Project 2 Report

Phan Nguyen

ECE-466

I. Implementation

- Loop over all the basic block and instruction using the same code that was provided in tutorial 3.

1. Make the CSE-dead

- However, when the project asked to remove the Dead instruction on the way, not insert it into the Worklist and remove it later. So, I have to come up with new way to loop over the instructions:

```
auto i = bb->begin();
   while( i != bb->end())
   if (isDead(*i)) {
              Instruction *j = &*i;
              CSEDead++;
              i = j->eraseFromParent();
            }
```

- I find out that if we i->eraseFromParent() then we will get segmentation fault since we will use that pointer later. So I make a new pointer and let it point to the current iterator, then let the current iterator = j->eraseFromParent().
- This work because the earseFromParent() will return the pointer point to the instruction after the one we just delete.

2. CSE-Simplify

- First, we have to check if the instruction can be simplify by using the return of the function call:

```
SimplifyInstruction(&*i,M->getDataLayout())
```

- If this returns something that not a nullptr, then we know that we can do the CSE-simplify here.
- In order to replace all use of this instruction with a new simplify value, we use:

```
Value *val = SimplifyInstruction(&*i,M->getDataLayout());
 i->replaceAllUsesWith(val);
```

- This will ensure that all the use of i will be replace by val.
- After that we will also remove I from the function by using the same:

```
i = j->eraseFromParent();
```

3. CSE- Elim

- For CSE-Elim, I have to pick the current instruction then go over all its dominance child to check if there is any instruction that is exactly the same. I also have to check and test with instructions we can suppress, because some will give segment fault and some will cause program to timed out. We do this by calling the canCSE() function.

- The canCSE() function will check if the opcode is good to suppress, it body is the same as the isDead() function. It will return true if opcode is ok and fault otherwise.

  ```
  auto j = i;
  j++;
  ```
- First, we have another pointer point to current instruction, then move it to the next instruction. Then we know that all instruction in the same basic block, that is after the current one will be dominated by it, so we use a while loop to stop when it the end of that basic block:

  ```
  while(j != bb->end())
  ```
- We need to check those dominance children to see if they are: same type, opcode, numb of operands. If so then we know that we can use CSE-Elim here:

  ```
  auto *val = dyn_cast<Value>(&*i);
  j->replaceAllUsesWith(val);
  ```
- This way we will replace the use of j by value of i. And we will just erase j from the parent like above.

II. Data benchmark

Table 1: Instruction with and without CSE pass.

<table>
<tr><th></th><th>#Instructions</th><th>#Loads</th><th>#Store</th><th>#CSE-dead</th><th>#CSE-Simplify</th><th>#CSE-Elim</th></tr>
<tr><td>Cse0</td><td>1</td><td>0</td><td>0</td><td>6</td><td>0</td><td>0</td></tr>
<tr><td>Cse1</td><td>12</td><td>2</td><td>4</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>Cse2</td><td>1</td><td>0</td><td>0</td><td>0</td><td>3</td><td>0</td></tr>
<tr><td>Cse3</td><td>5</td><td>2</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>Cse4</td><td>6</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>Cse5</td><td>4</td><td>2</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>Cse6</td><td>29</td><td>5</td><td>8</td><td>1</td><td>2</td><td>2</td></tr>
<tr><td>No-Cse0</td><td>7</td><td>0</td><td>0</td><td rowspan="7" colspan="3">0</td></tr>
<tr><td>No-Cse1</td><td>13</td><td>2</td><td>4</td></tr>
<tr><td>No-Cse2</td><td>4</td><td>0</td><td>0</td></tr>
<tr><td>No-Cse3</td><td>6</td><td>2</td><td>0</td></tr>
<tr><td>No-Cse4</td><td>7</td><td>1</td><td>1</td></tr>
<tr><td>No-Cse5</td><td>5</td><td>0</td><td>2</td></tr>
<tr><td>No-Cse6</td><td>34</td><td>6</td><td>8</td></tr>
</table>

Table 2: Instruction counts.

Instruction counts	**CSE**	**M2RCSE**	**NOCSE**
Adpcm	406	232	419
Arm	708	370	784
Basicmath	507	287	591
Bh	3113	1983	3301
Bitcount	541	339	665
Crc32	136	74	145
Dijkstra	304	223	322
Em3d	1139	627	1233
Fft	639	375	739

Hanoi	87	47	96
Hello	4	2	4
kmp	484	316	559
L2lat	88	57	97
Patricia	1059	718	1079
Qsort	132	92	148
Sha	575	362	661
Smatrix	230	180	315
sql	171143	108544	176711
susan	7838	4153	12630

Table 3: Timing counts.

Timing	**CSE**	**M2RCSE**	**NOCSE**
Adpcm	0.0	0.0	0.83
Arm	0.0	0.0	0.0
Basicmath	0.02	0.02	0.02
Bh	0.0	0.0	0.51
Bitcount	0.05	0.05	0.09
Crc32	0.02	0.03	0.05
Dijkstra	0.0	0.0	0.02
Em3d	0.01	0.0	0.19
Fft	0.01	0.01	0.02
Hanoi	0.0	0.0	1.15
kmp	0.02	0.02	0.08
L2lat	0.01	0.01	0.0
Patricia	0.01	0.02	0.02
Qsort	0.02	0.02	0.01
Sha	0.01	0.01	0.01
Smatrix	1.73	1.69	1.8
sql	0.0	0.0	0.0
susan	0.0	0.0	0.31

III. Explanation

-We can see that there is a huge difference between running my own CSE with and without Mem2reg first. The difference can be seen when we compare the benchmark in table 2. It is guaranteed that the instruction counts after we run mem2reg is always less than when we run without it.

- Because as the nature of mem2reg passes, it reduces the instruction for us. This not only helps to reduce the total instruction, but it also creates new opportunities for my CSE pass to run.
- Remember that in the CSE pass I implemented, two instructions need to have the same operant and both of it in exactly order to qualify for the pass. The mem2reg pass increase the change to do that, for example:

```
%1 = alloca i32, align 4
store i32 %x, i32 %i, align 4

%3 = add nsw i32 %i, 3
%4 = add nsw i32 %x, 3
```

-As we can see, without mem2reg, we won't know that %x and %i is just the same, and instruction of %3 and %4 will not have the same operand, thus we will be missing the opportunity.

IV. Output counter comparing

Table 4: test_466stats from gradescope

	CSE-Elim	CSE-Dead	CSE-Simplify
crc32	3	6	0
basicmath	46	32	6
bitcount	73	50	1
em3d	43	38	13
12lat	3	6	0
qsort	8	8	0
sql	2389	2295	884
fft	54	46	0
adpcm	6	7	0
bh	68	120	0
hanoi	4	4	0
sha	42	42	2
susan	2736	2039	17
arm	28	29	19
dijkstra	7	8	0
kmp	36	30	9
patricia	8	8	4
smatrix	39	46	0

-From the table above, we can compare the counter of the three passes. We see a trend where the CSE-simplify doesn't run much except in sql benchmark. This can be due to the nature of our cse pass, because the CSE-simplify is a constant folding pass. And the code needs to be in form: t= L op R, where both L and R, and the result is constant, but we won't meet a lot of that case.

-In other side, the counter of CSE-Elim and CSE-dead are very identical to each other and run a lot compared to the CSE-simplify. This can be because both of those pass use the same structure, such as checking the instruction type to see if we can optimize it, and the nature of the program language.