<Progress Report - Sprint 3>

SWPP 2020 Spring LLVM Project
2020.06.13

Team 4
Hyoung Uk Sul
Ahyoung Oh
Jaewook Lee
Jaeeun Lee

Progress Report: Hyoung Uk Sul

[Sprint 3] SimpleBackend Register Allocation Optimization Part 3

Change in Schedules (None)

Things were accomplished as planned in the previous sprint.

Done and Not Dones

Dones:

Up to the previous sprint, every change in depromotion focused on reducing the cost of the compiled program. To achieve such goal, I applied every method I could think up to better the performance of the compiler, relying heavily on my own heuristics rather than using already verified design methodologies. As a result, although the given benchmark tests and running "make test" passed, I feared that there could be some corner cases that catch this algorithm bad. Hence, an aspect of development that this sprint's version of depromotion algorithm focuses on is its safety. The updated implementation aims for and prevents many corner cases. Of course, it also introduces some new features to better the overall register allocation as well. The “Dones” of this sprint are the following.

1. Up to the previous sprint, the depromotion algorithm, upon facing eviction of a temporary register user, would simply put load instruction at every use of the value afterwards. This resulted in an inefficient handling of register allocation where there are multiple loads within the same basic block to load data for a temporary register user. There are many cases where these loads can be safely removed. Hence, this sprint's version of depromotion will check if there are duplicate loads within the same basic block, and if they have the same register ID and there is no dependency issue, it will remove such load instructions.
2. On the other hand, if an eviction indicated by the store to stack cannot reach a certain usage of evicted value, but yet there is another value that uses the same register ID in between, then there still occurs a dependency issue. Hence, adding loads must not rely on checking reachables blocks from the eviction point. Rather, it should check every usage of evicted value regardless of

eviction location. Thereby, this implementation fixes such issues and adds load instructions even at locations unreachable from eviction point.

3. If a value in the depromoted module turns out to be a constant (ex. add 1 2 will simply turn into a constant 3), then it does not need to and cannot use a register. However, the previous version deemed it will be a register user and hence evicted a register so that this constant value can then "use" a register. This caused a "ghost eviction" where there exists an eviction and no value uses the evicted register ID afterwards. This led to inefficient register allocation and general increase in cost of compiled programs. Hence, the new version fixes this and ensures that only valid instructions take up register IDs.
4. If a value is a ZExt instruction, but yet even its operand is another ZExt instruction, then there occurs a problem where there does not exist a value for this in a depromoted module. Hence, this implementation recursively retrieves ZExt operand until it is non-ZExt value.
5. LRU policy for assigning temporary registers to value is further enhanced. Up to this point, the policy utilized in depromotion was not strict LRU; rather, it only relied on point of creation to decide whether it is least recently used and should be evicted. However, this caused inefficiency in register handling and resulted in more load and store instructions introduced. Hence, this implementation applies strict LRU and every value's usage is recorded upon every use by other instructions as well. This resulted in a very good cost decrease in compiled programs.
6. Originally, the least number of temporary registers was 3. This was calculated upon the fact that the instruction that requires the most temporary registers required 3 temporary values. However, such calculation omitted the fact that the instruction itself also might need a register. Hence, now the value is 4.
7. When a cast instruction is only used once, in the instruction right after or two instructions after, the instruction's name was changed to its operand's name so that it can be ignored by the emitter. However, such an algorithm introduces errors where cast instruction is used only once two instructions after, but another instruction between the cast instruction and its usage is the instruction that evicts it. Hence, now the depromotion algorithm ensures that there is no eviction in between.

Not Dones:

Even though this is the last sprint, and all 6 weeks of development has been put into developing depromotion and register allocation, there are still some bits that could have been achieved but were not due to time and resource constraints. Some examples of such include:

1. The depromotion algorithm could have executed some register packing, but that would require a whole new sprint to be achieved.
2. Not all unnecessary cast instructions have been completely removed.
3. There could have been more accurate dependency tracking between registers to remove more unnecessary register allocations, but such job could not be done within polynomial time, and implementing heuristics would require yet again another sprint.

4. Global variables are always loaded even if they have already been loaded previously; this can be handled by the backend, but this could not be done within 3 sprints with all other things to implement.

Test Results:

The following shows the run of “test.sh” (explained in previous progress reports). This time, it compares the baseline compiler and the current version with **just the DepromoteRegisters turned on, everything else off**. Devoting all 3 sprints into depromotion algorithm, I was able to attain up to almost x4 times improvement on some cases, and at least around x2~3 improvement on most test cases.

```
StuartSul:swpp202001-team4 Stuart$ ./test.sh compare
________TESTING bitcount1________
input 1: PASSED with cost: 59.0000 (baseline: 218.2288)
input 2: PASSED with cost: 339.8000 (baseline: 1351.6672)
input 3: PASSED with cost: 100.6000 (baseline: 386.1456)
input 4: PASSED with cost: 17.4000 (baseline: 50.3120)
input 5: PASSED with cost: 27.8000 (baseline: 92.2912)
________TESTING bitcount2________
input 1: PASSED with cost: 85.8256 (baseline: 201.3744)
input 2: PASSED with cost: 544.9984 (baseline: 1256.4480)
input 3: PASSED with cost: 153.8512 (baseline: 357.6816)
input 4: PASSED with cost: 17.8000 (baseline: 45.0672)
input 5: PASSED with cost: 34.8064 (baseline: 84.1440)
________TESTING bitcount3________
input 1: PASSED with cost: 39.0000 (baseline: 135.0832)
input 2: PASSED with cost: 352.2000 (baseline: 1364.2656)
input 3: PASSED with cost: 93.0000 (baseline: 347.0112)
input 4: PASSED with cost: 17.4000 (baseline: 50.3120)
input 5: PASSED with cost: 28.2000 (baseline: 92.6976)
________TESTING bitcount4________
input 1: PASSED with cost: 9031.2656 (baseline: 25400.8304)
input 2: PASSED with cost: 9031.8496 (baseline: 25403.6560)
input 3: PASSED with cost: 9032.0496 (baseline: 25401.6144)
input 4: PASSED with cost: 9031.2368 (baseline: 25400.8016)
input 5: PASSED with cost: 9031.2400 (baseline: 25400.8048)
________TESTING bitcount5________
input 1: PASSED with cost: 97.4480 (baseline: 392.6752)
input 2: PASSED with cost: 224.8480 (baseline: 872.3872)
input 3: PASSED with cost: 115.6512 (baseline: 460.9920)
input 4: PASSED with cost: 79.2480 (baseline: 324.4704)
input 5: PASSED with cost: 97.4480 (baseline: 392.6496)
________TESTING bubble_sort________
input 1: PASSED with cost: 2024.7584 (baseline: 6628.9360)
input 2: PASSED with cost: 143991.8176 (baseline: 562437.2464)
input 3: PASSED with cost: 14212790.0339 (baseline: 61987402.0422)
________TESTING collatz________
```

input 1: PASSED with cost: 28.2064 (baseline: 68.4400)
input 2: PASSED with cost: 28.2064 (baseline: 68.4400)
_____TESTING gcd_____
input 1: PASSED with cost: 18.2000 (baseline: 46.5024)
input 2: PASSED with cost: 34.8000 (baseline: 105.4816)
input 3: PASSED with cost: 68.0000 (baseline: 223.4272)
input 4: PASSED with cost: 297.8000 (baseline: 1043.0336)
_____TESTING prime_____
input 1: PASSED with cost: 50.5568 (baseline: 117.4400)
input 2: PASSED with cost: 3185.0672 (baseline: 6678.1008)
input 3: PASSED with cost: 695783.3616 (baseline: 1548664.4224)
input 4: PASSED with cost: 2542265.5536 (baseline: 5628813.2864)
_____TESTING binary_tree_____
input 1: PASSED with cost: 919.3712 (baseline: 2402.8096)
input 2: PASSED with cost: 1762.5760 (baseline: 4310.6816)
input 3: PASSED with cost: 27900.4352 (baseline: 66113.1696)
input 4: PASSED with cost: 448803.6208 (baseline: 1033485.1248)
input 5: PASSED with cost: 1316127748.4000 (baseline: 1977050335.0849)
_____TESTING matmul1_____
input 1: PASSED with cost: 994.2240 (baseline: 2608.1088)
input 2: PASSED with cost: 4824.3296 (baseline: 13201.6704)
input 3: PASSED with cost: 210548.9120 (baseline: 594334.2144)
_____TESTING matmul2_____
input 1: PASSED with cost: 945.7888 (baseline: 2804.3072)
input 2: PASSED with cost: 4846.6720 (baseline: 15513.8560)
input 3: PASSED with cost: 235669.8784 (baseline: 807544.4224)
_____TESTING matmul3_____
input 1: PASSED with cost: 2932.3152 (baseline: 7797.6480)
input 2: PASSED with cost: 2932.3152 (baseline: 7797.6480)
input 3: PASSED with cost: 116840.8320 (baseline: 313852.4352)
_____TESTING matmul4_____
input 1: PASSED with cost: 6385.5280 (baseline: 20952.6976)
input 2: PASSED with cost: 6385.5280 (baseline: 20952.6976)
input 3: PASSED with cost: 338179.8688 (baseline: 1155651.0784)
_____TESTING rmq1d_naive_____
input 1: PASSED with cost: 121.3296 (baseline: 370.2880)
input 2: PASSED with cost: 788.6384 (baseline: 3041.7728)
input 3: PASSED with cost: 1189.1024 (baseline: 4770.9952)
input 4: PASSED with cost: 1414.1568 (baseline: 5578.9824)
input 5: PASSED with cost: 466224.1232 (baseline: 2439922.4944)
input 6: PASSED with cost: 2124241.5840 (baseline: 11471977.9616)
input 7: PASSED with cost: 404726347.5792 (baseline: 2721615862.1304)
_____TESTING rmq1d_sparsetable_____
input 1: PASSED with cost: 874.2624 (baseline: 2568.7232)
input 2: PASSED with cost: 7419.3488 (baseline: 22418.3616)
input 3: PASSED with cost: 10834.6768 (baseline: 32883.3120)
input 4: PASSED with cost: 9543.0928 (baseline: 27376.3424)
input 5: PASSED with cost: 644409.1616 (baseline: 1956788.2496)
input 6: PASSED with cost: 1103059.4816 (baseline: 2910040.9664)

```
input 7: PASSED with cost: 100306858.0569 (baseline: 135120657.9315)
________TESTING rmq2d_naive________
input 1: PASSED with cost: 277.3936 (baseline: 715.2544)
input 2: PASSED with cost: 1527.6672 (baseline: 4967.7056)
input 3: PASSED with cost: 11901.4144 (baseline: 42847.2224)
input 4: PASSED with cost: 10701.2640 (baseline: 32249.0896)
input 5: PASSED with cost: 3973835.0816 (baseline: 22665902.4304)
________TESTING rmq2d_sparsetable________
input 1: PASSED with cost: 3002.9456 (baseline: 8994.4592)
input 2: PASSED with cost: 17195.8304 (baseline: 52414.1120)
input 3: PASSED with cost: 94319.8544 (baseline: 289182.7600)
input 4: PASSED with cost: 133284.9056 (baseline: 330169.2336)
input 5: PASSED with cost: 884057682.4709 (baseline: 995120658.5224)
```

- New Test Cases:

FileCheck 1 (RegisterAllocationOpt-Test7.ll)	
Before Optimization	After Optimization

`Returned: 0`
`Cost: 71071.6400`
`Max heap usage (bytes): 80`
	`Returned: 0`
	`Cost: 20235.7472`
	`Max heap usage (bytes): 80`

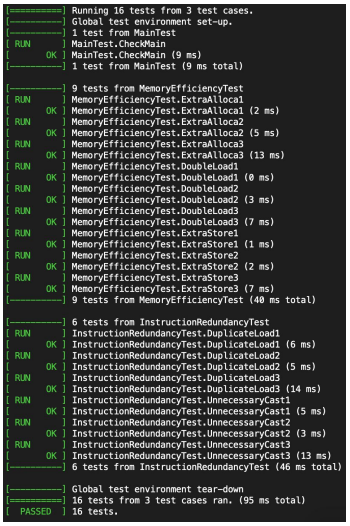
FileCheck 2 (RegisterAllocationOpt-Test8.ll	
Before Optimization	After Optimization

`Returned: 0`
`Cost: 1782.1120`
`Max heap usage (bytes): 0`
	`Returned: 0`
	`Cost: 504.9696`
	`Max heap usage (bytes): 0`

FileCheck 3 (RegisterAllocationOpt-Test9.ll)	
Before Optimization	After Optimization

`Returned: 0`
`Cost: 46966.2224`
`Max heap usage (bytes): 112`
	`Returned: 0`
	`Cost: 18050.4304`
	`Max heap usage (bytes): 112`

GoogleTest (Memory Redundancy Test + Instruction Redundancy Test)	


[==========] Running 16 tests from 3 test cases.
[----------] Global test environment set-up.
[----------] 1 test from MainTest
[RUN   ] MainTest.CheckMain
[OK] MainTest.CheckMain (9 ms)
[----------] 1 test from MainTest (9 ms total)
[----------] 9 tests from MemoryEfficiencyTest
[RUN   ] MemoryEfficiencyTest.ExtraAlloca1
[OK] MemoryEfficiencyTest.ExtraAlloca1 (2 ms)
[RUN   ] MemoryEfficiencyTest.ExtraAlloca2
[OK] MemoryEfficiencyTest.ExtraAlloca2 (5 ms)
[RUN   ] MemoryEfficiencyTest.ExtraAlloca3
[OK] MemoryEfficiencyTest.ExtraAlloca3 (13 ms)
[RUN   ] MemoryEfficiencyTest.DoubleLoad1
[OK] MemoryEfficiencyTest.DoubleLoad1 (0 ms)
[RUN   ] MemoryEfficiencyTest.DoubleLoad2
[OK] MemoryEfficiencyTest.DoubleLoad2 (3 ms)
[RUN   ] MemoryEfficiencyTest.DoubleLoad3
[OK] MemoryEfficiencyTest.DoubleLoad3 (7 ms)
[RUN   ] MemoryEfficiencyTest.ExtraStore1
[OK] MemoryEfficiencyTest.ExtraStore1 (1 ms)
[RUN   ] MemoryEfficiencyTest.ExtraStore2
[OK] MemoryEfficiencyTest.ExtraStore2 (2 ms)
[RUN   ] MemoryEfficiencyTest.ExtraStore3
[OK] MemoryEfficiencyTest.ExtraStore3 (7 ms)
[----------] 9 tests from MemoryEfficiencyTest (40 ms total)
[----------] 6 tests from InstructionRedundancyTest
[RUN   ] InstructionRedundancyTest.DuplicateLoad1
[OK] InstructionRedundancyTest.DuplicateLoad1 (6 ms)
[RUN   ] InstructionRedundancyTest.DuplicateLoad2
[OK] InstructionRedundancyTest.DuplicateLoad2 (5 ms)
[RUN   ] InstructionRedundancyTest.DuplicateLoad3
[OK] InstructionRedundancyTest.DuplicateLoad3 (14 ms)
[RUN   ] InstructionRedundancyTest.UnnecessaryCast1
[OK] InstructionRedundancyTest.UnnecessaryCast1 (5 ms)
[RUN   ] InstructionRedundancyTest.UnnecessaryCast2
[OK] InstructionRedundancyTest.UnnecessaryCast2 (3 ms)
[RUN   ] InstructionRedundancyTest.UnnecessaryCast3
[OK] InstructionRedundancyTest.UnnecessaryCast3 (13 ms)
[----------] 6 tests from InstructionRedundancyTest (46 ms total)
[----------] Global test environment tear-down
[==========] 16 tests from 3 test cases ran. (95 ms total)
[PASSED] 16 tests.

Progress Report: Ahyoung Oh

In sprint 3, I pushed two pull requests in total : 'Function Optimization' and 'Assembly Emitter Memory Access Optimization Part 2'. PR1 is about implementing existing passes as I was responsible for doing so in this sprint.

[Sprint 3] Function Optimization

Change in Schedules (None)

Things were accomplished as planned in the previous sprint.

Done and Not Dones

As I was responsible for implementing existing passes for this sprint, I've added two existing passes related to the function optimization.

1. Tail Call Elimination : This pass helps avoid allocating a new stack frame for a function because the caller will simply return the value that it gets from the callee. If it can prove that callees do not access their caller stack frame, they are marked as eligible for tail call elimination (by the code generator).

2. Sparse Conditional Constant Propagation : SCCP pass does sparse conditional constant propagation and merging, like following:

   Assumes values are constant unless proven otherwise

- Assumes BasicBlocks are dead unless proven otherwise
- Proves values to be constant, and replaces them with constants
- Proves conditional branches to be unconditional
  (from llvm.org)
  This pass works well with DCE pass so I've changed the order of the passes.

In addition to these two passes, I added one more existing pass named UnreachableBlockElimPass() during the code review process.

3. Unreachable Block Elimination : By implementing this pass, our team doesn't have to care more about unreachable basic block potentially being a problem.

Test Results:

- Existing Test Cases

This optimization decreased cost substantially in the following highlighted cases.

_____TESTING bitcount1_____
input 1: PASSED with cost: 58.2000 (sprint2: 58.2000) (sprint1: 75.4608) (baseline: 218.2288)
input 2: PASSED with cost: 333.6000 (sprint2: 333.6000) (sprint1: 443.0064) (baseline: 1351.6672)
input 3: PASSED with cost: 99.0000 (sprint2: 99.0000) (sprint1: 129.9120) (baseline: 386.1456)
input 4: PASSED with cost: 17.4000 (sprint2: 17.4000) (sprint1: 21.0096) (baseline: 50.3120)
input 5: PASSED with cost: 27.6000 (sprint2: 27.6000) (sprint1: 34.6224) (baseline: 92.2912)
_____TESTING bitcount2_____
input 1: PASSED with cost: 60.2000 (sprint2: 85.0256) (sprint1: 86.0256) (baseline: 201.3744)
input 2: PASSED with cost: 335.6000 (sprint2: 538.7984) (sprint1: 539.7984) (baseline: 1256.4480)
input 3: PASSED with cost: 101.0000 (sprint2: 152.2512) (sprint1: 153.2512) (baseline: 357.6816)
input 4: PASSED with cost: 19.4000 (sprint2: 17.8000) (sprint1: 18.8000) (baseline: 45.0672)
input 5: PASSED with cost: 29.6000 (sprint2: 34.6064) (sprint1: 35.6064) (baseline: 84.1440)
_____TESTING bitcount3_____
input 1: PASSED with cost: 39.0000 (sprint2: 39.0000) (sprint1: 49.4352) (baseline: 135.0832)
input 2: PASSED with cost: 352.2000 (sprint2: 352.2000) (sprint1: 461.6064) (baseline: 1364.2656)
input 3: PASSED with cost: 93.0000 (sprint2: 93.0000) (sprint1: 120.4992) (baseline: 347.0112)
input 4: PASSED with cost: 17.4000 (sprint2: 17.4000) (sprint1: 21.0096) (baseline: 50.3120)
input 5: PASSED with cost: 28.2000 (sprint2: 28.2000) (sprint1: 35.2224) (baseline: 92.6976)
_____TESTING bitcount4_____
input 1: PASSED with cost: 9030.6656 (sprint2: 9030.6656) (sprint1: 9783.1040) (baseline: 25400.8304)
input 2: PASSED with cost: 9031.2496 (sprint2: 9031.2496) (sprint1: 9783.8912) (baseline: 25403.6560)
input 3: PASSED with cost: 9031.4496 (sprint2: 9031.4496) (sprint1: 9783.8880) (baseline: 25401.6144)
input 4: PASSED with cost: 9030.6368 (sprint2: 9030.6368) (sprint1: 9783.0752) (baseline: 25400.8016)
input 5: PASSED with cost: 9030.6400 (sprint2: 9030.6400) (sprint1: 9783.0784) (baseline: 25400.8048)
_____TESTING bitcount5_____
input 1: PASSED with cost: 107.4816 (sprint2: 110.6816) (sprint1: 164.8288) (baseline: 392.6752)
input 2: PASSED with cost: 222.3008 (sprint2: 236.7008) (sprint1: 332.8736) (baseline: 872.3872)
input 3: PASSED with cost: 123.9008 (sprint2: 128.7008) (sprint1: 188.8384) (baseline: 460.9920)
input 4: PASSED with cost: 91.0528 (sprint2: 92.6528) (sprint1: 140.8224) (baseline: 324.4704)
input 5: PASSED with cost: 107.4560 (sprint2: 110.6560) (sprint1: 164.8288) (baseline: 392.6496)
_____TESTING bubble_sort_____
input 1: PASSED with cost: 1873.5584 (sprint2: 1873.5584) (sprint1: 3644.9712) (baseline: 6628.9360)
input 2: PASSED with cost: 130274.6176 (sprint2: 130274.6176) (sprint1: 315078.2912) (baseline: 562437.2464)
input 3: PASSED with cost: 12714888.8343 (sprint2: 12714888.8343) (sprint1: 36245394.9259) (baseline: 61987402.0422)
_____TESTING collatz_____
input 1: PASSED with cost: 29.8064 (sprint2: 28.2064) (sprint1: 29.2448) (baseline: 68.4400)
input 2: PASSED with cost: 29.8064 (sprint2: 28.2064) (sprint1: 29.2448) (baseline: 68.4400)
_____TESTING gcd_____
input 1: PASSED with cost: 20.4000 (sprint2: 18.2000) (sprint1: 22.2000) (baseline: 46.5024)
input 2: PASSED with cost: 32.6000 (sprint2: 34.8000) (sprint1: 53.0384) (baseline: 105.4816)
input 3: PASSED with cost: 57.0000 (sprint2: 68.0000) (sprint1: 114.7152) (baseline: 223.4272)
input 4: PASSED with cost: 225.2000 (sprint2: 297.8000) (sprint1: 542.4528) (baseline: 1043.0336)
_____TESTING prime_____
input 1: PASSED with cost: 47.1568 (sprint2: 47.1568) (sprint1: 52.3568) (baseline: 117.4400)
input 2: PASSED with cost: 2553.4896 (sprint2: 3169.6672) (sprint1: 3778.3760) (baseline: 6678.1008)
input 3: PASSED with cost: 538353.9648 (sprint2: 685431.9616) (sprint1: 902276.9104) (baseline: 1548664.4224)
input 4: PASSED with cost: 1974698.7648 (sprint2: 2509506.1536) (sprint1: 3309086.8640) (baseline: 5628813.2864)
_____TESTING binary_tree_____
input 1: PASSED with cost: 903.9712 (sprint2: 920.3712) (sprint1: 1414.0816) (baseline: 2402.8096)

input 2: PASSED with cost: 1747.7488 (sprint2: 1764.1488) (sprint1: 2721.3600) (baseline: 4310.6816)
input 3: PASSED with cost: 27992.7280 (sprint2: 28084.1280) (sprint1: 43829.8000) (baseline: 66113.1696)
input 4: PASSED with cost: 451561.4128 (sprint2: 452297.8128) (sprint1: 715705.5104) (baseline: 1033485.1248)
input 5: PASSED with cost: 1317017229.3856 (sprint2: 1317054853.7856) (sprint1: 1874833611.0864) (baseline: 1977050335.0849)
_____TESTING matmul1_____
input 1: PASSED with cost: 1080.0608 (sprint2: 1080.0608) (baseline: 2608.1088)
input 2: PASSED with cost: 5593.2960 (sprint2: 5593.2960) (baseline: 13201.6704)
input 3: PASSED with cost: 262675.9968 (sprint2: 262675.9968) (baseline: 594334.2144)
_____TESTING matmul2_____
input 1: PASSED with cost: 924.9600 (sprint2: 924.9600) (baseline: 2804.3072)
input 2: PASSED with cost: 4607.4304 (sprint2: 4607.4304) (baseline: 15513.8560)
input 3: PASSED with cost: 216849.7600 (sprint2: 216849.7600) (baseline: 807544.4224)
_____TESTING matmul3_____
input 1: PASSED with cost: 2935.5888 (sprint2: 2935.5888) (baseline: 7797.6480)
input 2: PASSED with cost: 2935.5888 (sprint2: 2935.5888) (baseline: 7797.6480)
input 3: PASSED with cost: 111759.8112 (sprint2: 111759.8112) (baseline: 313852.4352)
_____TESTING matmul4_____
input 1: PASSED with cost: 4754.8512 (sprint2: 4754.8512) (baseline: 20952.6976)
input 2: PASSED with cost: 4754.8512 (sprint2: 4754.8512) (baseline: 20952.6976)
input 3: PASSED with cost: 234463.7616 (sprint2: 234463.7616) (baseline: 1155651.0784)
_____TESTING rmq1d_naive_____
input 1: PASSED with cost: 112.4976 (sprint2: 112.4976) (baseline: 370.2880)
input 2: PASSED with cost: 802.5488 (sprint2: 802.5488) (baseline: 3041.7728)
input 3: PASSED with cost: 1220.9936 (sprint2: 1220.9936) (baseline: 4770.9952)
input 4: PASSED with cost: 1379.9520 (sprint2: 1379.9520) (baseline: 5578.9824)
input 5: PASSED with cost: 470616.1904 (sprint2: 470616.1904) (baseline: 2439922.4944)
input 6: PASSED with cost: 2125425.9712 (sprint2: 2125425.9712) (baseline: 11471977.9616)
input 7: PASSED with cost: 404698149.5664 (sprint2: 404698149.5664) (baseline: 2721615862.1304)
_____TESTING rmq1d_sparsetable_____
input 1: PASSED with cost: 810.1376 (sprint2: 810.1376) (baseline: 2568.7232)
input 2: PASSED with cost: 7184.0880 (sprint2: 7184.0880) (baseline: 22418.3616)
input 3: PASSED with cost: 10511.9632 (sprint2: 10511.9632) (baseline: 32883.3120)
input 4: PASSED with cost: 9290.9248 (sprint2: 9290.9248) (baseline: 27376.3424)
input 5: PASSED with cost: 627964.2848 (sprint2: 627964.2848) (baseline: 1956788.2496)
input 6: PASSED with cost: 1095536.2560 (sprint2: 1095536.2560) (baseline: 2910040.9664)
input 7: PASSED with cost: 103543434.6825 (sprint2: 103543434.6825) (baseline: 135120657.9315)
_____TESTING rmq2d_naive_____
input 1: PASSED with cost: 277.5056 (sprint2: 277.5056) (baseline: 715.2544)
input 2: PASSED with cost: 1558.0288 (sprint2: 1558.0288) (baseline: 4967.7056)
input 3: PASSED with cost: 12268.1504 (sprint2: 12268.1504) (baseline: 42847.2224)
input 4: PASSED with cost: 10646.2512 (sprint2: 10646.2512) (baseline: 32249.0896)
input 5: PASSED with cost: 3966246.5872 (sprint2: 3966246.5872) (baseline: 22665902.4304)
_____TESTING rmq2d_sparsetable_____
input 1: PASSED with cost: 2961.7776 (sprint2: 2961.7776) (baseline: 8994.4592)
input 2: PASSED with cost: 16860.2464 (sprint2: 16860.2464) (baseline: 52414.1120)
input 3: PASSED with cost: 92113.7072 (sprint2: 92113.7072) (baseline: 289182.7600)
input 4: PASSED with cost: 125085.9872 (sprint2: 125085.9872) (baseline: 330169.2336)
input 5: PASSED with cost: 878957463.0040 (sprint2: 878957463.0040) (baseline: 995120658.5224)

- New Test Cases : input .ll files are in the git repository

<table>
<tr><th colspan="2">FileCheck 1 (Tail Call Elim)</th></tr>
<tr><td>Before Optimization</td><td>After Optimization</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><th colspan="2">FileCheck 2</th></tr>
<tr><td>Before Optimization</td><td>After Optimization</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><th colspan="2">FileCheck 3</th></tr>
<tr><td>Before Optimization</td><td>After Optimization</td></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

[Sprint 3] Reordering Memory Access (Assembly Emitter Memory Access Optimization Part 2)

Change in Schedules (None)

Things were accomplished as planned in the previous sprint.

Done and Not Dones

In this sprint, for “Assembly Emitter Memory Access Optimization Part 2”, the optimizations that could be done by modifying the AssemblyEmitter.cpp code are implemented.

The cases to emit reset are generalized to the function level. In sprint2, for part 1 of this optimization, a reset instruction was emitted only when it’s 100% sure that the memory access has changed(either from heap to stack or from stack to heap) within the same block. However, it’d be more efficient if the memory access change could be detected within the same function. I've done this by analyzing the reachable blocks. So now, more reset instructions are emitted(due to cases like the head remaining in the stack for the BB1 and then the head moving to the heap for the next block).

Not Done :

I wasn’t able to erase instructions like r1 = mul r13 1 64 shown below because they were more complicated than I’ve thought to erase those by modifying AssemblyEmitter.cpp.

```
r1 = mul r13 1 64
```

With the example above, I tried to narrow down the cases to where r1 has only one use and the user of that use is the instruction right after emitCopy. I did till that point of detection but it was hard to replace all the uses of r1 with r13 as they had different types which led to assertion error. Therefore, to be safe, after discussing with our teammates, I decided not to implement this optimization.

Test Results:

- Existing Test Cases

This optimization decreased cost substantially in the following highlighted cases.

```
________TESTING bitcount1________
input 1: PASSED with cost: 58.2000 (sprint2: 58.2000)  (sprint1: 75.4608) (baseline: 218.2288)
input 2: PASSED with cost: 333.6000 (sprint2: 333.6000)  (sprint1: 443.0064) (baseline: 1351.6672)
input 3: PASSED with cost: 99.0000 (sprint2: 99.0000)  (sprint1: 129.9120) (baseline: 386.1456)
input 4: PASSED with cost: 17.4000 (sprint2: 17.4000)  (sprint1: 21.0096) (baseline: 50.3120)
input 5: PASSED with cost: 27.6000 (sprint2: 27.6000)  (sprint1: 34.6224) (baseline: 92.2912)
________TESTING bitcount2________
input 1: PASSED with cost: 85.0256 (sprint2: 85.0256)  (sprint1: 86.0256) (baseline: 201.3744)
input 2: PASSED with cost: 538.7984 (sprint2: 538.7984)  (sprint1: 539.7984) (baseline: 1256.4480)
input 3: PASSED with cost: 152.2512 (sprint2: 152.2512)  (sprint1: 153.2512) (baseline: 357.6816)
```

input 4: PASSED with cost: 17.8000 (sprint2: 17.8000) (sprint1: 18.8000) (baseline: 45.0672)
input 5: PASSED with cost: 34.6064 (sprint2: 34.6064) (sprint1: 35.6064) (baseline: 84.1440)
_____TESTING bitcount3_____
input 1: PASSED with cost: 39.0000 (sprint2: 39.0000) (sprint1: 49.4352) (baseline: 135.0832)
input 2: PASSED with cost: 352.2000 (sprint2: 352.2000) (sprint1: 461.6064) (baseline: 1364.2656)
input 3: PASSED with cost: 93.0000 (sprint2: 93.0000) (sprint1: 120.4992) (baseline: 347.0112)
input 4: PASSED with cost: 17.4000 (sprint2: 17.4000) (sprint1: 21.0096) (baseline: 50.3120)
input 5: PASSED with cost: 28.2000 (sprint2: 28.2000) (sprint1: 35.2224) (baseline: 92.6976)
_____TESTING bitcount4_____
input 1: PASSED with cost: 9030.6656 (sprint2: 9030.6656) (sprint1: 9783.1040) (baseline: 25400.8304)
input 2: PASSED with cost: 9031.2496 (sprint2: 9031.2496) (sprint1: 9783.8912) (baseline: 25403.6560)
input 3: PASSED with cost: 9031.4496 (sprint2: 9031.4496) (sprint1: 9783.8880) (baseline: 25401.6144)
input 4: PASSED with cost: 9030.6368 (sprint2: 9030.6368) (sprint1: 9783.0752) (baseline: 25400.8016)
input 5: PASSED with cost: 9030.6400 (sprint2: 9030.6400) (sprint1: 9783.0784) (baseline: 25400.8048)
_____TESTING bitcount5_____
input 1: PASSED with cost: 110.6816 (sprint2: 110.6816) (sprint1: 164.8288) (baseline: 392.6752)
input 2: PASSED with cost: 236.7008 (sprint2: 236.7008) (sprint1: 332.8736) (baseline: 872.3872)
input 3: PASSED with cost: 128.7008 (sprint2: 128.7008) (sprint1: 188.8384) (baseline: 460.9920)
input 4: PASSED with cost: 92.6528 (sprint2: 92.6528) (sprint1: 140.8224) (baseline: 324.4704)
input 5: PASSED with cost: 110.6560 (sprint2: 110.6560) (sprint1: 164.8288) (baseline: 392.6496)
_____TESTING bubble_sort_____
input 1: PASSED with cost: 1852.5344 (sprint2: 1873.5584) (sprint1: 3644.9712) (baseline: 6628.9360)
input 2: PASSED with cost: 130064.3776 (sprint2: 130274.6176) (sprint1: 315078.2912) (baseline: 562437.2464)
input 3: PASSED with cost: 12712786.4343 (sprint2: 12714888.8343) (sprint1: 36245394.9259) (baseline: 61987402.0422)
_____TESTING collatz_____
input 1: PASSED with cost: 28.2064 (sprint2: 28.2064) (sprint1: 29.2448) (baseline: 68.4400)
input 2: PASSED with cost: 28.2064 (sprint2: 28.2064) (sprint1: 29.2448) (baseline: 68.4400)
_____TESTING gcd_____
input 1: PASSED with cost: 18.2000 (sprint2: 18.2000) (sprint1: 22.2000) (baseline: 46.5024)
input 2: PASSED with cost: 34.8000 (sprint2: 34.8000) (sprint1: 53.0384) (baseline: 105.4816)
input 3: PASSED with cost: 68.0000 (sprint2: 68.0000) (sprint1: 114.7152) (baseline: 223.4272)
input 4: PASSED with cost: 297.8000 (sprint2: 297.8000) (sprint1: 542.4528) (baseline: 1043.0336)
_____TESTING prime_____
input 1: PASSED with cost: 45.0576 (sprint2: 47.1568) (sprint1: 52.3568) (baseline: 117.4400)
input 2: PASSED with cost: 3211.6384 (sprint2: 3169.6672) (sprint1: 3778.3760) (baseline: 6678.1008)
input 3: PASSED with cost: 686185.0688 (sprint2: 685431.9616) (sprint1: 902276.9104) (baseline: 1548664.4224)
input 4: PASSED with cost: 2511565.3472 (sprint2: 2509506.1536) (sprint1: 3309086.8640) (baseline: 5628813.2864)
_____TESTING binary_tree_____
input 1: PASSED with cost: 909.8592 (sprint2: 920.3712) (sprint1: 1414.0816) (baseline: 2402.8096)
input 2: PASSED with cost: 1749.4320 (sprint2: 1764.1488) (sprint1: 2721.3600) (baseline: 4310.6816)
input 3: PASSED with cost: 27962.1888 (sprint2: 28084.1280) (sprint1: 43829.8000) (baseline: 66113.1696)
input 4: PASSED with cost: 450872.3856 (sprint2: 452297.8128) (sprint1: 715705.5104) (baseline: 1033485.1248)
input 5: PASSED with cost: 1316899145.8368 (sprint2: 1317054853.7856) (sprint1: 1874833611.0864) (baseline: 1977050335.0849)
_____TESTING matmul1_____
input 1: PASSED with cost: 1080.0608 (sprint2: 1080.0608) (baseline: 2608.1088)
input 2: PASSED with cost: 5593.2960 (sprint2: 5593.2960) (baseline: 13201.6704)
input 3: PASSED with cost: 262675.9968 (sprint2: 262675.9968) (baseline: 594334.2144)
_____TESTING matmul2_____

input 1: PASSED with cost: 924.9600 (sprint2: 924.9600) (baseline: 2804.3072)
input 2: PASSED with cost: 4607.4304 (sprint2: 4607.4304) (baseline: 15513.8560)
input 3: PASSED with cost: 216849.7600 (sprint2: 216849.7600) (baseline: 807544.4224)
_____TESTING matmul3_____
input 1: PASSED with cost: 2935.5888 (sprint2: 2935.5888) (baseline: 7797.6480)
input 2: PASSED with cost: 2935.5888 (sprint2: 2935.5888) (baseline: 7797.6480)
input 3: PASSED with cost: 111759.8112 (sprint2: 111759.8112) (baseline: 313852.4352)
_____TESTING matmul4_____
input 1: PASSED with cost: 4754.8512 (sprint2: 4754.8512) (baseline: 20952.6976)
input 2: PASSED with cost: 4754.8512 (sprint2: 4754.8512) (baseline: 20952.6976)
input 3: PASSED with cost: 234463.7616 (sprint2: 234463.7616) (baseline: 1155651.0784)
_____TESTING rmq1d_naive_____
input 1: PASSED with cost: 112.4976 (sprint2: 112.4976) (baseline: 370.2880)
input 2: PASSED with cost: 802.5488 (sprint2: 802.5488) (baseline: 3041.7728)
input 3: PASSED with cost: 1220.9936 (sprint2: 1220.9936) (baseline: 4770.9952)
input 4: PASSED with cost: 1379.9520 (sprint2: 1379.9520) (baseline: 5578.9824)
input 5: PASSED with cost: 470616.1904 (sprint2: 470616.1904) (baseline: 2439922.4944)
input 6: PASSED with cost: 2125425.9712 (sprint2: 2125425.9712) (baseline: 11471977.9616)
input 7: PASSED with cost: 404698149.5664 (sprint2: 404698149.5664) (baseline: 2721615862.1304)
_____TESTING rmq1d_sparsetable_____
input 1: PASSED with cost: 810.1376 (sprint2: 810.1376) (baseline: 2568.7232)
input 2: PASSED with cost: 7184.0880 (sprint2: 7184.0880) (baseline: 22418.3616)
input 3: PASSED with cost: 10511.9632 (sprint2: 10511.9632) (baseline: 32883.3120)
input 4: PASSED with cost: 9290.9248 (sprint2: 9290.9248) (baseline: 27376.3424)
input 5: PASSED with cost: 627964.2848 (sprint2: 627964.2848) (baseline: 1956788.2496)
input 6: PASSED with cost: 1095536.2560 (sprint2: 1095536.2560) (baseline: 2910040.9664)
input 7: PASSED with cost: 103543434.6825 (sprint2: 103543434.6825) (baseline: 135120657.9315)
_____TESTING rmq2d_naive_____
input 1: PASSED with cost: 277.5056 (sprint2: 277.5056) (baseline: 715.2544)
input 2: PASSED with cost: 1558.0288 (sprint2: 1558.0288) (baseline: 4967.7056)
input 3: PASSED with cost: 12268.1504 (sprint2: 12268.1504) (baseline: 42847.2224)
input 4: PASSED with cost: 10646.2512 (sprint2: 10646.2512) (baseline: 32249.0896)
input 5: PASSED with cost: 3966246.5872 (sprint2: 3966246.5872) (baseline: 22665902.4304)
_____TESTING rmq2d_sparsetable_____
input 1: PASSED with cost: 2961.7776 (sprint2: 2961.7776) (baseline: 8994.4592)
input 2: PASSED with cost: 16860.2464 (sprint2: 16860.2464) (baseline: 52414.1120)
input 3: PASSED with cost: 92113.7072 (sprint2: 92113.7072) (baseline: 289182.7600)
input 4: PASSED with cost: 125085.9872 (sprint2: 125085.9872) (baseline: 330169.2336)
input 5: PASSED with cost: 878957463.0040 (sprint2: 878957463.0040) (baseline: 995120658.5224)

- New Test Cases : input .ll files are in the git repository

FileCheck 1	
Before Optimization	After Optimization

FileCheck 2

Before Optimization	After Optimization

FileCheck 3

Before Optimization	After Optimization

Progress Report: Jaewook Lee

[Sprint 3] Global Variable Load/Store Optimization

Change in Schedules : Loop Invariant Code Motion for GV Load -> General GV Load/Store Optimization

At first, I was planning to do the optimization only for the loop invariant code motion for global variable load instructions. First idea was just to hoist the load instruction. I noticed in sprint 2 that built-in passes are not hoisting GV load instructions in the loop. However, while implementing LICMGVLoadPass, I noticed that by hoisting several GV load instructions, the same load will be executed in the preheader block, thus causing unnecessary loads. Therefore, I implemented additional optimization that will replace the same load instructions.

Also, I noticed that GV load/store can be found in the AssemblyEmitter stage, which will reduce the cost when reset instruction is emitted before load or store. Therefore, I edited some parts in AssemblyEmitter.cpp.

Dones and Not Dones

Dones:

For this optimization I imported two important optimizations that are related to the global variables.

1. LICMGVLoadPass

First, I made an optimization pass (function level) for LICM GV Load instructions. For some reason, while importing LICMPass during the sprint 2, the GV load instructions in the loop were not hoisted, although it could be. Therefore, I made this pass to hoist the GV load instructions in the loop to its preheader block. With this pass, the cost caused by repetition of GV Load in the loop is reduced.
LICMGVLoadPass Explanation :
This pass hoists the Global Variable(GV) Load Instruction I in the loop L to the preheader.
Before it hoists the load instruction it checks following conditions :
a) Loop L has prehead where I can be hoisted.
b) GV is not updated (stored) in the loop L
c) No other load instruction that loads same GV is already hoisted
(if there is one, replace use of I with it)
After hoisting, if GV is stored in the prehead block, use that value without loading it again.
Lastly, remove all the replaced load instructions.

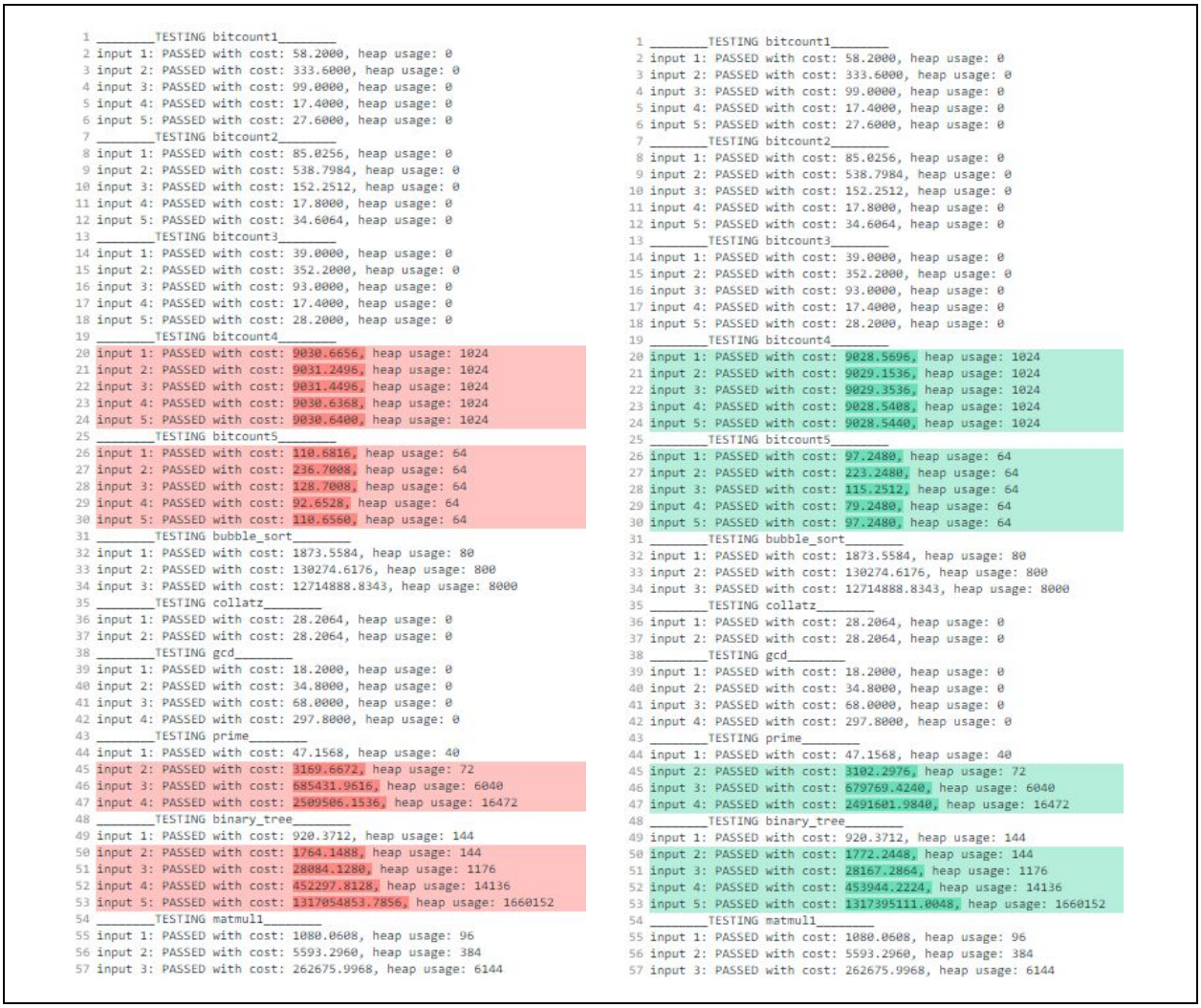
2. Added StackAccess Condition for "reset" instruction

Second, I added stack access conditions to increase the efficient use of "reset" instruction. This 'reset' instruction was implemented by our teammate in the sprint 2. However, we could not track heap/stack access by 100%. While making the pass related to the global variable, I discovered an additional way to track

heap-access(GlobalVariable Load or Store) and added this condition in the AssemblyEmitter.cpp file. This is expected to reduce the cost of head-traveling between stack and heap.

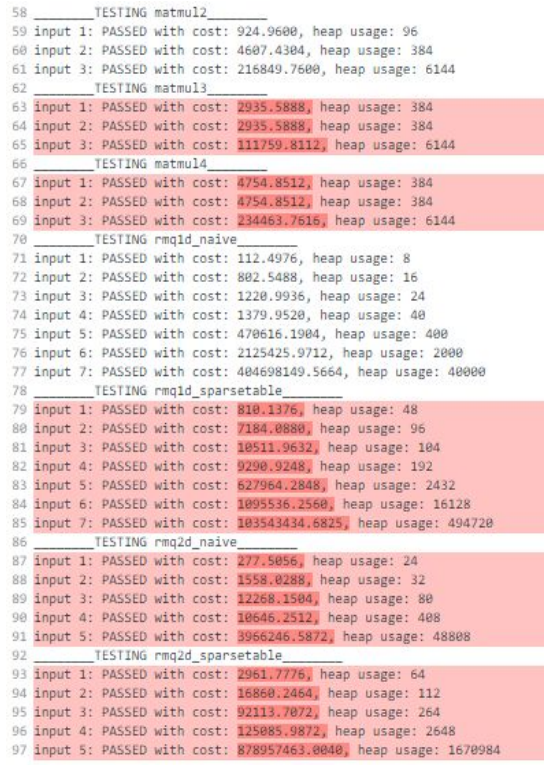
Test Results:

The cost of test cases that were using global variable in the loop were all reduced as you can see from the capture below. Also, some cases which used global variables but not in the loops were also reduced because of “reset” instruction that was added before its load/store instructions. You can see the increase in the some inputs of rmq2d_separatable. After analyzing the assembly code, I found out that that is because when the global variable’s size is 32, using it directly without loading again will cause some additional bitcast operations. However, since in most cases, cost will be reduced, we decided to keep this optimization.



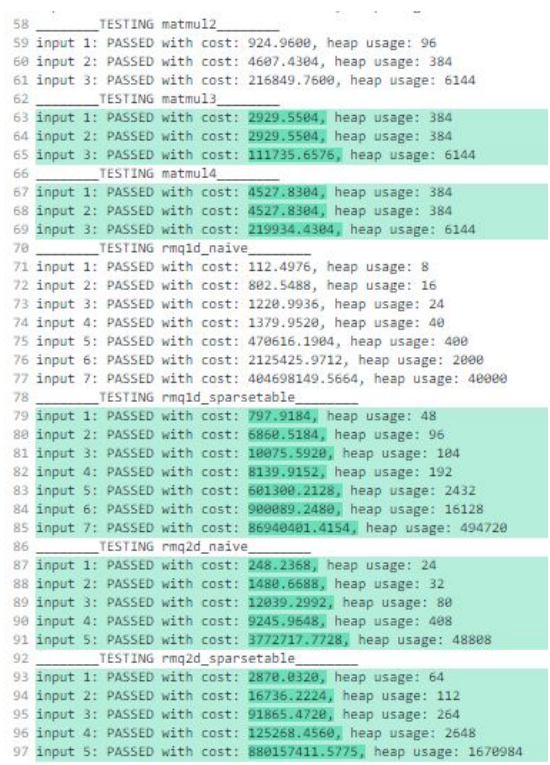
```
1  ________TESTING bitcount1________
2  input 1: PASSED with cost: 58.2000, heap usage: 0
3  input 2: PASSED with cost: 333.6000, heap usage: 0
4  input 3: PASSED with cost: 99.0000, heap usage: 0
5  input 4: PASSED with cost: 17.4000, heap usage: 0
6  input 5: PASSED with cost: 27.6000, heap usage: 0
7  ________TESTING bitcount2________
8  input 1: PASSED with cost: 85.0256, heap usage: 0
9  input 2: PASSED with cost: 538.7984, heap usage: 0
10 input 3: PASSED with cost: 152.2512, heap usage: 0
11 input 4: PASSED with cost: 17.8000, heap usage: 0
12 input 5: PASSED with cost: 34.6064, heap usage: 0
13 ________TESTING bitcount3________
14 input 1: PASSED with cost: 39.0000, heap usage: 0
15 input 2: PASSED with cost: 352.2000, heap usage: 0
16 input 3: PASSED with cost: 93.0000, heap usage: 0
17 input 4: PASSED with cost: 17.4000, heap usage: 0
18 input 5: PASSED with cost: 28.2000, heap usage: 0
19 ________TESTING bitcount4________
20 input 1: PASSED with cost: 9030.6656, heap usage: 1024
21 input 2: PASSED with cost: 9031.2496, heap usage: 1024
22 input 3: PASSED with cost: 9031.4496, heap usage: 1024
23 input 4: PASSED with cost: 9030.6368, heap usage: 1024
24 input 5: PASSED with cost: 9030.6400, heap usage: 1024
25 ________TESTING bitcount5________
26 input 1: PASSED with cost: 110.6816, heap usage: 64
27 input 2: PASSED with cost: 236.7008, heap usage: 64
28 input 3: PASSED with cost: 128.7008, heap usage: 64
29 input 4: PASSED with cost: 92.6528, heap usage: 64
30 input 5: PASSED with cost: 110.6560, heap usage: 64
31 ________TESTING bubble_sort________
32 input 1: PASSED with cost: 1873.5584, heap usage: 80
33 input 2: PASSED with cost: 130274.6176, heap usage: 800
34 input 3: PASSED with cost: 12714888.8343, heap usage: 8000
35 ________TESTING collatz________
36 input 1: PASSED with cost: 28.2064, heap usage: 0
37 input 2: PASSED with cost: 28.2064, heap usage: 0
38 ________TESTING gcd________
39 input 1: PASSED with cost: 18.2000, heap usage: 0
40 input 2: PASSED with cost: 34.8000, heap usage: 0
41 input 3: PASSED with cost: 68.0000, heap usage: 0
42 input 4: PASSED with cost: 297.8000, heap usage: 0
43 ________TESTING prime________
44 input 1: PASSED with cost: 47.1568, heap usage: 40
45 input 2: PASSED with cost: 3169.6672, heap usage: 72
46 input 3: PASSED with cost: 685431.9616, heap usage: 6040
47 input 4: PASSED with cost: 2509506.1536, heap usage: 16472
48 ________TESTING binary_tree________
49 input 1: PASSED with cost: 920.3712, heap usage: 144
50 input 2: PASSED with cost: 1764.1488, heap usage: 144
51 input 3: PASSED with cost: 28084.1280, heap usage: 1176
52 input 4: PASSED with cost: 452297.8128, heap usage: 14136
53 input 5: PASSED with cost: 1317054853.7856, heap usage: 1660152
54 ________TESTING matmul1________
55 input 1: PASSED with cost: 1080.0608, heap usage: 96
56 input 2: PASSED with cost: 5593.2960, heap usage: 384
57 input 3: PASSED with cost: 262675.9968, heap usage: 6144
```

```
1  ________TESTING bitcount1________
2  input 1: PASSED with cost: 58.2000, heap usage: 0
3  input 2: PASSED with cost: 333.6000, heap usage: 0
4  input 3: PASSED with cost: 99.0000, heap usage: 0
5  input 4: PASSED with cost: 17.4000, heap usage: 0
6  input 5: PASSED with cost: 27.6000, heap usage: 0
7  ________TESTING bitcount2________
8  input 1: PASSED with cost: 85.0256, heap usage: 0
9  input 2: PASSED with cost: 538.7984, heap usage: 0
10 input 3: PASSED with cost: 152.2512, heap usage: 0
11 input 4: PASSED with cost: 17.8000, heap usage: 0
12 input 5: PASSED with cost: 34.6064, heap usage: 0
13 ________TESTING bitcount3________
14 input 1: PASSED with cost: 39.0000, heap usage: 0
15 input 2: PASSED with cost: 352.2000, heap usage: 0
16 input 3: PASSED with cost: 93.0000, heap usage: 0
17 input 4: PASSED with cost: 17.4000, heap usage: 0
18 input 5: PASSED with cost: 28.2000, heap usage: 0
19 ________TESTING bitcount4________
20 input 1: PASSED with cost: 9028.5696, heap usage: 1024
21 input 2: PASSED with cost: 9029.1536, heap usage: 1024
22 input 3: PASSED with cost: 9029.3536, heap usage: 1024
23 input 4: PASSED with cost: 9028.5408, heap usage: 1024
24 input 5: PASSED with cost: 9028.5440, heap usage: 1024
25 ________TESTING bitcount5________
26 input 1: PASSED with cost: 97.2480, heap usage: 64
27 input 2: PASSED with cost: 223.2480, heap usage: 64
28 input 3: PASSED with cost: 115.2512, heap usage: 64
29 input 4: PASSED with cost: 79.2480, heap usage: 64
30 input 5: PASSED with cost: 97.2480, heap usage: 64
31 ________TESTING bubble_sort________
32 input 1: PASSED with cost: 1873.5584, heap usage: 80
33 input 2: PASSED with cost: 130274.6176, heap usage: 800
34 input 3: PASSED with cost: 12714888.8343, heap usage: 8000
35 ________TESTING collatz________
36 input 1: PASSED with cost: 28.2064, heap usage: 0
37 input 2: PASSED with cost: 28.2064, heap usage: 0
38 ________TESTING gcd________
39 input 1: PASSED with cost: 18.2000, heap usage: 0
40 input 2: PASSED with cost: 34.8000, heap usage: 0
41 input 3: PASSED with cost: 68.0000, heap usage: 0
42 input 4: PASSED with cost: 297.8000, heap usage: 0
43 ________TESTING prime________
44 input 1: PASSED with cost: 47.1568, heap usage: 40
45 input 2: PASSED with cost: 3102.2976, heap usage: 72
46 input 3: PASSED with cost: 679769.4240, heap usage: 6040
47 input 4: PASSED with cost: 2491601.9840, heap usage: 16472
48 ________TESTING binary_tree________
49 input 1: PASSED with cost: 920.3712, heap usage: 144
50 input 2: PASSED with cost: 1772.2448, heap usage: 144
51 input 3: PASSED with cost: 28167.2864, heap usage: 1176
52 input 4: PASSED with cost: 453944.2224, heap usage: 14136
53 input 5: PASSED with cost: 1317395111.0048, heap usage: 1660152
54 ________TESTING matmul1________
55 input 1: PASSED with cost: 1080.0608, heap usage: 96
56 input 2: PASSED with cost: 5593.2960, heap usage: 384
57 input 3: PASSED with cost: 262675.9968, heap usage: 6144
```



```
_______TESTING matmul2_______
input 1: PASSED with cost: 924.9600, heap usage: 96
input 2: PASSED with cost: 4607.4304, heap usage: 384
input 3: PASSED with cost: 216849.7600, heap usage: 6144
_______TESTING matmul3_______
input 1: PASSED with cost: 2935.5888, heap usage: 384
input 2: PASSED with cost: 2935.5888, heap usage: 384
input 3: PASSED with cost: 111759.8112, heap usage: 6144
_______TESTING matmul4_______
input 1: PASSED with cost: 4754.8512, heap usage: 384
input 2: PASSED with cost: 4754.8512, heap usage: 384
input 3: PASSED with cost: 234463.7616, heap usage: 6144
_______TESTING rmq1d_naive_______
input 1: PASSED with cost: 112.4976, heap usage: 8
input 2: PASSED with cost: 802.5488, heap usage: 16
input 3: PASSED with cost: 1220.9936, heap usage: 24
input 4: PASSED with cost: 1379.9520, heap usage: 40
input 5: PASSED with cost: 470616.1904, heap usage: 400
input 6: PASSED with cost: 2125425.9712, heap usage: 2000
input 7: PASSED with cost: 404698149.5664, heap usage: 40000
_______TESTING rmq1d_sparsetable_______
input 1: PASSED with cost: 810.1376, heap usage: 48
input 2: PASSED with cost: 7184.0880, heap usage: 96
input 3: PASSED with cost: 10511.9632, heap usage: 104
input 4: PASSED with cost: 9290.9248, heap usage: 192
input 5: PASSED with cost: 627964.2848, heap usage: 2432
input 6: PASSED with cost: 1095536.2560, heap usage: 16128
input 7: PASSED with cost: 103543434.6825, heap usage: 494720
_______TESTING rmq2d_naive_______
input 1: PASSED with cost: 277.5056, heap usage: 24
input 2: PASSED with cost: 1558.0288, heap usage: 32
input 3: PASSED with cost: 12268.1504, heap usage: 80
input 4: PASSED with cost: 10646.2512, heap usage: 408
input 5: PASSED with cost: 3966246.5872, heap usage: 48808
_______TESTING rmq2d_sparsetable_______
input 1: PASSED with cost: 2961.7776, heap usage: 64
input 2: PASSED with cost: 16860.2464, heap usage: 112
input 3: PASSED with cost: 92113.7072, heap usage: 264
input 4: PASSED with cost: 125085.9872, heap usage: 2648
input 5: PASSED with cost: 878957463.0040, heap usage: 1670984
```

<before optimization>



```
_______TESTING matmul2_______
input 1: PASSED with cost: 924.9600, heap usage: 96
input 2: PASSED with cost: 4607.4304, heap usage: 384
input 3: PASSED with cost: 216849.7600, heap usage: 6144
_______TESTING matmul3_______
input 1: PASSED with cost: 2929.5504, heap usage: 384
input 2: PASSED with cost: 2929.5504, heap usage: 384
input 3: PASSED with cost: 111735.6576, heap usage: 6144
_______TESTING matmul4_______
input 1: PASSED with cost: 4527.8304, heap usage: 384
input 2: PASSED with cost: 4527.8304, heap usage: 384
input 3: PASSED with cost: 219934.4304, heap usage: 6144
_______TESTING rmq1d_naive_______
input 1: PASSED with cost: 112.4976, heap usage: 8
input 2: PASSED with cost: 802.5488, heap usage: 16
input 3: PASSED with cost: 1220.9936, heap usage: 24
input 4: PASSED with cost: 1379.9520, heap usage: 40
input 5: PASSED with cost: 470616.1904, heap usage: 400
input 6: PASSED with cost: 2125425.9712, heap usage: 2000
input 7: PASSED with cost: 404698149.5664, heap usage: 40000
_______TESTING rmq1d_sparsetable_______
input 1: PASSED with cost: 797.9184, heap usage: 48
input 2: PASSED with cost: 6860.5184, heap usage: 96
input 3: PASSED with cost: 10075.5920, heap usage: 104
input 4: PASSED with cost: 8139.9152, heap usage: 192
input 5: PASSED with cost: 601300.2128, heap usage: 2432
input 6: PASSED with cost: 900089.2480, heap usage: 16128
input 7: PASSED with cost: 86940401.4154, heap usage: 494720
_______TESTING rmq2d_naive_______
input 1: PASSED with cost: 248.2368, heap usage: 24
input 2: PASSED with cost: 1480.6688, heap usage: 32
input 3: PASSED with cost: 12039.2992, heap usage: 80
input 4: PASSED with cost: 9245.9648, heap usage: 408
input 5: PASSED with cost: 3772717.7728, heap usage: 48808
_______TESTING rmq2d_sparsetable_______
input 1: PASSED with cost: 2870.0320, heap usage: 64
input 2: PASSED with cost: 16736.2224, heap usage: 112
input 3: PASSED with cost: 91865.4720, heap usage: 264
input 4: PASSED with cost: 125268.4560, heap usage: 2648
input 5: PASSED with cost: 880157411.5775, heap usage: 1670984
```

<after optimization>

- New Test Cases: input.ll files are provided in the git repository

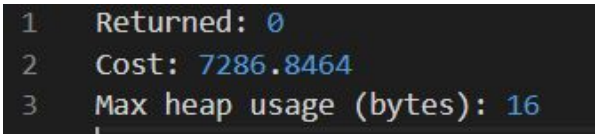
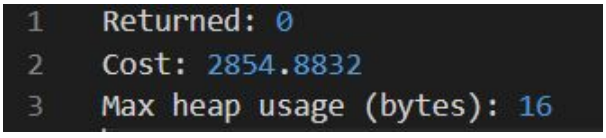
FileCheck 1	
Before Optimization	After Optimization

`Returned: 0`
`Cost: 3743.2048`
`Max heap usage (bytes): 8016`
	`Returned: 0`
	`Cost: 2307.5264`
	`Max heap usage (bytes): 8016`

FileCheck 2	
Before Optimization	After Optimization

`Returned: 0`
`Cost: 12021.2368`
`Max heap usage (bytes): 1624`
	`Returned: 0`
	`Cost: 9557.8048`
	`Max heap usage (bytes): 1624`

FileCheck 3	

Before Optimization	After Optimization
	

Progress Report: Jaeeun Lee

[Sprint 3] Function Inlining Pass

Change in Schedules (None)

Things were accomplished as planned in the previous sprint.

Dones and Not Dones

Dones:

Function inlining is a classic technique in program optimization in which a function call is simply replaced with an actual function body. The purpose of this optimization is to reduce function call overhead and to preserve program state, that is, stack and register status, which naturally lead to further overhead reduction.

I needed to be aware of the fact that in sprint 1 and sprint 2, there is a function outlining pass that outlines blocks if the function uses a lot of registers. To make both function inlining and outlining cost-effective, I previously planned and thought it would be right to run the inlining pass before running the outline pass and also trying to target cases only when inlining is cheaper. However, it turns out actually that inlining before outlining reduces costs in more cases. While changing the orders from running the inline pass after outline pass, I could reduce 4 more test cases. So I decided to run the inling pass after the outlining pass.

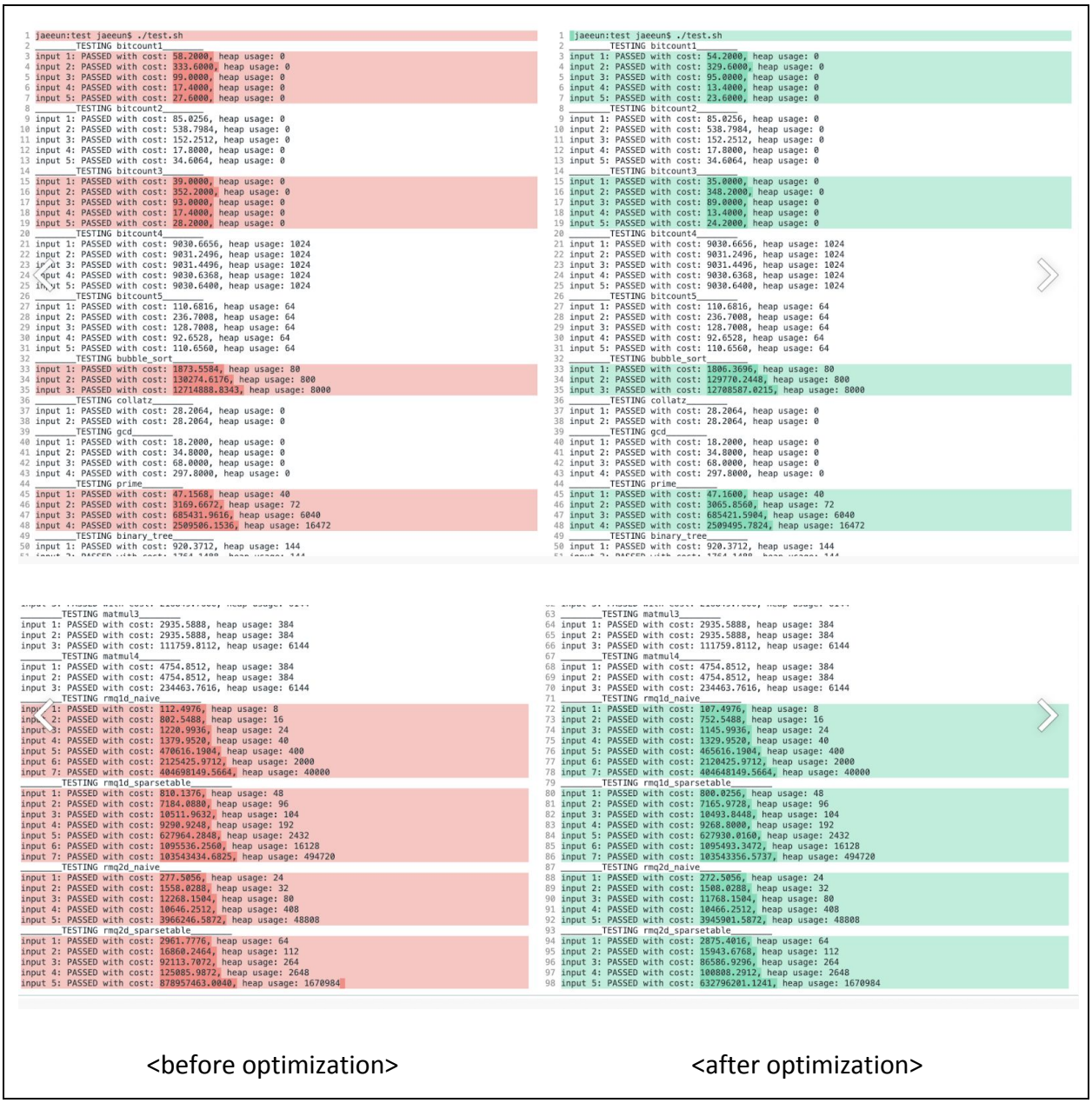
While there is an existing Function Inlining pass, I decided to implement my own, only targeting cases where the function is ‘small enough’ to inline. And by meaning 'small enough' it means if the function is declaring less than 14 registers. Also, I made sure that after inlining the targeted functions, I erased useless functions that are not called anymore. However, since the pass is running after the outlining pass, I did not erase the functions that were outlined & then inlined again, due to safety issues

Not Dones:

Originally, I did try to inline recursively, for example, if there is a function call in another function call, I tried to inline this as well, however I found out that there are many cases that this might actually be quite dangerous & also they increased the costs for many cases, so I did not implement this part on purpose.

Test Results:

- Existing Test Cases



<before optimization> <after optimization>

- New Test Cases : input .ll files are in the git repository

FileCheck 1	
Before Optimization	After Optimization

Returned: 0 Cost: 5225.9296 Max heap usage (bytes): 400	Returned: 0 Cost: 5028.6208 Max heap usage (bytes): 400
FileCheck 2	
Before Optimization	After Optimization
Returned: 0 Cost: 5292.8032 Max heap usage (bytes): 400	Returned: 0 Cost: 5091.0144 Max heap usage (bytes): 400
FileCheck 3	
Before Optimization	After Optimization
Returned: 0 Cost: 5288.8032 Max heap usage (bytes): 400	Returned: 0 Cost: 5090.5872 Max heap usage (bytes): 400