Code Tuning

Code tuning

- Used to write better code
- Needs better understanding of the programming language and its compiler
- Is equivalent of code optimization at Higher Language Level

Loop Fission and Fusion

Combining loops that operate over same range of values

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
a[i] = 1;
}
for (i = 0; i < 100; i++)
{
b[i] = 2;
}
```

After Loop Fusion

```
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
a[i] = 1;
b[i] = 2;
 }
```

Unswitching of loops

```
int i, w, x[1000], y[1000];
for (i = 0; i < 1000; i++)
{
x[i] = x[i] + y[i];
if (w)=1
y[i] = 0;
}
```

Improved Code After Unswitching

```
int i, w, x[1000], y[1000];
if (w)
{
for (i = 0; i < 1000; i++)
{
x[i] = x[i] + y[i];
y[i] = 0;
}}
Else {
for (i = 0; i < 1000; i++)
{
x[i] = x[i] + y[i];
}}
```

Loop Unrolling

```
for (int x = 0; x < 100; x++)
{
delete(x);
}
```

Improved code after unrolling

```
for (int x = 0; x < 100; x += 2)
{
delete(x);
delete(x+1);
}
```

Loop-invariant code

```
for (i = 0; i < n; ++i) {
x = y + z;
a[i] = 6 * i + x * x;
}
```

Improved code after taking out the loop-invariant code

```
x = y + z;
t1 = x * x;
for (i = 0; i < n; ++i) {
a[i] = 6 * i + t1;
}
```

Minimize work inside loops

```
For (i = 1;i<n/2;i++)
{…
}
```

Need to compute n/2 times in all iterations

Improved code after minimizing the work in loops

```
m = n/2;
For (i = 1;i< m;i++)
{…
}
```

Use of sentinel values

To find the number x in array a

```
While(i<n) and (x<>a[i])
{
i=i+1;
}
If (i<n)
Print number at position i
Else
Print number not present
```

Improved Code after using sentinel values

```
A[n+1]=x;
While(x<>a[i])
{
i=i+1;
}
if i==(n+1)
print number not present
else
print position of number i
```

Order condition testing in Switch Case & If-Else by frequency

```
Read(RegnNo)
Case Grade(student)
1: {......}
2: {......}
3: {......}
4: {......}
Endcase

Read(RegnNo)
Case Grade(student)
2: {......}
3: {......}
1: {......}
4: {......}
endcase
```

Common Sub expression Elimination

```
a = b * c + g;
d = b * c * d;
```

```
tmp = b * c;
a = tmp + g;
d = tmp * d;
```

Minimizing array references

If the same array element is repeatedly referred inside a loop, then move it outside the loop

```
For( a=0;a<5;a++)
{
for (b=0;b<10;b++)
{
Total[b]=total[b]*sum[a]
}
}
for( a=0;a<5;a++)
{
Sum_now=sum[a]
for (b=0;b<10;b++)
{
Total[b]=total[b]*sum_now;
}
}
```

Use Constants of Correct Type

Float x;
X=5; convert 5 to 5.0 and then
stores into x

int i;
i=3.14; convert 3.14 to 3 and store into
i

Precompute Results

```
For(i=0;i<100;i++)
{
Y=log(x)/log(2)
B=log(a)/log(2)
}
```

Improved Code after Precomputing

```
Twolog=log(2)
For(i=0;i<100;i++)
{
Y=log(x)/ Twolog
B=log(a)/ Twolog
```

Dead Code elimination

```
int xyz()
{
int a = 24;
int b = 25; /* Assignment to dead variable */
int c;
c = a- 2;
return c;
b = 24; /* Unreachable code */
return 0;
 }
```

Improved Code after eliminating Dead Code

```
int xyz()
{
int a = 24;
*/
int c;
c = a- 2;
return c;
}
```

Exploit Algebraic Identities

- Algebraic identities can be used to replace costlier operations by cheaper ones Whenever we need to find whether sqrt(x) <sqrt(y), we can use the algebraic identity which says x < y only when x < y. So it is enough to check if x < y in this case.
- not (A or B) is cheaper than not A and not B

Lazy computations

- n=x*x+2*y+z

```
if q>10 then
{ ......}
else q > n then
{.....}
else
{...}
```

Improved Code after Lazy computation

```
if q>10 then
{......}
else
{
n=x*x+2*y+z
if q > n then
{.....}
else
{.....}
```

Short circuiting and reordering

```
if (a>b) && (c>d) && (e>f)
{
.....
}
```

Using Locality of reference by avoiding cache misses

```
for i in 0..n
for j in 0..m
for k in 0..p
C[i][j] = C[i][j] + A[i][k] * B[k][j];

for i in 0..n
for k in 0..m
for j in 0..p
C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

- It is better to refer to several memory addresses that share the same row (spatial locality). By keeping the row number fixed, the second element changes more rapidly. In C and C++, this means the memory addresses are used more consecutively. Since j affects the column reference of both matrices C and B, it should be iterated in the innermost loop (this will fix the row iterators, i and k, while j moves across each column in the row). This will not change the mathematical result, but it improves efficiency. By switching the looping order for j and k, the speedup in large matrix multiplications becomes dramatic.