



# 编译优化 实验讲解

讲解人 课程助教 肖同欢

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

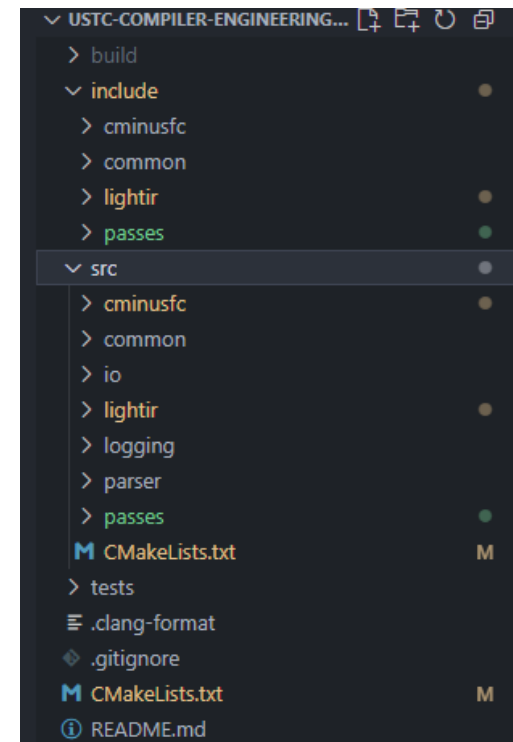
计算机科学与技术学院

2025年10月30日

# Lab 2 编译优化



- 在 IR 上通过 Pass 消除冗余代码，提高代码的执行效率
- 代码仓 include/ 和 passes/ 下增加了优化有关的工具类
- 可以阅读代码，并在三个优化函数内联，死代码删除，常量传播三个中选择至少一个进行实现
- 代码中给大家实现了一个 mem2reg 优化，可以参考
- 实现对应优化的 .cpp 文件即可，比如我想实现死代码删除，完善 src/passes/DeadCode.cpp 即可



# Lab 2 编译优化



- 代码仓 checkout 到对应的分支 lab2

The screenshot shows a GitHub repository interface. At the top, there's a navigation bar with 'main' selected, '2 Branches', and '0 Tags'. A search bar 'Go to file' and buttons 'Add file' and '<> Code' are also visible. A dropdown menu 'Switch branches/tags' is open, showing a search bar 'Find or create a branch...' and two tabs: 'Branches' and 'Tags'. Under 'Branches', 'main' is selected with a checkmark and labeled 'default', and 'lab2' is listed below it. A link 'View all branches' is at the bottom of the dropdown. The background shows a commit history table with columns for commit message and time ago.

Commit Message	Time Ago
lab init	7 months ago
modify num of TODOS	3 weeks ago
lab2: blank for warmup cpp	last year
add clang format config	last year
add clang format config	last year
(WIP) add lab2 code	last year
Update README.md	6 months ago

- 本次实验的目标是根据提供的实验框架完善实验中优化的实现
- 优化的调用 main.cpp
  - 首先要把参数传入 main.cpp, 可以使用 -dec, -const-prop, -func-inline 参数调用优化
  - 传入对应的函数, 就执行右侧对应的优化

```
for (int i = 1; i < argc; ++i) {  
    if (argv[i] == "-h"s || argv[i] == "--help")  
        print_help();  
    else if (argv[i] == "-o"s) {  
        if (output_file.empty() && i + 1 < argc)  
            output_file = argv[i + 1];  
        i += 1;  
    } else {  
        print_err("bad output file");  
    }  
    else if (argv[i] == "-emit-ast"s) {  
        emitast = true;  
    } else if (argv[i] == "-emit-llvm"s) {  
        emitllvm = true;  
    } else if (argv[i] == "-dce"s) {  
        dce = true;  
    } else if (argv[i] == "-const-prop"s) {  
        const_prop = true;  
    } else if (argv[i] == "-func-inline"s) {  
        func_inline = true;  
    } else {
```

```
PassManager PM(m.get());  
// optimization  
if(config.dce) {  
    PM.add_pass<Mem2Reg>();  
    PM.add_pass<DeadCode>();  
}  
  
if(config.func_inline) {  
    PM.add_pass<FunctionInline>();  
    PM.add_pass<DeadCode>();  
}  
  
if(config.const_prop) {  
    PM.add_pass<Mem2Reg>();  
    PM.add_pass<DeadCode>();  
    PM.add_pass<ConstPropagation>();  
    PM.add_pass<DeadCode>();  
}  
PM.run();
```

- **Include** ----- **light ir 代码库文件**
- **src** ----- **IR 自动生成框架源文件**
- **tests**
  - 1-parser ----- 生成抽象语法树阶段的测试（不用管）
  - 2-ir-gen ----- lab2 测试相关文件
    - autogen ----- lab2 相关测试
    - warmup ----- 大家可以用这里的内容来练手（单个测试）
      - answers ----- 每个测试用例对应的输出
      - testcases ----- 每个测试用例
      - eval\_lab2.py
      - eval\_lab2.sh ----- 测试用脚本

# Lab2 结果测试（对单个代码调试）



- 代码 build 成功后，可以使用可执行文件先测试一些简单的例子，比如我的可执行文件在 build/cminusfc

```

└─ build
   └─ tests
      ├── calc
      ├── calculator.h
      ├── cmake_install.cmake
      ├── CMakeCache.txt
      └── cminusfc
         └─ compile_command.txt
            └─ ~/USTC-Compiler-Engineering-2025-Ans/build/cminusfc
               ├── gcd_array_generator
               └── lexer
```

# Lab 2 结果测试（对单个代码调试）



- 先观察判断 ir 生成是否正确，再进行后面步骤
- 如果我们不使用优化，比如我执行了下面的命令

```
./build/cminusfc -o ./test.ll -emit-llvm ./build/test.cminus
```

- 其中 ./build/test.cminus 是我们要处理的文件，./test.ll 是我们要生成的文件，内容如下

```
void main(void) {  
    int a;  
    int b;  
    a = 1;  
    b = 1;  
    return 0;  
}
```

```
1 ; ModuleID = 'cminus'  
2 source_filename = "/root/USTC-Compiler-Engineering-2025-Ans/build/test.cminus"  
3  
4 declare i32 @input()  
5  
6 declare void @output(i32)  
7  
8 declare void @outputFloat(float)  
9  
10 declare void @neg_idx_except()  
11  
12 define void @main() {  
13     label_entry:  
14     %op0 = alloca i32  
15     %op1 = alloca i32  
16     store i32 1, i32* %op0  
17     store i32 1, i32* %op1  
18     %op2 = sitofp i32 0 to float  
19     ret float %op2  
20 }  
21
```

# Lab 2 结果测试（对单个代码调试）



- 先观察判断 ir 生成是否正确，再进行后面步骤
- 比如我执行了下面的命令

```
./build/cminusfc -o ./test.ll -emit-llvm -dce ./build/test.cminus
```

- 其中 ./build/test.cminus 是我们要处理的文件，./test.ll 是我们要生成的文件，注意，我们本次使用了死代码删除的功能，内容如下

```
void main(void) {  
    int a;  
    int b;  
    a = 1;  
    b = 1;  
    return 0;  
}
```

```
1 ; ModuleID = 'cminus'  
2 source_filename = "/root/USTC-Compiler-Engineering-2025-Ans/build/test.cminus"  
3  
4 declare i32 @input()  
5  
6 declare void @output(i32)  
7  
8 declare void @outputFloat(float)  
9  
10 declare void @neg_idx_except()  
11  
12 define void @main() {  
13     label_entry:  
14     %op0 = sitofp i32 0 to float  
15     ret float %op0  
16 }  
17
```



# Lab 2 结果测试（对单个代码调试）



- **ir 生成后，可以通过 clang 生成可执行文件**

```
# clang -o0 -w -no-pie ./test.ll -o ./test -L ./build/ -lcminus_io
```

优化层级，拦截所有警告，生成位置有关代码，生成的.ll 文件 可执行文件的位置， 连接的库位置，需要连接实验提供的io库

- **执行可执行文件后，可以通过 echo \$? 来查看 return code**

```
(base) root@cf85b2980346:/workspace/Compiler-Cminus-2025# clang -o0 -w ./test.ll -o ./test -L ./build/ -lcminus_io
(base) root@cf85b2980346:/workspace/Compiler-Cminus-2025# echo $?
0
(base) root@cf85b2980346:/workspace/Compiler-Cminus-2025#
```

# Lab 2 实验内容（大规模测试）



- 可以执行 `/test/2-ir-gen/autogen` 目录下的 `eval_lab2.sh` 脚本，查看对所有测试样例，结果在 `tests/2-ir-gen/autogen/eval_result`
- 可以查看每一个测试用例的成功与失败，80个用例按照难度分为5个等级，总共是100分，成功的测试用例会显示 `Success`，失败就显示 `Fail`
- 于 lab1 的不同之处在于，这次可以对 `eval_lab2.sh` 脚本传参数，比如

```
(base) root@a24524635295:~/USTC-Compiler-Engineering-2025-Ans/tests/2-ir-gen/autogen# bash eval_lab2.sh dce
Running with optimizations: dce
(base) root@a24524635295:~/USTC-Compiler-Engineering-2025-Ans/tests/2-ir-gen/autogen#
```

# Lab 2 实验内容 (大规模测试)



- 如果没问题如右图所示
  - 所有样例都显示 Success
  - 总分(total points)是满分
- 如果正确启用了优化, 开头会有提示

```
tests > 2-ir-gen > autogen > eval_result
1  Running with optimizations: -dce
2
3  =====lv0_1 START=====
4  return: Success
5  decl_int: Success
6  decl_float: Success
7  decl_int_array: Success
8  decl_float_array: Success
9  input: Success
10 output_float: Success
11 output_int: Success
12 points of lv0_1 is: 17
13 =====lv0_1 END=====
14
15 =====lv0_2 START=====
16 num_add_int: Success
17 num_sub_int: Success
18 num_mul_int: Success
```

```
65 negidx_voidfuncall: Success
66 selection1: Success
67 selection2: Success
68 selection3: Success
69 iteration1: Success
70 iteration2: Success
71 scope: Success
72 transfer_float_to_int: Success
73 transfer_int_to_float: Success
74 points of lv1 is: 31
75 =====lv1 END=====
76
77 =====lv2 START=====
78 funcall_chain: Success
79 assign_chain: Success
80 funcall_var: Success
81 funcall_int_array: Success
82 funcall_float_array: Success
83 funcall_array_array: Success
84 return_in_middle1: Success
85 return_in_middle2: Success
86 funcall_type_mismatch1: Success
87 funcall_type_mismatch2: Success
88 return_type_mismatch1: Success
89 return_type_mismatch2: Success
90 points of lv2 is: 23
91 =====lv2 END=====
92
93 =====lv3 START=====
94 complex1: Success
95 complex2: Success
96 complex3: Success
97 complex4: Success
98 points of lv3 is: 11
99 =====lv3 END=====
100
101 total points: 100
```

# Lab2实验内容（大规模测试）



- 如果有问题，比如

```
=====lv3 START=====
complex1: Success
complex2: Fail
complex3: Success
complex4: Success
points of lv3 is: 8
=====lv3 END=====

total points: 97
```

- 发现是 complex2 有问题
- 用例位置在
  - tests/2-ir-gen/autogen/testcases/lv3/complex2.cminus
- 参考答案和输入位置（可执行文件的输入输出，**不是参考的 IR !**）
  - tests/2-ir-gen/autogen/answers/lv3/complex2.in
  - tests/2-ir-gen/autogen/answers/lv3/complex2.out
- 按照最开始的单例测试方法，进行针对性测试，看看是哪里的问题

# Lab2实验提交方法



- 将 GitHub 仓地址发送到邮箱 [tonghuanxiao@mail.ustc.edu.cn](mailto:tonghuanxiao@mail.ustc.edu.cn)
- 将验证正确性成功的文件(eval\_result)一并发送到上述邮箱
- **注意**
  - 请确保 GitHub 有正常的提交记录以证明本实验是自己完成的
  - 请不要大规模使用 AI 完成任务
  - 请注意实验截止时间（11月23日晚12点）