

# 代码优化

华志远 梅骏逸 李帅东

2024 年 11 月

# 目录

<b>1 实验描述及内容</b>	<b>3</b>
<b>2 基本要求实验流程</b>	<b>3</b>
2.1 建立控制流图以及消除不可达指令和基本块	3
2.2 mem2reg 的简单情况	3
2.2.1 alloca 的指针没有被 use	4
2.2.2 alloca 的指针的 def 和 use 均在同一基本块中	4
<b>3 基本要求实验结果测试</b>	<b>5</b>
3.1 CPP-RISCV 框架	5
3.1.1 自动测试脚本	5
3.1.2 自动测试脚本结果	5
3.2 CPP-ARM 框架	6
3.2.1 自动测试脚本	6
3.2.2 自动测试脚本结果	6
<b>4 进阶要求实验流程</b>	<b>6</b>
4.1 完整 mem2reg 实验流程	6
4.2 进阶要求其他优化实验流程	8
4.2.1 AnalysisPass	8
4.2.2 TransformPass	9
<b>5 评分标准</b>	<b>11</b>
5.1 基本要求评分标准	11
5.2 进阶要求评分标准	11
5.2.1 mem2reg 评分标准	11
5.2.2 AnalysisPass 评分标准	11
5.2.3 TransformPass 评分标准	11

## 1 实验描述及内容

在之前的三次实验中，我们成功完成了中间代码生成的工作，并生成了正确的中间代码。如果你真正理解了中间代码生成的步骤，你也许会好奇为什么我们不使用相似的算法直接生成汇编，反而要先进行中间代码生成的步骤来增加工作量。在本次实验中，你将了解到中间代码的作用，同时你也会了解到为什么 GCC 的 O0 和 O2 往往会有好几倍的速度差距。

在本次实验中，我们将开始探索现代通用优化编译器是如何对代码进行优化的。实际上，这也是编译器最重要的组成部分。一般情况下，代码优化环节的代码量往往会占据一个编译器总体代码的 80% 以上，当然在本次实验中我们只需要实现其中几个最经典的优化即可。在进阶要求的自由发挥部分，你可以根据自己的兴趣自由选择一些优化来实现。

## 2 基本要求实验流程

### 2.1 建立控制流图以及消除不可达指令和基本块

```

1  int a = 5;
2  if(a > 5){
3      return 5;
4      if(a > 6){
5          return 7;
6      }
7  }
8  return 0;
9  int b = 10;

```

```

1  %r4 = alloca i32
2  %r1 = alloca i32
3  store i32 5, ptr %r1
4  %r2 = load i32, ptr %r1
5  %r3 = icmp sgt i32 %r2, 5
6  br i1 %r3, label %L1, label %L2
7
8  L1:
9  ret i32 5
10 %r5 = load i32, ptr %r1
11 %r6 = icmp sgt i32 %r5, 6
12 br i1 %r6, label %L3, label %L4
13
14 L3:
15 ret i32 7
16 br label %L4
17
18 L4:
19 ret i32 0
20 store i32 10, ptr %r4

```

```

1  %r4 = alloca i32
2  %r1 = alloca i32
3  store i32 5, ptr %r1
4  %r2 = load i32, ptr %r1
5  %r3 = icmp sgt i32 %r2, 5
6  br i1 %r3, label %L1, label %L2
7
8  L1:
9  ret i32 5
10 L2:
11 ret i32 0

```

在本次实验中，首先你需要建立控制流图并删除不可达的指令和基本块，算法流程如下：

1. 从函数的入口基本块开始进行一次 dfs，在 dfs 过程中检查每个基本块的跳转指令，并根据跳转指令建立控制流图。如果该基本块是第一次遇到，就遍历该基本块直到末尾或者第一次出现跳转/函数返回指令。如果出现了跳转/函数返回指令，就将该基本块该指令后的其余指令全部删除。
2. 在完成 dfs 后，如果该基本块没有被访问过，即意味着该基本块永远不会被执行到，我们可以将这些基本块全部删除。

### 2.2 mem2reg 的简单情况

在中间代码生成这一次实验中，你已经成功生成了 SSA 形式的 LLVMIR，但是这个 SSA 形式并不是真正的 SSA，而是借助内存操作指令 alloca/load/store 来存储局部变量的 SSA 形式。并且这种 SSA 形式会频繁的对内存进行读写，同时也不利于后续的编译优化，所以我们需要尽可能消除对局部变量的 alloca，load，store 指令的同时维护 SSA 性质。

我们考虑每一个 alloca 指令产生的指针，我们记对该指针的 load 为 use，对该指针的 store 为 def。mem2reg 优化要做的是在维持 SSA 性质的前提下，删去该 alloca 指令和相关的 load，store 指令，将在 alloca 出来的内存中存储的值改为用寄存器存储。

对于基础要求，我们只需要实现 mem2reg 的两种简单情况即可。

### 2.2.1 alloca 的指针没有被 use

```
1 int main()
2 {
3     int a = 5;
4     int b = 10;
5     return 0;
6 }
```

```
1 define i32 @main(){
2     %r1 = alloca i32
3     %r2 = alloca i32
4     store i32 5, ptr %r1
5     store i32 10, ptr %r2
6     ret i32 0
7 }
```

```
1 define i32 @main(){
2     ret i32 0
3 }
```

这种情况是最简单的一种,你只需要遍历一次所有指令,如果发现某个 alloca 结果没有任何 use(即没有被 load 过),就将与该 alloca 相关的指令全部删除。

cpp-riscv 框架中同一基本块中的指令采用 deque 存储,单次删除是线性的时间复杂度,这对于一些大型代码来说是不可接受的。所以我们可以先将要删除的指令标记。在检查完所有的 alloca 后,新建一个 deque,将没被标记的指令插入进该 deque 中,随后更换基本块中的 deque 即可,这样只需要线性时间就可以完成所有的删除操作。(对于这一思路以及时间复杂度分析,相信你在算法导论课程中已经有了一定了解,这里仅作一些提示,具体细节需要大家自行探索)

框架中的测试脚本对你编译时间有一定的要求,我们要求你的时间复杂度为线性或者线性对数 ( $\Theta(n \log n)$ ,  $\Theta(n \log n \log n)$  均可,  $n$  为指令数),如果你算法的时间复杂度不满足要求,会无法通过测试。

### 2.2.2 alloca 的指针的 def 和 use 均在同一基本块中

```
1 int main()
2 {
3     int a = getint();
4     int b = getint();
5     a = a + 5;
6     return a+b;
7 }
```

```
1 define i32 @main() {
2     %r2 = alloca i32
3     %r3 = alloca i32
4     %r4 = call i32 @getint()
5     store i32 %r4, ptr %r2
6     %r5 = call i32 @getint()
7     store i32 %r5, ptr %r3
8     %r6 = load i32, ptr %r2
9     %r7 = add i32 %r6, 5
10    store i32 %r7, ptr %r2
11    %r8 = load i32, ptr %r2
12    %r9 = load i32, ptr %r3
13    %r10 = add i32 %r8, %r9
14    ret i32 %r10
15 }
```

```
1 define i32 @main() {
2     %r4 = call i32 @getint()
3     %r5 = call i32 @getint()
4     %r7 = add i32 %4, 5
5     %r10 = add i32 %r7, %5
6     ret i32 %r10
7 }
```

如果我们确定了某一个 alloca 的 use 和 def 在同一基本块内,我们只需要遍历该基本块,同时维护一个变量 val 并将该变量初始化为 undef。如果我们遇到了一条与该 alloca 相关的 store 指令,就将 val 设置为该 store 指令要向内存中写的值,并删除该指令;如果我们遇到了一条与该 alloca 相关的 load 指令,就将所有使用该 load 结果的寄存器替换为 val,并删除该指令。

和之前的优化一样,我们同样对你的算法时间复杂度有一定要求,你需要在一次遍历中处理完所有的 alloca 指令,在处理 alloca 指令的过程中,我们可以先保存要替换的寄存器以及标记要删除的指令,但不实际执行相关操作;遍历完成后,我们可以在后续的常数级遍历中完成寄存器的替换和指令的删除。(相信你可以利用你在数据结构和算法导论课程中学习到的内容来快速地完成上述操作)

对于这一优化,框架中的测试脚本同样对你编译时间有一定的要求,你的时间复杂度依旧只能为线性或者线性对数 ( $\Theta(n \log n)$ ,  $\Theta(n \log n \log n)$  均可,  $n$  为指令数),如果你算法的时间复杂度不满足要

求，会无法通过测试。

## 3 基本要求实验结果测试

### 3.1 CPP-RISCV 框架

使用如下命令进行编译

---

```
make -j # 如果你更新的不是 SysY_parser.y 和 SysY_lexer.l 文件，直接 make -j 即可
```

---

假设你成功编译后想要查看你的编译器对项目根目录下的 example.sy 的编译结果，使用如下命令：

---

```
./bin/SysYc -llvm -o example.out.ll example.sy -O1
clang-15 example.out.ll -c -o tmp.o
clang-15 -static tmp.o lib/libsysy_x86.a
# 请注意，如果你的电脑架构不为 x86，你需要自行制作链接库用于链接，测试脚本中也需要进行修改。
./a.out # 运行程序
echo $? # 查看程序 main 函数的返回值，你可以自行比较是否符合预期
# 你还需要自行查看生成的 llvm 代码是否符合你的优化预期
```

---

#### 3.1.1 自动测试脚本

---

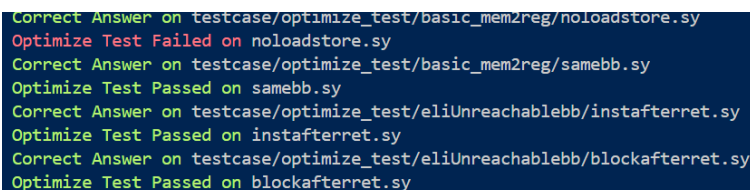
```
python3 optimize_test.py # 测试代码优化基本要求
# 进阶要求无测试脚本，线下检查时助教会检查你的代码是否正确优化
```

---

#### 3.1.2 自动测试脚本结果

对于每一个样例，有两个测试结果：

1. 第一行表示该样例的答案是否正确，代码优化需要保证生成代码的正确性。
2. 第二行表示该样例的优化是否正确，但需要注意测试脚本只能粗略地检查你的优化是否正确，具体会由助教线下进行检查。其中删去不可达基本块优化会检查你生成的代码体积大小。mem2reg 优化会检查你生成的 llvm 代码中是否存在 alloca 指令。



```
Correct Answer on testcase/optimize_test/basic_mem2reg/noloadstore.sy
Optimize Test Failed on testcase/optimize_test/basic_mem2reg/samebb.sy
Correct Answer on testcase/optimize_test/samebb.sy
Optimize Test Passed on testcase/optimize_test/eliUnreachablebb/instafterret.sy
Correct Answer on testcase/optimize_test/instafterret.sy
Optimize Test Passed on testcase/optimize_test/eliUnreachablebb/blockafterret.sy
Correct Answer on testcase/optimize_test/blockafterret.sy
Optimize Test Passed on testcase/optimize_test/blockafterret.sy
```

图 3.1: 测试结果示例

## 3.2 CPP-ARM 框架

大家在拿到指导书后，请先查看 lab3 分支中的 `Makefile` 文件和 `test` 文件夹是否及时更新。相较于前两周，`Makefile` 中添加了代码优化的自动化检查的命令，`test` 文件夹中更新了一些新的测试样例。

使用如下命令进行编译，并对 `example.sy` 进行测试。

---

```
make run
```

---

在编译完编译器后，可以使用如下的指令对 `test` 文件夹内的 `.sy` 进行批量操作。

---

```
make test
```

---

```
# 你还需要自行查看生成的 llvm 代码是否符合你的优化预期
```

---

### 3.2.1 自动测试脚本

---

```
make check_unreachable # 代码优化检查之 eliUnreachablebb
```

---

```
make check_mem2reg      # 代码优化检查至 basic_mem2reg
```

---

```
# 进阶要求无测试脚本，线下检查时助教会检查你的代码是否正确优化
```

---

### 3.2.2 自动测试脚本结果

对于每一个样例，有一个测试结果，即样例答案是否正确。

对于代码优化检查的测试样本，有两个测试结果，即样例答案是否正确、优化是否正确，需要注意测试脚本只能粗略地检查你的优化是否正确，具体会有助教线下进行检查。其中删去不可达基本块优化会检查你生成的代码体积大小。`mem2reg` 优化会检查你生成的 `llvm` 代码中是否存在 `alloca` 指令。但是需要注意，如果样例答案错误但优化正确，不得分！

## 4 进阶要求实验流程

### 4.1 完整 `mem2reg` 实验流程

基础要求中，你已经实现了 `mem2reg` 的两种简单情况，但实际上，`mem2reg` 要远比这两种情况复杂。

如图4.2所示，局部变量 `a` 在基本块 `B1` 中被定义，我们将其重命名为  $a_1$ ，`a` 在 `B2` 中也被定义了一次，我们将其重命名为  $a_2$ ，那么在基本块 `B3` 中对局部变量 `a` 的使用是  $a_1$  还是  $a_2$  呢，我们不能确定，因为我们无法知道程序执行时是从 `B1` 还是 `B2` 跳转到 `B3` 的。

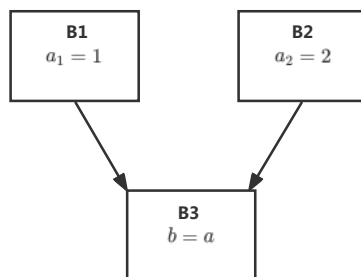


图 4.2: 重命名产生的问题

SSA 形式引入了  $\phi$  指令来解决这一问题, 如图4.3所示, 我们在 B3 的开头插入了  $\phi$  指令  $a_3 = \phi(a_1, a_2)$ , 如果从 B1 跳转到了 B3, 那么  $\phi(a_1, a_2) = a_1$ , 如果从 B2 跳转到了 B3, 那么  $\phi(a_1, a_2) = a_2$

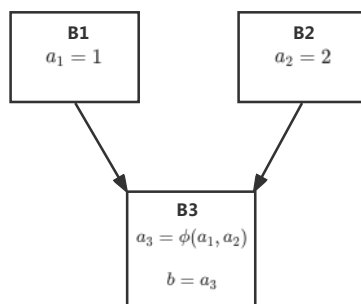


图 4.3: 静态单赋值形式

上述只是一个简单示例, 真实环境下的代码会有非常复杂的控制流, 如何在合适的位置插入  $\phi$  指令, 并且让  $\phi$  指令的数量最少就成了一个关键问题, 为了解决这一问题, 我们需要建立支配树, 计算支配边界得到合适的  $\phi$  指令插入位置, 并完成变量重命名, 下面会提供一些参考资料, 具体细节需要同学们自己探索。

**整体算法流程** 整体算法流程和介绍推荐参考[LLVM 如何构造 SSA: Mem2Reg Pass](#)。

**建立支配树** 推荐参考[支配树教程](#), 该算法网上参考资料较多, 大家也可以自行查阅其他资料。

**插入  $\phi$  指令** 推荐参考[Static Single Assignment Book](#)的 3.1.1 节和 3.1.2 节。

**变量重命名** 推荐参考[Static Single Assignment Book](#)的 3.1.3 节

**注意事项** 你在实现时不能只照着伪代码完成代码的编写, 你还需要了解为什么伪代码这么做是正确的, 我们会在线下检查时提问相关内容。

**实验结果** 如果你实现了完整的 mem2reg, 那么你的编译器可以完成如下的优化:

```

1 int main()
2 {
3     int x = 0;
4     int n = getint();
5     int i = 0, S = 0;
6     while(i < n){
7         if(i==10){
8             S = S*2;
9         }
10        S = S + 2;
11        i = i + 1;
12    }
13    return S;
14 }

```

```

1  define i32 @main(){
2      %1 = alloca i32
3      %2 = alloca i32
4      %3 = alloca i32
5      %4 = alloca i32
6      store i32 0, ptr %1
7      %5 = call i32 @getint()
8      store i32 %5, ptr %2
9      store i32 0, ptr %3
10     store i32 0, ptr %4
11     br label %6
12 6:
13     %7 = load i32, ptr %3
14     %8 = load i32, ptr %2
15     %9 = icmp slt i32 %7, %8
16     br i1 %9, label %10, label %21
17 10:
18     %11 = load i32, ptr %3
19     %12 = icmp eq i32 %11, 10
20     br i1 %12, label %13, label %16
21 13:
22     %14 = load i32, ptr %4
23     %15 = mul i32 %14, 2
24     store i32 %15, ptr %4
25     br label %16
26 16:
27     %17 = load i32, ptr %4
28     %18 = add i32 %17, 2
29     store i32 %18, ptr %4
30     %19 = load i32, ptr %3
31     %20 = add i32 %19, 1
32     store i32 %20, ptr %3
33     br label %6
34 21:
35     %22 = load i32, ptr %4
36     ret i32 %22
37 }

```

```

1  define i32 @main() {
2      %1 = call i32 @getint()
3      br label %2
4  2:
5      %3 = phi i32 [ 0, %0 ], [ %13, %10 ]
6      %4 = phi i32 [ 0, %0 ], [ %12, %10 ]
7      %5 = icmp slt i32 %3, %1
8      br i1 %5, label %6, label %14
9  6:
10     %7 = icmp eq i32 %3, 10
11     br i1 %7, label %8, label %10
12  8:
13     %9 = mul i32 %4, 2
14     br label %10
15 10:
16     %11 = phi i32 [ %9, %8 ], [ %4, %6 ]
17     %12 = add i32 %11, 2
18     %13 = add i32 %3, 1
19     br label %2
20 14:
21     ret i32 %4
22 }

```

## 4.2 进阶要求其他优化实验流程

在 Mem2Reg 中，你已经实现了支配树 (AnalysisPass) 以及 mem2reg(TransformPass)，下面你还需要实现更多的优化。这一部分属于进阶要求，我们不会提供详细的指导，但是会提供大致做法以及一些参考文献，需要同学们自行阅读并探索如何实现。

### 4.2.1 AnalysisPass

**AliasAnalysis** 指针分析，你需要实现函数 QueryAlias(Operand op1, Operand op2,...)，返回指针 op1 和 op2 指向的内存是否可能会重叠；以及 QueryInstModRef(Instruction I, Operand op)，返回指令 I 是否会读/写指针 op 指向的内存，由于 SysY 唯一会产生 ptr 类型的只有数组，所以不需要复杂的算法。

**LoopAnalysis** 循环分析，查找循环，建立循环森林等。该 Pass 网上参考资料较多，大家可以自行搜索相关资料。

**ScalarEvolution** 推荐参考 [Scalar Evolution: Change in the Value of Scalar Variables Over Iterations of the Loop](#)

**MemorySSA** 内存依赖分析，推荐参考 [llvm 的 MemorySSA 文档](#)，同时也有一些等价的实现方法，大家可以自行探索。

**LoopCarriedDependencyAnalysis** 推荐参考 [Dependence Analysis and Loop Transformations](#)。



### 4.2.2 TransformPass

**DCE** 普通的死代码消除，算法可参考该博客。

**TCO** 尾调用优化，你只需要实现尾递归转循环这一优化即可。该 Pass 算法较简单，大家可以自行设计算法来实现这一优化。

**Inline** 函数内联以及递归调用展开。对于递归调用展开，可以设置一个阈值，例如函数超过 200 行指令就不再展开。该 Pass 算法较简单，大家可以自行设计算法来实现这一优化。

**SCCP** 稀疏条件常量传播，推荐参考[Static Single Assignment Book](#)的 Part II，第 8 节。

**ADCE** 激进的死代码消除，算法可参考该博客。

**Fully Redundancy Elimination(Scalar)** 标量运算的完全冗余消除，你首先需要实现判断两条指令是否计算了相同值的函数，然后通过在支配树上进行一次 dfs 来消除完全冗余，可以参考 [llvm/Transforms/Scalar/EarlyCSE.cpp](#) 的注释进行编写

**Fully Redundancy Elimination in BasicBlock(Memory+Call)** 基本块内对内存操作和纯函数调用的完全冗余消除，需要完成的前置 Pass 可能有 AliasAnalysis。实现前置 Pass 后，具体冗余消除步骤较为简单，需要大家自行探索。

**Fully Redundancy Elimination(Memory)** 内存操作的完全冗余消除，需要完成的前置 Pass 可能有 AliasAnalysis 和 MemorySSA。可以参考 [llvm/Transforms/Scalar/EarlyCSE.cpp](#) 的注释进行编写。

**GCM** 原论文为[Global Code Motion/Global Value Numbering](#)

**SSAPRE** 原论文为[A new algorithm for partial redundancy elimination based on SSA form](#)，该优化难度较大，算法极难理解，请谨慎选择。

**LoopStrengthReduce** 循环归纳变量强度削弱，你需要实现 add, mul 运算和 GEP(getelementptr) 运算的归纳变量强度削弱，需要完成的前置 Pass 可能有 LoopAnalysis, ScalarEvolution, LoopSimplify。对于 GEP 指令，后面有一个示例可供参考。

**LICM(Scalar)** 标量运算的循环不变量外提。需要完成的前置 Pass 可能有 LoopAnalysis。该优化网上参考资料较多，大家可以自行搜索相关资料。

**LICM(Memory+Call)** 内存操作和纯函数调用的循环不变量外提，推荐阅读 llvm 源代码的 [lib/Transforms/Scalar/LICM.cpp](#) 注释来了解大致算法。需要完成的前置 Pass 可能有 LoopAnalysis, AliasAnalysis。

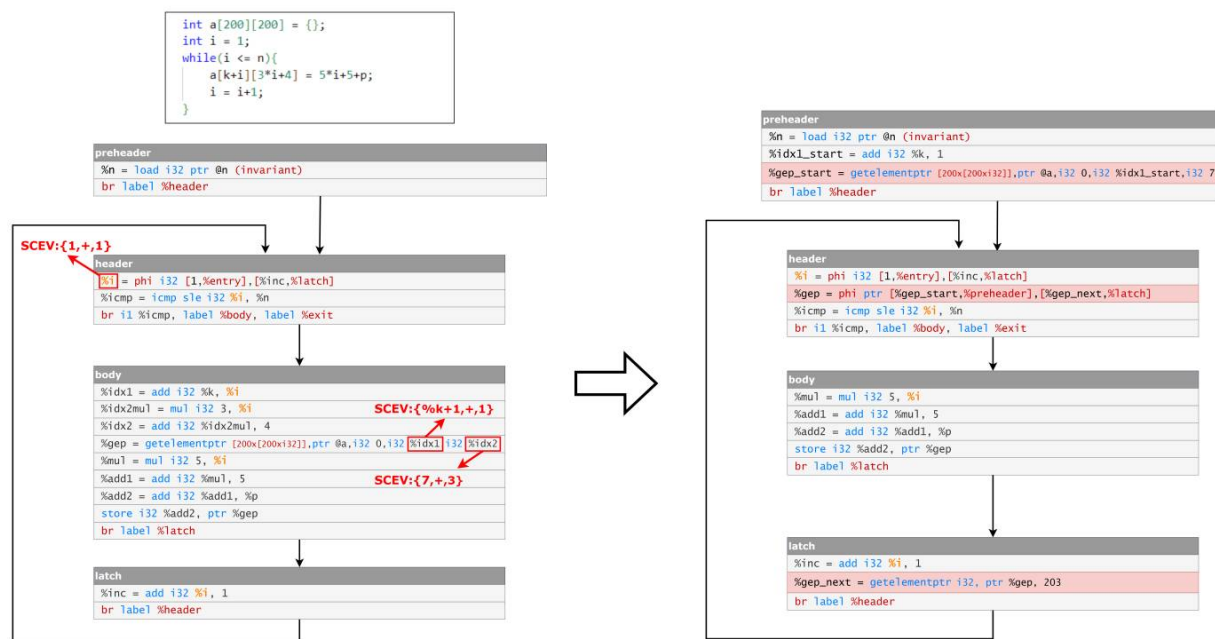


图 4.4: 循环强度削弱示例

**LoopUnroll** 循环展开, 对于常量次数的循环, 如果该常量较小, 需要将循环完全展开。对于执行次数为变量的循环, 如果循环步长为常量 (为了实现简单, 可以只考虑步长为 1 的情况), 需要将循环展开为 4 份 + 不被 4 整除的剩余部分的形式。需要完成的前置 Pass 可能有 LoopAnalysis, ScalarEvolution, LoopSimplify, LCSSA。完成前置 Pass 后, 对循环的变换只是工作繁琐, 但是思路较简单, 需要同学们自行探索具体实现方法。

**LoopFuse** 循环合并, 推荐参考Dependence Analysis and Loop Transformations和Revisiting Loop Fusion and its place in the loop transformation framework。需要完成的前置 Pass 可能有 LoopAnalysis, ScalarEvolution, AliasAnalysis, LoopCarriedDependencyAnalysis, LoopSimplify, LCSSA。

**LoopParallel** 循环自动并行化, 推荐参考Dependence Analysis and Loop Transformations。需要完成的前置 Pass 可能有 LoopAnalysis, ScalarEvolution, AliasAnalysis, LoopCarriedDependencyAnalysis, LoopSimplify, LCSSA。你可能还需要使用 pthread 或者系统调用编写自己的多线程链接库。该优化难度较大, 算法理解难度中等, 但是工作量较大, 请谨慎选择。

**Vectorize** 自动向量化, 推荐参考llvm 自动向量化文档和SLP 矢量化介绍来了解大致流程, 然后根据你的理解自己调研一些论文并进行实现。该优化难度极大, 请谨慎选择。

## 5 评分标准

### 5.1 基本要求评分标准

1. 完成控制流图的构建，得 0.5 分。
2. 完成删除不可达指令和不可达基本块的优化，得 0.5 分。
3. 完成删除没有 use 的 def 变量及相关指令的优化，得 1 分。
4. 完成 def 和 use 均在同一基本块特殊情况的 mem2reg，得 1 分。

### 5.2 进阶要求评分标准

#### 5.2.1 mem2reg 评分标准

1. 完成支配树的构建，得 0.6 分。
2. 完成 phi 指令插入，得 0.7 分。
3. 完成变量重命名，得 0.7 分。

#### 5.2.2 AnalysisPass 评分标准

AnalysisPass 的实现不作为评分依据，但是 AnalysisPass 是某些 TransformPass 实现的前置任务，当然如果你实现了 AnalysisPass 但是没有相对应的 TransformPass，我们会在线下检查时按照你的实现情况酌情给分。

#### 5.2.3 TransformPass 评分标准

这里首先对下方表格进行一些说明：

1. 部分 TransformPass 由于前置 pass(例如 adce 需要支配树作为前置) 已经给了一定分数，所以这里给分会稍低。
2. 有一些辅助用的 TransformPass 不会在下方表格中出现，例如 LoopSimplify, LCSSA 等，这些 Pass 的分数已经算在了需要这些辅助 pass 的关键优化中。当然，如果你正确实现了它们但是没有成功实现下方表格中的优化，线下检查时助教会根据你的完成情况给部分分数。
3. 对于自动向量化和自动并行化，会根据向量化/并行化的程度进行给分。
4. 对于窥孔优化，指令合并等比较容易的优化，不计算在本次实验的进阶要求分数中。
5. 溢出实验总分最多 3 分。
6. 根据往年编译系统实现赛的情况，一等奖队伍的中端优化大概是实现了下面列举的 80% 左右，如果你有兴趣参加明年的编译系统实现赛，可以用下面的表格作为一个参考。

优化名称	得分
普通的死代码消除 (dce)	0.5
尾递归转循环 (tco)	0.7
函数内联 + 递归调用展开	1
稀疏条件常量传播 (sccp)	1.5
激进死代码消除 (adce)	1(如果没完成 dce 可得 1.5 分)
完全冗余消除 (标量)	1.5
基本块内完全冗余消除 (内存操作 +call)	1
完全冗余消除 (内存操作)	1
部分冗余消除 SSAPRE (标量)	3 (可溢出实验总分)
部分冗余消除 GCM (标量)	2
部分冗余消除 SSAPRE (内存操作)	2 (可溢出实验总分)
部分冗余消除 GCM (内存操作)	1.5 (可溢出实验总分)
循环不变量外提 (标量)	1.5
循环不变量外提 (内存操作 +call)	1
循环归纳变量强度削弱	3
循环展开 (将可以展开的循环展开成 4 份 + 常量循环完全展开)	3
LoopFuse	3
自动并行化	5 (可溢出实验总分)
自动向量化	6 (可溢出实验总分)