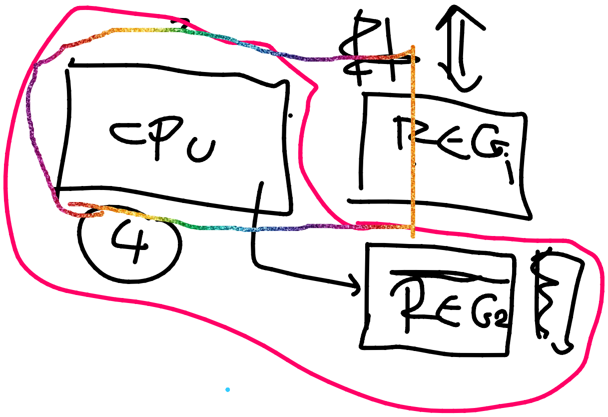
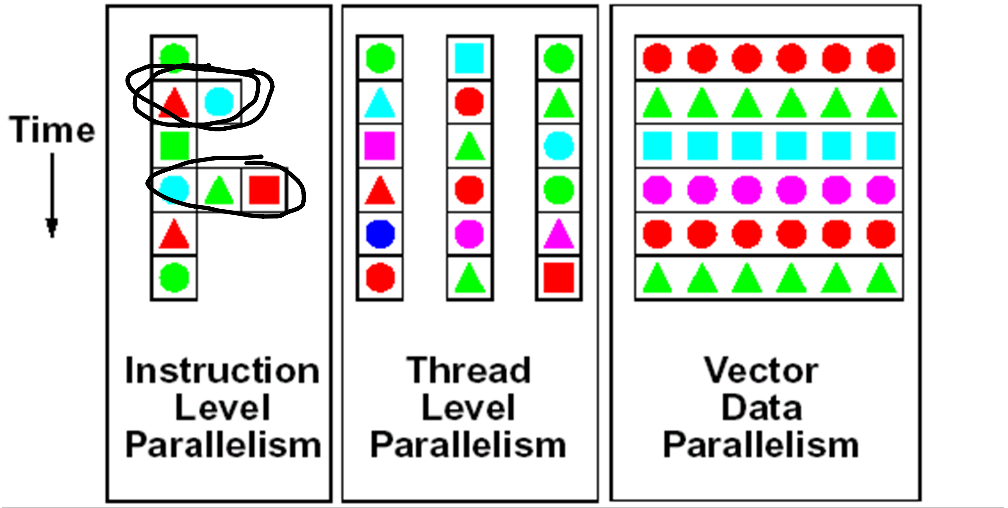
W1

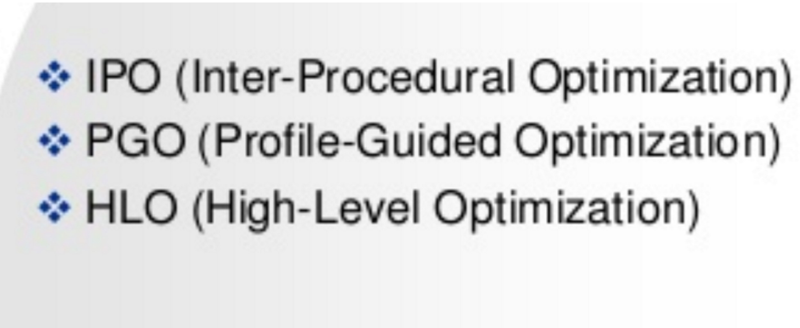
Thursday, October 15, 2020 10:38 AM



- Instruction Level Parallelism (ILP)
- Thread Level Parallelism (TLP)
- vector Data Parallelism (DP)



Instruction Level Parallelism (ILP)



- IPO (Inter-Procedural Optimization)
- PGO (Profile-Guided Optimization)
- HLO (High-Level Optimization)

IPO

Copy Propagation

- Consider

  X=Y
  Z=1.0+X
- The compiler might change this to

  X=Y
  Z=1.0+Y

```
a = 5;
b = 3;
    :
    :
n = a + b;
for (i = 0 ; i < n ; ++i) {
        :
}
```

n = 5 + 3

const
volatile
register
auto

```
n = 5 + 3;
for (i = 0 ; i < n ; ++i) {
        :
}
```

n = 8

8

Constant Folding

- Consider
  - const int J=100;
  - const int K=200;
  - int M= J+K;

volatile int k;

J=J+10;

vo k=a+15f k;

register int k;

int q=0;

q=4;

while(q==4){:::}

Dead Code Removal

```
debug = 0;  // set to False
    :
if (debug) {
    :
}
a = f(b);
```

volatile int debug;

⇒

debug = 0;
:
a = f(b);

int interrupt(...){
q=3;
}

#define Q 30

const int Q=30;

Strength Reduction

- Replace complex or difficult or expensive operations with simpler ones.
  - replacing division by a constant with multiplication by its reciprocal

int y=2*x;

y<<=1;

- Consider

  Y=X**2
  - ✓ Raising a Number to an exponent
    - first X is converted to a logarithm and then multiplied by two and then converted back.
- Y=X*X Is much more efficient.
  - Multiplication is Easier than raising a number to a power.

y=x*x;

- Y=X*X Is much more efficient.
  - Multiplication is Easier than raising a number to a power.

y = x*x;

y = pow(x,2.0);

x*5

(x<<2)+x

x<<2 +x

Induction Variable Simplification

- Consider

```
for(i=1; i<=N; i++)
{
        K=i*4+M;
        C=2*A[K];
        .....
}
```

ΑΡΧΙΖΟΥΜΕ 15:15

- Optimized as

```
K=M;
for(i=1; i<=N; i++)
{
        K=K+4;
        C=2*A[K];
        .......
}
```

f(x,n)

int a

a;

inline int f(~~void~~){ - - - - - - - }

Function In-lining & "inline" directive

- Consider

```
for(i=0; i<=N ; i++)
{
    ke[i]=kinetic_energy(mass[i], velocity[i]);
}
```

- Optimized as

```
inline float kinetic_energy(float m, float v)
{
    return 0.5*m*v*v;
}
```

```
foo(int z){
  int m = 5;
  return z + m;
}
main(){
  …
  x = foo(x);
  …
}
```

```
main(){
 …
 {
  int foo_z = x;
  int m = 5;
  int foo_return = foo_z + m;
  x = foo_return;
 }
 …
}
```

```
main(){
  …
  x = x + 5;
  …
}
```

Loop Invariant Conditionals

- Consider

N = . . .

```
DO I=1,K
        IF ( N.EQ.0 ) THEN
                A(I)=A(I)+B(I)*C
        ELSE
                A(I)=0
        ENDIF
ENDDO
```

- Optimized as

```
IF ( N.EQ.0 ) then
    DO I=1,k
            A(I)=A(I)+B(I)*C
    ENDDO
ELSE
    DO I=1,K
            A(I)=0
    ENDDO
ENDIF
```

```
for (i=0; i < 100 ;  ++i) {
  for (j=0; j < 100 ; ++j) {
    for (k=0 ; k < 100 ; ++k) {
      a[i][j][k] = i*j*k;
    }
  }
}
```

⇒

reg.

```
for (i = 0; i < 100 ; ++i) {
  t1 = a[i];
  for (j = 0; j < 100 ; ++j) {
    tmp = i * j;
    t2 = t1[j];
    for (k = 0 ; k < 100 ; ++k) {
      t2[k] = tmp * k;
    }
  }
}
```

Variable Renaming

- Consider

X=Y*Z
Q=R+X+X
X=A+B

- Optimized as

X_=Y*Z
q=R + X_ + X_
X= A+B

12 35 20
24 18 7
36 53 27

1203520
2401807
365327

union

X= A+B

24 0 18 0 7

36 53 27

union

$\underline{\text{unsigned char } Q_1}; Q_2, \ldots Q_8;$

$Q_1 + Q_2 \quad Q_3 + Q_4 \quad Q_5 + Q_6$

$Q_7 + Q_8$

Common Sub-Expression Elimination

- Consider
  - A=C*(F+G)
  - D=(F+G)/N
- Optimized as
  - temp=F+G
  - A=C*temp
  - D=temp/N

```
     a = c * d;
              :
              :
d = (c * d + t) * u
```

⇒

```
a = c * d;
       :
d = (a + t) * u
```

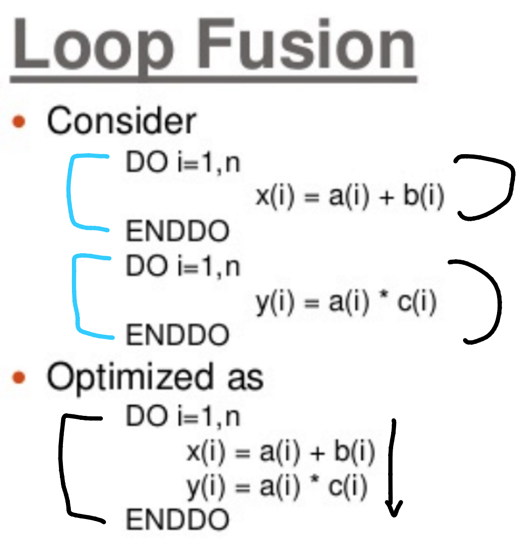
Loop Invariant Code Motion

- Consider

```
for(i=0; i<=N; i++)
{
        A[i]=F[i] + C*D;
        E=G[K];
}
```

- Optimized as

```
temp=C*D;
for(i=0; i<=N; i++)
{
        A[i]=F[i] + temp;
}
E=G[K];
```



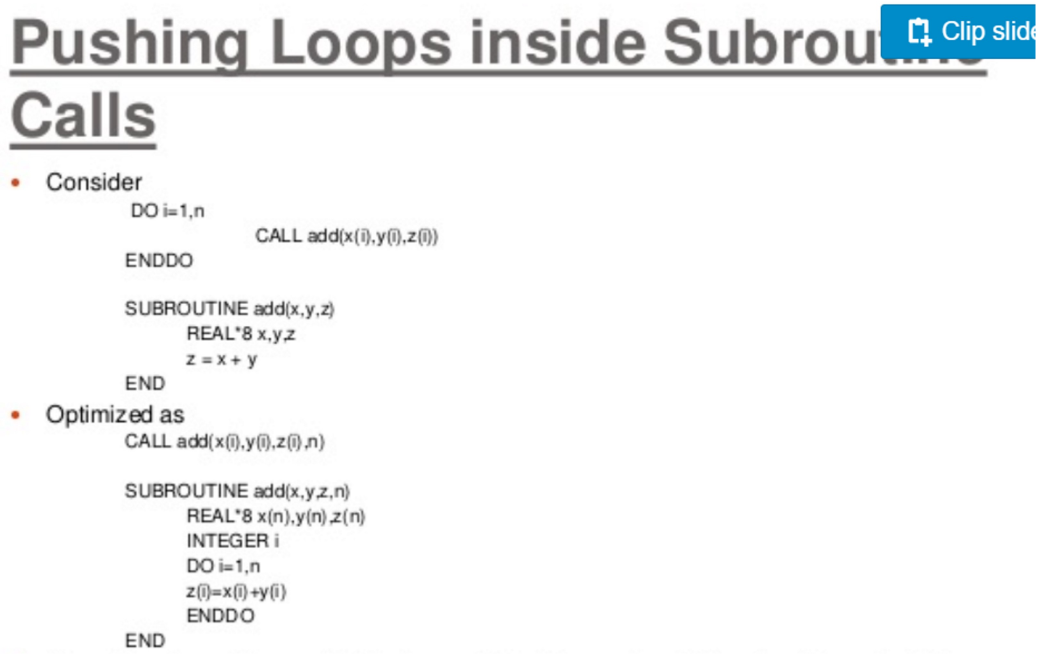
Loop Fusion

- Consider

```
DO i=1,n
          x(i) = a(i) + b(i)
ENDDO
DO i=1,n
          y(i) = a(i) * c(i)
ENDDO
```

- Optimized as

```
DO i=1,n
     x(i) = a(i) + b(i)
     y(i) = a(i) * c(i)
ENDDO
```



Pushing Loops inside Subroutine Calls

- Consider

```
 DO i=1,n
          CALL add(x(i),y(i),z(i))
ENDDO

SUBROUTINE add(x,y,z)
     REAL*8 x,y,z
     z = x + y
END
```

- Optimized as

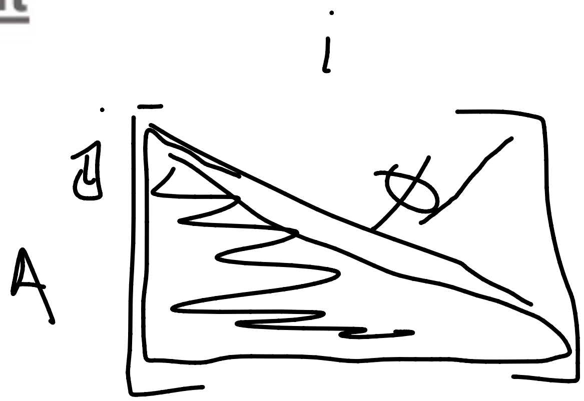
```
CALL add(x(i),y(i),z(i),n)

SUBROUTINE add(x,y,z,n)
     REAL*8 x(n),y(n),z(n)
     INTEGER i
     DO i=1,n
     z(i)=x(i)+y(i)
     ENDDO
END
```

Loop Index Dependent Conditionals

- Consider

```
DO I=1,N
    DO J=1,N
        IF( J.LT.I ) THEN
            A(J,I)=A(J,I) + B(J,I)*C
        ELSE
            A(J,I)=0.0
        ENDIF
    ENDDO
ENDDO
```



- Optimized as

```
DO I=1,N
    DO J=1,I-1
        A(J,I)=A(J,I) + B(J,I)*C
    ENDDO
    DO J=I,N
        A(J,I)=0.0
    ENDDO
ENDDO
```

Loop unrolling

- Consider

```
DO I=1,N
    A[I]=B[I]+C[I]
ENDDO
```

```
for(i=0;i<4;i++)
    A[i]=B[i]+C[i];
```

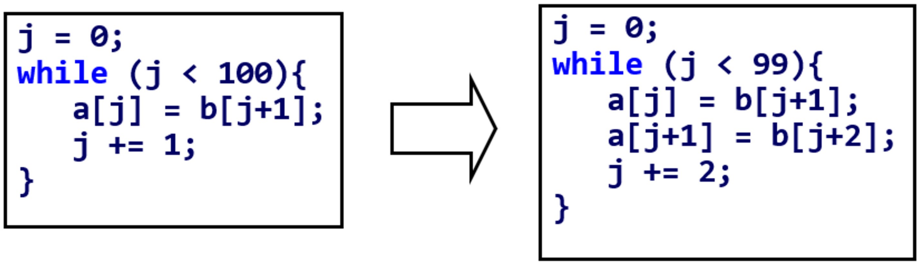
  - There are two loads for B[i] and C[i], one store for A[i], one addition B[i]+C[i], another addition for I=I+1 and a test i<=N. A total of six operations.

- Consider

```
DO I=1,N,2
    A(i)=B(i)+C(i)
    A(i+1)=B(i+1)+C(i+1)
ENDDO
```

```
for(i=1;i<N+1;i+=2{
```

  - four loads, two stores, three additions and one test for a total of ten.



```
j = 0;
while (j < 100){
   a[j] = b[j+1];
   j += 1;
}
```

```
j = 0;
while (j < 99){
   a[j] = b[j+1];
   a[j+1] = b[j+2];
   j += 2;
}
```

Replace Multiply by Shift

- A := A * 4;
  - Can be replaced by 2-bit left shift (signed/unsigned)
  - But must worry about overflow if language does
- A := A / 4;

Addition chains for multiplication

- If multiply is very slow (or on a machine with no multiply instruction like the original SPARC), decomposing a constant operand into sum of powers of two can be effective:

  X * 125 = x * 128 - x*4 + x

  - **two shifts**, **one subtract** and **one add**, which may be faster than one multiply

$13 \underset{\uparrow}{*} X$

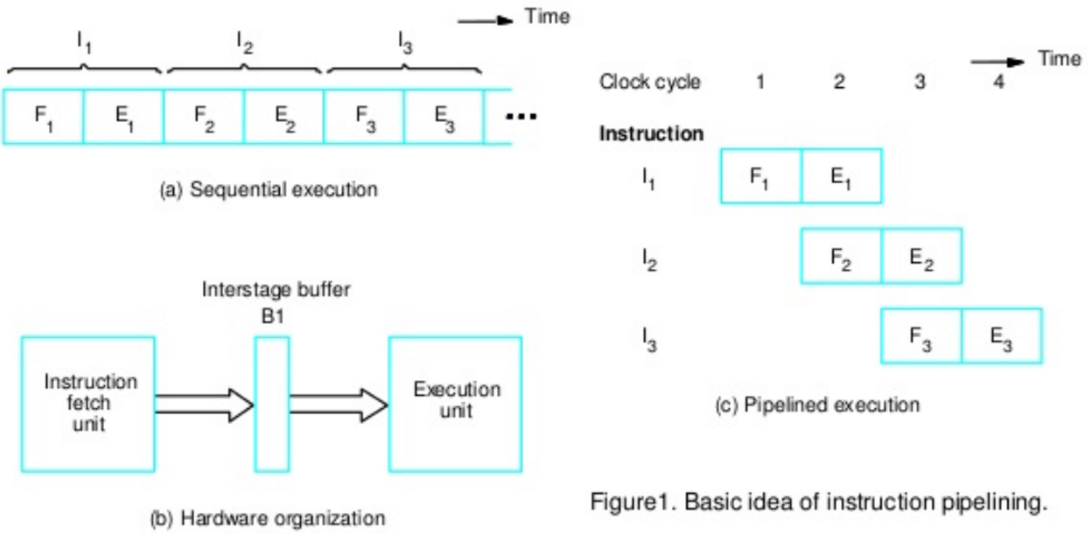
$X(8 + 4 + 1)$

$$\underbrace{(X<<3) + (X<<2) + X}_{\substack{\parallel \\ 13 * X}} \qquad \boxed{4t}$$

Instruction Pipelining in CPU

- ***Pipelining*** improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

Fetch + Execution



(a) Sequential execution

(b) Hardware organization

(c) Pipelined execution

Figure1. Basic idea of instruction pipelining.

APx1Z04Mt 14:15

Role of Cache Memory

HLO - High Level Optimization

Summary: gcc Optimization Levels

-g:
- Include debug information, no optimization

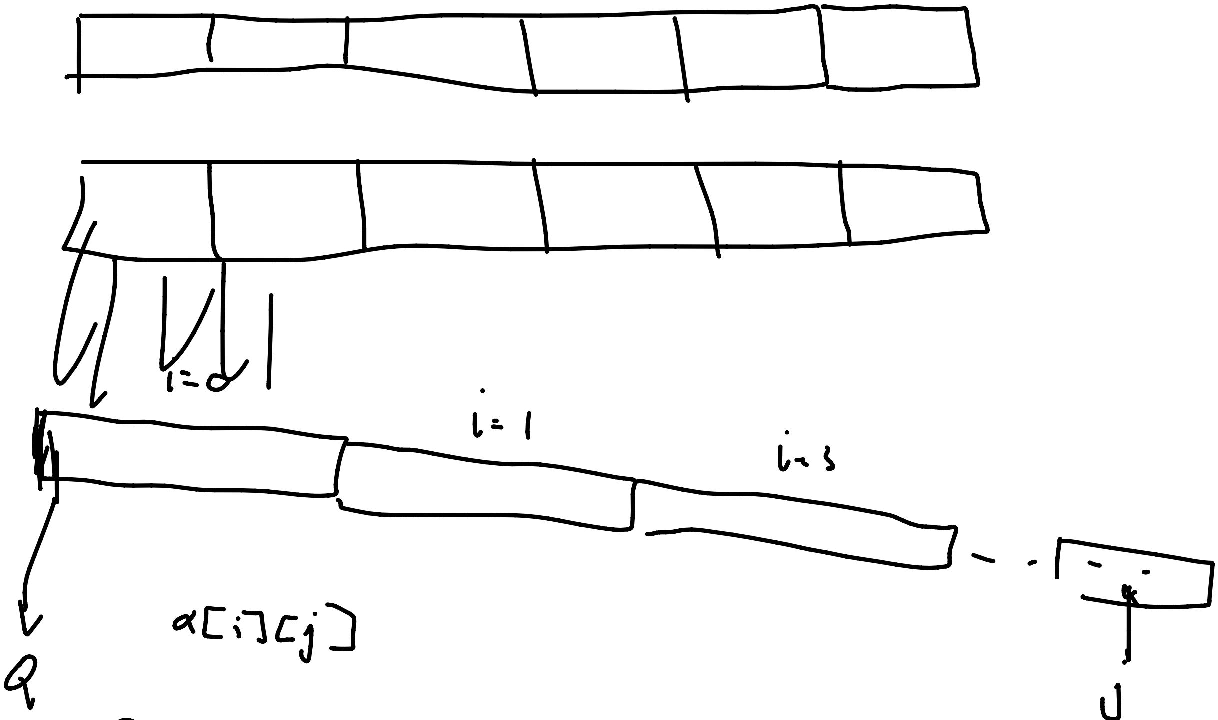
-O0:
- Default, no optimization

-O1:
- Do optimizations that don’t take too long
- CP, CF, CSE, DCE, LICM, inlining small functions

-O2:
- Take longer optimizing, more aggressive scheduling

-O3:
- Make space/speed trade-offs: loop unrolling, more inlining

-Os:
- Optimize program size



$a[i][j]$

$Q[i \cdot N+j]$