Compiler Optimizations

Types of Optimizations

- Peephole optimizations
- Local optimizations
- Loop optimizations
- Inter procedural or Whole program optimization
- Machine code optimization
- Language dependent and independent optimizations
- M/c dependent and independent optimizations

Factors affecting optimization

- The Architecture of the target CPU
  - #of CPU registers
  - RISC Vs. CISC
  - Pipelines
  - #of Functional units
- The Architecture of the Machine
  - Cache size
  - Memory Transfer rate
- Intended use of generated code
  - Debugging
  - General purpose use
  - Special purpose use
  - Embedded Systems

- Loop optimizations
- Data Flow optimization
- Dead code elimination
- Strength reduction

Strength Reduction

- A compiler uses methods to identify loops and recognize the characteristics of register values within those loops. For strength reduction, the compiler is interested in
- Loop invariants are essentially constants within a loop, but their value may change outside of the loop. Induction variables are changing by known amounts. The terms are relative to a particular loop.
- Strength reduction looks for expressions involving a loop invariant and an induction variable. Some of those expressions can be simplified. For example, the multiplication of loop invariant c and induction variable I can be replaced with successive weaker additions

- c = 8;

```
 for (i = 0; i < N; i++)
   {     y[i] = c * i;
   }
```

- c = 8; k = 0;

```
for (i = 0; i < N; i++)
   {     y[i] = k;
         k = k + c;
   }
```

Dead Code Elimination

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

```c
int main(void) {
  int a = 5;
  int b = 6;
  int c;
  c = a * (b >> 1);
  if (0) {   /* DEBUG */
    printf("%d\n", c);
  }
  return c;
}
```

Common Sub-Expression Elimination

- In the following code:

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

it may be worth transforming the code to:

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

if the time cost of storing and retrieving "tmp" outweighs the cost of calculating "b * c" an extra time.

- **Constant folding** is the process of simplifying constant expressions at compile time.
- i = 320 * 200 * 32; compilers would not actually generate 2MUL+ Store but replace with value compile time (in this case, 2,048,000), usually in the intermediate representation (IR) tree.
- Constant folding can be done in a compiler's front end on the IR tree that represents the high-level source language, before it is translated into three-address code, or in the back end, as an adjunct to constant propagation.

- **Constant propagation** is the process of substituting the values of known constants in expressions at compile time.
- int x = 14;
  int y = 7 - x / 2;
  return y * (28 / x + 2);

Propagating x yields:

- int x = 14;
  int y = 7 - 14 / 2;
  return y * (28 / 14 + 2);

Continuing Propagating

int x = 14; int y = 0; return 0;

Loop Optimizations

- **Loop fission** (or **loop distribution**) is a compiler optimization technique attempting to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. The goal is to break down large loop body into smaller ones to achieve better utilization of locality of reference. It is the reverse action to loop fusion

Original loop	After loop fission

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

Loop Fusion

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

is equivalent to:

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

Loop Interchange

```
for i from 0 to 10
  for j from 0 to 20
    a[i,j] = i + j
```

loop interchange would result in:

```
for j from 0 to 20
  for i from 0 to 10
    a[i,j] = i + j
```

Loop Inversion

```
int i, a[100];
i = 0;
while (i < 100) {
  a[i] = 0;
  i++;
}
```

is equivalent to:

```
int i, a[100];
i = 0;
if (i < 100) {
  do {
    a[i] = 0;
    i++;
  } while (i < 100);
}
```

Loop Invariant Code Motion

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

The calculations `x = y + z` and `x * x` can be moved outside the loop since within they are loop invariant—this:

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

Loop Peeling

```
int p = 10;
for (int i=0; i<10; ++i)
{
  y[i] = x[i] + x[p];
  p = i;
}
```

Notice that `p = 10` only for the first iteration, and for all other iterations, `p = i - 1`.

After peeling the first iteration, the code would look like this:

```
y[0] = x[0] + x[10];
for (int i=1; i<10; ++i)
{
  y[i] = x[i] + x[i-1];
}
```

- A procedure in a computer program is to delete 100 items from a collection. This is normally accomplished by means of a *for*-loop which calls the function *delete(item_number)*. If this part of the program is to be optimized, and the overhead of the loop requires significant resources compared to those for *delete(x)* loop, unwinding can be used to speed it up.

Normal loop

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

After loop unrolling

```
int x;
for (x = 0; x < 100; x+=5)
{
    delete(x);
    delete(x+1);
    delete(x+2);
    delete(x+3);
    delete(x+4);
}
```