Compiler Engineering
CS6383 - Sep 2019
Mini Assignment 5:

SCEV Mini Assignment Version 2

INSTRUCTOR: Dr. Ramakrishna Upadrasta

Report By:
Vijay Tadikamalla - CS17BTECH11040
Tungadri Mandal - CS17BTECH11043

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

SCEV (Scalar Evolution)

In general, by **Scalar Evolution,** we mean how the value of **scalars changes** in a program with the execution of code.
Scalar Evolution in LLVM is an **analysis of the LLVM program** that helps its clients reason about **induction variables**. It does this by mapping SSA values into objects of a SCEV type, and implementing algebra on top of it.
LLVM uses Scalar Evolution as an analysis pass. This helps LLVM to prepare loops for advanced optimizations and efficient execution.

How SCEV works?

To observe what SCEV does first we have to emit the LLVM file. On the ***'.ll'*** file we run the opt using the following command.

```
opt -analyze -scalar-evolution [FILENAME]
```

On running this command it outputs on what variables the SCEV optimizations can be applied and what the resulting SCEV expression would be.

SCEV operates on each function separately. It takes the mangled function name and displays the relevant expressions of that function under “Classifying expressions for: [Function_Name]”. Then it takes each induction variable and displays the relevant SCEV expression.

The SCEV also outputs important information like backedge-taken counts and other such relevant information.

Subset-Sum

The functions are placed in the same order in which they are defined in the original source file. So, for the subset-sum program, the first function in the SCEV output is also printSubset.

Then we can observe that it analyzes each variable individually including the parameter variables if they are included. In our case, it begins with the induction variable i. First, it shows that it was written in its phi form which is a hypothetical function used to represent such variables in SSA form.

```
%i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
```

In the next line, we can see the derived recurrence expression. For this case, it is the expression

```
-->  {0,+,1}<nuw><nsw><%for.cond> U: [0,-2147483648) S: [0,-2147483648)
```

Which is basically 0,1,2 and so on. Next, we can see all the derived variables also have the same SCEV expression.

```
%idxprom = sext i32 %i.0 to i64
-->  {0,+,1}<nuw><nsw><%for.cond> U: [0,2147483648) S: [0,2147483648)
Exits: (zext i32 (0 smax %size) to i64)
LoopDispositions: { %for.cond: Computable }
```

We also see terms like **nuw, nsw** and so on. NSW stands for no signed wrap and NUW stands for no unsigned wrap. This means that the result is undefined if unsigned and/or signed overflow, respectively, occurs. **Zext** stands for zero extend and **Sext** stands for sign extend. These abbreviations are enough for our cause. We also observe things like **loop disposition**. It is an enumeration and there are 3 possibilities. Computable, loop variant and loop invariant. Loop variant and invariant are self implied and computable means that though it is variant, it can be computed.

Next, we observe the extra loop execution count information.

```
Determining loop execution counts for: @_Z11printSubsetPii
Loop %for.cond: backedge-taken count is (0 smax %size)
Loop %for.cond: max backedge-taken count is 2147483647
Loop %for.cond: Predicated backedge-taken count is (0 smax %size)
```

The first piece of information we observe is that **backedge-taken** count which is the number of times the loop went back to the loop statement and this is generally the

number of times the loop was executed. The next line is **max backedge taken** count which is generally the maximum back edges allowed and depends on the type of the variable. The next one is **predicted back edge** count.

Next, we will demonstrate the power of the SCEV expression. Note how the expression for accessing the array id in the for loop of the function subset-sum is reduced to the following expression.

Matrix Chain Multiplication

```
    for (i = 1; i < n; i++)
        m[i][i] = 0;
```

We are accessing the **(i,i) th** element only and not any other indices of the array. We can make it easier to access by using SCEV to optimize how we access it.

```
%8 = load i32, i32* %i, align 4
%idxprom = sext i32 %8 to i64
%9 = mul nsw i64 %idxprom, %3
%arrayidx = getelementptr inbounds i32, i32* %vla, i64 %9
%10 = load i32, i32* %i, align 4
%idxprom1 = sext i32 %10 to i64
%arrayidx2 = getelementptr inbounds i32, i32* %arrayidx, i64 %idxprom1
store i32 0, i32* %arrayidx2, align 4
```

So, we can see that currently to get **arrayidx2** we gave to calculate **arrayidx** and **idxprom** and we may need even more steps to calculate those. But we can optimize it using SCEV.

```
%arrayidx2 = getelementptr inbounds i32, i32* %arrayidx, i64 %idxprom1
-->  {((4 + (4 * (zext i32 %n to i64))<nuw><nsw> + %vla),+,
      (4 + (4 * (zext i32 %n to i64))<nuw><nsw>)<nuw><nsw>}
```

So, as we can see how the array element is represented easily and simply through the recurrence expression. So, we don't have to calculate all the **arrayidx** and **idxprom** values. We can directly use the **vla** value and hence we can skip all the intermediate steps.

```
for (L = 2; L < n; L++)
{
      for (i = 1; i < n - L + 1; i++)
      {
            j = i + L - 1;
```

So, for this piece of code, we want to calculate the **index j** which we will be later using. But this is inside 2 loops.

```
%add9 = add nsw i32 %i.1, %L.0
-->  {{3,+,1}<nuw><%for.cond3>,+,1}<nw><%for.cond6> U: [3,-1) S:[3,-1)
%sub10 = sub nsw i32 %add9, 1
-->  {{2,+,1}<nuw><nsw><%for.cond3>,+,1}<nw><%for.cond6>
%idxprom11 = sext i32 %i.1 to i64
```

So, as we observe from the final result of **sub10** how we can easily calculate the value of **j** using the recurrence expression while maintaining it's relativity with its loop conditions.

Transitive Closure Of a Graph

```
for (i = 0; i < V; i++)
      for (j = 0; j < V; j++)
            reach[i][j] = graph[i][j];
```

So, here we want to access **(i,j)th index** of the **reach** array. But the indices to access the array is in a nested loop.

```
%arrayidx9 = getelementptr inbounds [4 x i32], [4 x i32]* %arrayidx7, i64
0, i64 %idxprom8
  -->  {{%reach,+,16}<nsw><%for.cond>,+,4}<nsw><%for.cond1>
```

So, SCEV uses the above-simplified recurrence expression. Note how we only use the loop conditions and we don't have to calculate **arrayidx7** and **idxprom8** and other relevant intermediate steps. We are directly using reach to get to our required indices.

M-Coloring Problem

```
    for (int i = 0; i < V; i++)
    printf(" %d ", color[i]);
```

Here we want to replace the variable we are passing to the ***printf*** which is inside a loop. Again, the **induction variable i** or the array element address is not **loop invariant** but they are **computable** and this can be easily seen in the SCEV output.

```
%arrayidx = getelementptr inbounds i32, i32* %color, i64 %idxprom
  -->  {%color,+,4}<nsw><%for.cond> U: full-set S: full-set
Exits: (16 + %color)       LoopDispositions: { %for.cond: Computable
}
```

Notice the **LoopDisposition** which tells us how the **variable V** and hence the condition though not being loop invariant is computable and hence can be resolved.

Hamiltonian Cycle Backtracking

```
for (int v = 1; v < V; v++)
```

Similarly, here we want to resolve the **induction variable v**. We want to replace the normal **phi** function and replace it with something more simple.

```
%v.0 = phi i32 [ 1, %if.end ], [ %inc, %for.inc ]
  -->  {1,+,1}<nuw><nsw><%for.cond>
```

Hence, we replace it with the corresponding recurrence expression.

Topological Sorting

```
      for (i = adj[v].begin(); i != adj[v].end(); ++i)
            if (!visited[*i])
                  topologicalSortUtil(*i, visited, Stack);
```

Here we would like to simplify the induction variable i. But please note that this is a list. So, the induction variable ***i***'s layout in memory will not be contiguous. Hence, we can't access the next element using pointer arithmetic and hence SCEV wouldn't be able to resolve it and simplify it.

```
Graph::Graph(int V)
{
      this->V = V;
      adj = new list<int>[V];
}
```

```
-->  {(32 + %call),+,24}<nw><%arrayctor.loop>
```

The above code is the constructor of the **graph class.**

We can see that we are allocating space to an array of lists of size **v**.
So, SCEV optimizes it by forming a recurrence. In this recurrence, there is an increment of **24 bytes**, i.e **sizeof(list<int>()) == 24 bytes.**

References

- http://llvm.org/devmtg/2018-04/slides/Absar-ScalarEvolution.pdf