Loop Strength Reduction

ZENG TAO

1.Problem Statement

Overall performance of a program is generally sensitive to the computation complexity inside loop bodies of the program since they execute repeatedly and takes significant amount of execution time. Thus, there exists various optimization llvm passes target loops, such as (LICM) Loop Invariant Code Motion, LSR (Loop Strength Reduction), and Induction Variable Elimination. However, this project will focus on the Induction Variable Identification and Loop Strength Reduction. Strength Reduction Optimization is a commonly used transformation which transfer expensive computations to cheaper ones. For example, a simple strength reduction algorithm will replace the instruction x = y * 8 with x = y << 3. This project will transfer the expensive computation such as multiplications and divisions to additions and subtractions inside the loop to promote the performance. Due to the time limitation, the scale of this project is limited to the loop strength reduction in simple natural loops.

According to the definition in the lecture slides, a basic induction variable is a variable X whose only definition inside the loop are assignments of the form X = X + c or X = X – c, where c is either a constant or a loop invariant variable. Based on this, an induction variable is either a basic induction variable B, or a variable defined once within the loop, whose value is a linear function of some basic induction variable at the time of the definition, such as A = c1 * B + c2.

From the above definition of induction variable, we observe that if the basic induction variable B increase by step c for each iteration: B = B + c, then the increase of induction variable A will be c*c1. Therefore, if we can initialize the A outside the loop, and increase A by c*c1 for each iteration, then the transformed instructions will be equivalent to the original ones in terms of the execution result, but it will have a higher performance.

The goal of this project is to reduce the strength of computation in the induction variables inside the natural loop. To achieve the goal, we first linear scan the instructions inside the loop to identify the basic induction variables, and then recursively scan the loop to find all the induction variables defined by the linear function of basic induction variables.

2. Methodology

To identify the induction variables in a loop, we first denote the basic induction variable i = i + c by triples: <i, 1, c>, more generally, an induction variable k = i * a + b can be denoted as: <i, a, b>. The algorithm of identify induction variables is shown as follows:

Linear scan loop body to find all basic induction variables.

Do

Scan loop to find all variables k with one assignment of form k = j*b, where j is an induction variable<I, c, d>, and make k an induction variable with triple <I, c*b, d>

Scan loop to find all variables k with one assignment of form k = j + b or j – b, where j is an induction variable<I, c, d>, and make k an induction variable with triple <I, c, b+d>

Util no more induction variables found.

To better demonstrate the loop strength reduction algorithm in this project, a simple example will be used. The following code is an unoptimized loop:

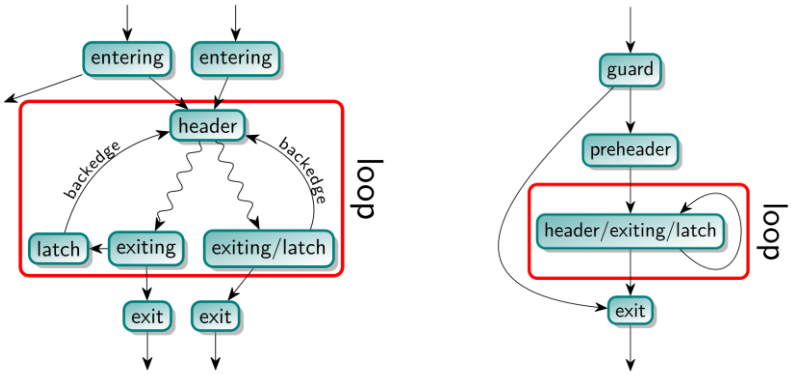
```
for(int i = 0;  i < 10; i += 2) {
  int j = 3 * i + 2;
  a[j] = a[j] - 2;
}
```

We observe from the above sample code that integer variable j is repeatedly computed for every iteration. Each iteration takes one multiplication and one addition for computing j. However, since j is a linear function of i, it is possible to calculate an initial value for j outside the loop, and then increase the j by a pre-computed stride in every iteration of the loop. The following are the transformed code to compute j inside the loop:

```
int i = 0;
int j = 2; // j = 3 * 0 + 2;
while(i < 10){
    j = j + 6; //j = j + 2 * 3;
    a[j] = a[j] - 2;
    i = i + 2;
}
```

After the optimization, the computation of j takes only one addition for each iteration. The above optimized code will be much more efficient than the original version.

The loop in a program could be complicated, as the following graph shown, it could have multiple entering blocks or exiting blocks, and multiple back edges. To make our transformation easier, a loop simplify transformation pass will be performed before the start of induction variable identification. After the transformation conducted successfully, all the loops in the program will be in form of a single preheader, a single backedge(which implies that there is a single latch), dedicated exits. That is not exit block for the loop has a predecessor that is outside the loop. This also implies that all exits blocks are dominated by the loop header.



3. Implementation

3.1 Induction Variable Identification

To perform the strength reduction, we have to identify all the induction variables inside the loop first. Our implementation will adopt a three-values tuple data structure in std to represent an induction variable, which is (basic induction variable, multiple factors, additive factor), and then use the DenseMap in llvm to store all the induction variables found. The key to the map is the instruction, and the value is the three-values tuple.

In the above example code, the basic induction variable i will be pushed into the map as an entry I ➔ (I, 1, 0). Later on, the induction variable j as a usage of the I will also be pushed into the map as an entry j ➔ (i, 3, 2). The multiple coefficients could be easily calculated by multiply the coefficient of the j and the additive number of the basic induction variable.

In the implementation of this project, it is found that the basic induction variables can be found through scan the preheader of the simple loop in SSA form. Thus, we first identify all the basic induction variables by scanning all the Phi nodes in the preheader, and then recursively scan the loop body to identify all other induction variables using the following two rules:

if an instruction of binary operation has been found in form of m = c * j, where j is an induction variable in the map represented as (I, a, b), then insert m => (I, a * c, b)

if an instruction of binary operation has been found in form of m = j + c, where j is an induction variable in the map represented as (I, a, b), then insert m => (I, a, b + c)

The algorithm will terminate when there is no change found in the map. For example, in the above simple loop, the basic induction variable i ➔ (i, 1, 0) will be inserted into the map at the first iteration, and then the induction variable j ➔(i, 3, 2), which is expressed as linear function of i will be inserted into the map at the second iteration. The algorithm will stop after the third iteration because no more induction variables found.

3.2 Strength Reduction

The strength reduction operation becomes straightforward after the loop simplification. It is guaranteed that all the Phi nodes in the loop header have exactly two incoming values, one is from the loop body and the other one is from outside. Thus, it is safe to take the latter one as the initial value of the induction variables in the loop header. In this project, we first iterate through the induction variables in the map, and then iterate the Phi nodes in the header to check if the variable in the phi node has been used by the induction variable. If it has, a new phi node will be inserted into the preheader to compute the initial value of the induction variable based on the values of the corresponding Phi node.

After the initial values of all induction variables has been computed in the preheader, the definition of induction variable in the loop bodies will be transformed to less expensive computation such as addition and subtract. This project iterates the all the induction variable instructions and recompute the stride based on the values of basic induction variable and the values stored in the tuple. The original instruction will be replaced by the new created add instruction. The following code shows the process of the transformation:

```
for (auto& indvar : IndVarMap) {
    tuple<Value*, int, int> t = indvar.second;
    if (PhiMap.count(indvar.first) &&
       (get<1>(t) != 1 || get<2>(t) != 0)){//not a basic indvar
        for (auto& I : *b_body) {
            if (auto op = dyn_cast<BinaryOperator>(&I)) {
```

```
                Value* lhs = op->getOperand(0);
                Value* rhs = op->getOperand(1);
                if (lhs == get<0>(t) || rhs == get<0>(t)) {
                    IRBuilder<> body_builder(&I);
                    tuple<Value*, int, int> t_basic = IndVarMap[&I];
                    int new_val = get<1>(t) * get<2>(t_basic);
                    PHINode* phi_val = PhiMap[indvar.first];
                    Value* new_incoming = body_builder.CreateAdd(phi_val,
                        ConstantInt::getSigned(phi_val->getType(), new_val));
                    phi_val->addIncoming(new_incoming, b_body);
                }
            }
        }
    }
}
```

4. Benchmarks

An opensource benchmark tool hyperfine has been adopted in this project to evaluate the performance of the optimized program. Hyperfine is a user-friendly benchmark tool with the features of warming up running, statistical analysis across multiple runs, and result exporting to various formats. Hyperfine will automatically perform at least 10 benchmark runs and measure at least 3 seconds, however, user can also customize those parameters according to the requirement such as following:

```
hyperfine --min-runs 10 --max-runs 50 './original.out' './optimized.out'
```

This Project has been evaluated in the following way:

1. Compiles all the benchmarks to the assembly code without any optimization.
2. Compiles all the benchmarks to the bitcode in SSA form, and perform the loop simplify pass and strength reduction pass on the bitcode.
3. Compiles all the optimized bitcode to assembly code related to the target machine.
4. Iterate all pairs of the unoptimized and optimized version of the output files, and conduct the performance comparison with hyperfine.

The routine of the performance evaluation is shown as the following shell script:

```
#conduct the performance evaluation with hyperfine.
for f in *.c
do
  filename="${f%.*}"
  opt_out=opt_m2r_${filename}.out
  orignal_out=${filename}.out

  hyperfine  './'${opt_out}  './'${orignal_out} -i --export-markdown
${filename}_report.md
done
```

The following are the 18 benchmarks we have evaluated with the transformation pass, except for the command that we marked as red, the optimized program gained performance improvement, the range of the performance improve varies from 2% to 81%. For the failed cases, one possible reason is that the transformation pass failed to identify induction variable correctly and insert the update instructions multiple times, which will result in the decease of the performance.

Command	Mean [ms]	Min [ms]	Max [ms]	Relative
`./opt_libminver.out`	1.7 ± 1.0	0.7	9.5	1
`./libminver.out`	1.7 ± 0.8	0.9	8.6	1.02 ± 0.76
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_crc_32.out`	2.1 ± 1.1	1.1	11	1
`./crc_32.out`	2.3 ± 1.1	1.4	18	1.09 ± 0.76
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_libedn.out`	1.9 ± 1.0	1	17.3	1
`./libedn.out`	3.1 ± 1.3	2	20.2	1.61 ± 1.05
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_libhuffbench.out`	1.4 ± 0.9	0.5	10.2	1
`./libhuffbench.out`	2.5 ± 1.2	1.5	15.6	1.84 ± 1.48
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_libminver.out`	1.7 ± 1.0	0.7	9.5	1
`./libminver.out`	1.7 ± 0.8	0.9	8.6	1.02 ± 0.76
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_libnsichneu.out`	2.2 ± 1.2	1.2	16.8	1
`./libnsichneu.out`	2.5 ± 1.3	1.3	19.2	1.13 ± 0.84
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_m2r_libst.out`	**1.9 ± 1.0**	**0.9**	**13.9**	**1.05 ± 0.74**
`./libst.out`	1.8 ± 0.8	0.9	10.6	1
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_libstatemate.out`	2.3 ± 1.5	1	14.4	1.12 ± 1.37
`./libstatemate.out`	**2.0 ± 2.1**	**0.9**	**48.5**	**1**
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_libud.out`	2.4 ± 1.4	1.4	19.9	1
`./libud.out`	3.1 ± 1.0	2	10.6	1.29 ± 0.86
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_libwikisort.out`	1.9 ± 1.0	0.9	11.2	1
`./libwikisort.out`	2.0 ± 1.0	1	9.9	1.05 ± 0.77
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_matmult-int.out`	2.6 ± 0.9	1.5	10	1
`./matmult-int.out`	3.5 ± 1.4	2.1	11.3	1.39 ± 0.74
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_md5.out`	2.2 ± 1.0	1.1	11.3	1
`./md5.out`	2.5 ± 1.0	1.5	13.3	1.14 ± 0.71
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**

`./opt_mont64.out`	2.1 ± 0.8	1.1	8	1
`./mont64.out`	2.5 ± 1.3	1.5	21.2	1.19 ± 0.78
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_nbody.out`	1.9 ± 1.0	0.9	10.6	1
`./nbody.out`	1.9 ± 1.2	1	16.9	1.03 ± 0.81
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_nettle-aes.out`	2.0 ± 1.1	0.6	8.9	1
`./nettle-aes.out`	2.1 ± 0.8	1.2	9.9	1.09 ± 0.75
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_nettle-sha256.out`	2.0 ± 1.1	0.9	14.1	1
`./nettle-sha256.out`	2.2 ± 0.9	1.3	10.4	1.13 ± 0.77
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_primecount.out`	1.9 ± 1.0	0.9	10.1	1
`./primecount.out`	3.3 ± 1.2	2.3	13.2	1.74 ± 1.13
Command	**Mean [ms]**	**Min [ms]**	**Max [ms]**	**Relative**
`./opt_tarfind.out`	2.2 ± 1.1	1.1	10.1	1
`./tarfind.out`	2.7 ± 1.4	1.3	10.9	1.43 0.47

5.Conclusion

From the above experiment result, we can conclude that by adopting the loop strength reduction, the expensive computations in the loop are replaced with less expensive ones. Therefore, the overall performance of the program could be improved significantly. However, in some cases, the pass failed to identify the induction variables correctly and insert duplicated instructions in the loop preheader. These issues will be addressed in the future work.