C Optimization Techniques

Team Emertxe

Optimization ?

- Program optimization or software optimization is the process of modifying a software system to make some aspect of it work more efficiently or use fewer resources.
- Optimization is a process of improving efficiency of a program in time (speed) or Space (size).

Categories of Optimization ?

- Space Optimization

```
1  /* Faster exection */
2  main()
3  {
4          ......
5
6          XXXXX
7          YYYYY
8          ZZZZZ
9
10         ......
11
12         XXXXX
13         YYYYY
14         ZZZZZ
15
16         ......
17
18         XXXXX
19         YYYYY
20         ZZZZZ
21 }
```

Time Optimization

```
1  /* Smaller Size */
2  main()
3  {
4          .......
5
6          function();
7
8          .......
9
10         function();
11
12         .......
13
14         function();
15 }
16
17 void function()
18 {
19         XXXXX
20         YYYYY
21         ZZZZZ
22 }
```

EMERTXE

General Phenomenon

Code size	speed
Bigger	Greater
Smaller	Lesser

Note : However this may not always true.

Compiler
Code Optimization

- Compilers can be designed to provide code optimization.
- Users should only focus on optimization not provided by the compiler such as choosing a faster and/or less memory intensive algorithm

Optimization: Methods

- Choice of Compiler
- Compiler settings
- Programming Algorithm and Techniques
- Rewriting program in Assembly
- Code profilers
- Disassemble & code analysis

Programming Algorithms & Techniques

- Data Handling
- Flow control
- Other Handling

Data Handling

- RAM Usage
- Data Types Usage
- Avoid Type Coversion
- Unsigned Advantage
- Float & Double
- Constant & Volatile
- Data Alignment – Arrangement & Packing
- Pass by Reference

RAM Usage

- Four main Component of program memory

  Text
  Data
  Heap
  Stack

- The reduction in one component will enable the increase in the others.

- Unlike stack and heap, which are dynamic in nature, global data is fixed in size.

Data Usage Types

The use of correct data type is important in a recursive calculation or large array processing.

The extra size, which is not required, is taking up much space and processing time.

Example:

If your variable range is varying in between 0 to 200 then you should take variable type of unsigned char.

Avoid Type Conversion

- Programmers should plan to use the same type of variables for processing. Type conversion must be avoided.

- Otherwise, precious cycles will be waste to convert one type to another (here Signed and Unsigned variables are considered as different types.)

Usage: Signed & Unsigned

In summary,

- **use unsigned types for:**

  Division and remainders

  Loop counters

  Array indexing

- **Use signed types for:**

  Integer-to-floating-point conversion

Floats & Doubles

- Maximum value of Float = **0x7F7F FFFF**
- Maximum Value of Double = **0x7F7F FFFF FFFF FFFF**
- To avoid the unnecessary type conversion or confusion, programmers can assign the letter 'f' following the numeric value.

  x = y + 0.2f;

Const & Volatile

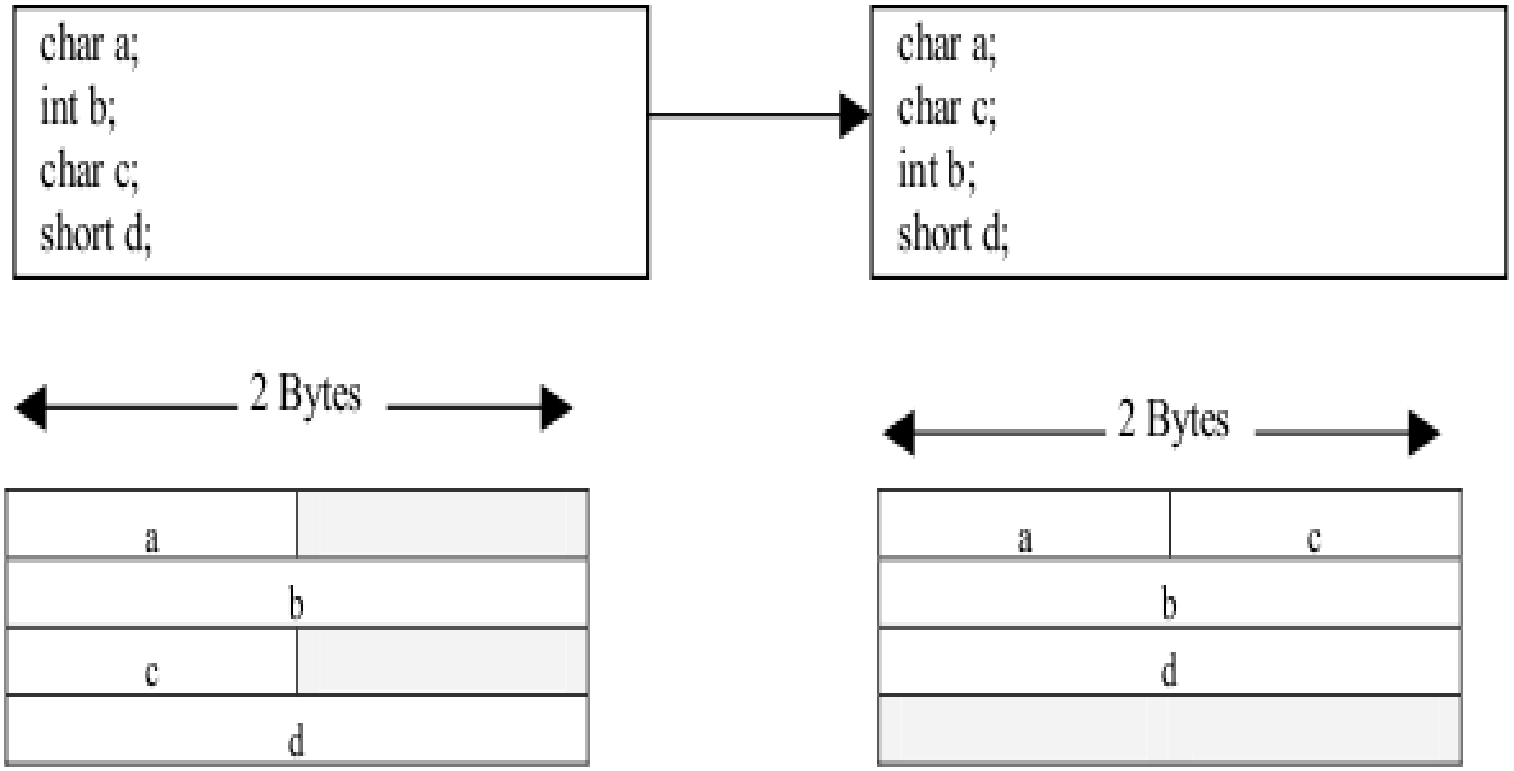
- The “const” keyword is to define the data as a constant, which will allocate it in the ROM space.

- Otherwise a RAM space will also be reserved for this data. This is unnecessary as the constant data is supposed to be read-only.

Data Alignment : Arrangment and Packing

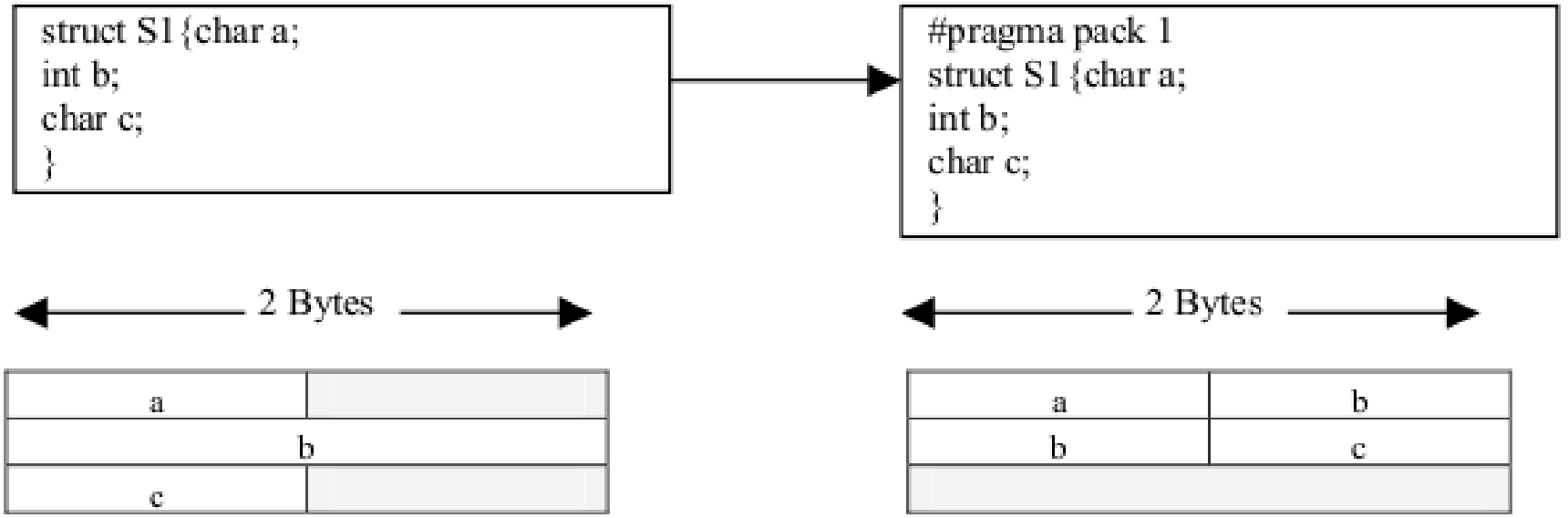
- The declaration of the component in a structure will determine how the components are being stored.
- Due to the Memory alignment, it is possible to have dummy area within the structure. It is advised to place all similar size variables in the same group.

Data Arrangment



Structure Padding

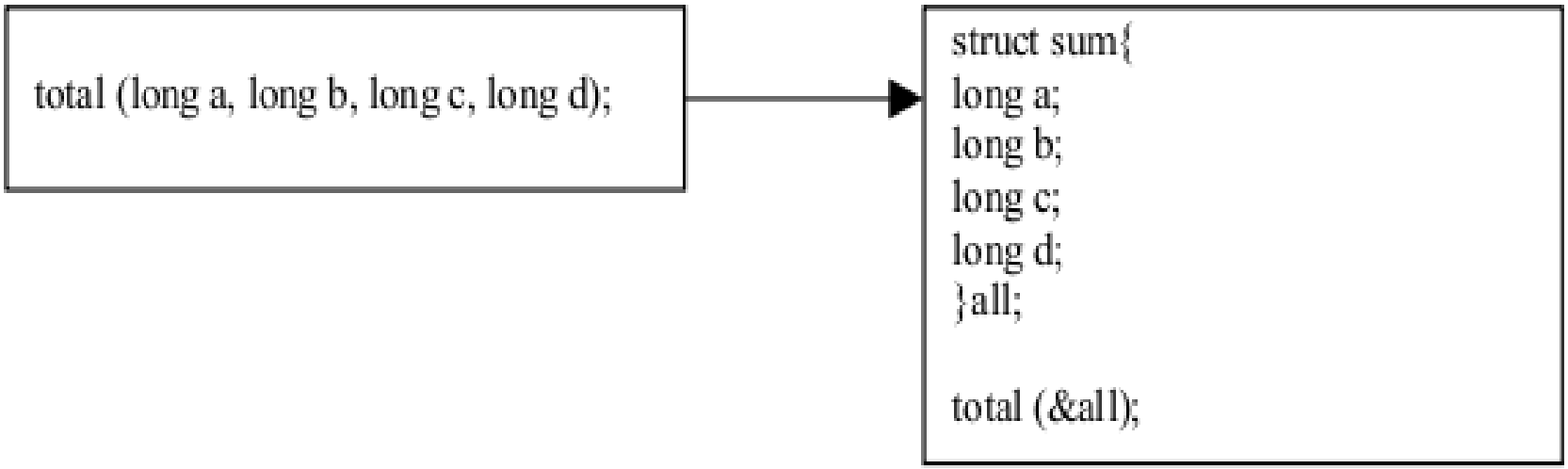
- As the structure is packed, integer b will not be aligned. This will improve the RAM size but operational speed will be degraded, as the access of ‘b’ will take up two cycles.



Pass by Reference

- Larger numbers of parameters may be costly due to the number of pushing and popping actions on each function call.

- It is more efficient to pass structure reference as parameters to reduce this overhead.



Arrays & Structure: Initialization

- Illustration

```
int a[3][3][3];
int b[3][3][3];
...
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    for(k=0;k<3;k++)
      b[i][j][k] = a[i][j][k];
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    for(k=0;k<3;k++)
      a[i][j][k] = 0;

for(x=0;x<100;x++)
    printf("%d\n",(int)(sqrt(x)));
```

```
typedef struct {
      int element[3][3][3];
} Three3DType;

Three3DType a,b;
...
b = a;

memset(a,0,sizeof(a));
```

Return Value

- The return value of a function will be stored in a register. If this return data has no intended usage, time and space are wasted in storing this information.
- Programmer should define the function as “void” to minimize the extra handling in the function.

Flow Control

- Use of switch statement
- Inline function
- Loop unrolling
- Loop Hoisting
- Loop Overhead

Switch & Non-Contiguous Case Expressions

- Use if-else statements in place of switch statements that have noncontiguous case expressions.

- If the case expressions are contiguous or nearly contiguous integer values, most compilers translate the switch statement as a jump table instead of a comparison chain.

Inline Function

- The technique will cause the compiler to replace all calls to the function, with a copy of the function’s code.
- This will eliminate the runtime overhead associated with the function call.
- This is most effective if the function is called frequently, but contains only a few lines of code.
- In-lining large functions will make the executable too large.

Loop Unrolling

- Loop unrolling tries to get rid of the checks completely or to reduce the number of checks.

- If you know a loop is only performed a certain number of times, or if you know the number of times it will be repeated is a multiple of a constant you can unroll this loop.

```
1 // old loop
2 for(int i=0; i<3; i++)
3 {
4         color_map[n+i] = i;
5 }
```

```
1 // unrolled version
2 int i = 0;
3 colormap[n+i] = i;
4 i++;
5 colormap[n+i] = i;
6 i++;
7 colormap[n+i] = i;
```

EMERTXE

Loop Hoisting

- Moving computations outside loops
- Saves computing time

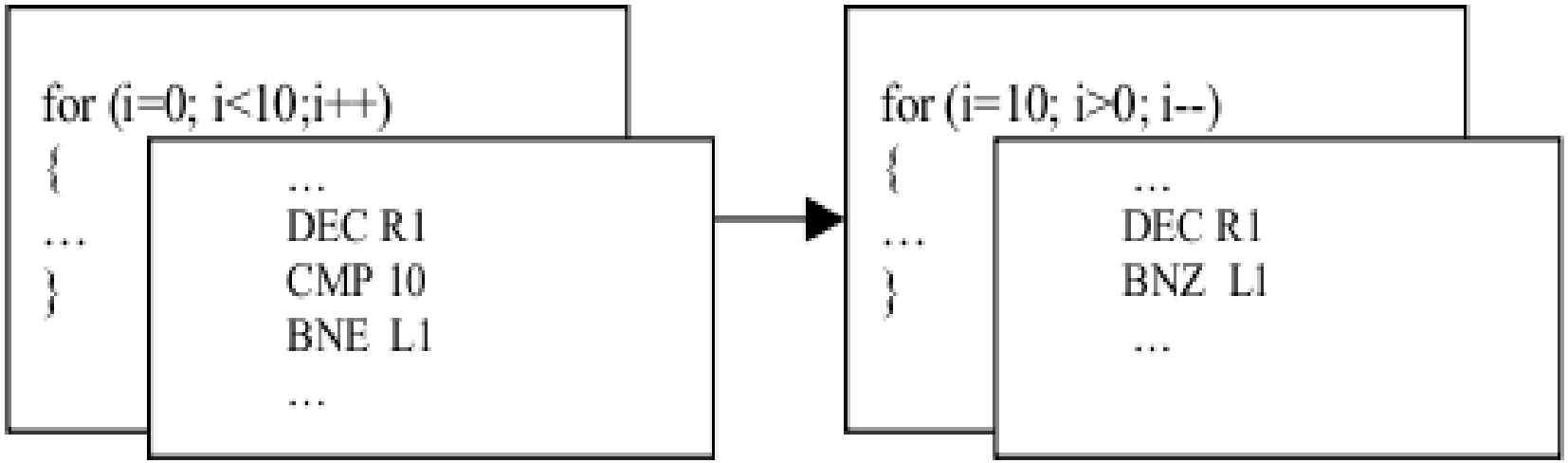
```
1  /* Old code */
2  for (i....)
3  {
4          if (CONSTANT0)
5          {
6                  DoWork0(i);
7          }
8          else
9          {
10                 DoWork1(i);
11         }
12 }
```

```
1  /* Optimized Code */
2  if (CONSTANT0) {
3          for (i....)
4          {
5                  DoWork0(i);
6          }
7  }
8  else {
9          for (i....)
10         {
11                 DoWork1(i);
12         }
13 }
```

ΣMERTXE

Loop Overhead

- The MCU have a conditional branch mechanism that works well when counting down from positive number to zero.
- It is easier to detect “Zero” (Branch non zero – single instruction) than a “predefined number” (Compare and perform jump – two instructions)



Other Handling

- Use of operators
- Use formula
- Inline Assembly
- Fixed point & Floating point

Use of operators

```
1  /* Old Code */
2  int i, j;
3  if (i > CONSTANT)
4  {
5          j = 1;
6  }
7  else
8  {
9          j = 0;
10 }
```

```
1 /* Optimized Code */
2
3 int i, j;
4 j = (i > CONSTANT) ? 1 : 0;
```

EMERTXE

Replacing: Integer Division With Multiplication

- Replace integer division with multiplication when there are multiple divisions in an expression.
- This is possible only if no overflow will occur during the computation of the product. The possibility of an overflow can be determined by considering the possible ranges of the divisors.
- Integer division is the slowest of all integer arithmetic operations.

Example

Avoid code that uses two integer divisions:

```
int i, j, k, m;

m = i / j / k;
```

Instead, replace one of the integer divisions with the appropriate multiplication:

```
m = i / (j * k);
```

Use finite differences to Avoid multiplies

```
for (i = 0 ; i < 10;  i++)
    printf(" %d\n", i*10);
```

```
for (i = 0 ; i < 100;  i+=10)
    printf(" %d\n", i);
```

Module by &

```
x = y %32 ;
```

```
x = y &31;
```

Extracting common subexpressions

- Manually extract common subexpressions where C compilers may be unable to extract them from floating-point expressions due to the guarantee against reordering of such expressions in the ANSI standard.

Manual Optimization

Example 1

Manual Optimization

Example 2

Use Formula

- Use formula instead of loops
- What is the formula for sum of naturals numbers?

Example

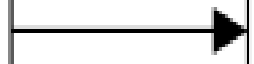
```
n=100;
for (x=0, y=1; y<=n; y++)
        x +=y;
```

```
n=100;
x = n* (n >>1);  //n^2 /2
```

Use Formula

Example

```
If ( a==b && c==d && e==f )
  {...}
```



```
If( (( a-b) | (c-d) | (e-f)) ==0 )
  {...}
```

Example

```
if( (x==1) || (x==2) || (x==4) || (x==8)
|| ... )
```

```
if( x&(x-1)==0 &&x!=0 )
```

Inline Assembly

In case you want write code is best optimized assembly code,

```
__asm__ ("movl %eax, %ebx\n\t"
         "movl $56, %esi\n\t"
         "movl %ecx, $label(%edx,%ebx,$4)\n\t"
         "movb %ah, (%ebx)");
```

THANK YOU