Subroutine In-line Expansion

- Subroutines may be expanded in-line at the point of the call
  - Rather than use a stack-based calling convention
- In-line expansion saves subroutine overheads and help code improvement
- In-line expansion may be decided by the compiler based on some optimization heuristics
  - E.g. short, non-recursive subroutines are always in-lined in some languages

Subroutine In-line Expansion

- In-line expansion can also be suggested by the programmer
  - E.g. C++
    ```
    inline int max (int a, int b) {
      return a > b ? a : b;
    }
    ```
  - E.g. Ada
    ```
    function max(a, b : integer) return integer is
      begin
        if a > b then return a; else return b; end if;
      end max;
      pragma inline (max);
    ```
Macros and In-line Expansion

- What is the difference between a macro and a programmer suggested expansion?
  - Optional in the second case
  - Most importantly, in-line expansion is an implementation technique with no effect in program semantics

- E.g.

  ```
  #define MAX(a,b) ((a) > (b) ? (a) : (b))
  ```

  - No type checking
  - What happens after `MAX(x++, y++)`?
  - The larger argument is incremented twice

In-line Expansion

- In-line expansion has some disadvantages
  - Increase in code size
  - It cannot be used with recursive subroutines

- It is sometimes useful to expand the first case in a recursion subroutine
  - Optimize the common case rule

  ```
  range_t bucket_contents (bucket *b, domain_t x) {
    if (b->key == x)
      return b->val;
    else if (b->next == 0)
      return ERROR;
    else
      return bucket_contents (b->next, x);
  }
  ```

- Most hash chains are only one bucket long
**Example: Control Flow Graph**

- **Basic blocks** are maximal-length set of sequential operations
  - Operations on a set of *virtual registers*
    - Unlimited
    - A new one for each computed value
  - Arcs represent interblock control flow

---

**Phases**

- Character stream
- Token stream
- Parse tree
- Abstract syntax tree with annotations (high-level IP)
- Control flow graph with pseudo-instructions in basic blocks (medium-level IP)
- Modified control flow graph
- Modified control flow graph
- (Almost) assembly language
  - Target code generation
  - Preliminary instruction scheduling
  - Register allocation
  - Final instruction scheduling
  - Pephole optimization

---

**Example**

```
\text{Start} \quad \text{a1 = \text{output}}
\text{call} \text{c1}
\text{i = iv}
\text{a1 = \text{ip1}}
\text{call} \text{c1}
\text{j = iv}
\text{v1 = i}
\text{v2 = j}
\text{v3 = v1 > v2}
\text{toz v3}
\text{v4 = i}
\text{v5 = j}
\text{v6 = v4 > v6}
\text{toz v6}
\text{v7 = i}
\text{v8 = j}
\text{v9 = v7 > v8}
\text{toz v9}
\text{v10 = j}
\text{v11 = 1}
\text{v12 = v10 - v11}
\text{j = v12}
\text{\text{End}}
```

**Lecture 35: In-line Expansion and Local Optimization**

April 19, 2002
Redundancy Elimination in Basic Blocks

- We will consider the example on the right.
- It computes the binomial coefficients
  \[ \binom{n}{m} \]
  for \(0 \leq m \leq n\).
- It is based on
  \[ \binom{n}{m} = \binom{n}{n-m} \]

```c
combinations (int n, int *A) {
    int t; 
    A[0] = 1; 
    A[n] = 1; 
    t = 1; 
    for (i = 1; i <= n/2; i++) { 
        t = (t * (n+1-i)) / i; 
        A[i] = t; 
        A[n-i] = t; 
    }
}
```

Syntax Tree for the `combinations` subroutine.
Naïve Control Flow Graph
for the combinations subroutine

Block 1:
sp := sp - 2
v1 := r4 − n
v2 := r5 − A
A := v2
v3 := A
v4 := 1
v5 := A
v6 := n
v7 := 4
v8 := v6 + v7
v9 := v5 + v6
v10 := 1
v9 := v10
v11 := 1
l := v11
v12 := 1
l := v12
goto Block 4

Block 2:
v13 := 1
v14 := n
v15 := 1
v16 := v14 + v15
v17 := 1
v18 := v16 − v17
v19 := v13 × v15
v20 := 1
v21 := v19 div v20
l := v21
v22 := A
v23 := 4
v24 := v20 × v24
v25 := v22 + v25
v27 := 1
v26 := v27
v28 := A
v29 := n
v30 := v31 := v29 − v30
v32 := 4
v33 := v31 × v32
v34 := v28 + v33
v35 := 1
v34 := v35
v36 := 1
v37 := 1
v36 := v36 + v37
l := v36
geto Block 3

Block 3:
v39 := l
v40 := n
v41 := 2
v42 := v40 div v41
v43 := v39 × v42
if v43 goto Block 2
else goto Block 4

Naïve Control Flow Graph

- Uses virtual registers
  - A new register for each new value
- ra is the return address, fp is the frame pointer
- n, A, I and t perform the appropriate displacement addressing with respect to the stack pointer (sp) register
- Parameter passing using r4 and r5
Value Numbering

• How can we eliminate redundant loads and computations?
  – Expression DAG
  – Value numbering

• In value numbering, the compilers assigns the same name (i.e., number) to any two or symbolically equivalent computations (i.e., values)

• A dictionary is used to keep track of values that have already been loaded or computed

Value Numbering

• If a value is already in a register, reuse that register
  – E.g., the load instruction can be eliminated \( vi := x \) if the value \( x \) is already in register \( vj \)
    » Replace all uses of \( vi \) by \( vj \)

• Similarly, we can get rid of small constants using immediate value
Value Numbering

• In \( v_i := v_j \text{ op } v_k \), we can use constant folding if the values in \( v_j \) and \( v_k \) are known to be constants
  – Local constant folding and constant propagation
  – At the same time, strength reduction and useless abstraction elimination

• A key that combine the registers and the operator is used to keep track of the previous operation in the dictionary

Control Flow Graph for combinations after local redundancy elimination and strength reduction

Block 1:
- \( sp := sp - 8 \)
- \( v1 := r4 \)
- \( n := v1 \)
- \( v2 := r5 \)
- \( A := v2 \)
- \( *v2 := 1 \)
- \( v9 := v1 <= 2 \)
- \( v9 := v2 + v8 \)
- \( *v9 := 1 \)
- \( t := 1 \)
- \( i := 1 \)
- goto Block 3

Block 2:
- \( v13 := t \)
- \( v14 := n \)
- \( v16 := v14 + 1 \)
- \( v17 := t \)
- \( v18 := v16 - v17 \)
- \( v19 := v13 \times v18 \)
- \( v21 := v19 \text{ div } v17 \)
- \( v22 := A \)
- \( v25 := v17 <= 2 \)
- \( v26 := v22 + v25 \)
- \( *v26 := v21 \)
- \( v31 := v14 - v17 \)
- \( v33 := v31 <= 2 \)
- \( v34 := v33 <= 2 \)
- \( *v34 := v21 \)
- \( v38 := v17 + 1 \)
- \( t := v21 \)
- \( i := v38 \)
- goto Block 3

Block 3:
- \( v39 := t \)
- \( v40 := n \)
- \( v42 := v40 >> 1 \)
- \( v43 := v39 <= v42 \)
- if \( v43 \) goto Block 2
- else goto Block 4

Block 4:
- \( sp := sp + 8 \)
- goto *ra
### Reading Assignment

- Read Scott
  - Sect. 8.2.3
  - Ch. 13.3

### Peephole Optimization

#### Common Techniques

**Elimination of redundant loads and stores**

\[
\begin{align*}
    r2 &:= r1 + 5 \\
i &:= r2 \\
r3 &:= i \\
r4 &:= r3 \times 3
\end{align*}
\]

becomes

\[
\begin{align*}
    r2 &:= r1 + 5 \\
i &:= r2 \\
r4 &:= r2 \times 3
\end{align*}
\]

**Constant folding**

\[
\begin{align*}
    r2 &:= 3 \times 2 \\
\end{align*}
\]

becomes

\[
\begin{align*}
    r2 &:= 6
\end{align*}
\]
**Peephole Optimization**

**Common Techniques**

**Constant propagation**

\[
\begin{align*}
 r2 &:= 4 \\
 r3 &:= r1 + r2 \quad \text{becomes} \quad r2 := 4 \\
 r2 &:= \ldots \\
 r3 &:= r1 + r2 \quad \text{becomes} \quad r2 := \ldots \\
 r3 &:= \star r3
\end{align*}
\]

\[
\begin{align*}
 r2 &:= 4 \\
 r3 &:= r1 + r2 \quad \text{becomes} \quad r3 := r1 + 4 \\
 r3 &:= \star r3 \quad \text{and then} \quad r3 := \star(r1+4)
\end{align*}
\]

\[
\begin{align*}
 r1 &:= 3 \\
 r2 &:= r1 \times 2 \quad \text{becomes} \quad r2 := 3 \times 2 \\
 r1 &:= 3 \\
 r2 &:= 6
\end{align*}
\]

**Copy propagation**

\[
\begin{align*}
 r2 &:= r1 \\
 r3 &:= r1 + r2 \quad \text{becomes} \quad r2 := r1 \\
 r2 &:= 5 \\
 r2 &:= 5 \\
 r3 &:= r1 + r1 \quad \text{and then} \quad r3 := r1 + r1
\end{align*}
\]

**Strength reduction**

\[
\begin{align*}
 r1 &:= r2 \times 2 \quad \text{becomes} \quad r1 := r2 + r2 \\
 r1 &:= r2 \div 2 \quad \text{becomes} \quad r2 := r2 \gg 1 \\
 r1 &:= r2 \times 0 \quad \text{becomes} \quad r1 := 0
\end{align*}
\]
Peephole Optimization
Common Techniques

Elimination of useless instructions

\[ r_1 := r_1 + 0 \]
\[ r_1 := r_1 \times 1 \]