Write-and-Do-Mini-Assignment#5: SCEV Mini Assignment Version 2

Sagar Jain
CS17BTECH11034

September 19, 2019

For all the given programs the following pattern is followed:

1. Convert .cpp file to llvm IR.
   `clang -O0 -Xclang -disable-O0-optnone -emit-llvm -S foo.cpp -o foo.ll`
2. Run mem2reg optimisation on IR code.
   `opt -mem2reg foo.ll -S > foo.memopt.ll`
3. Get SCEV analysis of the code.
   `opt -analyze -scalar-evolution foo.memopt.ll`
4. Try different optimisations on the code.
   `opt -indvars foo.memopt.ll -S > foo.indvar.ll`
   `opt -loop-reduce foo.memopt.ll -S > foo.lsr.ll`
   `opt -loop-vectorize foo.memopt.ll -S > foo.loopV.ll`
5. Following this use `diff` to get the optimisations introduced by opt.

Results

1. `hamiltonian-cycle-backtracking.cpp`

   - **SCEV Analysis**
     Scev classifies all variables, recognises basic induction variables and computes general induction variables.The following are examples of induction variables are recognised/computed by scev:

     The path pointer increases by 4 every iteration, since int is four bytes.

```
{%path,+,4}<nsw><%for.cond> U: full-set S: full-set  Exits: <<Unknown>>
    LoopDispositions: { %for.cond: Computable }
```

Wherever possible, scev also calculates the range and exit value of variables.

```
--> {2,+,1}<nuw><nsw><%for.cond> U: [2,7) S: [2,7)  Exits:
    <<Unknown>>  LoopDispositions: { %for.cond: Computable }
--> {0,+,1}<nuw><nsw><%for.cond> U: [0,6) S: [0,6)  Exits: 5
    LoopDispositions: { %for.cond: Computable }
```

- **Uses in optimisations**

  (a) Both **indvars** and **loop-reduce** at many occasions reduces an instruction by avoiding the sext operation.Example:

```
<   %idxprom6 = sext i32 %i.0 to i64
<   %arrayidx7 = getelementptr inbounds i32, i32* %path, i64
    %idxprom6
<   %2 = load i32, i32* %arrayidx7, align 4
<   %cmp8 = icmp eq i32 %2, %v
---
>   %arrayidx7 = getelementptr inbounds i32, i32* %path, i64
    %indvars.iv
>   %3 = load i32, i32* %arrayidx7, align 4
>   %cmp8 = icmp eq i32 %3, %v
```

```
<   %idxprom6 = sext i32 %i.0 to i64
<   %arrayidx7 = getelementptr inbounds i32, i32* %path, i64
    %idxprom6
<   %2 = load i32, i32* %arrayidx7, align 4
---
>   %scevgep = getelementptr i32, i32* %path, i64 %lsr.iv
>   %2 = load i32, i32* %scevgep, align 4
```

2. `matrix-chain-multiplication.cpp`

- **SCEV Analysis**

  The following are examples of induction variables are recognised/computed by scev:

  We have loops nested to a depth of three and operating over a 2d array, this gives scev a lot of opportunity to express the address of the values in terms of different induction variables. As we can see the pointer depends on the index of the row and also the coloumn this is an example of a **chained recurrence**.

```
%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>}<%for.cond> U: [0,-3)
    S: [-9223372036854775808,9223372036854775805)  Exits: (4 + (4 *
    (zext i32 %n to i64))<nuw><nsw> + ((zext i32 (-1 + (1 smax
    %n))<nsw> to i64) * (4 + (4 * (zext i32 %n to
    i64))<nuw><nsw>)<nuw><nsw>) + %vla)  LoopDispositions: {
    %for.cond: Computable }
```

The exit conditions help optimisations figure out the trip count, given here is an exit which itself is a scev expression.

```
{{2,+,1}<nuw><%for.cond6>,+,1}<nuw><%for.cond15> U: [2,0) S: [2,0)
   Exits: (2 + ({0,+,1}<nuw><nsw><%for.cond6> smax
   {{1,+,1}<nuw><nsw><%for.cond3>,+,1}<nw><%for.cond6>))<nuw>
   LoopDispositions: { %for.cond15: Computable, %for.cond6: Variant,
   %for.cond3: Variant }
```

- **Uses in optimisations**

  (a) Using ***uglygeps*** and ***bitcasts*** `loop-reduce` reduces both the number of instructions and the total number of cycles. Example:

```
<   %idxprom19 = sext i32 %i.1 to i64
<   %6 = mul nsw i64 %idxprom19, %1
<   %arrayidx20 = getelementptr inbounds i32, i32* %vla, i64 %6
<   %idxprom21 = sext i32 %k.0 to i64
<   %arrayidx22 = getelementptr inbounds i32, i32* %arrayidx20, i64
    %idxprom21
<   %7 = load i32, i32* %arrayidx22, align 4
<   %add23 = add nsw i32 %k.0, 1
<   %idxprom24 = sext i32 %add23 to i64
<   %8 = mul nsw i64 %idxprom24, %1
<   %arrayidx25 = getelementptr inbounds i32, i32* %vla, i64 %8
---
>   %uglygep14 = getelementptr i8, i8* %lsr.iv1013, i64 %lsr.iv2
>   %uglygep1415 = bitcast i8* %uglygep14 to i32*
>   %13 = load i32, i32* %uglygep1415, align 4
>   %idxprom24 = sext i32 %lsr.iv7 to i64
>   %14 = mul nsw i64 %idxprom24, %1
>   %arrayidx25 = getelementptr inbounds i32, i32* %vla, i64 %14
```

3. `m-coloring-problem.cpp`

- **SCEV Analysis**

  The following are examples of induction variables are recognised/computed by scev:

  The **isSafe** function checks all the neighbours of a single node, so it gets the

starting point from the variable %s and then adds four to get every consequent element.

```
%arrayidx = getelementptr inbounds i32, i32* %s, i64 %idxprom
  --> {%s,+,4}<nsw><%for.cond> U: full-set S: full-set  Exits: ((4 *
      (zext i32 (0 smax %size) to i64))<nuw><nsw> + %s)
      LoopDispositions: { %for.cond: Computable }
```

- **Uses in optimisations**

  (a) Using the scev output we can see that gere the number of instructions in actually halved when using `loop-reduce`.

```
<   %idxprom = sext i32 %v to i64
<   %arrayidx = getelementptr inbounds [4 x i8], [4 x i8]* %graph,
    i64 %idxprom
<   %idxprom1 = sext i32 %i.0 to i64
<   %arrayidx2 = getelementptr inbounds [4 x i8], [4 x i8]*
    %arrayidx, i64 0, i64 %idxprom1
<   %0 = load i8, i8* %arrayidx2, align 1
<   %tobool = trunc i8 %0 to i1
---
>   %scevgep2 = getelementptr i8, i8* %scevgep1, i64 %lsr.iv
>   %1 = load i8, i8* %scevgep2, align 1
>   %tobool = trunc i8 %1 to i1
```

4. `subset-sum.cpp`

- **SCEV Analysis**

  The following are examples of induction variables are recognised/computed by scev:

  Most of the loops in this program are trivial and the only scev expressions present are those related to indices and pointers, for example.

```
%arrayidx26 = getelementptr inbounds i32, i32* %s, i64 %idxprom25
 --> {((4 * (sext i32 %ite to i64))<nsw> +
     %s)<nsw>,+,4}<nsw><%for.cond> U: full-set S: full-set  Exits: ((4
     * (zext i32 ((-1 * %ite) + (%s_size smax %ite)) to
     i64))<nuw><nsw> + (4 * (sext i32 %ite to i64))<nsw> + %s)
     LoopDispositions: { %for.cond: Computable }
```

- **Uses in optimisations**

  (a) Similar optimisations by indavar and loop-reduce.

```
<   %idxprom = sext i32 %i.0 to i64
<   %arrayidx = getelementptr inbounds i32, i32* %A, i64 %idxprom
<   %0 = load i32, i32* %arrayidx, align 4
```

```
<   %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
    ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 5, i32 %0)
---
>   %arrayidx = getelementptr inbounds i32, i32* %A, i64 %indvars.iv
>   %1 = load i32, i32* %arrayidx, align 4
>   %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
    ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 5, i32 %1)
```

```
<   store i32 %5, i32* %arrayidx24, align 4
<   %idxprom25 = sext i32 %i.0 to i64
<   %arrayidx26 = getelementptr inbounds i32, i32* %s, i64
    %idxprom25
<   %6 = load i32, i32* %arrayidx26, align 4
<   %add27 = add nsw i32 %sum, %6
---
>   store i32 %6, i32* %arrayidx24, align 4
>   %scevgep1 = getelementptr i32, i32* %s, i64 %lsr.iv
>   %7 = load i32, i32* %scevgep1, align 4
>   %add27 = add nsw i32 %sum, %7
```

5. `topological-sorting.cpp`

- **SCEV Analysis**
  The following are examples of induction variables are recognised/computed by scev:
  This program consists of just two loops and both of them are quite simple the induction variables and addresses are the only scev expressions.

```
--> {(8 + %call)<nsw>,+,24}<nuw><%arrayctor.loop> U: full-set S:
    full-set  Exits: (8 + (24 * ((-8 + (8 * (sext i32 %V to
    i64)))<nsw> /u 8)) + %call)  LoopDispositions: { %arrayctor.loop:
    Computable }
--> {(32 + %call),+,24}<nw><%arrayctor.loop> U: full-set S: full-set
     Exits: (32 + (24 * ((-8 + (8 * (sext i32 %V to i64)))<nsw> /u
    8)) + %call) LoopDispositions: { %arrayctor.loop: Computable }
```

- **Uses in optimisations**

(a)

```
<   %idxprom8 = sext i32 %i3.0 to i64
<   %arrayidx9 = getelementptr inbounds i8, i8* %call, i64 %idxprom8
<   %6 = load i8, i8* %arrayidx9, align 1
<   %tobool = trunc i8 %6 to i1
---
>   %scevgep = getelementptr i8, i8* %call, i64 %lsr.iv
>   %5 = load i8, i8* %scevgep, align 1
>   %tobool = trunc i8 %5 to i1
```

6. `transitive-closure-of-a-graph.cpp`

- **SCEV Analysis**
  The following are examples of induction variables are recognised/computed by scev:

  Since we are given a 4x4 graph, every row is 16 bytes, we can observe this in the scev of the pointer in the *transitiveClosure* function which iterates over the rows.

```
%arrayidx = getelementptr inbounds [4 x i32], [4 x i32]* %graph, i64
    %idxprom
 --> {%graph,+,16}<nsw><%for.cond> U: full-set S: full-set  Exits:
    {%graph,+,16}<nsw><%for.cond>  LoopDispositions: { %for.cond1:
    Invariant, %for.cond: Computable }
```

  Simillarly for the pointer which iterates over the coloumns of the reach matrix.

```
%arrayidx37 = getelementptr inbounds [4 x [4 x i32]], [4 x [4 x
    i32]]* %reach, i64 0, i64 %idxprom36
--> {%reach,+,16}<nsw><%for.cond16> U: [0,-15) S:
    [-9223372036854775808,9223372036854775793)  Exits:
    {%reach,+,16}<nsw><%for.cond16>  LoopDis
```

- **Uses in optimisations**

  (a) `loop-reduce` removes unecessary casting which reduces the overall number of instructions in the loops.

```
<   %idxprom31 = sext i32 %k.0 to i64
<   %arrayidx32 = getelementptr inbounds [4 x [4 x i32]], [4 x [4 x
    i32]]* %reach, i64 0, i64 %idxprom31
<   %idxprom33 = sext i32 %j.1 to i64
<   %arrayidx34 = getelementptr inbounds [4 x i32], [4 x i32]*
    %arrayidx32, i64 0, i64 %idxprom33
<   %3 = load i32, i32* %arrayidx34, align 4
<   %tobool35 = icmp ne i32 %3, 0
---
>   %scevgep6 = getelementptr [4 x [4 x i32]], [4 x [4 x i32]]*
    %lsr.iv4, i64 0, i64 0, i64 %lsr.iv1
>   %5 = load i32, i32* %scevgep6, align 4
>   %tobool35 = icmp ne i32 %5, 0
```