MP5

CS 426 FALL 2024

Avancha, Ritvik
Ren, Hao

Lab Section
15th December 2024
TA Name : Sathia, Vimarsh

0.1 Introduction

0.1.1 Loop Analysis

A back edge is defined to be an edge from n to d where d dominates n. Also d is the header of the loop.

After finding the backedge and the header, we starts from n to reverse-iterate (go from n to its predecessors and so on) through the loop using dfs, until we reach the loop header. During dfs, each time we reach a block belonging to the loop, we add the basic block to the loop. When adding a basic block to a loop, we also compute its preheader, and its associated exit block. An exit block is a block whose has a successor not in the loop. A preheader is a predecessor of the header which is not in the loop.

This allows us to compute all loops, where each loop consists of its internal basic blocks, its preheader and its exit blocks.

0.1.2 Loop Invariant Code Motion

After Loop Analysis (LA), for each loop, we hoist all the loop-invariant instructions out of the loop. The only thing that we need to check is if an instruction is loop invariant with respect to the loop. We only consider computation instruction, load instruction and store instruction.

Then we check if all the operands for an instruction are defined outside of the loop. Since it's in SSA form (meaning even in unit test, we do a mem2reg pass before this pass), each use have a unique def. Hence we trace the def and compute the associated instruction. We check if the instruction is inside the loop. If it is, the current instruction is not loop-invariant. Otherwise, we still can't determine if it's loop-invariant because of memory aliasing.

Hence, if it's a memory instruction, we check if alias exist in the loop by using alias analysis object. Also we check if there are any call instruction or invokation instructions that have side effects to this memory instruction. If there are no aliasing also this instruction is safe to hoist, then we hoist the instruction to the preheader. The meaning for an instruction to be "safe-to-hoist" is that it either has no side-effect or it dominates all exits.

The detailed logics is in the following diagram.

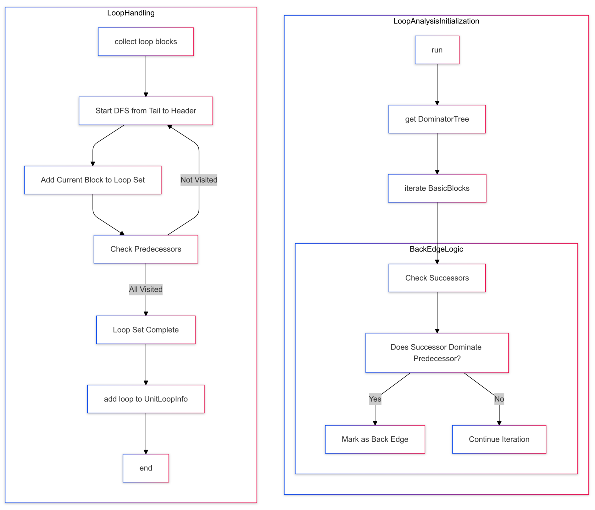

Figure 1
Loop Analysis Process Overview

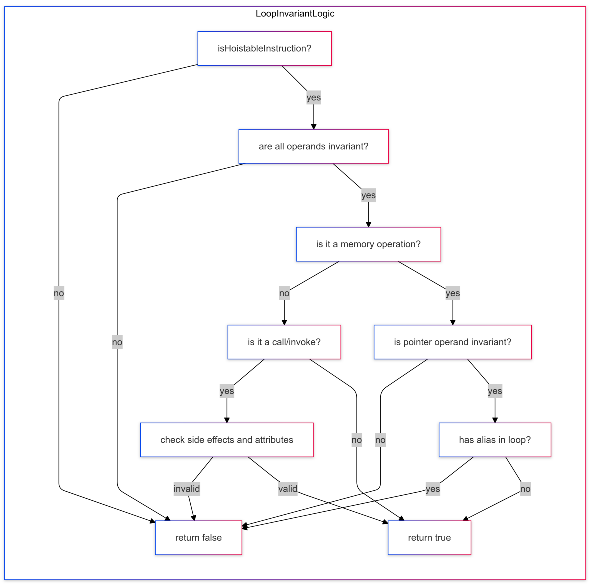

Figure 2
LICM Overview

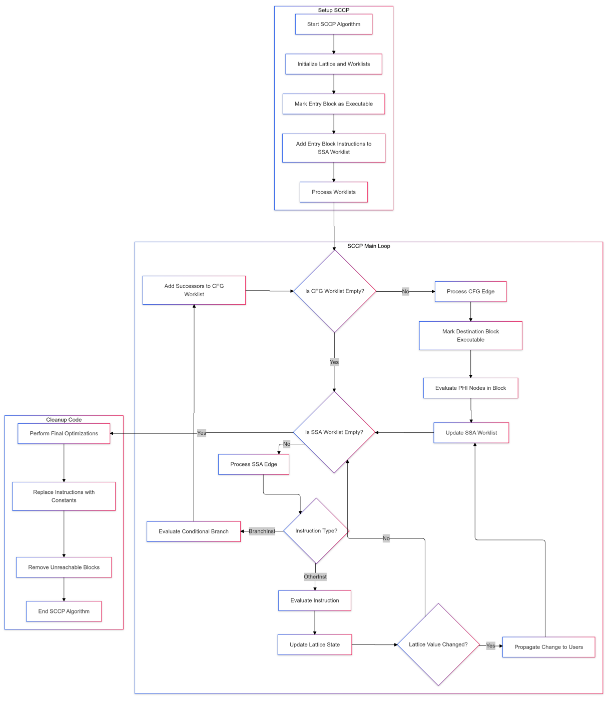
Setup SCCP
Start SCCP Algorithm
Initialize Lattice and Worklists
Mark Entry Block as Executable
Add Entry Block Instructions to SSA Worklist
Process Worklists
SCCP Main Loop
Add Successors to CFG Worklist
Is CFG Worklist Empty?
No
Process CFG Edge
Mark Destination Block Executable
Yes
Evaluate PHI Nodes in Block
Cleanup Code
Perform Final Optimizations
Yes
Is SSA Worklist Empty?
Update SSA Worklist
No
Process SSA Edge
Replace Instructions with Constants
Evaluate Conditional Branch
Branchinst
Instruction Type?
No
Remove Unreachable Blocks
Otherinst
Evaluate Instruction
End SCCP Algorithm
Update Lattice State
Lattice Value Changed?
Yes
Propagate Change to Users

FIGURE 3
SCCP Overview

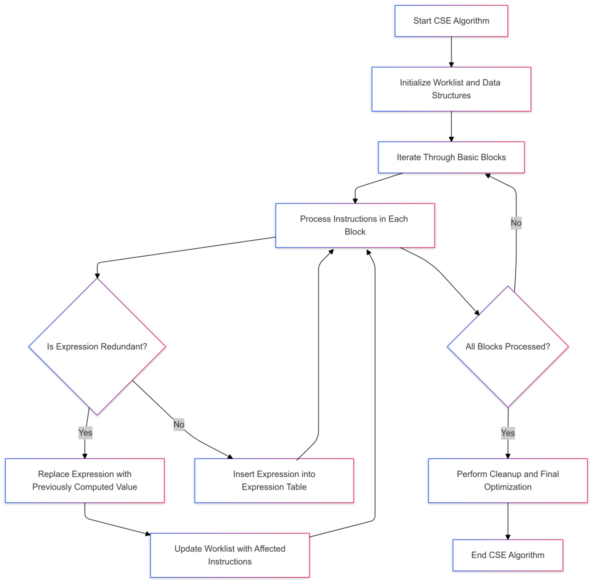
Start CSE Algorithm
Initialize Worklist and Data Structures
Iterate Through Basic Blocks
Process Instructions in Each Block
No
Is Expression Redundant?
All Blocks Processed?
Yes
No
Yes
Replace Expression with Previously Computed Value
Insert Expression into Expression Table
Perform Cleanup and Final Optimization
Update Worklist with Affected Instructions
End CSE Algorithm

FIGURE 4
CSE Overview

0.1.3 SPARSE CONDITIONAL CONSTANT PROPAGATION

In SCCP, we want to determine if constant definitions can be propagated downstream to their uses, which can allow for more informed analysis of the Control Flow Graph (CFG) to prune unused branches and remove redundant constant definitions. For a motivating example, consider the following: int x = 2 + 2; if (x == 4) return 100; else return 0; Once we propagate the constant x into the if condition, the condition should always evaluate to true thus the else condition would never be taken.

0.1.4 COMMON SUBEXPRESSION ELIMINATION

In CSE, we substitute all uses of each available instruction to the earliest defined expression. For example, x = a + b, y = a + b, z = y; will be reduced ot x = a + b, z = x; This eliminates all instructions which has available expressions.

0.2 STATUS OF THE WORK

We implemented LA, LICM, SCCP, including all optimizations and requirements listed in the document. The specifics progress will be introduced in the later sections.

We also implemented CSE, including almost all optimizations. Except that we miss one optimization opportunity where aliasing happen but the aliased instructions store the same value to the memory location. This will be introduced later in the CSE discussion section. But besides this our program successfully handle other optmization opportunities

Correctness is ensured through the use of a set of varying test cases and running our optimization on a set of benchmarks. Each test case is manually created, designed to fully verify the logic and return meaningful values.

0.3 COLLOBORATION

Hao implemented LA and LICM while Ritvik observed. Ritvik implemented SCCP while Hao observed. Hao and Ritvik implemented and debugged different parts of CSE interchangeably. Similarly, Hao and Ritvik worked on the Python benchmarking scripting and Makefile build and test automation together.

0.4 HOW TO RUN END-TO-END PIPELINE

To compile the optimization, run tests and run benchmark simply do

```
make all
```

in the project directory

0.5 EXPERIMENT RESULT

TABLE 1
LICM & LA (Table 1)

Program	Hoisted Loads	Hoisted Stores	Hoisted Computation
almabench.c	0	0	1
huffbench.c	2	0	1
puzzle.c	0	0	0
n-body.c	0	0	0
whetstone.c	1	0	9
partialsums.c	0	0	1
nesting.c	0	0	0
spectral-norm.c	0	0	0
doloop.c	0	1	1
PR491.c	0	0	0
matmul.c	0	0	3
one-iter.c	0	0	0
lpbench.c	0	0	2
fannkuch.c	0	0	0
random.c	0	0	0
nsieve-bits.c	0	0	0
ffbench.c	0	0	12
recursive.c	0	0	0

TABLE 2
SCCP (Table 2)

Program	Instructions Removed	Unreachable Blocks	Constants Replaced
almabench.c	0	0	0
huffbench.c	0	0	0
puzzle.c	0	0	0
n-body.c	0	0	0
whetstone.c	0	0	0
partialsums.c	2	0	2
nesting.c	0	0	0
spectral-norm.c	0	0	0
doloop.c	0	0	0
PR491.c	0	0	0
matmul.c	0	0	0
one-iter.c	0	0	0
lpbench.c	0	0	0
fannkuch.c	0	0	0
random.c	0	0	0
nsieve-bits.c	3	0	3
ffbench.c	5	0	5
recursive.c	10	0	10

- The LILCM experiment are ran using the mem2reg and unit-licm passes.
- The SCCP experiment are ran using the mem2reg and unit-sccp passes.
- The CSE experiment are ran using the mem2reg and unit-cse passes.

The reason for using mem2reg before the unit-* pass is that transforming the code to SSA form and promoting memory instruction to register instructions before optimization can hugely increase the optimization space for SCCP and CSE. To set the experiment condition the same for each optimization, we also add mem2reg to LICM. Although it doesn't imply signficant gain for LICM.

TABLE 3
CSE (Table 3)

Program	Instructions Removed
almabench.c	85
huffbench.c	49
puzzle.c	4
n-body.c	24
whetstone.c	31
cse.c	2
partialsums.c	1
spectral-norm.c	13
doloop.c	1
matmul.c	11
lpbench.c	34
fannkuch.c	12
LA_test.c	4
nsieve-bits.c	9
ffbench.c	28
recursive.c	5

0.6 DISCUSSION

0.6.1 LOOP ANALYSIS AND LICM

This example (combining tests/correctness/{licm2, licm3, licm4}.c to tests/correctness/licm5.c) fanstically shows that hoisted load, hoisted store and hoisted compute work. And it clearly shows that we missed an opportunity to optimize out aliasing loads if no stores are aliasing with them. This missing opportunity is caused by a conservative optimization to skip all aliasing memory instructions. This also justifies the loop analysis, which is the backbone of LICM. Because without LA working, LICM can never accurately handle optimizations. The detailed justifcation is in comments before each functions in the following example.

```
// hoist 1 store & 2 compute (addition & icmp) successfully. Store & compute are
    both loop-variant also no aliasing.
int do_loop1(int x, int y, int* z) {
    do{
        x = *z + y;
    } while(x < 0);
    return x;
}
// missed opportunity. There are two loads with aliasing. But no store. Hence, it's
    okay to have aliasing if all memory access are loads. But we didn't optimize
    this aggressively. (We skip as long as there are aliasing).
int do_loop2(int x, int y, int* z, int* p) {
    int pp;
    do {
        x = *z + y;
        pp = *p + y;
    } while (x < 0);
    return x;
}
// hoist 1 load and 1 compute (addition) because load & compute are both loop-
    invariant also no aliasing.
int do_loop3(int x, int y, int* z) {
    int pp = 2;
    do{
        *z = pp;
    } while(x < 0);
```

```c
    return x;
}

int main() {
    int x = 2;
    int y = 3;
    return do_loop1(2, 3, &x) + do_loop2(3, 2, &y, &x) + do_loop3(1, 1, &x);
}
```

LISTING 1
Example for LA and LICM

0.6.2 SCCP

The example below illustrates the motivation behind the Sparse Conditional Constant Propagation optimization pass. The constant definition of x (x = 2+2) can be propagated down to the conditional check (if (x == 4) –> if (2+2 == 4)). This allows for the else code block to be eliminated as it is dead code since that control path will not be taken by the program. Additionally, this example also demonstrates room for improvement since this optimization propagates constant definitions but could be further optimized by performing constant folding to reduce computational overhead at runtime. For the sake of keeping optimization passes concise and independent, we choose not to bundle a constant folding implementation into this pass.

```c
#include <stdio.h>

int constant_replace() {
    int x = 2 + 2;
    if (x == 4) {
        return 100;
    } else {
        return 0;
    }
}

int main() {
    return constant_replace();
}
```

LISTING 2
Example of Constant Replacement

0.6.3 CSE

The example tests/correctness/cse-demo.c excellently shows that removed instruction statistics can be contributed from different sources of IR patterns. Through this, and the following explanation in the comments, we can clearly see how to justify the optimization strategies and what's missing.

```c
// missed opportunity: since apparently *x = p and *x = q are aliasing, I just skip
    this. But it can be optimized because the values storing to it is the same. But
    I found it hard to generalize this case, so I skip this opportunity.
void test1(char c, int* x, volatile int* y) {
    int p = c;
    int q = c;
    *x = p;
    *x = q;
    *y = p;
    *y = q;
}
// remove 1 instruction because d = x + y is dominant by c = x + y
void test2(float x, float y, int* m, int* n) {
    int c;
    int d;
    c = x + y;
    *m = c;

    d = x + y;
    *n = d;
}
void g(int x) {
    return ;
}
// remove 1 instruction (icmp) because it's precomputed in if, the icmp for g(!p)
    can be removed and substituted.
void test3(int* p) {
    if(!p) {
        g(!p);
    }
}

int main() {
    int x = 3;
    int y = 4;
    int m = 5;
    test1('a', &x, &y);
    test2(1.0, 2.0, &m, &x);
    test3(&m);
    return x + y + m;
}
```

LISTING 3
Example for CSE

0.7 CONCLUSION

In this MP, we use test-driven approach to develop optimization.

The way to develop tests is trying as many patterns for the corresponding optimization as possible to "hack" the optimization, and carefully examine the statistics computed. We developed 27 tests other than benchmark testing for correctness and successfully evaluate our solution using provided benchmarks.

One can do a simple

```
make all
```

in the project directory to run an end-to-end pipeline to evaluate and test our solution.

During the development, we ran into numerous bugs exposed by our tests and we fixed incrementally. While still leaving optimization opportunities in CSE pass, we managed to carefully document them in this report in details.