COMPILER PROJECT 2

VINDHYA CHHABRA

USC ID : 9773611581

vindhyac@usc.edu

Program Transformations

The single most important criterion that a compiler must meet is correctness—the code that the compiler produces should have the same "meaning" as the program presented for compilation. Each transformation implemented in the compiler must meet this standard.The goal of code-improving transformations is to reduce execution time and run-time memory requirements. Some commonly used llvm transform passes used in this project are :

1) *-constprop (simple constant propagation)*

This pass implements constant propagation and merging. It looks for instructions involving only constant operands and replaces them with a constant value instead of an instruction.

2) *-mem2reg(promote memory to register)*

This file promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses.

3) *-reassociate (Reassociate Expressions)*

This pass reassociates commutative expressions in an order that is designed to promote better constant propagation, GCSE, LICM PRE, etc.

4) *-loop-reduce(Loop strength reduction)*

This pass performs a strength reduction on array references inside loops that have as one or more of their components the loop induction variable. This is accomplished by creating a new value to hold the initial value of the array access for the first iteration, and then creating a new GEP instruction in the loop to increment the value by the appropriate amount.

5) *-indvars(canonicalize induction variables)*

This transformation analyzes and transforms the induction variables (and computations derived from them) into simpler forms suitable for subsequent analysis and transformation.

6) *-licm(loop invariant code motion)*

This pass performs loop invariant code motion, attempting to remove as much code from the body of a loop as possible.

This pass uses alias analysis for two purposes:

1.Moving loop invariant loads and calls out of loops.

2.Scalar Promotion of Memory

7) *-memcpyopt(memcpy optimization)*

This pass performs various transformations related to eliminating memcpy calls, or transforming sets of stores into memsets.

8) *-dse (dead store elimination)*

A trivial dead store elimination that only considers basic-block local redundant stores.

9) *-instcombine(combine redundant instructions)*

Combine instructions to form fewer, simple instructions. This pass does not modify the CFG.

This pass is where algebraic simplification happens.

10) *-die(Dead instruction elimination)*
performs a single pass over the function, removing instructions that are obviously dead.
11) -*dce(Dead code elimination)*
It is similar to dead instruction elimination, but it rechecks instructions that were used by removed instructions to see if they are newly dead

----

Tool : LLVM provides opt tool .The opt command is the modular LLVM optimizer and analyzer. It takes LLVM source files as input, runs the specified optimizations or analyses on it, and then outputs the optimized file or the analysis results.

For analysis, using:

1) -stats
Print statistics.
2)-time-passes
Record the amount of time needed for each pass and print it to standard error.
3)-o <filename>
Specify the output filename.
4)-{passname}
opt provides the ability to run any of LLVM's optimization or analysis passes in any order.The order in which the options occur on the command line are the order in which they are executed (within pass constraints).

In the below codes, I have used **-instcount** for counting the various types of instructions. To display that, I have used -stats.
To find execution time, I have used the flag , **-time-passes** which times each pass and prints elapsed time for each on exit.

For this, we can also use **llvm-bcanalyzer** which gives us the following things :
The llvm-bcanalyzer command is a small utility for analyzing bitcode files. The tool reads a bitcode file (such as generated with the llvm-as tool) and produces a statistical report on the contents of the bitcode file.

The following items are always printed by llvm-bcanalyzer. They comprise the summary output.
Bitcode Analysis Of Module
This just provides the name of the module for which bitcode analysis is being generated.
Bitcode Version Number,File Size,Module Bytes,Function Bytes,Global Types Bytes,Constant Pool Bytes,Module Globals Bytes,Instruction List Bytes,Compaction Table Bytes,Symbol Table Bytes,Dependent Libraries Bytes,Number Of Bitcode Blocks,Number Of Functions,Number Of Types,Number Of Constants,Number Of Basic Blocks,Number Of Instructions,Number Of Long Instructions,Number Of Operands,Number Of Compaction Tables,Number Of Symbol Tables,Number Of Dependent Libs,Total Instruction Size,Average Instruction Size,Maximum Type Slot Number,Maximum Value Slot Number,Bytes Per Value,Bytes Per Global,Bytes Per Function,Bytes Saved With VBR

Code 1 :

In fig 1.
opt all1.bc -o all1opt1.bc -instcount -stats

After applying optimizations as seen in fig 2 and fig3 respectively,
optimization 1 : opt -stats -time-passes -mem2reg -reassociate -licm all1.bc -o all1optnew.bc -instcount
optimization 2: opt -stats -time-passes -mem2reg -reassociate -dce -dse -memcpyopt -licm all1.bc -o all1optnew.bc -instcount

In both optimizations, stats are same and instructions are same but the exeution time is more in case of optimization 2 hence we chose 1 over 2.
Result for the optimization analyzed by llvm-bcanalyzer

```
Record Histogram:
          Count    # Bits   %% Abv  Record Kind
            22      1060           DEBUG_LOC
            12       942           INST_CALL
             9       240 100.00   INST_BINOP
             6       132           DEBUG_LOC_AGAIN
             4       100           INST_BR
             3       138           INST_PHI
             2        20 100.00   INST_RET
             2        44           DECLAREBLOCKS
             1        40           INST_CMP2
```

Code 2:

Before optimization, fig 4 tells instruction count for testproj2.c

After applying optimizations as seen in fig 5, fig6, fig 7 respectively,
optimization 1 : opt -stats -instcombine -reassociate -loop-unroll -mem2reg testproj2.bc -o testproj2opt.bc -instcount
optimization 2: opt -licm -dse -dce -die -constprop -indvars -instcombine -reassociate -loop-unroll -mem2reg -stats testproj2.bc -o testproj2opt.bc
optimization 3: opt -stats -mem2reg -instcombine -reassociate -loop-unroll testproj2.bc -o testproj2opt.bc -instcount

In both optimizations, stats are same and instructions are same but the execution time is more in case of optimization 2 hence we chose 1 over 2.

According to fig 7
 1 instcombine - Number of allocas copied from constant global
19 instcombine - Number of constant folds
12 instcombine - Number of dead inst eliminated
 1 instcombine - Number of dead stores eliminated
 1 instcombine - Number of instructions sunk

```
 8 instcombine - Number of insts combined
 1 instcount   - Number of Add insts
10 instcount   - Number of Br insts
17 instcount   - Number of Call insts
 2 instcount   - Number of PHI insts
 1 instcount   - Number of Ret insts
11 instcount   - Number of basic blocks
31 instcount   - Number of instructions (of all types)
17 instcount   - Number of memory instructions
 1 instcount   - Number of non-external functions
 4 mem2reg     - Number of PHI nodes inserted
14 mem2reg     - Number of alloca's promoted
 6 mem2reg     - Number of alloca's promoted with a single store

For this sequence :
llvm-bcanalyzer gives the result
        Record Histogram:
                  Count    # Bits   %% Abv  Record Kind
                    27      1320        DEBUG_LOC
                    17      1442        INST_CALL
                    10       256        INST_BR
                     2        98        INST_PHI
                     1        16 100.00  INST_RET
                     1        33 100.00  INST_BINOP
                     1        22        DECLAREBLOCKS

According to fig 5
 1 instcombine - Number of allocas copied from constant global
 2 instcombine - Number of constant folds
19 instcombine - Number of dead inst eliminated
39 instcombine - Number of insts combined
 2 instcount   - Number of Add insts
 1 instcount   - Number of Alloca insts
 1 instcount   - Number of BitCast insts
10 instcount   - Number of Br insts
44 instcount   - Number of Call insts
 4 instcount   - Number of GetElementPtr insts
 3 instcount   - Number of ICmp insts
 2 instcount   - Number of Load insts
 3 instcount   - Number of Mul insts
 1 instcount   - Number of Or insts
 4 instcount   - Number of PHI insts
 1 instcount   - Number of Ret insts
 4 instcount   - Number of SExt insts
 1 instcount   - Number of Shl insts
 2 instcount   - Number of Store insts
11 instcount   - Number of basic blocks
83 instcount   - Number of instructions (of all types)
```

53 instcount - Number of memory instructions
1 instcount - Number of non-external functions
3 mem2reg - Number of PHI nodes inserted
9 mem2reg - Number of alloca's promoted
5 mem2reg - Number of alloca's promoted with a single store
4 reassociate - Number of insts reassociated

testproj2op.txt gives the diff of the two .ll files which shoes all the optimizations.

llvm-bcanalyzer gives the result

```
    Record Histogram:
          Count    # Bits   %% Abv  Record Kind
             63      3036           DEBUG_LOC
             44      3602           INST_CALL
             13       286           DEBUG_LOC_AGAIN
             10       256           INST_BR
              7       218 100.00    INST_BINOP
              5       101 100.00    INST_CAST
              4       148           INST_INBOUNDS_GEP
              4       220           INST_PHI
              3       126           INST_CMP2
              2        80           INST_STORE
              2        30 100.00    INST_LOAD
              1        46           INST_ALLOCA
              1        16 100.00    INST_RET
              1        22           DECLAREBLOCKS
```

Hence we observe, according to number of instructions, optimization3 Is better than optimization1 which is further better than optimization1.

In the below sequence

opt -stats -instcombine -loop-unroll -mem2reg -reassociate testproj2.bc -o testproj2opt3.bc -instcount , The stats are as follows, '

```
Record Histogram:
          Count    # Bits   %% Abv  Record Kind
             41      2018           DEBUG_LOC
             22      1636           INST_CALL
             12       264           DEBUG_LOC_AGAIN
             10       256           INST_BR
              6       185 100.00    INST_BINOP
              5       101 100.00    INST_CAST
              4       148           INST_INBOUNDS_GEP
              4       220           INST_PHI
              3       126           INST_CMP2
              2        80           INST_STORE
              2        30 100.00    INST_LOAD
              1        46           INST_ALLOCA
              1        16 100.00    INST_RET
              1        22           DECLAREBLOCKS
```

This is better than optimization1 but worse than optimization3.

In terms of execution time :

There is OProfile event counting tool as well.OProfile provides the ocount tool for collecting raw event counts on a per-application, per-process, per-cpu, or system-wide basis. Unlike the profiling tools, post-processing of the data collected is not necessary -- the data is displayed in the output of ocount.

Code 3(Kernel Code)

Matrix Multiplication

It involves arithmetic calculations, array access and loops.

Bad sequence of transformation is -reassociate -indvars -constprop -memcpyopt -die.
Preferred sequence of transformation is -memcpyopt -indvars -reassociate -constprop -die.

Command used to profile the optimized codes are:

ocount -- event=CPU_CLK_UNHALTED,INST_RETIRED ./<executable>

Here, event type CPU_CLK_UNHALTED gives CPU clock cycles, and INST_RETIRED gives number of instructions.

According to figure : matrixmulunopt figure, CPU Clock Cycles are : 976847 and no of instructions are 511442

According to figure : matrixmulopt1 figure, CPU Clock Cycles are : 953488 and no of instructions are 500007

According to figure : matrixmulopt2 figure, CPU Clock Cycles are : 764136 and no of instructions are 515773

Also,
Speedup with respect to unoptimized code:
Speedup = Old Execution Time /New Execution Time

= (Instruction count(I) x CPI x Clock cycle(C))/(Instruction count(I) x CPI x Clock cycle(C))
(511442 * (976847/511442))*0.001283697)/((515773*(764136/515773))*0.001283697)

Code 4(Kernel Code)

Hashquad.c

It involves arithmetic calculations, array access, dynamic access and loops.
In indvars, This transformation should be followed by all of the desired loop transformations have

been performed.
This gives more no of instructions as compared to when indvars is done later
opt -stats -time-passes -indvars -mem2reg -instcombine -licm hashquad.bc -o hashquadoptnew.bc -instcount // refer fig

So we chose // refer fig 8
opt -stats -time-passes -loop-reduce -mem2reg -instcombine -licm -indvars hashquad.bc -o hashquadoptnew.bc -instcount

```
//==-------------------------------------------------------------------------===
                    ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0287 seconds (0.0673 wall clock)
```

llvm-bcanalyzer result

```
        Record Histogram:
                Count    # Bits   %% Abv  Record Kind
                   98      2156        DEBUG_LOC_AGAIN
                   93      5424        DEBUG_LOC
                   46      4042        INST_CALL
                   33       882        INST_BR
                   24       924        INST_INBOUNDS_GEP
                   24       378 100.00  INST_LOAD
                   20       400 100.00  INST_CAST
                   16       444 100.00  INST_BINOP
                   13       550        INST_CMP2
                   11       536        INST_PHI
                    8        68 100.00  INST_RET
                    8       176        DECLAREBLOCKS
                    5       212        INST_STORE
                    3        12 100.00  INST_UNREACHABLE
```

For fig8b, we can see it also takes less execution time. (Used time-passes for this)// comparison done with figb

```
===-------------------------------------------------------------------------===
                    ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0254 seconds (0.0267 wall clock)
```

So opt 2 is the obvious choice.

Llvm-bcanalyzer esutl

```
        Record Histogram:
                Count    # Bits   %% Abv  Record Kind
                   97      2134        DEBUG_LOC_AGAIN
```

93	5424		DEBUG_LOC
46	4114		INST_CALL
33	882		INST_BR
24	924		INST_INBOUNDS_GEP
24	378	100.00	INST_LOAD
19	**380**	**100.00**	**INST_CAST**
16	438	100.00	INST_BINOP
13	550		INST_CMP2
11	536		INST_PHI
8	68	100.00	INST_RET
8	176		DECLAREBLOCKS
5	212		INST_STORE
3	12	100.00	INST_UNREACHABLE

Code 5

This code has loop, dead code, and scope for strength reduction and loop invariant code motion.

dse_noopt_loop shows unoptimized code while dse_optimized_loop_opt shows code after dse optimization

//Refer fig 9
seq 1
-die -dse -dce -mem2reg -instcombine

seq 2
-instcombine -stats -die -dse -dce -mem2reg

seq1 gives
39 instcount - Number of instructions (of all types)
12 instcount - Number of memory instructions

Record Histogram:

Count	# Bits	%% Abv	Record Kind
22	1060		DEBUG_LOC
17	476	100.00	INST_BINOP
14	308		DEBUG_LOC_AGAIN
12	966		INST_CALL
4	100		INST_BR
3	162		INST_PHI
2	20	100.00	INST_RET
2	44		DECLAREBLOCKS
1	40		INST_CMP2

seq2 gives
58 instcount - Number of instructions (of all types)
30 instcount - Number of memory instructions

Record Histogram:

Count	# Bits	%% Abv	Record Kind
41	1958		DEBUG_LOC
30	2010		INST_CALL
18	509	100.00	INST_BINOP
14	308		DEBUG_LOC_AGAIN
4	100		INST_BR
3	168		INST_PHI
2	20	100.00	INST_RET
2	44		DECLAREBLOCKS
1	40		INST_CMP2

seq1 is better than seq2

** Prefered : constprop to be done after die, indvars after strength reduction (after loop transformations)