Local Common Subexpression Elimination

Report of Assignment 2

Chih-Yung Liang
Graduate Institute of Networking and Multimedia
National Taiwan University
r05944012@csie.ntu.edu.tw

Shih-Kai Lin
Dept. of Computer Science & Information Engineering
National Taiwan University
r05922043@csie.ntu.edu.tw

1 METHODOLOGY

To reuse common subexpressions in a local region, the transformation is implemented as a *BasicBlockPass* that iterates each basic block of given code region. A map data structure, named *ExprHash*, mapping expressions formed with operands and operation kind to already existed instructions is used to record reusable expressions. The key-value pair *ExprHash* stores is:

Key : OpCode (as *unsigned*) and 5 Operands (as *Value* *)

Value : Exist instr. that evaluates the expression (as *Value* *)

It is noticeable that for expressions not requiring as many as 5 operands, the extra unused operand fields of the key of *ExprHash* are marked empty with *nullptr*.

The procedure of the transformation is to iterate each reusable instruction in the basic block under processing and store the expression the instruction evaluates in *ExprHash*, where reusable ones are instructions of types defined in Table 1. For expressions already in *ExprHash*, instead of re-insert into *ExprHash*, use exist evaluator instructions to replace all uses of duplicated expression evaluators. However, store instructions between load instructions with common expression can modify the data to load and invalidates exist evaluated load results.

1.1 Memory Modification Handling

When the basic block iteration reaches a memory writing instruction, the transformer has to check if the written address aliases any address read by memory reading instructions in *ExprHash*. It has to be conservative, so only memory reading instructions, which must be load instructions, that are sure not to alias the memory writing instruction, which must be a store instruction, are kept in *ExprHash*; otherwise, they are removed from the map. The alias analysis result is taken from *BasicAliasAnalysis* provided by LLVM itself, which basically distinguishes different globals, stack allocations, and heap allocations. Indexes into arrays with statically differing subscripts are also reported as *NoAlias* by the alias analysis result.

Table 1: Types of Reusable Instruction

BinaryOperator	CmpInst	ExtractElementInst
GetElementPtrInst	InsertValueInst	InsertElementInst
PHINode	SelectInst	ShuffleVectorInst
CastInst	ExtractValueInst	LoadInst

Table 2: Available Tests

noeffect1	noeffect2	noeffect3	eli-loads1	eli-loads2
eli-loads3	eli-adds1	eli-adds2	eli-adds3	eli-ptr1
eli-ptr2	eli-ptr3	eli-ptr4		

2 BUILDING & TESTING

2.1 Building the Program with CMake

```
$ # Building
$ mkdir build && cd build
$ cmake .. && make
$ # Usage
$ opt -load Transform/MyPass.so -my-cse ...
```

2.2 Testing CSE with CTest

The test script automatically compiles C sources into LLVM IR with *Clang*, performs the transformation with *Opt*, and compares the difference made by the transformation using *Vimdiff*. The usage of the script is:

```
$ make CSE-test-<test name>
$ # For example:
$ make CSE-test-eli-adds1
```

Table 2 lists available tests for the transformation, where the source for each test is placed under `Tests/CSE`. Figure 1 illustrates the test result of *eli-adds1*.

Figure 1: Test result for the test *eli-adds1*