North Carolina State University

ECE 566 - Compiler Optimization and Scheduling
Prof. James Tuck

Project 3:
Common Subexpression Elimination Plus Simple Load/Store Optimization

Rhushikesh Prabhune
Unity ID: rprabhu

1. Modifications made to the CSE algorithm:

The given CSE algorithm was further modified by running dead code elimination, constant folding and basic common subexpression elimination again after one complete round of the original algorithm.

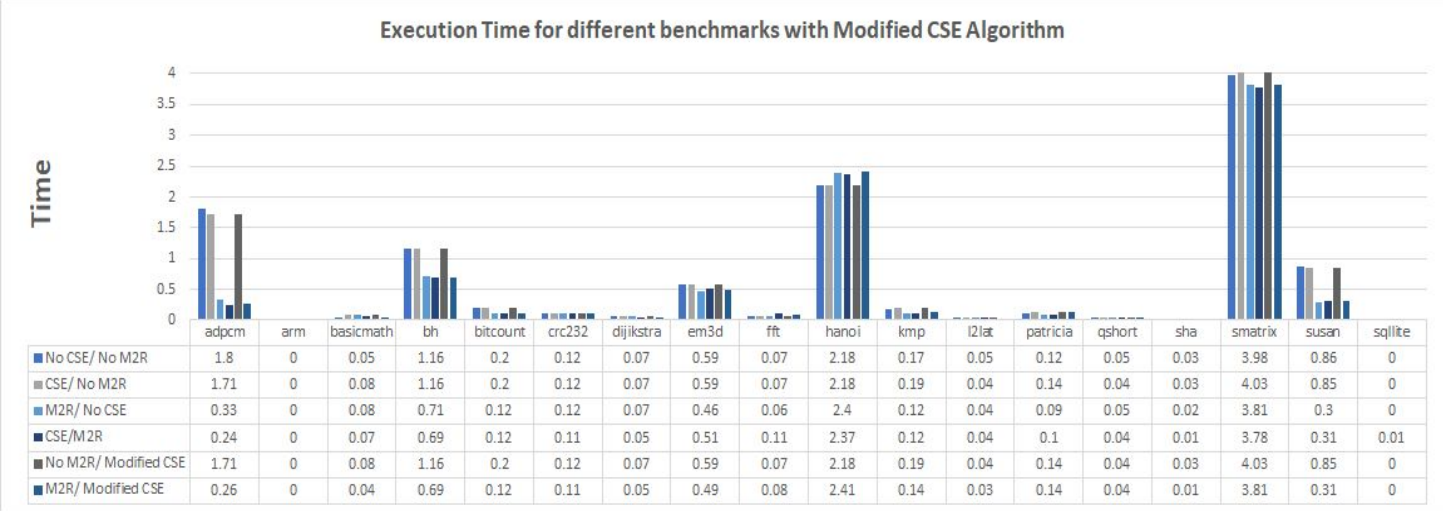
This ensured that dead instructions were removed and the remaining instructions were checked for any possibility of simplification through constant folding. Running the basic CSE pass ensured that the resulting instructions from the earlier passes were checked for literal matches and eliminated if possible.

```
make[2]: Leaving directory '/ncstate_ece566_spring2020/simple/p3/C++/p3-test_final/Benchmarks/susan'
make[2]: Entering directory '/ncstate_ece566_spring2020/simple/p3/C++/p3-test_final/Benchmarks/sqlite'
CSE_Dead.....300
CSE_Basic.....6522
CSE_Simplify..630
CSE_RLd.......1825
CSE_RSt.......19
CSE_LdSt......2161
Total Instructions Deleted: 11457
CSE_Dead(after modification).....326
CSE_Basic(after modification).....6960
CSE_Simplify(after modification)..650
CSE_RLd(after modification).......1825
CSE_RSt(after modification).......19
CSE_LdSt(after modification)......2161
Total Instructions Deleted(after modification): 11941
[built sql]
```

The improvement is seen in *sqlite* benchmark as the number of dead instructions in the above dataset. The original CSE algorithm eliminates 11,457 instructions while the modified CSE eliminates 11,941 instructions which accounts to a 4.25% increase in the number of eliminated instructions. The basic CSE algorithm results in the most increase in instruction elimination in the second iteration because redundant load elimination can create chances for the CSE algorithm for instructions which were earlier not literal matches. The number of dead instructions also increases in the second iteration after some of the instructions are deleted by the previous passes. The load-store elimination pass can open up more chances for Instruction simplification which can be seen to increase after the pass is run in the second iteration.

```
make[2]: Entering directory '/ncstate_ece566_spring2020/simple/p3/C++/p3-test_final/Benchmarks/susan'
CSE_Dead.....0
CSE_Basic.....213
CSE_Simplify..2
CSE_RLd.......364
CSE_RSt.......9
CSE_LdSt......217
Total Instructions Deleted: 805
CSE_Dead(after modification).....0
CSE_Basic(after modification).....385
CSE_Simplify(after modification)..2
CSE_RLd(after modification).......364
CSE_RSt(after modification).......9
CSE_LdSt(after modification)......217
Total Instructions Deleted(after modification): 977
[built susan]
make[2]: Leaving directory '/ncstate_ece566_spring2020/simple/p3/C++/p3-test_final/Benchmarks/susan'
```

The *susan* benchmark also shows an increase of 21.4% in the number of eliminated instructions for the modified CSE algorithm. A similar trend is observed as seen in the *sqlite* benchmark where the basic CSE algorithm gives a steep increase in the eliminated instructions.

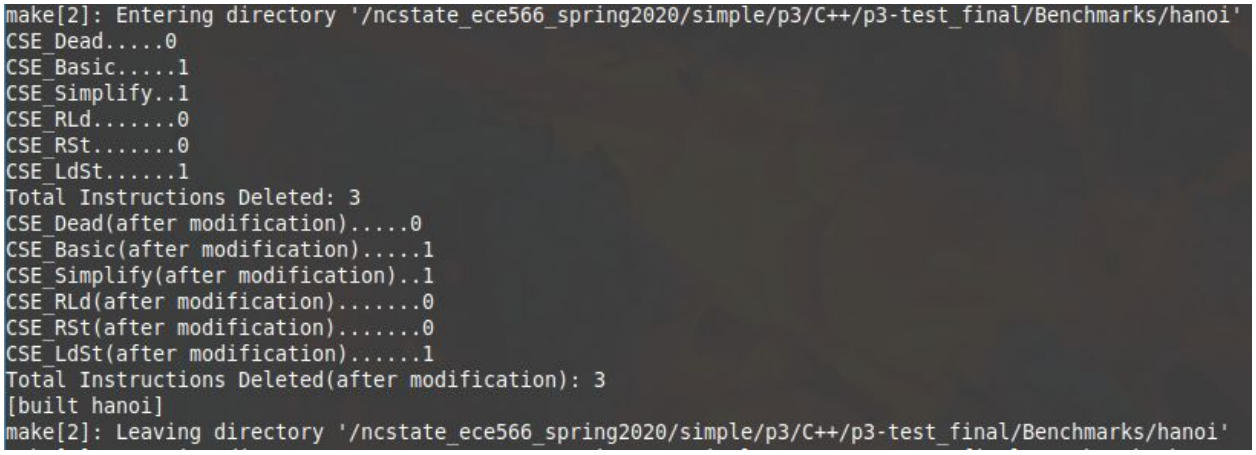


	adpcm	arm	basicmath	bh	bitcount	crc232	dijikstra	em3d	fft	hanoi	kmp	l2lat	patricia	qshort	sha	smatrix	susan	sqllite
No CSE/ No M2R	1.8	0	0.05	1.16	0.2	0.12	0.07	0.59	0.07	2.18	0.17	0.05	0.12	0.05	0.03	3.98	0.86	0
CSE/ No M2R	1.71	0	0.08	1.16	0.2	0.12	0.07	0.59	0.07	2.18	0.19	0.04	0.14	0.04	0.03	4.03	0.85	0
M2R/ No CSE	0.33	0	0.08	0.71	0.12	0.12	0.07	0.46	0.06	2.4	0.12	0.04	0.09	0.05	0.02	3.81	0.3	0
CSE/M2R	0.24	0	0.07	0.69	0.12	0.11	0.05	0.51	0.11	2.37	0.12	0.04	0.1	0.04	0.01	3.78	0.31	0.01
No M2R/ Modified CSE	1.71	0	0.08	1.16	0.2	0.12	0.07	0.59	0.07	2.18	0.19	0.04	0.14	0.04	0.03	4.03	0.85	0
M2R/ Modified CSE	0.26	0	0.04	0.69	0.12	0.11	0.05	0.49	0.08	2.41	0.14	0.03	0.14	0.04	0.01	3.81	0.31	0

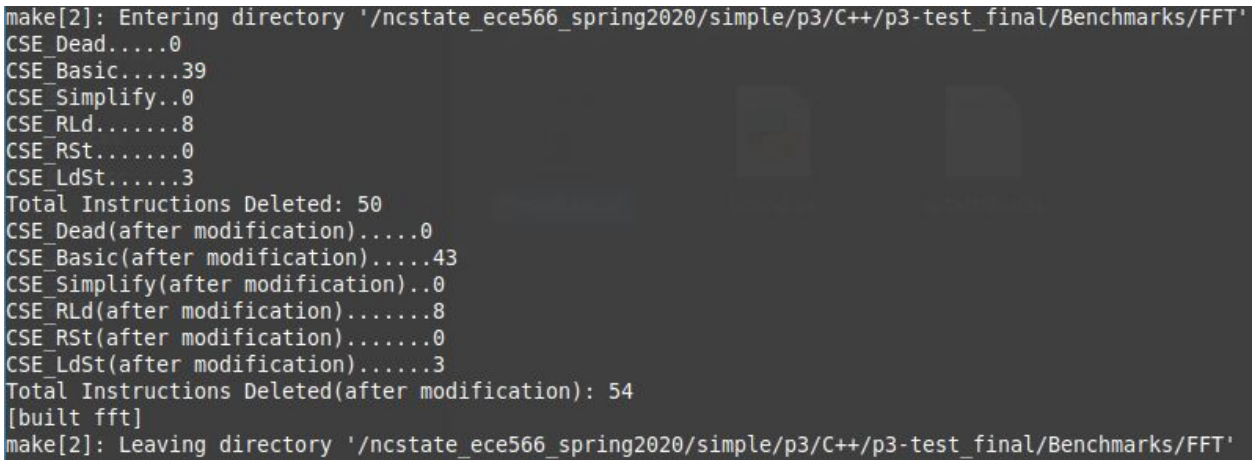
It can be seen that the *basicmath, bitcount, crc232, dijkstra, em3d, l2lat, sha, susan, sqlite* benchmarks have a decrease in execution time with modified CSE as compared to the original CSE algorithm given by the professor.

The hanoi benchmark does not show any improvement in the execution time after further optimization. Rather the execution time is seen to increase

which can be accounted by the differences in execution time for each run of the same program. The figure below shows the details regarding the *hanoi* benchmark.

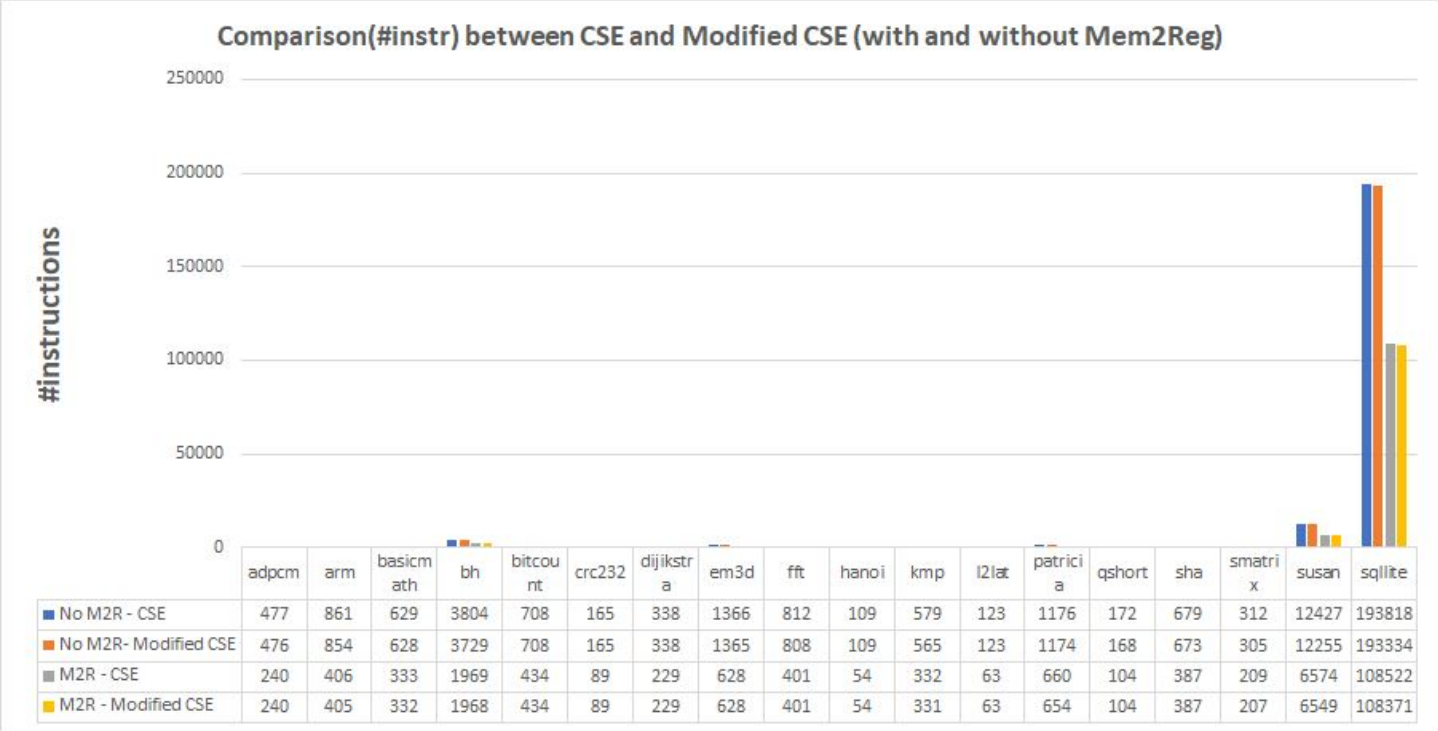


The *FFT* benchmark only shows a decrease of 4 instructions with the modified CSE pass and therefore does not show any significant improvement in execution time. A similar trend is observed with *patricia* and *qshort* benchmarks.



The modified CSE pass can offer a considerable improvement in execution time if the initial number of instructions is a very large number resulting in more chances of elimination even after the first round of elimination. Therefore, although we see a significant decrease in the number of instructions executed (as shown below), the execution time is not seen to decrease to a large extent. Moreover, the execution time also depends on the

type of instructions along with the number of instructions.



	adpcm	arm	basicmath	bh	bitcount	crc232	dijikstra	em3d	fft	hanoi	kmp	l2lat	patricia	qshort	sha	smatrix	susan	sqllite
No M2R - CSE	477	861	629	3804	708	165	338	1366	812	109	579	123	1176	172	679	312	12427	193818
No M2R- Modified CSE	476	854	628	3729	708	165	338	1365	808	109	565	123	1174	168	673	305	12255	193334
M2R - CSE	240	406	333	1969	434	89	229	628	401	54	332	63	660	104	387	209	6574	108522
M2R - Modified CSE	240	405	332	1968	434	89	229	628	401	54	331	63	654	104	387	207	6549	108371

After running the original and modified CSE passes respectively, it is seen that the number of remaining instructions is the least with memory-register promotion and modified CSE pass. The *sqlite* benchmark shows a decrease of 484 instructions without memory-register promotion and 151 instructions with memory-register promotion. As explained above, more the number of instructions, more the chances for the modified CSE algorithm to eliminate instructions which is why *sqlite* shows the highest improvement in terms of instruction elimination. The other benchmarks also show an improvement although not as much as *sqlite.*

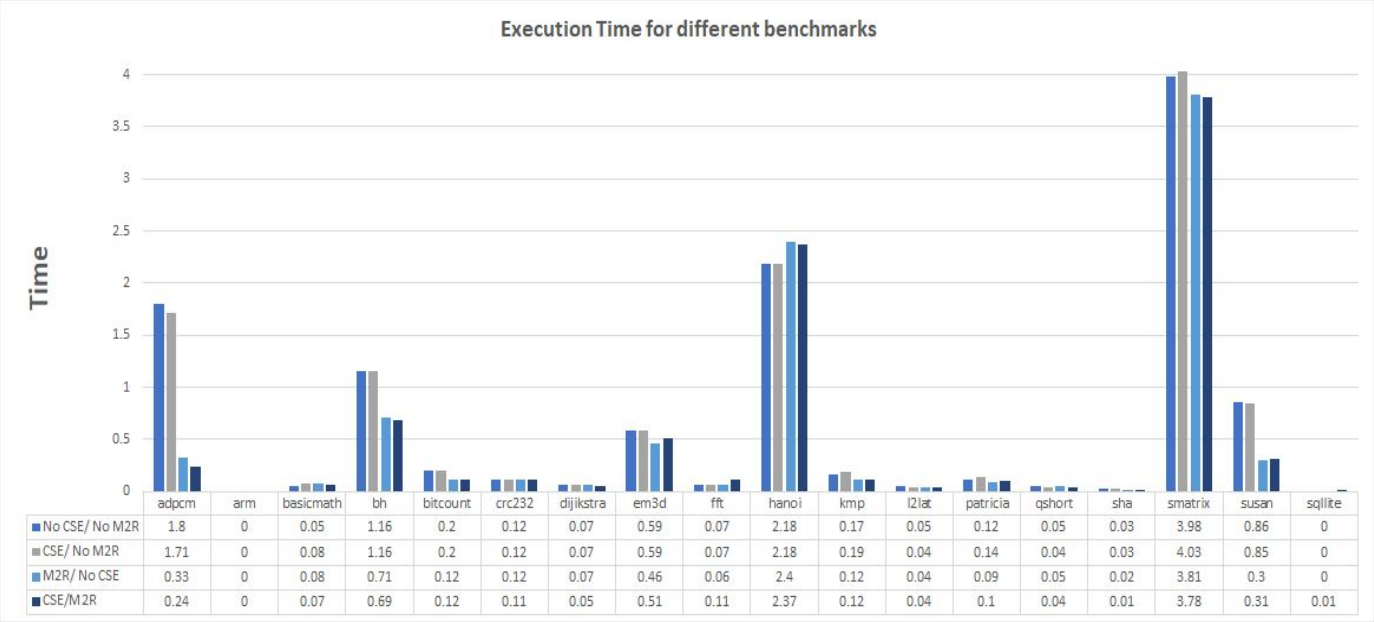
```
////////////////////////////////////////////////////////////////////////////////
void LLVMCommonSubexpressionElimination_Cpp(Module *M)
{
  DCE_pass(*M);
  ConstantFolding(*M);
  CSE_Basic_Pass(*M);
  LoadElim_Pass(*M);
  LdSt_Elim(*M);

//   print out summary of results
  fprintf(stderr,"CSE_Dead.....%d\n",  CSEDead);
  fprintf(stderr,"CSE_Basic.....%d\n", CSEElim);
  fprintf(stderr,"CSE_Simplify..%d\n", CSESimplify);
  fprintf(stderr,"CSE_RLd.......%d\n", CSELdElim);
  fprintf(stderr,"CSE_RSt.......%d\n", CSERStElim);
  fprintf(stderr,"CSE_LdSt......%d\n", CSELdStElim);
  fprintf(stderr,"Total Instructions Deleted: %d\n",total_deleted_instr);
  DCE_pass(*M);
  ConstantFolding(*M);
  CSE_Basic_Pass(*M);

  // fprintf(stderr,"CSE_Dead(after modification).....%d\n",  CSEDead);
  // fprintf(stderr,"CSE_Basic(after modification).....%d\n", CSEElim);
  // fprintf(stderr,"CSE_Simplify(after modification)..%d\n", CSESimplify);
  // fprintf(stderr,"CSE_RLd(after modification).......%d\n", CSELdElim);
  // fprintf(stderr,"CSE_RSt(after modification).......%d\n", CSERStElim);
  // fprintf(stderr,"CSE_LdSt(after modification)......%d\n", CSELdStElim);
  // fprintf(stderr,"Total Instructions Deleted(after modification): %d\n",total_deleted_instr);
}
```

The code snippet above shows the modifications. The commented prints can be used to obtain the modified counter values.

2. Comparison of Execution time for different benchmarks(with and without mem2reg promotion):



	adpcm	arm	basicmath	bh	bitcount	crc232	dijikstra	em3d	fft	hanoi	kmp	l2lat	patricia	qshort	sha	smatrix	susan	sqllite
No CSE/ No M2R	1.8	0	0.05	1.16	0.2	0.12	0.07	0.59	0.07	2.18	0.17	0.05	0.12	0.05	0.03	3.98	0.86	0
CSE/ No M2R	1.71	0	0.08	1.16	0.2	0.12	0.07	0.59	0.07	2.18	0.19	0.04	0.14	0.04	0.03	4.03	0.85	0
M2R/ No CSE	0.33	0	0.08	0.71	0.12	0.12	0.07	0.46	0.06	2.4	0.12	0.04	0.09	0.05	0.02	3.81	0.3	0
CSE/M2R	0.24	0	0.07	0.69	0.12	0.11	0.05	0.51	0.11	2.37	0.12	0.04	0.1	0.04	0.01	3.78	0.31	0.01

The table and graph above denotes the execution time with four different configurations of CSE pass and memory-register promotion. There is a steep decrease in the execution time when mem2reg promotion is applied as compared to when it is not applied. This is expected since the promotion eliminates memory instructions from the instruction stream and memory instructions can take many cycles to execute depending on cache hits and misses. Promoting the memory value to a register will help run the instruction way more faster. Therefore, when many such promotions are carried out, there is a significant decrease in the execution time as seen in the graph above.

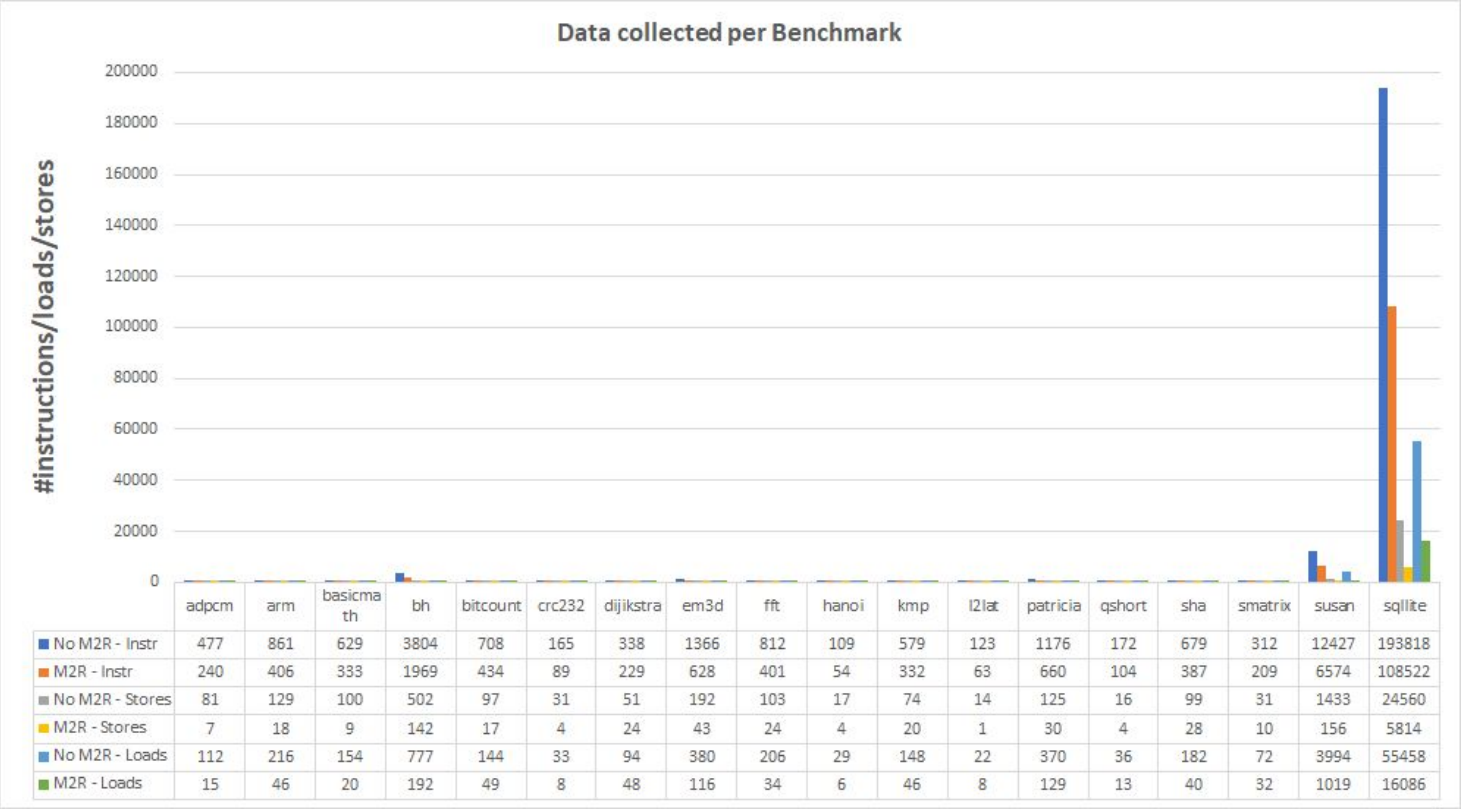
However, the difference in execution times with and without CSE pass is not as significant as the mem2reg promotion. This is mainly because CSE does not affect memory instructions as much as mem2reg promotion does. It does eliminate loads and stores but not as much as mem2reg promotion. The CSE pass only eliminates instructions that are literal matches which includes load instructions if they have the same address operand, and store instructions if they have the same value and

address operand. Since CSE can only eliminate literal matches, there may be many more instructions that are not literal matches but do the same thing when executed.

Therefore, it can be seen from the graph that there is a small decrease in the execution time with and without the CSE pass. Some of the benchmarks show an increase in execution time with the CSE pass. This small difference can be ignored since the execution timing values will be different for the same program at different runs. The *adpcm* and *susan* benchmarks show a high decrease in execution time after the two passes denoting a high number of memory instructions in their instruction stream. As for the benchmarks which do not show any significant decrease, we can conclude that they do not have as many memory operations to be deleted.

The *arm* and *bitcount* benchmarks fail even for the default unoptimized code because of floating point issues. The floating point values may change depending on the machine which is used for execution. For the *l2lat* benchmark there is nothing compared (program output).

3. Comparison of the number of instructions, the total number of loads, and the total number of stores:



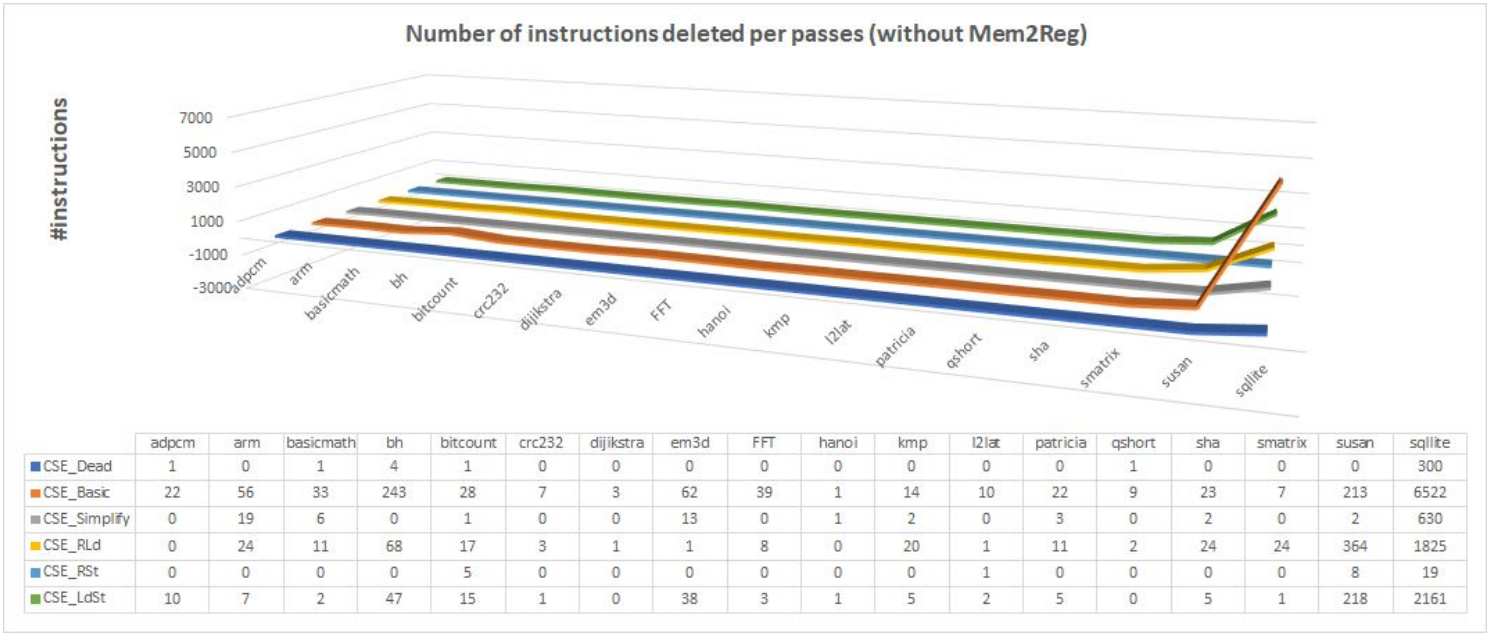
	adpcm	arm	basicmath	bh	bitcount	crc232	dijikstra	em3d	fft	hanoi	kmp	l2lat	patricia	qshort	sha	smatrix	susan	sqllite
No M2R - Instr	477	861	629	3804	708	165	338	1366	812	109	579	123	1176	172	679	312	12427	193818
M2R - Instr	240	406	333	1969	434	89	229	628	401	54	332	63	660	104	387	209	6574	108522
No M2R - Stores	81	129	100	502	97	31	51	192	103	17	74	14	125	16	99	31	1433	24560
M2R - Stores	7	18	9	142	17	4	24	43	24	4	20	1	30	4	28	10	156	5814
No M2R - Loads	112	216	154	777	144	33	94	380	206	29	148	22	370	36	182	72	3994	55458
M2R - Loads	15	46	20	192	49	8	48	116	34	6	46	8	129	13	40	32	1019	16086

The table and graph above denotes the number of total instructions in the instruction stream before and after the mem2reg pass(with CSE pass run after the mem2reg pass). It also shows the number of load and store instructions before and after applying the mem2reg pass. The mem2reg promotion eliminates loads and stores by promoting the values to registers whenever such a transition is possible. This is why we see a significant decrease in the number of load and store instructions (as seen in the table) after the mem2reg promotion is applied. Most of the instructions which have been eliminated from the total number of instructions belong to the loads and stores eliminated in the promotion pass.

More the number of instructions, more is the opportunity to eliminate instructions in most cases. We see that the *sqlite* benchmark shows the

highest decrease in the number of stores from 24,560 to 5814 after the promotion pass. The number of loads also decreased from 55,458 to 16,068. The *susan* benchmark also shows a similar trend. Even though some benchmarks have a lesser number of instructions, the promotion pass still does eliminate a considerable amount of instructions which can be due to a large amount of memory operations in the instruction stream which is a common case in most applications.

4. Comparison of output counters collected from the passes:

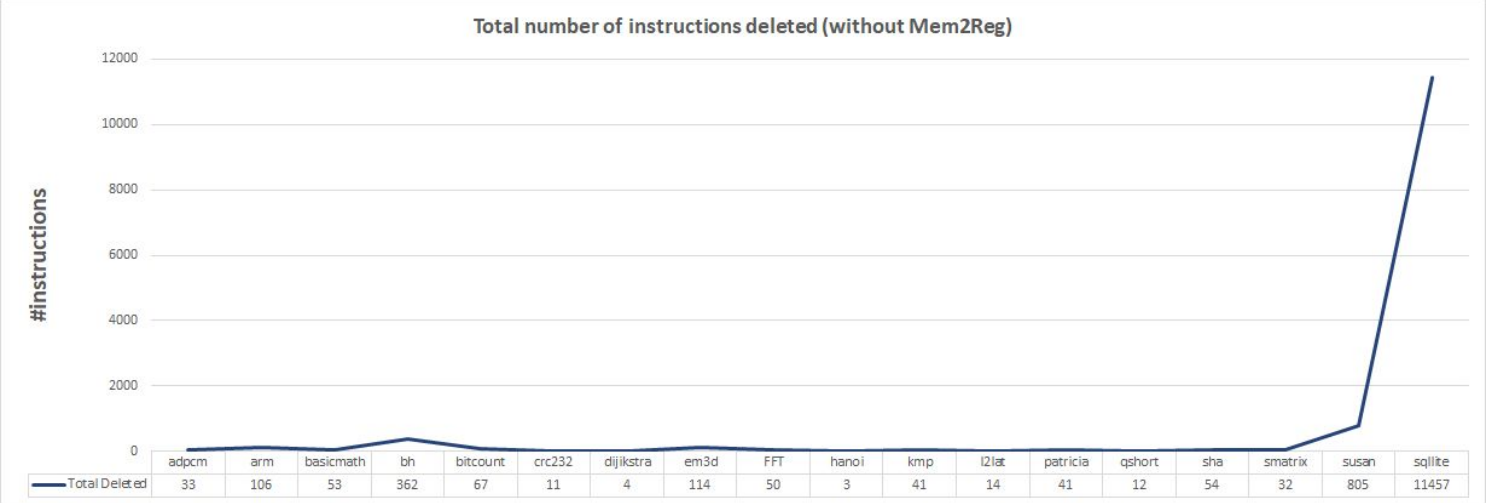


	adpcm	arm	basicmath	bh	bitcount	crc232	dijikstra	em3d	FFT	hanoi	kmp	l2lat	patricia	qshort	sha	smatrix	susan	sqllite
CSE_Dead	1	0	1	4	1	0	0	0	0	0	0	0	0	1	0	0	0	300
CSE_Basic	22	56	33	243	28	7	3	62	39	1	14	10	22	9	23	7	213	6522
CSE_Simplify	0	19	6	0	1	0	0	13	0	1	2	0	3	0	2	0	2	630
CSE_RLd	0	24	11	68	17	3	1	1	8	0	20	1	11	2	24	24	364	1825
CSE_RSt	0	0	0	0	5	0	0	0	0	0	0	1	0	0	0	0	8	19
CSE_LdSt	10	7	2	47	15	1	0	38	3	1	5	2	5	0	5	1	218	2161

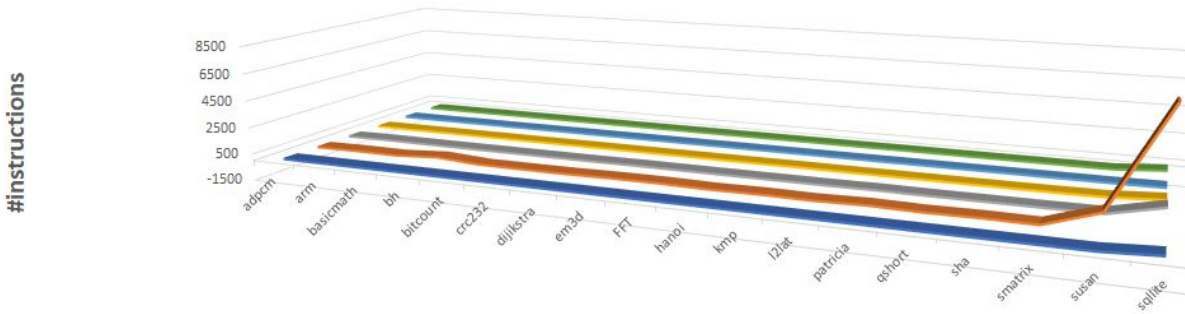
The table and graph above shows the number of instructions eliminated by the corresponding passes. We see that over all the applications, the Basic CSE pass has the highest number of instruction count followed load and load-store elimination. Without mem2reg pass, the number of deleted loads will be large because of the redundant loads. Similar is the case with redundant stores to the same address. The *sqlite* benchmark has the highest number of eliminations due to Basic CSE and

LdSt elimination suggesting a large code base and many opportunities for elimination. The benchmark *bh* also has a higher amount of load eliminations. The *susan* benchmark has zero dead code but a considerable amount of opportunities for CSE and redundant load eliminations.

The trend that is noticed is that most benchmarks have a considerable amount of literal matches which is why CSE has a high instruction count. Benchmarks also have memory instructions which is why we have load and store eliminations. Dead code is not found much in the benchmarks except for *sqlite* which has 300 dead code eliminations. Instruction simplifications in the form of constant folding are not as rare as dead code but are less than literal matches found in the code.

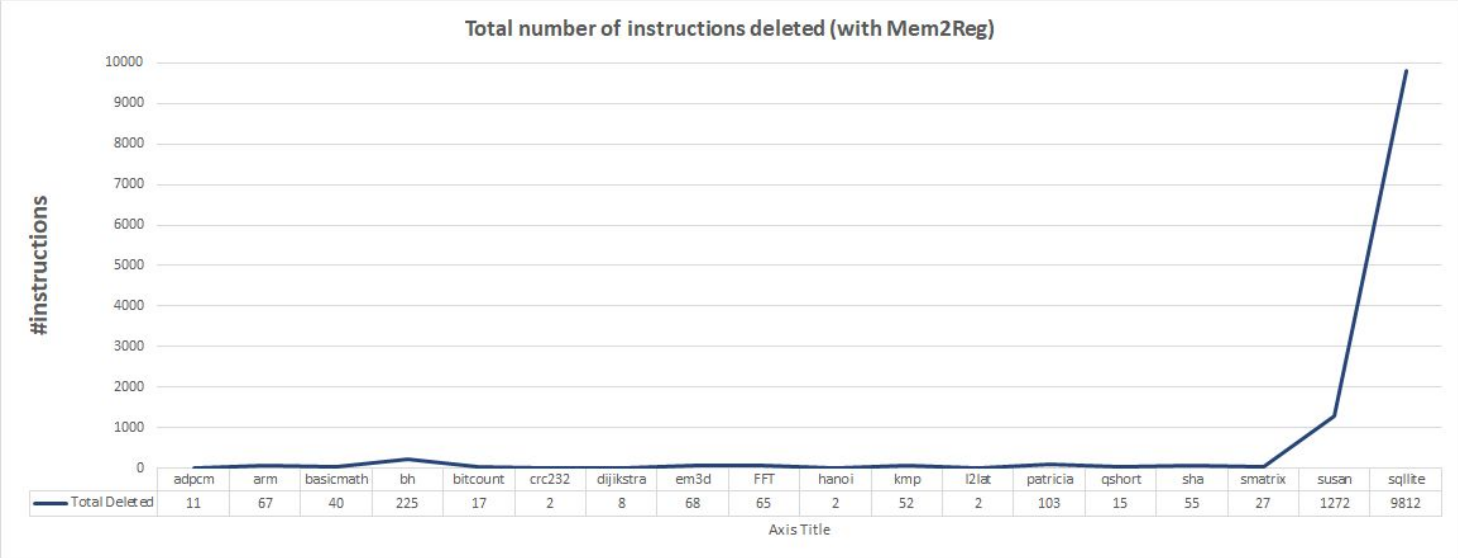


The *sqlite* benchmark has the highest number of deleted instructions which is followed by *susan* and *bh* benchmarks. This trend is seen due to the number of available instructions being fairly larger in these benchmarks. The *hanoi* benchmark has the least number of deleted instructions, 3, suggesting that it is already fairly optimized.



	adpcm	arm	basicmath	bh	bitcount	crc232	dijikstra	em3d	FFT	hanoi	kmp	l2lat	patricia	qshort	sha	smatrix	susan	sqllite
CSE_Dead	2	0	1	1	1	0	0	3	0	0	0	0	0	1	0	0	0	188
CSE_Basic	7	43	25	218	11	2	7	50	54	1	38	2	76	14	51	20	1219	8268
CSE_Simplify	2	21	6	1	2	0	0	14	7	1	2	0	19	0	2	0	16	829
CSE_RLd	0	2	1	5	0	0	1	0	4	0	11	0	4	0	1	6	34	236
CSE_RSt	0	0	3	0	1	0	0	0	0	0	0	0	0	0	0	0	1	29
CSE_LdSt	0	1	4	0	2	0	0	1	0	0	1	0	4	0	1	1	2	262

The number of instructions deleted is lesser after the mem3reg promotion which is the expected case. Unlike without the promotion pass, we see that the number of load and store eliminations are very rare except for the *sqlite* benchmark. The observed trend here is that the benchmarks have a high amount of Basic CSE optimizations, higher than the amount without mem2reg promotion. This is because Basic CSE can now eliminate the instructions which were not literal matches before the promotion pass due to the load instructions. Since the load instructions are already eliminated, the Basic CSE can now come into picture to delete the remaining instructions wherever elimination is possible. Hence, we see that load and store eliminations are very less. Instruction simplifications are greater in number in this case as compared to when promotion pass was not applied. Dead code has a similar trend as compared to when promotion pass was not used which is expected since promotion pass does not help dead code elimination.



The *sqlite* benchmark has the highest number of deleted instructions which is followed by *susan* and *bh* benchmarks. This trend is seen due to the number of available instructions being fairly larger in these benchmarks. It is also seen that the number is lesser than its counterpart where mem3reg promotion is not used. This is expected since the promotion pass will eliminate many of the instructions before the code is sent to these passes for optimization. The *sqlite* benchmark has 9812 deleted instructions after promotion pass as compared to 11,457 without the pass.