INF-341: Compiladores
Tarea #3
"¿Optimiza?"

Anghelo Carvajal
201473062-4

3 de diciembre de 2018

Interesa saber qué mejoras al código es capaz de efectuar su compilador preferido. Algunos tienen diversas opciones (o niveles) de optimización. Explore las relevantes. Al efecto, considere al menos las siguientes situaciones:

1. Pregunta 1

¿Efectúa *constant folding*?

1.1. Respuesta 1

Teniendo en consideración el siguiente código, donde tenemos una expresión constante:

Código 1: `pregunta1.c`

```c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv){
    int i = 60*60*24*30;

    printf("%i", i);


    return 0;
}
```

Al compilar el código a *assembly*, resulta lo siguiente:

Código 2: `pregunta1.s`

```
        .file   "pregunta1.c"
        .section        .rodata
.LC0:
        .string "%i"
        .text
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        movl    $2592000, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
```

Lo cual podemos notar que en vez de dejar la multiplicación para el *run-time*, deja expresado el resultado de esta multiplicación como un valor constante ya calculado.

2. Pregunta 2

En GCC, es posible indicar que una función depende solo de sus argumentos, como por ejemplo el prototipo `__attribute__((pure)) int sqr(int);`.

¿Aprovecha realmente esto? O sea, si la función se llama varias veces con el mismo argumento, ¿la llama una sola vez?

2.1. Respuesta 2

Considerando el siguiente código, en el cual llamamos a la misma función *pure* mas de una vez con el mismo argumento:

Código 3: `pregunta2.c`

```c
#include<stdio.h>
#include<stdlib.h>

int __attribute__((pure)) square_pure(char *x){
    return x!=NULL ? x[0] + x[1] - x[2] : 0;
}

int main(int argc, char **argv){
    int i, j, k;
    char coso[100];

    int ret = scanf("%s", coso);

    i = square_pure(coso);
    j = square_pure(coso);
    k = square_pure(coso);

    int result = i+j+k;

    printf("%i", result);
    return 0;
}
```

El código anterior genera el siguiente código intermedio:

Código 4: `pregunta2.c.191t.optimized`

```
;; Function square_pure (square_pure, funcdef_no=38, decl_uid=3124, cgraph_uid=38, symbol_order=

Removing basic block 5
square_pure (char * x)
{
  int iftmp.0_1;
  char _4;
  int _5;
  char _6;
  int _7;
  int _8;
  char _9;
  int _10;
  int iftmp.0_11;

  <bb 2>:
  if (x_2(D) != 0B)
    goto <bb 3>;
  else
    goto <bb 4>;

  <bb 3>:
  _4 = *x_2(D);
  _5 = (int) _4;
  _6 = MEM[(char *)x_2(D) + 1B];
  _7 = (int) _6;
  _8 = _5 + _7;
```

```
  _9 = MEM[(char *)x_2(D) + 2B];
  _10 = (int) _9;
  iftmp.0_11 = _8 - _10;

  <bb 4>:
  # iftmp.0_1 = PHI <iftmp.0_11(3), 0(2)>
  return iftmp.0_1;

}



;; Function main (main, funcdef_no=39, decl_uid=3128, cgraph_uid=39, symbol_order=39) (executed

main (int argc, char * * argv)
{
  int result;
  char coso[100];
  int i;
  int _4;

  <bb 2>:
  scanf ("%s", &coso);
  i_3 = square_pure (&coso);
  _4 = i_3 + i_3;
  result_5 = i_3 + _4;
  __printf_chk (1, "%i", result_5);
  coso ={v} {CLOBBER};
  return 0;

}
```

De lo anterior podemos ver claramente que en vez de llamar 3 veces a la función `square_pure`, la llama una única vez y suma a si mismo ese resultado las veces necesarias (en este caso 3). Esto ocurre debido a que el compilador sabe que el output no va a cambiar, por lo que puede hacer la optimización.

3. Pregunta 3

¿Expande funciones en línea según dirige por ejemplo el listado 1? ¿Lo hace automáticamente para funciones «simples» (como la anterior) si fue definida antes? ¿Si se define después?

3.1. Respuesta 3

Se plantea el siguiente código con una función *inline*.
Nota: se usa *static inline* en lugar de *inline* a secas para entregar el resultado en un único archivo.

Código 5: `pregunta3.c`

```c
#include<stdio.h>
#include<stdlib.h>

static inline int sqr(char *x){
    return x!=NULL ? x[0] + x[1] - x[2] : 0;
}

int main(int argc, char **argv){
    char i[100];

    int retVal = scanf("%s", i);

    int j = sqr(i);

    printf("%i\n", j);
    return 0;
}
```

El código anterior genera el siguiente código intermedio:

Código 6: `pregunta3.c.191t.optimized`

```
;; Function main (main, funcdef_no=39, decl_uid=3128, cgraph_uid=39, symbol_order=39) (executed

main (int argc, char * * argv)
{
  char i[100];
  char _3;
  int _5;
  char _6;
  int _7;
  int _8;
  char _9;
  int _10;
  int iftmp.0_11;

  <bb 2>:
  scanf ("%s", &i);
  _3 = MEM[(char *)&i];
  _5 = (int) _3;
  _6 = MEM[(char *)&i + 1B];
  _7 = (int) _6;
  _8 = _5 + _7;
  _9 = MEM[(char *)&i + 2B];
  _10 = (int) _9;
  iftmp.0_11 = _8 - _10;
  __printf_chk (1, "%i\n", iftmp.0_11);
  i ={v} {CLOBBER};
  return 0;

}
```

Como podemos ver, en vez de haber un llamado a la función en cuestión, se insertaron un montón de lineas de códigos que corresponderían al llamado de esta función.

De modo que se esta respetando el *inline*.

4. Pregunta 4

¿Se propagan valores de variables? O sea, por ejemplo si se asigna `a = 1;` y luego (después de varias otras líneas que no la afectan) se usa esta variable, ¿usa el valor? ¿Por ejemplo, elimina un `if (a != 1) { ... }`?

4.1. Respuesta 4

Aquí indicamos un valor para `a`, y luego preguntamos si esa variable contiene un valor distinto al que le ingresamos originalmente:

Código 7: `pregunta4.c`

```
#include<stdio.h>


int main(){
    int a = 1;
    printf("Linea que no afecta a la variable a");
    printf("Linea que no afecta a la variable a");
    printf("Linea que no afecta a la variable a");

    int j = 15;

    if(a != 1){
        printf("no deberia impirmir.");
    }

    printf("el valor de a era %i", a);

    return 0;
}
```

El código anterior genera el siguiente código intermedio:

Código 8: `pregunta4.c.191t.optimized`

```
;; Function main (main, funcdef_no=23, decl_uid=2480, cgraph_uid=23, symbol_order=23) (executed

main ()
{
  <bb 2>:
  __printf_chk (1, "Linea que no afecta a la variable a");
  __printf_chk (1, "Linea que no afecta a la variable a");
  __printf_chk (1, "Linea que no afecta a la variable a");
  __printf_chk (1, "el valor de a era %i", 1);
  return 0;

}
```

Como el compilador sabe que este valor no ha cambiado, se salta completamente el *if*, no incluyendo ni la comparacion ni sus contenidos.

5. Pregunta 5

¿Se recalculan (sub)expresiones que no varían?

5.1. Respuesta 5

Tenemos el siguiente `for` donde un calculo es dependiente del índice, y el otro no:

Código 9: `pregunta5.c`

```
#include<stdio.h>


int main(){
    int a[20];
    int x = 5;
    for(int y = 0; y < 10; y++){
        a[y] = x * x + y * y;
    }
    return a[9];
}
```

El código anterior genera el siguiente código intermedio:

Código 10: `pregunta5.c.191t.optimized`

```

;; Function main (main, funcdef_no=23, decl_uid=2480, cgraph_uid=23, symbol_order=23) (executed

Removing basic block 5
main ()
{
  unsigned long ivtmp.3;
  int y;
  int a[20];
  int _4;
  int _5;
  int _8;

  <bb 2>:

  <bb 3>:
  # ivtmp.3_1 = PHI <ivtmp.3_11(3), 0(2)>
  y_12 = (int) ivtmp.3_1;
  _4 = y_12 * y_12;
  _5 = _4 + 25;
  MEM[symbol: a, index: ivtmp.3_1, step: 4, offset: 0B] = _5;
  ivtmp.3_11 = ivtmp.3_1 + 1;
```

```
23    if (ivtmp.3_11 != 10)
24      goto <bb 3>;
25    else
26      goto <bb 4>;
27
28    <bb 4>:
29    _8 = a[9];
30    a ={v} {CLOBBER};
31    return _8;
32
33  }
```

Podemos ver que en vez de dejar que en *run-time* se calcule la expresión que sabemos que siempre sera la misma (`x*x;`), el compilador cambia ese calculo por el resultado.

Esto lo podemos ver en la linea 20, donde se ve que deja expresado el `25`.

La linea dice: `_5 = _4 + 25;`.

6. Pregunta 6

¿Se calculan expresiones cuando se requieren solamente?

6.1. Respuesta 6

En el siguiente código podemos ver que calculamos una expresión numérica, la cual solo usamos dentro de un `if`:

Código 11: `pregunta6.c`

```
1  #include<stdbool.h>
2  #include <stdio.h>
3
4  int main(int argc, char **argv){
5      double y;
6      int x;
7      bool flag = argc> 1;
8
9      int ret = scanf("%i", &x);
10
11     y = ((3 * x + 20) * x - 135) * x + 42;
12     if(flag)
13         printf("y = %lf", y);
14
15     return 0;
16 }
```

El código anterior genera el siguiente código intermedio:

Código 12: `pregunta6.c.191t.optimized`

```
1
2  ;; Function main (main, funcdef_no=23, decl_uid=2482, cgraph_uid=23, symbol_order=23) (executed
3
4  Removing basic block 5
5  main (int argc, char * * argv)
6  {
7    int x;
8    double y;
9    int x.0_5;
10   int _6;
11   int _7;
12   int _8;
13   int _9;
14   int _10;
15   int _11;
16
17   <bb 2>:
18   scanf ("%i", &x);
19   if (argc_2(D) > 1)
20     goto <bb 3>;
```

```
21    else
22      goto <bb 4>;
23
24    <bb 3>:
25    x.0_5 = x;
26    _6 = x.0_5 * 3;
27    _7 = _6 + 20;
28    _8 = x.0_5 * _7;
29    _9 = _8 + -135;
30    _10 = x.0_5 * _9;
31    _11 = _10 + 42;
32    y_12 = (double) _11;
33    __printf_chk (1, "y = %lf", y_12);
34
35    <bb 4>:
36    x ={v} {CLOBBER};
37    return 0;
38
39  }
```

Podemos ver que en el código generado solo calcula el numero necesario si es que la `flag` se cumple, a pesar de que dejamos el calculo fuera del *if*.

7. Pregunta 7

¿Maneja variables de inducción, que tienen una relación lineal con el índice del ciclo? Por ejemplo, en el listado 4 podemos inicializar una variable temporal con `tmp = 15;` y en cada ciclo incrementar `tmp += 4;`.

7.1. Respuesta 7

No logre que el compilador hiciera uso de variables de induccion.

8. Pregunta 8

En ciclos con cuerpo corto (como los ejemplos previos) el costo del ciclo puede ser una fracción relevante del costo total. La modificación de *loop unrolling* ejecuta varias iteraciones en cada ciclo. Detalle el ambiente empleado (sistema operativo, arquitectura, compilador, opciones).

8.1. Respuesta 8

Se intento de varias formas que el compilador realizara un *loop unrolling* sin exito, incluso usando el flag `-funroll-all-loops`.

Se adjunta un código usado para testear el *loop unrolling*.

Código 13: `pregunta8.c`

```c
#include <stdio.h>

int main(){
    int x, i;

    int ret = scanf("%i", &x);

    for(i = 0; i < 500; i++){
        printf("%i", i);
    }
    printf("%i\n", x);
    return 0;
}
```

9. Detalles de arquitectura

Detalle el ambiente empleado (sistema operativo, arquitectura, compilador, opciones)

- Sistema operativo: Ubuntu 16.04
- Arquitectura: Linux version 4.15.0-39-generic (builddlcy01-amd64-012) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.10)) #42 16.04.1-Ubuntu SMP Wed Oct 24 17:09:54 UTC 2018
- Compilador: gcc (Ubuntu 5.4.0-6ubuntu1 16.04.10) 5.4.0 20160609
- Opciones del compilador: -S -save-temps -fdump-tree-optimized -std=gnu11 -O1