Int*ra*procedural ("global") optimizations

Enlarge scope of analysis to whole procedure

- more opportunities for optimization
- have to deal with branches, merges, and loops

Can do constant propagation,
common subexpression elimination, etc.
at global level

Can do new things, e.g. **loop optimizations**

Optimizing compilers usually work at this level

Code motion

Goal: move **loop-invariant** calculations out of loops

Can do at source level or at intermediate code level

Source:

```
for (i = 0; i < 10; i = i+1) {
   a[i] = a[i] + b[j];
   z = z + 10000;
}
```

Transformed source:

```
t1 = b[j];
t2 = 10000;
for (i = 0; i < 10; i = i+1) {
   a[i] = a[i] + t1;
   z = z + t2;
}
```


Code motion at intermediate code level

Source:

```
for (i = 0; i < 10; i = i+1) {
   a[i] = b[j];
}
```

Unoptimized intermediate code:

```
   *(fp + ioffset) = 0;
label top;
   t0 = *(fp + ioffset);
   iffalse (t0 < 10) goto done;
   t1 = *(fp + joffset);
   t2 = t1 * 4;
   t3 = fp + t2;
   t4 = *(t3 + boffset);
   t5 = *(fp + ioffset);
   t6 = t5 * 4;
   t7 = fp + t6;
   *(t7 + aoffset) = t4;
   t9 = *(fp + ioffset);
   t10 = t9 + 1;
   *(fp + ioffset) = t10;
   goto top;
label done;
```


Loop induction variable elimination

For-loop index is **induction variable**

- incremented each time around loop
- offsets & pointers calculated from it

If used only to index arrays, can rewrite with pointers

- compute initial offsets/pointers before loop
- increment offsets/pointers each time around loop
- no expensive scaling in loop

Source:

```
for (i = 0; i < 10; i = i+1) {
   a[i] = a[i] + x;
}
```

Transformed source:

```
for (p = &a[0]; p < &a[10]; p = p+4) {
   *p = *p + x;
}
```

- then do loop-invariant code motion

Global register allocation

Try to allocate local variables to registers

If lifetimes of two locals don't overlap, can give to same register
Try to allocate most-frequently-used variables to registers first

Example:

```
int foo(int n, int x) {
   int sum;
   int i;
   int t;
   sum = x;
   for (i = n; i > 0; i=i-1) {
      sum = sum + i;
   }
   t = sum * sum;
   return t;
}
```


Int*er*procedural optimizations

Expand scope of analysis to procedures calling each other

Can do local & intraprocedural optimizations at larger scope

Can do new optimizations, e.g. **inlining**

Inlining

Replace procedure call with body of called procedure

Source:

```
final double pi = 3.1415927;
double circle_area(double radius) {
   return pi * (radius * radius);
}
...
double r = 5.0;
...
double a = circle_area(r);
```

After inlining:

```
...
double r = 5.0;
...
double a = pi * r * r;
```

(Then what?)

Summary

Enlarging scope of analysis yields better results

- today, most optimizing compilers work at the intraprocedural (aka global) level

Optimizations organized as collections of passes, each rewriting IL in place into better version

Presence of optimizations makes other parts of compiler (e.g. intermediate and target code generation) easier to write