Code optimization: (optional phase, to improve IC.)

1. platform Dependent Techniques
2. platform Independent techniques

1. Techniques (Machine Dependent)

→ peephole optimization
→ Instruction level parallelism
→ Data level parallelism
→ Cache Optimization
→ Redundant Resources.

2. → Loop optimization (Independent)
   → constant folding
   → constant propagation
   → Common Subexpression Elimination.

A. Peephole optimization :

↓

performing optimizations on small parts of code, replacing sections with shorter & faster code, where the inputs are consistent.

→ 5 ways of implementating it

1. Redundant load/ store elimination

redundancy — duplication.

Eg: Input code:

```
x = y + 4
i = x
k = i
j = x * 6
```

optimized

```
x = y + 4
i = x
j = x * 6
```

2. constant folding.

$x = 3 * 3$

optimized code

$x = 9$.

3. Strength reduction

replacement of high execution time operator with low execution time.

→ $i = j * 2$

optimized

$i = j + j$

4. Null sequences:

unnecessary operators are removed.

Eg: i = 13 + 5

K = i + 10

optimized:

K = 28

5. Combining operators:

Input:

i = j * (10 + 8)

optimized i = j * 18

B. Instruction level parallelism:

It's an architecture, where multiple operations can be performed parallely in a particular process with it's own set of resources – registers, memory, identifiers etc...

Example:

1. load $R1 \leftarrow R2$
2. add $R3 \leftarrow R3$, "1"
3. add $R4 \leftarrow R4, R2$
4. add $R3 \leftarrow R3$ "1"
5. add $R4 \leftarrow R3, R2$
6. store $[R4] \leftarrow R0$

All these operations happen simultaneously.

Example - 2 :

Let the sequence of instructions be -

1. y1 = x1*1010
2. y2 = x2*1100
3. z1 = y1+0010
4. z2 = y2+0101
5. t1 = t1+1
6. p = q*1000
7. clr = clr+0010
8. r = r+0001

Sequential record of execution vs. Instruction-level Parallel record of execution -

CYCLE	OPERATION
1	y1 = x1*1010
2	nop
3	nop
4	y2 = x2*1100
5	nop
6	nop
7	z1 = y1+0010
8	z2 = y2+0101
9	t1 = t1+1
10	p = q*1000
11	clr = clr+0010
12	r = r+0001

Fig. a

CYCLE	INT ALU	INT ALU	FLOAT ALU	FLOAT ALU
1	t1 = t1+1	clr = clr+0010	y1 = x1*1010	y2 = x2*1100
2	r = r+0001		p = q*1000	
3	nop			
4	z1 = y1+0010	z2 = y2+0101		

Fig. b

C. Data level parallelism

Concurrent execution of same task on each multiple computing core.

Example:

Summing the contents of an array of size 'N':

→ Single core - system: one thread simply sum the elements

[0] .... [N-1]

→ Dual - core system:

However, thread A running on core 0, could sum elements

[0] ..... [N/2 - 1]

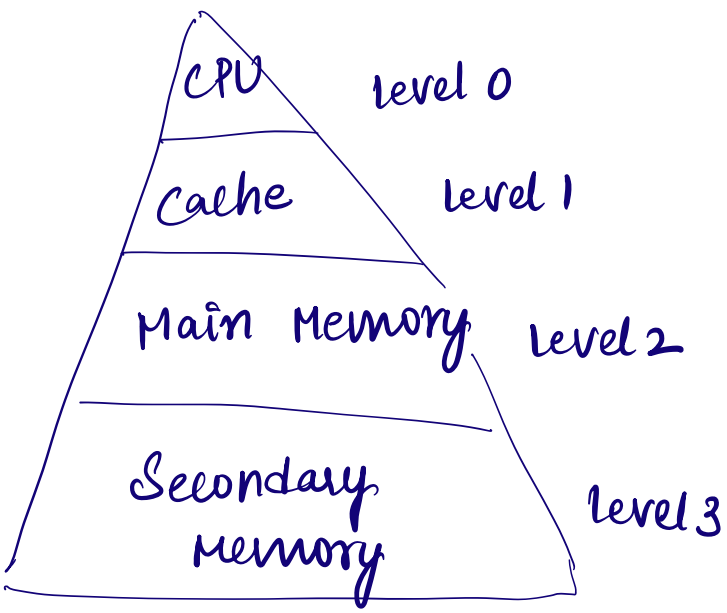
→ while thread B, running on core 1, could sum [N/2] .... [N-1].

So, 2 threads would be running in parallel on separate computing cores.

D. Cache optimization Techniques

Usually, in any device memories that are large, fast and affordable are preferred. But all these three can't be achieved at same time. Cost of memory depends on its speed and capacity.

With Hierarchical memory system, all these three can be achieved.



Cache is used in storing the frequently used data and instructions.

Usually it is expensive, the larger the cache memory, the higher the cost.

→ Optimization of Cache performance ensures that it is utilized in a very efficient manner to its full potential.

Average Memory Access Time (AMAT):

Helps in analyzing Cache memory and its performance. The lesser the AMAT, the better the performance is.

AMAT = Hit Ratio * Cache access time
+ Miss Ratio * Main Memory access time.

$$= (h * t_c) + (1-h) * (t_c + t_m)$$

E. Redundant resources:

Here all the redundant resources will be eliminated. we have partial redundancy elimination (PRE) that eliminates expressions that are redundant on some but not all paths.

2. Machine Independent Techniques:

A. Loop Optimization:

Process of increasing execution speed and reducing the overheads associated with loops.

we have few Loop optimization techni-ques.

1. Frequency Reduction (Code Motion)

Amount of code in loop is decreased. statement / expression, which can be moved outside the loop without affecting the semantics of program, is moved outside the loop.

Example:

```
while (j < 10)
{
    a = sin(x) / cos(x) + j;
    j++;
}
```

Optimized:

```
t = sin(x) / cos(x)
while (j < 10)
{
    a = t + j;
    j++;
}
```

2. Loop Unrolling:

Remove / reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and test instruction.

Eg:

```
for(int i=0; i<3; i++)
  Printf ("Hi\n");
```

optimized:

```
Printf ("Hi \n");
Printf ("Hi \n");
Printf ("Hi \n");
```

3. Loop Jamming:

combining the two or more loops in a single loop. Reduces the time taken to compile the many number of loops.

Eg:

```
for (int i=0; i<3; i++)
    a = i+6;
for (int i=0; i<3; i++)
    b = i+10;
```

optimized:

```
for (int i=0; i<3; i++)
{   a = i+6;
    b = i+10;
}
```

B. Constant folding:

eliminates expressions that calculates a value that already be determined before code executions.

Example:

```
int f(void)
{ return 3+5;
}
```

optimized

```
int f(void)
{
 return 8;
}
```

C. Constant propagation:

Process of replacing the constant value of variables in the expression.

Example: $Pi = \frac{22}{7}$

instead

$Pi = 3.14$

→ $a = 30$

$b = 20 - \frac{a}{2}$

Instead optimized code is

$a = 30$

$b = 20 - \frac{30}{2}$

Eg:

```
int x=5;
int y= x* 2;
int z= a[y];
```

Optimize:

```
int y =5 * 2;
int z= a[y];
```

D. Common Sub-Expression Elimination:

Expression or sub-expression that has been appeared again during computation has to be eliminated

Types:

1. Local CSEE : single basic blocks
2. Global CSEE : used for an entire procedure

Example:

```
a = 10;
b = a + 1 * 2;
c = a + 1 * 2;
d = c + a;
```

Optimized code:

```
a = 10;
b = a + 1 * 2;
d = b + a;
```

→

```
x = 11;
y = 11 * 2;
z = x * 2;
```

optimized

```
y = 11 * 2;
```