Introduction
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.

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:

   ```pascal
   PROCEDURE P(n);
   BEGIN
     FOR k:=1 TO n DO · · · END;
   END P;
   FOR i:=1 TO 10000 DO P(i) END;
   ```

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 optimizations:
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.)

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

**Peephole Optimization**
- Can be done at the machine code level or at the intermediate code level.
- Examine a “window” of instructions.
- Improve code in window.
- Slide window.
- Repeat until “optimal”.

---

Local vs. Global vs. Inter-procedural Optimization

---

Local, Global, Inter-Proc.
Local Optimization

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

\[
A := A + 1; \Rightarrow \begin{align*}
&\text{set } A, \%10 \\
&\text{set } A, \%11 \\
&\text{ld } [\%11], \%11 \\
&\text{add } \%11, 1, \%11 \\
&\text{st } \%11, [\%10] \\
&\downarrow \\
&\text{set } A, \%10 \\
&\text{ld } [\%10], \%11 \\
&\text{add } \%11, 1, \%11 \\
&\text{st } \%11, [\%10]
\end{align*}
\]

- Peephole optimization—Redundant Loads

- Peephole optimization—Jumps-to-jumps

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

\[
\text{if } a < b \text{ goto } L1 \Rightarrow \text{if } a < b \text{ goto } L3 \\
L1: \text{goto } L2 \\
\ldots \\
L2: \text{goto } L3 \\
\ldots \\
L1: \text{goto } L3 \\
\ldots \\
L2: \text{goto } L3
\]

Algebraic Simplification

- Beware of numerical problems:

\[
(x \times 0.00000001) \times 10000000000.0
\]

may produce a different result than

\[
(x \times 1000.0)
\]

- FORTRAN requires that parenthesis be honored:

\[
(5.0 \times x) \times (6.0 \times y)
\]

can't be evaluated as \((30.0 \times x \times y)\).

- Note that multiplication is often faster than division.
Algebraic Simplification...

\[ x := x + 0; \Rightarrow \]
\[ x := x - 0; \Rightarrow \]
\[ x := x \times 1; \Rightarrow \]
\[ x := 1 \times 1; \Rightarrow x := 1 \]
\[ x := x / 1; \Rightarrow \]
\[ x := x ** 2; \Rightarrow x := x \times x; \]
\[ f := f / 2.0; \Rightarrow f := f \times 0.5; \]

Reduction in Strength

\[ \text{SHL}(x, y) = \text{shift } x \text{ left } y \text{ steps.} \]

- Multiplications (and divisions) by constants can be replaced by cheaper sequences of shifts and adds.

\[ x := x \times 32 \Rightarrow x := \text{SHL}(x, 5); \]
\[ x := x \times 100 \]
\[ \downarrow \]
\[ x := x \times (64 + 32 + 4) \]
\[ \downarrow \]
\[ x := x \times 64 + x \times 32 + x \times 4 \]
\[ \downarrow \]
\[ x := \text{SHL}(x, 6) + \text{SHL}(x, 5) + \text{SHL}(x, 2) \]

Local, Global, Inter-Procedural

\[ \text{FUNCTION P(X,n): INT;} \]
\[ \quad \text{IF } n = 3 \text{ THEN RETURN } X[1] \text{ ELSE RETURN } X[n]; \]
\[ \quad \text{CONST R = 1;} \]
\[ \quad \text{BEGIN} \]
\[ \quad \quad K := 3; \ldots \]
\[ \quad \quad \text{IF } P(X,K) = X[1] \text{ THEN } X[1] := R \times (X[1] \times 2) \]
\[ \quad \text{BEGIN} \]
\[ \quad \quad K := 3; \ldots \]
\[ \quad \quad \text{IF } P(X,K) = X[1] \text{ THEN } X[1] := X[1] \times X[1] \]
FUNCTION P (X,n): INT;
    IF n=3 THEN RETURN X[1] ELSE RETURN X[n]
BEGIN
    K := 3;
    ...
END

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

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

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

---

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 strength.
- Copy propagation.
- Live variable analysis.
- Uninitialized Variable Analysis.
Control Flow Graphs

Perform optimizations over the control flow graph of a procedure.

- **Common Sub-Expression Elimination**
  - $t_9 := 4 \times j$
  - $x := A[t_9]$  
  - $x := A[t_3]$
  - $t_3 := 4 \times j$
  - $t_5 := j \times 4$
  - $A[t_5] := 20$
  - $t_3 := 4 \times j$
  - No changes to $j$ here!

- **Copy Propagation**
  - $x := t_3$
  - $A[t_4] := t_3$
  - $A[t_4] := t_3$
  - No more uses of $x$!

- **Dead Code Elimination**
  - if $x > 20$ goto $B_4$
  - $t_1 := 4 \times j$

- **Many optimizations produce $X := Y$.**
- After an assignment $X := Y$, replace references to $X$ by $Y$. Remove the assignment if possible.
- A piece of code is dead if we can determine at compile time that it will never be executed.
If \( i \) and \( j \) are updated simultaneously in a loop, and \( j = i \cdot c_1 + c_2 \) (\( c_1, c_2 \) are constants) we can remove one of them, and/or replace \( \cdot \) by +.

**Loop Unrolling**

**FOR** \( i := 1 \) **TO** 5 **DO**

\[
A[i] := i \\
\]

**END**

\[
\]

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

Inter-procedural Optimizations

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.

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;
Inline Expansion—Expanded Code

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

Inline Expansion—After copy propagation

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

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

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;
machine dependent vs. machine independent 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.

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

Loop Invariants

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

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
END
**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

$T4$ holds $I*J: I, I + I, I + I + I, \ldots, I*J$.

$T5$ holds $T2*K = I*J*K$.

---

**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
    - $T1[K] := T5$
    - END;
  - END;
  - $T4 := T4 + I$
- END;
END

We replace $T2$ by $T4$.

---

**Strength Reduction**

**Copy Propagation**
Example/d – Expand Array Indexing

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

```pascal
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; END
```

Example/e – Strength Red. + Copy Prop.

```pascal
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
```

Loop Unrolling
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++;
    END;
  END;
  T4:=T4 + I; T7:=T7 + 100;
END; T6:=T6 + 10000;
END

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.

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.
```c
#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);
                case (~BB_S) : *d ^= ((~s ^ *d) & mask);
                /* Another 12 cases... */
                case (BB_X) : *d |= mask; break;
            }; d++;
        }; d++; s++;
    }
}

main() {bitblt(mask,src,dest,...,BB_S);}
```

**Example — Expanded Code**

```c
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);
                case (~BB_S) : *d ^= ((~s ^ *d) & mask);
                /* Another 12 cases... */
                case (BB_X) : *d |= mask; break;
            }; d++;
        }; d++; s++;
    }
}
```

**Example — Dead Code Elimination**

```c
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++;
    }
}
```

**Proebsting’s Law**

```
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 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.
```
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

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?

Readings and References

- Read Louden: 468–484.