<Progress Report - Sprint 2>

SWPP 2020 Spring LLVM Project
2020.05.30

Team 4
Hyoung Uk Sul
Ahyoung Oh
Jaewook Lee
Jaeeun Lee

Progress Report: Hyoung Uk Sul

[Sprint 2] SimpleBackend Register Allocation Optimization Part 2

Change in Schedules (None)

Things were accomplished as planned in the previous sprint.

Done and Not Dones

Dones: This sprint's implementation of DepromoteRegisters class expedites improvements made in the previous sprint. While the attainment of sprint 1 more leans towards establishing a system and infrastructure for efficient register ID assignment, this implementation expands on this basis to maximize the efficiency with regard to register allocation, leading to decrease in overall memory instructions, and hence decrease in total cost of a run.

The following explains how this implementation expands on to developments made in the last sprint to increase the efficiency of register allocation.

1. In the previous sprint, a phi node was always assigned a temporary register, and hence its use always required a load and a store. However, a phi is often associated with a loop; therefore, there occurs big overhead if phi nodes always move between stack and registers. Hence, some phi nodes are now given permanent registers. They are stored by mul 1, and used as if it is any other ordinary LLVM value. If a phi node uses a constant value, however, it is still given a temporary register since LLVM IR does not allow assigning constant value to a register.
2. There exist instructions that require temporary registers amidst its operations. For instance, a SExt instruction would require that it goes through multiple arithmetic operations (and, mul, or) in the backend to realize IR sign extension. However, these temporary registers are not used after calculations are done. Hence, such temporary registers now use the same register ID as the resulting value, and do not incur eviction of any other values that are on registers.
3. If an instruction uses an operand that is already saved on a register, and this operand is never used again after this instruction, then this operand no longer needs to take up its register space.

Therefore, an instruction will take its operand's register ID if the operand is used only once in this instruction.

4. If a value is used within a loop, then there is more chance that it will be used more often than values outside any loop. Hence, if a value is within a loop, it is more likely to become a permanent register user. This is implemented by giving them extra "priority points."
5. For compatibility between assembly emitter and LLVM IR, there are multiple truncate(after_trunc)/i1(_to_i1) instructions created during depromotions. However, these instructions are completely ignored in the emitter. Thereby, it does not matter what register IDs they are allocated to. Unlike how it was in the previous sprint, they will no longer evict another register for itself; rather, it will be given just a random register ID (r1, r2, ...) they can use.
6. Bitcasts do nothing in assembly emitter; they are simply merged down into multiplication by 1. However, the current emitter is kind enough to ignore bitcasts that use the same register ID as its operand. Hence, if a cast instruction is just used once right after and never more, it is given the name of its operand so that assembly emitter ignores it.
7. After everything is well-depromoted, the implementation from sprint 1 goes through the entire module to resolve any dependency issues introduced during depromotion. Although this incurs more cost with regard to register allocation, it is necessary because the simple LRU policy that is used here cannot approximate every possible CFG structure in input IR. However, such resolution brings unnecessary memory management instructions such as double loads, no load after store, store right after load, and much more. Hence, this implementation goes through a bit more operations after resolving dependency issues to clear up such unnecessary memory instructions.

Not Dones: Although, up to this point, many possible optimizations were successfully accomplished, the depromotion algorithm still has some bits to improve on. Furthermore, because optimization from part 1 and 2 mostly rely on heuristics in the intention of speeding up compiled programs, they are prone to errors from corner cases. Even though the implementation successfully passes benchmarks and test cases, there might still exist some corner cases that this fails to catch.

Specifically, in the next sprint, what should be explored is the possibility of reducing costs by preventing depromotion algorithm from creating unnecessary loads and stores, and also reducing unsafety in unknown corner cases. In order to do this, the new implementation will have to track every possible flow of control in CFG of a given program to ensure that there does not exist unnecessary allocations and there does exist necessary allocations.

Test Results:

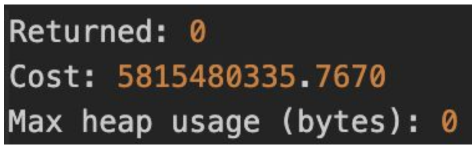
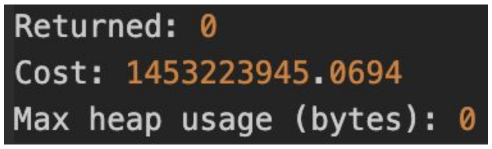
"test.sh", the bash script that I wrote in the last sprint, was updated so that it also prints the cost from sprint 1 and the baseline. Note that the heap usage was omitted because there was no improvement on that department. The following shows a run of *test.sh*. As shown, improvement in register allocation greatly reduced the running cost overall, especially if to be compared with the baseline. Highlighted parts indicate the best improvement cases.

```
StuartSul:swpp202001-team4 Stuart$ ./test.sh compare
_________TESTING bitcount1_________
input 1: PASSED with cost: 58.2000 (sprint1: 75.4608) (baseline: 218.2288)
input 2: PASSED with cost: 333.6000 (sprint1: 443.0064) (baseline: 1351.6672)
input 3: PASSED with cost: 99.0000 (sprint1: 129.9120) (baseline: 386.1456)
input 4: PASSED with cost: 17.4000 (sprint1: 21.0096) (baseline: 50.3120)
input 5: PASSED with cost: 27.6000 (sprint1: 34.6224) (baseline: 92.2912)
_________TESTING bitcount2_________
input 1: PASSED with cost: 85.0256 (sprint1: 86.0256) (baseline: 201.3744)
input 2: PASSED with cost: 538.7984 (sprint1: 539.7984) (baseline: 1256.4480)
input 3: PASSED with cost: 152.2512 (sprint1: 153.2512) (baseline: 357.6816)
input 4: PASSED with cost: 17.8000 (sprint1: 18.8000) (baseline: 45.0672)
input 5: PASSED with cost: 34.6064 (sprint1: 35.6064) (baseline: 84.1440)
_________TESTING bitcount3_________
input 1: PASSED with cost: 39.0000 (sprint1: 49.4352) (baseline: 135.0832)
input 2: PASSED with cost: 352.2000 (sprint1: 461.6064) (baseline: 1364.2656)
input 3: PASSED with cost: 93.0000 (sprint1: 120.4992) (baseline: 347.0112)
input 4: PASSED with cost: 17.4000 (sprint1: 21.0096) (baseline: 50.3120)
input 5: PASSED with cost: 28.2000 (sprint1: 35.2224) (baseline: 92.6976)
_________TESTING bitcount4_________
input 1: PASSED with cost: 9030.6656 (sprint1: 9783.1040) (baseline: 25400.8304)
input 2: PASSED with cost: 9031.2496 (sprint1: 9783.8912) (baseline: 25403.6560)
input 3: PASSED with cost: 9031.4496 (sprint1: 9783.8880) (baseline: 25401.6144)
input 4: PASSED with cost: 9030.6368 (sprint1: 9783.0752) (baseline: 25400.8016)
input 5: PASSED with cost: 9030.6400 (sprint1: 9783.0784) (baseline: 25400.8048)
_________TESTING bitcount5_________
input 1: PASSED with cost: 110.6816 (sprint1: 164.8288) (baseline: 392.6752)
input 2: PASSED with cost: 236.7008 (sprint1: 332.8736) (baseline: 872.3872)
input 3: PASSED with cost: 128.7008 (sprint1: 188.8384) (baseline: 460.9920)
input 4: PASSED with cost: 92.6528 (sprint1: 140.8224) (baseline: 324.4704)
input 5: PASSED with cost: 110.6560 (sprint1: 164.8288) (baseline: 392.6496)
_________TESTING bubble_sort_________
input 1: PASSED with cost: 2045.8624 (sprint1: 3644.9712) (baseline: 6628.9360)
input 2: PASSED with cost: 144217.2576 (sprint1: 315078.2912) (baseline: 562437.2464)
input 3: PASSED with cost: 14216484.4339 (sprint1: 36245394.9259) (baseline: 61987402.0422)
_________TESTING collatz_________
input 1: PASSED with cost: 28.2064 (sprint1: 29.2448) (baseline: 68.4400)
input 2: PASSED with cost: 28.2064 (sprint1: 29.2448) (baseline: 68.4400)
_________TESTING gcd_________
input 1: PASSED with cost: 18.2000 (sprint1: 22.2000) (baseline: 46.5024)
input 2: PASSED with cost: 34.8000 (sprint1: 53.0384) (baseline: 105.4816)
input 3: PASSED with cost: 68.0000 (sprint1: 114.7152) (baseline: 223.4272)
input 4: PASSED with cost: 297.8000 (sprint1: 542.4528) (baseline: 1043.0336)
_________TESTING prime_________
input 1: PASSED with cost: 50.5568 (sprint1: 52.3568) (baseline: 117.4400)
input 2: PASSED with cost: 3189.2656 (sprint1: 3778.3760) (baseline: 6678.1008)
input 3: PASSED with cost: 696570.5616 (sprint1: 902276.9104) (baseline: 1548664.4224)
input 4: PASSED with cost: 2544421.4320 (sprint1: 3309086.8640) (baseline: 5628813.2864)
_________TESTING binary_tree_________
```

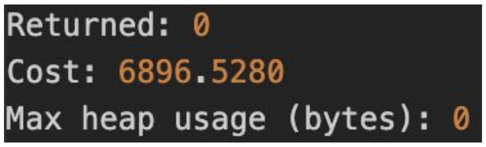
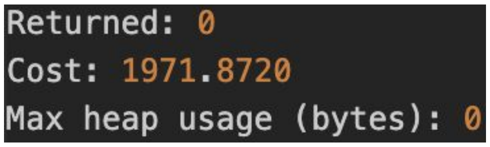
```
input 1: PASSED with cost: 942.0336 (sprint1: 1414.0816) (baseline: 2402.8096)
input 2: PASSED with cost: 1825.7536 (sprint1: 2721.3600) (baseline: 4310.6816)
input 3: PASSED with cost: 28619.0176 (sprint1: 43829.8000) (baseline: 66113.1696)
input 4: PASSED with cost: 461458.1456 (sprint1: 715705.5104) (baseline: 1033485.1248)
input 5: PASSED with cost: 1340779297.0720 (sprint1: 1874833611.0864) (baseline: 1977050335.0849)
```

- New Test Cases:

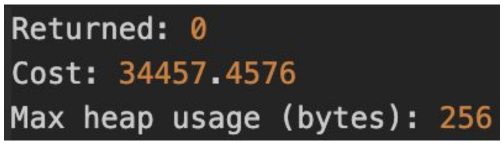
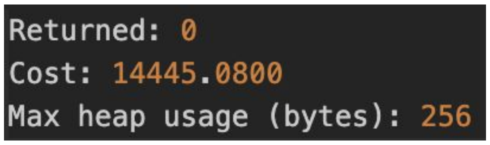
FileCheck 1 (RegisterAllocationOpt-Test4.ll)	
Before Optimization	After Optimization

 `Returned: 0`
`Cost: 5815480335.7670`
`Max heap usage (bytes): 0`
	 `Returned: 0`
	`Cost: 1453223945.0694`
	`Max heap usage (bytes): 0`

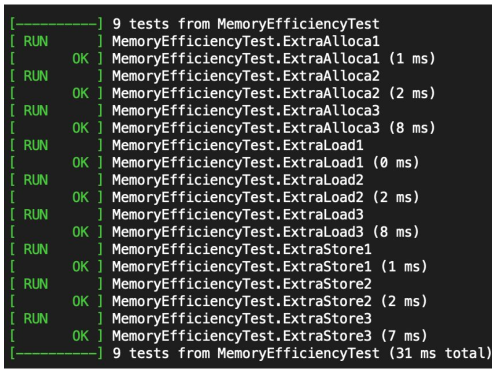
FileCheck 2 (RegisterAllocationOpt-Test5.ll)	
Before Optimization	After Optimization

 `Returned: 0`
`Cost: 6896.5280`
`Max heap usage (bytes): 0`
	 `Returned: 0`
	`Cost: 1971.8720`
	`Max heap usage (bytes): 0`

FileCheck 3 (RegisterAllocationOpt-Test6.ll)	
Before Optimization	After Optimization

 `Returned: 0`
`Cost: 34457.4576`
`Max heap usage (bytes): 256`
	 `Returned: 0`
	`Cost: 14445.0800`
	`Max heap usage (bytes): 256`

GoogleTest (Memory Redundancy Check)	



```
[----------] 9 tests from MemoryEfficiencyTest
[ RUN      ] MemoryEfficiencyTest.ExtraAlloca1
[       OK ] MemoryEfficiencyTest.ExtraAlloca1 (1 ms)
[ RUN      ] MemoryEfficiencyTest.ExtraAlloca2
[       OK ] MemoryEfficiencyTest.ExtraAlloca2 (2 ms)
[ RUN      ] MemoryEfficiencyTest.ExtraAlloca3
[       OK ] MemoryEfficiencyTest.ExtraAlloca3 (8 ms)
[ RUN      ] MemoryEfficiencyTest.ExtraLoad1
[       OK ] MemoryEfficiencyTest.ExtraLoad1 (0 ms)
[ RUN      ] MemoryEfficiencyTest.ExtraLoad2
[       OK ] MemoryEfficiencyTest.ExtraLoad2 (2 ms)
[ RUN      ] MemoryEfficiencyTest.ExtraLoad3
[       OK ] MemoryEfficiencyTest.ExtraLoad3 (8 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 (31 ms total)
```

Progress Report: Ahyoung Oh

[Sprint 2] Reordering Memory Access (Assembly Emitter Memory Access Optimization Part 1)

Change in Schedules (None)

Things were accomplished as planned in the previous sprint.

Done and Not Dones

I've tried a lot of methods to emit reset when the memory access location has changed (either from heap to stack or from stack to heap). Before implementing directly in the AssemblyEmitter I first made a separate cpp file that detects the memory access instructions and tracks the change in the memory access.

However, it was quite complicated to manage the result of the pass in the AssemblyEmitter. So I ended up modifying the AssemblyEmitter.cpp code itself which was the most intuitive and simple. I succeeded in detecting the memory access location changes within the same basic block to be conservative. Thus, in the next sprint, I'm trying to expand the cases so that I could detect the memory access location changes even between the basic blocks, that is, within the same function.

Test Results:

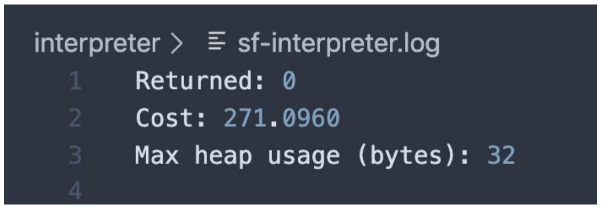
- Existing Test Cases

This optimization decreased cost substantially in the following highlighted cases.

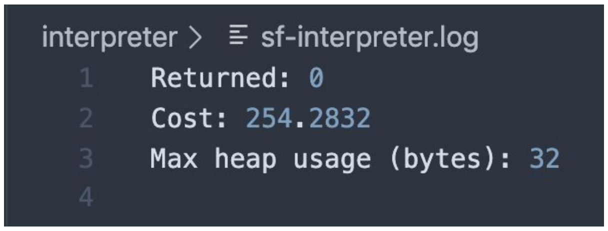
```
________TESTING bitcount1________
input 1: PASSED with cost: 75.4608 (sprint1: 75.4608) (baseline: 218.2288)
input 2: PASSED with cost: 443.0064 (sprint1: 443.0064) (baseline: 1351.6672)
input 3: PASSED with cost: 129.9120 (sprint1: 129.9120) (baseline: 386.1456)
input 4: PASSED with cost: 21.0096 (sprint1: 21.0096) (baseline: 50.3120)
input 5: PASSED with cost: 34.6224 (sprint1: 34.6224) (baseline: 92.2912)
________TESTING bitcount2________
input 1: PASSED with cost: 86.0256 (sprint1: 86.0256) (baseline: 201.3744)
input 2: PASSED with cost: 539.7984 (sprint1: 539.7984) (baseline: 1256.4480)
input 3: PASSED with cost: 153.2512 (sprint1: 153.2512) (baseline: 357.6816)
input 4: PASSED with cost: 18.8000 (sprint1: 18.8000) (baseline: 45.0672)
input 5: PASSED with cost: 35.6064 (sprint1: 35.6064) (baseline: 84.1440)
________TESTING bitcount3________
input 1: PASSED with cost: 49.4352 (sprint1: 49.4352) (baseline: 135.0832)
input 2: PASSED with cost: 461.6064 (sprint1: 461.6064) (baseline: 1364.2656)
input 3: PASSED with cost: 120.4992 (sprint1: 120.4992) (baseline: 347.0112)
input 4: PASSED with cost: 21.0096 (sprint1: 21.0096) (baseline: 50.3120)
input 5: PASSED with cost: 35.2224 (sprint1: 35.2224) (baseline: 92.6976)
________TESTING bitcount4________
input 1: PASSED with cost: 9793.3984 (sprint1: 9783.1040) (baseline: 25400.8304)
input 2: PASSED with cost: 9795.0016 (sprint1: 9783.8912) (baseline: 25403.6560)
input 3: PASSED with cost: 9794.1824 (sprint1: 9783.8880) (baseline: 25401.6144)
input 4: PASSED with cost: 9793.3696 (sprint1: 9783.0752) (baseline: 25400.8016)
input 5: PASSED with cost: 9793.3728 (sprint1: 9783.0784) (baseline: 25400.8048)
________TESTING bitcount5________
input 1: PASSED with cost: 144.2272 (sprint1: 164.8288) (baseline: 392.6752)
input 2: PASSED with cost: 312.2912 (sprint1: 332.8736) (baseline: 872.3872)
input 3: PASSED with cost: 168.2528 (sprint1: 188.8384) (baseline: 460.9920)
input 4: PASSED with cost: 120.1920 (sprint1: 140.8224) (baseline: 324.4704)
input 5: PASSED with cost: 144.2016 (sprint1: 164.8288) (baseline: 392.6496)
________TESTING bubble_sort________
input 1: PASSED with cost: 3602.8432 (sprint1: 3644.9712) (baseline: 6628.9360)
input 2: PASSED with cost: 314642.6112 (sprint1: 315078.2912) (baseline: 562437.2464)
input 3: PASSED with cost: 36239598.1259 (sprint1: 36245394.9259) (baseline: 61987402.0422)
________TESTING collatz________
input 1: PASSED with cost: 29.2448 (sprint1: 29.2448) (baseline: 68.4400)
input 2: PASSED with cost: 29.2448 (sprint1: 29.2448) (baseline: 68.4400)
________TESTING gcd________
input 1: PASSED with cost: 22.2000 (sprint1: 22.2000) (baseline: 46.5024)
input 2: PASSED with cost: 53.0384 (sprint1: 53.0384) (baseline: 105.4816)
input 3: PASSED with cost: 114.7152 (sprint1: 114.7152) (baseline: 223.4272)
input 4: PASSED with cost: 542.4528 (sprint1: 542.4528) (baseline: 1043.0336)
________TESTING prime________
input 1: PASSED with cost: 52.3568 (sprint1: 52.3568) (baseline: 117.4400)
input 2: PASSED with cost: 3769.6912 (sprint1: 3778.3760) (baseline: 6678.1008)
input 3: PASSED with cost: 900200.9104 (sprint1: 902276.9104) (baseline: 1548664.4224)
input 4: PASSED with cost: 3301258.6592 (sprint1: 3309086.8640) (baseline: 5628813.2864)
________TESTING binary_tree________
input 1: PASSED with cost: 1353.6912 (sprint1: 1414.0816) (baseline: 2402.8096)
input 2: PASSED with cost: 2618.0960 (sprint1: 2721.3600) (baseline: 4310.6816)
input 3: PASSED with cost: 42793.2368 (sprint1: 43829.8000) (baseline: 66113.1696)
input 4: PASSED with cost: 694260.5216 (sprint1: 715705.5104) (baseline: 1033485.1248)
input 5: PASSED with cost: 1730206962.6256 (sprint1: 1874833611.0864) (baseline: 1977050335.0849)
```

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

FileCheck 1 (1 reset)	
Before Optimization	After Optimization



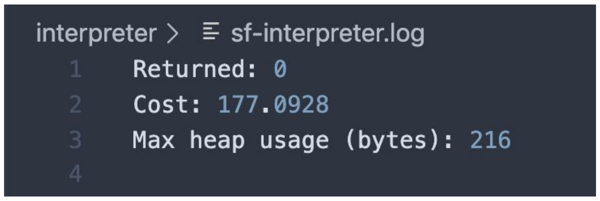
```
Returned: 0
Cost: 271.0960
Max heap usage (bytes): 32
```



```
Returned: 0
Cost: 254.2832
Max heap usage (bytes): 32
```

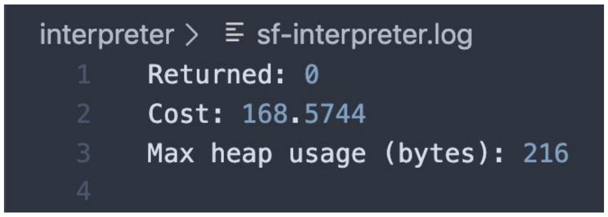
FileCheck 2 (4 resets)

Before Optimization



```
Returned: 0
Cost: 177.0928
Max heap usage (bytes): 216
```

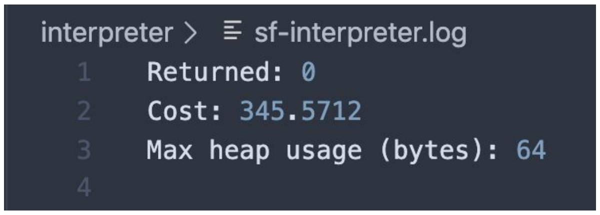
After Optimization



```
Returned: 0
Cost: 168.5744
Max heap usage (bytes): 216
```

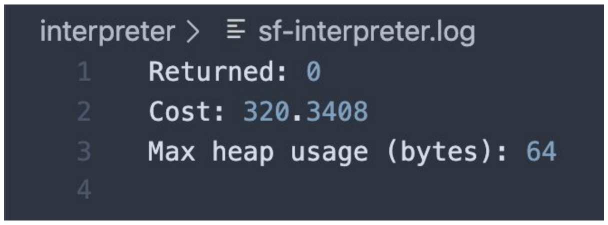
FileCheck 3 (5 resets)

Before Optimization



```
Returned: 0
Cost: 345.5712
Max heap usage (bytes): 64
```

After Optimization



```
Returned: 0
Cost: 320.3408
Max heap usage (bytes): 64
```

Progress Report: Jaewook Lee

In sprint 2, I pushed two pull requests in total : ‘Loop Optimization’ and ‘Malloc to Alloca Conversion Part 2’. I was responsible for importing already-existing passes in this sprint, so I added several existing LLVM passes.

[Sprint 2] Loop Optimization

Change in Schedules : Garbage Collector Pass -> Loop Optimization

The original plan was to implement garbage collector pass as I wrote in Document A after sprint 1. Because this was a new idea, we asked our TA about this new plan. However, we got an answer from him that although it is a good idea, it would be better to implement the passes or optimizations that will reduce the cost of existing benchmark test cases. Therefore, we had to change our plan.

The new plan was to add loop optimization passes. There were many existing LLVM passes that were handling loops, but simply adding all those loops did not show significant reduction in the cost. Actually, some passes increased the cost or sometimes caused some errors. Therefore, I studied each optimization pass and decided which ones to add, and in which order to add.

Dones and Not Dones

Dones: In this optimization, I added several passes related to loop optimization. The passes I checked were as follows : LoopInstSimplifyPass, LoopSimplifyCFGPass, LICMPass, LoopStrengthReducePass, IndVarSimplifyPass, LoopDeletionPass, LoopSimplifyPass, LoopUnrollPass. All the passes except LoopSimplifyPass and LoopUnrollPass were Loop-level passes so I had to declare a new LoopPassManager in order to add them. I studied what each pass was doing and tested whether it is reducing the cost or not, analyzing the reason if the cost increases.

LoopInstSimplifyPass & LoopSimplifyCFGPass : These two passes were simplifying the loop structure and instructions so that the later loop passes can be applied more easily. For example, if the loop part is mixed with non-loop part, it is dividing them into several basic blocks so that they can be divided separately with loop-part and non-loop part. I added these in the beginning of the loop opt. Also, in order to avoid too many basic blocks, we have to add simplifyCFGPass at the end.

LICMPass : This was one of the main pass that reduced the cost of our test cases. This is well-know Loop Invariant Code Motion Pass, and it moves the loop-invariant instructions out of the loop so that it reduces the number of it being executed during the runtime. In order to add this loop, OptimizationRemarkEmitterAnalysis had to be cached before.

LoopDeletionPass : This pass simply erases the loop that has no effect, such as empty loop. This pass reduces the cost the most when it is added after all the loop-related passes are done and simplifyCFG is run, because these passes might make the loop unnecessary or dead. That is why I declared another loop pass manager (LPM2) to add this pass.

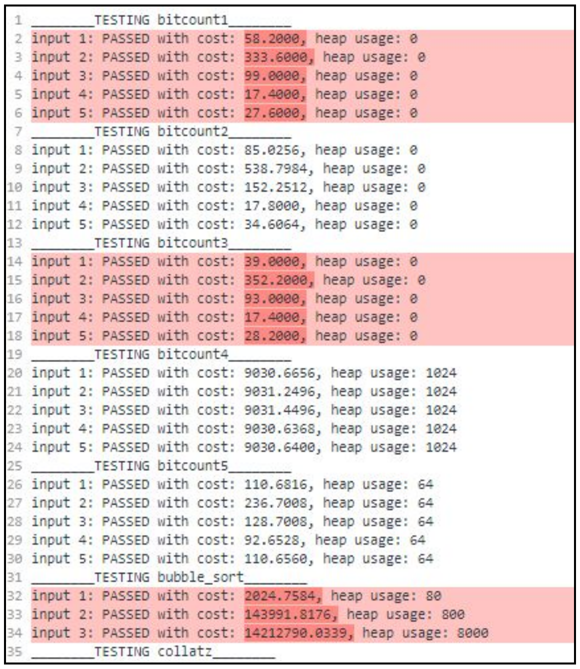
LoopUnrollPass : This pass is function pass because it is observing the whole function before deciding whether to unroll the loop or how to unroll. Adding this pass showed a cost reduction in a test case, so I added it.

Not Dones: I had to give up implementing some passes because it was actually increasing the cost in some cases, or caused unexpected behaviors, or errors. I could not fix the pass itself, so I had to give up adding it. I am planning to make some loop-opt passes by myself so that it can work in our test cases safely.

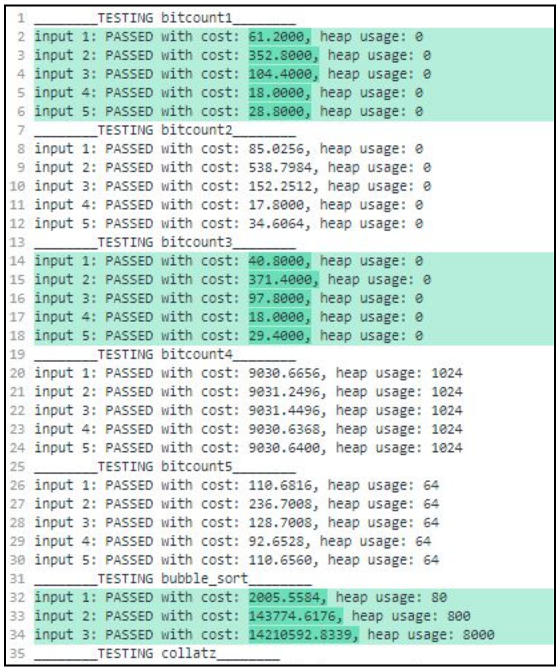
IndVarSimplifyPass : This simplifies the induction variable operation in the loop. For instance, for(i=5;i*i<100;i++) -> for(i=5;i<10;i++). I was planning to add this pass, but after code review and some more experiments, in some cases this pass was giving out unexpected results such as zero-extend to i65 or else. This kind of extension is not allowed in our compiler, so I gave up implementing this.

Test Results:

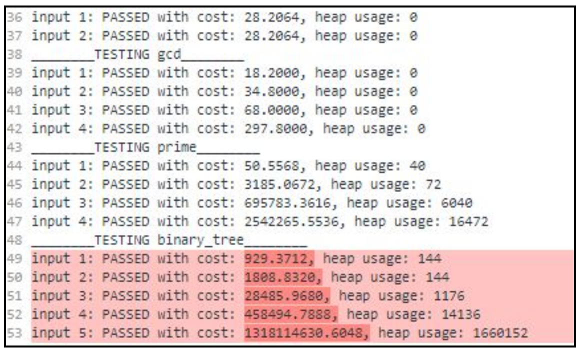
This optimization showed some cost increase in two cases, but this increase occurs because in these two cases, the loop is not repeated a lot of time. Combined with other optimizations such as GVNPass (which I did for my second pull request, explained below) or tail call elimination(which will be implemented in the next sprint) and other simplebackend optimizations that are made by my teammates. It showed the reduction in the cost in some big for-loop cases.



```
________TESTING bitcount1________
input 1: PASSED with cost: 58.2000, heap usage: 0
input 2: PASSED with cost: 333.6000, heap usage: 0
input 3: PASSED with cost: 99.0000, heap usage: 0
input 4: PASSED with cost: 17.4000, heap usage: 0
input 5: PASSED with cost: 27.6000, heap usage: 0
________TESTING bitcount2________
input 1: PASSED with cost: 85.0256, heap usage: 0
input 2: PASSED with cost: 538.7984, heap usage: 0
input 3: PASSED with cost: 152.2512, heap usage: 0
input 4: PASSED with cost: 17.8000, heap usage: 0
input 5: PASSED with cost: 34.6064, heap usage: 0
________TESTING bitcount3________
input 1: PASSED with cost: 39.0000, heap usage: 0
input 2: PASSED with cost: 352.2000, heap usage: 0
input 3: PASSED with cost: 93.0000, heap usage: 0
input 4: PASSED with cost: 17.4000, heap usage: 0
input 5: PASSED with cost: 28.2000, heap usage: 0
________TESTING bitcount4________
input 1: PASSED with cost: 9030.6656, heap usage: 1024
input 2: PASSED with cost: 9031.2496, heap usage: 1024
input 3: PASSED with cost: 9031.4496, heap usage: 1024
input 4: PASSED with cost: 9030.6368, heap usage: 1024
input 5: PASSED with cost: 9030.6400, heap usage: 1024
________TESTING bitcount5________
input 1: PASSED with cost: 110.6816, heap usage: 64
input 2: PASSED with cost: 236.7008, heap usage: 64
input 3: PASSED with cost: 128.7008, heap usage: 64
input 4: PASSED with cost: 92.6528, heap usage: 64
input 5: PASSED with cost: 110.6560, heap usage: 64
________TESTING bubble_sort________
input 1: PASSED with cost: 2024.7584, heap usage: 80
input 2: PASSED with cost: 143991.8176, heap usage: 800
input 3: PASSED with cost: 14212790.0339, heap usage: 8000
________TESTING collatz________
```

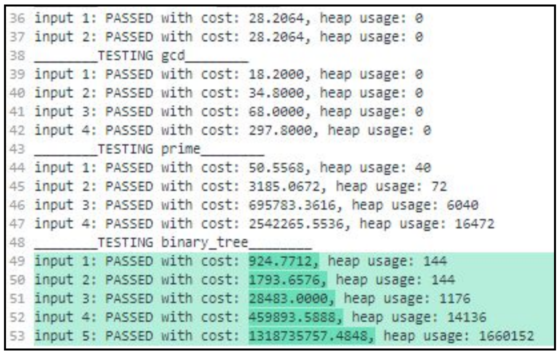


```
________TESTING bitcount1________
input 1: PASSED with cost: 61.2000, heap usage: 0
input 2: PASSED with cost: 352.8000, heap usage: 0
input 3: PASSED with cost: 104.4000, heap usage: 0
input 4: PASSED with cost: 18.0000, heap usage: 0
input 5: PASSED with cost: 28.8000, heap usage: 0
________TESTING bitcount2________
input 1: PASSED with cost: 85.0256, heap usage: 0
input 2: PASSED with cost: 538.7984, heap usage: 0
input 3: PASSED with cost: 152.2512, heap usage: 0
input 4: PASSED with cost: 17.8000, heap usage: 0
input 5: PASSED with cost: 34.6064, heap usage: 0
________TESTING bitcount3________
input 1: PASSED with cost: 40.8000, heap usage: 0
input 2: PASSED with cost: 371.4000, heap usage: 0
input 3: PASSED with cost: 97.8000, heap usage: 0
input 4: PASSED with cost: 18.0000, heap usage: 0
input 5: PASSED with cost: 29.4000, heap usage: 0
________TESTING bitcount4________
input 1: PASSED with cost: 9030.6656, heap usage: 1024
input 2: PASSED with cost: 9031.2496, heap usage: 1024
input 3: PASSED with cost: 9031.4496, heap usage: 1024
input 4: PASSED with cost: 9030.6368, heap usage: 1024
input 5: PASSED with cost: 9030.6400, heap usage: 1024
________TESTING bitcount5________
input 1: PASSED with cost: 110.6816, heap usage: 64
input 2: PASSED with cost: 236.7008, heap usage: 64
input 3: PASSED with cost: 128.7008, heap usage: 64
input 4: PASSED with cost: 92.6528, heap usage: 64
input 5: PASSED with cost: 110.6560, heap usage: 64
________TESTING bubble_sort________
input 1: PASSED with cost: 2005.5584, heap usage: 80
input 2: PASSED with cost: 143774.6176, heap usage: 800
input 3: PASSED with cost: 14210592.8339, heap usage: 8000
________TESTING collatz________
```



```
input 1: PASSED with cost: 28.2064, heap usage: 0
input 2: PASSED with cost: 28.2064, heap usage: 0
________TESTING gcd________
input 1: PASSED with cost: 18.2000, heap usage: 0
input 2: PASSED with cost: 34.8000, heap usage: 0
input 3: PASSED with cost: 68.0000, heap usage: 0
input 4: PASSED with cost: 297.8000, heap usage: 0
________TESTING prime________
input 1: PASSED with cost: 50.5568, heap usage: 40
input 2: PASSED with cost: 3185.0672, heap usage: 72
input 3: PASSED with cost: 695783.3616, heap usage: 6040
input 4: PASSED with cost: 2542265.5536, heap usage: 16472
________TESTING binary_tree________
input 1: PASSED with cost: 929.3712, heap usage: 144
input 2: PASSED with cost: 1808.8320, heap usage: 144
input 3: PASSED with cost: 28485.9680, heap usage: 1176
input 4: PASSED with cost: 458494.7888, heap usage: 14136
input 5: PASSED with cost: 1318114630.6048, heap usage: 1660152
```

<before optimization>

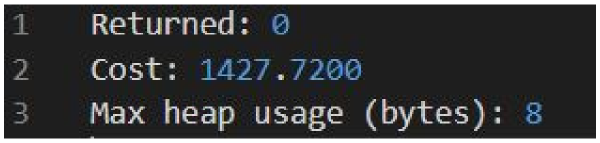
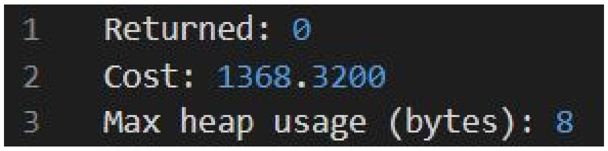


```
input 1: PASSED with cost: 28.2064, heap usage: 0
input 2: PASSED with cost: 28.2064, heap usage: 0
________TESTING gcd________
input 1: PASSED with cost: 18.2000, heap usage: 0
input 2: PASSED with cost: 34.8000, heap usage: 0
input 3: PASSED with cost: 68.0000, heap usage: 0
input 4: PASSED with cost: 297.8000, heap usage: 0
________TESTING prime________
input 1: PASSED with cost: 50.5568, heap usage: 40
input 2: PASSED with cost: 3185.0672, heap usage: 72
input 3: PASSED with cost: 695783.3616, heap usage: 6040
input 4: PASSED with cost: 2542265.5536, heap usage: 16472
________TESTING binary_tree________
input 1: PASSED with cost: 924.7712, heap usage: 144
input 2: PASSED with cost: 1793.6576, heap usage: 144
input 3: PASSED with cost: 28483.0000, heap usage: 1176
input 4: PASSED with cost: 459893.5888, heap usage: 14136
input 5: PASSED with cost: 1318735757.4848, heap usage: 1660152
```

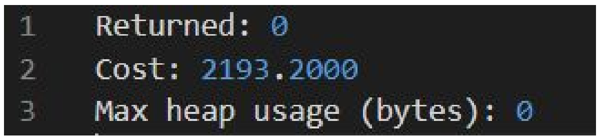
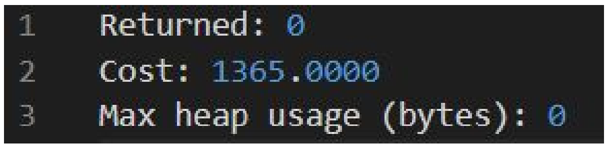
<after optimization>

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

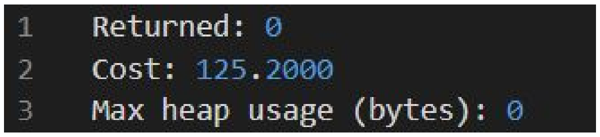
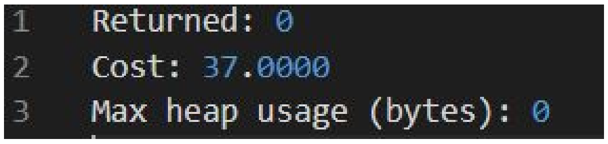
FileCheck 1 (LICM)	
Before Optimization	After Optimization

 `Returned: 0`
`Cost: 1427.7200`
`Max heap usage (bytes): 8`
	 `Returned: 0`
	`Cost: 1368.3200`
	`Max heap usage (bytes): 8`

FileCheck 2 (LoopDeletion)	
Before Optimization	After Optimization

 `Returned: 0`
`Cost: 2193.2000`
`Max heap usage (bytes): 0`
	 `Returned: 0`
	`Cost: 1365.0000`
	`Max heap usage (bytes): 0`

FileCheck 3 (LoopUnroll)	
Before Optimization	After Optimization

 `Returned: 0`
`Cost: 125.2000`
`Max heap usage (bytes): 0`
	 `Returned: 0`
	`Cost: 37.0000`
	`Max heap usage (bytes): 0`

[Sprint 2] Malloc to Alloca Conversion Part 2 : Add GVN Pass

Change in Schedules :

This schedule I did not mention in the previous Document A that was written right after the sprint 1. However, I mentioned that I would have to add the GVN Pass to make Malloc2AllocPass work in more general cases because some unnecessary load/store instructions deter Malloc2AllocPass to convert malloc and remove free although it is possible to convert.

In addition, I tried to expand the Heap -> Stack conversion using the special character of main function, which is that the main function is never called recursively. My original plan was to convert all malloc instructions in the main function to alloca (only if it does not cause stack overflow).

Dones and Not Dones

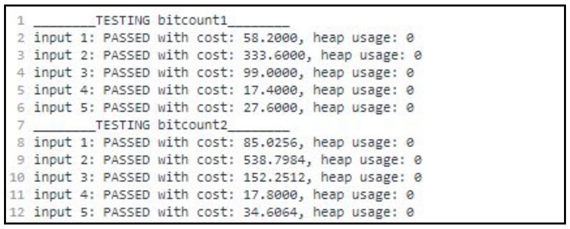
Dones: I added GVN Pass, which is already existing pass. By implementing this pass, my original Malloc2AllocPass is working more widely as you can see from my filetest 1 and 3. Also, this erases many repeated or useless store/load instructions, and makes positive synergy with other optimizations such as loop opt or function outlining, memory reordering.

In addition to this, while making Malloc2AllocinMainPass, I revised some codes in Malloc2AllocPass so that it can be utilized for the use in other passes.

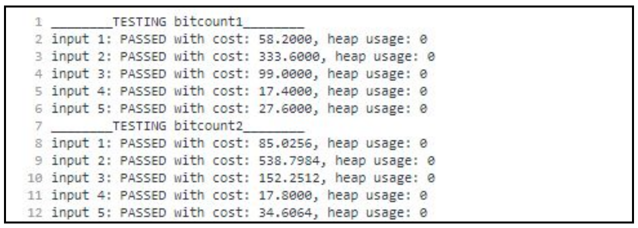
Not Dones: I was not able to add the Malloc2AllocinMainPass, because after the thorough code review, we made a conclusion that this pass was too dangerous to add. This was working on an assumption that all the malloc-ed pointers will be sent to the other function in one of the two ways: by being stored in global variables(such as in case of binary tree or prime) or through argument of the called function. However, we noticed that there are some other cases and had to give up adding this pass into our compiler.

Test Results:

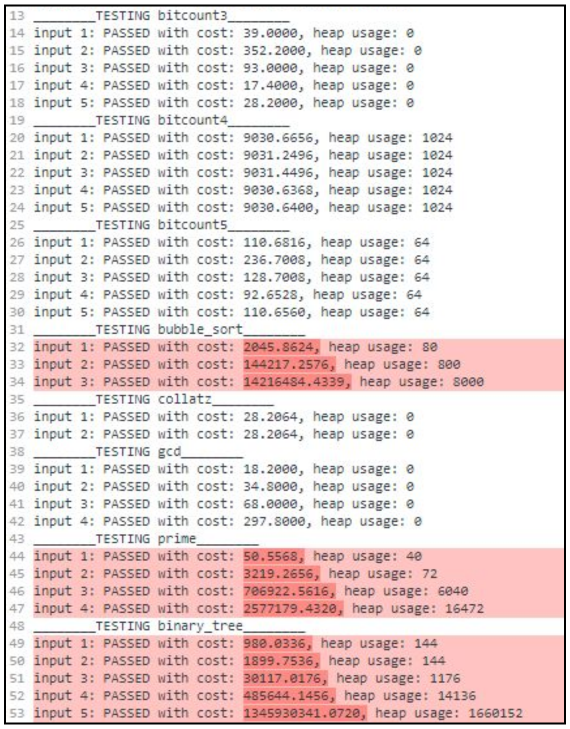
This optimization showed cost reduction in these 3 cases as we can see below.



```
________TESTING bitcount1________
input 1: PASSED with cost: 58.2000, heap usage: 0
input 2: PASSED with cost: 333.6000, heap usage: 0
input 3: PASSED with cost: 99.0000, heap usage: 0
input 4: PASSED with cost: 17.4000, heap usage: 0
input 5: PASSED with cost: 27.6000, heap usage: 0
________TESTING bitcount2________
input 1: PASSED with cost: 85.0256, heap usage: 0
input 2: PASSED with cost: 538.7984, heap usage: 0
input 3: PASSED with cost: 152.2512, heap usage: 0
input 4: PASSED with cost: 17.8000, heap usage: 0
input 5: PASSED with cost: 34.6064, heap usage: 0
```

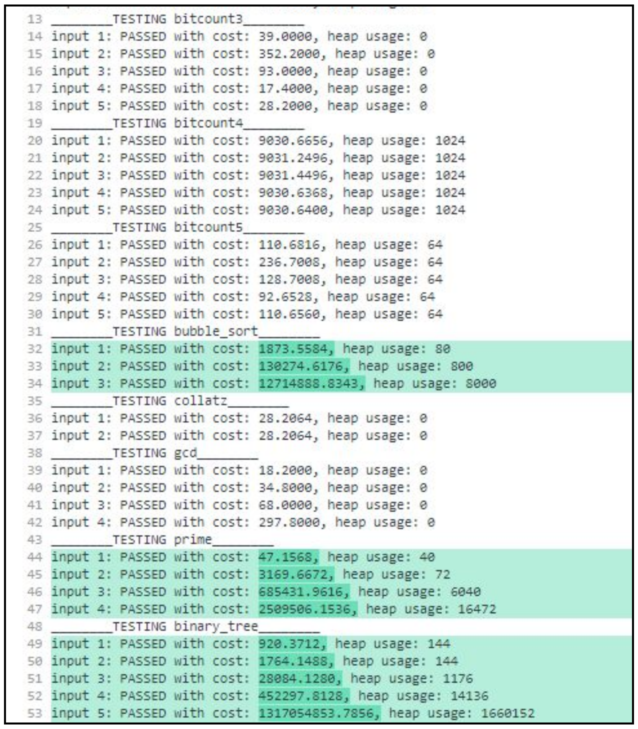


```
________TESTING bitcount1________
input 1: PASSED with cost: 58.2000, heap usage: 0
input 2: PASSED with cost: 333.6000, heap usage: 0
input 3: PASSED with cost: 99.0000, heap usage: 0
input 4: PASSED with cost: 17.4000, heap usage: 0
input 5: PASSED with cost: 27.6000, heap usage: 0
________TESTING bitcount2________
input 1: PASSED with cost: 85.0256, heap usage: 0
input 2: PASSED with cost: 538.7984, heap usage: 0
input 3: PASSED with cost: 152.2512, heap usage: 0
input 4: PASSED with cost: 17.8000, heap usage: 0
input 5: PASSED with cost: 34.6064, heap usage: 0
```



```
________TESTING bitcount3________
input 1: PASSED with cost: 39.0000, heap usage: 0
input 2: PASSED with cost: 352.2000, heap usage: 0
input 3: PASSED with cost: 93.0000, heap usage: 0
input 4: PASSED with cost: 17.4000, heap usage: 0
input 5: PASSED with cost: 28.2000, heap usage: 0
________TESTING bitcount4________
input 1: PASSED with cost: 9030.6656, heap usage: 1024
input 2: PASSED with cost: 9031.2496, heap usage: 1024
input 3: PASSED with cost: 9031.4496, heap usage: 1024
input 4: PASSED with cost: 9030.6368, heap usage: 1024
input 5: PASSED with cost: 9030.6400, heap usage: 1024
________TESTING bitcount5________
input 1: PASSED with cost: 110.6816, heap usage: 64
input 2: PASSED with cost: 236.7008, heap usage: 64
input 3: PASSED with cost: 128.7008, heap usage: 64
input 4: PASSED with cost: 92.6528, heap usage: 64
input 5: PASSED with cost: 110.6560, heap usage: 64
________TESTING bubble_sort________
input 1: PASSED with cost: 2045.8624, heap usage: 80
input 2: PASSED with cost: 144217.2576, heap usage: 800
input 3: PASSED with cost: 14216484.4339, heap usage: 8000
________TESTING collatz________
input 1: PASSED with cost: 28.2064, heap usage: 0
input 2: PASSED with cost: 28.2064, heap usage: 0
________TESTING gcd________
input 1: PASSED with cost: 18.2000, heap usage: 0
input 2: PASSED with cost: 34.8000, heap usage: 0
input 3: PASSED with cost: 68.0000, heap usage: 0
input 4: PASSED with cost: 297.8000, heap usage: 0
________TESTING prime________
input 1: PASSED with cost: 50.5568, heap usage: 40
input 2: PASSED with cost: 3219.2656, heap usage: 72
input 3: PASSED with cost: 706922.5616, heap usage: 6040
input 4: PASSED with cost: 2577179.4320, heap usage: 16472
________TESTING binary_tree________
input 1: PASSED with cost: 980.0336, heap usage: 144
input 2: PASSED with cost: 1899.7536, heap usage: 144
input 3: PASSED with cost: 30117.0176, heap usage: 1176
input 4: PASSED with cost: 485644.1456, heap usage: 14136
input 5: PASSED with cost: 1345930341.0720, heap usage: 1660152
```

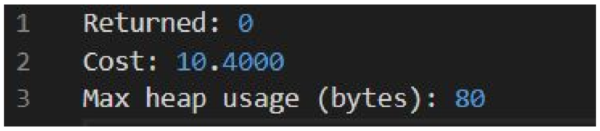
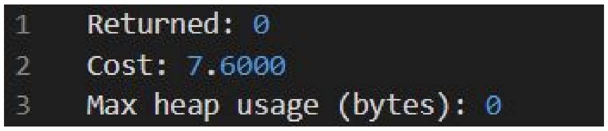
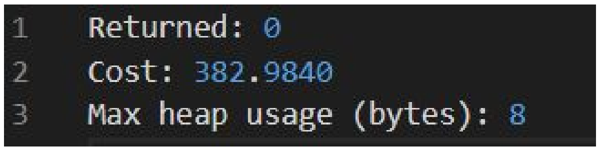
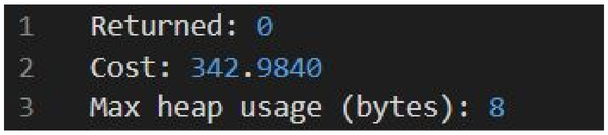
<before optimization>

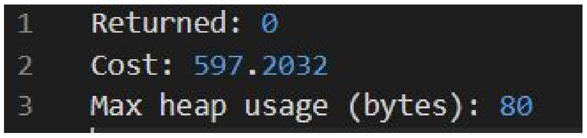
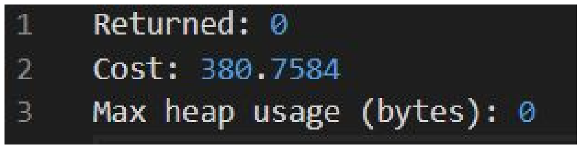


```
________TESTING bitcount3________
input 1: PASSED with cost: 39.0000, heap usage: 0
input 2: PASSED with cost: 352.2000, heap usage: 0
input 3: PASSED with cost: 93.0000, heap usage: 0
input 4: PASSED with cost: 17.4000, heap usage: 0
input 5: PASSED with cost: 28.2000, heap usage: 0
________TESTING bitcount4________
input 1: PASSED with cost: 9030.6656, heap usage: 1024
input 2: PASSED with cost: 9031.2496, heap usage: 1024
input 3: PASSED with cost: 9031.4496, heap usage: 1024
input 4: PASSED with cost: 9030.6368, heap usage: 1024
input 5: PASSED with cost: 9030.6400, heap usage: 1024
________TESTING bitcount5________
input 1: PASSED with cost: 110.6816, heap usage: 64
input 2: PASSED with cost: 236.7008, heap usage: 64
input 3: PASSED with cost: 128.7008, heap usage: 64
input 4: PASSED with cost: 92.6528, heap usage: 64
input 5: PASSED with cost: 110.6560, heap usage: 64
________TESTING bubble_sort________
input 1: PASSED with cost: 1873.5584, heap usage: 80
input 2: PASSED with cost: 130274.6176, heap usage: 800
input 3: PASSED with cost: 12714888.8343, heap usage: 8000
________TESTING collatz________
input 1: PASSED with cost: 28.2064, heap usage: 0
input 2: PASSED with cost: 28.2064, heap usage: 0
________TESTING gcd________
input 1: PASSED with cost: 18.2000, heap usage: 0
input 2: PASSED with cost: 34.8000, heap usage: 0
input 3: PASSED with cost: 68.0000, heap usage: 0
input 4: PASSED with cost: 297.8000, heap usage: 0
________TESTING prime________
input 1: PASSED with cost: 47.1568, heap usage: 40
input 2: PASSED with cost: 3169.6672, heap usage: 72
input 3: PASSED with cost: 685431.9616, heap usage: 6040
input 4: PASSED with cost: 2509506.1536, heap usage: 16472
________TESTING binary_tree________
input 1: PASSED with cost: 920.3712, heap usage: 144
input 2: PASSED with cost: 1764.1488, heap usage: 144
input 3: PASSED with cost: 28084.1280, heap usage: 1176
input 4: PASSED with cost: 452297.8128, heap usage: 14136
input 5: PASSED with cost: 1317054853.7856, heap usage: 1660152
```

<after optimization>

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

FileCheck 1	
Before Optimization	After Optimization
 Returned: 0 Cost: 10.4000 Max heap usage (bytes): 80	 Returned: 0 Cost: 7.6000 Max heap usage (bytes): 0
FileCheck 2	
Before Optimization	After Optimization
 Returned: 0 Cost: 382.9840 Max heap usage (bytes): 8	 Returned: 0 Cost: 342.9840 Max heap usage (bytes): 8
FileCheck 3	

Before Optimization	After Optimization
 ```Returned: 0``` ```Cost: 597.2032``` ```Max heap usage (bytes): 80```	 ```Returned: 0``` ```Cost: 380.7584``` ```Max heap usage (bytes): 0```

Progress Report: Jaeeun Lee

[Sprint 2] Function Outlining Pass Part 2

Change in Schedules (None)

Things were accomplished as planned in the previous sprint.

Dones and Not Dones

Dones: Function Outline Pass Part 2 takes care of more cases where it is efficient to outline a block in a function. While part 1 only took care of the case when we can split a single 'big block' into smaller blocks, part 2 takes care of the extended cases where in a function that uses a lot of registers we can outline whole blocks out into a whole new function.

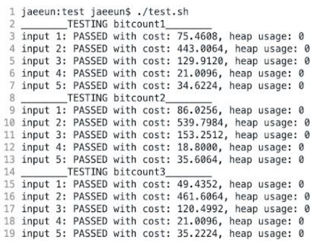
Code Extractor does much of the work of the actual outlining, but in this pass I made sure that the outlined function will not have more than 10 function arguments, as it will cause an error if the outlined function contains more than 16 function args. So, I added the condition where we only break whole blocks into a whole new function by actually counting the number of 'to-be outlined' function arguments and if it exceeds 12, try to break the block into smaller pieces (so that it outlines to a function with less than 12 args) and then outline the smaller blocks.

So, in this pass the function outlining pass can recursively break 'big' blocks in to smaller blocks that (if outlining is possible) and also break next blocks into a whole new function so that we can save the costs of allocas, store, and loads.

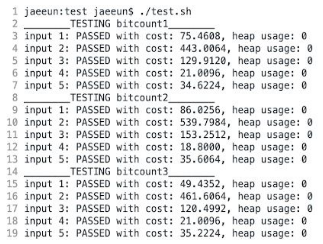
Not Dones: Originally, in the plan I tried to extract 'loops', however I decided this was a redundant process and that it does not really whether a block is a loop body or not, the only thing important is that it should outline blocks that use a lot of regs in the previous block. So this part due to the redundancy, was not implemented.

Test Results:

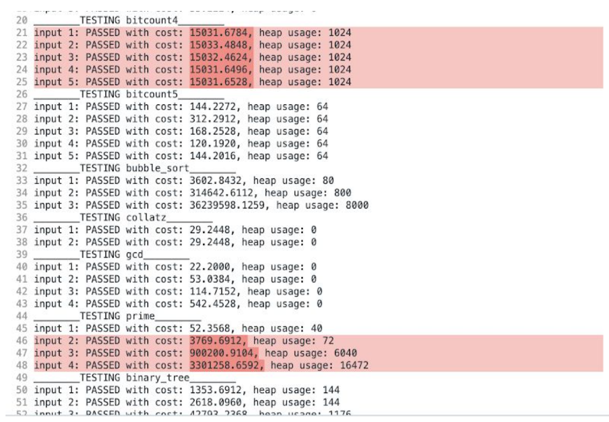
- Existing Test Cases



```
jaeeun:test jaeeun$ ./test.sh
________TESTING bitcount1________
input 1: PASSED with cost: 75.4608, heap usage: 0
input 2: PASSED with cost: 443.0064, heap usage: 0
input 3: PASSED with cost: 129.9120, heap usage: 0
input 4: PASSED with cost: 21.0096, heap usage: 0
input 5: PASSED with cost: 34.6224, heap usage: 0
________TESTING bitcount2________
input 1: PASSED with cost: 86.0256, heap usage: 0
input 2: PASSED with cost: 539.7984, heap usage: 0
input 3: PASSED with cost: 153.2512, heap usage: 0
input 4: PASSED with cost: 18.8000, heap usage: 0
input 5: PASSED with cost: 35.6064, heap usage: 0
________TESTING bitcount3________
input 1: PASSED with cost: 49.4352, heap usage: 0
input 2: PASSED with cost: 461.6064, heap usage: 0
input 3: PASSED with cost: 120.4992, heap usage: 0
input 4: PASSED with cost: 21.0096, heap usage: 0
input 5: PASSED with cost: 35.2224, heap usage: 0
```

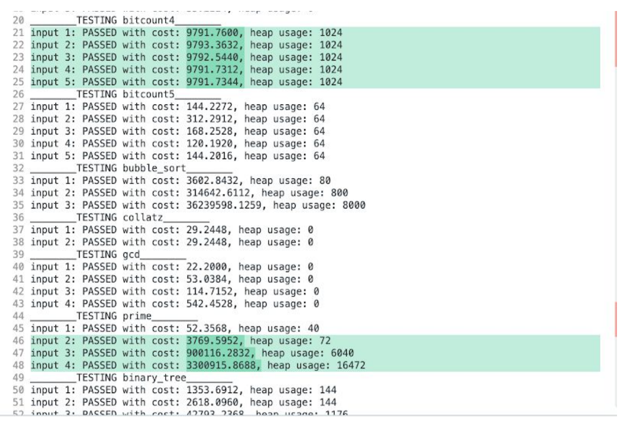


```
jaeeun:test jaeeun$ ./test.sh
________TESTING bitcount1________
input 1: PASSED with cost: 75.4608, heap usage: 0
input 2: PASSED with cost: 443.0064, heap usage: 0
input 3: PASSED with cost: 129.9120, heap usage: 0
input 4: PASSED with cost: 21.0096, heap usage: 0
input 5: PASSED with cost: 34.6224, heap usage: 0
________TESTING bitcount2________
input 1: PASSED with cost: 86.0256, heap usage: 0
input 2: PASSED with cost: 539.7984, heap usage: 0
input 3: PASSED with cost: 153.2512, heap usage: 0
input 4: PASSED with cost: 18.8000, heap usage: 0
input 5: PASSED with cost: 35.6064, heap usage: 0
________TESTING bitcount3________
input 1: PASSED with cost: 49.4352, heap usage: 0
input 2: PASSED with cost: 461.6064, heap usage: 0
input 3: PASSED with cost: 120.4992, heap usage: 0
input 4: PASSED with cost: 21.0096, heap usage: 0
input 5: PASSED with cost: 35.2224, heap usage: 0
```



```
________TESTING bitcount4________
input 1: PASSED with cost: 15031.6784, heap usage: 1024
input 2: PASSED with cost: 15033.4848, heap usage: 1024
input 3: PASSED with cost: 15032.4624, heap usage: 1024
input 4: PASSED with cost: 15031.6496, heap usage: 1024
input 5: PASSED with cost: 15031.6528, heap usage: 1024
________TESTING bitcount5________
input 1: PASSED with cost: 144.2272, heap usage: 64
input 2: PASSED with cost: 312.2912, heap usage: 64
input 3: PASSED with cost: 168.2528, heap usage: 64
input 4: PASSED with cost: 120.1920, heap usage: 64
input 5: PASSED with cost: 144.2016, heap usage: 64
________TESTING bubble_sort________
input 1: PASSED with cost: 3602.8432, heap usage: 80
input 2: PASSED with cost: 314642.6112, heap usage: 800
input 3: PASSED with cost: 36239598.1259, heap usage: 8000
________TESTING collatz________
input 1: PASSED with cost: 29.2448, heap usage: 0
input 2: PASSED with cost: 29.2448, heap usage: 0
________TESTING gcd________
input 1: PASSED with cost: 22.2000, heap usage: 0
input 2: PASSED with cost: 53.0384, heap usage: 0
input 3: PASSED with cost: 114.7152, heap usage: 0
input 4: PASSED with cost: 542.4528, heap usage: 0
________TESTING prime________
input 1: PASSED with cost: 52.3568, heap usage: 40
input 2: PASSED with cost: 3769.6912, heap usage: 72
input 3: PASSED with cost: 900200.9104, heap usage: 6040
input 4: PASSED with cost: 3301258.6592, heap usage: 16472
________TESTING binary_tree________
input 1: PASSED with cost: 1353.6912, heap usage: 144
input 2: PASSED with cost: 2618.0960, heap usage: 144
input 3: PASSED with cost: 42793.2368, heap usage: 1176
```

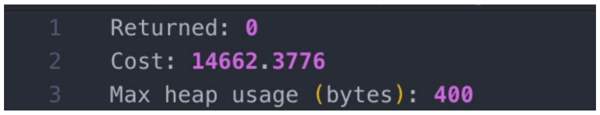
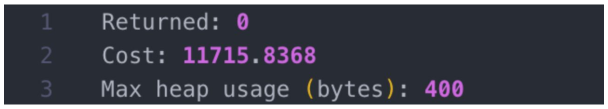
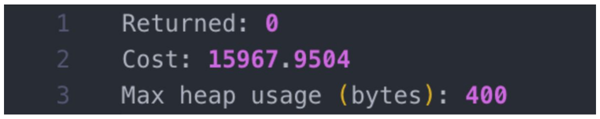
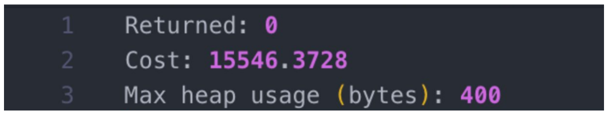
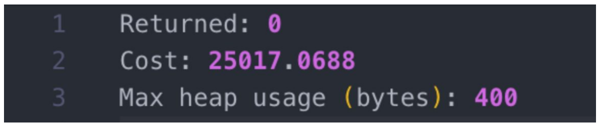
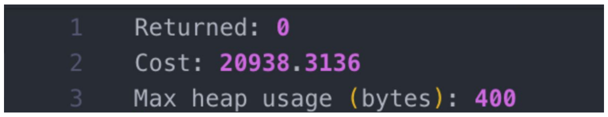
<before optimization>



```
________TESTING bitcount4________
input 1: PASSED with cost: 9791.7600, heap usage: 1024
input 2: PASSED with cost: 9793.3632, heap usage: 1024
input 3: PASSED with cost: 9792.5440, heap usage: 1024
input 4: PASSED with cost: 9791.7312, heap usage: 1024
input 5: PASSED with cost: 9791.7344, heap usage: 1024
________TESTING bitcount5________
input 1: PASSED with cost: 144.2272, heap usage: 64
input 2: PASSED with cost: 312.2912, heap usage: 64
input 3: PASSED with cost: 168.2528, heap usage: 64
input 4: PASSED with cost: 120.1920, heap usage: 64
input 5: PASSED with cost: 144.2016, heap usage: 64
________TESTING bubble_sort________
input 1: PASSED with cost: 3602.8432, heap usage: 80
input 2: PASSED with cost: 314642.6112, heap usage: 800
input 3: PASSED with cost: 36239598.1259, heap usage: 8000
________TESTING collatz________
input 1: PASSED with cost: 29.2448, heap usage: 0
input 2: PASSED with cost: 29.2448, heap usage: 0
________TESTING gcd________
input 1: PASSED with cost: 22.2000, heap usage: 0
input 2: PASSED with cost: 53.0384, heap usage: 0
input 3: PASSED with cost: 114.7152, heap usage: 0
input 4: PASSED with cost: 542.4528, heap usage: 0
________TESTING prime________
input 1: PASSED with cost: 52.3568, heap usage: 40
input 2: PASSED with cost: 3769.5952, heap usage: 72
input 3: PASSED with cost: 900116.2832, heap usage: 6040
input 4: PASSED with cost: 3300915.8688, heap usage: 16472
________TESTING binary_tree________
input 1: PASSED with cost: 1353.6912, heap usage: 144
input 2: PASSED with cost: 2618.0960, heap usage: 144
input 3: PASSED with cost: 42793.2368, heap usage: 1176
```

<after optimization>

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

FileCheck 1	
Before Optimization	After Optimization
 Returned: 0 Cost: 14662.3776 Max heap usage (bytes): 400	 Returned: 0 Cost: 11715.8368 Max heap usage (bytes): 400
FileCheck 2	
Before Optimization	After Optimization
 Returned: 0 Cost: 15967.9504 Max heap usage (bytes): 400	 Returned: 0 Cost: 15546.3728 Max heap usage (bytes): 400
FileCheck 3	
Before Optimization	After Optimization
 Returned: 0 Cost: 25017.0688 Max heap usage (bytes): 400	 Returned: 0 Cost: 20938.3136 Max heap usage (bytes): 400

Appendix: Updated Development Plan (Number indicates the order from document A)

	Sprint 1	Sprint 2	Sprint 3
Hyoung Uk Sul	1	5	10
	SimpleBackend Register Allocation Part 1	SimpleBackend Register Allocation Part 2	SimpleBackend Register Allocation Part 3
Ahyoung Oh	4	6	13
	Arithmetic Optimizations	Reordering Memory Accesses Part 1	Reordering Memory Accesses Part 2
			12
			Tail Call Elimination
Jaewook Lee	3	7	14
	Malloc to Alloca Conversion Part 1	Loop Optimization	Loop Invariant Code Motion Pass : Global Variable Loading
		8	
		Malloc to Alloca Conversion Part 2 : Add GVN Pass	
Jaeeun Lee	2	9	11
	Function Outlining Part 1	Function Outlining Part 2	Function Inlining