Local Value Numbering Implementation using LLVM's Framework

Tasmia Rahman

1 Introduction:

Local Value Numbering is a technique that eliminates usage of redundant expressions. Firstly, it determines the expressions that are redundant in a basic block and then replaces the redundant evaluations with the reuse of a previously computed value.

I have implemented this algorithm using llvm framework. I have worked on this implementation in three phases. At first, I implemented the basic version of LVN algorithm. Afterwards, I implemented the extended version of LVN. In this version, LVN includes several other optimizations, for example - Commutative operations, Constant folding and Algebraic identities. Both of these versions have one limitation. They work for instructions that contain only two operands. However, I later extended my implementation so that it can work for more than two operands. While, this extension needs more works, right now it works for several cases that I tested.

2 The Algorithm:

Let's consider the following four instructions in a basic block:

a = b + c ---> 1

b = a - d ---> 2

c = b + c ---> 3

d = a - d ---> 4

Local Value Numbering algorithm determines that Instruction 4 is a redundant expression as the same expression has already been evaluated in Instruction 2. It can be noticed that instruction 3 is however not an redundant expression even though it already has been evaluated in instruction 1 since one of the operands of that instruction gets updated in Instruction 2. LVN is able to determine all these crucial factors.

In addition to that, extended version of LVN can determine that Commutative operations such as a * b and b * a, a + b and b + a may not be evaluated more than once. However, it also recognizes that a - b and b - a is not a commutative operation. LVN also includes constant folding where if both of the operands are constants, instead of loading the operands LVN can perform the operation and fold the answer directly into the code. Lastly, LVN can apply algebraic identities such as a + 0 = a, a * 0 = 0, etc.

3 LLVM Framework:

Using LLVM framework, I have built three optimization passes. These are: lvn, extendedlvn and lvnv2.

Below are the descriptions of each of these passes:

4 Passes:

4.1 Pass 1: lvn:

This pass implements the basic version of Local Value Numbering algorithm. The pass is in the LVN.cpp file inside LVN folder. Since, LLVM works on the IR of the source code, we first need to create the IR with the following command:

../build/bin/clang -emit-llvm -S -o lvn.ll lvn.c

After that, we can load the pass with the following command:

../build/bin/opt -load ../build/lib/libLVN.so -lvn -S <lvn.ll> lvn.opt.ll

Please note that libLVN.so is created when we build the pass. When the pass is loaded, the optimized IR will be written to lvn.opt.ll. If we compare the IR between the lvn.ll(unoptimized) and lvn.opt.ll(optimized) files, we will see that the redundant expressions are eliminated and replaced with a store instruction that stores the previously computed value to the destination variable.

Below are the redundant instructions:

```
%16 = load i32, i32* %1, align 4
%17 = load i32, i32* %4, align 4
%18 = sub nsw i32 %16, %17
store i32 %18, i32* %4, align 4
```

Below is the instruction that replaces the redundant expression above:

```
store i32 %12, i32* %4, align 4
```

You can notice that this store instruction stores the value from %12 as %12 has the previously computed value. Thus it eliminates the redundant expressions and optimizes our code.

4.2 Pass 2: extendedlvn:

This pass implements the extended version of Local Value Numbering. I tested this pass with the input1.c file. So, first we need to create the IR of this source code with the following command:

../build/bin/clang -emit-llvm -S -o input1.ll input1.c

After that we can load the pass with the following command:

../build/bin/opt -load ../build/lib/libExtendedLVN.so -extendedlvn -S <input1.ll> input1.opt.ll

If we compare input1.ll and input1.opt.ll, we can see that it has been optimized intensely, considering crucial details of the extended version of the algorithm.

4.3 Pass 3: lvnv2:

In this pass, I extended the basic version of Local Value Numbering so that it works for instructions that have more than two operands. Both of the previous passes do not work for instructions that have more than two operands. Hence, I wanted to extend the implementation to include this feature. There are more works needed so that this extension can work for all the features of local value numbering, specially with the implementation of Pass2. So for now, this implementation works for certain cases but in future I plan to extend it so that it works for all cases.

I tested this pass with input.c file and the pass worked for the case that is demonstrated in that source code. So, just as before, let's create the IR with the following command:

../build/bin/clang -emit-llvm -S -o input.ll input.c

After that, we load the pass with the following command:

../build/bin/opt -load ../build/lib/libLVNV2.so -lvnv2 -S <input.ll> input.opt.ll

If we compare the IR between the input.ll(unoptimized) and input.opt.ll(optimized) files, we will see that the redundant expressions are eliminated and replaced with a store instruction that stores the previously computed value to the destination variable.

In input.c we have,

```
int main(){
        int a = 6, b = 3, c, d, e, f;
        a = b + c + d + e + f;
        f = b + c + d + e;
        return 0;
}
```

Below are the redundant instructions for the instruction 3 of the source code:

```
%17 = load i32, i32* %3, align 4
%18 = load i32, i32* %4, align 4
%19 = add nsw i32 %17, %18
%20 = load i32, i32* %5, align 4
%21 = add nsw i32 %19, %20
%22 = load i32, i32* %6, align 4
%23 = add nsw i32 %21, %22
store i32 %23, i32* %7, align 4
```

Below is the instruction that replaces the redundant expression above:

```
store i32 %14, i32* %7, align 4
```

5 Conclusion:

The source codes of the optimization passes and also the test inputs are provided along with this report. Please refer to the tutorial by Ksenia Burova in order to set up llvm as well as for instruction on how to compile and build the passes.