实验预备工作实验报告

徐云凯 1713667

摘要

GCC 编译器是使用最广泛的 C/C++编译器。GCC 编译器编译 C 语言程序主要分为预处理、编译、汇编、链接 4 个主要阶段。本预备工作通过使用 GCC 编译 3 个自己编写的简单 C 语言程序，在分步骤编译和逐步尝试各种参数的过程中探索 GCC 编译器的技术细节，为之后自己制作编译器打下基础。

关键词：编译原理 GCC 编译器 编译优化

一、 引言

编译器对代码的处理主要分为预处理、编译、汇编、链接几个部分。其中预处理器去除注释，将#include、#define、#ifdef 等预处理指令替换为相应的代码块，该步骤不进行代码优化调整。编译器进行词法分析、语法分析、语义分析、中间代码生成、代码优化与代码生成等步骤后，将高级语言翻译为汇编代码，该步骤可以借助 AST 树分析语法结构，检查类型错误等语法错误；借助 CFG 图等结构优化程序流程，进行代码优化。汇编器将汇编代码转化为对应平台的机器指令。链接器将机器指令进行连接，形成完整的可执行文件。

在探索过程中，通过由简单到复杂的顺序。先后构建不含 include 语句的最简单程序，查看编译各个阶段输出的文件，以了解编译输出文件含义。随后构建具有 include 语句，代码流程有优化空间的代码，运行并测试在不同优化条件，不同参数下的输出结果。

二、 编译流程分析

2.1. 使用不含 include 语句的简单程序探索输出文件含义

当文件中包含 include 时，预处理之后代码会加入大量 c 语言库文件，从而变得极其复杂难以分析其具体含义。这里先编写了一段简单的不包含头文件的程序初步探索编译器各个部分的作用。

```
1  // simple.c
2  void main() {
3      int n = 100;
4      while(n > 0)
5          n--;
6  }
```

此段代码经过预处理后可以得到如下 simple.i 文件# 1 "simple.c"

```
1  # 1 "<built-in>"
2  # 1 "<command-line>"
3  # 31 "<command-line>"
4  # 1 "/usr/include/stdc-predef.h" 1 3 4
5  # 32 "<command-line>" 2
6  # 1 "simple.c"
7  void main() {
8      int n = 100;
9      while(n > 0)
```

```
        n--;
    }
```

由于没有#include <stdio.h>，这里预处理器做的基础操作变得简单明了。预处理器删除注释代码，替换宏定义等预处理指令（这里不涉及），在文件头添加了一些以“#”开头的行同步指令，它们帮助 gcc 为#included 文件中的错误提供正确的错误消息。其含义如下：

 # line-number "source-file" [flags]

flags 标志的含义（空格分隔）：

1 - 开始新文件

2 - 返回上一个文件

3 - 以下文本来自系统头文件（#include <> vs #include “”）

4 - 以下文本应被视为包含在隐式 extern “C”块中。

在探索预处理器的过程中，有一些有意思的发现。一些诸如 unix，vax，linux 等字符串，即便没有任何#define 语句，没有#include 任何文件，它们在 gcc 编译器中都会被默认认为是宏。比如 linux 会被 gcc 的预处理器自动替换为 1。这是由于早期的 C 语言（1989 年 ANSI C 标准出台前）对于编译目标平台的标记方式没有严格约束，GNU 编译器的实现方式是将对应平台字符串作为宏定义进入预处理器，并且默认开发者会避开这些字符串。而在 1989 年 ANSI C 标准出台后，限制了编译器对符号的使用，编译器中预定义宏的符号只能以两个下划线开头，或者一个下划线后紧接着大写字母。目前在 gcc 中想要避免这些字符串被替换，应当在编译时使用-std=c90 -pedantic 参数明确 C 语言标准。

随后，对源码进行语法分析，并绘制控制流程图。

使用“-fdump-tree-original-raw”参数生成抽象语法树如下（simple.c.003t.original）

```
;; Function main (null)
;; enabled by -tree-original

@1   bind_expr    type: @2       vars: @3       body: @4
@2   void  type   name: @5       algn: 8
@3   var  decl    name: @6       type: @7       scpe: @8
                  srcp: simple.c:2              init: @9
                  size: @10      algn: 32       used: 1
@4   statement_list              0  : @11      1 : @12        2 : @13
```

```
                    3    : @14      4 : @15        5 : @16
                    6    : @17
@5   type_decl      name: @18       type: @2
@6   identifier_node                strg: n        lngt: 1
@7   integer_type   name: @19       size: @10      algn: 32
                    prec: 32        sign: signed   min : @20
                    max : @21
@8   function_decl   name: @22       type: @23      srcp: simple.c:1
                    link: extern
@9   integer_cst     type: @7        int: 100
@10 integer_cst     type: @24       int: 32
```

以上是截取的语法树文件前 20 行。由于 simple.c 的代码足够简单，很容易直接看出其每一行的具体含义。该文件的格式为：@sig　name　attribute list •••。文件使用链表方式记录了语法树，每一个“@”符号作为地址符号开始一个新的节点，其后为节点名称，以及节点属性列表。例如 13 行@6 identifier_node 标识符节点作为叶节点，没有后继，只有两个属性标识符名称和名称字符串长度。

随后在-O0 优化等级下，分别使用“-fdump-tree-all-graph”和“-fdump-rtl-all-graph”两个参数获取得中间代码生成的多阶段输出，使用插件将其绘图如图 1。

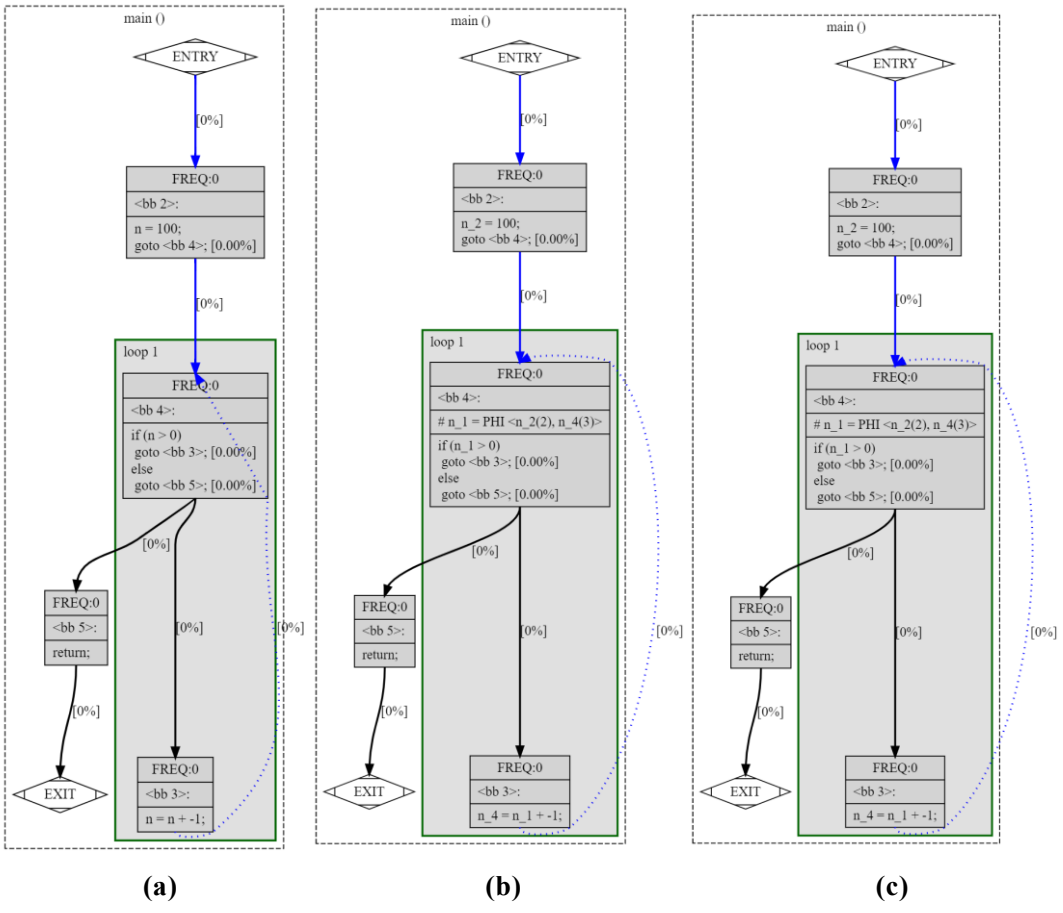


图 1　simple.c 的 CFG, (a)原始 CFG，(b)SSA 后的 CFG，(c)完全优化后的 CFG

由于该程序较为简单，优化前后代码流程不变。但是可以看出一些优化的痕迹，比如加入 n_1，n_2，n_4 等临时变量，优化循环体中的相邻指令变量依赖，使代码利于流水线运行。

然后将 simple.i 编译到汇编代码。

```
        .file   "simple.c"
        .text
        .globl  main
        .type   main, @function
    main:
    .LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $100, -4(%rbp)
        jmp .L2
    .L3:
        subl    $1, -4(%rbp)
    .L2:
        cmpl    $0, -4(%rbp)
        jg  .L3
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
    .LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section    .note.GNU-stack,"",@progbits
```

汇编代码中的几个标签与 CFG 图优化完成后的图节点一致。但是代码中仍然存在 nop 气泡指令。因为循环体太小，流水线中的数据冒险不能完全避免。

最后将汇编代码转换到机器码并链接形成最终可执行文件。这一过程的输出文件都是二进制编码难以查看。

在编译工作完成后，再通过反汇编，对比.obj 中间文件和最终生成的可执行文件中的编码，可以发现链接所做的工作。相比于.obj 中间文件，最终生成的可执行文件，在 main 程序段的开头和结尾附加了多个程序段。加在 main 片段之前的分别是_init，.plt，__cxa_finalize@plt，_start，deregister_tm_clones，register_tm_clones，__do_global_dtors_aux，frame_dummy。加在 main 片段之后的是__libc_csu_init，__libc_csu_fini 和_fini。查阅资

料了解到它们是系统标准启动文件，对于任何需要在 linux 系统中启动的程序都必须附加。

此时，再尝试再链接时加入“-static”参数，对此时链接生成的可执行文件进行反汇编，与之前的可执行性文件反汇编结果进行对比，同时与中间文件反汇编结果进行对比。发现此时的可执行文件除了附加正常的系统标准启动文件之外，还附加了多个程序块，之前的程序块也有不同程度的扩增。反汇编结果达到了 146k 行。.plt 块新增了多个_GLOBAL_OFFSET_TABLE，新增了_IO_FILE_*等多个与 IO 相关的块。此外，很多的内存管理相关程序块也被附加到程序中。另外，很多在程序中并未使用的库函数与对象也被加入其中，如 strcmp()，strlen()等。

2.2. 使用常规程序探索编译器优化

前面使用不包含#include <stdio.h>的代码，以简化编译输出，便于研究输出内容含义，这里使用包含标准输入输出的正常代码以研究编译器对代码流程的优化。

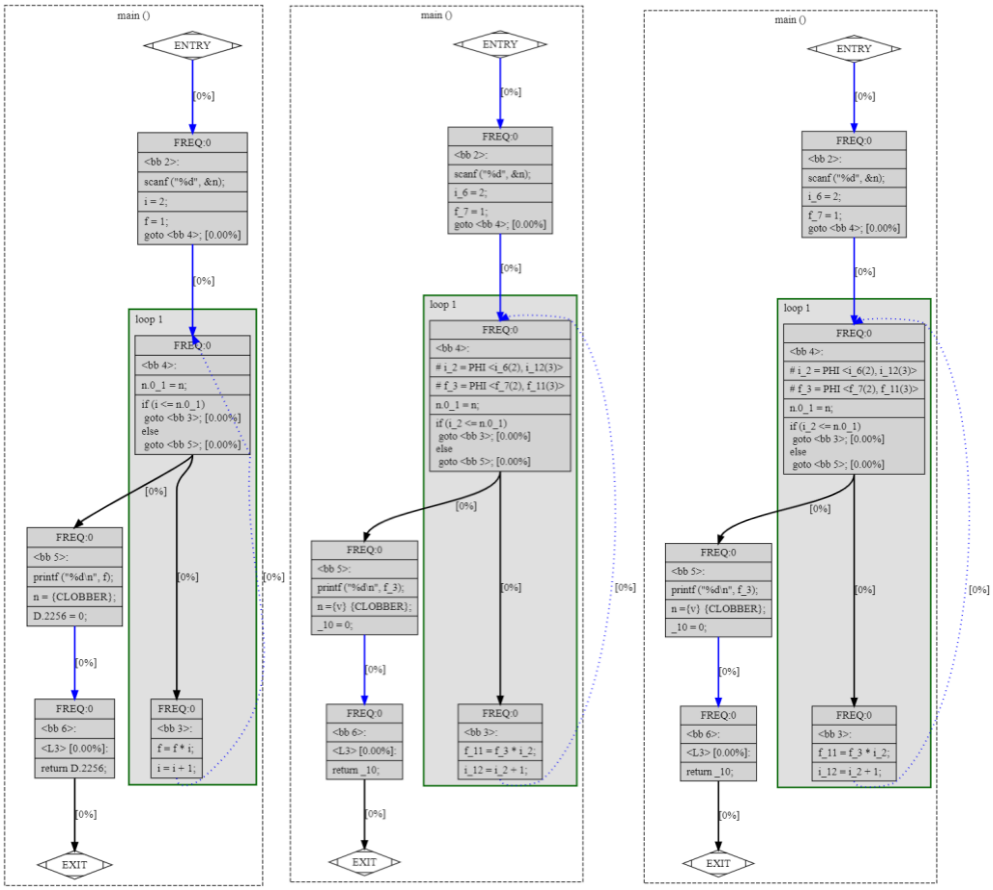
实验指导中所给的代码不包含#include 语句，并且是 C++代码，在 gcc 上编译可能产生问题，故稍作修改，将其变为 C 语言代码。修改后的代码见附录（附录代码不包含计时部分）。

首先对程序代码进行预处理。这里发现预处理得到的 p1.i 文件前 7 行与 2.1 节中的预处理输出一致。但是这一次预处理器替换<stdio.h>后，预处理输出代码增加了 788 行来自头文件的代码。

随后使用-fdump-tree-original-raw 参数进行语法分析，得到抽象语法树（AST）。前文已对齐进行分析，这里不再赘述。

随后生成源码的 CFG。其不同阶段优化后的 CFG 如图 2。对比优化前后可以发现其同样对循环体中的计算增加了临时变量。但是仍然没有观察到流程上的重大变化。

最后对源码进行编译生成汇编代码，汇编代码翻译到机器码后链接形成可执行文件。同样对汇编器输出的中间文件和链接器输出的可执行文件进行反汇编得到反汇编代码。对比可执行文件的反汇编代码与中间文件反汇编代码，同样可以发现可执行文件与之前的 2.1 节相比，附加了同样的前后代码块。



(a) **(b)** **(c)**

图 2　O0 优化下阶乘程序的 CFG, (a)原始 CFG，(b)SSA 后的 CFG，(c)完全优化后的 CFG

接下来探索编译器优化指令。先将编译器优化级别由-O0 改为-O1。重新输出 CFG 和优化后的 CFG。可以发现相比于 O0 优化，此时 CFG 有了不少变化。

首先是 CFG 首次将 printf 的内部结构纳入图中（图 3），在后面的优化中，printf()的流程被和主程序流程合并考虑，一同优化。最后的 CFG 中只包含一张图，说明此时的流程是包含主程序 main()函数代码和头文件函数代码在内的优化后情况。

最后，尝试一些-fno-*指令。

-fno-builtin

当使用的函数名与 C 语言运行库里面已经存在的函数名冲突的时候，附加该参数可以避免冲突，优先使用自己定义的函数名。

-fno-rtti　-fno-exceptions

禁用运行时类型信息和禁用异常机制。对于已经完善的程序代码，如果在生产环境中，需要高性能、低资源占用的运行。可以使用这两个参数进行编译以禁用相应部件，提高

运行效率。

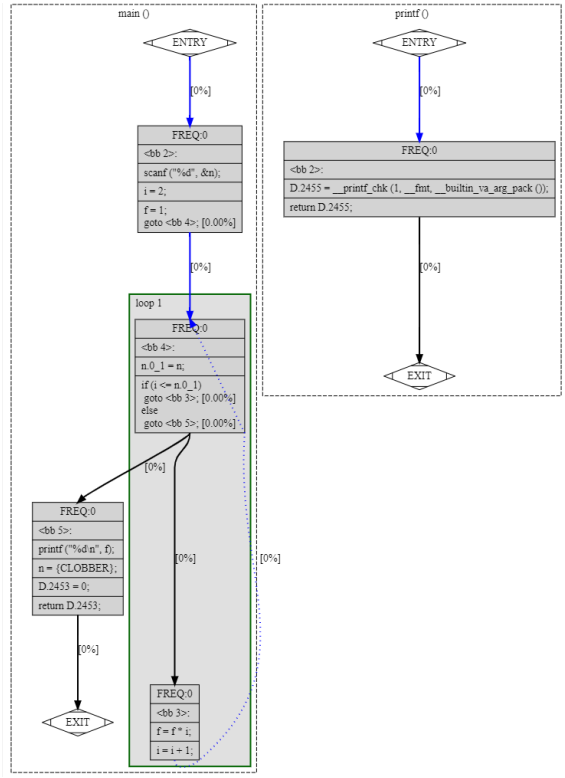


图 3　O1 优化下的阶乘程序原始 CFG

三、 结论

（1） GCC 编译器编译程序分为预处理、编译、汇编、链接 4 个主要阶段。

（2） 预处理阶段执行预处理指令，进行宏替换，完成包含文件附加等操作。

（3） 编译阶段分为**词法分析，语法分析，语义分析，中间代码生成，代码优化**与

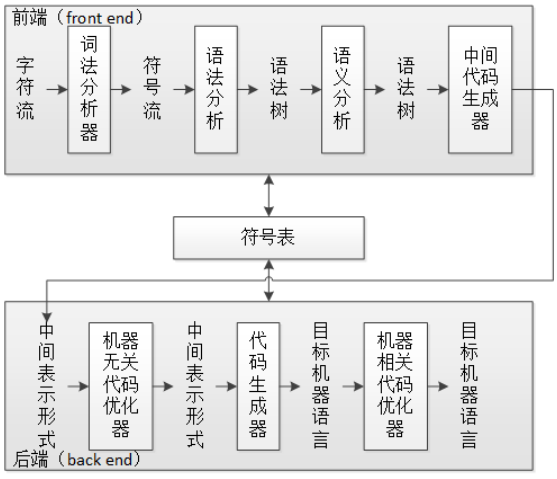


图 4　编译具体步骤

转换到**汇编语言**等步骤。其中**语法分析**借助 AST 树，分析代码的语法结构，将其转换为机器可以处理的树数据结构，**语义分析**及之后的优化算法较多，主要是围绕 CFG 图进行流程优化。这一过程具体进行的优化内容由编译器优化选项控制。-O0 是不做任何优化，这是默认的编译选项。-O 和-O1 对程序做部分编译优化，具体包括合并栈弹出，分支的精简，简化循环条件，减少相邻指令的寄存器依赖性（这一条在第一个程序中非常明显）等。-O2 是更高级别优化，包括消除尾递归，消除循环计数变量，排查无用的指针和变量，优化寄存器的使用，重排指令以防止数据冒险带来的空指令。最后是-O3 最高级别优化，此时会内联简单函数到调用函数，使用伪寄存器网络优化寄存器使用，将无变化的条件分支移出循环等。

（4） 汇编阶段是后端部分，将汇编语言转换为对应平台的机器码。该过程主要分为**指令选择**、**寄存器分配**、**指令调度**和**指令编码** 4 个步骤。**指令选择**过程通常使用成本驱动模型，用树匹配方法或选择 DAG 技术选择对应指令进行代码映射。**寄存器分配**分为两个步骤，先为指令处理向量，然后执行分配。**指令调度**主要是为了减少寄存器依赖性，针对流水线进行优化。**指令编码**只需要根据前面选择的寄存器确定对应的指令即可。

（5） 链接阶段分为动态链接和静态链接两种，默认的动态链接不会将 C++库函数和一些操作系统提供的函数放入编译好的程序中，而是在运行时（runtime）进行链接调用。静态链接则相反，其会将所有的库装入编译好的程序中，此种链接方式会带来较大的可执行文件，但是可以获得尽量好的平台兼容性。

参考文献

[1] 项炜.浅析 GCC 的代码优化机制[J].现代经济信息,2008(04):154-155.

[2] 张滇. 基于 GCC 的中间代码优化技术研究[D].哈尔滨理工大学,2008.

[3] 种瓜大爷 gcc 程序的编译过程和链接原理[Z].https://blog.csdn.net/czg13548930186/article/details/78331692

[4] 编译器后端[Z].https://www.freedesktop.org/wiki/Software/Beignet/Backend/compiler_backend/

附录

Makefile

```
.PHONY: pre, ast, ir, asm, obj, exe, antiobj, antiexe

default:
	cc -O0 -o p1.out p1.c

pre:
	cc -E -o p1.i p1.c

ast:
	cc -fdump-tree-original-raw p1.c

cfg:
	cc -O0 -fdump-tree-all-graph p1.c

ir:
	cc -O0 -fdump-rtl-all-graph p1.c

asm:
	cc -O0 -S -masm=att -o p1.S p1.i

obj:
	cc -O0 -c -o p1.o p1.S

antiobj:
	objdump -d p1.o > p1-anti-obj.S
	nm p1.o > p1-nm-obj.txt

exe:
	cc -O0 -o p1.out p1.o

antiexe:
	objdump -d p1.out > p1-anti-exe.S
	nm p1.out > p1-nm-exe.txt

clean:
	-rm *.c.*

clean-all:
	-rm *.c.* *.o *.S *.i *.dot *.out *.txt
```

simple.c

```
void main() {
    int n = 100;
    while(n > 0)
        n--;
}
```

p1.c

```c
#include <stdio.h>

int main()
{
    int i, n, f;

    scanf("%d", &n);
    i = 2;
    f = 1;
    while (i <= n)
    {
        f = f * i;
        i = i + 1;
    }
    printf("%d\n", f);
}
```

p2.c

```c
#include <stdio.h>

int main()
{
int a, b, i, t, n;

a = 0;
b = 1;
i = 1;
scanf("%d", &n);
printf("%d ", b);
while (i < n)
{
      t = b;
            b = a + b;
            printf("%d ", b);
            a = t;
            i = i + 1;
}
printf("\n");
}

```