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

Notes

Nothing should be said on the title slide.

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

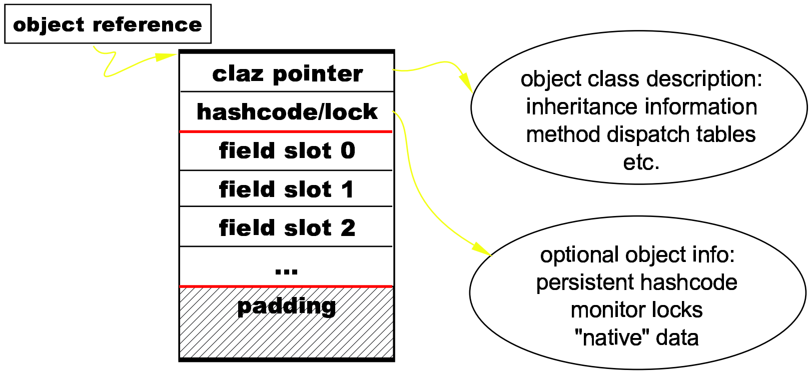
Applying transformations to reduce the memory consumption of the program.

Notes

This talk is about size optimizations for Java programs. Our goal is to reduce the amount of memory used by object-oriented programs (in this case, Java) by using static whole-program analyses to identify opportunities and applying transformations to effect the reduction.

Structure of a Java Object

- Typical of many O-O languages.



object reference

claz pointer

hashcode/lock

field slot 0

field slot 1

field slot 2

...

padding

object class description:
inheritance information
method dispatch tables
etc.

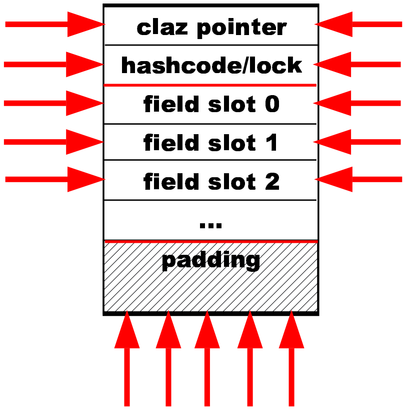
optional object info:
persistent hashcode
monitor locks
"native" data

Notes

Here’s our starting point. This is how most Java implementations lay out objects. I want you to notice that there are three kinds of space in this layout, helpfully delineated with red lines. The first section of the object usually consists of information required by the runtime implementation but not directly specified by the programmer. This includes a claz pointer, which points to an external structure of information about the object’s type, and some information to support the hashcode and locking semantics of Java. The second section of the object contains the fields declared by the programmer. If this were a car object, these fields might indicate the color and model of the car. The last section consists of padding which the runtime implementation will add to bring the various parts of the object to certain alignment boundaries.

Strategy

Push hard on all the bits.



claz pointer

hashcode/lock

field slot 0

field slot 1

field slot 2

...

padding

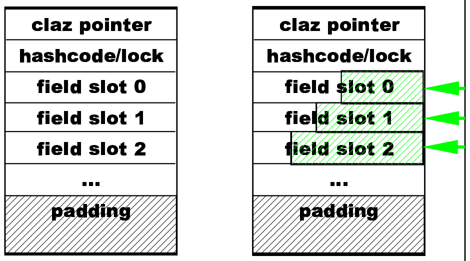
Notes

Our strategy is simple: @ we’re going to push hard on all the bits, in the object header, in the fields of the object, and even on the padding bytes, in order to reduce the size of each object and thus the total allocated and live memory of the program.

How to compress objects

Three broad techniques:

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



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

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

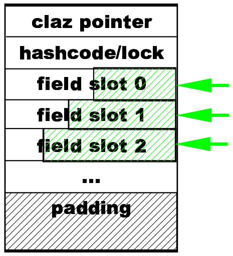
Notes

There are three broad techniques we are going to use to effect our size reductions: @ field compression, which reduces the size of the programmer-declared fields, @ mostly-constant field elimination, to completely eliminate memory used to store common values, and @ header optimizations, which leverage more-efficient representations for the data needed by the runtime. We will look at each of these in order, starting with. . .

How to compress objects

Three broad techniques:

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



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

Notes

. . . field compression.

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

Notes

Field compression targets the space directly allocated by the programmer. In the sample class at the bottom of the slide, we define an object representing cars. Its first field is a color, and it is declared an integer which allows the enumeration of up to 2^{32} different colors of cars.
@ The first analysis we do is a standard sparse conditional constant propagation pass over the whole program to identify unused, unread, or constant fields. @ Suppose we're building Ford Model-T's. Since they only come in black, this field will be constant and can be removed.
@ A novel contribution is the next step, bitwidth analysis, which discovers tighter upper bounds on field sizes. @ Actually, Model-T's were produced in several different colors before Ford started mass-production. Our bitwidth analysis could determine that we really only need two bits to store colors, since our program only ever stores three different colors in a Car object.
@ After we perform the analysis, we use the results to reduce the space used by the fields. We'll talk about this later.

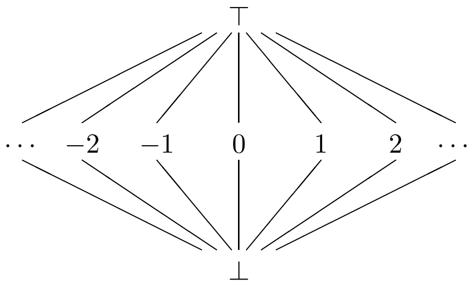
How are these analyses performed?

Notes

So how do we actually obtain the information we need to do field compression?
We'll look first at a simple constant propagation analysis, and then see how, by a series of extensions, we can obtain a powerful bitwidth analysis.

Intraprocedural Analysis

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



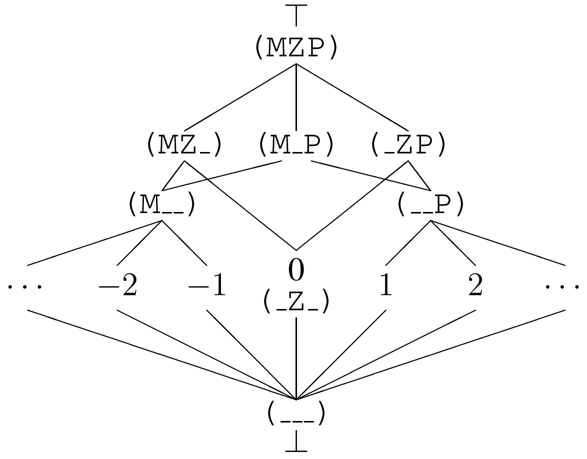
$$\boxed{\mathtt{i} = \bot\bot \sqcap 1111 \sqcap 21 \sqcap 2 = \top}$$

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

Notes

Let's look at a quick example of how the Sparse Predicated Typed Constant analysis is done. @ We have this simple program, which assigns values to an integer variable `i` and then tests the result. @ When we perform the dataflow analysis, we will abstract the value domain of the program using this lattice of integer constants. The $\bot$ value indicates that nothing is known about the value of a variable. @ At the start of the analysis, we know nothing about the value of `i`. @ When the analysis finds that the first assignment to `i` is executable, then @ it will perform a meet operation on the previous value ($\bot$) and the new value (1) to obtain @ 1, because 1 is the minimum element greater-than-or-equal-to both operands. The analysis now knows that the variable `i` can have the value 1. @ Later, it finds that the second assignment to `i` is executable, and @ it computes $1 \sqcap 2$, @ which yields the $\top$ element in the lattice. The $\top$ element usually means, "I give up, I can't constrain this value any more, it could be anything." This example illustrates the limitations of the simplified SPTC lattice shown here, because @ when we now look at the final if statement, the analysis can't tell that, one way or another, `i` will always be positive and thus this comparison will always be true. The $\top$ element means that *any* value is possible for `i`, which is a very conservative approximation.

A signed integer lattice



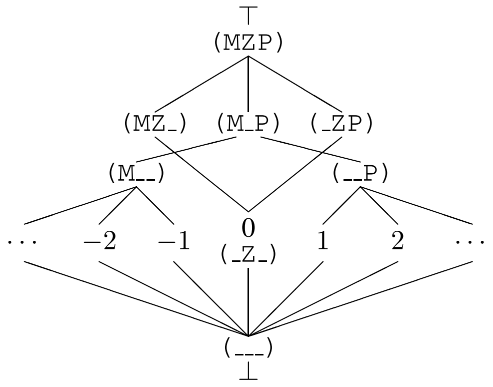
An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain.

Notes

So let's see how we'd extend the lattice to make the analysis stronger. Here we have a lattice that allows us to classify values as positive or negative, even if we don't know what the actual value will be. @ Here if we perform meet on 1 and 2 we get the element `(__P)`, which indicates "any positive number". If we then do a meet with a negative number, say, -2, @ we'll get `(M_P)`, or "a non-zero number". If we later discover an assignment of zero, @ we finally get the $\top$ element.

Example, redux

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



$$\boxed{\texttt{i} = \bot\bot \sqcap 1111 \sqcap 21 \sqcap 2 = \texttt{(\_\_P)}}$$

Notes

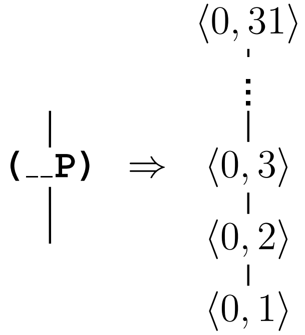
With the new lattice, we start at $\bot$ as before. @ Looking at `i=1`, @ we do a meet with 1 @ to get 1. But now, when we see `i=2`, @ and meet with 2, @ we get `(__P)`, or "a positive integer." @ This time, when we get to the comparison, we can tell that `i>0` will always be true.

Extending the lattice

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

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

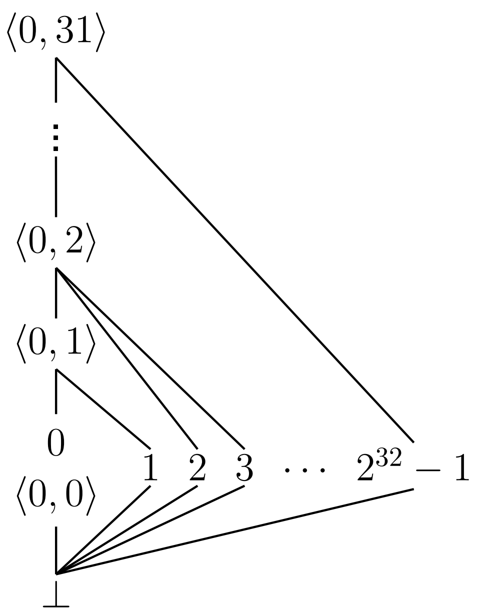
In lattice context:



Notes

To perform bitwidth analysis, we need only extend this signed integer value lattice a little further. We replace all the letters M and P in the previous lattice entries with positive integers m and p indicating the *bitwidths* of the negative and positive portions of the possible values. @ We now represent these lattice entries as tuples $\langle m, p \rangle$, and use the $\langle 0, 0 \rangle$ tuple to represent zero — what our previous lattice would have called `(_Z_)`. @ We can imagine expanding each node in our previous lattice with distinct tuples, with the ordering relations shown.

Bitwidth lattice detail

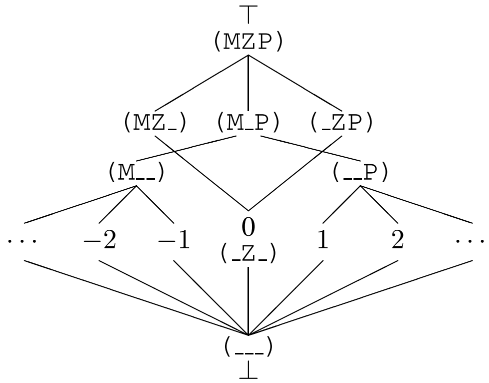


Notes

The picture is actually a little more complicated. Here we see a small piece of the new expanded lattice. We see that performing a meet of any two positive integers will result in a tuple which accurately reflects the minimum bitwidth needed to represent both numbers.

Example redux, redux

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



$$\boxed{\texttt{i} = \bot\bot \sqcap 1111 \sqcap 21 \sqcap 2 = \langle 0,2\rangle}$$

Notes

Revisiting our example: we still start with $\texttt{i} = \bot$. @ Looking at `i=1`, @ we do a meet with 1 @ to get 1. @ We look at `i=2`, @ and meet with 2, @ now we get the tuple $\langle 0,2\rangle$, indicating that `i` can not be negative and that we only need two bits to store the positive portion. @ We can still tell at the comparison point that `i>0` will always be true, but the real value of this analysis will be the space reductions we obtain when we apply it to fields.

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.

Notes

For completeness: here are some of the combination rules we use to perform abstract evaluation of unary and binary operations using our bitwidth value domain. I'm afraid time doesn't permit us to go through these individually at this point, but these are in the paper. Note for the first rule that we're tracking the bitwidth of the *absolute value* of the number, so the rule for negation is simply interchanging the positive and negative portions of the tuple.

If questioned: The first entry simply says that negation exchanges the positive and negative bitwidths. The second entry gives the rules for addition: we have to add one to the width to allow for carry out. The rule for multiplication should remind you that we're operating in the log-domain. The underlying numeric representation shows through in our rules for logical-AND: when anding two positive integers, the resulting bitwidth will match the smaller of the two inputs, since leading zeros will force zeros on the output. But if negative numbers are possible, we must use a far more conservative rule to account for the leading ones in the twos-complement representation of negative numbers. The m-component of the tuple roughly identifies the leftmost *zero*, so clearly the largest m will dictate where the leftmost zero can be in the result. The p component identifies the leftmost *one*, and since the all-ones value -1 is included in all negative ranges, the largest positive value input could emerge unchanged. We cannot create a larger positive value because the AND operation cannot create ones anywhere there is a zero in the input.

Interprocedural analysis

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

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

Notes

Our examples have all been intraprocedural. We use a *field-based* technique to perform the analysis interprocedurally, maintaining a single analysis value for each distinct object field. @ Instead of maintaining a value for `i`, we maintain a value for field `f`, @ and this works even when the various accesses take place in different methods.

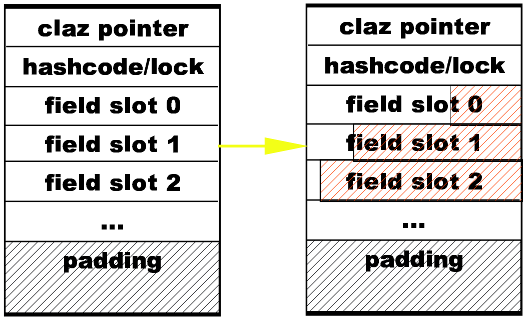
All cars are black

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

Notes

Returning to our car example, we can see how our initial Sparse Predicated Typed Constant analysis could determine that, if all cars' model is FORD, then all car's color will be BLACK. Having determined this, we can simply substitute BLACK for the color wherever it appears and remove the field. However, if there are non-FORD cars, we can still use our bitwidth analysis to determine a bound on how many bits we need to represent the various car colors that we see assigned to the color field.

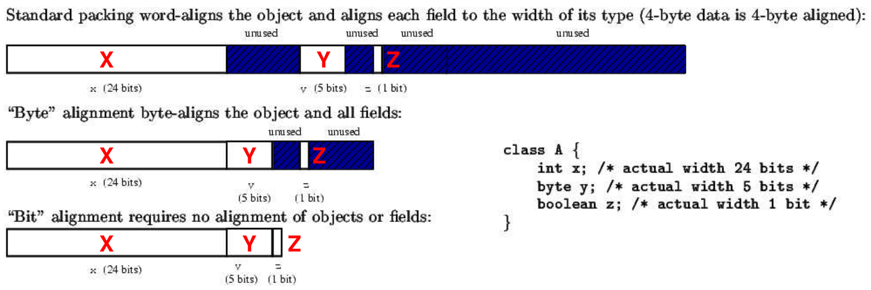
Field compression using bitwidths



Notes

Once we've used our analysis to determine accurate widths for fields, we would like to shrink the objects' allocations to use only the space actually needed for each field. So if we have less than 256 colors, we can use a single byte in the object structure to represent color.

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.

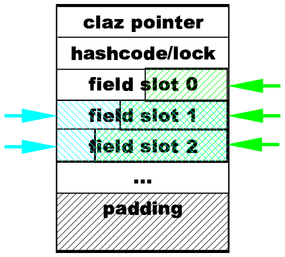
Notes

The actual situtation is a little more complicated. There is padding within and at the end of the object, required for various reasons. The heap allocator may prefer to have the chunks it returns aligned on certain boundaries, and the machine architecture may prefer to access, for example, word data aligned at word boundaries. At some runtime cost, we can overcome these limitations. At the top, is the standard “Java” object packing. All fields are aligned to their natural size, so that, word-sized fields will always begin at word boundaries. In this work we implemented a “byte” alignment strategy, where all fields are placed at the nearest *byte* boundaries, irrespective of their preferred alignment. One can also imagine a “bit” alignment strategy where each field uses *exactly* the number of bits it requires. At runtime we must then perform bit-masking and -extraction operations to access the fields. We found very little additional space-savings potential from going to bit alignment, which is why the numbers we will present use “byte” alignment.

How to compress objects

Three broad techniques:

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



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

Notes

Let’s return to our outline. So far we have talked about field compression; using bitwidth analysis to reduce the static size of fields allocated in objects, and using field packing to reduce the amount of object padding. Now we will talk about “Mostly-constant field elimination,” our second space-reduction technique.

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.

Notes

It’s easy to remove constant fields; we just replace the field reference with the appropriate constant. @ But our key idea here is that it is also possible to replace *mostly*-constant fields. @ We identify fields which have a certain value “most of the time” using @ static analysis and profiling. @ We can then transform the objects to remove fields with the common value, so that we only spend space on fields with unusual values. @ Our techniques for doing this are called static specialization and externalization. The types of “mostly-constant” fields that can be removed are slightly different with the two techniques; we will look at static specialization first.

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);
    }
}
```

Notes

We’re going to jump right in with an example. This is `java.lang.String` from the Java standard class library. It has three fields: a character array and an integer length and offset. This representation was chosen to allow you to perform the substring operation in constant time; you don’t have to copy the string, you just create a new string with a different offset and length and share the same character array.

The interesting thing about `java.lang.String` is that the `offset` field is almost *always* zero. Only strings created with the `substring()` method have non-zero offset fields. And the offset field is never changed after the object is created. These are the key properties needed to enable static specialization.

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.

Notes

In order to apply our technique, we need to find a “mostly-constant” field, which in our example is `String.offset`, with the value “mostly zero”, and, importantly, the value of that field cannot 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).

Problems:

- The code could directly access the to-be-removed field.
- Allocation sites directly instantiate the old class.

Notes

So, how are we going to save memory in `java.lang.String`? We're going to split the class in two: there will be a `SmallString` class without the offset field, and a `BigString` class that does have it.
@ A couple of problems that come up:

- @ What happens to places in `SmallString` where the removed offset field is referenced?
- @ And when we see an allocation of `String`, do we change it to `SmallString` or `BigString`?

Specialization example:

java.lang.String

```
public final class StringSmallString {
    private final char value[];
~~    private final int offset;~~
    private final int count;
    protected int getOffset() { return 0; }
    ...
    public char charAt(int i) {
        return value[offsetoffsetgetOffset()+1];
    }
    public String substring(int start) {
        int noff = offsetoffsetgetOffset() + start;
        int ncnt = count - start;
        return new String(value, noff, ncnt);
    }
}
```

Notes

Let's go back and see how this works. We take the String class, rename it @ SmallString and remove the offset field. But what do we do with the references to the now-deleted field? @ We solve this problem by *virtualizing* the field: adding a getOffset() accessor method (which always returns zero) and replacing references to offset with calls to getOffset().

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; }
}
```

Notes

Now we create BigString as an almost-trivial subclass of SmallString. It shared all of the same functionality, except BigString *has* an offset field, and implements getOffset() in the way you'd expect. So any object which needs a non-zero offset can instatiate BigString and things will work as expected.

From our key properties, remember that there are no assignments *to* the offset field except in the constructor. But what do we do there? And how do we tell if we're supposed to replace a call to the String constructor with a SmallString or a BigString?

It turns out there are three cases to worry about.

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;
}
```

Notes

In the first case, the field is constant in the constructor. Every instantiation which uses this constructor will set `offset` to zero. (And as we mentioned before, it is never reset after construction.) So we juse @ remove the assignment to `offset` and replace the instantiation of `String` with that of `SmallString`.

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;
}
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);
```

Notes

But what if we have code like this, where we don't know what value `x` will have when the `String` is created? This is essentially the case for the `substring` method of `String`. In this case, the value of the offset field is a simple function of some parameter given to the constructor. Here's our solution: @ we add a test around the allocation site, so that we can construct a Small or Big string depending on the value `x` actually has at runtime.

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;
}
```

Notes

But what about code such as this? Here we know *nothing* about the value of offset, as it is derived programmatically *within the constructor*. There’s no easy test we can do outside the constructor. @ Here we must simply give up and always allocate an instance of `BigString`. Code such as this is not actually found in `java.lang.String`, but this solution allows us to take advantage of frequently-constant fields in many cases with a fairly neutral (no cost, no gain) fallback in the worst case.

Static specialization

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

Notes

Let’s review the static specialization transformation: we split the target class into two parts, one of which has the field and one which doesn’t, and use virtual accessor functions to hide this split from users of the class. We can do this recursively on multiple fields; we use profiling to determine the best splitting order in that case. We have two constraints: @ the fields must be “mostly-constant”, and then cannot be modified after the completion of the constructor.

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.~~

Notes

But what if the first of these conditions hold, @ but the second doesn't?

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.

Notes

Sometimes fields are *run-time* constants but not *compile-time* constants. @ Some examples are sparse matrices, an example from the SPEC benchmark suite, and short linked lists. In graphics applications, the background color will typically be another example. What are we going to do in this case?

@ Our solution is to exploit the relationship between fields and maps to reduce memory overhead when we're storing the common 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.
- 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.

Notes

Read this slide.

. . . This means we don’t actually have to store this value in the map; we can just remove the entry instead.

Externalization example:

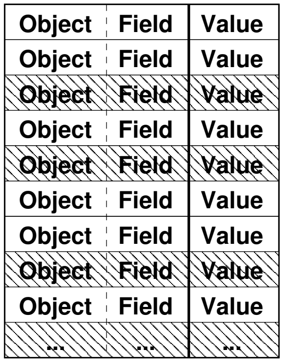
java.lang.String

```
public final class String {
    private final char value[];
    ~~private final int offset;~~
    private final int count;
    public char charAt(int i) {
        return value[offsetoffsetgetOffset()+1];
    }
    public String substring(int start) {
        int noff = offsetoffsetgetOffset() + 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();
    }
}
```

Notes

Let’s see how it would work if we took our static specialization example and externalized the field, instead. @ Again, the field is deleted from the class. But now, the @ `getOffset()` method references an external map.

External map implementation



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
...	...	...

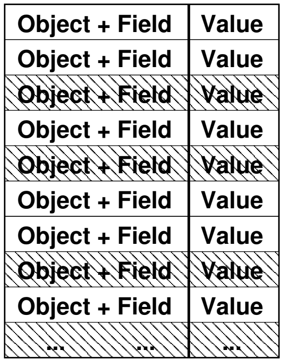
- “open addressed” for low overhead.
- load-factor of 2/3
- two-word key and one-word values means break-even point is 82%

  (i.e. field may not differ from the “mostly-constant” value in more than 18% of objects.)

Notes

The efficiency of this external map crucially dictates how much space savings we are able to achieve with this scheme. So let’s look at implementation for a moment. We choose an “open-addressed” hash table, to keep our overhead low. The alternative is some sort of linked-bucket structure, and the links between buckets add crucially to our overhead. @ Every hash table needs to have some entries empty in order to have good performance; we assume a load-factor of 2/3, which means that 1/3 of the hashtable slots will be empty. @ We can do the math, and find that a two-word key and one-word values mean our break-even point is 82%; @ that means that no more than 18% of the fields can *differ* from the “mostly-constant” value in order to attain any space savings at all.

We can do better!



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%.

Notes

We can do better than this, though: since all we really need is a unique identifier for the object/field pair, we can simply @ offset the object pointer to obtain a one-word key. @ Note that this is like using a pointer to the *field* instead of a field ID and a pointer to the *object*; the field we would like to point to is not actually present in the object, though. This means that @ the scheme imposes a limit of the number of fields which may be externalized, based on the number of non-externalized fields, including header fields, remaining in the object. @ This limitation is not overly burdensome, however, and we lower our break-even point to 66%.

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.

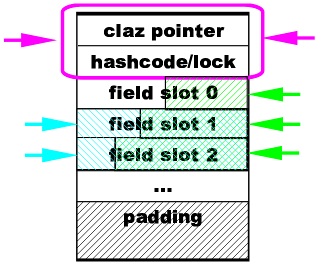
Notes

A few more implementation details: First, we use value profiling to identify where externalization will be worthwhile. Remember, we need at least 2/3 of the values to be some constant. @ When we did the profiling, we looked at integer constants from -5 to 5 and at `null` for pointer types. @ We could that 0 and 1 were by far the most common “mostly-constant” values; in a production implementation you could look only at these two and get most of our savings.

How to compress objects

Three broad techniques:

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

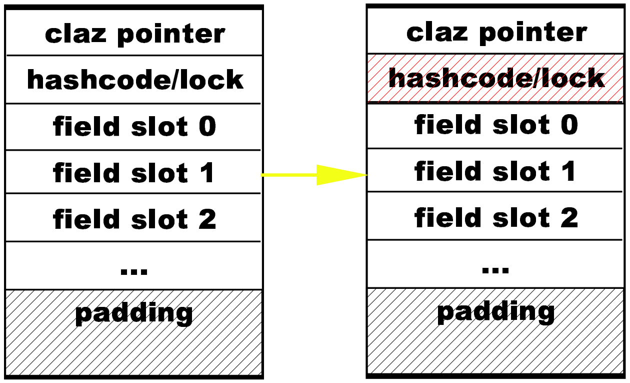


Notes

Returning to our outline. We talked about using bitwidth and constant analysis to compress fields, and at two different techniques for eliminating the space required to store “mostly-constant” values. Now we’ll quickly look at header optimizations, which reduce the amount of overhead needed by the runtime implementation.

Header optimizations:

Hashcode/Lock compression



Notes

Of the two words in a typical object header, let's look at the hashcode/lock field first. We were able to completely eliminate this field in a large proportion of our objects.

Header optimizations:

Hashcode/Lock compression

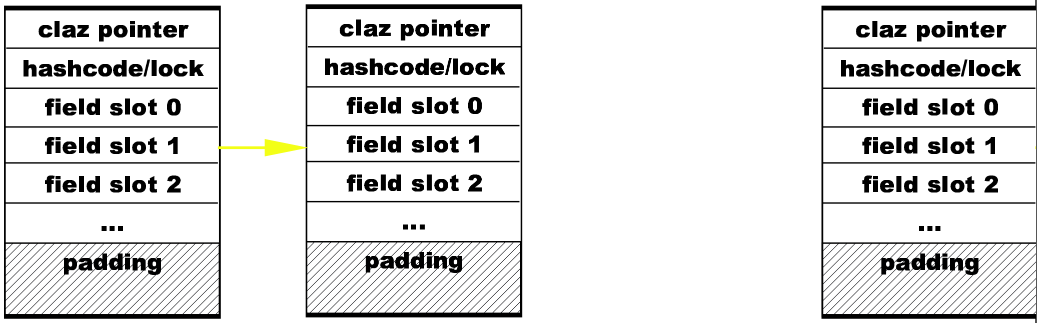
- Implemented as a special case of field externalization.
- The hashcode/lock field often unused because:
  - Most objects do not use their built-in hashcode.
  - Most objects are not synchronization targets.
- Could also use a static pointer analysis.

Notes

We implemented this as a special case of field externalization. @ Although we can't always tell at compile-time, in fact this field is very infrequently used. Most object types either are never used as keys in a hashtable, or implement their own hashing method not based on the identity-based hash specified by the Java semantics. Similarly, most programs are either not multi-threaded, or when they are, perform synchronization on only a small handful of their object types. Using externalization means that we allocate space as-needed in an external map; @ we could also use a static pointer analysis and techniques similar to static specialization to remove these fields from types where they are statically unused.

Header optimizations:

claz compression



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

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

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

Notes

And now looking at the second word: the claz pointer typically points directly to some structure with information about the object type. @ We can save space by imposing a level of indirection and using a small *table index* instead.

Header optimizations:

claz compression

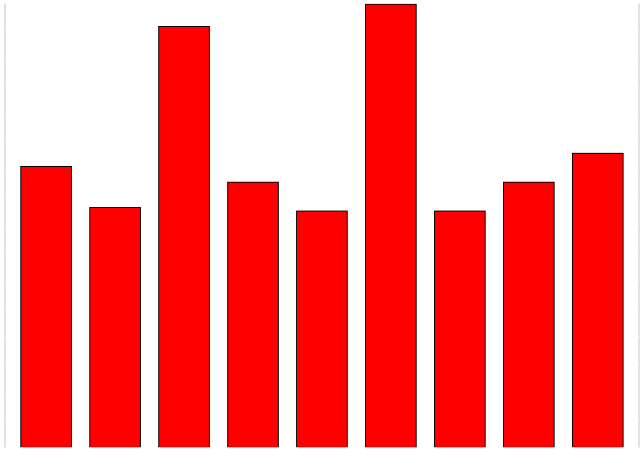
- replace `claz` pointer with a (smaller) table index.
- With co-operation of GC, works in dynamic environments.
- Many applications use less than 256 object types.

Notes

Our static analysis can tell how many *instantiated* types are in the program, and thus bound the index size. @ But this can even work in a dynamic environment, with a little co-operation from the GC. @ We'd like to reduce the claz pointer to a single byte. Is this possible in practice? How many classes are in typical applications?

Class statistics

Class statistics for applications in SPECjvm98 benchmark suite:



Notes

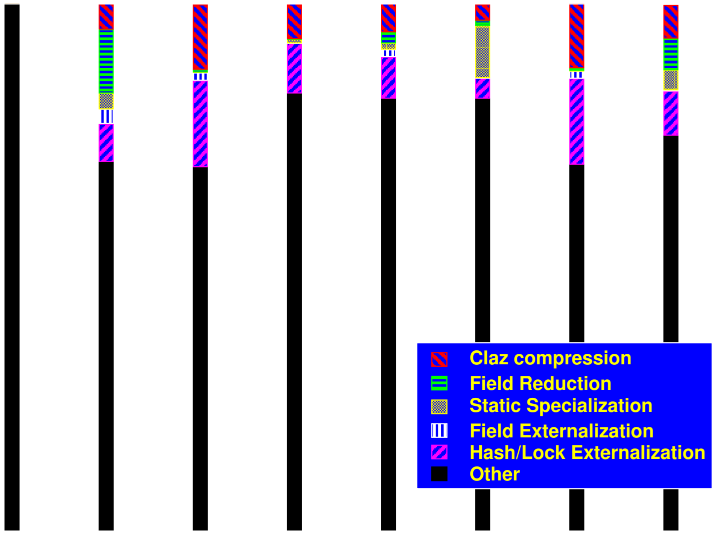
Here are the class statistics, etc. Note that all but three are below 256 classes, or *1 byte* of claz index. I have never seen an application that required more than 2 bytes of claz index.

How well does it work?

Notes

So how well does all this work?

Reduction in total allocations

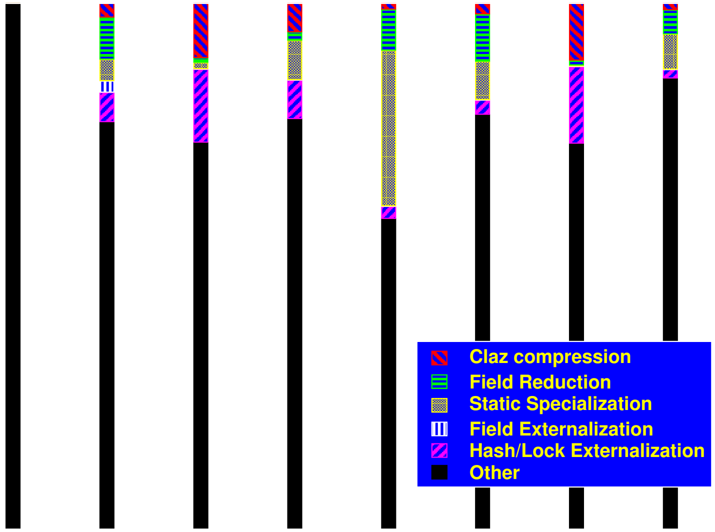


Notes

This is the reduction in *total number of bytes allocated* during the run of the programs. The programs at the bottom are from the SPECjvm98 benchmark suite, and represent real, complete, Java applications. Let me quickly step through them: here on the left is `compress`, a gzip-compression program. `jess` in an expert system, `raytrace` is a graphics rendering program — `mtrt` is actually the same program, but run in multi-threaded mode; the numbers are slightly different because (in theory) we can't do as much hash/lock externalization, because the locking capability of Java is being used. In practice, you can see it doesn't have much effect. `db` is a simple database, `javac` is the Sun java compiler, `mpegaudio` is an MP3 decoder, and `jack` is a parser generator.

But if you're paying attention, you'll realize that these aren't actually the *really* interesting numbers. A reduction in total allocated bytes is great, it reduces the amount of allocation and collection work the garbage collector has to do, but what you'd really like to see is a reduction in the maximum *live data* in the program. This would mean that the program would run in a smaller memory environment.

Reduction in total live data

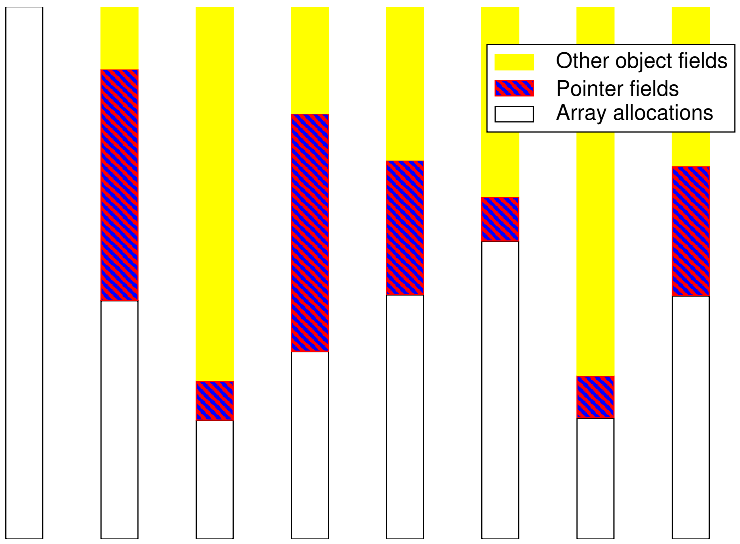


Notes

And that is what we see here. You'll notice the numbers look roughly the same, but they improve a lot for javac; what this shows you is that the objects we target in javac are precisely the ones which are live the longest and stress the system the most.

OK. You no doubt have noticed by now that compress is getting no help at all by anything we are doing. Why is that?

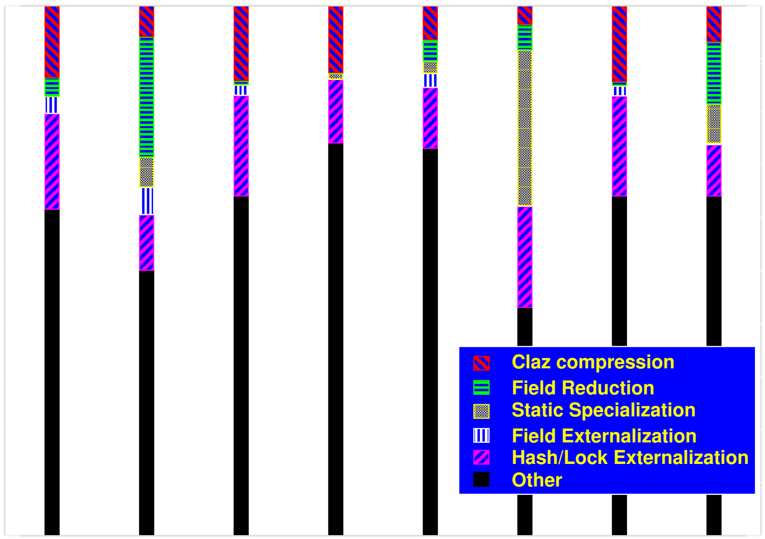
Available reduction opportunities



Notes

This graph shows the percent of total allocated bytes which consist of arrays (as opposed to objects), and pointer fields in objects. Our transformations are all object-oriented, and none of them currently target arrays. In addition, the field compression techniques I've been describing are fundamentally integer-based; apart from sometimes being able to remove mostly-null fields, they are not terribly effective on fields of pointer type. So the yellow bars show you how much of the program is *left* for our techniques to optimize, after you take away the pointers and the arrays. Compress is *all* array allocations; hence our poor performance should make sense now. The thing to notice here is that we do a fantastic job with the portion of the program allocation that we're targetting.
Indeed, if you want us to look good, you might consider how we do solely on the *object* allocations in the program; discounting all array allocations.

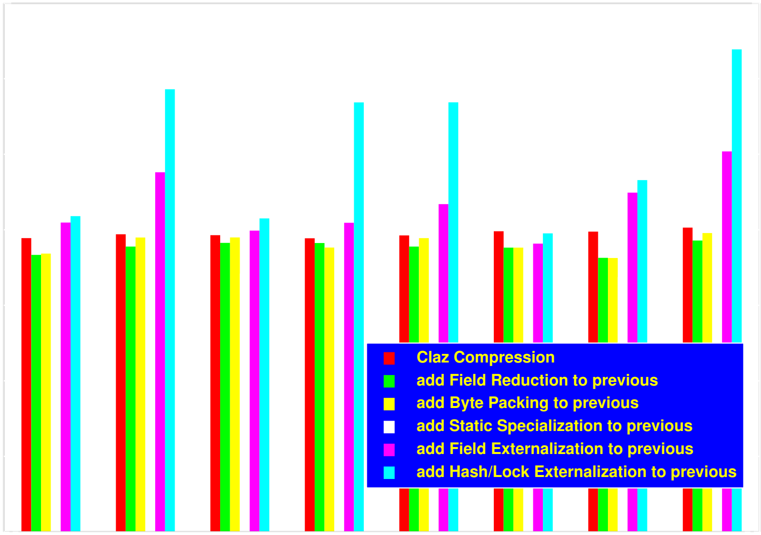
Reduction in object allocations



Notes

And that's what we show here. There are very sizable reductions now. But they have to be taken with a grain of salt: we know already that compress has almost no object allocations, so even though we do well on those that are present, it doesn't "really" matter.

Moderate performance impact



Notes

We've talked about some transformations that impact performance; this graph shows you what that performance impact really is. One represents the speed of the program before we touch it. You'll notice that the first three transformations, which are claz/field compression and byte-packing, offer a moderate speed *improvement* over the original code. This is wholly due to better performance in the garbage collector and cache as we shrink the objects. The potential performance penalty for the claz indirection and the unaligned memory accesses are overwhelmed by the GC and caching improvements.

Static specialization, the white bar, gives back some of that performance gain. This is due to the fact that we've virtualized field accesses, so where you previously had a simple memory access, you know have a method call and the overhead that goes with that. There are various techniques we could use to mitigate this.

Field externalization costs a little more, because it's a more expensive form of field virtualization. In mtrt and jack it seems we are probably externalizing too many fields, and we're getting significant performance penalties. In jack, if you look at our live data numbers, you'll see that the space gain this is giving us is minimal; so our heuristics for choosing fields definitely need further tweaking.

In four cases, the final step, hash/lock externalization, costs about 30% of our performance. These are programs which lock extensively, although none of them strictly need to do *any* locking. Static techniques for lock-removal will mitigate this cost greatly.

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*.

Notes

So how can we go even further with this? @ First off, we currently do nothing with allocated arrays. Further, we make decisions for all instances of a class, so if a class is used in multiple very different ways, we are forced to pick one strategy for all uses. @ The solution is to use pointer analysis to help discriminate among objects and arrays, which will allow us to apply the techniques we've been describing. @ On a related note, we don't attempt to compress fields with pointer values at all. @ We have a region-based approach to limiting pointer sizes which we feel is worth exploring in this regard, and @ Zhang and Gupta have described an orthogonal technique which it may be useful to incorporate. @ Our "mostly-constant" transformations are dependent on profiling. @ We could either find good heuristics to remove the profiling requirement, @ or embrace run-time profiling, which would allow us to avoid transforming "hot" fields. @ Finally, we've used maps to implement fields via externalization; @ it's worth exploring whether fields can more space-efficiently implement some *maps* in the program.

Related Work

- Reducing lock overhead.
  - Bacon, Sweeney (OOPSLA ’96)
  - Onodera, Kawachiya (OOPSLA ’99)
  - Agesen, Detlefs, Garthwaite, Knippel, Ramakrishna, White (OOPSLA ’99)
- Escape analysis.
  - Aldrich, Chambers, Sirer, Eggers (SAS ’99)
  - Bogda, Hözle (OOPSLA ’99)
  - Whaley, Rinard (OOPSLA ’99)
  - Choi, Gupta, Serrano, Sreedhar, Midkiff (OOPSLA ’99)
  - Ruf (PLDI ’00)
  - Sălcianu, Rinard (PPoPP ’01)

Notes

The has been related work on reducing lock overhead, and in escape analyses to statically remove locking operations.

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)

Notes

We’ve also seen some surveys to attempt to quantify the space and time usage of Java programs, and devise efficient runtime representations.
Bitwidth analyses have been investigated, starting with my Master’s thesis in 99. An early focus was reducing the size of generated circuits in hardware compilers; the Europar paper also investigates applying the technique to MMX-like SIMD instruction sets.
Sweeney and Tip investigated a dead member analysis in C++ which is a subset of the constant field elimination technique I discussed at the beginning of this talk.

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.

Notes

Read this slide.

It's worth mentioning the "memory wall" in this regard. We've already seen a decent improvement in performance in many cases for some of these transformations, caused simply by reducing memory costs (including GC) and improving caching. This is despite adding extra instructions in the form of indirections, unaligned accesses, and virtualization. One can expect that the memory wall will continue to get steeper and the benefits of increasing the effective cache size will get greater, even as the ALU cost of unpacking operations becomes releatively cheaper. We believe this work has shown that space optimizations can be remarkably effective.

Size Optimizations for Java Programs

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

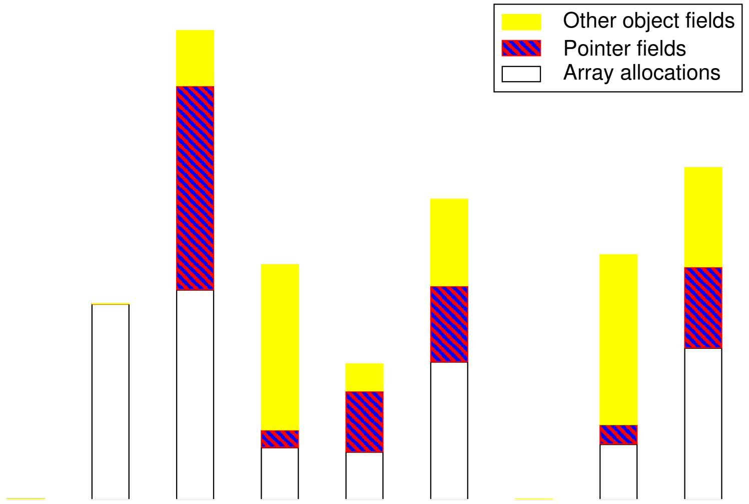
This talk:
http://flex-compiler.lcs.mit.edu/Harpoon/papers.html

Notes

The Graveyard Of Unused Slides follows this point.

Notes

Available reduction opportunities



Notes

This is the “total bytes” version of the slide.

Bitwidth analysis

Motivation:

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

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

- The compiler can do it for us!

Notes

Why specify widths manually when the compiler can do it?