



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

编译原理实验报告

预备工作 1——了解编译器及 LLVM IR 编程

谢畅 唐文涛

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2023 年 9 月 17 日

摘要

本文介绍了编译过程及 LLVM IR 编程相关的内容。在文章开头描述了一个完整的编译过程,包括预处理、编译、汇编和链接等阶段。接下来介绍了实验的分工和实验平台。正文部分结合具体的程序,首先详细介绍了 G++ 编译器的各个处理过程。随后讨论了 LLVM 编译器的各个处理阶段。最后,通过设计两个程序并通过 LLVM IR 实现并验证正确性,熟悉了 LLVM IR 的特性。

关键字: 编译器 gcc llvm/clang

目录

1 一个完整的编译过程	1
1.1 预处理	1
1.2 编译	1
1.3 汇编	1
1.4 链接	1
2 实验分工与实验平台	2
2.1 实验分工	2
2.2 G++ 编译器实验平台	2
2.3 LLVM 编译器实验平台	2
3 G++ 编译器的各个处理过程	2
3.1 预处理器	4
3.2 编译器	4
3.2.1 语法分析	4
3.2.2 中间代码生成与优化(一)	6
3.2.3 中间代码生成与优化(二)	8
3.3 编译器——开启优化选项	9
3.3.1 -O1 优化	9
3.3.2 -O3 优化	10
3.4 汇编器	11
3.5 链接器	12
4 llvm 编译器的各个处理阶段	13
4.1 预处理器	14
4.2 编译器	15
4.2.1 词法分析	15
4.2.2 语法分析	16
4.2.3 中间代码生成	17
4.2.4 代码优化的介绍	18
4.2.5 代码优化的程序对比	19
4.2.6 代码生成	27
4.3 汇编器	27

4.4 链接器	28
5 LLVM IR 程序设计	29
5.1 SysY 语言特性	29
5.2 程序一	30
5.2.1 语言特性	30
5.2.2 程序伪代码与设计思路	30
5.2.3 LLVM IR 实现	31
5.2.4 正确性验证	34
5.3 程序二	34
5.3.1 设计思路	34
5.3.2 伪代码实现	35
5.3.3 LLVM IR 实现	35
5.3.4 正确性验证	38
6 总结与展望	40

1 一个完整的编译过程

编译过程分为四个阶段：预处理（Preprocessing）、编译（Compilation）、汇编（Assembly）、链接（Linking）。执行这四个阶段的程序（预处理器、编译器、汇编器、和链接器）一起构成了编译系统。

1.1 预处理

在预处理阶段，系统会

- 对其中的伪指令（以 # 开头的指令）进行处理：将所有的 #define 删除，并且展开所有的宏定义；处理条件编译指令，如 #if、#elif、#else、endif 等；处理头文件包含指令，如 #include，将被包含的文件插入到该预编译指令的位置
- 删除所有的注释
- 添加行号和文件名标识

1.2 编译

编译器将预处理完的文本文件（后缀名.i）进行一系列的词法分析、语法分析、语义分析和优化，翻译成一个汇编语言程序（后缀名.s）。编译过程可分为 6 步：词法分析、语法分析、语义分析、源代码优化、代码生成、目标代码优化。

- 词法分析：扫描器将源代的字符序列分割成一系列的 Token。lex 工具可实现词法扫描。
- 语法分析：语法分析器将 Token 产生语法树。yacc 工具可实现语法分析。
- 语义分析：静态语义（在编译器可以确定的语义）、动态语义（只能在运行期才能确定的语义）。
- 源代码优化：源代码优化器，将整个语法书转化为中间代码（中间代码是与目标机器和运行环境无关的）。中间代码使得编译器被分为前端和后端。编译器前端负责产生机器无关的中间代码；编译器后端将中间代码转化为目标机器代码。
- 目标代码生成：代码生成器。
- 目标代码优化：目标代码优化器。

1.3 汇编

汇编阶段将编译完的汇编代码文件翻译成机器指令，保存在后缀为.o 的目标文件中。

这个文件是一个 ELF 格式的文件（Executable and Linkable Format，可执行可链接文件格式），包括可以被执行的文件和可以被链接的文件（如目标文件.o，可执行文件.exe，共享目标文件.so），有其固定的格式。

1.4 链接

由汇编程序生成的目标文件并不能被立即执行，还需要通过链接器，将有关的目标文件彼此相连接，使得所有的目标文件成为一个能够被操作系统载入执行的统一整体。

链接处理可以分为两种：

- 静态链接：直接在编译阶段就把静态库加入到可执行文件当中去。优点：不用担心目标用户缺少库文件。缺点：最终的可执行文件会较大；且多个应用程序之间无法共享库文件，会造成内存浪费。
- 动态链接：在链接阶段只加入一些描述信息，等到程序执行时再从系统中把相应的动态库加载到内存中去。优点：可执行文件小；多个应用程序之间可以共享库文件。缺点：需要保证目标用户有相应的库文件。

2 实验分工与实验平台

2.1 实验分工

唐文涛负责编写1：一个完整的编译过程，3：g++ 编译器的各个处理阶段，谢畅负责编写4：LLVM 编译器的各个处理阶段。最后5：LLVM IR 程序设计中，谢畅负责编写程序一，唐文涛负责编写程序二。

2.2 G++ 编译器实验平台

- VMware 虚拟机
- X86 架构 Linux 平台
- 操作系统：Ubuntu 22.04.3
- 编译器版本：g++ (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0

2.3 LLVM 编译器实验平台

- Vscod ssh 连接 VMware 虚拟机
- X86 架构 Linux 平台
- 操作系统：Ubuntu 22.04.3
- 编译器版本：LLVM 14.0.0

3 G++ 编译器的各个处理过程

本节将使用 G++ 编译一段构造的代码，通过观察生成的中间文件，了解 G++ 编译器的工作流程。在编写时，考虑到了如函数、常量、宏、头文件等因素，代码及编写意图如下：

```
1 //include 指令仅在“预处理”部分使用
2 //在探索其他部分时将删去
3 #include<iostream>
4 #define ZERO 0
5 int func(int x,int y)
6 {
7     int z=x*y+3;
```

```
8     return z;
9 }
10
11 int main(){
12     int x,y,j,t;
13     //use macro
14     x=ZERO;
15
16     //y=1;
17     //dead code
18     if(5>4)
19         y=1;
20     else
21         y=2;
22
23     //j=1;
24     //constant folding
25     j=5-4;
26
27     int a[100],b[100],c[100];
28     for(int i=0;i<100;i++)
29     {
30         a[i]=i;
31         b[i]=i+1;
32     }
33     //Vectorize
34     for(int i=0;i<100;i++)
35     {
36         c[i]=a[i]+b[i];
37     }
38
39     //Function Inlining
40     //Constant Propagation
41     int z=func(x,y);
42     z=z+c[0];
43     return z;
44 }
```

3.1 预处理器

预处理是编译过程中的第一个阶段，它主要负责对源代码进行文本处理和转换，以生成经过宏展开、头文件包含、条件编译等处理的中间代码。

使用指令 `g++ main.c -E -o main.i` 得到预处理后的文件 `main.i`，可以发现**删除了所有注释，代码的规模扩大到了 3 万多行，文件开头增加了一些头文件**，这是因为预处理时对 `#include` 指令会替换对应的头文件。另外，`main.cpp` 中使用的**宏定义 `x=ZERO` 也被替换为 `x=0`**。

源代码预处理之后对应的部分如下：

```
1  # 4 "main.cpp"
2  int func(int x,int y)
3  {
4      int z=x*y+3;
5      return z;
6  }
7
8  int main(){
9      int x,y,j,t;
10     x=0;
11
12     if(5>4)
13         y=1;
14     else
15         y=2;
16
17     j=5-4;
18     .....
19 }
```

3.2 编译器

3.2.1 语法分析

`g++` 将词法分析生成的词法单元来构建抽象语法树。通过 `-fdump-tree-original-raw` flag 获得文本格式的 AST 输出“`main.cpp.005t.original`”

使用 Github 开源项目 [1] 将 `.original` 文件转为 `.dot` 文件，分别**可视化函数 `func` 与 `main`**，可以进一步观察 `g++` 编译器生成的抽象语法树，以 `func` 函数为例，如图1所示：

3.2.2 中间代码生成与优化 (一)

首先通过“g++ main.cpp -fdump-tree-all-graph -o main”指令获得中间代码生成的多阶段的输出, 通过阅读各个阶段的中间代码, 分析中间代码生成与优化的步骤。

这一阶段中, g++ 输出了约 50 个中间文件, 其中一半为文本文件对应的.dot 文件, 所以我们推测 g++ 此时进行了 20 多个阶段的中间代码转换, 如图2

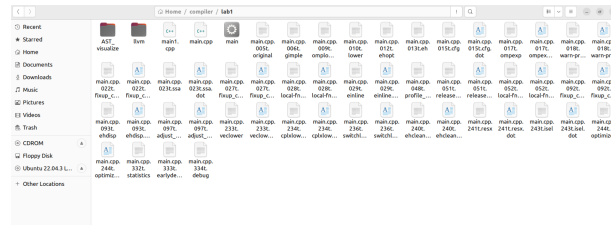


图 2: -fdump-tree-all-graph 产生的中间代码

这些代码介于 c++ 与汇编语言, 可读性较强。逐一阅读源码文件, 可以找出**优化较明显的几个文件**以及其**优化之处**。

“main.cpp.005t.original”文件对 C++ 代码**增加了地址标记,进行了语法标识**,如 cleanup_point, expr_stmt 等, 同时**将 for 循环改写为 if-else 分支和跳转语句 goto**。

此外, 这一阶段还**计算出了常量表达式**, 如源代码中的“j=5-4”在此处为“j=1”。

```

1 //main.cpp.005t.original
2 //计算出常量表达式
3 <<cleanup_point <<< Unknown tree: expr_stmt
4 (void) (j = 1) >>>>;
5
6 //.....
7 //for 循环改写为 if-else 分支和跳转语句 goto
8 {
9     int i = 0;
10
11     <<cleanup_point      int i = 0;>>;
12     goto <D.2366>;
13     <D.2365>;
14     <<cleanup_point <<< Unknown tree: expr_stmt
15 (void) (a[i] = i) >>>>;
16     <<cleanup_point <<< Unknown tree: expr_stmt
17 (void) (b[i] = i + 1) >>>>;
18     <<cleanup_point (void) i++ >>;
19     <D.2366>;
20     if (i <= 99) goto <D.2365>; else goto <D.2363>;

```

```

21     <D.2363>;
22 }

```

在“main.cpp.006t.gimple”中，**程序被改写为 gimple 语言**——一种三地址表示的中间语言，引入了临时变量来保存中间值。同时加入了 **try,finally** 等关键词将源代码中的异常处理结构表示为一系列的控制流操作。

gimple 是 gcc 编译器的一种中间表示，位于编译器的前端。它在源代码与底层机器代码之间起到了中间层的作用，允许编译器在此阶段进行许多优化和分析。gimple 简单的语法结构使得它相对容易理解和分析。这种简单性有助于编译器执行各种优化操作，如常量折叠、控制流分析和数据流分析。另外，gimple 文件通常是与目标平台无关的，这意味着它可以在不同的体系结构上重复使用。这有助于提高编译器的可移植性。

```

1  //main.cpp.006t.gimple
2
3  try
4  {
5      .....
6      //gimple 表示的 for 循环
7      {
8          int i;
9          i = 0;
10         goto <D.2366>;
11         <D.2365>:
12         a[i] = i;
13         _1 = i + 1;
14         b[i] = _1;
15         i = i + 1;
16         <D.2366>:
17         if (i <= 99) goto <D.2365>; else goto <D.2363>;
18         <D.2363>:
19     }
20     .....
21 }
22 finally
23 {
24     a = {CLOBBER};
25     b = {CLOBBER};
26     c = {CLOBBER};
27 }
28 .....

```

29

“main.cpp.015t.cfg”将程序表示为控制流图，并对 **dead code 等进行了优化**。在本实验的样例中，存在语句“if(5<4)...else...”，这将导致 else 分支成为 dead code。在之前的 pass 中，这一分支语句仍然保留，直到在此文件中将该分支语句直接表示为“y=1”。

控制流图通常由基本块组成，这可以在以下“main.cpp.015t.cfg”代码中看到。每个基本块是一组连续的代码，其中只有一个入口点和一个出口点。它由节点（表示基本块）和边（表示控制流转移）组成。节点表示代码块，而边表示代码块之间的控制流转移。控制流图可以与数据流图结合使用，以表示数据的流动和依赖关系。这有助于编译器进行数据流分析和优化。

```
1 //main.cpp.015t.cfg
2 //优化 dead code
3 <bb 3> :
4     y = 1;
5
6     <bb 4> :
7     j = 1;
8     i = 0;
9     //for 循环部分
10    goto <bb 6>; [INV]
11
12    <bb 5> :
13    a[i] = i;
14    _1 = i + 1;
15    b[i] = _1;
16    i = i + 1;
17
18    <bb 6> :
19    if (i <= 99)
20        goto <bb 5>; [INV]
21    else
22        goto <bb 7>; [INV]
```

3.2.3 中间代码生成与优化（二）

这一部分通过“g++ main.cpp -fdump-rtl-all-graph -o main ”指令获得中间代码生成的多阶段的输出。该部分生成的文本文件可读性较差，.dot 文件可读性较好。

分析各阶段产生的.dot 文件，可以发现他们都是**基于汇编语言的控制流图**，各个文件间差异较小，较明显的一点是起初以“r77”，“r89”等指代的寄存器在最终替换为了“ax”，“bp”等 x86 架构下具体的寄存器。

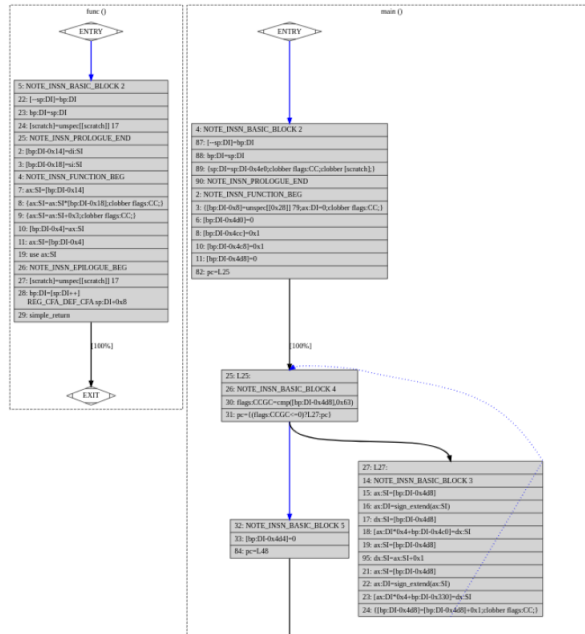


图 3: -fdump-rtl-all-graph 产生的一个.dot 文件可视化

3.3 编译器——开启优化选项

在上一节中，我们使用没有优化选项的指令进行编译，并分析了中间文件。这一节中，我们将测试-O1 以及-O2 优化选项对中间文件的影响。

3.3.1 -O1 优化

使用“g++ main.cpp -O1 -fdump-tree-all-graph -o main”指令编译 main.cpp。加入-O1 优化选项后，生成的中间文件数量明显增多，约有 200 个，这是由于要进行更多的优化工作，如图4

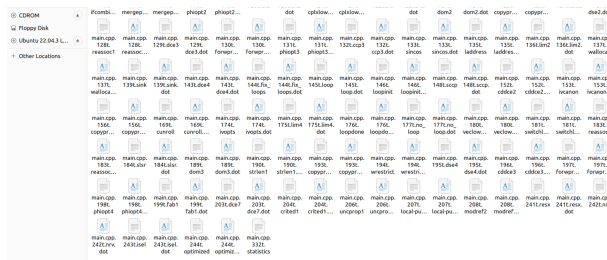


图 4: 加入-O1 选项后-fdump-tree-all-graph 产生的中间文件

分析最终优化产生的控制流图“main.cpp.244t.optimized.dot”(如图5,6), 并与上一节的控制流图进行对比，可以发现：

- 程序开头删去了无用的三个变量声明语句
- 函数 func 变为内联函数
- 由于函数参数为程序开头声明的常量，进行常量传播后内联后的函数体进一步简化
- 计算出了每个分支跳转的概率，以优化分支预测

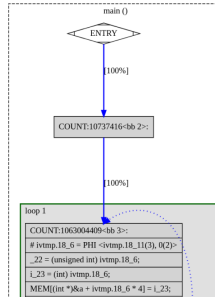


图 5: 文件开头删去无用变量声明语句

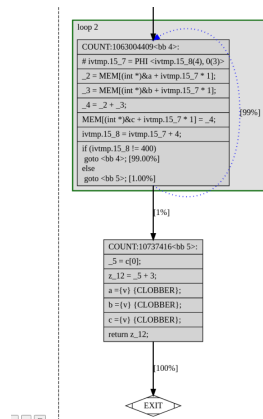


图 6: 内联函数, 以及计算出了分支跳转的概率

逐一分析所生成的中间文件,可以发现分支跳转的概率计算在“main.cpp.048t.profile_estimate”文件所对应的步骤完成,该步骤计算了每个基本块之间的转移概率,如图7

```

53 Predictions for bb 2
54 1 edges in bb 2 predicted to even probabilities
55 Predictions for bb 3
56 1 edges in bb 3 predicted to even probabilities
57 Predictions for bb 4
58 first match heuristics: 99.00%
59 combined heuristics: 99.00%
60 loop iterations heuristics of edge 4->8: 1.00%
61 Predictions for bb 8
62 1 edges in bb 8 predicted to even probabilities
63 Predictions for bb 5
64 1 edges in bb 5 predicted to even probabilities
65 Predictions for bb 6
66 first match heuristics: 99.00%
67 combined heuristics: 99.00%
68 loop iterations heuristics of edge 6->7: 1.00%
69 Predictions for bb 7
70 1 edges in bb 7 predicted to even probabilities
71 cyclic probability of bb 4 is 0.990000; turning freq 1.000000 to 100.000021
72 cyclic probability of bb 6 is 0.990000; turning freq 1.000000 to 100.000021

```

图 7: 分支跳转概率计算

3.3.2 -O3 优化

使用“g++ main.cpp -O3 -fdump-tree-all-graph -fdump-rtl-all-graph -o main”指令编译main.cpp, 获取 tree 和 rtl 两阶段的中间文件。

分析“-fdump-tree-all-graph”指令产生的最后一轮优化后的文件“main.cpp.244t.optimized”(如图8), 可以发现其基本结构与-O1 优化时相似, 但使用了向量寄存器。

```

40
41 <bb 3> [local count: 268435396]:
42 # vect_vec_iv_16_36 = PHI < 37(3), { 0, 1, 2, 3 }(2)>
43 # ivtmp_36_1 = PHI <ivtmp_36_21(3), 0(2)>
44 37 = vect_vec_iv_16_36 + { 4, 4, 4, 4 };
45 MEM <vector(4) int> [(int *)&a + ivtmp_36_1 * 1] = vect_vec_iv_16_36;
46 vect_1_19_42 = vect_vec_iv_16_36 + { 1, 1, 1, 1 };
47 MEM <vector(4) int> [(int *)&b + ivtmp_36_1 * 1] = vect_1_19_42;
48 ivtmp_36_21 = ivtmp_36_1 + 16;
49 if (ivtmp_36_21 != 400)
50 goto <bb 3>; [96.00%]
51 else
52 goto <bb 4>; [4.00%]
53
54 <bb 4> [local count: 268435325]:

```

图 8: main.cpp.244t.optimized 中的一个 for 循环

为进一步了解自动向量化相关信息，我们分析了“-fdump-rtl-all-graph”指令产生的控制流图“main.cpp.319r.alignments.dot”中相应的 for 循环（如图9），可以发现它使用了 **SSE 指令集中的 xmm 寄存器，进行 4 路 SIMD 并行化**。由于并行化减少了 for 循环的次数，故分支跳转的概率也发生了变化。

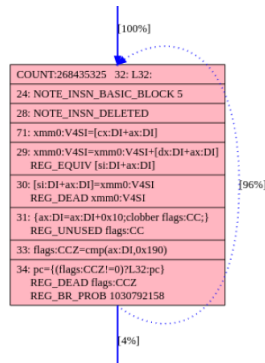


图 9: main.cpp.319r.alignments.dot 中的一个 for 循环

3.4 汇编器

使用指令“g++ main.i -S -o main.S”生成 x86 格式目标代码，再使用“g++ main.S -c -o main.o”生成二进制目标文件。由于 main.o 由机器码组成，无法解读，因此使用 objdump 进行反汇编，得到“main-anti-obj.S”。将其与 g++ 编译器生成的目标文件 main.S 对比，可以发现汇编器将源文件转换为机器码，并放在文件的特定地址（如图10）。

```

main.S  x  main-anti-obj.S  x  main.i  x
1
2 main.o:      file format elf64-x86-64
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <.Z4funcii>:
8 0: f3 0f 1e fa      endbr64
9 4: 55               push    %rbp
10 5: 48 89 e5         mov     %rsp,%rbp
11 8: 89 7d ec         mov     %edi,-0x14(%rbp)
12 b: 89 75 e8         mov     %esi,-0x18(%rbp)
13 e: 8b 45 ec         mov     -0x14(%rbp),%eax
14 11: 0f af 45 e8      imul    -0x18(%rbp),%eax
15 15: 83 c0 03         add     $0x3,%eax
16 18: 89 45 fc         mov     %eax,-0x4(%rbp)
17 1b: 8b 45 fc         mov     -0x4(%rbp),%eax
18 1e: 5d               pop     %rbp
19 1f: c3               ret
20

```

图 10: 使用 objdump 对 main.o 进行反汇编

使用 Linux 自带的 nm 工具查看 main.o，可以得到其中各段的信息（如图11）。由此可知汇编器将不同函数的机器码以及符号信息等放入了不同的段中。通过查阅相关资料可知，其中 T 代表代码段的数据，b 代表未被初始化的全局数据，U 代表符号未定义。

```

1      U _cxa_atexit
2      U _dso_handle
3      U _GLOBAL_OFFSET_TABLE_
4 0000000000000189 t _GLOBAL_sub_I_Z4funcii
5 0000000000000020 T main
6      U _stack_chk_fail
7 0000000000000133 t _Z41__static_initialization_and_destruction_0ii
8 0000000000000000 T _Z4funcii
9      U _ZNSt8ios_base4InitC1Ev
10     U _ZNSt8ios_base4InitD1Ev
11 0000000000000000 b _ZStL8__ioinit

```

图 11: 使用 nm 查看 main.o

综上可以总结出汇编器做了以下一些工作：

- 对源文件进行词法分析、语法分析，生成机器码指令
- 形成代码段、数据段等
- 把所有的段按照 elf 文件的格式组装起来，形成可重定位目标文件

3.5 链接器

链接是编译过程中的最后一个阶段，其主要功能是将编译器生成的目标文件（通常是机器码文件）合并成一个可执行文件或共享库。链接器的主要功能包括符号解析、符号重定位、生成符号表、处理库文件、生成可执行文件或共享库等。

gcc 默认编译选项下进行动态链接，产生的可执行文件在程序运行时加载共享库。为与静态链接对比，我们采用以下两种指令进行编译：

- g++ main.cpp -o main_dynamic
- g++ main.cpp -static -o main_static

使用 objdump 对产生的可执行文件进行反汇编,发现动态链接产生的可执行文件“main_dynamic-anti-exe.S”规模较小，只有**两百多行**汇编代码。而静态链接对应的“main_static-anti-exe.S”规模较大，约有 **40 万行**（如图12）。这是因为静态链接时将需要的二进制代码都“拷贝”到可执行文件中，而动态链接时仅仅“拷贝”一些重定位和符号表信息，这些信息可以在程序运行时完成真正的链接过程。



图 12: 两种链接方式对应反汇编文件的规模对比

4 llvm 编译器的各个处理阶段

在这里将对两个源程序进行分析，第一个源程序为简单的斐波那契数列程序，在这里将对此程序分析 llvm 的所有阶段，第二个程序添加丰富的语言特性，以便观察代码优化阶段 llvm 的处理过程，只对第二个程序分析代码优化部分。采取的平台为 Vscode+llvm 编译器 +ubuntu linux 虚拟机。

1 fibo.cpp

```
1  #include<iostream>
2  using namespace std;
3  #define z 0
4  //斐波那契数列
5  int main(){
6      int a,b,i,t,n;
7      a=z;
8      b=1;
9      i=1;
10     cin>>n;
11     cout<<a<<endl;
12     cout<<b<<endl;
13     while(i<n){ //循环部分
14         t=b;
15         b=a+b;
16         cout<<b<<endl;
17         a=t;
18         i=i+1;
19     }
20     return 0;
21 }
```

2 sample.cpp

```
1  #include<iostream>
2  using namespace std;
3  #define z 0;
4  //-dse 函数内部没有用到 dd, 会被删掉
5  int foo(int aa, int bb, int cc,int dd){
6      int sum = aa + bb;
7      return sum/cc;
8  }
9  int main(){
10     int m,n;
11     int i,j;
12     //实际上 j 在后面没有用到
```

```
13      // -dse
14      i=z;
15      m=i;
16      n=m;
17      // 优化
18      if(4<5){
19          i=5;
20      }
21      if(i<10){
22          cout<<i;
23      }
24      for(i=2;i*i<1000;i++){
25          m++;
26      }
27      // 自动向量化
28      int a[100],b[100],c[100];
29      for(i=0;i<100;i++){
30          a[i]=i;
31          b[i]=i+1;
32      }
33      for(i=0;i<100;i++){
34          c[i]=a[i]+b[i];
35      }
36      // mem2reg
37      cout<<foo(1,2,3,4)<<m<<c[50];
38      return 0;
39  }
```

4.1 预处理器

预处理阶段的工作仅仅是将头文件，宏文件，条件编译部分复制到对应位置，删除注释部分。这里我采取如下命令:clang -E fibo.cpp -o fibo.i

值得注意的是，对于头文件而言，若属于库文件，则须以 `<>` 去包含此文件，若是自己定义的文件，则须以 `"` 包含。

从下图分析: 源程序 fibo.cpp 最初为 21 行，而在经过预处理之后，将庞大的头文件的代码包含进去，导致 fibo.i 文件有 30198 行。而在 # 2 "fibo.cpp" 2 后面与源代码几乎一致，只是将宏定义的 `z` 替换为常量 0，删去注释部分。

此外，`"ifdef XXX"` 等条件编译也会在此阶段进行替换。

```

30178 # 2 "fibonacci.cpp" 2
30179 using namespace std;
30180
30181
30182 int main(){
30183     int a,b,i,t,n;
30184     a=0;
30185     b=1;
30186     i=1;
30187     cin>>n;
30188     cout<<a<<endl;
30189     cout<<b<<endl;
30190     while(i<n){
30191         t=b;
30192         b=a+b;
30193         cout<<b<<endl;
30194         a=t;
30195         i=i+1;
30196     }
30197     return 0;
30198 }
30199

```

图 13: fibo.i 结果

4.2 编译器

4.2.1 词法分析

在这个阶段将通过分词器将源程序按照一定逻辑分词，且分出它的基本组成单元。这里我使用如下命令行:clang -E -Xclang -dump-tokens fibonacci.cpp > tokens.txt 2>&1。具体分词部分如下图所示

```

126691 namespace 'namespace' [LeadingSpace] Loc=<fibonacci.cpp:2:7>
126692 identifier 'std' [LeadingSpace] Loc=<fibonacci.cpp:2:17>
126693 semi ';' Loc=<fibonacci.cpp:2:20>
126694 int 'int' [StartOfLine] Loc=<fibonacci.cpp:3:1>
126695 identifier 'main' [LeadingSpace] Loc=<fibonacci.cpp:3:5>
126696 l_paren '(' Loc=<fibonacci.cpp:3:9>
126697 r_paren ')' Loc=<fibonacci.cpp:3:10>
126698 l_brace '{' Loc=<fibonacci.cpp:3:11>
126699 int 'int' [StartOfLine] [LeadingSpace] Loc=<fibonacci.cpp:4:5>
126700 identifier 'a' [LeadingSpace] Loc=<fibonacci.cpp:4:9>
126701 comma ',' Loc=<fibonacci.cpp:4:10>
126702 identifier 'b' Loc=<fibonacci.cpp:4:11>
126703 comma ',' Loc=<fibonacci.cpp:4:12>
126704 identifier 'i' Loc=<fibonacci.cpp:4:13>
126705 comma ',' Loc=<fibonacci.cpp:4:14>
126706 identifier 't' Loc=<fibonacci.cpp:4:15>
126707 comma ',' Loc=<fibonacci.cpp:4:16>
126708 identifier 'n' Loc=<fibonacci.cpp:4:17>
126709 semi ';' Loc=<fibonacci.cpp:4:18>
126710 identifier 'a' [StartOfLine] [LeadingSpace] Loc=<fibonacci.cpp:5:5>
126711 equal '=' Loc=<fibonacci.cpp:5:6>
126712 numeric_constant '0' Loc=<fibonacci.cpp:5:7>
126713 semi ';' Loc=<fibonacci.cpp:5:8>
126714 identifier 'b' [StartOfLine] [LeadingSpace] Loc=<fibonacci.cpp:6:5>
126715 equal '=' Loc=<fibonacci.cpp:6:6>
126716 numeric_constant '1' Loc=<fibonacci.cpp:6:7>
126717 semi ';' Loc=<fibonacci.cpp:6:8>
126718 identifier 'i' [StartOfLine] [LeadingSpace] Loc=<fibonacci.cpp:7:5>
126719 equal '=' Loc=<fibonacci.cpp:7:6>
126720 numeric_constant '1' Loc=<fibonacci.cpp:7:7>
126721 semi ';' Loc=<fibonacci.cpp:7:8>
126722 identifier 'cin' [StartOfLine] [LeadingSpace] Loc=<fibonacci.cpp:8:5>

```

图 14: tokens 部分示例

对得到的 tokens 序列进行分析，我们可以发现分词器对源程序是按照一定逻辑进行分词的。

如上图引号左侧的 `ci` 的词语是其逻辑类别。而在最右侧的是分词的词语所在源程序的位置。

我们可以解读一下这些分词的逻辑类别：`namespace` 和 `int` 是关键字, `identifier` 表示标识符, `semi` 表示分号, `numeric_constant` 表示数字, `equal` 表示等号。可以发现, 分词器按照 `c++` 的语言逻辑进行分词, 并且把词语的逻辑类别与所在位置整理好, 这有利于编译器进行下一步的相关工作与处理。

当对代码进行词法分析时, 将代码拆分为不同的标记或词法单元是非常重要的。这有助于创建语法树和执行语义分析。根据上图的 `tokens` 序列, 可以得出如下的逻辑类别, 这些类别用以标记单词的类型。

- `namespace`: 关键字, 用于指定命名空间。
- `identifier`: 标识符, 用于表示变量、函数或其他实体的名称。
- `semi`: 分号, 用于表示语句的结束。
- `int`: 关键字, 表示整数类型。
- `equal`: 等号, 用于将一个值赋给变量。
- `numeric_constant`: 数字常量, 表示整数值。

可以发现, 这些逻辑类别是根据代码语法和规则定义的, 它们帮助编译器理解代码的结构和含义。分词器按照 `c++` 的语言逻辑进行分词, 并且把词语的逻辑类别与所在位置整理好, 这有利于编译器进行下一步的语法分析和语义分析, 以检查代码的正确性并生成执行代码。

4.2.2 语法分析

在此阶段通过词法分析生成的词法单元来构建抽象语法树 (ast), 这里我使用如下命令行: `clang -E -Xclang -ast-dump fibo.cpp > ast_dump.txt 2>&1`

```

|-UsingDirectiveDecl 0x1d3d5d8 <fibonacci.cpp:2:1, col:17> col:17 Namespace 0x1d27ca50 'std'
^-FunctionDecl 0x1d3d658 <line:3:1, line:19:1> line:3:5 main 'int ()'
  ^-CompoundStmt 0x1d4e458 <col:11, line:19:1>
    |-DeclStmt 0x1d3d9a8 <line:4:5, col:18>
      |-VarDecl 0x1d3d710 <col:5, col:9> col:9 used a 'int'
      |-VarDecl 0x1d3d790 <col:5, col:11> col:11 used b 'int'
      |-VarDecl 0x1d3d810 <col:5, col:13> col:13 used i 'int'
      |-VarDecl 0x1d3d890 <col:5, col:15> col:15 used t 'int'
      ^-VarDecl 0x1d3d910 <col:5, col:17> col:17 used n 'int'
      |-BinaryOperator 0x1d3da00 <line:5:5, col:7> 'int' lvalue '='
      |-DeclRefExpr 0x1d3d9c0 <col:5> 'int' lvalue Var 0x1d3d710 'a' 'int'
      ^-IntegerLiteral 0x1d3d9e0 <col:7> 'int' 0
      |-BinaryOperator 0x1d3da60 <line:6:5, col:7> 'int' lvalue '='
      |-DeclRefExpr 0x1d3da20 <col:5> 'int' lvalue Var 0x1d3d790 'b' 'int'
      ^-IntegerLiteral 0x1d3da40 <col:7> 'int' 1
      |-BinaryOperator 0x1d3dac0 <line:7:5, col:7> 'int' lvalue '='
      |-DeclRefExpr 0x1d3da80 <col:5> 'int' lvalue Var 0x1d3d810 'i' 'int'
      ^-IntegerLiteral 0x1d3daa0 <col:7> 'int' 1
      |-CXXOperatorCallExpr 0x1d41600 <line:8:5, col:10> 'std::basic_istream<char>::__istream_type'
      |-ImplicitCastExpr 0x1d415e8 <col:8> 'std::basic_istream<char>::__istream_type &(*) (int &)'
      |-DeclRefExpr 0x1d41570 <col:8> 'std::basic_istream<char>::__istream_type &(int &)' lvalue
      |-DeclRefExpr 0x1d3dae0 <col:5> 'std::istream': 'std::basic_istream<char>' lvalue Var 0x1d3c
      ^-DeclRefExpr 0x1d3db00 <col:10> 'int' lvalue Var 0x1d3d910 'n' 'int'
      |-CXXOperatorCallExpr 0x1d4b230 <line:9:5, col:14> 'std::basic_ostream<char>::__ostream_type'
      |-ImplicitCastExpr 0x1d4b218 <col:12> 'std::basic_ostream<char>::__ostream_type &(*) (std::b
      ^-DeclRefExpr 0x1d4b198 <col:12> 'std::basic_ostream<char>::__ostream_type &(std::basic_c
      |-CXXOperatorCallExpr 0x1d4a700 <col:5, col:11> 'std::basic_ostream<char>::__ostream_type':
      |-ImplicitCastExpr 0x1d4a6e8 <col:9> 'std::basic_ostream<char>::__ostream_type &(*) (int)'
      ^-DeclRefExpr 0x1d4a668 <col:9> 'std::basic_ostream<char>::__ostream_type &(int)' lvalue
      ^-DeclRefExpr 0x1d41828 <col:5> 'std::ostream': 'std::basic_ostream<char>' lvalue Var 0x1d
      ^-ImplicitCastExpr 0x1d4a650 <col:11> 'int' <LValueToRValue>
      ^-DeclRefExpr 0x1d41848 <col:11> 'int' lvalue Var 0x1d3d710 'a' 'int'

```

图 15: ast 部分语法树

Clang AST 的最顶层结构叫做 TranslationUnit，它被叫做“编译单元”。它的子节点前面跟了很多个 TypedefDecl，这些都是 Clang 的内置定义。然后上图中显示的关于 main 函数对应的 FunctionDecl，是我们要重点分析的对象。

下面简单介绍 ast 内部各种节点的意义。

- FunctionDecl 节点对应我们的 main 函数
- ParmVarDecl 节点对应的是函数的参数。
- CompoundStmt 对应的是函数体，也就是函数内部的各种语句等等。此节点的孩子节点对应函数内部的各种语句变量，以及最后的返回语句。
- VarDecl 节点对应着函数内部定义的各种变量 variables。
- ReturnStmt 节点对应着函数的返回语句

可以发现 Clang AST 中两个最基本的节点是语句 (Stmt) 和声明 (Decl)。

4.2.3 中间代码生成

在这里，llvm 的中间代码生成即 llvm IR。而针对 llvm IR 有文本格式.ll 与二进制编码格式.bc 文件。

1. 遇到问题: 我使用的中间代码生成的命令行为:clang -S -emit-llvm fibo.cpp -o fibo.ll。但是，在后来的代码优化阶段，我发现使用任何的 passes 都无法优化源程序，即便我设计的代码是专门针对 passes 设计的。

2. 解决问题: 使用的中间代码命令行为:clang -S -emit-llvm -O0 -Xclang -disable-O0-optnone fibo.cpp -o fibo.ll

值得注意的是,为了体现代码优化后的.ll 文件与原有.ll 文件的差异,这里我们首先要添加-O0 选项,其次要添加-disable-O0-optnone 选项。

在设置-O0 编译时,clang 会对每个函数设置 optnone 属性,这会阻止之后使用任何的优化,所以我们必须添加-disable-O0-optnone 选项,这样才能在代码优化阶段对源程序进行优化,体现出.ll 文件的不同。所以现在,我们的命令行要添加-disable-O0-optnone 选项。

```
define dso_local noundef i32 @main() #4 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    %5 = alloca i32, align 4
    %6 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    store i32 0, i32* %2, align 4
    store i32 1, i32* %3, align 4
    store i32 1, i32* %4, align 4
    %7 = call noundef nonnull align 8 dereferenceable(16) @"class.std::basic_istrea
    %8 = load i32, i32* %2, align 4
    %9 = call noundef nonnull align 8 dereferenceable(8) @"class.std::basic_ostrea
    %10 = call noundef nonnull align 8 dereferenceable(8) @"class.std::basic_ostrea
    %11 = load i32, i32* %3, align 4
    %12 = call noundef nonnull align 8 dereferenceable(8) @"class.std::basic_ostrea
    %13 = call noundef nonnull align 8 dereferenceable(8) @"class.std::basic_ostrea
    br label %14
```

图 16: llvm ir 的部分文本形式代码

4.2.4 代码优化的介绍

在 llvm 的代码优化里,有众多现有的优化 passes。这些 passes 可以分为三类:analysis passes、transform passes、utility passes。由于 transform passes 会对中间代码形式的程序进行变化,所以在这里我们主要探究 transform passes 对程序的优化。

在下面的实践中,采取 opt -<module name> -S <test.ll> -o <test-optimized.ll> 的命令,分析特定的优化阶段上,llvm 所起的优化作用。

下面介绍几种我主要研究的优化 passes(实践中有实际效果的):

- mem2reg: 非常重要的优化 pass, 将程序中的局部内存变量(内存引用)转化为寄存器变量,从而提高程序的性能。
- early-cse: 旨在消除在编译器生成的中间表示(IR)中的公共子表达式,以减少冗余的计算和提高代码的执行效率,有助于生成更高效的代码。
- dse、dce: dse 会消除不必要的存储操作,而 dce 会消除不可达、无效或不会被执行的代码
- instcombine: 其目标是将 IR 中的一些指令组合成更简洁和高效的形式,以便后续的编译器优化能够更容易地操作和生成更好的目标代码
- loop-extract: 通常在其他更高级别的优化之前运行,以减少源函数的复杂性并提高代码的模块化程度。这对于大型项目和复杂函数来说特别有用,可以更容易地对循环进行局部优化。这可以包括循环展开、向量化、循环分块等。

- loop-rotate: 主要用于改善循环的性能。通过改变循环的基本块顺序, 它可以减少分支预测失败、改善内存访问模式或提高指令级并行性, 从而使循环更加高效。
- loop-unroll: 须与 loop-extract 结合起来用, 更容易对循环进行展开等操作。
- jump-threading: 将条件分支的控制流进行优化, 以减少不必要的分支指令, 从而提高程序的性能。

4.2.5 代码优化的程序对比

1.mem2reg

命令行: `opt -mem2reg -S fibo.ll -o test-fibo.ll`

针对 fibo.cpp 的

```
int a,b,i,t,n;
a=z;
b=1;
i=1;
cin>>n;
while(i<n){ //循环部分
    ...
}
```

源程序的 ll 文件

```
1  %1 = alloca i32, align 4
2  %2 = alloca i32, align 4
3  %3 = alloca i32, align 4
4  %4 = alloca i32, align 4
5  %5 = alloca i32, align 4
6  %6 = alloca i32, align 4
7  store i32 0, i32* %1, align 4
8  store i32 0, i32* %2, align 4
9  store i32 1, i32* %3, align 4
10 store i32 1, i32* %4, align 4
11 ...
12 br label %14
13
14 14:                                     ; preds = %18, %0
15 %15 = load i32, i32* %4, align 4
16 %16 = load i32, i32* %6, align 4
17 %17 = icmp slt i32 %15, %16
18 br i1 %17, label %18, label %29
19 ...
```

优化后的 ll 文件

```

1  %1 = alloca i32, align 4
2  ...
3  br label %7
4
5  7:                                     ; preds = %10, %0
6  %.02 = phi i32 [ 1, %0 ], [ %11, %10 ]
7  %.01 = phi i32 [ 0, %0 ], [ %.02, %10 ]
8  %.0 = phi i32 [ 1, %0 ], [ %14, %10 ]
9  %8 = load i32, i32* %1, align 4
10 %9 = icmp slt i32 %.0, %8
11 br i1 %9, label %10, label %15
12 ...

```

将程序中的局部内存变量提升为寄存器变量，这一阶段的优化对后续其他阶段有着基础性的作用。比如之后的 loop-unroll 必须建立在此阶段上。只有将内存变量提升为寄存器变量后，才能以应用其他代码优化。

在 mem2reg 优化后，可以看出 int a,b,i,t,n; 这些定义的局部变量除了 n 外，全部被删去。这里我认为由于 n 是需要输入的一个局部变量，所以只保留了一个局部变量。在源程序 cout<<a 与 cout<<b 两行代码中，优化之后直接把这两个变量给删掉，直接输出数字的值。所以可以看出在 mem2reg 阶段进行了寄存器分配和优化，将局部变量存储在寄存器中。

此外，优化后的代码有 phi 指令，它用于合并不同路径上的值。在这个示例中，%.02、%.01 和 %.0 是根据控制流的不同路径而来的不同值，优化后的代码中使用了 phi 指令来实现循环变量的更新，使得代码更加紧凑和高效。

同时大部分 alloca 指令被消除了，因为 LLVM 编译器进行了寄存器分配和优化，将局部变量存储在寄存器中，而不是栈上。这种方式允许在不同的基本块之间传递变量的值，而无需使用显式的加载和存储操作，因此减少了 load 等指令，也可以发现优化后的 fibo.ll 代码减少了 20 行。

2.early-cse

命令行:opt -early-cse -S sample.ll -o test-sample.ll

针对 sample.cpp 前面定义局部变量与数组的部分分析

源程序的 ll 代码

```

1  %1 = alloca i32, align 4
2  %2 = alloca i32, align 4
3  %3 = alloca i32, align 4
4  %4 = alloca i32, align 4
5  %5 = alloca i32, align 4
6  %6 = alloca [100 x i32], align 16
7  %7 = alloca [100 x i32], align 16
8  %8 = alloca [100 x i32], align 16
9  store i32 0, i32* %1, align 4
10 store i32 0, i32* %4, align 4

```

```

11  %9 = load i32, i32* %4, align 4
12  store i32 %9, i32* %2, align 4
13  %10 = load i32, i32* %2, align 4
14  store i32 %10, i32* %3, align 4
15  store i32 5, i32* %4, align 4
16  %11 = load i32, i32* %4, align 4
17  %12 = icmp slt i32 %11, 10
18  br i1 %12, label %13, label %16

```

优化后的 ll 代码

```

1  %1 = alloca i32, align 4
2  %2 = alloca i32, align 4
3  %3 = alloca i32, align 4
4  %4 = alloca i32, align 4
5  %5 = alloca [100 x i32], align 16
6  %6 = alloca [100 x i32], align 16
7  %7 = alloca [100 x i32], align 16
8  store i32 0, i32* %1, align 4
9  store i32 0, i32* %4, align 4
10 store i32 0, i32* %2, align 4
11 store i32 0, i32* %3, align 4
12 store i32 5, i32* %4, align 4
13 br i1 true, label %8, label %10

```

- 定义局部变量的语句，为其申请内存的语句，以及为局部变量赋值的语句都非常整齐，同时删去了一个不会用到的局部变量 j。
- 直接将对应数字常量赋值给对应局部变量，而不是像源代码一样将局部变量的值去赋值给源代码，这样代码的逻辑更清晰，执行效率更高效

3.dse、dce

命令行: `opt -dse -dce -S sample.ll -o test-sample.ll`

源程序的 ll 代码

```

1  %5 = alloca i32, align 4
2  %6 = alloca i32, align 4
3  %7 = alloca i32, align 4
4  %8 = alloca i32, align 4
5  %9 = alloca i32, align 4
6  ...
7  ret i32 %15

```

优化后的 ll 代码

```

1  %5 = alloca i32, align 4
2  %6 = alloca i32, align 4
3  %7 = alloca i32, align 4
4  %8 = alloca i32, align 43
5  ...
6  ret i32 %14

```

针对 sample.cpp 里面的 foo 函数，可以发现参数 dd 在函数体中未用到。在下面优化的代码中，很明显的看到少分配了一个局部变量，即删去了不必要的存储变量。

4.instcombine

命令行: `opt -instcombine -S sample.ll -o test-sample.ll`

针对 sample.cpp 的如下部分分析

```

int m,n;
int i,j;
i=z;
m=i;
n=m;
if(4<5){
    i=5;
}
if(i<10){
    cout<<i;
}

```

原来的 ll 代码

```

1  %1 = alloca i32, align 4
2  %2 = alloca i32, align 4
3  %3 = alloca i32, align 4
4  %4 = alloca i32, align 4
5  %5 = alloca i32, align 4
6  %6 = alloca [100 x i32], align 16
7  %7 = alloca [100 x i32], align 16
8  %8 = alloca [100 x i32], align 16
9  store i32 0, i32* %1, align 4
10 store i32 0, i32* %4, align 4
11 %9 = load i32, i32* %4, align 4
12 store i32 %9, i32* %2, align 4
13 %10 = load i32, i32* %2, align 4
14 store i32 %10, i32* %3, align 4
15 store i32 5, i32* %4, align 4
16 %11 = load i32, i32* %4, align 4

```

```

17 %12 = icmp slt i32 %11, 10
18 br i1 %12, label %13, label %16

```

优化后的 ll 代码

```

1 %1 = alloca i32, align 4
2 %2 = alloca i32, align 4
3 %3 = alloca [100 x i32], align 16
4 %4 = alloca [100 x i32], align 16
5 %5 = alloca [100 x i32], align 16
6 store i32 0, i32* %2, align 4
7 store i32 0, i32* %1, align 4
8 store i32 5, i32* %2, align 4
9 br i1 true, label %6, label %9

```

- 可以发现此优化阶段首先会删除不必要的局部变量，优化后删去了 3 个局部变量的定义。观察 sample.cpp 源代码可以发现只有 i 和 m 两个局部变量有用，这说明 instcombine 优化很有效。
- 可以发现源代码中的条件分支 if(4<5) 直接被删除，而 i 直接被赋值为 5；同时，由于 5<10，无条件跳转到%6 代码块，说明 instcombine 阶段会分析条件分支的条件语句，若为条件可被判断真假，会直接删去跳转语句 (或者将跳转条件更改为 true/false)。

综上，可以发现 instcombine 这一阶段，对指令的分析与指令的重组有着巨大的作用，此外还会对跳转语句的条件进行相关分析判断，可能将跳转语句删除。

5.loop-extract

命令行:opt -loop-extract -S sample.ll -o test-sample.ll

这里针对 sample.cpp 进行分析

```

1 codeRepl2:                                ; preds = %16
2   call void @main.extracted.2(i32* %4, i32* %2)
3   br label %17
4 17:                                         ; preds = %codeRepl2
5   store i32 0, i32* %4, align 4
6   br label %codeRepl1
7 codeRepl1:                                ; preds = %17
8   call void @main.extracted.1(i32* %4, [100 x i32]* %6, [100 x i32]* %7)
9   br label %18
10 18:                                        ; preds = %codeRepl1
11   store i32 0, i32* %4, align 4
12   br label %codeRepl
13 codeRepl:                                 ; preds = %18
14   call void @main.extracted(i32* %4, [100 x i32]* %6, [100 x i32]* %7, [100 x i32]* %8)
15   br label %19

```

如上面的源代码所示, loop-extract 将 sample.cpp 中的三个循环语句提取出来, 定义了 main.extracted.1, main.extracted.1, main.extracted.2 函数。这一阶段的分析将有利于之后的 loop-rotate 与 loop-unroll 等等, 即以更容易地对循环进行局部优化。这可以包括循环展开、向量化、循环分块等。

6.loop-unroll

经过研究发现经过 loop-unroll 优化前, 必须要首先经过 mem2reg、loop-extract 优化。这两个阶段应该是 loop-unroll 的先决条件。

- `opt -mem2reg -loop-extract -S sample.ll -o test-sample2.ll`
- `opt -mem2reg -loop-extract -loop-unroll -S sample.ll -o test-sample.ll`

针对 sample.cpp 中下部所示的循环, 进行探究

```
for(i=0;i<100;i++){
    c[i]=a[i]+b[i];
}
```

优化后的对应.ll 代码

```
1 5:                                     ; preds = %3
2  %6 = sext i32 %.2 to i64
3  %7 = getelementptr inbounds [100 x i32], [100 x i32]* %0, i64 0, i64 %6
4  %8 = load i32, i32* %7, align 4
5  %9 = sext i32 %.2 to i64
6  %10 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %9
7  %11 = load i32, i32* %10, align 4
8  %12 = add nsw i32 %8, %11
9  %13 = sext i32 %.2 to i64
10 %14 = getelementptr inbounds [100 x i32], [100 x i32]* %2, i64 0, i64 %13
11 store i32 %12, i32* %14, align 4
12 br label %15
13
14 15:                                     ; preds = %5
15 %16 = add nuw nsw i32 %.2, 1
16 br label %17
17
18 17:                                     ; preds = %15
19 %18 = sext i32 %16 to i64
20 %19 = getelementptr inbounds [100 x i32], [100 x i32]* %0, i64 0, i64 %18
21 %20 = load i32, i32* %19, align 4
22 %21 = sext i32 %16 to i64
23 %22 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %21
24 %23 = load i32, i32* %22, align 4
25 %24 = add nsw i32 %20, %23
```

```

26 %25 = sext i32 %16 to i64
27 %26 = getelementptr @inbounds [100 x i32], [100 x i32]* %2, i64 0, i64 %25
28 store i32 %24, i32* %26, align 4
29 br label %27
30
31 27:                                     ; preds = %17
32 %28 = add nuw nsw i32 %16, 1
33 br label %3, !llvm.loop !6
34
35 .exitStub:                             ; preds = %3
36 ret void

```

对比上述代码，可以发现循环展开后：在标签 5 和标签 17 处，而且它们的指令语句结构一模一样，展开了一次循环，而不是简单地增加迭代器的值并跳回到标签 3。展开后的循环计算了多次迭代的结果，因此它在不同的基本块中执行了相同的计算。这通常有助于更好地利用现代处理器的流水线和指令级并行性。

7.jump-threading

- `opt -mem2reg -S sample.ll -o test-sample.ll`
- `opt -mem2reg -jump-threading -S sample.ll -o test-sample2.ll`

根据上面的代码，我们对 jump-threading 阶段所做的事情进行具体的分析，这里源代码为 sample.cpp 中的

```

int m,n;
int i,j;i=z;m=i;n=m;
if(4<5){
    i=5;
}
if(i<10){
    cout<<i;
}

```

原来的.ll 代码

```

1 %1 = alloca [100 x i32], align 16
2 %2 = alloca [100 x i32], align 16
3 %3 = alloca [100 x i32], align 16
4 %4 = icmp slt i32 5, 10
5 br i1 %4, label %5, label %7
6 5:                                     ; preds = %0
7 %6 = call @noundef_nonnull_align_8_dereferenceable(8) @"class.std::basic_ostream"*
8 @_ZNSolsEi(@"class.std::basic_ostream"*noundef_nonnull_align_8_dereferenceable(8)
9 @_ZSt4cout, i32 noundef 5)
10 br label %7

```

```

11 7:                                     ; preds = %5, %0
12 br label %8

```

优化后的.ll 代码

```

1  %1 = alloca [100 x i32], align 16
2  %2 = alloca [100 x i32], align 16
3  %3 = alloca [100 x i32], align 16
4  %4 = call @noundef_nonnull_align_8_dereferenceable(8) @"class.std::basic_ostream"*
5  @_ZNSolsEi(@"class.std::basic_ostream"* @noundef_nonnull_align_8_dereferenceable(8)
6  @_ZSt4cout, i32 @noundef 5)
7  br label %5

```

在下面的优化代码中, 我们明显可以发现指令的顺序以及相关分支的跳转被优化了。在 sample.cpp 中的 if(4<5) 直接被删去, 而之后的 if(i<10) 此分支也被删去, 因为在前面 i 被赋值为 5, 此阶段编译器直接算出一些条件的真假, 将分支语句给删除, 而直接跳转至%5 的代码块, 即 br label %5。

很明显, 经过 jump-threading 后, 代码的可读性和简洁性得到了大大的提升。

8.O3 优化

命令行: opt -O3 -S sample.ll -o test-sample.ll, 针对 sample.cpp 的代码有如下优化

```

1  %0 = alloca [100 x i32], align 16
2  %1 = alloca [100 x i32], align 16
3  %2 = alloca [100 x i32], align 16
4  ....
5  %4 = bitcast [100 x i32]* %0 to <4 x i32>*store <4 x i32> <i32 0, i32 1, i32 2, i32 3>
6  , <4 x i32>* %4, align 16
7  %5 = bitcast [100 x i32]* %1 to <4 x i32>*store <4 x i32> <i32 1, i32 2, i32 3, i32 4>
8  , <4 x i32>* %5, align 16
9  store <4 x i32> <i32 1, i32 2, i32 3, i32 4>, <4 x i32>* %5, align 16
10 %6 = getelementptr inbounds [100 x i32], [100 x i32]* %0, i64 0, i64 4
11 %7 = bitcast i32* %6 to <4 x i32>*
12 store <4 x i32> <i32 4, i32 5, i32 6, i32 7>, <4 x i32>* %7, align 16
13 ...
14 %104 = add nsw <4 x i32> %wide.load40, %wide.load
15 %105 = bitcast [100 x i32]* %2 to <4 x i32>*
16 store <4 x i32> %104, <4 x i32>* %105, align 16

```

首先对于 O3 优化, sample.cpp 前面定义的一系列局部变量与无用的条件分支, 全部被删除。剩下的只有 a,b,c 三个数组的赋值与计算操作。可以说明 **O3 优化包含了 llvm 几乎所有的优化, 开展了自动向量化, 对数组的循环操作进行了完全展开, 即完全消除了循环, 消除了无用的条件语句, 消除了无用的变量**, 下面会对上面的部分 llvm ir 中间代码进行分析。

%4 和 %5 是将 %0 和 %1 数组强制转换为 $<4 \times i32>^*$ 指针的位转换。这些位转换允许将这些数组视为每次处理四个整数的向量。而 %6、%8、%10 等用于获取 %0 和 %1 数组中后续矢量化存储的元素指针。

而在为 %0、%1 数组赋值后，开始为 %2 数组赋值。%wide.load 和 %wide.load40 是从 %0 和 %1 数组中加载值的向量。%104 计算从 %0 和 %1 加载的向量的元素逐个相加之和，最后将加法的结果存储在 %2 数组中。由于一次只能处理数组的四个元素，所以之后还要继续重复上述操作。

综上所述，这段代码实际上是在两个数组 %0 和 %1 的元素上执行矢量化操作，并将结果存储在第三个数组 %2 中。它旨在利用矢量化指令和硬件中的并行性以获得更好的性能。

4.2.6 代码生成

运行如下命令：`llc fibo.ll -o fibo.S`，llvm 可以生成对应的目标代码 `llvm.S`

```
main:                                     # @main
.cfi_startproc
# %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $32, %rsp
    movl    $0, -24(%rbp)
    movl    $0, -12(%rbp)
    movl    $1, -4(%rbp)
    movl    $1, -8(%rbp)
    movq    _ZSt3cin@GOTPCREL(%rip), %rdi
    leaq    -16(%rbp), %rsi
    callq   _ZNSirsEri@PLT
    movl    -12(%rbp), %esi
    movq    _ZSt4cout@GOTPCREL(%rip), %rdi
    callq   _ZNSolsEi@PLT
    movq    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostre
    movq    %rax, %rdi
    callq   _ZNSolsEPFRSoS_E@PLT
    movl    -4(%rbp), %esi
    movq    _ZSt4cout@GOTPCREL(%rip), %rdi
    callq   _ZNSolsEi@PLT
```

图 17: 部分 fibo.S 代码

根据图17，我们发现代码生成器将中间语言 llvm ir 形式转换为了 x86-64 汇编语言，以供下一步汇编器的应用。

4.3 汇编器

这里使用命令：`llc fibo.bc -filetype=obj -o fibo.o`，即可运行 llvm 中的汇编器工具，将汇编文件 `fibo.bc` 翻译成机器程序，并打包成可重定位目标程序的 `o` 文件。

同时为了对 `o` 文件进行分析，这里采取反汇编工具 `objdump`，得出对应的汇编代码。下面给出 `fibo.S` 和 `fibo.o` 的反汇编的部分代码图片。

经过分析图18与图19，查阅相关资料，我们可以简要概括汇编器在此阶段的功能：

- 汇编器首先会解析.S 文件中的汇编语法，包括指令、标签、操作数等。它会识别和理解汇编代码中的不同部分。
- 符号解析：汇编器会解析和处理汇编代码中的符号（如函数名、变量名等），并将它们与实际的内存地址或偏移量关联起来。这些符号包括 `main` 等标签。

```

main:                                     # @main
.cfi_startproc
# %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
subq    $32, %rsp
movl    $0, -24(%rbp)
movl    $0, -12(%rbp)
movl    $1, -4(%rbp)
movl    $1, -8(%rbp)
movq    _ZSt3cin@GOTPCREL(%rip), %rdi
leaq    -16(%rbp), %rsi
callq   _ZNSirsEri@PLT
movl    -12(%rbp), %esi
movq    _ZSt4cout@GOTPCREL(%rip), %rdi
callq   _ZNSolsEi@PLT
movq    _ZSt4endlc@GOTPCREL(%rip), %rdi
movq    %rax, %rdi
callq   _ZNSolsEPFRSoS_E@PLT
movl    -4(%rbp), %esi
movq    _ZSt4cout@GOTPCREL(%rip), %rdi

```

图 18: 部分 fibo.S 代码

```

0000000000000000 <main>:
0: 55                push    rbp
1: 48 89 e5           mov     rbp, rsp
4: 48 83 ec 20        sub     rsp, 0x20
8: c7 45 e8 00 00 00 mov     DWORD PTR [rbp-0x18], 0x0
f: c7 45 f4 00 00 00 mov     DWORD PTR [rbp-0xc], 0x0
16: c7 45 fc 01 00 00 mov     DWORD PTR [rbp-0x4], 0x1
1d: c7 45 f8 01 00 00 mov     DWORD PTR [rbp-0x8], 0x1
24: 48 8b 3d 00 00 00 mov     rdi, QWORD PTR [rip+0x0]
2b: 48 8d 75 f0        lea     rsi, [rbp-0x10]
2f: e8 00 00 00 00     call   34 <main+0x34>
34: 8b 75 f4           mov     esi, DWORD PTR [rbp-0xc]
37: 48 8b 3d 00 00 00 mov     rdi, QWORD PTR [rip+0x0]
3e: e8 00 00 00 00     call   43 <main+0x43>
43: 48 8b 35 00 00 00 mov     rsi, QWORD PTR [rip+0x0]
4a: 48 89 c7           mov     rdi, rax
4d: e8 00 00 00 00     call   52 <main+0x52>
52: 8b 75 fc           mov     esi, DWORD PTR [rbp-0x4]
55: 48 8b 3d 00 00 00 mov     rdi, QWORD PTR [rip+0x0]
5c: e8 00 00 00 00     call   61 <main+0x61>
61: 48 8b 35 00 00 00 mov     rsi, QWORD PTR [rip+0x0]
68: 48 89 c7           mov     rdi, rax
6b: e8 00 00 00 00     call   70 <main+0x70>

```

图 19: 部分 fibo.o 反汇编代码

- 指令生成：汇编器会将 fibo.S 中的每个汇编指令翻译成目标机器的指令。比如图 16 中的 pushq、movq、subq、leaq、callq、movl 等指令。
- 目标文件生成：汇编器会将生成的目标代码（通常是机器码）写入目标文件（.o 文件），这个文件包含了二进制表示的汇编程序。这个目标文件可以被链接器（如 ld）用来创建可执行程序。
- 地址解析和修正：汇编器会处理相对地址和绝对地址，确保代码中的跳转和引用都指向正确的内存位置。在此阶段通常包括计算相对偏移量以及将标签解析为地址。图 17 中左侧显示了每条指令的地址。

4.4 链接器

由汇编器生成的目标文件往往不能够直接执行。而链接器的作用是将可重定位的机器代码和其他可重定位的目标文件以及库文件链接在一起，最终才可以形成真正在机器上运行的代码。

这里我们对比如下指令，如图 20、21 中的结果所示：

- clang fibo.o -o fibo -no-pie -lstdc++
- clang fibo.o -o fibo -static

其中-no-pie 是为了让链接器不要创建一个可执行文件作为位置无关可执行文件，不添加这个选项会报错。

在一个非-PIE 可执行文件中，某些类型的重定位（如 R_X86_64_32）是不允许的，因为它们需要位置无关性来正常工作。这就是为什么链接器报错的原因。

使用-static 选项时，编译器会告诉链接器将所需的库静态地链接到可执行文件中，而不是动态链接到共享库。这意味着可执行文件将包含 C++ 标准库的一个静态版本，而不需要在运行时从系统中加载动态库。

这时编译器会自动解决位置无关可执行文件（PIE）的问题，因为 PIE 主要涉及到将可执行文件中的某些部分加载到不确定的内存位置。但是静态链接不需要 PIE，因此也不会报错。

- 可以声明变量和常量，进行初始化。
- 支持赋值操作、表达式、语句块、if 条件语句、while 循环和 return 语句。
- 表达式可以进行算术运算 (+、-、*、/、%、《、》)、关系运算 (==、>、<、>=、<=、!=) 和逻辑运算 (&&、||、!)。
- 可以定义和调用函数。
- 支持变量和常量的作用域，包括在函数和语句块中声明的变量和常量。
- 可以处理一维和二维数组，包括声明和访问数组元素。同时，支持 break 和 continue 语句。

针对将要实现的 SysY 编译器的语言特性，我们设计了两个 LLVM IR 程序，分别覆盖了语言特性的不同方面。设计思路为先编写伪代码，得到代码的一个整体逻辑思路，在此基础上编写 LLVM IR 代码，最后为了验证此代码的正确性，我们根据此伪代码逻辑编写 C 语言，使用 C 语言编译出的可执行文件进行验证。

5.2 程序一

5.2.1 语言特性

此程序的重点在于：整型变量、浮点数变量、一维数组的存储以及读取，函数、循环语句，各种算术指令 (移位、取整、加减乘除)，逻辑运算 (大小的比较)、跳转语句，最后循环内部的 continue 等功能也有实现。

5.2.2 程序伪代码与设计思路

Algorithm 1 llvm ir 程序对应的伪代码

```

1: constant = 2
2: function CALCULATE(a, b, c)
3:   temp = b << c
4:   a = a + temp
5:   return a
6: end function
7: function MAIN
8:   float n
9:   int i, j, k
10:  int array m[10]
11:  input n
12:  i = 5
13:  j = constant
14:  k = constant
15:  if i < (j + k) then
16:    i = i + 1
17:  else
18:    i = i - 1
19:  end if
20:  n = n + CALCULATE(i, j, k)

```

```

21:   for i = 0 to 9 do
22:       m[i] = i % 5
23:       if i == 5 then
24:           m[i] = m[i] + 2
25:       end if
26:   end for
27:   n = n / m[5]
28:   output n
29: end function

```

- calculate 函数：接收 3 个形参，b 和 c 做左移位操作运算，得到的结果与 a 相加，将结果返回
- main 函数：定义了一个浮点数 n，三个 int 操作数 i,j,k，以及一个一维数组 m，先让 i,j,k 做相关数学运算，其中也包括相关分支语句，然后进行一个循环语句，里面进行一维数组元素的储存，其中会进行整型与浮点数的类型强制转换，最后做浮点数的运算。

5.2.3 LLVM IR 实现

在 llvm ir 的语言特性中，申请整型临时变量空间用 alloca i32, align 4，申请浮点数临时变量空间 alloca float, align 4，申请数组临时变量空间 alloca [10 x i32], align 16。而这些对应的是指针变量，我们获取这些空间对应的值要用 store，赋给这些空间对应的值要用 load 指令。

而相关运算，逻辑、分支指令有: shl, srem, add nsw, div, icmp slt, fadd, fdiv, br 等，而关于类型的转换有: sitofp ... to float、fpext ... to double、sext i32 ... to i64。最后关于数组元素的读取用的是指针，getelementptr 即可以获取一个指向数组元素的指针，数组元素的存储与读取直接用 load 与 store 处理对应的指针即可。

具体实现思路如下:

首先对于 calculate 函数，

1. 此函数内部是对于三个参数进行相关逻辑运算，因此首先我们要申请三个临时变量空间，获取这些空间对应的指针变量%4、%5、%6。然后将函数的形参的值赋值给这些临时变量。前期整型临时变量创建完成后，我们要开始进行相关的数学逻辑运算执行。

2. 取出%5 和%6 指针对应的值%7、%8，然后让%5«%6，获得的值储存在%9 中，再取出%4 指针对应的值%10，然后进行加法运算，用 add nsw 进行加法运算，最后将执行的运算结果值返回

对于 main 函数中:

1. 首先定义一个返回值的临时变量%1，而后面定义一个浮点数变量 n，三个整型变量 i,j,k，一个一维整型数组 m。然后浮点数变量是由我们自己输入的，这里使用的是 scanf 函数，值得注意的是在最后面对 scanf 函数进行声明，也就是 declare。然后对这里的三个整型变量首先我们分别进行赋值即使用 store i32 5,i32* %3,align 4 进行赋值。赋值操作执行完后，开始执行相关 if 语句的判断，若 j+k 的值大于 i，则为 i++; 若 j+k 的值小于 i，则为 i-，这里减法使用 add ..., -1 即可。这里采取的是使用 icmp slt 判断 i 与 j+k 的大小，表示对两个 i32 类型的整数执行有符号整数的小于比较，结果会存储在一个虚拟寄存器%12 中，它的的值为 1（真）表示 i<j+k，为假表示 i>=j+k，再根据%12 的值进行分支语句的跳转，分别有代码块 13 与 15。

2. 下面进行浮点数与整型的相关运算，以及函数的调用。首先获取浮点数 n，也就是 load %2 对应空间的值，然后调用 calculate 函数，这里调用函数采取语句: call i32 @calculate(i32 %19,i32

%9,i32 %10), 然后这里整型与浮点数的转换采取 sitofp 指令, 将整型转为浮点数后, 使用浮点数的运算指令 fadd, 进行加法运算。

3. 最后是数组的存储与读取。这里有一个循环, 从代码块 23 到代码块 35 是循环体。首先这里将%3 作为计数器, 初始置为 0, 然后进行循环终止条件的比较, 即 icmp slt, 将储存的结果作为 br 指令的判断值, 即判断此时是否跳出循环。

4. 下面最重要的是如何对数组中特定的一个元素进行读取与赋值操作。数组的定义采用的是 alloca [10 x i32], 创建了一个包括 10 个 i32 元素的整型数组。关于数组特定元素的处理, 这里采用的是 getelementptr, 获取到此元素的一个指针, 若要赋值, 采取 store 指令; 若要读取, 采取 load。值得注意的是 store、load 的第二个操作数都只能是指针。循环体中也进行了一些运算操作, % 运算使用 srem 指令, 加法使用 add 等等。循环中的 continue 实现, 使用 br 跳转即可。

```

1  target triple = "x86_64-pc-linux-gnu"
2
3  @.str = private unnamed_addr constant [3 x i8] c"%f\00", align 1
4  ;c"%f\00" 为一个包含字符 "%","f" 和 "\00" 的数组
5
6  define i32 @calculate(i32 %0, i32 %1, i32 %2) #0 { ;#0 代表默认地址空间
7      %4 = alloca i32, align 4
8      %5 = alloca i32, align 4
9      %6 = alloca i32, align 4 ; 定义三个临时变量
10     store i32 %0, i32* %4, align 4
11     store i32 %1, i32* %5, align 4
12     store i32 %2, i32* %6, align 4 ; 函数形参赋值
13     %7 = load i32, i32* %5, align 4 ; 取出%5 指针对应的值
14     %8 = load i32, i32* %6, align 4 ; 取出%6 指针对应的值
15     %9 = shl i32 %7, %8 ; 运算符 <<
16     %10 = load i32, i32* %4, align 4
17     %11 = add nsw i32 %9, %10 ; %4+%5<<%6 加法运算
18     ret i32 %11 ; 返回运算得到的最终值
19 }
20
21 define i32 @main() #0 {
22     %1 = alloca i32, align 4 ; 返回值
23     %2 = alloca float, align 4 ; 浮点数 输入的值
24     %3 = alloca i32, align 4
25     %4 = alloca i32, align 4
26     %5 = alloca i32, align 4 ; 定义三个临时变量, 后续计算
27     %6 = alloca [10 x i32], align 16 ; 一维数组
28     store i32 0, i32* %1, align 4 ; 返回值, 默认为 0
29     %7 = call i32 @__isoc99_scanf(i8* getelementptr inbounds
30     ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), float* %2)
31     store i32 5, i32* %3, align 4
32     store i32 2, i32* %4, align 4
33     store i32 2, i32* %5, align 4 ; 为%3 %4 %5 三个变量赋值

```

```

34     %8 = load i32, i32* %3, align 4
35     %9 = load i32, i32* %4, align 4
36     %10 = load i32, i32* %5, align 4
37     %11 = add nsw i32 %9, %10 ; 得到%4+%5 的值
38     %12 = icmp slt i32 %8, %11 ; 条件判断 %3 与%4+%5 判断大小
39     br i1 %12, label %13, label %15 ; if 语句, 跳转
40 13:
41     %14 = add nsw i32 %8, 1 ; %8+1 也就是%3++
42     store i32 %14, i32* %3, align 4
43     br label %17
44 15:
45     %16 = add nsw i32 %8, -1 ; %8-1 也就是%9--
46     store i32 %16, i32* %3, align 4
47     br label %17
48 17:
49     %18 = load float, float* %2, align 4 ; 获取浮点数%2
50     %19 = load i32, i32* %3, align 4
51     %20 = call i32 @calculate(i32 %19, i32 %9, i32 %10)
52     %21 = sitofp i32 %20 to float
53     %22 = fadd float %18, %21
54     store float %22, float* %2, align 4
55     store i32 0, i32* %3, align 4
56     br label %23
57 23: ; 循环体 从 23 到 34
58     %24 = load i32, i32* %3, align 4 ; 取%3 指针对应的值
59     %25 = icmp slt i32 %24, 10 ; 循环终止条件 %3 >=10
60     br i1 %25, label %26, label %36 ; 根据%25 判断循环终止条件
61 26:
62     %27 = srem i32 %24, 5 ; 进行% 运算
63     %28 = sext i32 %24 to i64 ; 取出 现有%3 值, 计数器
64     %29 = getelementptr inbounds [10 x i32], [10 x i32]* %6, i64 0, i64 %28
65     ; 得到对应一维数组%6 的%28 索引的指针
66     store i32 %27, i32* %29, align 4
67     %30 = icmp eq i32 %24, 5 ; 比较 eq %3 和 5
68     br i1 %30, label %31, label %34 ; 跳转至%34 即为 continue
69 31: ; 满足%3==5 进行一系列操作
70     %32 = load i32, i32* %29, align 4
71     %33 = add nsw i32 %32, 2 ; 为%6 数组对应值 +2
72     store i32 %33, i32* %29, align 4
73     br label %34
74 34: ; 不满足%3==5 时
75     %35 = add nsw i32 %24, 1 ; 循环下标 +1 新一轮循环
76     store i32 %35, i32* %3, align 4
77     br label %23 ; 继续循环

```

```

78 36:      ; 跳出循环, 最终的代码块
79      ; 取出数组下标 5 的对应值
80      %37 = getelementptr inbounds [10 x i32], [10 x i32]* %6, i64 0, i64 5
81      %38 = load i32, i32* %37, align 4
82      %39 = sitofp i32 %38 to float ; 类型转换为 float
83      %40 = fdiv float %22, %39      ; 浮点数的除法运算
84      store float %40, float* %2, align 4 ; 赋值给%2 对应地址的值
85      %41 = fpext float %40 to double ; 类型转换为 double 输出 double
86      %42 = call i32 @__isoc99_scanf(i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
87      [3 x i8]* @.str, i64 0, i64 0), double %41)
88      %43 = load i32, i32* %1, align 4 ; 返回值 0
89      ret i32 %43
90  }
91  declare i32 @__isoc99_scanf(i8* , ...) #1 ;#1 为常量地址空间
92  declare i32 @printf(i8* , ...) #1

```

5.2.4 正确性验证

采取如下指令:

```

1 llvm-as final_own.ll -o final_own.bc
2 llc final_own.bc -filetype=obj -o final_own.o
3 clang final_own.o -o final_own -no-pie
4 ./final_own

```

```

● prayer@prayer-virtual-machine:~/compile/lab1/llvm_ir编写与示例$ ./final_own
1.0
● 6.500000prayer@prayer-virtual-machine:~/compile/lab1/llvm_ir编写与示例$ ./final_own
2.0
● 7.000000prayer@prayer-virtual-machine:~/compile/lab1/llvm_ir编写与示例$ ./final_own
3.0
○ 7.500000prayer@prayer-virtual-machine:~/compile/lab1/llvm_ir编写与示例$

```

图 22: 程序一验证结果

由伪代码逻辑, 程序最终输出为 $(n+i-1+j \ll k)/m[5]$, 当 $n=1.0$ 时, 结果为 6.5; 当 $n=2.0$ 时, 结果为 7.0; 当 $n=3.0$ 时, 结果为 7.5, 与上图程序的结果一致, llvm ir 程序正确。

5.3 程序二

5.3.1 设计思路

该程序设计的重点在于二维数组的实现, 以及函数调用、多重循环。

- 函数 func 接受两个整数参数 x 和 y 。在函数体内, 将 x 和 y 相乘, 并将乘积作为函数返回值返回

- main 函数定义了一个整数变量 num，常量 zero 赋值为 0，以及一个二维整数数组 a 初始化为全零。使用嵌套循环初始化数组 a 的元素为 i+j，并将 i+j 累加到变量 num。最后调用函数 func 计算 2 乘以 3 的结果，将结果加到 num。使用 printf 函数输出 num 的值。

5.3.2 伪代码实现

以下是程序的伪代码：

Algorithm 2 Pseudo Code

```

1: function FUNC( $x, y$ )
2:    $z \leftarrow x \times y$ 
3:   return  $z$ 
4: end function
5: function MAIN
6:    $num$ 
7:    $num \leftarrow ONE$ 
8:   const  $zero \leftarrow 0$ 
9:   Integer array  $a[3][3]$ 
10:  for  $i \leftarrow zero$  to 3 do
11:    for  $j \leftarrow zero$  to 3 do
12:       $a[i][j] \leftarrow i + j$ 
13:       $num \leftarrow num + a[i][j]$ 
14:    end for
15:  end for
16:   $num \leftarrow num + \text{FUNC}(2, 3)$ 
17:  PRINTF("%d",  $num$ )
18: end function

```

5.3.3 LLVM IR 实现

使用 LLVM IR 实现上述程序逻辑，实现思路如下：

实现 func 函数时，首先使用 alloca 指令为要使用的变量 x,y,z 分配空间。在这里就是为 int32 型变量分配 4 字节的空间，并通过 align 指令指出对齐类型为 4。再使用 store 指令将函数参数 %0,%1 的值存入相应位置。

接下来计算 $x*y$ 的值，并赋值给 z。简单起见，**每次使用变量值时都使用 load 指令从相应位置加载值到新的虚拟寄存器中**，而不是查看之前申请过的虚拟寄存器。所以此处将 x,y 的值载入虚拟寄存器 %6,%7，并使用 mul 指令计算出 $x*y$ 并赋值给虚拟寄存器 %8，最后将 %8 的值存入 z 的内存地址，并返回 z 值。

需要注意的时，LLVM IR 中的虚拟寄存器，也就是形如 "%n"（其中 n 为正整数）的表达式，其类型与值都是固定的，一旦赋值（如 alloca、load、add 等指令）之后无法修改。在声明新的虚拟寄存器时，**其 n 值应为连续的正整数**，也就是说不能任意赋值，只能是上一次申请的虚拟寄存器 n 值加 1。

在实现 main 函数时，需要注意的是**二维数组以及双重循环的实现**。

首先二维数组 $a[3][3]$ 通过 alloca 分配空间，在本程序中（下方代码第 20 行）分配了一个 $[3 \times [3 \times i32]]$ 的空间，并将数组指针返回给虚拟寄存器 %3。由于 C 语言中数组为 16 字节对齐，

所以 `align` 指令需要指定为 16。

在访问数组元素，如 `a[i][j]` 时，首先使用指令

`getelementptr inbounds [3 x [3 x i32]], [3 x [3 x i32]]*` 虚拟寄存器 1, i64 0, i64 虚拟寄存器 2

获得指向 `a[i]` 的指针。其中 `inbounds [3 x [3 x i32]]` 指出偏移量不会越界。虚拟寄存器 1 类型为 `[3 x [3 x i32]]*`，也就是数组 `a` 的指针。虚拟寄存器 2 给出了数组的偏移量，也就是 `i`。

接下来使用指令

`getelementptr inbounds [3 x i32], [3 x i32]*` 虚拟寄存器 1, i64 0, i64 虚拟寄存器 2

获取指向 `a[i][j]` 的指针，其类型为 `i32*`，最后使用 `load` 指令加载 `a[i][j]` 的值。

此处需要注意的是，由于目标机器使用 64 位地址，所以在获取地址偏移量时，也就是加载 `i, j` 时，需要使用 `sext` 指令将 `i32` 类型扩展位 `i64` 类型。

以下为可以编译执行的 LLVM IR：

```

1  @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
2
3  define i32 @func(i32 noundef %0, i32 noundef %1) #0{
4  %3 = alloca i32, align 4 ;z
5  %4 = alloca i32, align 4 ;x
6  %5 = alloca i32, align 4 ;y
7  store i32 %0, i32* %4, align 4
8  store i32 %1, i32* %5, align 4
9  %6 = load i32, i32* %4, align 4; load x to %4
10 %7 = load i32, i32* %5, align 4; load y to %5
11 %8 = mul nsw i32 %6, %7
12 store i32 %8, i32* %3, align 4
13 %9 = load i32, i32* %3, align 4
14 ret i32 %9
15 }
16
17 define i32 @main() #0{
18 %1 = alloca i32, align 4 ;num
19 %2 = alloca float, align 4 ;zero
20 ; 分配一个 [3 x [3 x i32]] 的空间
21 ; 数组采用 16 字节对齐
22 %3 = alloca [3 x [3 x i32]], align 16 ;a[3][3]
23 %4 = alloca i32, align 4 ;i
24 %5 = alloca i32, align 4 ;j
25 store i32 1, i32* %1, align 4 ;num=1
26 store float 0.000000e+00, float* %2, align 4 ;zero=0、
27 ; 外层循环变量 i 初始化
28 store i32 0, i32* %4, align 4 ;i=0
29 br label %6
30
31 6:
32 ; 外层循环条件 i<3

```

```

33 %7 = load i32,i32* %4,align 4 ;i
34 %8 = icmp slt i32 %7, 3 ;i<3
35 br i1 %8,label %9,label %37
36
37 9:
38 ; 内层循环变量 j 初始化
39 store i32 0,i32* %5,align 4 ;j=0
40 br label %temp
41
42 temp:
43 ; 内层循环条件 j<3
44 %10 = load i32,i32* %5,align 4
45 %11 = icmp slt i32 %10, 3 ;j<3
46 br i1 %11,label %12,label %34
47
48 12:
49 ; 计算 i+j
50 %13 = load i32,i32* %4,align 4
51 %14 = load i32,i32* %5,align 4
52 %15 = add nsw i32 %13,%14 ;i+j
53
54 ; 获取 a[i][j] 的地址并存入 i+j 的结果
55 %16 = load i32,i32* %4,align 4 ;i
56 %17 = sext i32 %16 to i64 ; 32->64
57 ;get pointer of a[i]
58 %18 = getelementptr inbounds [3 x [3 x i32]], [3 x [3 x i32]]* %3,i64 0,i64 %17
59 %19 = load i32,i32* %5,align 4 ;j
60 %20 = sext i32 %19 to i64 ; 32->64
61 ;get pointer of a[i][j]
62 %21 = getelementptr inbounds [3 x i32],[3 x i32]* %18,i64 0,i64 %20
63 store i32 %15,i32* %21,align 32 ;a[i][j]=i+j
64
65 ; 获取 a[i][j] 地址并取出 a[i][j]
66 ; 计算 a[i][j]+num 并将结果存入 num
67 %22 = load i32,i32* %1,align 32 ;num
68 %23 = load i32,i32* %4,align 4 ;i
69 %24 = sext i32 %23 to i64 ; 32->64
70 ;get pointer of a[i]
71 %25 = getelementptr inbounds [3 x [3 x i32]], [3 x [3 x i32]]* %3,i64 0,i64 %24
72 %26 = load i32,i32* %5,align 4 ;j
73 %27 = sext i32 %26 to i64 ; 32->64
74 ;get pointer of a[i][j]
75 %28 = getelementptr inbounds [3 x i32],[3 x i32]* %25,i64 0,i64 %27
76 %29 = load i32,i32* %28,align 4 ;a[i][j]

```



```

77  %30 = add nsw i32 %22,%29 ;num + a[i][j]
78  store i32 %30,i32* %1,align 32
79  br label %31
80
81  31:
82  ;j 自增
83  %32 = load i32,i32* %5,align 4
84  %33 = add nsw i32 %32,1 ;j++
85  store i32 %33,i32* %5,align 4
86  br label %temp
87
88  34:
89  ;i 自增
90  %35 = load i32,i32* %4,align 4
91  %36 = add nsw i32 %35,1 ;i++
92  store i32 %36,i32* %4,align 4
93  br label %6
94
95  37:
96  %38 = load i32, i32* %1, align 4
97  %39 = call i32 @func(i32 noundef 2, i32 noundef 3)
98  %40 = add nsw i32 %38, %39 ;num=num+func(2,3)
99  store i32 %40, i32* %1, align 4
100
101  %41 = load i32, i32* %1, align 4
102  %42 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([3 x i8],
103  [3 x i8]* @.str, i64 0, i64 0), i32 noundef %41)
104  ret i32 0
105  }
106
107  declare dso_local i32 @__isoc99_scanf(i8*, ...)
108
109  declare dso_local i32 @printf(i8*, ...)
110
111  attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all"
112  "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8"
113  "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
114  "tune-cpu"="generic" }

```

5.3.4 正确性验证

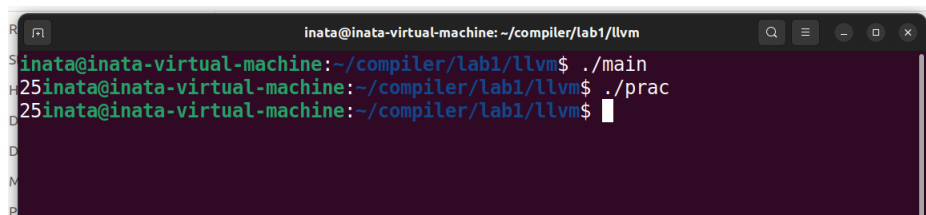
编写脚本 batch.sh 将上述 LLVM IR (文件名 prac.ll) 编译为可执行文件 prac。

```
1      ;batch.sh
2      llvm-as -o prac.bc prac.ll
3      llc -o prac.s prac.bc
4      as -o prac.o prac.s
5      gcc -o prac prac.o
```

为对比验证正确性, 我们根据程序逻辑编写了相应的 C 语言代码如下: 以下是相应的 C 语言代码:

```
1
2  #include<stdio.h>
3  int func(int x,int y)
4  {
5      int z=x*y;
6      return z;
7  }
8
9  int main(){
10     int num;
11     num=1;
12     const float zero=0;
13
14     int a[3][3];
15     for(int i=zero;i<3;i++)
16     {
17         for(int j=zero;j<3;j++)
18         {
19             a[i][j]=i+j;
20             num+=a[i][j];
21         }
22     }
23     num=num+func(2,3);
24     printf("%d",num);
25 }
```

编译 C 语言程序形成可执行文件 main, 与 prac 一起运行对比, 可以发现运行结果一致, 正确性得到验证。

A terminal window titled 'inata@inata-virtual-machine: ~/compiler/lab1/llvm'. The prompt is 'inata@inata-virtual-machine:~/compiler/lab1/llvm\$'. The first command entered is './main', followed by './prac'. The prompt returns after each command. The terminal has a dark purple background and white text. The window title bar shows standard Linux window controls (minimize, maximize, close) and a search icon.

```
inata@inata-virtual-machine: ~/compiler/lab1/llvm
inata@inata-virtual-machine:~/compiler/lab1/llvm$ ./main
25inata@inata-virtual-machine:~/compiler/lab1/llvm$ ./prac
25inata@inata-virtual-machine:~/compiler/lab1/llvm$
```

图 23: 运行结果验证

6 总结与展望

在本次作业中，我们首先对一个编译器的工作流程进行了大致的回顾，然后借助具体程序分别研究了 `g++` 与 `llvm` 编译器各个阶段的工作细节。其中的重点在于编译阶段，在这一部分我们通过研究编译器生成的中间文件，细致地分析了编译器在代码优化方面所做的工作。最后我们从不同的角度设计了两个 LLVM IR 程序，在此过程中学会了 LLVM IR 的语法。此外这两个程序分别覆盖了 `SysY` 编译器的不同语言特性，为未来设计 `SysY` 编译器打下了基础。

参考文献

- [1] aywala. gcc-original-to-dot.