Name: Saurabh Bansode
<sbansod@ncsu.edu>
Unity ID: 200539958

# **ECE 566 Project 2:** Common Subexpression Elimination Plus Simple Load/Store Optimization report

## Q1. Describe your implementation and any embellishments you made over my description.

| **Function name** | **Description** |
|---|---|
| **DeadInstRemoval(M)** | In each basic block, code iterates over the instructions.<br><br>Checks for dead instruction, if dead removes from parent, if not, simplifies the instruction using **simplifyInstruction** API.<br><br>Simplified instruction is then substituted for the older instruction using **replaceAllUsesWith** API |
| **local_CSE(M)** | While iterating over all instructions, check if each instruction, lets call it **INST** is any one of the below and sets a boolean flag:<br>1. Load 2. Store 3. Alloca 4. Call 5. Return 6. Branch<br><br>If the flag is set, it does not perform any CSE.<br><br>If the flag is not set, then all the instructions that are after **INST** are checked for type as load, branch, store etc is checked.<br><br>Comparison done using **isIdenticalTo** API.<br><br>If identical, the matched instruction that is later in program order and is inserted in a vector data structure. All uses of the later instruction are replaced with **INST.**<br><br>Once all the instructions within a basicblock are checked, the vector is deleted. For each instruction deleted from the vector with the help of eraseFromParent, CSEElim is incremented once.<br><br>The instruction **INST** and the basic block is then passed to global_CSE, explained in the next step |
| **global_CSE(inst_in, basicblock_in)** | By utilising **getParent**, and **getDescendants** API, I successfully created the Dominator Tree for each function the basic block **basicblock_in** belonged to. |

| | |
|---|---|
| | Then I checked if the instruction received as argument, **inst_in,** was present in any other basic blocks of the function **basicblock_in** was a part of using the isIdenticalTo API.<br>If any hits, I replaced the matched instruction with **inst_in,** using **replaceAllUsesWith** API. I inserted the instruction in a vector. Incremented the **CSEElim** counter for each instruction pushed onto the stack.<br><br>At the end of iteration for each basic block, I deleted the duplicate instructions using **eraseFromParent** API.<br><br>The combination of localCSE and globalCSE made sure that instructions repeating both locally and globally are replaced. |
| **elim_red_load(M)** | Iterated over each instruction of all basic blocks in the module. For each instruction being iterated, let's call it **INST,** I **checked if it was a load.**<br><br>If yes, I iterated over remaining instructions in that basic block. If the instruction being iterated over was a **store or call instruction, I broke out of the for loop** and looked for another instruction altogether.<br><br>If the instruction being iterated over was a load, and it was a **non-volatile** instruction, I obtained the type of the instructions using **getType()** API and compared them.<br><br>If all 3 conditions matched, I checked if the 0th operand of the load instruction, which is the address, are the same, which means it's a redundant load.<br><br>I replaced the later load with an earlier load. And pushed the later load in a vector to be deleted later.<br>Incremented the **CSELdElim** counter every time an instruction was pushed onto the vector. |
| **elim_red_stores(M)** | When iterating over all instructions of the basic block, check if the instruction is a store.<br>If yes type-casted it to a store instruction using **dyn_cast** and named it **s_inst.**<br><br>Used a for loop to iterate over the **remaining instructions** (lets call **r_inst**) in the block to see if any redundant stores/loads are present.<br><br>Checked if<br>1. **r_inst** and **s_inst** are have same address<br>2. **s_inst**'s operand type = type of **r_inst**<br>3. **r_inst** is non-volatile |

| | |
|---|---|
| | 4. **r_inst** is a load instruction<br>If all conditions match, increment **CSEStore2Load,** replace all uses of r_inst with s_inst using **replaceAllUsesWith** API.<br><br>Second condition was checked for any store elimination |

**Q2. Collect data per benchmark that compares the number of instructions, the total number of loads, the total number of stores, the counters you collect in the CSE pass**

**Benchmark data focusing on counters, with just CSE enabled and no mem2reg**

| Bench-mark Name | Instructions | Loads | Stores | CSEDead | CSEElim | CSE Simplify | CSELdElim | CSEStore2Load | CSEStElim |
|---|---|---|---|---|---|---|---|---|---|
| adpcm | 418 | 122 | 81 | 1 | 0 | 4 | 0 | 0 | 0 |
| arm | 744 | 123 | 116 | 0 | 14 | 40 | 24 | 0 | 0 |
| basicmath | 566 | 156 | 100 | 2 | 10 | 19 | 11 | 0 | 0 |
| bh | 3104 | 825 | 494 | 51 | 26 | 83 | 67 | 0 | 0 |
| bitcount | 637 | 158 | 98 | 1 | 0 | 7 | 17 | 0 | 0 |
| crc32 | 32141 | 33 | 29 | 0 | 0 | 0 | 3 | 1 | 0 |
| dijkstra | 319 | 94 | 51 | 0 | 0 | 8 | 1 | 0 | 0 |
| em3d | 1221 | 418 | 192 | 1 | 0 | 35 | 1 | 0 | 0 |
| fft | 699 | 208 | 102 | 1 | 16 | 8 | 8 | 0 | 0 |
| hanoi | 96 | 29 | 16 | 0 | 0 | 1 | 0 | 0 | 0 |
| hello | 4 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| kmp | 529 | 146 | 71 | 0 | 0 | 16 | 27 | 0 | 0 |
| l2lat | 89 | 24 | 15 | 4 | 0 | 1 | 1 | 0 | 0 |
| patricia | 1043 | 370 | 108 | 1 | 4 | 46 | 8 | 0 | 0 |
| qsort | 142 | 36 | 16 | 1 | 0 | 3 | 2 | 0 | 0 |
| sha | 624 | 186 | 99 | 0 | 5 | 36 | 24 | 1 | 0 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| smatrix | 291 | 75 | 31 | 2 | 0 | 1 | 22 | 0 | 0 |
| sql | 171601 | 57082 | 21897 | 300 | 414 | 3960 | 1834 | 0 | 0 |
| susan | 12181 | 4209 | 1438 | 5 | 68 | 22 | 364 | 0 | 0 |

**Timing without mem2reg**

| | |
|---|---|
| adpcm | 2.32 |
| arm | 0 |
| basicmath | 0.08 |
| bh | 1.11 |
| bitcount | 0.23 |
| crc | 0 |
| dijkstra | 0.05 |
| em3d | 0.4 |
| fft | 0.07 |
| hanoi | 5.53 |
| hello | 0 |
| kmp | 0.21 |
| l2lat | 0.03 |
| patricia | 0.11 |
| qsort | 0.04 |
| sha | 0 |
| smatrix | 7.35 |
| sql | 0 |
| susan | 0.7 |

**Benchmark data focusing on counters, with just CSE enabled AND mem2reg**

| Bench-mark Name | Instructions | Loads | Stores | CSEDead | CSEElim | CSE Simplify | CSELdElim | CSEStore2Load | CSEStElim |
|---|---|---|---|---|---|---|---|---|---|
| adpcm | 240 | 15 | 7 | 2 | 7 | 6 | 0 | 0 | 0 |
| arm | 394 | 47 | 18 | 0 | 33 | 42 | 2 | 0 | 0 |
| basicmath | 326 | 24 | 12 | 1 | 20 | 19 | 1 | 0 | 0 |
| bh | 1832 | 192 | 142 | 1 | 168 | 84 | 5 | 0 | 0 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| bitcount | 426 | 51 | 18 | 1 | 7 | 7 | 0 | 0 | 0 |
| crc32 | 83 | 8 | 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| dijkstra | 225 | 48 | 24 | 0 | 7 | 8 | 1 | 0 | 0 |
| em3d | 634 | 117 | 43 | 3 | 50 | 36 | 0 | 0 | 0 |
| fft | 385 | 38 | 24 | 0 | 52 | 9 | 0 | 0 | 0 |
| hanoi | 52 | 6 | 4 | 0 | 1 | 1 | 0 | 0 | 0 |
| hello | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| kmp | 331 | 47 | 20 | 0 | 38 | 16 | 11 | 0 | 0 |
| l2lat | 57 | 8 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| patricia | 638 | 133 | 30 | 0 | 76 | 48 | 4 | 0 | 0 |
| qsort | 89 | 13 | 4 | 1 | 10 | 3 | 0 | 0 | 0 |
| sha | 371 | 41 | 28 | 0 | 46 | 36 | 1 | 0 | 0 |
| smatrix | 210 | 34 | 10 | 0 | 20 | 1 | 5 | 0 | 0 |
| sql | 103159 | 16340 | 5843 | 188 | 7076 | 4087 | 239 | 5 | 0 |
| susan | 6544 | 1021 | 157 | 0 | 1210 | 34 | 34 | 0 | 0 |

**Timing with mem2reg**

| | |
|---|---|
| adpcm | 2.89 |
| arm | 0 |
| basicmath | 0.09 |
| bh | 1.31 |
| bitcount | 0.28 |
| crc | 0.39 |
| dijkstra | 0.09 |
| em3d | 0.45 |
| fft | 0.07 |
| hanoi | 4.04 |
| hello | 0 |
| kmp | 0.23 |
| l2lat | 0.05 |

| | |
|---|---|
| patricia | 0.14 |
| qsort | 0.05 |
| sha | 0.06 |
| smatrix | 7.07 |
| sql | 0 |
| susan | 1.59 |

## Q3. Explain the difference in results for these two configurations using your data for Q2

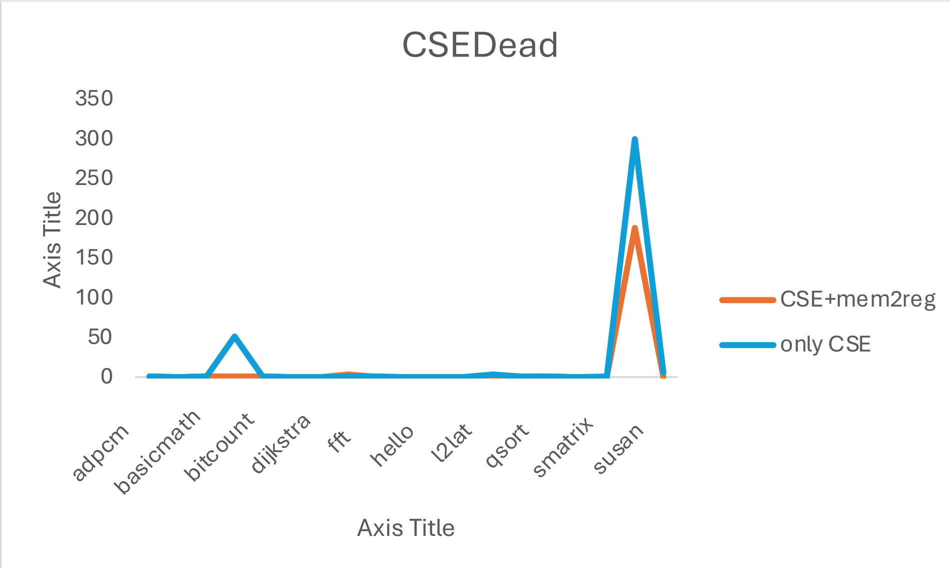
Difference in results for the configs used in Q2 is below:

| **Just CSE, NO mem2reg** | **CSE and mem2reg** |
|---|---|
| More Instructions, more stores, more loads | Compared to when we do not apply memory to register promotion, we get very few instructions, very few stores and loads. When we apply mem2reg optimization |
| Looking at susan and emd3 benchmark, I conclude that without mem2reg optimization, there are more dead instruction removal | With memory to register optimization applied, there are more CSE eliminations for susan, fft and bh benchmarks |
| There is very little information on store elimination. | Store elimination is practically not present in this data |
| | |

## Q4. Compare the output counters you collected.

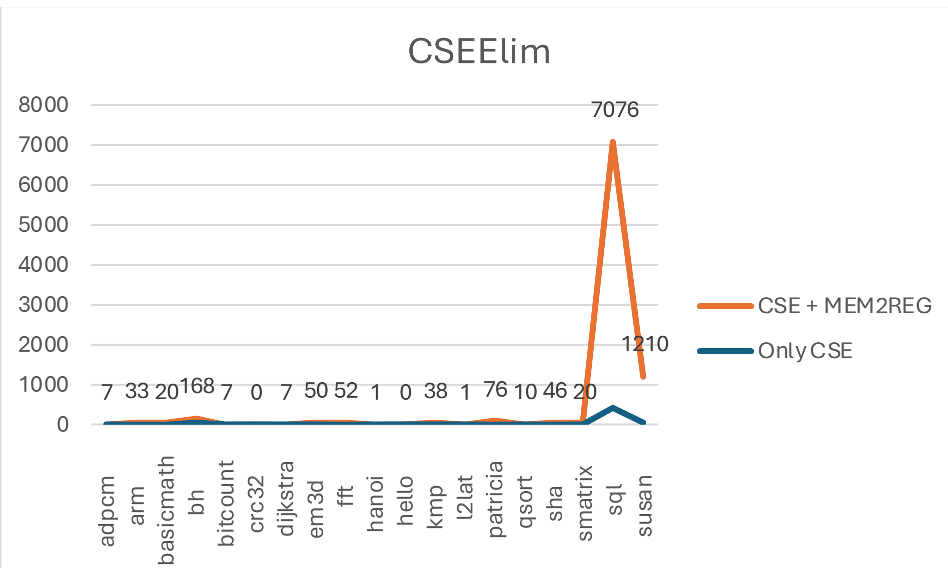
**CSEDead**: Apart from susan and bitscount, there is not a significant reduction in dead instructions.

When Mem2reg is applied along with CSE, we can see lot less instructions being removed.

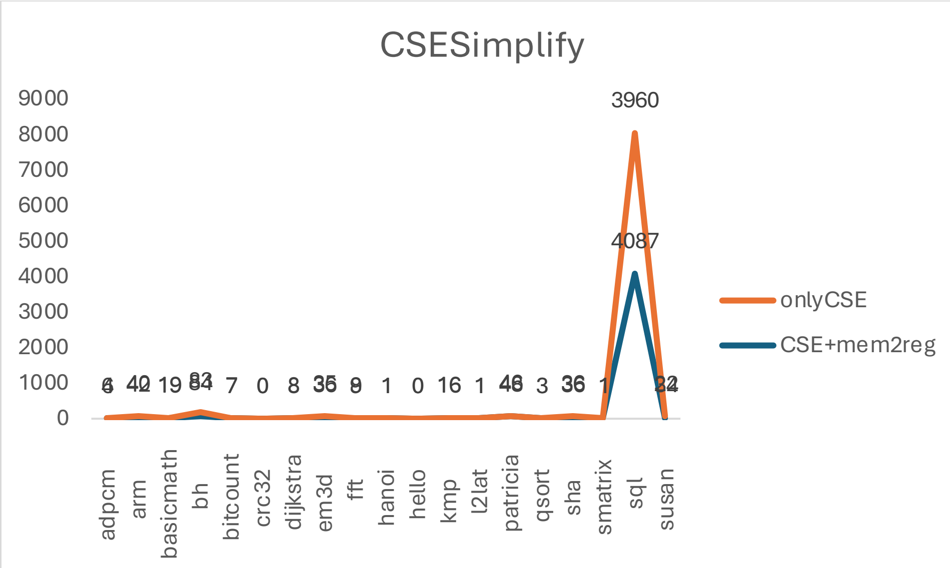



**CSEElim**: For sql benchmark, when mem2reg is applied, there is humongous amount of instructions

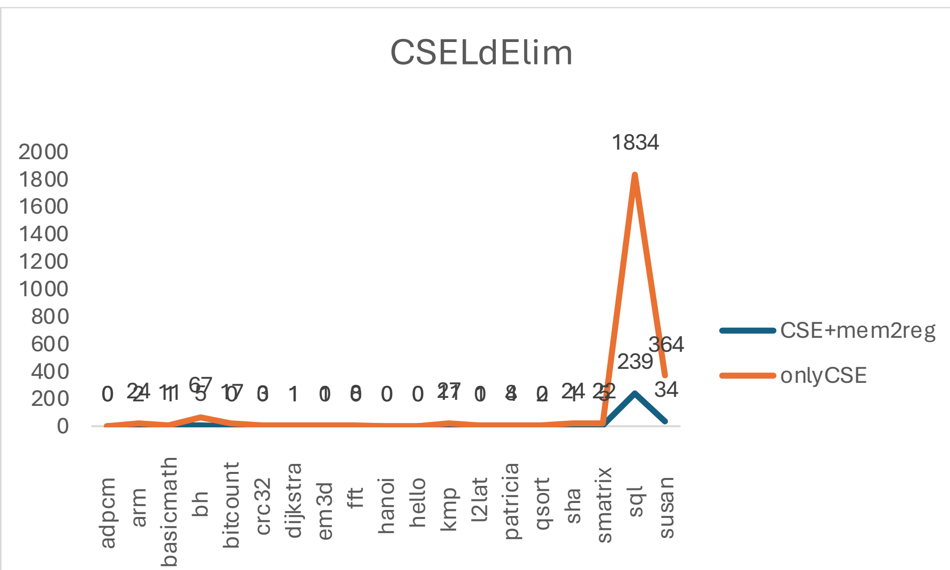
Being eliminated as means for CSEElim. Other benchmarks have minimal differences.




**CSESimplify**: For sql benchmark, there is significant increase in instructions being simplified when both CSE and memory 2 register optimizations are applied.

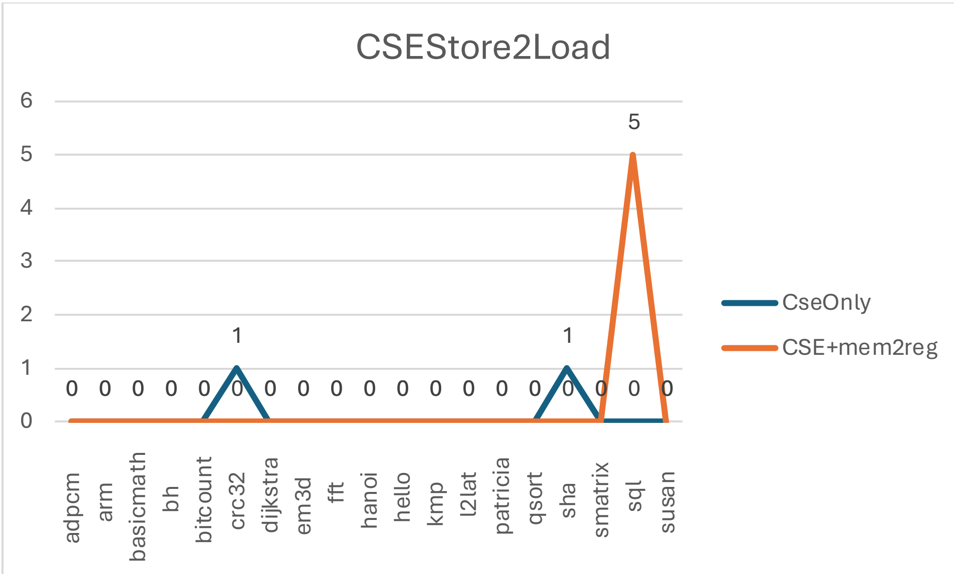



For **CSELdElim**, bh and sql benchmarks display that applying memory 2 register optimization along with CSE improves in reducing the number of redundant load instructions.




**CSEStore2Load –** For CSE alone, only crc32 and sha has any instructions removed.

Whereas when CSE+mem2reg is applied, both crc32 and sha show no instructions being removed, however sql shows huge improvement.




**CSEStElim:** For CSE Store Eliminations, neither CSE alone nor CSE+mem2reg shows any instructions being removed. This means that my implementation for CSE store elimination is flawed.