CSc 553

Principles of Compilation

X16 : Optimization I

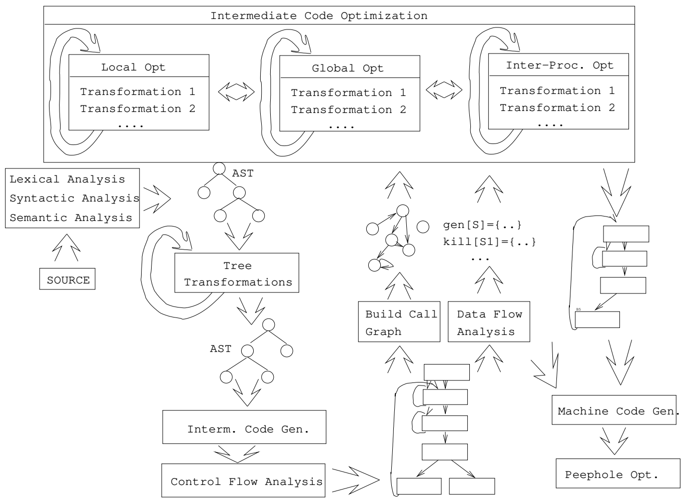
Department of Computer Science
University of Arizona

collberg@gmail.com

Christian Collberg

March 30, 2011

Introduction



Intermediate Code Optimization
Local Opt
Transformation 1
Transformation 2
....
Global Opt
Transformation 1
Transformation 2
....
Inter-Proc. Opt
Transformation 1
Transformation 2
....
Lexical Analysis
Syntactic Analysis
Semantic Analysis
AST
SOURCE
Tree
Transformations
AST
Interm. Code Gen.
Control Flow Analysis
gen[S]={..}
kill[S1]={..}
...
Build Call
Graph
Data Flow
Analysis
Machine Code Gen.
Peephole Opt.

What do we Optimize?

What do we Optimize I?

1. Optimize everything, all the time. The problem is that optimization interferes with debugging. In fact, many (most) compilers don't let you generate an optimized program with debugging information. The problem of debugging optimized code is an important research field.
   Furthermore, optimization is probably the most time consuming pass in the compiler. Always optimizing everything (even routines which will never be called!) wastes valuable time.
2. The programmer decides what to optimize. The problem is that the programmer has a local view of the code. When timing a program programmers are often very surprised to see where most of the time is spent.

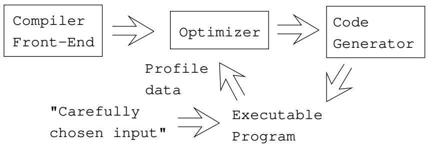
What do we Optimize II?

3. Turn optimization on when program is complete. Unfortunately, optimizers aren't perfect, and a program that performed OK with debugging turned on often behaves differently when debugging is off and optimization is on.
4. Optimize inner loops only. Unfortunately, procedure calls can hide inner loops:

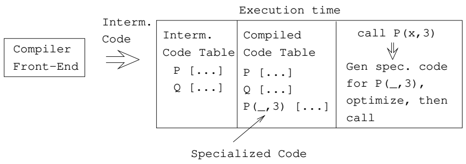
```
PROCEDURE P(n);
BEGIN
   FOR k:=1 TO n DO ··· END;
END P;

FOR i:=1 TO 10000 DO P(i) END;
```

What do we Optimize III?

5. Use profiling information to guide what to optimize.



6. Runtime code generation/optimization. We delay code generation and optimization until execution time. At that time we have more information to guide the otpimizations:



Local vs. Global vs. Inter-procedural Optimization

Local, Global, Inter-Proc. I

- Some compilers optimize more **aggressively** than others. An aggressive compiler optimizes over a large piece of code, a simple one only considers a small chunk of code at a time.

Local Optimization

- Consider each basic block by itself.
- All compilers do this.

Global Optimization

- Consider each procedure by itself.
- Most compilers do this.

Inter-Procedural Opt.

- Consider the control flow between procedures.
- A few compilers do this.

Local Optimization I

Transformations:

- Local common subexpression elimination.
- Local copy propagation.
- Local dead-code elimination.
- Algebraic optimization.
- Jump-to-jump removal.
- Reduction in strength.

Peephole Optimization:

- On machine and/or interm. code.

1. Examine a “window” of instructions.
2. Improve code in window.
3. Slide window.
4. Repeat until “optimal”.

Local Opt. II

Redundant Loads:

```
A := A + 1;  ⇒
   set   A, %l0
   set   A, %l1
   ld    [%l1], %l1
   add   %l1, 1, %l1
   st    %l1, [%l0]  ⇒
      set   A, %l0
      ld    [%l0], %l1
      add   %l1, 1, %l1
      st    %l1, [%l0]
```

Jumps-to-jumps:

```
      if a < b goto L1 .....
L1:   goto L2 .....
L2:   goto L3 .....   ⇒
      if a < b goto L3 .....
L1:   goto L3 .....
L2:   goto L3 .....
```

Local Opt. IV

Algebraic Simplification:

```
x := x + 0;     ⇒
x := x - 0;     ⇒
x := x * 1;     ⇒
x := 1 * 1;     ⇒
x := x / 1;     ⇒
x := x ** 2;    ⇒   x := x * x;
f := f / 2.0;   ⇒   f := f * 0.5;
```

Reduction in Strength:

```
x := x * 32; ⇒ x := SHL(x, 5);

x := x * 100; ⇒
x := x * (64 + 32 + 4) ⇒
x := x * 64 + x *32 + x *4 ⇒
x := SHL(x,6) + SHL(x,5) + SHL(x,2)
```

Local, Global,Inter-Proc. II

Original Code

```
FUNCTION P (X,n): INT;
   IF n = 3 THEN RETURN X[1]
   ELSE RETURN X[n];
CONST R = 1;
BEGIN
   K := 3; ...
   IF P(X,K) = X[1] THEN
      X[1] := R * (X[1] ** 2)
```

After Local Opt

```
FUNCTION P (X,n): INT;
   IF n = 3 THEN RETURN X[1]
   ELSE RETURN X[n]
BEGIN
   K := 3; ...
   IF P(X,K) = X[1] THEN
      X[1] := X[1] * X[1]
```

Local, Global, Inter-Proc. III

After Local Opt

```
FUNCTION P (X,n): INT;
   IF n = 3 THEN RETURN X[1]
   ELSE RETURN X[n]
BEGIN
   K := 3;
   ...
   IF P(X,K) = X[1] THEN
      X[1] := X[1] * X[1]
```

After Global Opt

```
FUNCTION P (X,n): INT;
   IF n = 3 THEN RETURN X[1]
   ELSE RETURN X[n]
BEGIN
   ...
   IF P(X,3) = X[1] THEN
      X[1] := X[1] * X[1]
```

Local, Global, Inter-Proc. IV

```
FUNCTION P (X,n): INT;
   IF n = 3 THEN RETURN X[1]
   ELSE RETURN X[n]
BEGIN
   IF P(X,3) = X[1] THEN
      X[1] := X[1] * X[1]
```

After Inter-Procedural Opt

```
BEGIN
   IF TRUE THEN
      X[1] := X[1] * X[1]
```

After Another Local Opt

```
BEGIN
   X[1] := X[1] * X[1]
```

- Delete P if it isn't used elsewhere. This can maybe be deduced by an inter-procedural analysis.

Local Optimization

Local Optimization I

Transformations

- Local common subexpression elimination.
- Local copy propagation.
- Local dead-code elimination.
- Algebraic optimization.
- Jump-to-jump removal.
- Reduction in strength.

Peephole Optimization

- On machine and/or interm. code.

1. Examine a “window” of instructions.
2. Improve code in window.
3. Slide window.
4. Repeat until “optimal”.

Redundant Loads

- A naive code generator will generate the same address or variable several times. Peephole optimization over the generated code will easily remove these.

```
A := A + 1;
      ⇓

set A, %l0
set A, %l1
ld  [%l1], %l1
add %l1, 1, %l1
st  %l1, [%l0]
      ⇓

set A, %l0
ld  [%l0], %l1
add %l1, 1, %l1
st  %l1, [%l0]
```

Jumps-to-jumps

- Complicated boolean expressions (with many `and`, `or`, `nots`) can easily produce lots of jumps to jumps. A peephole optimization pass over the generated code can remove these.

```
    if a < b goto L1
    ...
L1: goto L2
    ...
L2: goto L3
```

$$\Downarrow$$

```
    if a < b goto L3
    ...
L1: goto L3
    ...
L2: goto L3
```

Algebraic Simplification

- Beware of numerical problems:
  1. $(x * 0.00000001) * 10000000000.0$ may produce a different result than $(x * 1000.0)$!
- FORTRAN requires that parenthesis be honored: $(5.0 * x) * (6.0 * y)$ can't be evaluated as $(30.0 * x * y)$.
- Note that multiplication is often faster than division.

```
x := x + 0;    ⇒
x := x - 0;    ⇒
x := x * 1;    ⇒
x := 1 * 1;    ⇒  x := 1
x := x / 1;    ⇒
x := x ** 2;   ⇒  x := x * x;
f := f / 2.0;  ⇒  f := f * 0.5;
```

Reduction in Strength

- SHL(x,y) = shift x left y steps.
- Multiplcation (and division) by a constant is a common operation. They can be replaced by cheaper sequences of shifts and adds.

```
x := x * 32 ;
     ⇓
x := SHL(x, 5);

x := x * 100 ;
     ⇓
x := x * (64 + 32 + 4)
     ⇓
x := x * 64 + x *32 + x *4
     ⇓
x := SHL(x,6) + SHL(x,5) +
      SHL(x,2)
```

Global Optimization

Global Optimization I

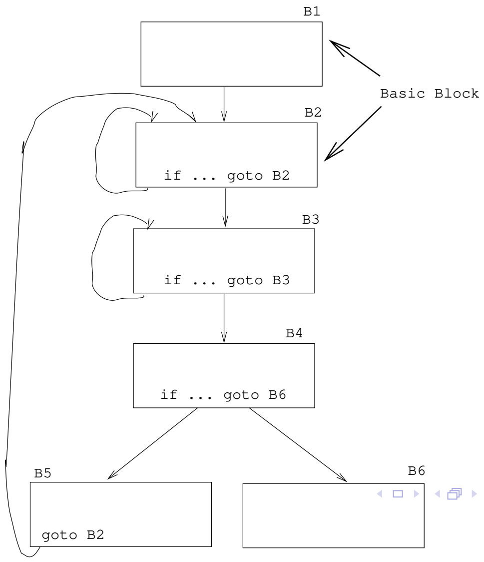
- Makes use of control-flow and data-flow analysis.

Transformations

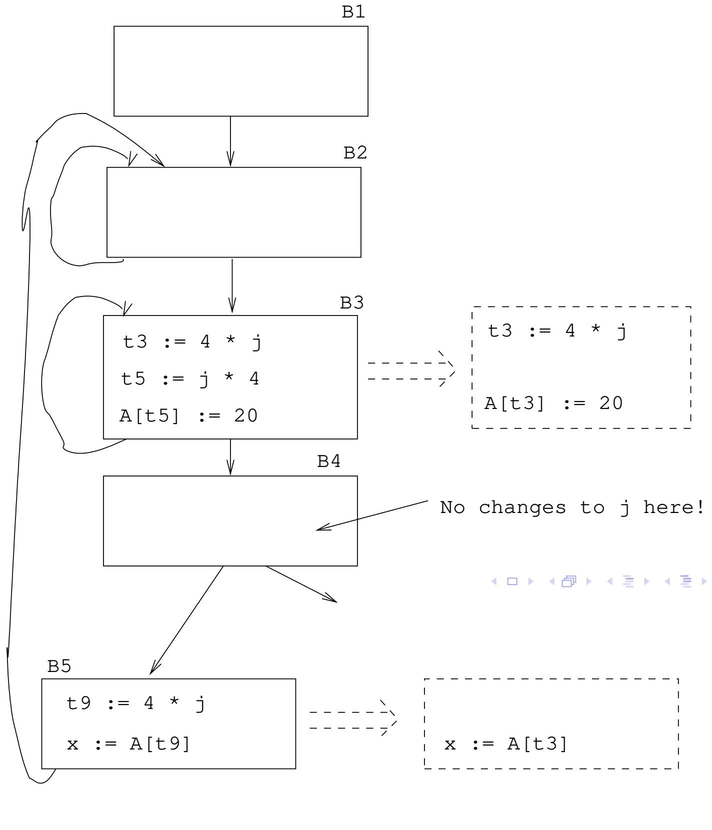
- Dead code elimination.
- Common subexpression elimination (local and global).
- Loop unrolling.
- Code hoisting.
- Induction variables.
- Reduction in strenght.
- Copy propagation.
- Live variable analysis.
- Uninitialized Variable Analysis.

Control Flow Graphs

- We perform our optimizations over the control flow graph of a procedure.

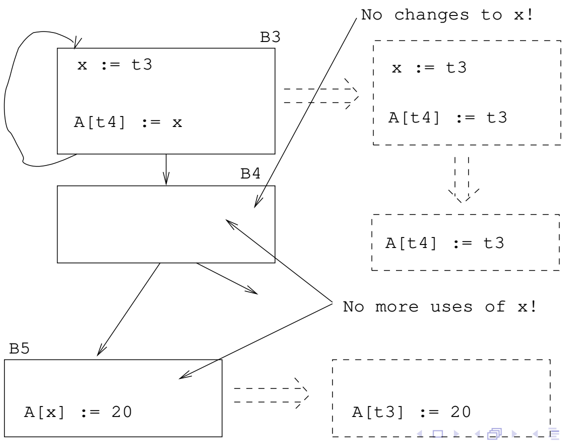


Common Sub-Expr. Elimination



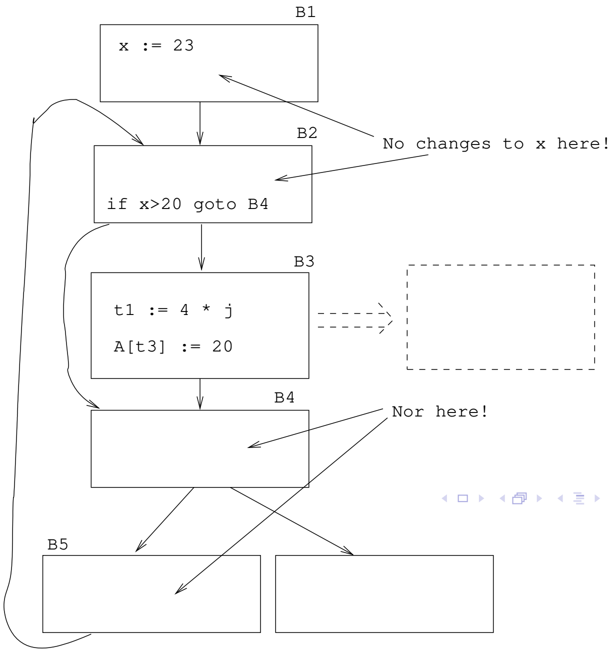
Copy Propagation

- Many optimizations produce $X := Y$.
- After an assignment $X := Y$, replace references to X by Y. Remove the assignment if possible.



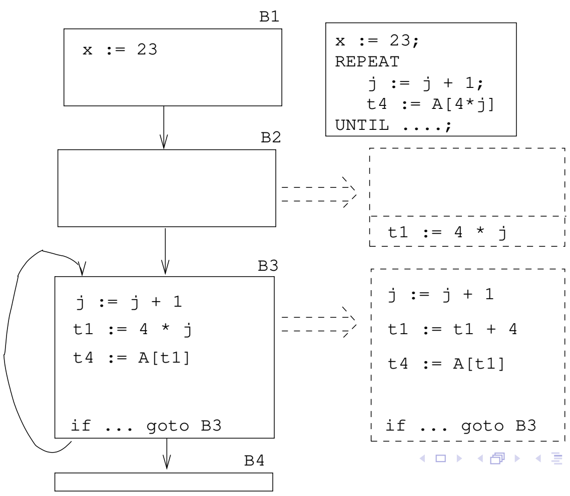
Dead Code Elimination

- A piece of code is dead if we can determine at compile time that it will never be executed.



Induction Variables

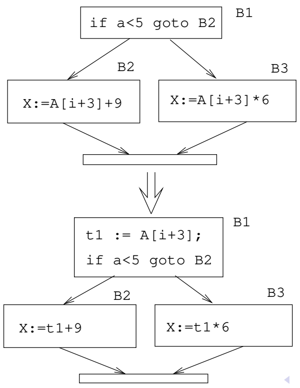
- If i and j are updated simultaneously in a loop, and $j = i * c_1 + c_2$ (c_1, c_2 are consants) we can remove one of them, and/or replace $*$ by $+$.



Code Hoisting

- Move code that is computed twice in different basic blocks to a common ancestor block.

IF `a < 5` **THEN** `X := A[i+3] + 9;`
ELSE `X := A[i+3] * 6` **END**



Loop Unrolling

Loop Unrolling

Constant Bounds

```
FOR i := 1 TO 5 DO A[i]:=i END
            ⇓
A[1] := 1; A[2] := 2; A[3] := 3;
A[4] := 4; A[5] := 5;
```

Variable Bounds

```
FOR i := 1 TO n DO A[i] := i END
            ⇓
i := 1;
WHILE i <= (n-4) DO
   A[i]:=i; A[i+1]:=i+1; A[i+2]:=i+2;
   A[i+3]:=i+3; A[i+4]:=i+4; i:=i+5;
END;
WHILE i<=n DO A[i]:=i; i:=i+1; END
```

- Loop unrolling increases code size. How does this effect caching?

Inter-procedural Optimizations

Inter-procedural Opt.

- Consider the *entire* program during optimization.
- How can this be done for languages that support separately compiled modules?

Transformations

- Inline expansion
  - Replace a procedure call with the code of the called procedure.
- Procedure Cloning
  - Create multiple specialized copies of a single procedure.
- Inter-procedural constant propagation
  - If we know that a particular procedure is always called with a constant parameter with a specific value, we can optimize for this case.

Inline Expansion I

Original Code:

```
FUNCTION Power (n, exp:INT):INT;
   IF exp < 0 THEN result := 0;
   ELSIF exp = 0 THEN result := 1;
   ELSE result := n;
        FOR i := 2 TO exp DO
            result := result * n;
        END; END;
   RETURN result;
END Power;

BEGIN X := 7; PRINT Power(X,2) END;
```

Expanded Code:

```
BEGIN
   X := 7;
      result := X;
      FOR i := 2 TO 2 DO
         result := result * X;
      END;
   PRINT result;
END
```

Inline Expansion II

After copy propagation

```
X := 7;
   result := 7;
   FOR i := 2 TO 2 DO
      result := result * 7;
   END;
PRINT result;
```

After loop unrolling

```
X := 7;
   result := 7;
      result := result * 7;
PRINT result;
```

After constant folding

```
result := 49;
PRINT result;
```

Procedure Cloning

Original Code:

```
FUNCTION Power (n, exp:INT):INT;
   IF exp < 0 THEN result := 0;
   ELSIF exp = 0 THEN result := 1;
   ELSE result := n;
      FOR i := 2 TO exp DO
         result := result * n;
      END;
   RETURN result;
END Power;

BEGIN PRINT Power(X,2), Power(X,7) END;
```

Cloned Routines:

```
FUNCTION Power0 (n):INT; RETURN 1;
FUNCTION Power2 (n):INT; RETURN n * n;
FUNCTION Power3 (n):INT; RETURN n * n * n;
FUNCTION Power (n, exp:INT):INT;
   (* As before *)
```

Transformed Code:

```
BEGIN PRINT Power2(X), Power(X,7) END;
```

Machine Dependent vs. Machine Independent Optimization

Machine (In-)Dependent Opt.? I

- Optimizations such as inline expansion and loop unrolling seem pretty machine independent. You don't need to know anything special about the machine architecture to implement these optimizations, in fact, both inline expansion and loop unrolling can be applied at the source code level. (May or may not be true for inline expansion, depending on the language).
- However, since both inline expansion and loop unrolling normally increase the code size of the program, these optimizations do, in fact, interact with the hardware.

Machine (In-)Dependent Opt.? I

- A loop that previously might have fit in the instruction cache of the machine, may overflow the cache once it has been unrolled, and therefore increase the cache miss rate so that the unrolled loop runs slower than the original one.
- The unrolled loop may even be spread out over more than one virtual memory page and hence affect the paging system adversely.
- The same argument holds for inline expansion.

Example I

Example I/a – Loop Invariants

Original code:

```
FOR I:= 1 TO 100 DO
   FOR J := 1 TO 100 DO
      FOR K := 1 TO 100 DO
         A[I][J][K] := (I*J)*K;
      END;
   END;
END
```

Find loop invariants:

```
FOR I:= 1 TO 100 DO
   T3 := ADR(A[I]);
   FOR J := 1 TO 100 DO
      T1 := ADR(T3[J]);
      T2 := I * J;
      FOR K := 1 TO 100 DO
         T1[K] := T2 * K
      END;
   END;
END
```

Example I/b – Strength Reduct.

```
FOR I:= 1 TO 100 DO
   T3 := ADR(A[I]);
   FOR J := 1 TO 100 DO
      T1 := ADR(T3[J]); T2 := I * J;
      FOR K := 1 TO 100 DO T1[K]:=T2*K END;
   END;
END
```

After strength reduction:

```
FOR I:= 1 TO 100 DO
   T3 := ADR(A[I]); T4 := I;
   FOR J := 1 TO 100 DO
      T1 := ADR(T3[J]);
      T2 := T4; (* T4 = I*J *)
      T5 := T2; (* Init T2*K *)
      FOR K := 1 TO 100 DO
         T1[K] := T5; T5 := T5 + T2;
      END;
      T4 := T4 + I;
   END; END
```

- T4 holds `I*J`: $I, I+I, I+I+I, \cdots I*J$.
  T5 holds `T2*K = I*J*K`.

Example I/c – Copy Propagation

```
FOR I:= 1 TO 100 DO
   T3 := ADR(A[I]); T4 := I;
   FOR J := 1 TO 100 DO
      T1 := ADR(T3[J]); T2 := T4; T5 := T2;
      FOR K := 1 TO 100 DO
         T1[K] := T5; T5 := T5 + T2;
      END; T4 := T4 + I;
   END;
END
```

After Copy Propagation:

```
FOR I:= 1 TO 100 DO
   T3 := ADR(A[I]); T4 := I;
   FOR J := 1 TO 100 DO
      T1 := ADR(T3[J]); T5 := T4;
      FOR K := 1 TO 100 DO
         T1[K] := T5; T5 := T5 + T4;
      END;
      T4 := T4 + I;
   END;
END
```

- We replace T2 by T4.

Example I/d – Array Indexing

- Expand subscripting operations. Pascal array indexing turns into C-like address manipulation!

Expand Indexing:

```
VAR A:ARRAY[1..100,1..100,1..100] OF INT;

FOR I:= 1 TO 100 DO
   T3 := ADR(A) + (10000*I)-10000;
   T4 := I;
   FOR J := 1 TO 100 DO
      T1 := T3 +(100*J)-100;
      T5 := T4;
      FOR K := 1 TO 100 DO
         (T1+K-1)↑ := T5;
         T5 := T5 + T4;
      END;
      T4 := T4 + I;
   END; END
```

Example I/e – Array Indexing

Strength Red. + Copy Prop.:

```
T6 := ADR(A);
FOR I:= 1 TO 100 DO
   T4 := I;
   T7 := T6;
   FOR J := 1 TO 100 DO
      T5 := T4;
      T8 := T7;
      FOR K := 1 TO 100 DO
         T8↑ := T5;
         T5 := T5 + T4;
         T8 := T8 + 1;
      END;
      T4 := T4 + I;
      T7 := T7 + 100;
   END;
   T6 := T6 + 10000;
END
```

Example I/f – Loop Unrolling

```
T6 := ADR(A);
FOR I:= 1 TO 100 DO
   T4 := I; T7 := T6;
   FOR J := 1 TO 100 DO
      T5 := T4; T8 := T7;
      FOR K := 1 TO 10 DO
         T8↑ := T5; T5 += T4; T8 ++;
         T8↑ := T5; T5 += T4; T8 ++;
         T8↑ := T5; T5 += T4; T8 ++;
         T8↑ := T5; T5 += T4; T8 ++;
         T8↑ := T5; T5 += T4; T8 ++;
         T8↑ := T5; T5 += T4; T8 ++;
         T8↑ := T5; T5 += T4; T8 ++;
         T8↑ := T5; T5 += T4; T8 ++;
         T8↑ := T5; T5 += T4; T8 ++;
         T8↑ := T5; T5 += T4; T8 ++;
      END;
      T4 := T4 + I; T7 := T7 + I00;
   END;
   T6 := T6 + 10000;
END
```

Example II

Example II/a – Inline Expansion

- `ftp://cs.washington.edu/pub/pardo`. The code has been simplified substantially...
- `bitblt` copies image region regions while performing an operation on the moved part.
- `s` is the source, `d` the destination, `i` the index in the x direction, `j` the index in the y direction.
- Every time around the loop we have to execute a switch (case) statement, which is very inefficient.
- Here we'll show how `bitblt` can be optimized by inlining. It's also amenable to **run-time** (dynamic) code generation. I.e. we include the code generator in the executable and generate code for `bitblt` when we know what it's arguments are.

Example II/b – Inline Expansion

Original Code

```
#define BB_S (0xc)
bitblt (mask_t m, word s, word d, int op)
{for (j=0; j<dy; ++j) {
  for (i=nw+1; i>0; --i) {
   switch (op) {
     case (0) :  *d &= ~mask;  break;
     case (BB_D&~BB_S) :
        *d ^= ((s &*d) & mask); break;
     case (~BB_S) :
        *d ^= ((~s ^ *d) & mask); break;
     /* Another 12 cases... */
     case (BB_X) : *d |= mask; break;
   }; d++;
  }; d++; s++;
 }
}
main () {bitblt(mask,src,dest,...,BB_S);}
```

Example II/c – Inline Expansion

Expanded Code

```
main () {
d = src; s=dst;
for (j=0; j<dy; ++j) {
  for (i=nw+1; i>0; --i) {
   switch (BB_S) {
     case (0) : *d &= ~mask; break;
     case (BB_D&~BB_S) :
        *d ^= ((s &*d) & mask); break;
     case (~BB_S) :
        *d ^= ((~s ^ *d) & mask); break;
     /* Another 12 cases... */
     case (BB_X) :  *d |= mask; break;
   }; d++;
  }; d++; s++;
 }
}
```

Example II/d – Inline Expansion

After Dead Code Elim

```
main () {
   d = src; s=dst;
   for (j=0; j<dy; ++j) {
      for (i=nw+1; i>0; --i) {
         d ^= ((s ^ *d) & mask);
         d++;
     };
     d++; s++;
}
```

Summary

Summary

- Read the Dragon book: 530–532, 585–602.
- Debugging optimized code: See the Dragon book. pp. 703–711.
- Difficult problems:
  - Which transformations are actually profitable?
  - How do we avoid unsafe optimizations?
  - What part of the code should we optimize?
  - How do we take machine dependencies (cache size) into account?
  - At which level(s) do we optimize (source, interm. code, machine code)?
  - How do we order the different optimizations?