<Progress Report - Sprint 1>

SWPP 2020 Spring LLVM Project
2020.05.16

Team 4
Hyoung Uk Sul
Ahyoung Oh
Jaewook Lee
Jaeeun Lee

Progress Report: Hyoung Uk Sul

[Sprint 1] SimpleBackend Register Allocation Optimization

Change in Schedules (Function Outlining -> Register Allocation Optimization)

My original plan for this sprint was to implement function outlining. However, after taking a close examination at the current status quo of the provided backend system, our team has discovered that a significant amount of cost could be reduced by optimizing the backend itself prior to implementing any other optimization passes, especially in its methodology on register allocation. Hence, our team concluded that one person needs to work on the backend this sprint, and I volunteered to be the worker.

Done and Not Dones

Because this is not a pass, but rather an ongoing effort to improve the given system, there are a myriad of things that could be done and were not able to be done. What *is* done, however, are the following:

1. SimpleBackend does not make allocas for every LLVM register anymore.
    a. It will count the number of usages for every instruction value, and frequently used registers will never be stored in stack, but rather permanently stationed on backend registers.
    b. The number of registers that will be stationed as permanent depends on the total number of “temporary” registers needed. Such a number is decided upon what kind of instructions are used within the function, what functions are called, and etc.
2. SimpleBackend does not make stores/loads for every usage of an instruction value.
    a. For “permanently stationed” values, stores and loads are completely needless now.
    b. For “temporary stationed” values, rather than storing after its definition and loading before every usage, they are only stored and loaded when all of the 16 registers are being used.
    c. For this purpose, the backend registers are “registered” to certain values at a time, and are evicted when they are no longer used for a while. Upon eviction, store instructions

are created to store the evicted value to stack, and load instruction is called when they are used again.

   d. Specific policy used for this feature is **LRU (Least Recently Used)**. In other words, the values that were stationed on the backend register and were not used for the longest time are evicted upon register request from new values.
   e. In cases where CFG structure becomes very complex, this could cause dependency issues. Hence, after all instructions are visited and taken care of, every case of eviction is checked once again to ensure that there exists no dependency issues. If there exists an issue, additional store and load instructions are inserted.

3. SimpleBackend does not use the default LLVM Instruction Visitor Pattern anymore.
   a. The original SimpleBackend uses the default LLVM instruction Visitor Pattern, which visits every basic block in the order they are presented in the code, regardless of their logical order.
   b. This could cause following problems:
      i. The resulting assembly and running cost will differ based on the order of blocks given in the input LLVM IR, even if their logical order is the same.
      ii. The new implementation explained above relies heavily on the visiting order of InstVistor, and this could raise dependency issues of registers.
   c. Hence, instead of directly calling visit(Module) function from run(), visitModule(Module) function is called first. Within the visitModule(), visitFunction() is called in order. Visiting order is the same up to this point. However, in visitFunction(), visit(BasicBlock) is called in their logical order; specifically, in their BFS order. This ensures that within each function, the entire process occurs in the same manner regardless of their presence order in LLVM IR.

There are also many things that were not accomplished. Some of such are the following:

1. If there was alloca instruction inserted due to occurrence of a temporary-registered value, but no stores to allocated stack were made because there was sufficient share of registers, the alloca instruction can be deleted safely.
2. If there was a store instruction inserted due to eviction of a temporary-registered value, but such value is never loaded after, then the store instruction can be removed.
3. Upon deciding what values to permanently station to backend registers, if there is a tie on the number of usages, then the ones that get to be stationed must be such that minimizes stores and loads due to the nature of memory access order. The current implementation does not consider this criterion.
4. The bitcast instruction seemed to cause unnecessary cost because what it does in the final assembly output is adding mul by 1. Some time needs to be spent investing on this issue.

Test Results:

During this sprint, I made a simple bash script named “test.sh” which tests inputs and outputs for every given test case, and outputs whether they gave correct outputs and the running cost. The following screenshots are from running “test.sh”.

- Existing Test Cases

```
____TESTING bitcount1____
  input 1: PASSED with Cost: 218.2288
  input 2: PASSED with Cost: 1351.6672
  input 3: PASSED with Cost: 386.1456
  input 4: PASSED with Cost: 50.3120
  input 5: PASSED with Cost: 92.2912
____TESTING bitcount2____
  input 1: PASSED with Cost: 201.3744
  input 2: PASSED with Cost: 1256.4480
  input 3: PASSED with Cost: 357.6816
  input 4: PASSED with Cost: 45.0672
  input 5: PASSED with Cost: 84.1440
____TESTING bitcount3____
  input 1: PASSED with Cost: 135.0832
  input 2: PASSED with Cost: 1364.2656
  input 3: PASSED with Cost: 347.0112
  input 4: PASSED with Cost: 50.3120
  input 5: PASSED with Cost: 92.6976
____TESTING bitcount4____
  input 1: PASSED with Cost: 25400.8304
  input 2: PASSED with Cost: 25403.6560
  input 3: PASSED with Cost: 25401.6144
  input 4: PASSED with Cost: 25400.8016
  input 5: PASSED with Cost: 25400.8048
____TESTING bitcount5____
  input 1: PASSED with Cost: 392.6752
  input 2: PASSED with Cost: 872.3872
  input 3: PASSED with Cost: 460.9920
  input 4: PASSED with Cost: 324.4704
  input 5: PASSED with Cost: 392.6496
____TESTING bubble_sort____
  input 1: PASSED with Cost: 6628.9360
  input 2: PASSED with Cost: 562437.2464
  input 3: PASSED with Cost: 61987402.0422
____TESTING collatz____
  input 1: PASSED with Cost: 68.4400
  input 2: PASSED with Cost: 68.4400
____TESTING gcd____
  input 1: PASSED with Cost: 46.5024
  input 2: PASSED with Cost: 105.4816
  input 3: PASSED with Cost: 223.4272
  input 4: PASSED with Cost: 1043.0336
____TESTING prime____
  input 1: PASSED with Cost: 117.4400
  input 2: PASSED with Cost: 6678.1008
  input 3: PASSED with Cost: 1548664.4224
  input 4: PASSED with Cost: 5628813.2864
____TESTING binary_tree____
  input 1: PASSED with Cost: 2402.8096
  input 2: PASSED with Cost: 4310.6816
  input 3: PASSED with Cost: 66113.1696
  input 4: PASSED with Cost: 1033485.1248
  input 5: PASSED with Cost: 1977050335.4539
```

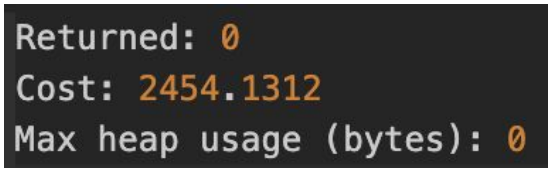
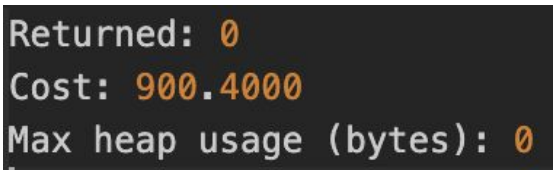
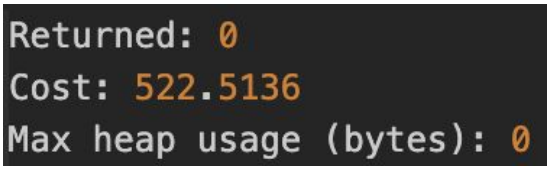
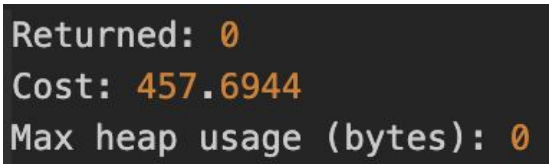
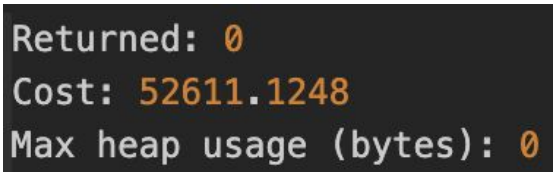
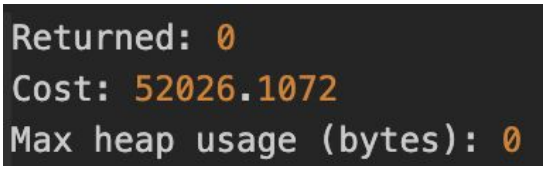
<before optimization>

```
____TESTING bitcount1____
  input 1: PASSED with Cost: 76.2608
  input 2: PASSED with Cost: 449.2064
  input 3: PASSED with Cost: 131.5120
  input 4: PASSED with Cost: 21.0096
  input 5: PASSED with Cost: 34.8224
____TESTING bitcount2____
  input 1: PASSED with Cost: 79.8128
  input 2: PASSED with Cost: 484.8992
  input 3: PASSED with Cost: 139.8256
  input 4: PASSED with Cost: 19.8000
  input 5: PASSED with Cost: 34.8032
____TESTING bitcount3____
  input 1: PASSED with Cost: 49.4352
  input 2: PASSED with Cost: 461.6064
  input 3: PASSED with Cost: 120.4992
  input 4: PASSED with Cost: 21.0096
  input 5: PASSED with Cost: 35.2224
____TESTING bitcount4____
  input 1: PASSED with Cost: 10063.3136
  input 2: PASSED with Cost: 10065.9360
  input 3: PASSED with Cost: 10064.0976
  input 4: PASSED with Cost: 10063.2848
  input 5: PASSED with Cost: 10063.2880
____TESTING bitcount5____
  input 1: PASSED with Cost: 136.1936
  input 2: PASSED with Cost: 305.7472
  input 3: PASSED with Cost: 160.4320
  input 4: PASSED with Cost: 111.9456
  input 5: PASSED with Cost: 136.1680
____TESTING bubble_sort____
  input 1: PASSED with Cost: 3653.7760
  input 2: PASSED with Cost: 319630.3680
  input 3: PASSED with Cost: 36740879.7229
____TESTING collatz____
  input 1: PASSED with Cost: 30.2256
  input 2: PASSED with Cost: 30.2256
____TESTING gcd____
  input 1: PASSED with Cost: 21.2000
  input 2: PASSED with Cost: 48.0352
  input 3: PASSED with Cost: 101.6992
  input 4: PASSED with Cost: 475.3472
____TESTING prime____
  input 1: PASSED with Cost: 52.3568
  input 2: PASSED with Cost: 3447.6528
  input 3: PASSED with Cost: 790127.5856
  input 4: PASSED with Cost: 2900104.9216
____TESTING binary_tree____
  input 1: PASSED with Cost: 1281.1168
  input 2: PASSED with Cost: 2397.9744
  input 3: PASSED with Cost: 38666.1568
  input 4: PASSED with Cost: 631008.0128
  input 5: PASSED with Cost: 1834106963.3826
```

<after optimization>

- New Test Cases : input .ll files are in the git repository (RegisterAllocationOpt-Testx.ll).

FileCheck 1	
Before Optimization	After Optimization

	
FileCheck 2	
Before Optimization	After Optimization
	
FileCheck 3	
Before Optimization	After Optimization
	

Progress Report: Ahyoung Oh

[Sprint 1] Arithmetic Pass

Change in Schedules (None)

Done and Not Dones

My original plan for Arithmetic pass was to apply following optimizations:

1) add a,a -> mul a,2
2) sub a,a -> 0
3) shl a,C -> mul a, 2^C
4) shr a,C -> udiv a, 2^C
5) %cond = icmp eq(X,Y)
   br i1 %cond, label %BB1, label %BB2
   ->
   %cond = xor x,y
   br i1 %cond, label %BB2, label %BB1
6) %cond = icmp neq(X,Y)
   br i1 %cond, label %BB1, label %BB2
   ->
   %cond = xor x,y
   br i1 %cond, label %BB1, label %BB2

I've successfully implemented 1)~ 4) but I couldn't implement 5) and 6) because of the type mismatch error(Assertion failed: (New->getType() == getType() && "replaceAllUses of value with new value of different type!"). In the planning phase I never thought of that error but dealing with it, I realized that it does make sense as BinaryOperator and IcmpInst are different instruction types. Making their types the same by casting or other processes might increase cost so I excluded them in my optimization. Also as I realized that m_Shr just matches the case of logical shift right, I changed that to m_aShr || m_lShr.

In addition to 1 ~ 4, I implemented four more optimizations during the development and code-review phase.

1) Integer equality propagation
   : Considering cost (constantint < argument < instruction), if a constant int was one of the operands, I propagated constant int, and if an argument was one of the operands, I propagated argument as argument is stored in the register. And for the rest, I propagated the first operand.
2) add a,0 -> a
3) sub a,0 -> a
4) mul a,0 -> 0

As a result, my final Arithmetic Pass first does integer equality propagation(this should be done first because it can further optimize later by matching into one of the patterns of interest) and then iterates through every instruction in every basic block again and looks for the patterns of interest that I'm willing to replace. And then for each pattern vector, I replace the original instructions to the cheaper instructions. The reason I separated into these two parts is to prevent iterator invalidation.

Test Results:

- Existing Test Cases

```
____TESTING bitcount1____
  input 1: PASSED with Cost: 218.2288
  input 2: PASSED with Cost: 1351.6672
  input 3: PASSED with Cost: 386.1456
  input 4: PASSED with Cost: 50.3120
  input 5: PASSED with Cost: 92.2912
____TESTING bitcount2____
  input 1: PASSED with Cost: 201.3744
  input 2: PASSED with Cost: 1256.4480
  input 3: PASSED with Cost: 357.6816
  input 4: PASSED with Cost: 45.0672
  input 5: PASSED with Cost: 84.1440
____TESTING bitcount3____
  input 1: PASSED with Cost: 135.0832
  input 2: PASSED with Cost: 1364.2656
  input 3: PASSED with Cost: 347.0112
  input 4: PASSED with Cost: 50.3120
  input 5: PASSED with Cost: 92.6976
____TESTING bitcount4____
  input 1: PASSED with Cost: 25400.8304
  input 2: PASSED with Cost: 25403.6560
  input 3: PASSED with Cost: 25401.6144
  input 4: PASSED with Cost: 25400.8016
  input 5: PASSED with Cost: 25400.8048
____TESTING bitcount5____
  input 1: PASSED with Cost: 392.6752
  input 2: PASSED with Cost: 872.3872
  input 3: PASSED with Cost: 460.9920
  input 4: PASSED with Cost: 324.4704
  input 5: PASSED with Cost: 392.6496
____TESTING bubble_sort____
  input 1: PASSED with Cost: 6628.9360
  input 2: PASSED with Cost: 562437.2464
  input 3: PASSED with Cost: 61987402.0422
____TESTING collatz____
  input 1: PASSED with Cost: 68.4400
  input 2: PASSED with Cost: 68.4400
____TESTING gcd____
  input 1: PASSED with Cost: 46.5024
  input 2: PASSED with Cost: 105.4816
  input 3: PASSED with Cost: 223.4272
  input 4: PASSED with Cost: 1043.0336
____TESTING prime____
  input 1: PASSED with Cost: 117.4400
  input 2: PASSED with Cost: 6678.1008
  input 3: PASSED with Cost: 1548664.4224
  input 4: PASSED with Cost: 5628813.2864
____TESTING binary_tree____
  input 1: PASSED with Cost: 2402.8096
  input 2: PASSED with Cost: 4310.6816
  input 3: PASSED with Cost: 66113.1696
  input 4: PASSED with Cost: 1033485.1248
  input 5: PASSED with Cost: 1977050335.4539
```

<before optimization>

```
____TESTING bitcount1____
  input 1: PASSED with Cost: 217.4288
  input 2: PASSED with Cost: 1345.4672
  input 3: PASSED with Cost: 384.5456
  input 4: PASSED with Cost: 50.3120
  input 5: PASSED with Cost: 92.0912
____TESTING bitcount2____
  input 1: PASSED with Cost: 200.5744
  input 2: PASSED with Cost: 1250.2480
  input 3: PASSED with Cost: 356.0816
  input 4: PASSED with Cost: 45.0672
  input 5: PASSED with Cost: 83.9440
____TESTING bitcount3____
  input 1: PASSED with Cost: 135.0832
  input 2: PASSED with Cost: 1364.2656
  input 3: PASSED with Cost: 347.0112
  input 4: PASSED with Cost: 50.3120
  input 5: PASSED with Cost: 92.6976
____TESTING bitcount4____
  input 1: PASSED with Cost: 25400.2304
  input 2: PASSED with Cost: 25403.0560
  input 3: PASSED with Cost: 25401.0144
  input 4: PASSED with Cost: 25400.2016
  input 5: PASSED with Cost: 25400.2048
____TESTING bitcount5____
  input 1: PASSED with Cost: 392.4752
  input 2: PASSED with Cost: 870.7872
  input 3: PASSED with Cost: 460.5920
  input 4: PASSED with Cost: 324.4704
  input 5: PASSED with Cost: 392.4496
____TESTING bubble_sort____
  input 1: PASSED with Cost: 6628.9360
  input 2: PASSED with Cost: 562437.2464
  input 3: PASSED with Cost: 61987402.0422
____TESTING collatz____
  input 1: PASSED with Cost: 68.4400
  input 2: PASSED with Cost: 68.4400
____TESTING gcd____
  input 1: PASSED with Cost: 46.5024
  input 2: PASSED with Cost: 105.4816
  input 3: PASSED with Cost: 223.4272
  input 4: PASSED with Cost: 1043.0336
____TESTING prime____
  input 1: PASSED with Cost: 117.4400
  input 2: PASSED with Cost: 6678.1008
  input 3: PASSED with Cost: 1548664.4224
  input 4: PASSED with Cost: 5628813.2864
____TESTING binary_tree____
  input 1: PASSED with Cost: 2402.8096
  input 2: PASSED with Cost: 4310.6816
  input 3: PASSED with Cost: 66113.1696
  input 4: PASSED with Cost: 1033485.1248
  input 5: PASSED with Cost: 1977050335.4539
```

<after optimization>

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

<table>
<tr><th colspan="2">FileCheck 1(do Add and Sub)</th></tr>
<tr><td>Before Optimization</td><td>After Optimization</td></tr>
<tr><td>

swpp202001-interpreter > sf-interpreter.log

```
Returned: 0
Cost: 52.6544
Max heap usage (bytes): 0

```

</td><td>

swpp202001-interpreter > sf-interpreter.log

```
Returned: 0
Cost: 46.8416
Max heap usage (bytes): 0

```

</td></tr>
<tr><th colspan="2">FileCheck 2(do Shift)</th></tr>
<tr><td>Before Optimization</td><td>After Optimization</td></tr>
<tr><td>

swpp202001-interpreter > sf-interpreter.log

```
Returned: 0
Cost: 83.3344
Max heap usage (bytes): 0

```

</td><td>

swpp202001-interpreter > sf-interpreter.log

```
Returned: 0
Cost: 80.1344
Max heap usage (bytes): 0

```

</td></tr>
<tr><th colspan="2">FileCheck 3(do Integer Eq Propagation)</th></tr>
<tr><td>Before Optimization</td><td>After Optimization</td></tr>
<tr><td>

swpp202001-interpreter > sf-interpreter.log

```
Returned: 0
Cost: 82.9440
Max heap usage (bytes): 0

```

</td><td>

swpp202001-interpreter > sf-interpreter.log

```
Returned: 0
Cost: 81.7440
Max heap usage (bytes): 0

```

</td></tr>
</table>

Progress Report: Jaewook Lee

[Sprint 1] Malloc to Alloca Conversion Pass

Change in Schedules (None)

Done and Not Dones

Done:

In sprint 1, I completed what I originally planned for : converting ‘static(not dynamic)’ ‘local(freed in the same function/ not captured)’ malloc instructions to alloca instructions. This successfully reduces the cost as will be shown below if the function satisfies mentioned conditions.

My function-level pass successfully converts Malloc instruction to alloca instruction. It is divided into 3 stages.

Stage 1 : Find all malloc call instructions in the function and check if they satisfy the conditions

> Condition 1 : Not a dynamic allocation.
>
> Condition 2 : The size of the allocated memory is lower than 2048 bytes.
>
> If the malloc call instruction satisfies the conditions, it is stored in ‘PossibleMallocs’ vector.

Stage 2 : Check if malloc calls instructions in ‘PossibleMallocs’ are replaceable.

> It is replaceable only if it is freed before the function ends. In other words, there should be free calls on every path from the malloc instruction to the exit block. Heap-allocated memory should not be captured.
>
> If the malloc call instruction is replaceable, it is stored in ‘ReplaceableMallocs’ vector and corresponding free instructions are stored in ‘RemovableFrees’ vector.

Stage 3 : Replace mallocs in ‘ReplaceableMallocs’ with alloca and remove frees in ‘RemovableFrees’

Not Done:

However, while making the pass and testing it, I found out that some cases are not being optimized. For instance,

```
%a = alloca i64*, align 8
%call = call noalias i8* @malloc(i64 32) #3
%0 = bitcast i8* %call to i64*
store i64* %0, i64** %a, align 8
....
%7 = load i64*, i64** %a, align 8
%8 = bitcast i64* %7 to i8*
call void @free(i8* %8) #3
```

As you can see in the code above, although the malloc is being freed in the same function, my pass cannot track the pointer allocated by malloc with the freed by free because it is stored and loaded between the pointer-conversion. This is because I used 'getUnderlyingObject( )' to track the original pointer, but it does not do the flow-sensitive analysis (which means sensitive to the sequence of the instructions). It can only track the pointer-conversion of bitcast. So in the case above, with %8, I can only get %7, not %0 or %call.

In order to make this happen, I have to use GVN Pass which already exists as LLVM Pass. We are going to import this GVN Pass in the next sprint to fully utilize Malloc2AllocPass.

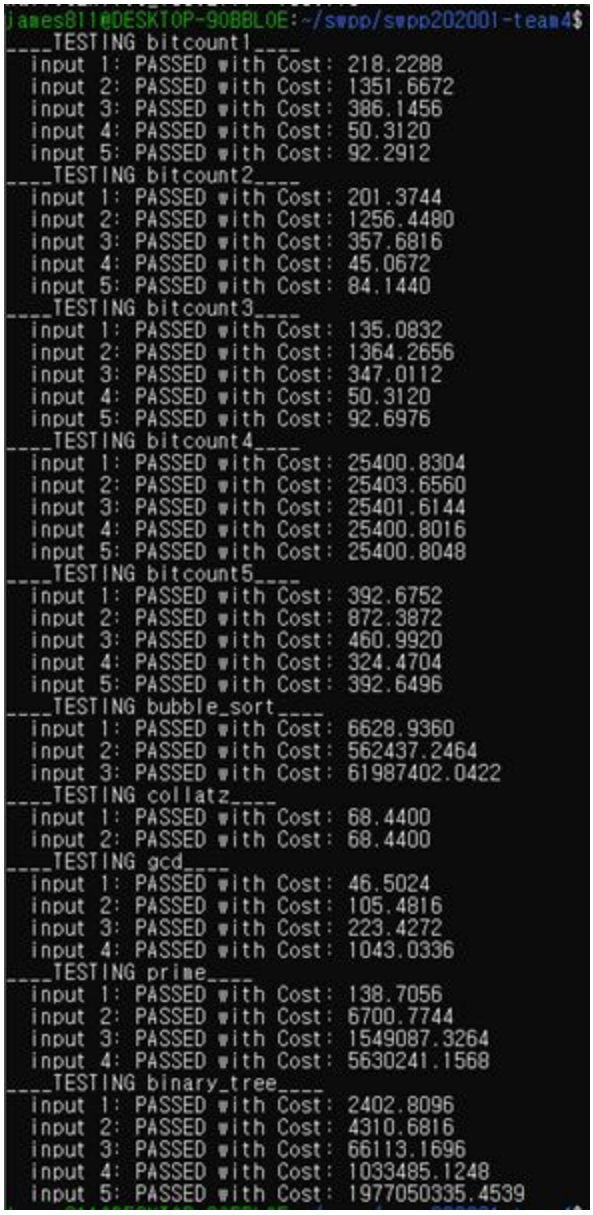
Also, my Malloc2AllocPass might unintentionally remove the frees to the memory allocated in the heap. For instance,

```
void test4( ) {
   int *x;
   int cond = 1;
   if (cond ) x = malloc(32);
   else x = malloc(3000);
   free (x)
}
```

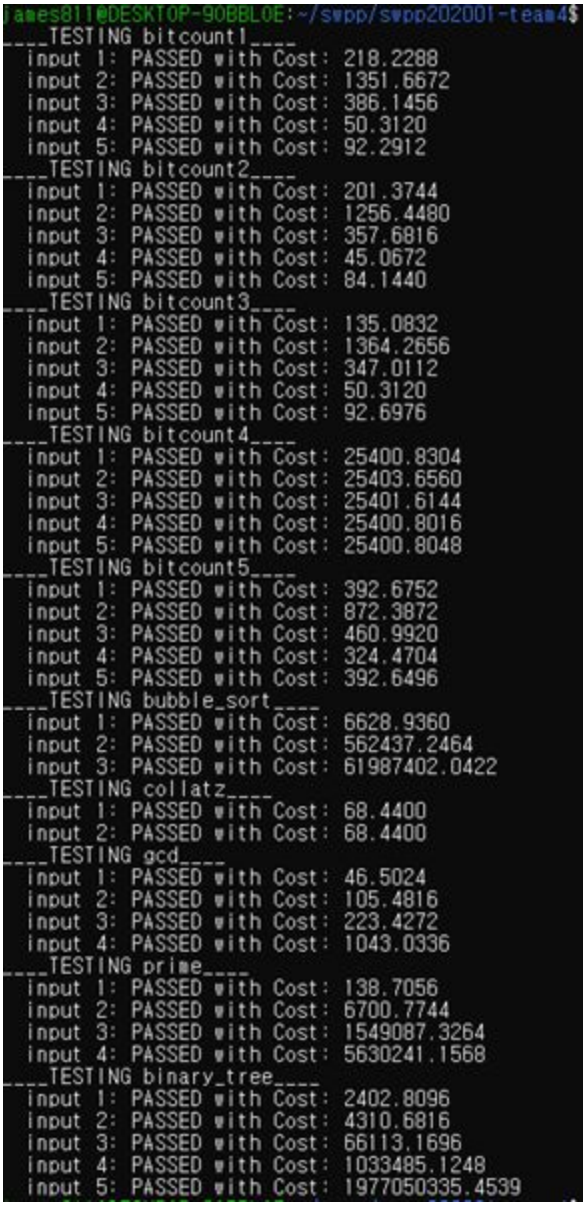
This case is in the git repository as Malloc2AllocPass_Test3. My pass removes malloc(32) but does not remove malloc(3000). However, free(x) will be removed. This might cause inefficiency in memory usage. I can fix this by adding a new IR instruction that frees memory allocated by malloc(3000), but could not do this in this sprint due to the 200 line- limitation.

Test Results:

- Existing Test Cases



<before optimization>



<after optimization>

This is the result of the existing tests. Unfortunately, there were no cases in the provided tests that my optimization pass could reduce the cost, because there were few tests that had not dynamic ‘malloc’ and corresponding ‘free’ in the same function. Only bubble_sort, binary_tree, and prime used malloc in their code, and none of them was freeing that allocated heap memory in the same function. Therefore, I had to check the performance of Malloc2AllocPass by making my own test.

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

FileCheck 1

Before Optimization	After Optimization
Returned: 0 Cost: 361.3600 Max heap usage (bytes): 64	Returned: 0 Cost: 307.5232 Max heap usage (bytes): 64

FileCheck 2

Before Optimization	After Optimization
Returned: 0 Cost: 67350.1248 Max heap usage (bytes): 2400	Returned: 0 Cost: 60089.0624 Max heap usage (bytes): 2400

FileCheck 3

Before Optimization	After Optimization
Returned: 0 Cost: 318.2928 Max heap usage (bytes): 32	Returned: 0 Cost: 262.1456 Max heap usage (bytes): 32

FileCheck 4 (revision from FileCheck 2 to show max heap usage improvement)

Before Optimization	After Optimization
Returned: 0 Cost: 36470.5424 Max heap usage (bytes): 2048	Returned: 0 Cost: 29208.7376 Max heap usage (bytes): 0

Progress Report: Jaeeun Lee

[Sprint 1] Function Outlining Pass

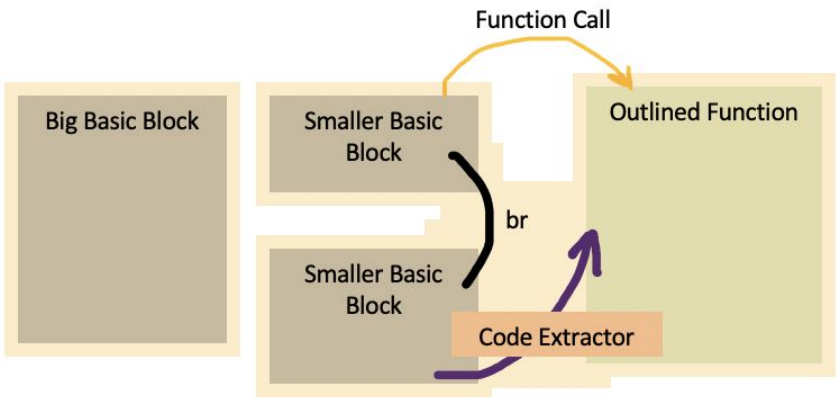
Change in Schedules (Packing Registers -> Function Outlining Pass)

The original plan for sprint 1 was to implement the <Packing Register Optimization> and this was assigned to me. But due to the fact that packing registers is quite complicated to implement in the first sprint, and mainly because our team members decided that reducing stores and loads in current backend is urgent (one person could work on SimpleBackend at once to prevent conflicts and dependencies), I decided to implement the Function Outlining pass in this sprint instead, while Hyoung Uk makes changes to the SimpleBackend.

Done and Not Dones

The Function Outlining pass's main purpose is to reduce costly memory access and change them into function calls. The goal was to outline 'blocks' in functions to a separate function, but in this sprint, the optimization deals with the case where we split 'big' blocks in to smaller blocks. By 'big' , we mean blocks that use more than 13 registers. So, in this pass, we split the blocks into 2 parts if the block is big enough, and the latter part is outlined into a new function.

How it Works:



This optimization was done by keeping track of how many registers the basic block used, and if it uses more than 13 registers, we save the 14th register usage instruction. Then, from that part we break the block into two parts and linking it with a 'br' instruction. After that, we outline the latter block using the CodeExtractor. Because additional branches are introduced, we also added the SimplifyCFGPass with this pass together, so that we could eliminate redundant blocks and make the truly take advantage of the this function outlining optimization.

The original plan was to deal with more cases, such as extracting a whole block out of the function into a new outlined function, or extracting multiple blocks. But in this pass, we focused only on the case where we split one block into half. The more complicated cases are planned to be dealt with in the sprint 2, due to the time constraints and because we had to make sure that this optimization would not cause any other problems or result in increase costs.

Test Results:

- Existing Test Cases

```
____TESTING bitcount1____
  input 1: PASSED with Cost: 218.2288
  input 2: PASSED with Cost: 1351.6672
  input 3: PASSED with Cost: 386.1456
  input 4: PASSED with Cost: 50.3120
  input 5: PASSED with Cost: 92.2912
____TESTING bitcount2____
  input 1: PASSED with Cost: 201.3744
  input 2: PASSED with Cost: 1256.4480
  input 3: PASSED with Cost: 357.6816
  input 4: PASSED with Cost: 45.0672
  input 5: PASSED with Cost: 84.1440
____TESTING bitcount3____
  input 1: PASSED with Cost: 135.0832
  input 2: PASSED with Cost: 1364.2656
  input 3: PASSED with Cost: 347.0112
  input 4: PASSED with Cost: 50.3120
  input 5: PASSED with Cost: 92.6976
____TESTING bitcount4____
  input 1: PASSED with Cost: 25400.8304
  input 2: PASSED with Cost: 25403.6560
  input 3: PASSED with Cost: 25401.6144
  input 4: PASSED with Cost: 25400.8016
  input 5: PASSED with Cost: 25400.8048
____TESTING bitcount5____
  input 1: PASSED with Cost: 392.6752
  input 2: PASSED with Cost: 872.3872
  input 3: PASSED with Cost: 460.9920
  input 4: PASSED with Cost: 324.4704
  input 5: PASSED with Cost: 392.6496
____TESTING bubble_sort____
  input 1: PASSED with Cost: 6628.9360
  input 2: PASSED with Cost: 562437.2464
  input 3: PASSED with Cost: 61987402.0422
____TESTING collatz____
  input 1: PASSED with Cost: 68.4400
  input 2: PASSED with Cost: 68.4400
____TESTING gcd____
  input 1: PASSED with Cost: 46.5024
  input 2: PASSED with Cost: 105.4816
  input 3: PASSED with Cost: 223.4272
  input 4: PASSED with Cost: 1043.0336
____TESTING prime____
  input 1: PASSED with Cost: 117.4400
  input 2: PASSED with Cost: 6678.1008
  input 3: PASSED with Cost: 1548664.4224
  input 4: PASSED with Cost: 5628813.2864
____TESTING binary_tree____
  input 1: PASSED with Cost: 2402.8096
  input 2: PASSED with Cost: 4310.6816
  input 3: PASSED with Cost: 66113.1696
  input 4: PASSED with Cost: 1033485.1248
  input 5: PASSED with Cost: 1977050335.4539
```

<before optimization>

```
____TESTING bitcount1____
  input 1: PASSED with Cost: 218.2288
  input 2: PASSED with Cost: 1351.6672
  input 3: PASSED with Cost: 386.1456
  input 4: PASSED with Cost: 50.3120
  input 5: PASSED with Cost: 92.2912
____TESTING bitcount2____
  input 1: PASSED with Cost: 224.5600
  input 2: PASSED with Cost: 1442.6704
  input 3: PASSED with Cost: 405.0208
  input 4: PASSED with Cost: 44.0992
  input 5: PASSED with Cost: 89.2144
____TESTING bitcount3____
  input 1: PASSED with Cost: 135.0832
  input 2: PASSED with Cost: 1364.2656
  input 3: PASSED with Cost: 347.0112
  input 4: PASSED with Cost: 50.3120
  input 5: PASSED with Cost: 92.6976
____TESTING bitcount4____
  input 1: PASSED with Cost: 25167.5312
  input 2: PASSED with Cost: 25170.3568
  input 3: PASSED with Cost: 25168.3152
  input 4: PASSED with Cost: 25167.5024
  input 5: PASSED with Cost: 25167.5056
____TESTING bitcount5____
  input 1: PASSED with Cost: 392.6752
  input 2: PASSED with Cost: 872.3872
  input 3: PASSED with Cost: 460.9920
  input 4: PASSED with Cost: 324.4704
  input 5: PASSED with Cost: 392.6496
____TESTING bubble_sort____
  input 1: PASSED with Cost: 6624.5568
  input 2: PASSED with Cost: 557892.4752
  input 3: PASSED with Cost: 61491953.3509
____TESTING collatz____
  input 1: PASSED with Cost: 67.5424
  input 2: PASSED with Cost: 67.5424
____TESTING gcd____
  input 1: PASSED with Cost: 51.6304
  input 2: PASSED with Cost: 122.7440
  input 3: PASSED with Cost: 264.9584
  input 4: PASSED with Cost: 1248.3824
____TESTING prime____
  input 1: PASSED with Cost: 121.4848
  input 2: PASSED with Cost: 8008.0592
  input 3: PASSED with Cost: 1930300.2496
  input 4: PASSED with Cost: 7081422.9121
____TESTING binary_tree____
  input 1: PASSED with Cost: 2615.5408
  input 2: PASSED with Cost: 4885.1904
  input 3: PASSED with Cost: 75950.1440
  input 4: PASSED with Cost: 1189850.7664
  input 5: PASSED with Cost: 2009952135.7711
```

<after optimization>

The costs of the test cases have decreased as shown above after the optimization. Note that after decreasing all the store and loads in the SimpleBackend (which is handled by Hyoung Uk in this sprint),

the cost will decrease even more (In order to actually take advantage of the function outlining, this optimization will have to be combined with register allocation optimization)

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

<table>
<tr><td colspan="2">FileCheck 1</td></tr>
<tr><td>Before Optimization</td><td>After Optimization</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td colspan="2">FileCheck 2</td></tr>
<tr><td>Before Optimization</td><td>After Optimization</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td colspan="2">FileCheck 3</td></tr>
<tr><td>Before Optimization</td><td>After Optimization</td></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

Appendix: Updated Development Plan

	Sprint 1	Sprint 2	Sprint 3
Hyoung Uk Sul	1	5	9
	SimpleBackend Register Allocation Part 1	SimpleBackend Register Allocation Part 2	Function Inlining
Ahyoung Oh	4	6	6
	Arithmetic Optimizations	Reordering Memory Accesses Part 1	Reordering Memory Accesses Part 2
Jaewook Lee	3	7	11
	Malloc to Alloca Conversion	Runtime Garbage Collector	Dead Argument Elimination
Jaeeun Lee	2	8	12
	Function Outlining Part 1	Function Outlining Part 2	Induction Variable Strength Reduction Pass