实验二：LLVM IR死代码删除优化

* 1. 实验目的

1.实现LLVM IR的死代码删除优化，减少代码运行时执行的代码条数。

2.保持程序语义不变，确保优化后代码的正确性。

1.2 实验环境

1. 硬件环境：鲲鹏开发板。

2. 软件环境：openEuler操作系统。

3. 测试环境：提供一组Cminus语言的测试代码。

1.3 实验内容

实现死代码删除优化，包括常真常假跳转的删除和单来源Phi指令的删除。

1.4 实验过程

1.4.1 常真常假跳转删除

对于每个BasicBlock，检查其末尾的跳转指令，如果其跳转指令是Cond\_Br且Cond字段为一常量，则将Cond\_Br替换为Br，并修改对应BasicBlock间的关系。

1.4.2 单来源Phi指令删除

对于每个BasicBlock，检查开头的Phi指令，如果Phi指令只有两个操作数，则意味着这个BasicBlock只有一个可能的跳转来源，直接删除这个Phi指令。

* 1. void DeadCodeDeletion() 简单的死代码删除

常真常假条件跳转删除

|  |
| --- |
| *// 1.常真常假分支删除*  **for** (**auto** f : m\_->get\_functions())  {  **for** (**auto** bb : f->get\_basic\_blocks())  {  *// 对于每个bb观察它的br指令是否为条件跳转*  **auto** terminstr = bb->get\_terminator();  **auto** brinstr = **dynamic\_cast**<BranchInst \*>(terminstr);  **if** (brinstr && brinstr->is\_cond\_br())  {  **auto** condition = terminstr->get\_operand(0);  **auto** truebb = **static\_cast**<BasicBlock \*>(terminstr->get\_operand(1));  **auto** falsebb = **static\_cast**<BasicBlock \*>(terminstr->get\_operand(2));  **auto** condition\_value = cast\_constantint(condition);  *// condition\_value不是常数int*  **if** (!condition\_value)  {  *// 下一个块*  **continue**;  }  **auto** curr\_bb = bb;  *// 常真常假分支删除*  **auto** value = condition\_value->get\_value();  printf("%d\n", value);  **if** (value == 1 || value == 0)  {  *// 后继所有BasicBlock删除该前驱并修改phi指令*  printf("READY TO DELETE\n");  **for** (**auto** succbb : curr\_bb->get\_succ\_basic\_blocks())  {  *// 考察所有的后继块前驱，删除跳转不到的后继块的前驱*  succbb->remove\_pre\_basic\_block(curr\_bb);  *// 修改 常真跳转的falsebb 和常假跳转的truebb 里面的的 curr\_bb phi*  **if** ((value == 1 && succbb != truebb) || (value == 0 && succbb != falsebb))  {  *// 修改phi指令，删除*  **for** (**auto** instr : succbb->get\_instructions())  {  **if** (instr->is\_phi())  {  std::cout << succbb->get\_name() << std::endl;  **for** (**int** i = 1; i < (instr->get\_num\_operand()); i = i + 2)  {  **if** (instr->get\_operand(i) == curr\_bb)  {  *// 删掉<value , curr\_bb>这一条*  instr->remove\_operands(i - 1, i);  }  }  }  }  }  }  *// 删除分支并改为直接跳转到对应的BB*  bb->delete\_instr(brinstr);  bb->get\_succ\_basic\_blocks().clear();  **if** (value == 1)  {  BranchInst::create\_br(truebb, bb);  bb->add\_succ\_basic\_block(truebb);  }  **if** (value == 0)  {  BranchInst::create\_br(falsebb, bb);  bb->add\_succ\_basic\_block(falsebb);  }  *// std::cout <<bb->get\_name() + " OK" << std::endl;*  }  }  **else**  {  *// std::cout << "NOT A COND\_BR" <<std::endl;*  }  }  } |

优化前后的IR对比（举一个例子）并辅以简单说明：

|  |
| --- |
| test.cminus:  int x;  int main(void){  int a;  float b;  x = 1;  a = 1+2;  b = (a > x) + a;  return 0;  } |

优化前test.ll

|  |
| --- |
| @x = global i32 0  declare i32 @input()  declare void @output(i32)  declare void @outputFloat(float)  declare void @neg\_idx\_except()  define i32 @main() {  label\_entry:  %op0 = alloca i32  %op1 = alloca float  store i32 1, i32\* @x  %op2 = add i32 1, 2  store i32 %op2, i32\* %op0  %op3 = load i32, i32\* %op0  %op4 = load i32, i32\* @x  %op5 = icmp sgt i32 %op3, %op4  %op6 = zext i1 %op5 to i32  %op7 = load i32, i32\* %op0  %op8 = add i32 %op6, %op7  %op9 = sitofp i32 %op8 to float  store float %op9, float\* %op1  %op10 = alloca i32  store i32 0, i32\* %op10  %op11 = load i32, i32\* %op10  ret i32 %op11  } |

优化后test.ll

|  |
| --- |
| @x = global i32 0  declare i32 @input()  declare **void** @output(i32)  declare **void** @outputFloat(**float**)  declare **void** @neg\_idx\_except()  define i32 @main() {  label\_entry:  %op0 = alloca i32  %op1 = alloca **float**  store i32 1, i32\* @x  store i32 3, i32\* %op0 *// a = 1 + 2 优化成 a = 3*  store **float** 0x4010000000000000, **float**\* %op1 *// b = (a > x) + a 优化成 b = 4;*  %op10 = alloca i32  store i32 0, i32\* %op10  ret i32 0  } |

**死代码删除**

实现思路：

主要实现了两个简单功能：

2.常真常假跳转的删除

主要思路是对于每个BasicBlock，检查其末尾的跳转指令，如果其跳转指令是Cond\_Br且Cond字段为一常量，则将Cond\_Br替换为Br，并修改对应BasicBlock间的关系。

3.单来源Phi指令的删除

主要思路是对于每个BasicBlock，检查开头的Phi指令，如果Phi指令只有两个操作数，则意味着这个BasicBlock只有一个可能的跳转来源，直接删除这个Phi指令，并将向前寻找值，替换需要用到Phi指令得到值的变量。

相应代码：

2.文件:

ConstantPropagation.hpp: 头文件

ConstantPropagation.cpp: C++文件

3.函数：

void ConstPropagation::DeadCodeDeletion(): 实现上述两个功能

优化前后的IR对比（举一个例子）并辅以简单说明：

test.cminus:

|  |
| --- |
| int main(void){  int a;  a = 1;  if(a == 1){  return 0;  }  else{  return 1;  }  } |

优化前test.ll:

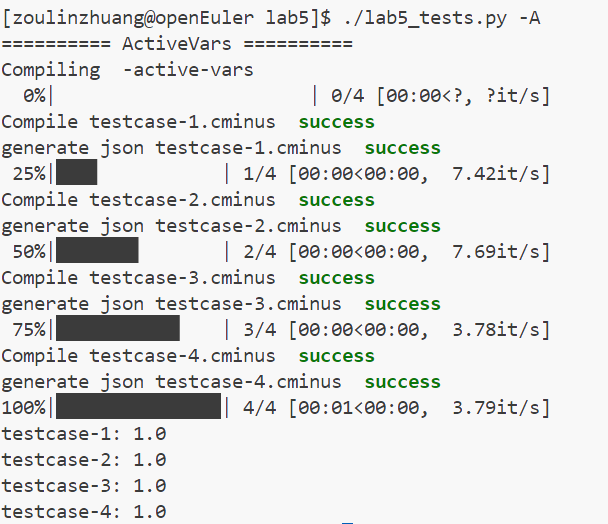
|  |
| --- |
| declare i32 @input()  declare void @output(i32)  declare void @outputFloat(float)  declare void @neg\_idx\_except()  define i32 @main() {  label\_entry:  %op0 = alloca i32  store i32 1, i32\* %op0  %op1 = load i32, i32\* %op0  %op2 = icmp eq i32 %op1, 1  %op3 = zext i1 %op2 to i32  %op4 = icmp ne i32 %op3, 0  br i1 %op4, label %label\_trueBB0, label %label\_falseBB0  label\_trueBB0: ; preds = %label\_entry  %op5 = alloca i32  store i32 0, i32\* %op5  %op6 = load i32, i32\* %op5  ret i32 %op6  br label %label\_endBB0  label\_falseBB0: ; preds = %label\_entry  %op7 = alloca i32  store i32 1, i32\* %op7  %op8 = load i32, i32\* %op7  ret i32 %op8  br label %label\_endBB0  label\_endBB0: ; preds = %label\_trueBB0, %label\_falseBB0  ret i32 0  ret i32 0  } |

优化后test.ll:

|  |
| --- |
| declare i32 @input()  declare **void** @output(i32)  declare **void** @outputFloat(**float**)  declare **void** @neg\_idx\_except()  define i32 @main() {  label\_entry:  %op0 = alloca i32  store i32 1, i32\* %op0  br label %label\_trueBB0 *// 只会跳转到label\_trueBB0*  label\_trueBB0: ; preds = %label\_entry  %op5 = alloca i32  store i32 0, i32\* %op5  ret i32 0  br label %label\_endBB0  label\_falseBB0: *// label\_falseBB0此时不可达，其实可以进行删除，但本次试验没有1去实现*  %op7 = alloca i32  store i32 1, i32\* %op7  ret i32 1  br label %label\_endBB0  label\_endBB0: ; preds = %label\_trueBB0, %label\_falseBB0  ret i32 0  ret i32 0  } |

1.5 实验结果

通过死代码删除优化，成功移除了LLVM IR中的无效指令，提高了代码运行效率。



1.6 实验总结

本次实验加深了对LLVM IR优化技术的理解，掌握了死代码删除优化的具体实现方法，并通过实验验证了优化效果。

姓名：邹林壮 学号：202208040412