General Code Optimization Techniques
Wesley Myers
wesley.y.myers@gmail.com

General Code Optimization Techniques

Introduction

Normally, programmers do not always think of hand optimizing code. Most programmers leave it to the GCC compiler to optimize their code, but sometimes the programmer must in fact think of their own optimizations. Embedded systems face this problem every day. Size can be an issue when there is very little memory. Sometimes time becomes an issue when the embedded system is real time and must meet a deadline. Programmers must become creative and crafty to optimize their code. This paper discusses various techniques to optimize code in space and time.

Common Sub Expression Elimination

Common sub expression elimination is an easy way to simplify code and save space. The best way to detect this is by seeing where the same code elements arise multiple times. The main benefit of this optimization is that there is less code to be accommodated in terms of memory. Besides eliminating code that may do the same thing, large chunks of code can be turned into functions, thereby making the code easier to read. In addition, this method will make bug fixes easier. In Figure 1, code size is optimized by eliminating the repeating code, however time is lost in that a jump must occur to go to the code and a jump must occur to return to original calling function. To the left is the main body of code where the same body of code appears multiple times, labeled "Function X." If we pull out that function, as shown to the right, the total memory required is reduced, however note that the execution takes longer because we still go over the same code, but now we have added jumps, as shown by the arrows.

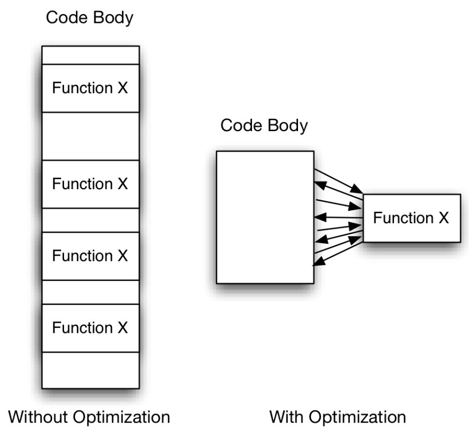


Figure 1 – Common Sub Expression Optimization Example

Dead-Code Elimination

Compilers have a hard time trying to eliminate dead code. By giving the compiler free reign of this, the compiler could "optimize" out sections of code unintentionally. Thus it is mainly up to the human to detect these problems. This optimization targets memory in that less code is needed to be stored in memory.

Many times these kinds of functions occur for testing. Figure 2 illustrates a simple example of some standard code that a tester may write. Here, debug is essentially a flag that tells debug statements to print.

```
debug = 0;

if(debug)
{
     printf("testing…");
}
```

Figure 2 – Testing Code

The problem with this is that it generates a lot of dead code while debug is equal to zero. The way to eliminate this is by using the preprocessor declaration #ifdef to eliminate unused code, as shown in Figure 3. So instead you have the following

```
#ifdef debug

printf("testing…");

#endif
```

Figure 3 – Testing Code with Optimization

If debug is not defined, then that line of code won't even be included after compilation. The preprocessor will go through and check for these declarations. #IFDEF basically looks at what is immediately after it, in this case debug. It then goes to check if debug is defined (if #define debug is somewhere up above). If it is, then the printf statement will be included in the final binary. If not, then that line of code will not be included. #ENDIF is basically the closing bracket of the #IFDEF. Anything between the two may or may not exist depending on the definition.

Loop Unrolling

Loop unrolling can be done when the code does multiple iterations of work in each cycle of a loop. This technique is used to save a jump back to the start of the loop. Making the jump can cost time if the array size is considerably large. Consider a loop that must iterate 10 times and executes one line of code, as shown in Figure 4. This often happens when using or manipulating an array. Consider that the execution takes one unit of time and that a jump takes only one unit of time. The result after compilation is that the total time taken is 20 units of time. If instead two lines of code were executed such that the loop executed 5 times instead, as shown in Figure 5, the total time taken is 15 units of time and we have shaved off 25% of the original time!

```
int myArray[10];

//somewhere array gets data…

int sum = 0, i;

for(i = 0; i < 10; i++)
{
     sum += myArray[i];
}
```

Figure 4 – Typical Array Iteration Code

```
int myArray[10];

//somewhere array gets data…

int sum = 0, i;

for(i = 0; i < 5; i+=2)
{
     sum += myArray[i];
     sum += myArray[i+1];
}
```

Figure 5 – Array Iteration Code with Loop Unrolling

__inlining of Functions

__inlining of functions is best used when there is a very small and simple function that takes inputs. When a function with argument is called in the ARM processor, the registers to be used in the new function inputs must be moved into input registers for the called function. This will be explained further in this section. __inline removes the need for these operations and makes the function and inserts it into the main body of code. Doing this can help save you clock cycles, but it can also possibly increase your code size. This optimization is the opposite of common sub expression elimination.

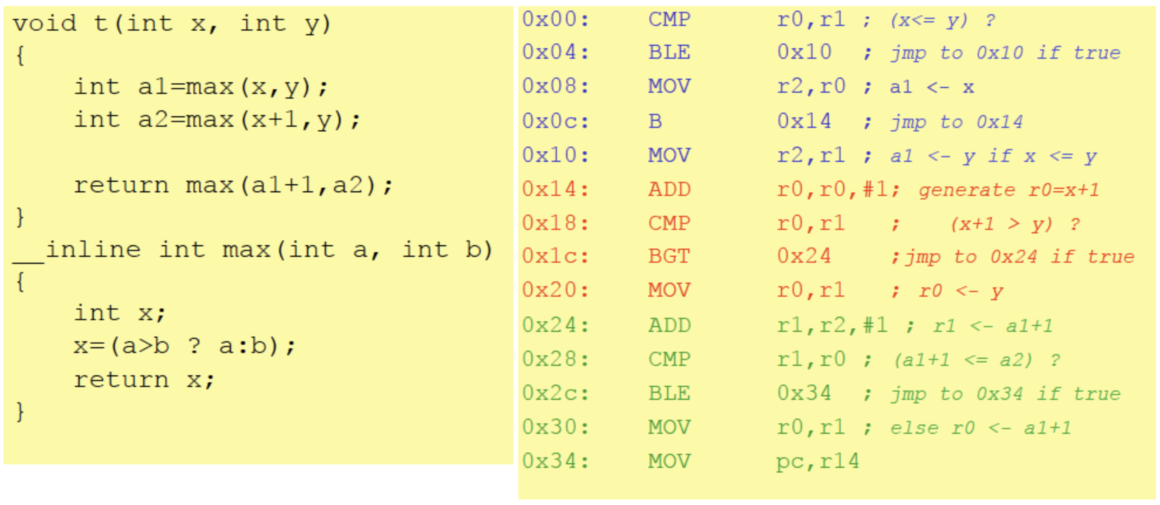
```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}
int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

```
max
$a
0x00:    CMP      r0,r1;  (x > y) ?
0x04:    BGT      0x0c;   return if (x > y)
0x08:    MOV      r0,r1;  else r0 <- y
0x0c:    MOV      pc,r14  return
t
0x10:    STMFD    r13!,{r4,r14}; save registers
0x14:    MOV      r2,r0;            r2 <- x
0x18:    MOV      r3,r1;            r3 <- y
0x1c:    MOV      r1,r3;            r1 <- y
0x20:    MOV      r0,r2;            r0 <- x
0x24:    BL       max  ;            r0 <- max(x,y)
0x28:    MOV      r4,r0;            r4 <- a1
0x2c:    MOV      r1,r3;            r1 <- y
0x30:    ADD      r0,r2,#1;         r0 <- x+1
0x34:    BL       max  ;           r0 <- max(x+1,y)
0x38:    MOV      r1,r0 ;          r1 <- a2
0x3c:    ADD      r0,r4,#1 ;        r0 <- a1+1
0x40:    LDMFD    r13!,{r4,r14} ; restore
0x44:    B        max  ;
```

Figure 6 – Code without inline Optimization [2]

In Figure 6, what we see is assembly code, essentially what is generated by the compiler. What we see is a function called "t" (0x10) that has two input variables, x and y. It calls another function called "max" (0x0) with two input variables, a and b. Variable x is stored in r0 and variable y is stored in r1. R0 typically holds the first variable, r1 typically holds the second variable, and so on and so on. In order to call "max," r0 must have x and r1 must have y, so they are loaded into the registers. BL means that the code branches to the "max" function, so we move to 0x0 and start executing. "Max" does its thing and we return back to the main body of code. Essentially this saving process happens again as we call "max" again. One can see that this is a bit tedious and uses up cycles. By inlining the function, we can eliminate all these register saves.



```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}
__inline int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

```
0x00:    CMP     r0,r1 ; (x<= y) ?
0x04:    BLE     0x10  ; jmp to 0x10 if true
0x08:    MOV     r2,r0 ; a1 <- x
0x0c:    B       0x14  ; jmp to 0x14
0x10:    MOV     r2,r1 ; a1 <- y if x <= y
0x14:    ADD     r0,r0,#1; generate r0=x+1
0x18:    CMP     r0,r1   ;    (x+1 > y) ?
0x1c:    BGT     0x24    ;jmp to 0x24 if true
0x20:    MOV     r0,r1   ; r0 <- y
0x24:    ADD     r1,r2,#1 ; r1 <- a1+1
0x28:    CMP     r1,r0 ; (a1+1 <= a2) ?
0x2c:    BLE     0x34  ; jmp to 0x34 if true
0x30:    MOV     r0,r1 ; else r0 <- a1+1
0x34:    MOV     pc,r14
```

Figure 7 – Optimization with __inline [2]

In Figure 7, __inline is implemented for the max function. Essentially the code for max is now in the main body of "t" (0x0). What this means is that we have saved the overhead of saving the variables into registers.

Summary

With a few tricks, it can be really simple to optimize code and gain extra clock cycles. With great power comes great responsibility. Sometimes optimizing too much can result in corner cases that can lead to your system doing strange things. Be cognizant that optimization almost always involves a space versus time tradeoff. Optimizing for time can increase the code size, and thus one must be aware of how an optimization may affect the program such as optimizing for time may result in the program being too large to fit in memory.

Bibliography

[1] Narasimhan, Priya. "Code Optimization." *18-349 Embedded Real-Time Systems*. Carnegie Mellon University, Aug 2011. Web. Jun 2012. <https://www.ece.cmu.edu/~ee349/docs/06_CodeOptimization_handout.pdf>.