# CSc 453 — Compilers and Systems Software

**22 : Optimization**

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

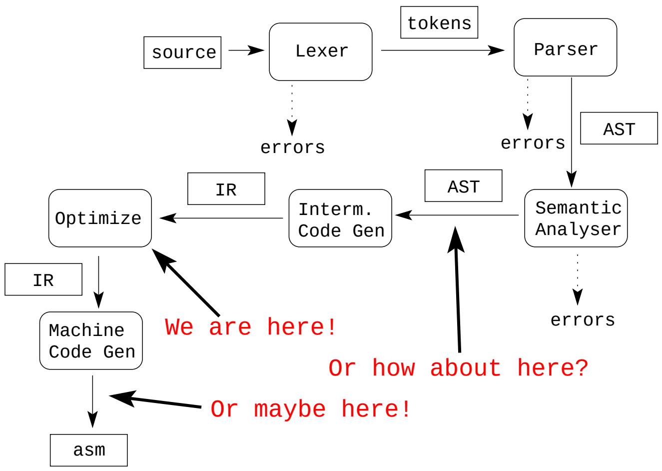


November 1, 2009

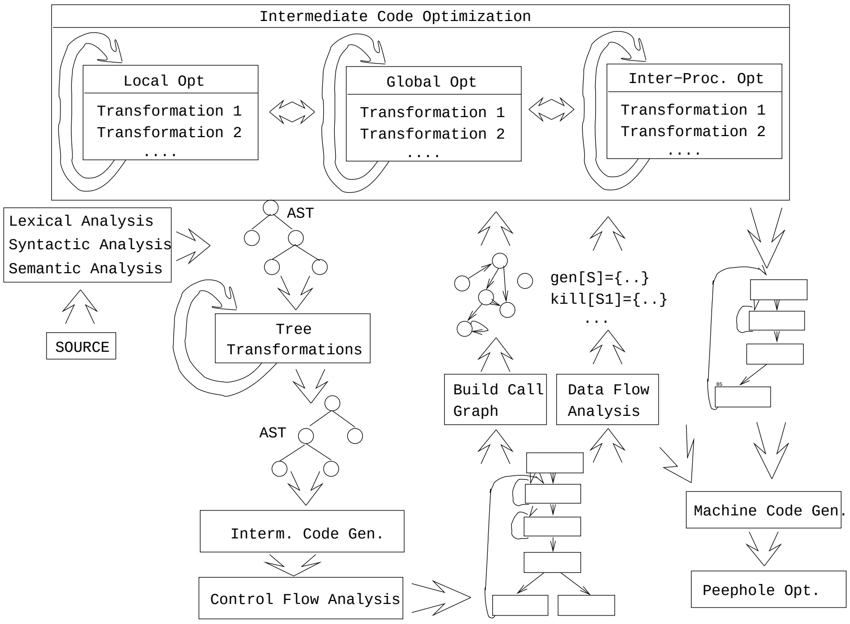
## 1

# Introduction

## 2 Compiler Phases



## 3




## 4

# What do we Optimize?

## 5 What do we Optimize?

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.

## 6

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.

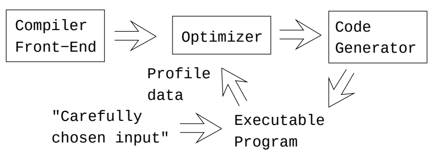
4. Optimize inner loops only. Unfortunately, procedure calls can hide inner loops:

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

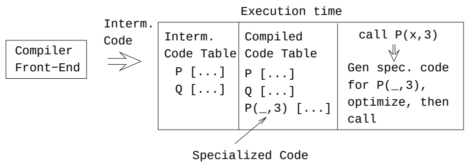
```
FOR i:=1 TO 10000 DO P(i) END;
```

## 7

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:




## 8

# Local vs. Global vs. Inter-procedural Optimization

## 9 Local, Global, Inter-Proc.

- 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.)

**Global Optimization:**

- Consider each procedure by itself. (Most compilers.)

**Inter-Procedural Optimization:**

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

## 10 Local Optimization — Transformations

- Local common subexpression elimination.
- Local copy propagation.
- Local dead-code elimination.

- Algebraic optimization.
- Jump-to-jump removal.
- Reduction in strength.

## 11 Peephole Optimization

- Can be done at the machine code level or at the intermediate code level.

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

## 12

# Local Optimization

## 13 Peephole optimization—Redundant Loads

- A naive code generator will generate the same address or variable several times.

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]
```

## 14 Peephole optimization—Jumps-to-jumps

- Complicated boolean expressions (with many `and`, `or`, `nots`) can easily produce lots of jumps to jumps.

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

## 15 Algebraic Simplification

- Beware of numerical problems:

  $$(x * 0.00000001) * 10000000000.0$$

  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.

## 16 Algebraic Simplification...

```
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;
```

## 17 Reduction in Strength

- `SHL(x,y)` = shift `x` left `y` steps.

- Multiplications (and divisions) by constants 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)
```

## 18 Local, Global,Inter-Procedural

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)
```

## 19

```
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 Optimization

```
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]
```

## 20

```
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 Optimization

```
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]
```

## 21

```
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.

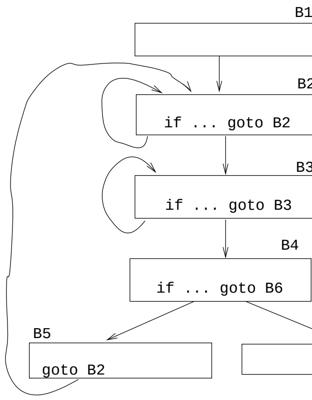
**22**

# Global Optimization

## 23 Global Optimization

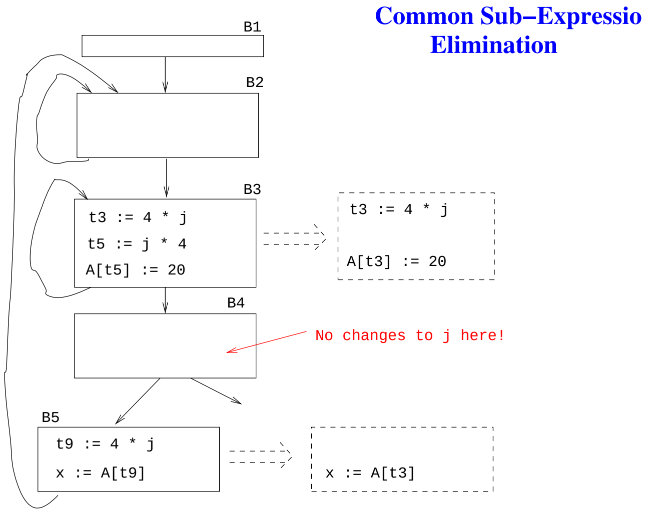
- Makes use of control-flow and data-flow analysis.
- Dead code elimination.
- Common subexpression elimination (local and global).
- Loop unrolling.
- Code hoisting.
- Induction variables.
- Reduction in strenght.
- Copy propagation.
- Live variable analysis.
- Uninitialized Variable Analysis.

## 24 Control Flow Graphs

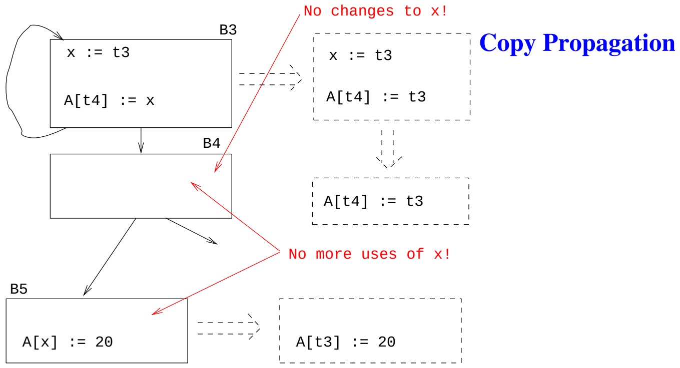



Perform optimizations over the control flow graph of a procedure.

## 25

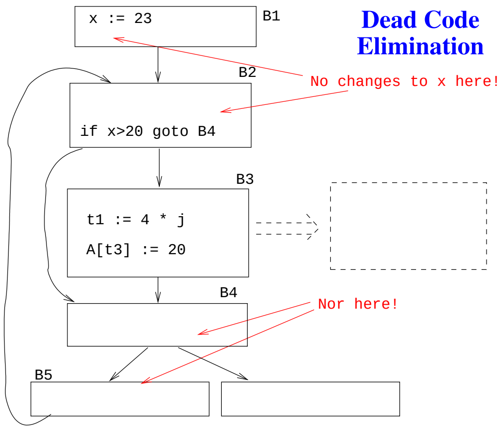



## 26



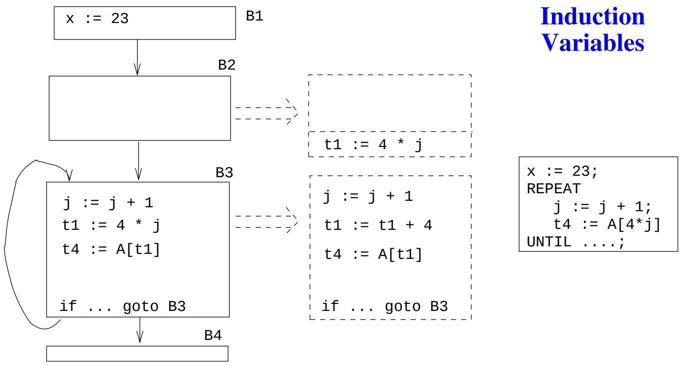

- Many optimizations produce $X := Y$.
- After an assignment $X := Y$, replace references to $X$ by $Y$. Remove the assignment if possible.

## 27



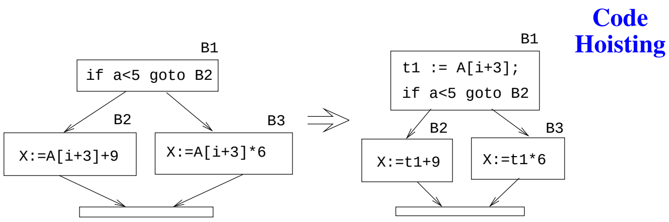

- A piece of code is dead if we can determine at compile time that it will never be executed.

## 28




- 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 $+$.

## 29




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

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

## 30

# Loop Unrolling

## 31 Loop Unrolling – Constant Bounds

**FOR** `i := 1` **TO** `5` **DO**
  `A[i]:=i`
**END**

$\Downarrow$

```
A[1] := 1; A[2] := 2; A[3] := 3;
A[4] := 4; A[5] := 5;
```

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

## 32 Loop Unrolling – 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
```

## 33

# Inter-procedural Optimizations

## 34 Inter-procedural Optimization

- 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.

## 35 Inline Expansion—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;
```

## 36 Inline Expansion—Expanded Code

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

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

## 37 Inline Expansion—After copy propagation

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

## 38 Inline Expansion—More optimization

After loop unrolling

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

After constant folding

```
BEGIN
   result := 49;
   PRINT result;
END
```

## 39 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;
```

## 40 Procedure Cloning—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 *)
END Power;
```

Transformed Code:

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

## 41

# Machine Dependent vs. Machine Independent Optimization

## 42 Machine (In-)Dependent Optimization?

- 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.

## 43 Machine (In-)Dependent Optimization?

- 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.

## 44

# Example

## 45

# Loop Invariants

## 46 Example – 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
```

## 47 Example/a – Find Loop Invariants

```
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
```

```
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
```

## 48

# Strength Reduction

## 49 Example/b – Strength Reduction

```
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
```

```
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`.

## 50

# Copy Propagation

## 51 Example/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
```

```
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.

## 52

# Strength Reduction...

## 53 Example/d – Expand Array Indexing

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

```
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
```

## 54 Example/e – 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 + I00;
   END;
   T6 := T6 + 10000;
END
```

## 55

# Loop Unrolling

## 56 Example/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
```

## 57

# Example...

## 58 Example – 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.

## 59 Example – Inline Expansion...

- 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 its arguments are.

## 60

```
#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);}
```

## 61 Example — 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++;
   }
}
```

## 62 Example – Dead Code Elimination

```
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++;
   }
}
```

## 63 Proebsting's Law

**Compiler Advances Double Computing Power Every 18 Years**

*I claim the following simple experiment supports this depressing claim. Run your favorite set of benchmarks with your favorite state-of-the-art optimizing compiler. Run the benchmarks both with and without optimizations enabled. The ratio of of those numbers represents the entirety of the contribution of compiler optimizations to speeding up those benchmarks. Let's assume that this ratio is about 4X for typical real-world applications, and let's further assume that compiler optimization work has been going on for about 36 years. These assumptions lead to the conclusion that compiler optimization advances double computing power every 18 years. QED.*

## 64 Proebsting's Law...

*This means that while hardware computing horsepower increases at roughly 60%/year, compiler optimizations contribute only 4%. Basically, compiler optimization work makes only marginal contributions.*

*Perhaps this means Programming Language Research should be concentrating on something other than optimizations. Perhaps programmer productivity is a more fruitful arena.* `http://research.microsoft.com/~toddpro/papers/law.htm`

## 65

# Summary

## 66 Readings and References

- Read Louden: 468–484.
- Or, read the Dragon book: 530–532, 585–602.
- Debugging optimized code: See the Dragon book. pp. 703–711.
- Also see this support for Proebsting's Law: *On Proebsting's Law*, `http://www.cs.virginia.edu/~jks6b/on_proebstings_law.pdf` by Kevin Scott.

## 67 Summary

- 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?