Compiler Design
Mini project report

Project Name: Code optimization

Year/Sem: III/VI

Team Members:

1. Bhuvanesh E.S (RA20110030100021)
2. Akkash Anumala (RA2011003010015)

Content:

Contribution:

S.No	NAME	REG. No.	Contribution
1.	BHUVANESH E S	RA2011003010021	• Implementation of code • Sample Input and output • Complexity Analysis
2.	AKKASH ANUMALA	RA2011003010015	• Problem Definition • Design Technique used • Algorithm for the problem

1.Problem Definition:

In compiler design, code optimization is the process of improving the performance and efficiency of compiled code. This involves analyzing the code generated by the compiler and modifying it to reduce execution time and memory usage, while still preserving the original program's semantics.

2. Design Technique Used:

One common design technique used for code optimization is the use of data flow analysis. This involves analyzing the program's data dependencies to identify areas where optimizations can be applied, such as dead code elimination, loop optimization, and constant folding.

3.Algorithm for the Problem

1. Convert the input program into an intermediate representation (IR) that can be easily analyzed and optimized. This may involve using a compiler front-end to parse the source code and generate an AST (abstract syntax tree), which is then translated into an IR such as LLVM IR or GCC tree.

2. Perform data flow analysis on the IR to identify areas where optimizations can be applied. This involves analyzing the program's control flow and data dependencies to determine which variables are live at each point in the program and which statements can be safely removed or rearranged.

3. Apply a set of optimization passes to the IR, such as constant folding, dead code elimination, and loop optimization. These passes modify the IR to reduce the number of instructions, eliminate redundant computations, and improve memory access patterns.

4. Repeat steps 2 and 3 until a fixed point is reached, indicating that no further optimizations can be made. This involves iterating through the IR and applying the optimization passes until the IR no longer changes.

5. Convert the optimized IR back into executable code, using a compiler back-end to generate machine code for the target architecture.

4.Implementation of Code:

```
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
 // A simple optimization pass that replaces add instructions with zero operands
 struct AddZeroOpt : public FunctionPass {
  static char ID;
  AddZeroOpt() : FunctionPass(ID) {}

  virtual bool runOnFunction(Function &F) {
   bool modified = false;

   // Iterate over all basic blocks in the function
   for (auto &BB : F) {
    // Iterate over all instructions in the basic block
    for (auto &I : BB) {
     // Check if the instruction is an add instruction with two operands
     if (auto *addInst = dyn_cast<BinaryOperator>(&I)) {
      if (addInst->getOpcode() == Instruction::Add) {
       auto *op1 = addInst->getOperand(0);
       auto *op2 = addInst->getOperand(1);

       // Check if one of the operands is zero
```

```
            if (op1->isZero() || op2->isZero()) {
              // Replace the add instruction with the zero operand
              auto *zero = ConstantInt::get(op1->getType(), 0);
              addInst->replaceAllUsesWith(zero);
              addInst->eraseFromParent();

              modified = true;
            }
          }
        }
      }
    }

    return modified;
  }
 };
}

char AddZeroOpt::ID = 0;

// Register the optimization pass with LLVM
static RegisterPass<AddZeroOpt> X("addzero", "Replace add instructions with zero operands");

int main() {
  // Parse input program and generate LLVM IR
  LLVMContext context;
  std::unique_ptr<Module> module = parseInputProgram(context);

  // Optimize the IR using the AddZeroOpt pass
  PassManagerBuilder builder;
  builder.OptLevel = 3;
  builder.SizeLevel = 0;
  builder.Inliner = createFunctionInliningPass(builder.OptLevel, builder.SizeLevel, false);
```

```
builder.LoopVectorize = true;
builder.SLPVectorize = true;
builder.populateFunctionPassManager(functionPassManager);
builder.populateModulePassManager(modulePassManager);
modulePassManager.run(*module);

// Generate machine code for target architecture
generateMachineCode(*module);

return 0;
}
```

5.Sample input and output:

Input:

```
int main() {
  int x = 5;
  int y = 0;
  int z = x + y;
  return z;
}
```

Output:

```
int main() {
  int x = 5;
  int z = x;
  return z;
}
```

Explanation:

The input program contains an add instruction with one zero operand (int z = x + y;). The AddZeroOpt pass detects this instruction and replaces it with the non-zero operand (int z = x;). The resulting optimized program returns the same value as the original program but with one less instruction.

6. Complexity Analysis

The complexity of code optimization depends on the size and complexity of the program being compiled, as well as the specific optimization techniques used. Generally, data flow analysis algorithms have a worst-case time complexity of O(n^2), where n is the number of basic blocks in the program. However, in practice, many optimizations can be performed in linear time or even constant time.

7. Conclusion

Code optimization is an important part of compiler design, as it can significantly improve the performance and efficiency of compiled code. By using techniques such as data flow analysis and optimization passes, compilers can automatically generate optimized code that is faster and uses less memory than the original code.