Compiler Optimizations

Assignment -1

Kartik Srinivas - ES20BTECH11015

A Course Homework Assignment

Indian Institute of Technology
Hyderabad

April 17, 2022

1 GCC and LLVM Common Optimization Options

The main principle of any compiler is that it first processes the code, brings it into an intermediate representation(called IR)and then performs optimizations. After this the ”backend” of the compiler reads this code and transforms it into assembly code specific to the architecture of the CPU it is going to be run on.

Both GCC and LLVM provide frontends for several languages. Their compilers for C are called gcc and clang respectively. Their manual pages that can be accessed via the terminal by using the command :- ”man gcc/clang”.

The front ends offer various common flags for optimization, for example:

a) -O1 (Level 1 Optimizations)

b) -O2

c) -O3

d) -ffastmath (Optimizing calculations for floating point calculations)

Both gcc and clang may not generate the same assembly code under these optimizations(even for the same architecture and language!)

2 GCC and LLVM Frontends

2.1 Frontend - Language Support

2.1.1 GCC

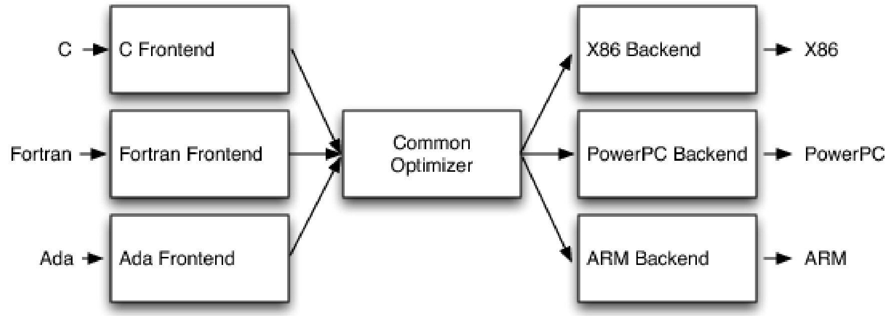
GCC distribution contains front ends for C (gcc), C++ (g++), Objective C, Fortran, Ada (GNAT), Go, and D.

2.1.2 LLVM

The languages supported are..clang (C and C++),llgo (Go), kaleidoscope (Haskell),flang (Fortran) ,dragonegg (LLVM backend for GCC),rust,emscripten (Javascript),rubinius (Ruby),il-wasm (CIL (C))

2.2 Extra Stuff I found out about:-

2.2.1 The ”end” of the frontend - the Intermediate Representation(IR)



The Intermediate representation (IR) for the Gcc compiler can be seen after using the ”-fdump-tree” flag and we can see the optimized code after using ”fdump-tree-optimized” flag. High level Optimizations are carried out in these intermediate forms.

The original code for this experiment:-

```
#include<stdio.h>
#define N 100
/*My name is Kartik !*/
int fibonacci(int n){
  if(n<2)
  {
    return n;
  }
  else return fibonacci(n-1) + fibonacci(n-2);
}

        int main()
        {
               int c,d;
               c = 10;
               d = fibonacci(10);
               printf("d = %d",d);
        }
```

Compilation:-

```
gcc example_es15.c -fdump-tree-original -O2 -fdump-tree-optimized
```

Original IR :-

```
;; Function fibonacci (null)
;; enabled by -tree-original
{
  if (n <= 1)
    {
      return n;
    }
  else
    {
      return fibonacci (n + -1) + fibonacci (n + -2); //notice how the notation
    has changed!
```

```
    }
}
```

Optimized IR :- This is an O2 optimization of the fibonacii function (this has been explained further under O2 optimizations later in this file) The IR has clearly completely changed.

```

fibonacci (int n)
{
  int _2;
  int _3;
  unsigned int _4;
  int prephitmp_7;
  int add_acc_8;
  unsigned int _9;
  int _10;
  int _11;
  unsigned int _12;
  int _14;
  int _16;
  int add_acc_17;
  unsigned int _18;
  int _19;
  unsigned int _20;
  int _21;

  <bb 2> [local count: 236223200]:
  if (n_6(D) <= 1)
    goto <bb 3>; [22.00%]
  else
    goto <bb 4>; [78.00%]

  <bb 3> [local count: 236223201]:
  # prephitmp_7 = PHI <_10(5), n_6(D)(2)>
  return prephitmp_7;

  <bb 4> [local count: 837518624]:
  # n_15 = PHI <n_6(D)(2), _3(4)>
  # add_acc_17 = PHI <0(2), add_acc_8(4)>
  _12 = (unsigned int) n_15;
  _20 = _12 + 4294967295;
  _21 = (int) _20;
  _2 = fibonacci (_21);
  _3 = n_15 + -2;
  add_acc_8 = _2 + add_acc_17;
  if (_3 <= 1)
    goto <bb 5>; [22.00%]
  else
    goto <bb 4>; [78.00%]

  <bb 5> [local count: 184254096]:
  _19 = n_6(D) + -2;
  _18 = (unsigned int) n_6(D);
  _9 = _18 + 4294967294;
  _4 = _9 >> 1;
```

```
  _14 = (int) _4;
  _11 = _14 * -2;
  _16 = _11 + _19;
  _10 = add_acc_8 + _16;
  goto <bb 3>; [100.00%]

}
```

LLVM IR:-

As you can see the LLVM IR is far shorter and more readable than the GCC IR Note that the "tail call optimization" which we will mention under O2 optimizations is probably happening here as indicated in line 12.

```
1
2 define dso_local i32 @fibonacci(i32 noundef %0) local_unnamed_addr #0 !dbg !7 {
3   call void @llvm.dbg.value(metadata i32 %0, metadata !13, metadata !DIExpression
      ()), !dbg !14
4   %2 = icmp slt i32 %0, 2, !dbg !15
5   br i1 %2, label %11, label %3, !dbg !17
6
7 3:                                                ; preds = %1, %3
8   %4 = phi i32 [ %8, %3 ], [ %0, %1 ]
9   %5 = phi i32 [ %9, %3 ], [ 0, %1 ]
10  call void @llvm.dbg.value(metadata i32 %4, metadata !13, metadata !DIExpression
      ()), !dbg !14
11  %6 = add nsw i32 %4, -1, !dbg !18
12  //is this a tail call optimization ?
13  %7 = tail call i32 @fibonacci(i32 noundef %6), !dbg !19
14  %8 = add nsw i32 %4, -2, !dbg !20
15  %9 = add nsw i32 %7, %5, !dbg !21
16  call void @llvm.dbg.value(metadata i32 %8, metadata !13, metadata !DIExpression
      ()), !dbg !14
17  %10 = icmp ult i32 %4, 4, !dbg !15
18  br i1 %10, label %11, label %3, !dbg !17
19
20 11:                                               ; preds = %3, %1
21  %12 = phi i32 [ 0, %1 ], [ %9, %3 ]
22  %13 = phi i32 [ %0, %1 ], [ %8, %3 ]
23  %14 = add nsw i32 %13, %12, !dbg !21
24  ret i32 %14, !dbg !22
25 }
```

3 GCC and Clang Backends

We will try to see the ARM and the X86 architectures assembly code generated by gcc and clang for the following code:-

```
#include<stdio.h>
#define N 100
/*My name is Kartik !*/
int fibonacci(int n){
  if(n<2)
  {
```

```
    return n;
  }
  else return fibonacci(n-1) + fibonacci(n-2);
}

        int main()
        {
               int c,d;
               c = 10;
               d = fibonacci(10);
               printf("d = %d",d);
        }
```

3.1 X86

The generated assembly for the X86 for clang and gcc have less difference , the naming conventions for the labels are different and the exit of the fibonacci function is slightly different. The definition type for the output string is different as well, clang uses ".asciz " while gcc uses ".string"

X86 gcc:-

```
mov     rbx, QWORD PTR [rbp-8]
        leave ---------"uses leave"---------
        ret

---The string definition for printing"--
.LC0:
        .string "d = %d"
```

X86 Clang 14.0 :-

```
---"exit of fibonacci"---
 mov     eax, dword ptr [rbp - 4]
        add     rsp, 16 ------"adds 16 to stack pointer"-----
        pop     rbp ------"pops , rbp - base pointer"-------
        ret
---The string definition for printing"--
.L.str:
        .asciz  "d = %d"
```

3.2 ARM

The instruction set created is clearly different (since the CPU is different) but the main function calls remain the same and the same differences continue (slightly different uses of instructions during the exit of the fibonacii function). The instruction "bl" is used for calling the fibonacii function.

ARM GCC :-

```
ldr     r3, [r7, #4]
        subs    r3, r3, #1
        mov     r0, r3
```

```
        bl      fibonacci ----"calling of the function"-------
        mov     r4, r0
        ldr     r3, [r7, #4]
        subs    r3, r3, #2
        mov     r0, r3
        bl      fibonacci
        mov     r3, r0
        add     r3, r3, r4
```

ARM Clang:-

```
ldr     w8, [sp, #8]
        subs    w0, w8, #1                  // =1
        bl      fibonacci
        ldr     w8, [sp, #8]
        subs    w8, w8, #2                  // =2
        str     w0, [sp, #4]                // 4-byte Folded Spill
        mov     w0, w8
        bl      fibonacci
        ldr     w8, [sp, #4]                // 4-byte Folded Reload
        add     w9, w8, w0
        stur    w9, [x29, #-4] -----"Different Instruction used"-----
```

4 Gcc Optimization Levels

The gcc compiler does not store ”the optimized code” in a readable language.(it performs optimization in some intermediate languages usually). This means that the optimizations can only be seen after analyzing the assembly code generated. To do this we can compile using the ’-S’ flag.

Original Code:

```
#include<stdio.h>
#define N 100
/*My name is Kartik !*/
int fibonacci(int n){
  if(n<2)
  {
    return n;
  }
  else return fibonacci(n-1) + fibonacci(n-2);
}

        int main()
        {
                int a,  b, c;
                int K = 10;
                a = N;
                for(int i =0 ;i<10; i++)
                {
                a ++ ;
                b = 20;
                c = a + b;
                }
```

```
                c = fibonacci(K);
                printf("c = \n %d",c);
                return 0;

        }
```

4.1 Optimization O0

This is equivalent to no optimization. The code is just executed. There are seperate statements for evaluation of the variable ”b” and incrementing ”a” all present within the loop. These things are **NOT present in higher optimization levels!**.Assembly for part in the loop is as follows:-)

```
.L6:
        add     DWORD PTR [rbp-4], 1
        mov     DWORD PTR [rbp-20], 20
        mov     edx, DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-20] --- //(assignment of b)
        add     eax, edx -- //(addition of a and b)
        mov     DWORD PTR [rbp-16], eax
        add     DWORD PTR [rbp-8], 1
.L5:
        cmp     DWORD PTR [rbp-8], 9
        jle     .L6
```

4.2 Optimization O1

Optimize. Optimizing compilation takes somewhat more time. This optimization seems to really like shortening the code size :). The analyzed assembly does not ”unroll” the recursion too much, it keeps it in the ”calling” form(under **L4** - fibonacci has been called twice)

Assembly(for only fibonacci part):

```
fibonacci:
        push    rbp
        push    rbx
        sub     rsp, 8
        mov     ebx, edi
        cmp     edi, 1
        jg      .L4
.L2:
        mov     eax, ebx
        add     rsp, 8
        pop     rbx
        pop     rbp
        ret
.L4:
        lea     edi, [rdi-1]
        call    fibonacci-------------//(1)'st call
        mov     ebp, eax
        lea     edi, [rbx-2]
        call    fibonacci-------------//(2)'nd call
```

```
        lea     ebx, [rbp+0+rax]
        jmp     .L2
```

4.3 Optimization O2

This is a far more aggressive optimization, according to my understanding what it does is it unrolls the fibonacii sequence in a **loop style** until a certain stage(The code size has increased significantly). On reading more, the terminology for such a change is apparently called a **"tail call optimization"** (this involves just calling one fibonacci function again and again on not creating 2 branches of computation) Since the assembly code is huge (there are 18 such levels like **L18** of this form !) I am posting only the part pertaining to the function call

```

//level 18
.L18:
        lea     edi, [r14-1]
        mov     DWORD PTR [rsp+60], r9d
        sub     r14d, 2
        mov     DWORD PTR [rsp+56], esi
        mov     DWORD PTR [rsp+52], r10d
        mov     DWORD PTR [rsp+48], ecx
        mov     DWORD PTR [rsp+44], r8d
        mov     DWORD PTR [rsp+40], edx
        mov     DWORD PTR [rsp+36], r11d
        call    fibonacci----------------"(single call)"
        mov     r10d, DWORD PTR [rsp+52]
        mov     r11d, DWORD PTR [rsp+36]
        mov     edx, DWORD PTR [rsp+40]
        mov     r8d, DWORD PTR [rsp+44]
        add     r10d, eax
        cmp     r14d, 1
        mov     ecx, DWORD PTR [rsp+48]
        mov     esi, DWORD PTR [rsp+56]
        mov     r9d, DWORD PTR [rsp+60]
        jg      .L18
        lea     eax, [rbp-3]
        and     eax, 1
        add     eax,
```

4.4 Optimization O3, Ofast

This optimization makes almost no further changes to my code (It remains the smae as what we obtained after O2). But the O3 optimization does set extra flags that are not present in O2 optimizations. Ofast does not make any more changes than O2 to my code .Ofast is not very preferable however..since it disregards the "standard compliance rules"(messes with your code, therefore it is not very safe).

4.5 Optimization Os

What the optimization flag Os does is fantastic (it unrolls the recursion and solves it like a for loop, but does not take too much code size as well (it is more compact than the code obtained after O2 optimization) (It repeatedly adds fibonacci(n -1) ..etc. The code is both compact and the fibonacii function does not have 2 branches of computation.

Assembly code :-

```

fibonacci:
        push    rbx
        xor     ebp, ebp
        push    rbx
        mov     ebx, edi
        push    rcx
.L3:
        cmp     ebx, 1
        jle     .L5--------"exit condition?"
        lea     edi, [rbx-1]
        sub     ebx, 2
        call    fibonacci -"(calls fibonacii)"
        add     ebp, eax --"(repeatedly adds fibonacii)"
        jmp     .L3 --------"(jumps back to the beginning (the for loop))"
.L5:
        lea     eax, [rbx+rbp]
        pop     rdx
        pop     rbx
        pop     rbp
        ret
```

5 Conclusion

This completes the analysis of gcc optimization flags, and their effect on the assembly code generated.