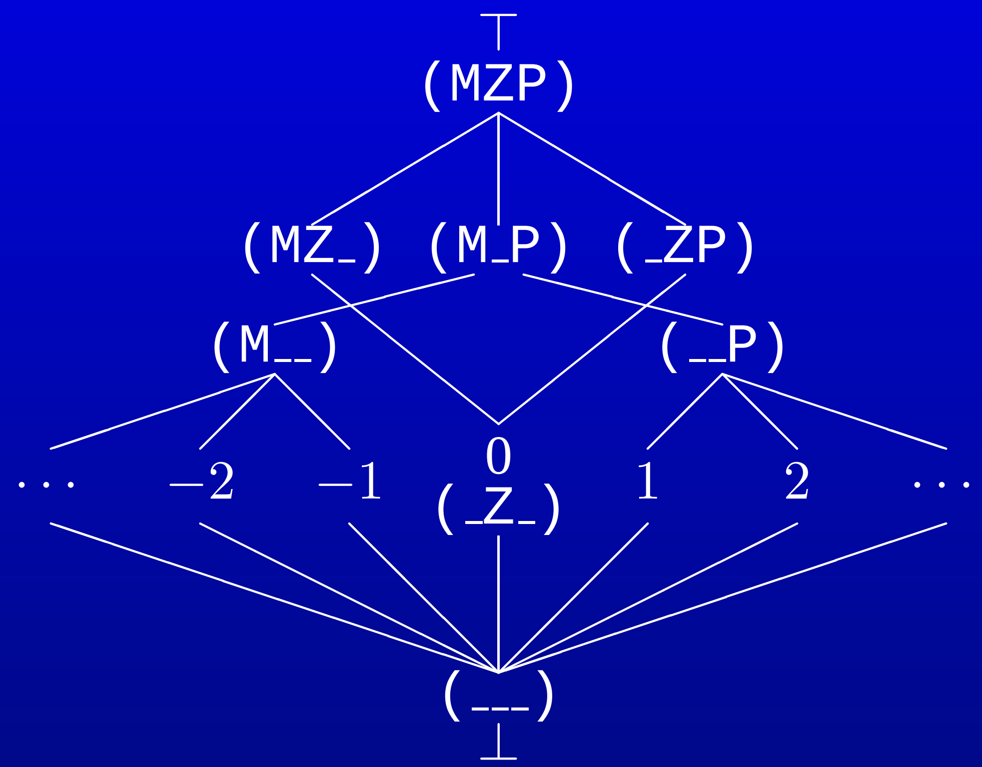
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



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

Example, redux

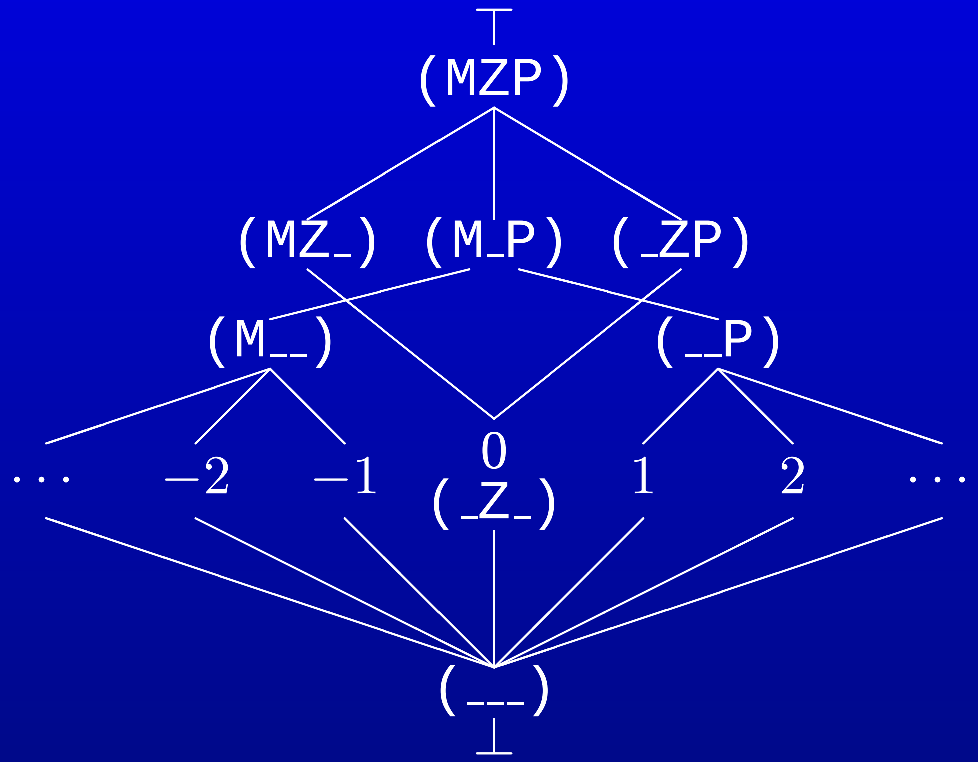
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\boxed{\mathbf{i} = 1}$$

Example, redux

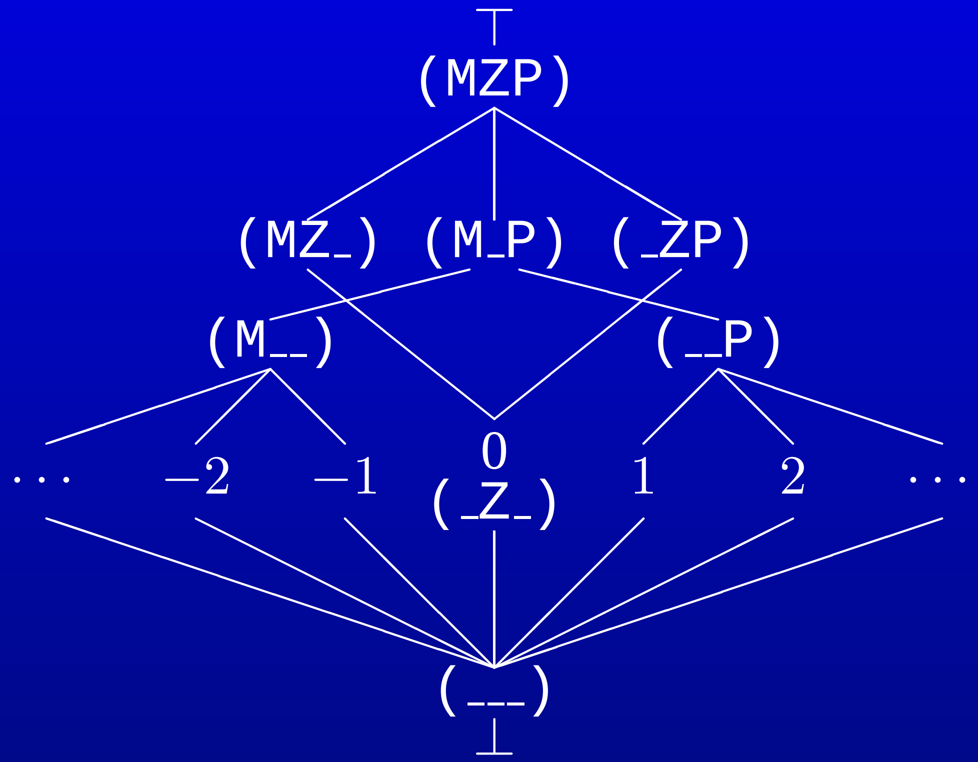
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



i = 1

Example, redux

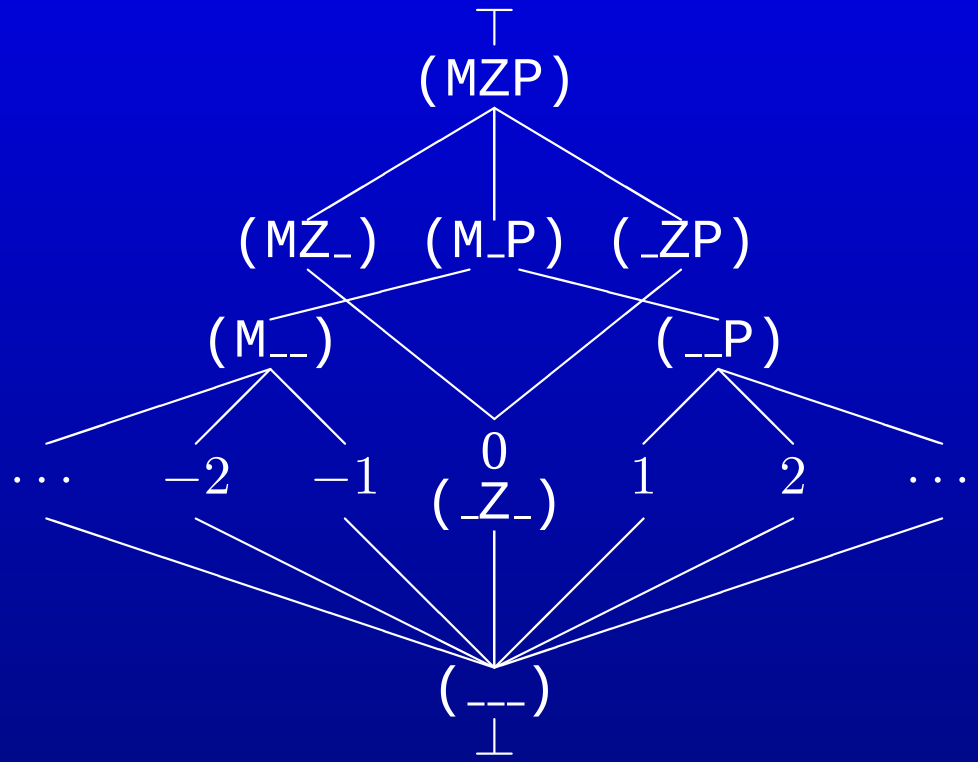
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\mathbf{i} = 1 \sqcap 2$$

Example, redux

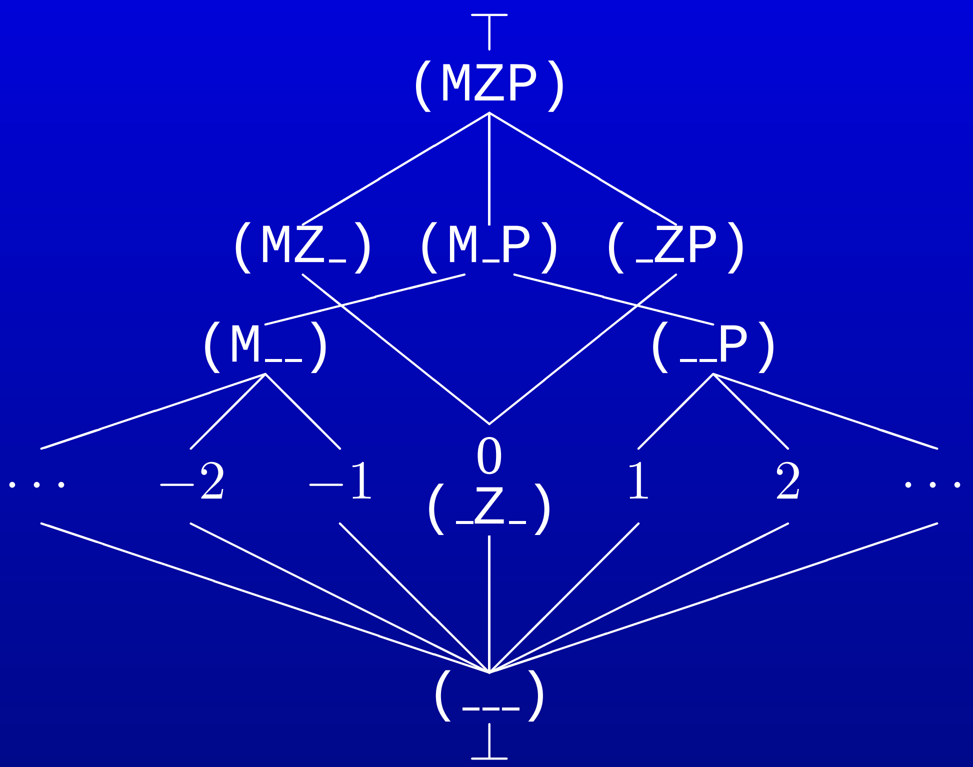
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



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

Example, redux

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



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

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 $m = p = 0$.

(__P) $\Rightarrow \langle 0, p \rangle$

(M__) $\Rightarrow \langle m, 0 \rangle$

(_Z_) $\Rightarrow \langle 0, 0 \rangle$

In lattice context: **(__P)** $\Rightarrow$

$\langle 0, 31 \rangle$

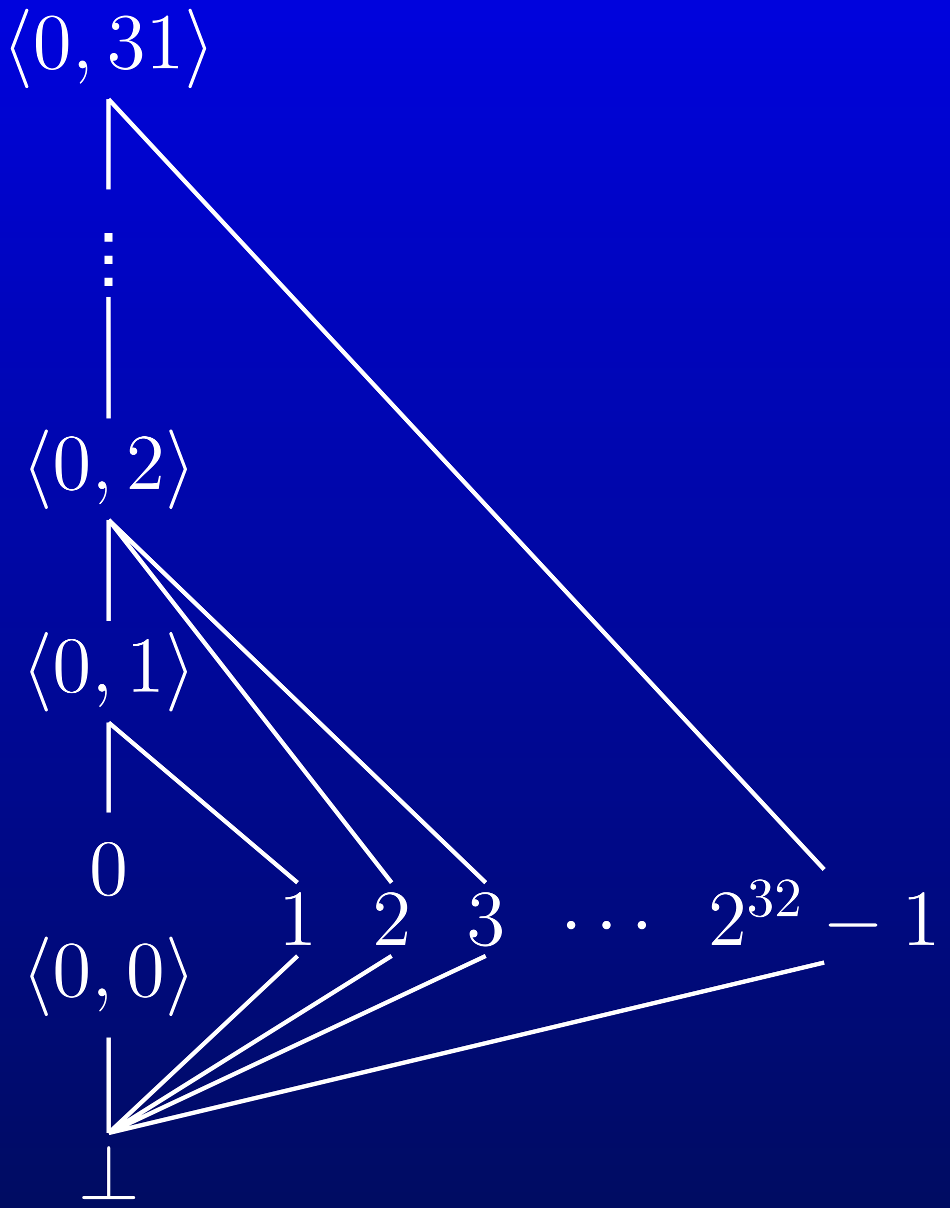
$\vdots$

$\langle 0, 3 \rangle$

$\langle 0, 2 \rangle$

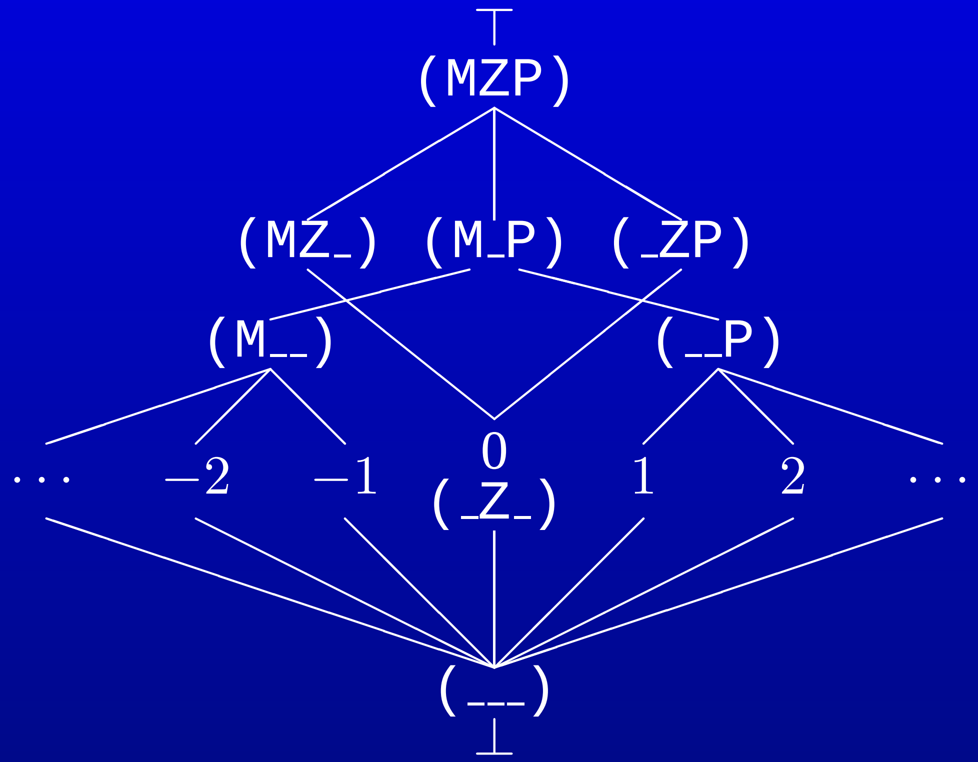
$\langle 0, 1 \rangle$

Bitwidth lattice detail



Example, redux redux

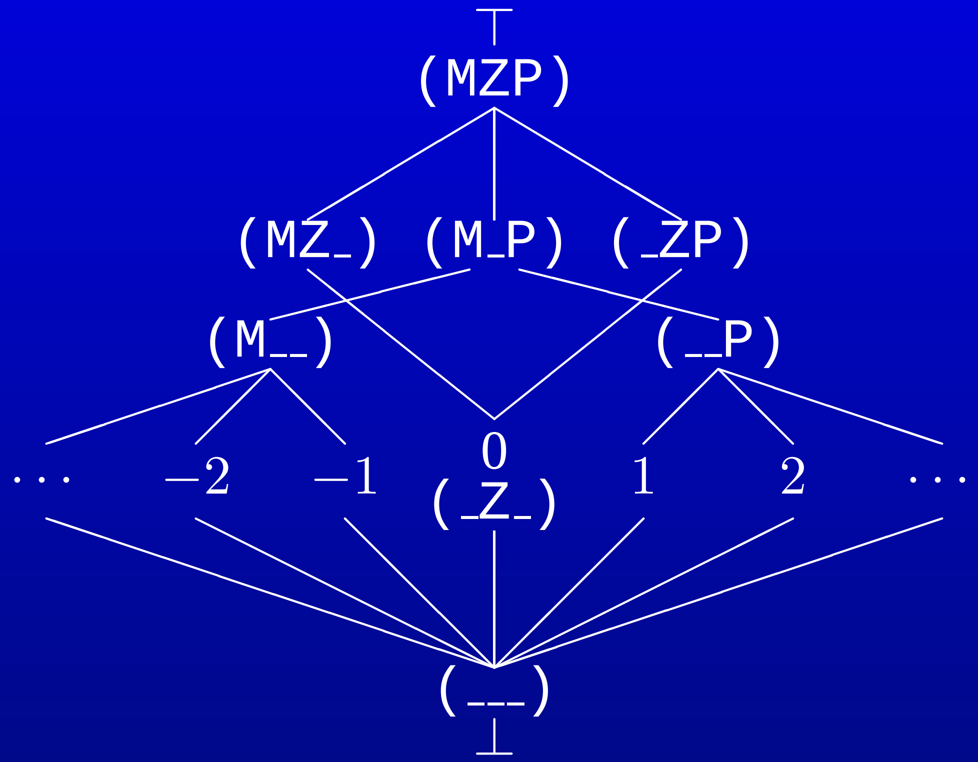
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\mathbf{i} = \bot$

Example, redux redux

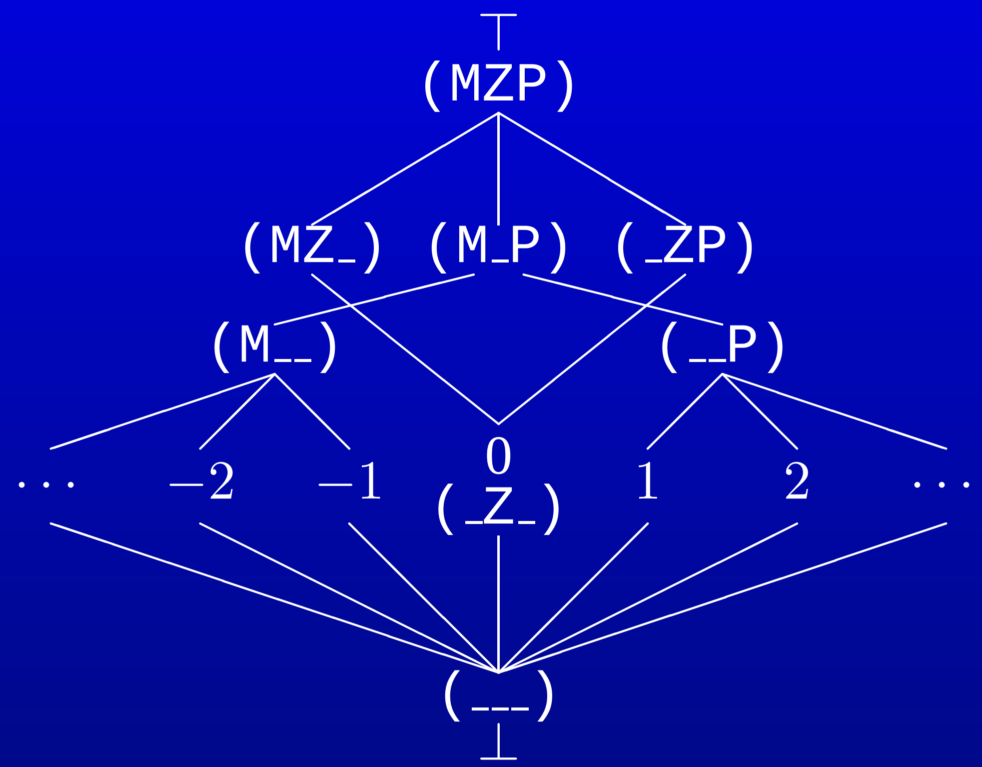
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\mathbf{i} = \bot$$

Example, redux redux

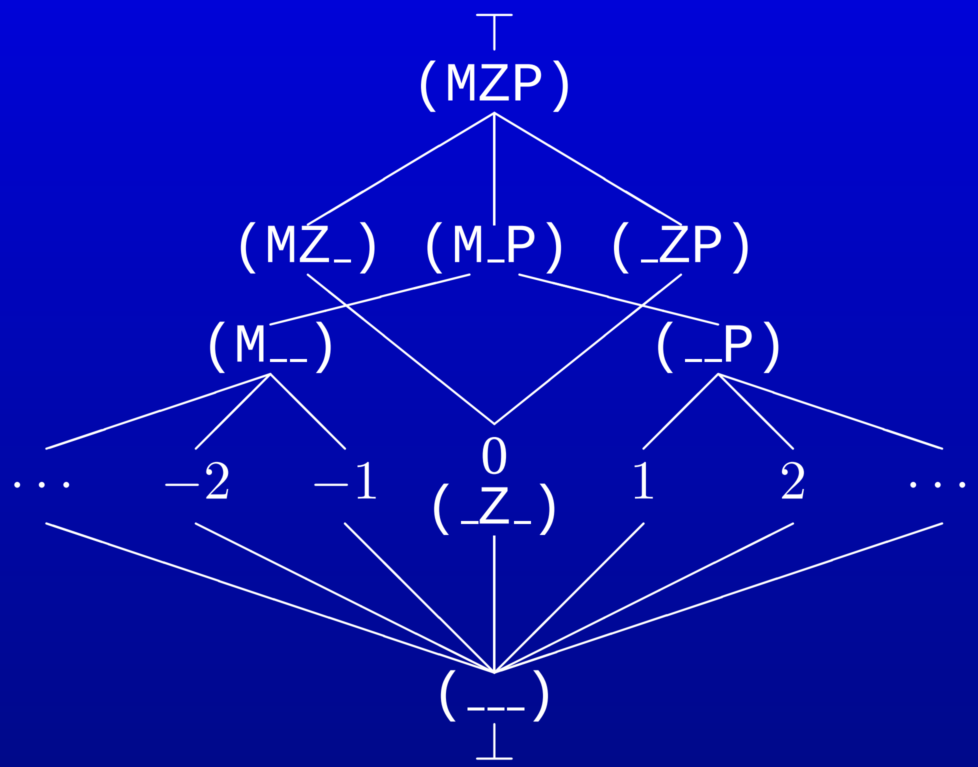
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\mathbf{i} = \bot \sqcap 1$$

Example, redux redux

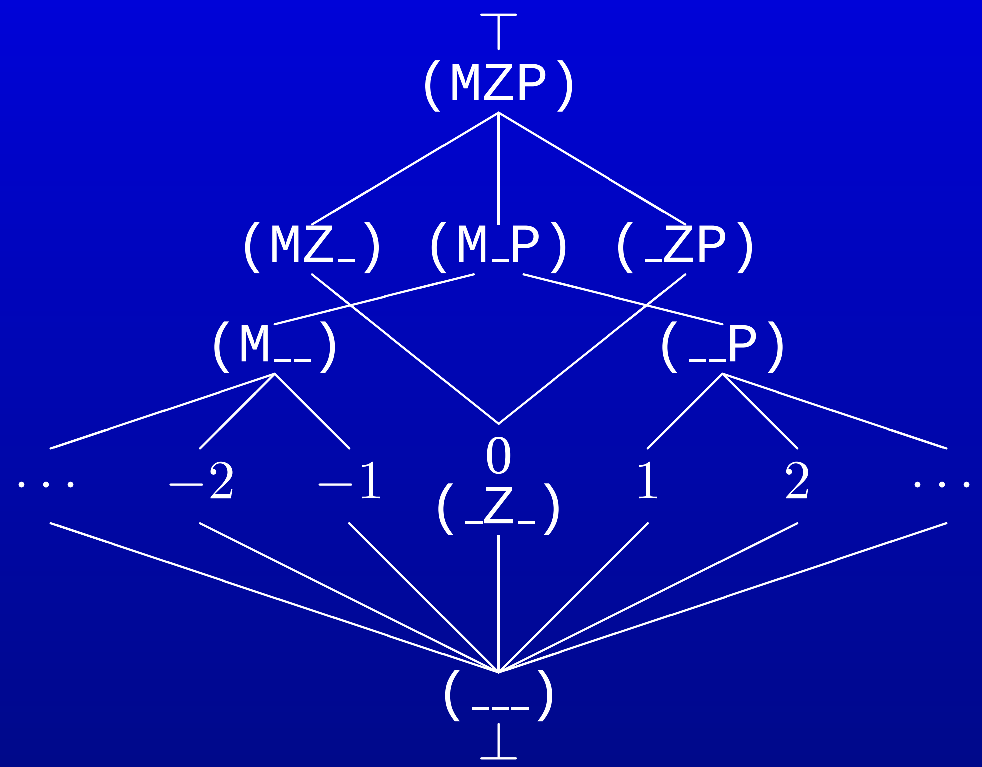
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



i $= 1$

Example, redux redux

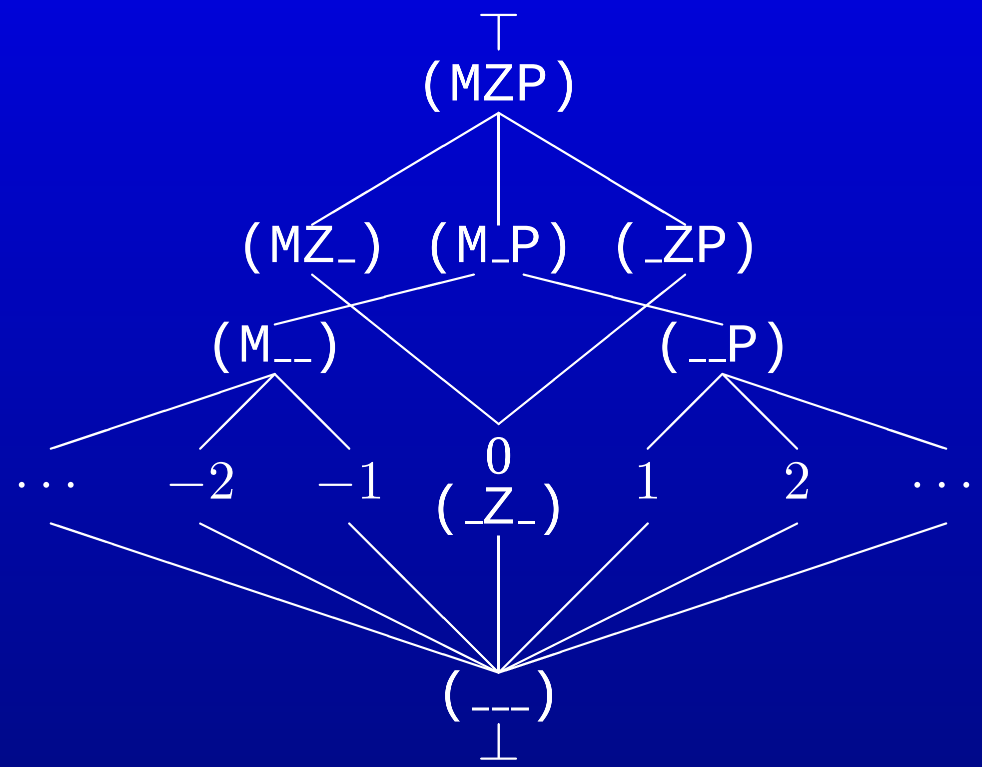
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\boxed{\mathtt{i} = 1}$$

Example, redux redux

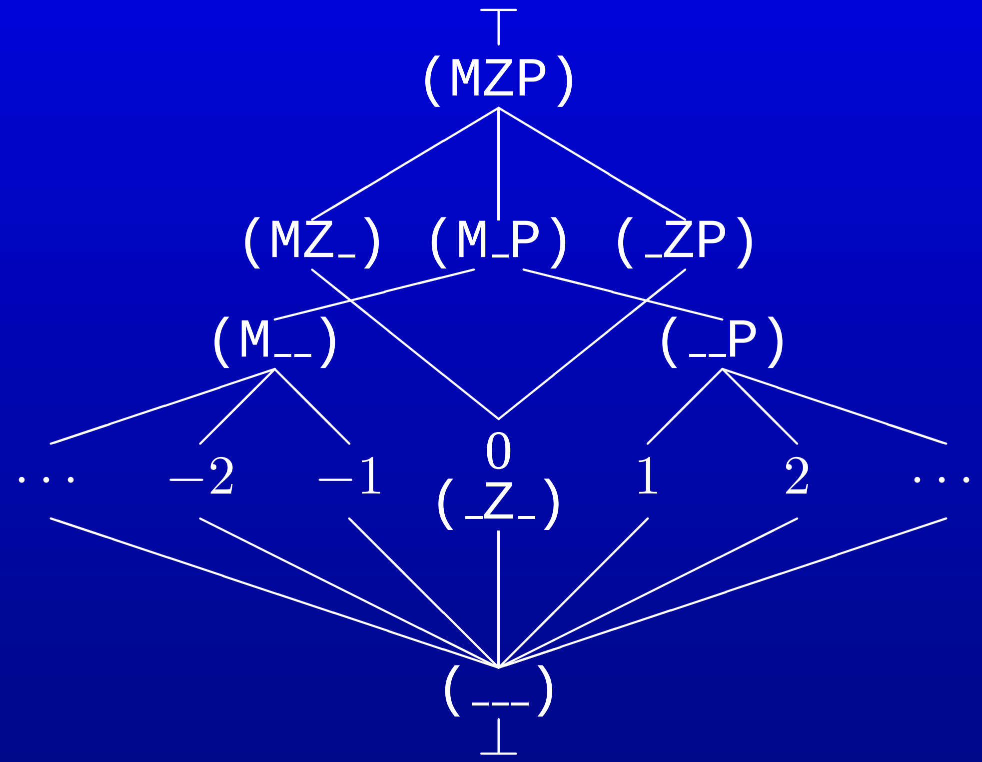
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$\mathbf{i} = 1 \sqcap 2$

Example, redux redux

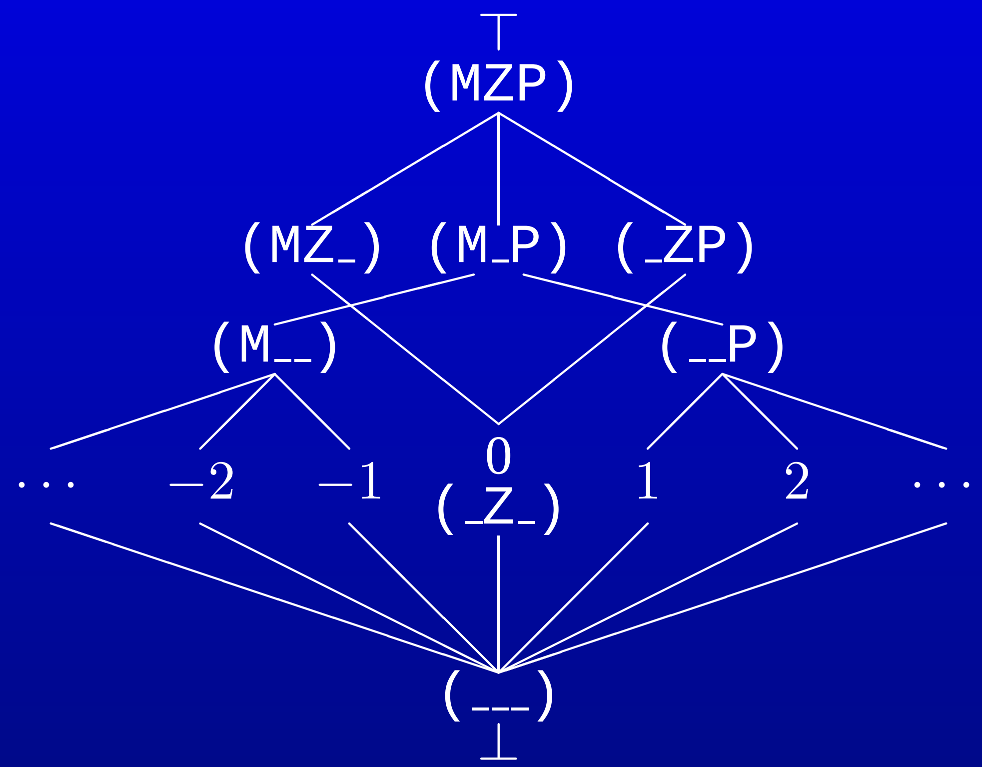
```
int foo() {
    if (...)
        i=1;
    else
        i=2;
    if (i>0)
        ⋮
}
```



$$\mathbf{i} = 1 \sqcap 2 = \langle 0, 2 \rangle$$

Example, redux redux

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



$$\mathbf{i} = 1 \sqcap 2 = \langle 0, 2 \rangle$$

Bitwidth combination rules

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

Some combination rules for bit-width analysis.

Interprocedural analysis

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

Interprocedural analysis

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

Interprocedural analysis

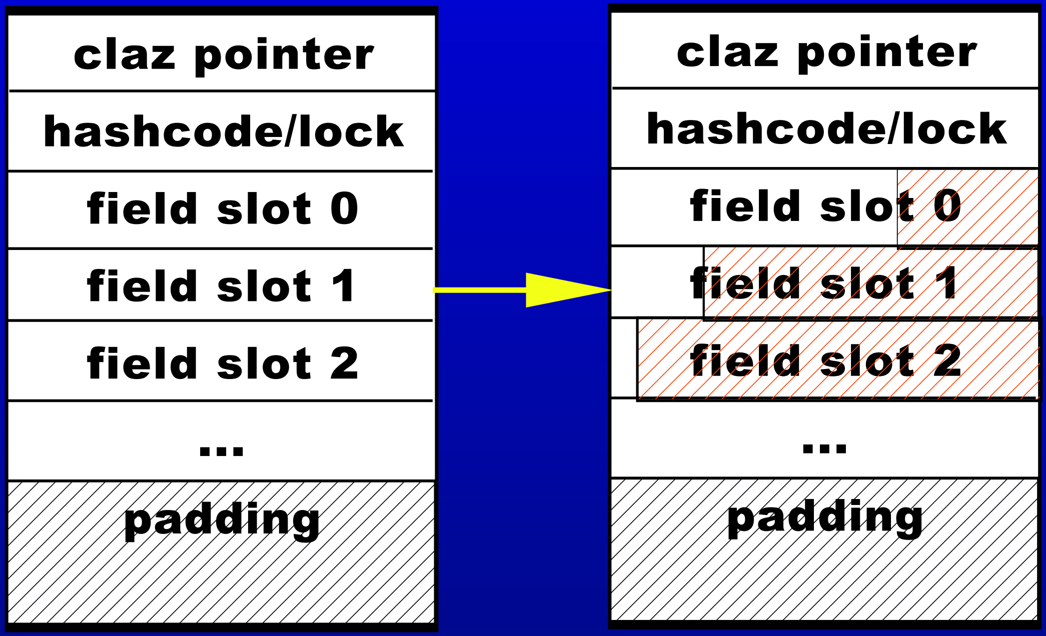
```
int foo() {
    if (...)
        this.f=1;
    else
        this.f=2;
    if (this.f>0)
        ⋮
}
```

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

All cars are black

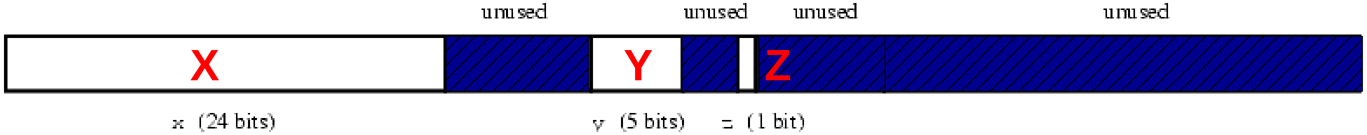
```
void paint(int color) {
    if (this.model == FORD)
        color = BLACK;
    this.color = color;
}
```

Field compression using bitwidths

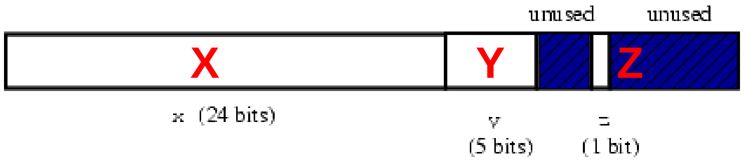


Field packing

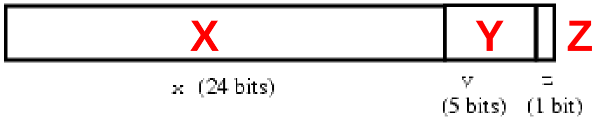
Standard packing word-aligns the object and aligns each field to the width of its type (4-byte data is 4-byte aligned):



"Byte" alignment byte-aligns the object and all fields:



"Bit" alignment requires no alignment of objects or fields:



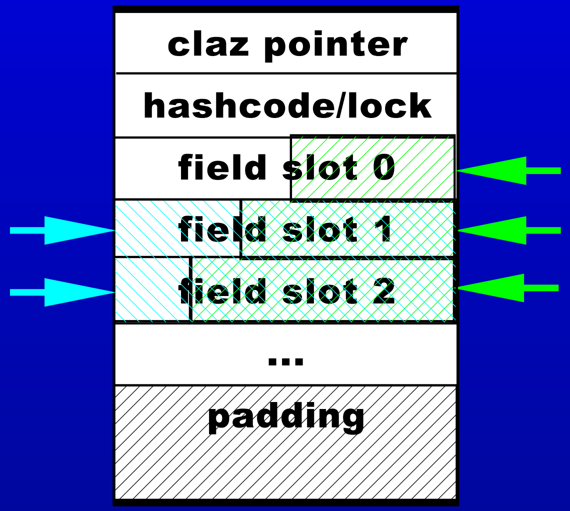

```
class A {
    int x; /* actual width 24 bits */
    byte y; /* actual width 5 bits */
    boolean z; /* actual width 1 bit */
}
```

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

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

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 $\perp$.
- To achieve our storage savings, we map $\perp$ to the frequent “mostly-constant” value.

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 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

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%

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

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.

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

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.
- Limits the number of fields which may be externalized, based on the size of the object.

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.
- 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.

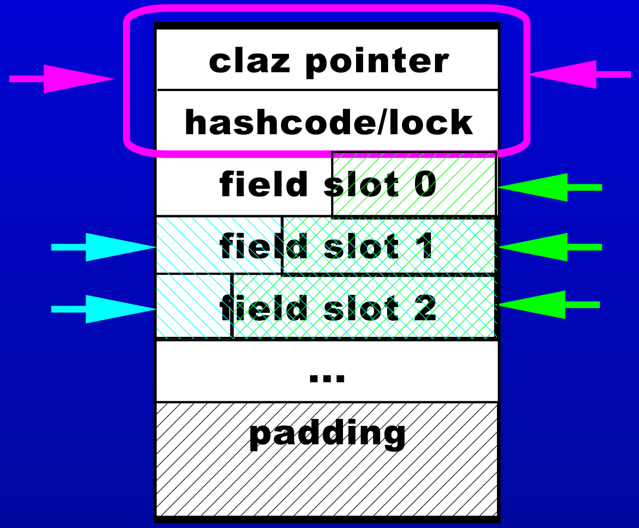
Other details

- 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

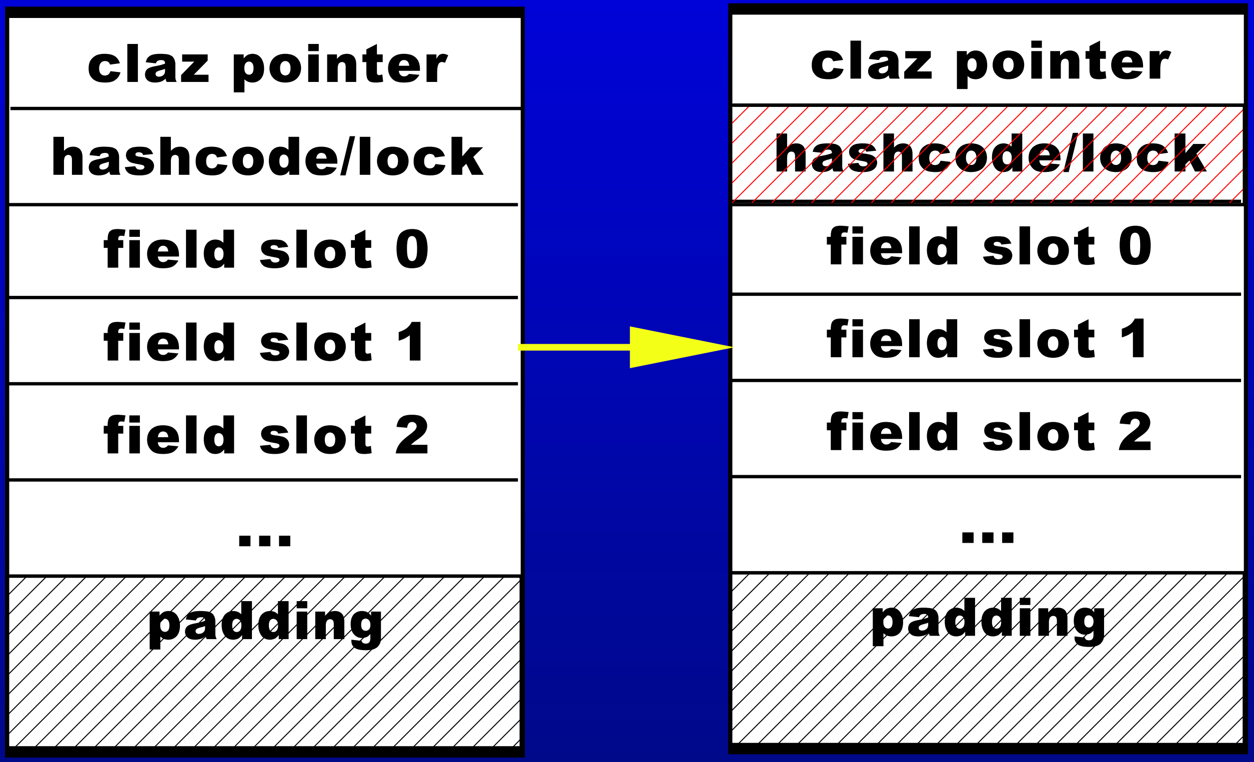
Three broad techniques:

- 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.

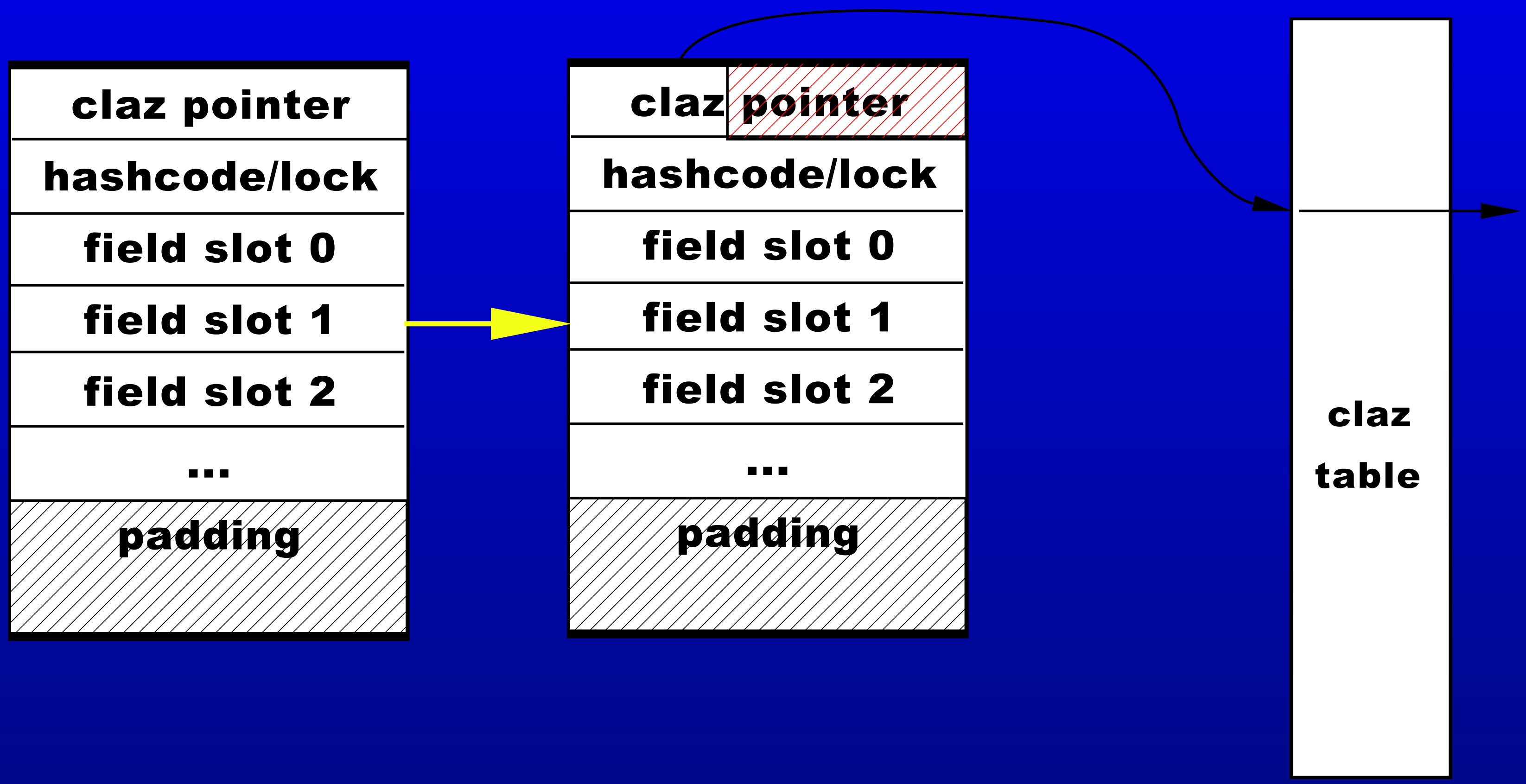
Header optimizations:

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

- 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.

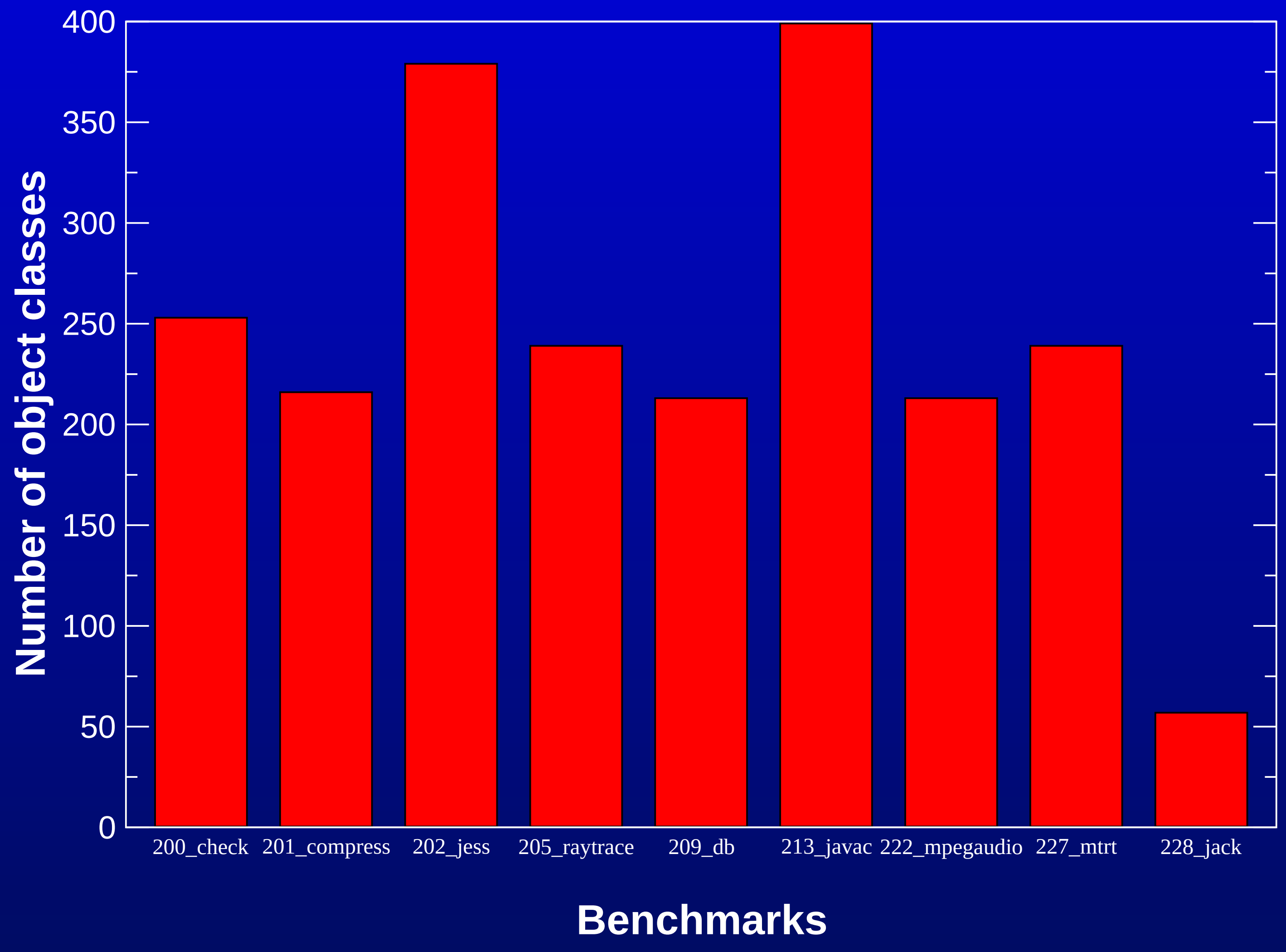
Header optimizations:

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.

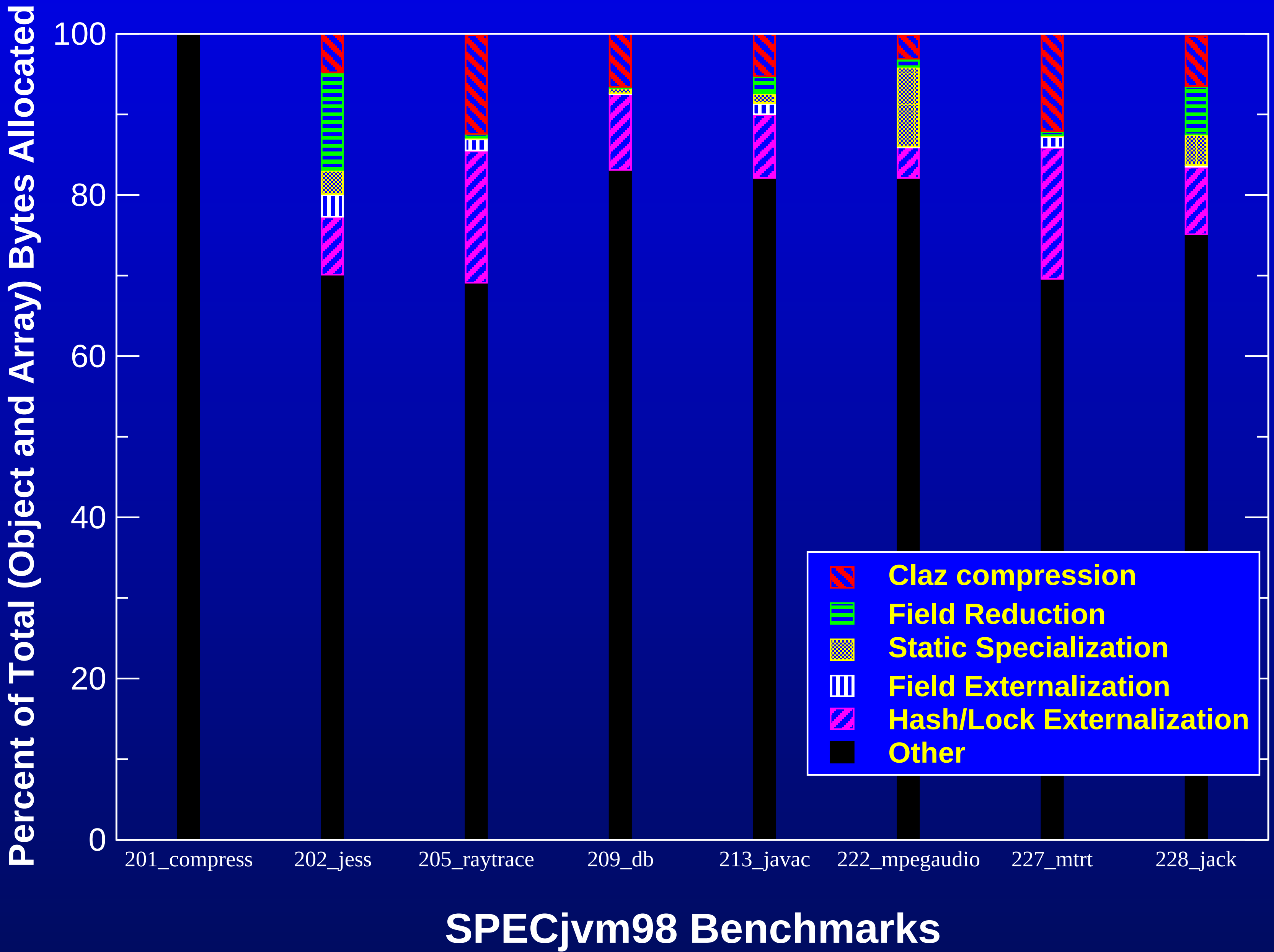
Class statistics

Class statistics for applications in SPECjvm98 benchmark suite:

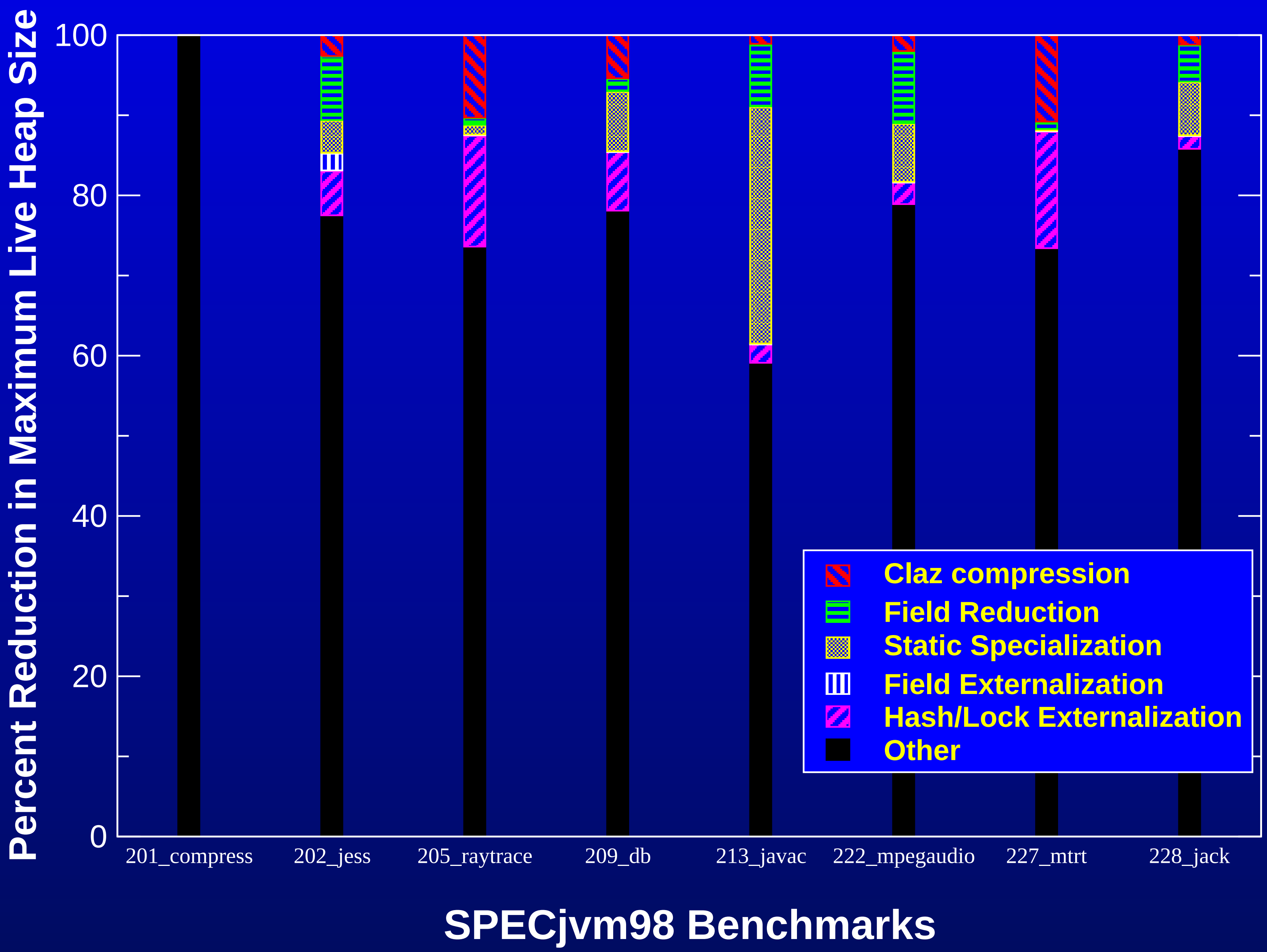


How well does it work?

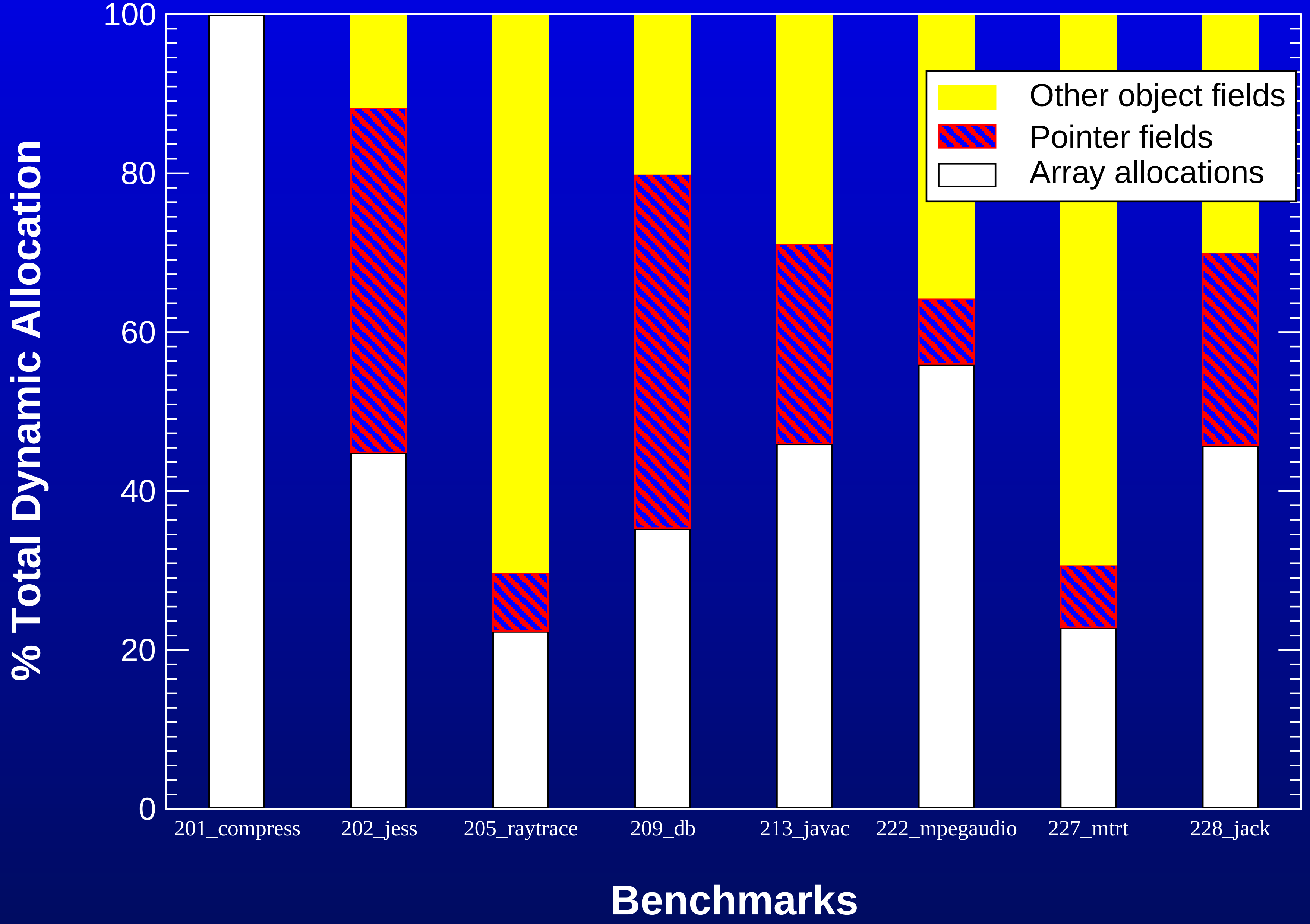
Reduction in total allocations



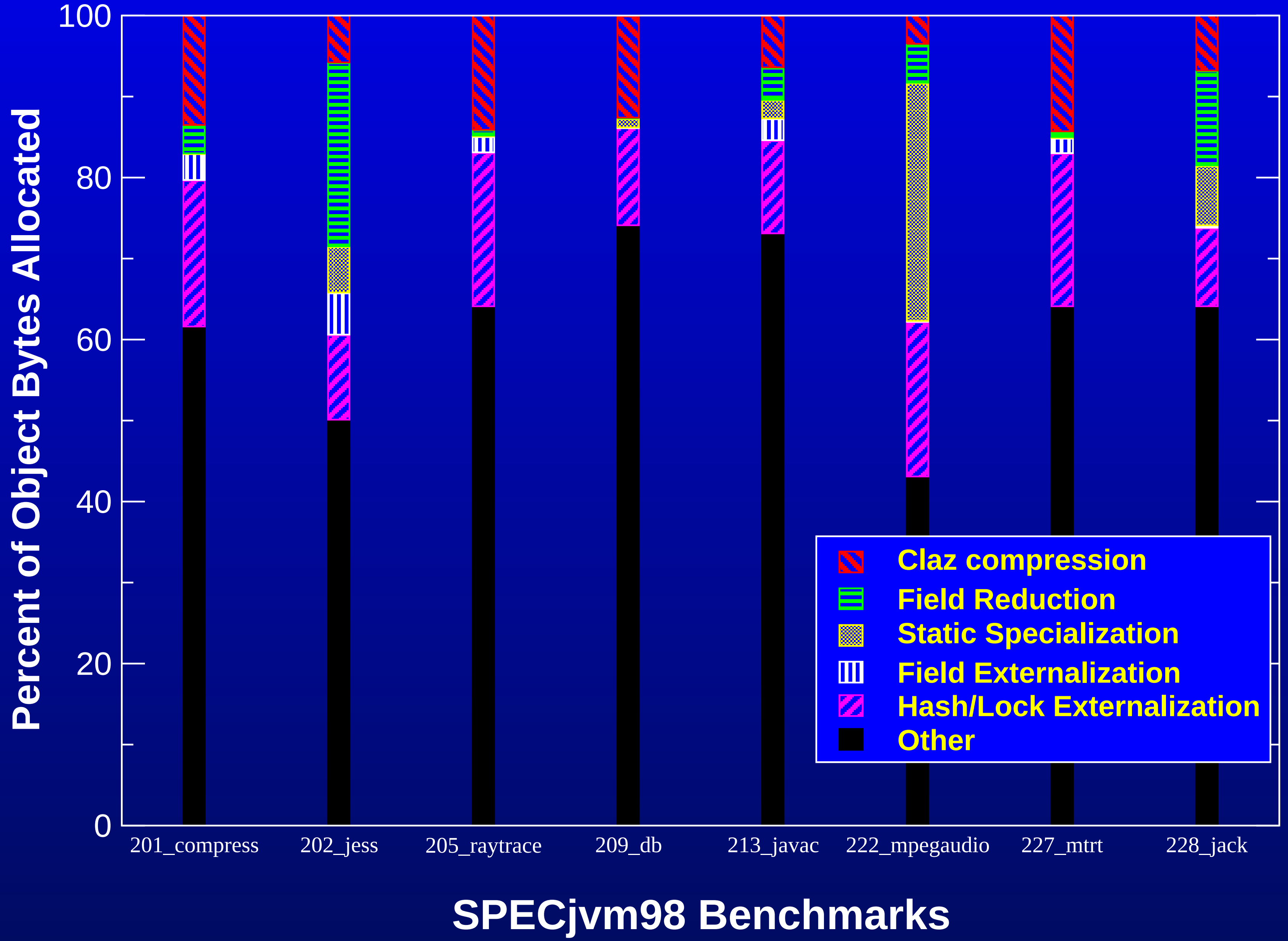
Reduction in total live data



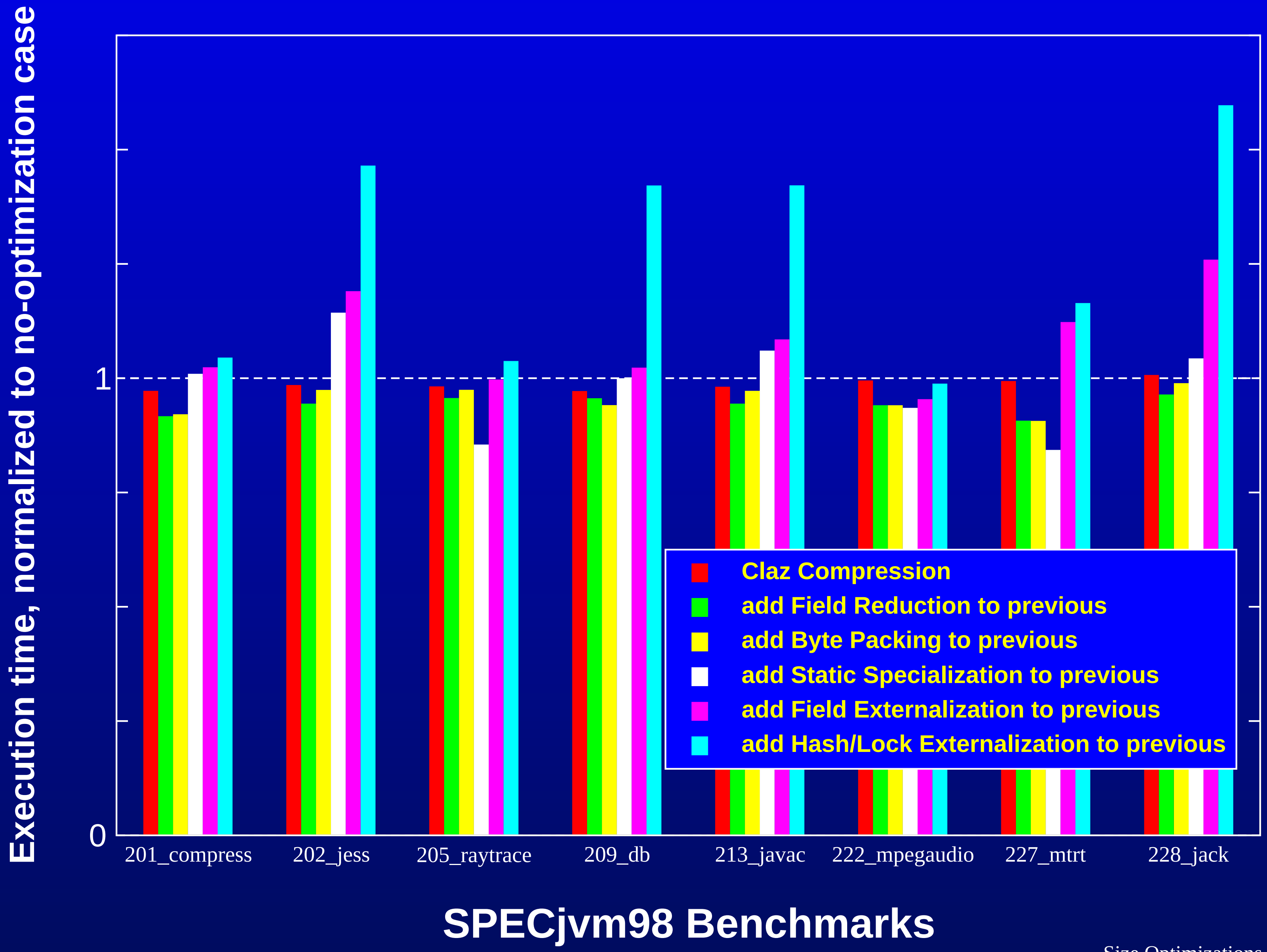
Available reduction opportunities



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 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

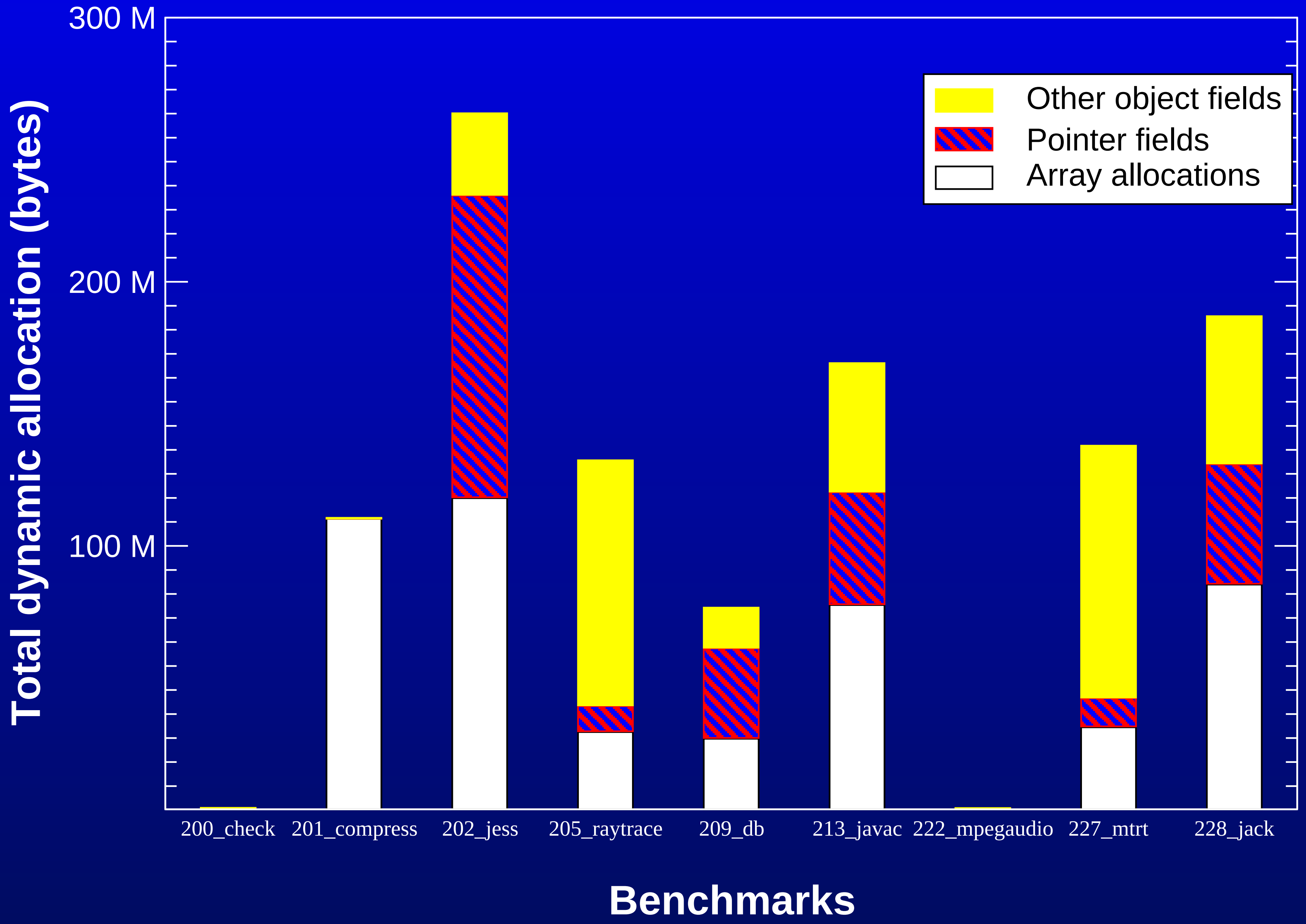
- We achieved substantial space savings on typical object-oriented applications.
  - In one case, over 40% reduction in total live data.
- Even more space reduction is possible!

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



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!