[SWPP 202001 Project]

Team 1 Development Progress : Sprint I

Team 1(이진우, 정덕인, 김도현, 임지규)

1A. GV to malloc in main(임지규)

1A-1. Basic description of the task

This chunk has two goals: one is to implement the GV to malloc pass and embed it in our project with standard GV optimizing passes, and other is to implement malloc in main to GV.

GV to malloc pass moves all the global variables to the *malloc* heap assignment in *main()*. Naive assembler of our project translates GV into the *malloc* instruction in *main()*. Why not before? Our project specification ensures that there exists main() in the IR, allowing this optimization in any cases. This pass will be activated later than the LLVM bundle passes which find and optimize GV uses, but before the heap allocation optimization passes(3-1 from the index above) for maximum efficiency.

However, some heap allocations originating from GV may not be further modified into stack allocation or register by other passes till the end. These leftovers should be converted to GV to reduce cost of *call* instruction. Implementation details about the pass of this role is found in 1B of the progress document.

1A-2. Implementation details changed during the *Dev. sprint-1*

1. Global value are replaced into malloc function inside the main.
2. The malloc variable was put as the argument of function foo.(foo means the other functions except main function in llvm code.)
3. The use of the original GV is changed to the new malloc variable pointer.

```
for GV in Module
  size = GV.size()
  m = MakeNewMalloc(size)
  for f in functions
    f.GiveArguments(m)
  for every call in module:
    rewrite calls to match signature
  GV.setUses(m)
```

This above is the pseudocode we planned before.
The overall structure of this code was not changed and implemented same.
But there are some differences between this code and the final code.
This changes have been made to fix some bugs or to make the processing efficient.

1. The malloc function was pre-created at the beginning of the 'run' function and then used for that MakeNewMalloc() function.
We thought the malloc function would be created within the MakeNewMalloc() function but it caused an unexpected behavior that created multiple malloc functions. One malloc() is sufficient to modify the whole code.

2. The last statement of the pseudocode - GV.setuses(m) had to be changed.
Contrary to what we initially thought, it did not work that way because the replacing variables vary from function to function.
To create the expected behavior, we had to find the right pointer value in each function which points to the malloc pointer, and used the standard Value::replaceUsesWithIf() to replace the original uses of GV to the function-local representative of it.

1-3. Test results & Improvements

1. LLVM IR input

```
@GV = global i32 0, align 4
@GV2 = global i32 1, align 4

define i32 @main() #0 {
  %1 = alloca i32, align 4
```

```
  store i32 0, i32* %1, align 4
  store i32 10, i32* @GV2, align 4
  %gvtest = load i32, i32* @GV2, align 4
  %2 = call i32 @foo()
  ret i32 %2
}

define i32 @foo() #0 {
  %1 = load i32, i32* @GV, align 4
  %2 = mul nsw i32 2, %1
  ret i32 %2
}
```

2. LLVM IR output

```
@GV = global i32 0, align 4
@GV2 = global i32 1, align 4

define i32 @main() {
  %ma1 = call i8* @malloc(i64 32)
  %gv1 = bitcast i8* %ma1 to i32*
  store i32 1, i32* %gv1
  %ma0 = call i8* @malloc(i64 32)
  %gv0 = bitcast i8* %ma0 to i32*
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, i32* %gv1, align 4
  %gvtest = load i32, i32* %gv1, align 4
  %2 = call i32 @foo.1(i32* %gv0, i32* %gv1)
  ret i32 %2
}
```

1. LLVM IR input

```
@GV = global i32 2, align 4
@arr = global [3 x i32] [i32 1, i32 2, i32 3]

define i32 @main() #0 {
  %idx_0 = getelementptr [3 x i32], [3 x i32]* @arr, i64 0, i64 0
  store i32 10, i32* %idx_0
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %3 = call i32 @foo()
```

```
  store i32 %3, i32* %2, align 4
  %4 = load i32, i32* @GV, align 4
  %5 = load i32, i32* %2, align 4
  %6 = add nsw i32 %4, %5
  ret i32 %6
}

define i32 @foo() #0 {
  %1 = load i32, i32* @GV, align 4
  %2 = load i32, i32* @GV, align 4
  %3 = mul nsw i32 %1, %2
  ret i32 %3
}
```

2. LLVM IR output

```
@GV = global i32 2, align 4
@arr = global [3 x i32] [i32 1, i32 2, i32 3]

define i32 @main() {
  %ma0 = call i8* @malloc(i64 32)
  %gv0 = bitcast i8* %ma0 to i32*
  store i32 2, i32* %gv0
  %idx_0 = getelementptr [3 x i32], [3 x i32]* @arr, i64 0, i64 0
  store i32 10, i32* %idx_0
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %3 = call i32 @foo.1(i32* %gv0)
  store i32 %3, i32* %2, align 4
  %4 = load i32, i32* %gv0, align 4
  %5 = load i32, i32* %2, align 4
  %6 = add nsw i32 %4, %5
  ret i32 %6
}

; Function Attrs: noinline optnone
declare i8* @malloc(i64) #0

define i32 @foo.1(i32* %gv0) {
  %1 = load i32, i32* %gv0, align 4
  %2 = load i32, i32* %gv0, align 4
```

```
  %3 = mul nsw i32 %1, %2
  ret i32 %3
}
```

1. LLVM IR input

```
@GV = global i32 3, align 4

define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 2, i32* @GV, align 4
  %2 = load i32, i32* @GV, align 4
  ret i32 %2
}
```

2. LLVM IR output

```
@GV = global i32 3, align 4

define i32 @main() {
  %ma0 = call i8* @malloc(i64 32)
  %gv0 = bitcast i8* %ma0 to i32*
  store i32 3, i32* %gv0
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 2, i32* %gv0, align 4
  %2 = load i32, i32* %gv0, align 4
  ret i32 %2
}
```

The results were as expected. But one different thing is in case of array type of global variables. In case of global arrays, We decided not to process the type since it causes multiple errors. For further implementations relating to memory accesses, we strictly plan to just transform scalar variables and not arrays. Heap allocated array pointers are larger in multiples than their type size, so it could be easily distinguished.

1B. MallocToGVPass (이진우)

1B-1. Basic description of the task

After the memory allocation optimization, GVs which initially resided inside the

main() function as a malloc() call are lowered to stack or register memory. However, some GVs(now malloc() calls in main()) may still stay in the main() function for several reasons.[1] Returning these remaining GV-malloc()s to GV may reduce the call cost since we pass the address of them by function arguments. Functions can easily retrieve the address of GVs by the symbol table interpretation.

In consequence, we present a pass which transforms malloc() calls in entry block of main() to GV. This pass will clean up the malloc calls formed by its counterpart, GVToMalloc pass(1A). This pass will leave dead arguments which originally carried the pointer values of malloc()-ized GVs, so performing Dead Argument Elimination after this pass is strongly recommended.

1B-2. Implementation details changed during the *Dev. sprint*

All features are successfully implemented as planed. However, we found out that no standard Alias Analysis pass finds two arguments which obviously shares values. This feature was needed to connect a malloc() call and function arguments which receive them as parameters. Instead importing any pre-developed analysis passes, we traversed over the call-graph to dynamically determine which arguments obviously contain the same value as the target malloc().

1B-3. Test results & Improvements

1. LLVM IR input

```
define i32 @main() {
    %m0.0 = call i8* @malloc(i64 4)
    %m0.1 = bitcast i8* %m0.0 to i32*
    store i32 0, i32* %m0.1
    %ret = call i32 @foo(i32* %m0.1);
    ret i32 %ret
}

define i32 @foo(i32* %ptr) {
    %r0 = load i32, i32* %ptr
    %r1 = add i32 %r0, 3
    store i32 %r1, i32* %ptr
    ret i32 %r1
}
```

2. LLVM IR output

1) Note that the pass Heap-to-stack+arg lowering will be responsible for this decision.

```
@gv.m0.1 = common global i32 0

define i32 @main() {
  %m0.0 = call i8* @malloc(i64 4)
  %m0.1 = bitcast i8* %m0.0 to i32*
  %ret = call i32 @foo(i32* @gv.m0.1)
  ret i32 %ret
}

define i32 @foo(i32* %ptr) {
  %r0 = load i32, i32* @gv.m0.1
  %r1 = add i32 %r0, 3
  store i32 %r1, i32* @gv.m0.1
  ret i32 %r1
}
```

1. LLVM IR input

```
define i32 @main() {
    %m0.0 = call i8* @malloc(i64 4)
    %m0.1 = bitcast i8* %m0.0 to i32*
    store i32 5, i32* %m0.1
    %foo = call i32 @foo(i32* %m0.1);
    %bar = call i32 @bar(i32 5, i32* %m0.1);
    %ret = add i32 %foo, %bar
    ret i32 %ret
}

define i32 @foo(i32* %ptr) {
    %r0 = load i32, i32* %ptr
    %r1 = add i32 %r0, 3
    store i32 %r1, i32* %ptr
    %r2 = call i32 @bar(i32 -2, i32* %ptr)
    ret i32 %r1
}

define i32 @bar(i32 %n, i32* %ptr2) {
    %r3 = load i32, i32* %ptr2
    %r4 = mul i32 %r3, %n
    store i32 %r4, i32* %ptr2
    ret i32 %r4
```

```
}
```

2. LLVM IR output

```
define i32 @main() {
  %m0.0 = call i8* @malloc(i64 4)
  %m0.1 = bitcast i8* %m0.0 to i32*
  %foo = call i32 @foo(i32* @gv.m0.1)
  %bar = call i32 @bar(i32 5, i32* @gv.m0.1)
  %ret = add i32 %foo, %bar
  ret i32 %ret
}

define i32 @foo(i32* %ptr) {
  %r0 = load i32, i32* @gv.m0.1
  %r1 = add i32 %r0, 3
  store i32 %r1, i32* @gv.m0.1
  %r2 = call i32 @bar(i32 -2, i32* @gv.m0.1)
  ret i32 %r1
}

define i32 @bar(i32 %n, i32* %ptr2) {
  %r3 = load i32, i32* @gv.m0.1
  %r4 = mul i32 %r3, %n
  store i32 %r4, i32* @gv.m0.1
  ret i32 %r4
}
```

1. LLVM IR input

```
define i32 @main() {
    %m0.0 = call i8* @malloc(i64 4)
    %m0.1 = bitcast i8* %m0.0 to i32*
    store i32 0, i32* %m0.1
    call void @foo(i32 10, i32* %m0.1);
    ret i32 0
}

define void @foo(i32 %n, i32* %ptr) {
    %r0 = load i32, i32* %ptr
    %r1 = add i32 %r0, %n
    store i32 %r1, i32* %ptr
```

```
    %n.next = sub i32 %n, 1
    %n.cmp = icmp eq i32 %n.next, 0
    br i1 %n.cmp, label %stop, label %recur
stop:
    ret void
recur:
    call void @foo(i32 %n.next, i32* %ptr)
    ret void
}

define i8* @malloc(i64 %size.malloc) {
    %ptr.malloc = inttoptr i64 0 to i8*
    ret i8* %ptr.malloc
}
```

2. LLVM IR output

```
@gv.m0.1 = common global i32 0

define i32 @main() {
  %m0.0 = call i8* @malloc(i64 4)
  %m0.1 = bitcast i8* %m0.0 to i32*
  call void @foo(i32 10, i32* @gv.m0.1)
  ret i32 0
}

define void @foo(i32 %n, i32* %ptr) {
  %r0 = load i32, i32* @gv.m0.1
  %r1 = add i32 %r0, %n
  store i32 %r1, i32* @gv.m0.1
  %n.next = sub i32 %n, 1
  %n.cmp = icmp eq i32 %n.next, 0
  br i1 %n.cmp, label %stop, label %recur

stop:                                             ; preds = %0
  ret void

recur:                                            ; preds = %0
  call void @foo(i32 %n.next, i32* @gv.m0.1)
  ret void
}
```

1. LLVM IR input

```
define i32 @main() {
    %m0.0 = call i8* @malloc(i64 4)
    %ret = call i32 @foo(i8* %m0.0);
    ret i32 %ret
}

define i32 @foo(i8* %ptr) {
    %r0 = load i8, i8* %ptr
    %r1 = add i8 %r0, 3
    store i8 %r1, i8* %ptr
    %r2 = sext i8 %r0 to i32
    ret i32 %r2
}

define i8* @malloc(i64 %size.malloc) {
    %ptr.malloc = inttoptr i64 0 to i8*
    ret i8* %ptr.malloc
}
```

2. LLVM IR output

```
@gv.m0.0 = common global i8

define i32 @main() {
  %m0.0 = call i8* @malloc(i64 4)
  %ret = call i32 @foo(i8* @gv.m0.0)
  ret i32 %ret
}

define i32 @foo(i8* %ptr) {
  %r0 = load i8, i8* @gv.m0.0
  %r1 = add i8 %r0, 3
  store i8 %r1, i8* @gv.m0.0
  %r2 = sext i8 %r0 to i32
  ret i32 %r2
}
```

The test performs the exact transformation it should. The semantics of the code is perfectly preserved while the malloc() in main() is safely transformed to GV. Value::getNumUses()==0 is checked if the arguments which carry malloc() addresses are dead properly. However, this transformation does not support array types currently.

Currently, we do not exactly understand the properties of GV linkage types. in

this version, we assign the 'CommonLinkage', represented as *common* in LLVM IR. If any changes are needed, we will modify it later. However, we predict that this is enough for our project now.

Co-test involving this pass and the GVToMalloc() pass is not yet performed since running GVToMalloc() pass still emit run-time errors now. After the development and bugfixes are complete, we will run a small test which runs GVToMalloc, MallocToGv, and Dead Argument Elimiation altogether(in respective to their given order), and the results will be analyzed to verify both passes.

4. Verifying Loop Optimization Pipeline 1(김도현)

4-1. Basic description of the task

This chunk is to embed the loop optimization pipeline of LLVM standard to our project. The list of existing passes which we will implement here is as following.

- Loop Simplify
- Loop-Closed SSA Form(Terminal phi node)
- Loop Invariant Code Motion
- Loop Rotate
- Loop Unswitching

These optimizations reduce redundant codes within the loops, and modify the call structure of the basic blocks. The task is to create a pass which wraps all these passes, so a single run through the pass will optimize loops as desired.

4-2. Implementation details changed during the *Dev. sprint1*

```
for Function f in Module
  LoopSimplifyPass(f)
    for Loop l in Function
      LCSSAPass(l)
      LICMPass(l)
      LoopRotatePass(l);
      LoopUnswitchingPass(l);
```

These optimization passes should be ordered as written above. LoopSimplify pass is function pass and the others are loop pass. I tried to organize them with PassManager class, but I could not. Thus, no PR is successfully made. This work will be completed by the next sprint with the Pipeline 2 Verifiication.

5. Vectorized Load & Store (정덕인)

5-1. Basic description of the task

The pass implemented here receives the functions which have load/store instruction with consequence memory access, and vectorizes the load/store. Our system has 64-bit register, which means that it can load at most 2 *int32*, 4 *int16* registers at once. We may load multiple variables as an *int64* type and then split it using *udiv* & *urem* if a sequencial load happens, and vice versa(sequential store will use *or* and *mul* instructions). This reduces the cost of load & store especially when the data type is small.

To perform vectorized load & store, we should find two consecutive load and store are happening in the code, However, most consequent memory access is done by looping, so Loop Unrolling pass compatible with the Vectorize pass is required. Loop unrolling pass unrolls and jams the loop, and then rearrange the load/store instructions to let Vectorize pass perform. Also, Global value numbering has to be activated to unify the base pointers which allows the pass to find the consecutive load and stores easily.

5-2. Implementation details changed during the *Dev. sprint*

Before Sprint 1, we planned to detect 8 consecutive load/store and collect according to the types of instructions. However, this method can get some problems. This methos has low applicability, because 8 or more consecutive load/store instructions are rare in code. Even after loop unrolling, most of loop unrollers do not unroll with factor of 8 or more,

In this reason, we implemented little bit different as previous plan. New version of pass detect two consecutive load/store, and make these into one load/store of bigger type. Detailed algorithm is following; first we find chains of add instructions, which are expressing sum of instruction and constant 1. Chain means sequence of instructions, and operands of instructions should be shown consequently. After finding these chains, then we find the list of getelementptr instructions of three types(i8*, i16*, i32*). From chains and list, we can find consecutive load/store instruction. We call function that converts two consecutive load/store from before into single load/store instruction.

After implementing above algorithm, we found some weird cases. These are case of multiple conversion, such as i8* to i32*, i8* to i64*, etc. After one conversion happened, the index accesses of getelementptr instructions changes into forms that difference of every adjacent index pair is 2. Because we detect convertible instructions by collecting getelementptr instructions of consecutive

index accessing, we cannot apply same conversion again. This situation cannot be resolved easily, so we will make effort to figure out solution for this in next sprints.

5-3. Test results & Improvements

1. LLVM IR input

```
; ModuleID = 'input1.ll'
source_filename = "input1.ll"

define i32 @main() {
  %1 = alloca [1024 x i32], align 16
  br label %2

.preheader:                                       ; preds = %2
  br label %20

2:                                                ; preds = %0, %2
  %.015 = phi i64 [ 0, %0 ], [ %18, %2 ]
  %3 = mul i64 %.015, %.015
  %4 = trunc i64 %3 to i32
  %5 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %.015
  store i32 %4, i32* %5, align 4
  %6 = add nuw nsw i64 %.015, 1
  %7 = mul i64 %6, %6
  %8 = trunc i64 %7 to i32
  %9 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %6
  store i32 %8, i32* %9, align 4
  %10 = add nuw nsw i64 %6, 1
  %11 = mul i64 %10, %10
  %12 = trunc i64 %11 to i32
  %13 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %10
  store i32 %12, i32* %13, align 4
  %14 = add nuw nsw i64 %10, 1
  %15 = mul i64 %14, %14
  %16 = trunc i64 %15 to i32
  %17 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %14
  store i32 %16, i32* %17, align 4
  %18 = add nuw nsw i64 %14, 1
  %19 = icmp ult i64 %18, 1024
```

```
  br i1 %19, label %2, label %.preheader

20:                                               ; preds = %.preheader, %20
  %.04 = phi i64 [ 0, %.preheader ], [ %36, %20 ]
  %.023 = phi i32 [ 0, %.preheader ], [ %35, %20 ]
  %21 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %.04
  %22 = load i32, i32* %21, align 4
  %23 = add nsw i32 %.023, %22
  %24 = add nuw nsw i64 %.04, 1
  %25 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %24
  %26 = load i32, i32* %25, align 4
  %27 = add nsw i32 %23, %26
  %28 = add nuw nsw i64 %24, 1
  %29 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %28
  %30 = load i32, i32* %29, align 4
  %31 = add nsw i32 %27, %30
  %32 = add nuw nsw i64 %28, 1
  %33 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %32
  %34 = load i32, i32* %33, align 4
  %35 = add nsw i32 %31, %34
  %36 = add nuw nsw i64 %32, 1
  %37 = icmp ult i64 %36, 1024
  br i1 %37, label %20, label %38

38:                                               ; preds = %20
  %.02.lcssa = phi i32 [ %35, %20 ]
  ret i32 %.02.lcssa
}
```

2. LLVM IR output

```
; ModuleID = 'test-ir/input1_opt.ll'
source_filename = "input1.ll"

define i32 @main() {
  %1 = alloca [1024 x i32], align 16
  br label %2

.preheader:                                       ; preds = %2
  br label %28
```

```
2:                                                ; preds = %2, %0
  %.015 = phi i64 [ 0, %0 ], [ %26, %2 ]
  %3 = mul i64 %.015, %.015
  %4 = trunc i64 %3 to i32
  %5 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %.015
  %6 = add nuw nsw i64 %.015, 1
  %7 = mul i64 %6, %6
  %8 = trunc i64 %7 to i32
  %9 = bitcast i32* %5 to i64*
  %10 = zext i32 %4 to i64
  %11 = zext i32 %8 to i64
  %12 = mul i64 %11, 4294967296
  %13 = or i64 %12, %10
  store i64 %13, i64* %9
  %14 = add nuw nsw i64 %6, 1
  %15 = mul i64 %14, %14
  %16 = trunc i64 %15 to i32
  %17 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %14
  %18 = add nuw nsw i64 %14, 1
  %19 = mul i64 %18, %18
  %20 = trunc i64 %19 to i32
  %21 = bitcast i32* %17 to i64*
  %22 = zext i32 %16 to i64
  %23 = zext i32 %20 to i64
  %24 = mul i64 %23, 4294967296
  %25 = or i64 %24, %22
  store i64 %25, i64* %21
  %26 = add nuw nsw i64 %18, 1
  %27 = icmp ult i64 %26, 1024
  br i1 %27, label %2, label %.preheader

28:                                               ; preds = %28, %.preheader
  %.04 = phi i64 [ 0, %.preheader ], [ %50, %28 ]
  %.023 = phi i32 [ 0, %.preheader ], [ %49, %28 ]
  %29 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %.04
  %30 = bitcast i32* %29 to i64*
  %31 = load i64, i64* %30
  %32 = urem i64 %31, 4294967296
  %33 = trunc i64 %32 to i32
  %34 = udiv i64 %31, 4294967296
```

```
  %35 = trunc i64 %34 to i32
  %36 = add nsw i32 %.023, %33
  %37 = add nuw nsw i64 %.04, 1
  %38 = add nsw i32 %36, %35
  %39 = add nuw nsw i64 %37, 1
  %40 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %39
  %41 = bitcast i32* %40 to i64*
  %42 = load i64, i64* %41
  %43 = urem i64 %42, 4294967296
  %44 = trunc i64 %43 to i32
  %45 = udiv i64 %42, 4294967296
  %46 = trunc i64 %45 to i32
  %47 = add nsw i32 %38, %44
  %48 = add nuw nsw i64 %39, 1
  %49 = add nsw i32 %47, %46
  %50 = add nuw nsw i64 %48, 1
  %51 = icmp ult i64 %50, 1024
  br i1 %51, label %28, label %52

52:                                               ; preds = %28
  %.02.lcssa = phi i32 [ %49, %28 ]
  ret i32 %.02.lcssa
}
```

1. LLVM IR input

```
; ModuleID = 'input2.ll'
source_filename = "input2.ll"

define i32 @main() {
  %1 = alloca [1024 x i32], align 16
  br label %2

.preheader:                                       ; preds = %2
  br i1 true, label %.lr.ph, label %.preheader._crit_edge

.preheader._crit_edge:                            ; preds = %.preheader
  br label %42

.lr.ph:                                           ; preds = %.preheader
  br label %16
```

```
2:                                                ; preds = %0, %2
  %.02 = phi i64 [ 0, %0 ], [ %14, %2 ]
  %3 = trunc i64 %.02 to i32
  %4 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %.02
  store i32 %3, i32* %4, align 4
  %5 = add nuw nsw i64 %.02, 1
  %6 = trunc i64 %5 to i32
  %7 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %5
  store i32 %6, i32* %7, align 4
  %8 = add nuw nsw i64 %5, 1
  %9 = trunc i64 %8 to i32
  %10 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %8
  store i32 %9, i32* %10, align 4
  %11 = add nuw nsw i64 %8, 1
  %12 = trunc i64 %11 to i32
  %13 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %11
  store i32 %12, i32* %13, align 4
  %14 = add nuw nsw i64 %11, 1
  %15 = icmp ult i64 %14, 1024
  br i1 %15, label %2, label %.preheader

16:                                               ; preds = %.lr.ph, %16
  %.11 = phi i64 [ 0, %.lr.ph ], [ %40, %16 ]
  %17 = sub nuw nsw i64 1024, %.11
  %18 = sub nuw nsw i64 %17, 1
  %19 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %18
  %20 = load i32, i32* %19, align 4
  %21 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %.11
  store i32 %20, i32* %21, align 4
  %22 = add nuw nsw i64 %.11, 1
  %23 = sub nuw nsw i64 1024, %22
  %24 = sub nuw nsw i64 %23, 1
  %25 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %24
  %26 = load i32, i32* %25, align 4
  %27 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %22
  store i32 %26, i32* %27, align 4
  %28 = add nuw nsw i64 %22, 1
  %29 = sub nuw nsw i64 1024, %28
  %30 = sub nuw nsw i64 %29, 1
  %31 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %30
```

```
  %32 = load i32, i32* %31, align 4
  %33 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %28
  store i32 %32, i32* %33, align 4
  %34 = add nuw nsw i64 %28, 1
  %35 = sub nuw nsw i64 1024, %34
  %36 = sub nuw nsw i64 %35, 1
  %37 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %36
  %38 = load i32, i32* %37, align 4
  %39 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %34
  store i32 %38, i32* %39, align 4
  %40 = add nuw nsw i64 %34, 1
  %41 = icmp ult i64 %40, 512
  br i1 %41, label %16, label %._crit_edge

._crit_edge:                                      ; preds = %16
  br label %42

42:                                               ; preds =
%.preheader._crit_edge, %._crit_edge
  ret i32 0
}
```

2. LLVM IR output

```
; ModuleID = 'test-ir/input2_opt.ll'
source_filename = "input2.ll"

define i32 @main() {
  %1 = alloca [1024 x i32], align 16
  br label %2

.preheader:                                       ; preds = %2
  br i1 true, label %.lr.ph, label %.preheader._crit_edge

.preheader._crit_edge:                            ; preds = %.preheader
  br label %58

.lr.ph:                                           ; preds = %.preheader
  br label %24

2:                                                ; preds = %2, %0
```

```
  %.02 = phi i64 [ 0, %0 ], [ %22, %2 ]
  %3 = trunc i64 %.02 to i32
  %4 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %.02
  %5 = add nuw nsw i64 %.02, 1
  %6 = trunc i64 %5 to i32
  %7 = bitcast i32* %4 to i64*
  %8 = zext i32 %3 to i64
  %9 = zext i32 %6 to i64
  %10 = mul i64 %9, 4294967296
  %11 = or i64 %10, %8
  store i64 %11, i64* %7
  %12 = add nuw nsw i64 %5, 1
  %13 = trunc i64 %12 to i32
  %14 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %12
  %15 = add nuw nsw i64 %12, 1
  %16 = trunc i64 %15 to i32
  %17 = bitcast i32* %14 to i64*
  %18 = zext i32 %13 to i64
  %19 = zext i32 %16 to i64
  %20 = mul i64 %19, 4294967296
  %21 = or i64 %20, %18
  store i64 %21, i64* %17
  %22 = add nuw nsw i64 %15, 1
  %23 = icmp ult i64 %22, 1024
  br i1 %23, label %2, label %.preheader

24:                                               ; preds = %24, %.lr.ph
  %.11 = phi i64 [ 0, %.lr.ph ], [ %56, %24 ]
  %25 = sub nuw nsw i64 1024, %.11
  %26 = sub nuw nsw i64 %25, 1
  %27 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %26
  %28 = load i32, i32* %27, align 4
  %29 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %.11
  %30 = add nuw nsw i64 %.11, 1
  %31 = sub nuw nsw i64 1024, %30
  %32 = sub nuw nsw i64 %31, 1
  %33 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %32
  %34 = load i32, i32* %33, align 4
  %35 = bitcast i32* %29 to i64*
  %36 = zext i32 %28 to i64
```

```
  %37 = zext i32 %34 to i64
  %38 = mul i64 %37, 4294967296
  %39 = or i64 %38, %36
  store i64 %39, i64* %35
  %40 = add nuw nsw i64 %30, 1
  %41 = sub nuw nsw i64 1024, %40
  %42 = sub nuw nsw i64 %41, 1
  %43 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %42
  %44 = load i32, i32* %43, align 4
  %45 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %40
  %46 = add nuw nsw i64 %40, 1
  %47 = sub nuw nsw i64 1024, %46
  %48 = sub nuw nsw i64 %47, 1
  %49 = getelementptr inbounds [1024 x i32], [1024 x i32]* %1, i64 0, i64 %48
  %50 = load i32, i32* %49, align 4
  %51 = bitcast i32* %45 to i64*
  %52 = zext i32 %44 to i64
  %53 = zext i32 %50 to i64
  %54 = mul i64 %53, 4294967296
  %55 = or i64 %54, %52
  store i64 %55, i64* %51
  %56 = add nuw nsw i64 %46, 1
  %57 = icmp ult i64 %56, 512
  br i1 %57, label %24, label %._crit_edge

._crit_edge:                                      ; preds = %24
  br label %58

58:                                               ; preds = %._crit_edge,
%.preheader._crit_edge
  ret i32 0
}
```

1. LLVM IR input

```
; ModuleID = 'input3.ll'
source_filename = "input3.ll"

define void @_Z1fv() {
  %1 = alloca [100 x i16], align 16
  br label %2
```

```
2:                                                ; preds = %0, %2
  %.01 = phi i64 [ 0, %0 ], [ %18, %2 ]
  %3 = urem i64 %.01, 3
  %4 = trunc i64 %3 to i16
  %5 = getelementptr inbounds [100 x i16], [100 x i16]* %1, i64 0, i64 %.01
  store i16 %4, i16* %5, align 2
  %6 = add nuw nsw i64 %.01, 1
  %7 = urem i64 %6, 3
  %8 = trunc i64 %7 to i16
  %9 = getelementptr inbounds [100 x i16], [100 x i16]* %1, i64 0, i64 %6
  store i16 %8, i16* %9, align 2
  %10 = add nuw nsw i64 %6, 1
  %11 = urem i64 %10, 3
  %12 = trunc i64 %11 to i16
  %13 = getelementptr inbounds [100 x i16], [100 x i16]* %1, i64 0, i64 %10
  store i16 %12, i16* %13, align 2
  %14 = add nuw nsw i64 %10, 1
  %15 = urem i64 %14, 3
  %16 = trunc i64 %15 to i16
  %17 = getelementptr inbounds [100 x i16], [100 x i16]* %1, i64 0, i64 %14
  store i16 %16, i16* %17, align 2
  %18 = add nuw nsw i64 %14, 1
  %19 = icmp ult i64 %18, 100
  br i1 %19, label %2, label %20

20:                                               ; preds = %2
  ret void
}
```

2. LLVM IR output

```
; ModuleID = 'test-ir/input3_opt.ll'
source_filename = "input3.ll"

define void @_Z1fv() {
  %1 = alloca [100 x i16], align 16
  br label %2

2:                                                ; preds = %2, %0
  %.01 = phi i64 [ 0, %0 ], [ %26, %2 ]
  %3 = urem i64 %.01, 3
```

```
  %4 = trunc i64 %3 to i16
  %5 = getelementptr inbounds [100 x i16], [100 x i16]* %1, i64 0, i64 %.01
  %6 = add nuw nsw i64 %.01, 1
  %7 = urem i64 %6, 3
  %8 = trunc i64 %7 to i16
  %9 = bitcast i16* %5 to i32*
  %10 = zext i16 %4 to i32
  %11 = zext i16 %8 to i32
  %12 = mul i32 %11, 65536
  %13 = or i32 %12, %10
  store i32 %13, i32* %9
  %14 = add nuw nsw i64 %6, 1
  %15 = urem i64 %14, 3
  %16 = trunc i64 %15 to i16
  %17 = getelementptr inbounds [100 x i16], [100 x i16]* %1, i64 0, i64 %14
  %18 = add nuw nsw i64 %14, 1
  %19 = urem i64 %18, 3
  %20 = trunc i64 %19 to i16
  %21 = bitcast i16* %17 to i32*
  %22 = zext i16 %16 to i32
  %23 = zext i16 %20 to i32
  %24 = mul i32 %23, 65536
  %25 = or i32 %24, %22
  store i32 %25, i32* %21
  %26 = add nuw nsw i64 %18, 1
  %27 = icmp ult i64 %26, 100
  br i1 %27, label %2, label %28

28:                                               ; preds = %2
  ret void
}
```

The pass made correct answer to all of given test codes. However, more cases are to be considered to validate cases.

10. Liveness Analysis + Register Coloring (이진우)

10-1. Basic description of the task

This task aims to build the Liveness Analysis module. It dynamically determines the live range of all the registers, and performs $O(L \cdot N^2)$ analysis to

search through an overlapping pair among all N IR registers and in code length of L. This analysis is globally done, however it is ensured that registers in different functions are considered to be not overlapping in any cases. i.e. it is a FunctionPass wrapped within the ModulePass.

Also, we posit to provide register coloring. Because of the topological property of interference graphs in SSA program, we can ensure P time(Currently $O(LV^2)$) in coloring registers using the lexicographic BFS which finds the Perfect Elimination Order, and the greedy coloring.

10-2. Implementation details changed during the *Dev. sprint*

All features intended to be developed are developed. However, the algorithm first suggested for finding live intervals of value is changed. First we posited the algorithm which starts from the definition and search to find all uses, but now we start from the use and climb up to the definition in reverse order of execution. Since a definition dominates its uses in SSA programs, we ensure that every path starting from a user reaches the definition, a place to stop.

Also, some changes involving the plan in further sprints were made, We figured out that the original plan for task 11 in sprint 2, which allows to exploit the computable registers as a function argument, involves a module-level NP time of coloring which violates our original purpose. So we discarded the task 11 planned, and also removed the feature to support it. Now the pass does not color the arguments, since it is ensured in our system that they do not overlap with computable registers. Also, RegisterGraph class does NOT have a interface to coallocate registers which is requested. This actually makes the pass Function-local, however because of time we did not modify the signature of the LivenessAnalysisPass as a FunctionPass. Instead, we inserted some useful interfaces to receive valuable results by the function pointer.

Finally, several modifications are made in deciding which instructions to ignore(*i.e.* do not allocate register). *alloca, GEP*s and *bitcast, ptrtoint* operations are not valid assembly operations; so they must not be grouped in any color to assign and exploit the registers properly. The current list of them is defined in DO_NOT_CONSIDER set defined in LivenessAnalysis.h header. The consequences of other passes should be continuously examined to determine which IR instructions are to be excluded in the coloring procedure.

10-3. Test results & Improvements

Five tests are done to validate the behavior of the pass.

1. LLVM IR input

```
define i32 @main() #0 {
entry:
  %r0 = alloca [10 x i32], align 16
  br label %BB1
BB1:
  %r1 = phi i32 [ 0, %entry ], [ %r6, %BB3 ]
  %r2 = icmp slt i32 %r1, 5
  br i1 %r2, label %BB2, label %BB4
BB2:
  %r3 = sub nsw i32 %r1, 2
  %r4 = sext i32 %r1 to i64
  %r5 = getelementptr inbounds [10 x i32], [10 x i32]* %r0, i64 0, i64 %r4
  store i32 %r3, i32* %r5, align 4
  br label %BB3
BB3:
  %r6 = add nsw i32 %r1, 1
  br label %BB1
BB4:
  %r7 = add nsw i32 5, 3
  %r8 = ashr i32 %r7, 1
  %r9 = sext i32 %r8 to i64
  %r10 = getelementptr inbounds [10 x i32], [10 x i32]* %r0, i64 0, i64 %r9
  %r11 = load i32, i32* %r10, align 4
  ret i32 %r11
}
```

2. Register coloring result

```
main 3 colors needed
color 0 : r11 r2 r4 r9 r5 r6 r7 r8 r10
color 1 : r3
color 2 : r1
```

1. LLVM IR input

```
define i32 @main() #0 {
entry:
  br label %BB1
BB1:
```

```
  %r0 = phi i32 [ 16, %entry ], [ %r8, %BB5 ]
  %r1 = phi i32 [ 0, %entry ], [ %r9, %BB5 ]
  %r2 = icmp sgt i32 %r0, 1
  br i1 %r2, label %BB2, label %BB6
BB2:
  %r3 = srem i32 %r0, 2
  %r4 = icmp eq i32 %r3, 0
  br i1 %r4, label %BB3, label %BB4
BB3:
  %r5 = sdiv i32 %r0, 2
  br label %BB5
BB4:
  %r6 = mul nsw i32 %r0, 3
  %r7 = add nsw i32 %r6, 1
  br label %BB5
BB5:
  %r8 = phi i32 [ %r5, %BB3 ], [ %r7, %BB4 ]
  %r9 = add nsw i32 %r1, 1
  br label %BB1
BB6:
  ret i32 %r1
}
```

2. Register coloring result

```
main 5 colors needed
color 0 : r6 r4 r3 r2 r9
color 1 : r8 r0
color 2 : r7
color 3 : r5
color 4 : r1
```

1. LLVM IR input

```
define i32 @main() {
entry:
  %r0 = add i32 10, 5
  %r1 = add i32 3, 4
  %r2 = call i32 @foo(i32 %r0, i32 %r1)
  %r3 = urem i32 %r2, 2
  %r4 = icmp eq i32 %r3, 0
  br i1 %r4, label %BB1, label %BB2
```

```
BB1:
  ret i32 0
BB2:
  ret i32 %r2
}

define i32 @foo(i32 %r5, i32 %r6) {
  %r7 = add i32 %r5, %r6
  %r8 = mul i32 %r5, %r6
  %r9 = sub i32 %r8, %r7
  ret i32 %r9
}
```

2. Register coloring result

```
main 2 colors needed
color 0 : r0 r3 r4
color 1 : r1 r2

foo 2 colors needed
color 0 : r9 r8
color 1 : r7
```

1. LLVM IR input

```
define i32 @main(i32 %n) {
entry:
  %r0 = icmp eq i32 %n, 0
  %r1 = icmp eq i32 %n, 1
  %r2 = or i1 %r0, %r1
  br i1 %r2, label %ret1, label %BB1
ret1:
  ret i32 1
BB1:
  %r3 = sub i32 %n, 1
  %r4 = call i32 @main(i32 %r3)
  %r5 = sub i32 %n, 2
  %r6 = call i32 @main(i32 %r5)
  %r7 = add i32 %r4, %r6
  ret i32 %r7
}
```

2. Register coloring result

```
main 2 colors needed
color 0 : r5 r6 r7 r3 r2 r0
color 1 : r4 r1
```

1. LLVM IR input

```
define i32 @main() {
  %a0 = add i32 3, 5
  %b0 = add i32 4, 2
  %c0 = add i32 8, 3
  %d0 = add i32 6, 5
  %e0 = call i32 @foo(i32 %a0, i32 %b0, i32 %c0, i32 %d0)
  %f0 = call i32 @bar(i32 %a0, i32 %c0, i32 %b0, i32 %d0)
  %g0 = add i32 %e0, %f0
  ret i32 %g0
}

define i32 @foo(i32 %a1, i32 %b1, i32 %c1, i32 %d1) {
  %e1 = add i32 %a1, %b1
  %f1 = add i32 %c1, %d1
  %g1 = add i32 %e1, %f1
  %h1 = call i32 @bar(i32 %a1, i32 %b1, i32 %c1, i32 %d1)
  %i1 = add i32 %g1, %h1
  ret i32 %i1
}

define i32 @bar(i32 %a2, i32 %b2, i32 %c2, i32 %d2) {
  %e2 = mul i32 %a2, %b2
  %f2 = mul i32 %c2, %d2
  %g2 = mul i32 %e2, %f2
  ret i32 %g2
}
```

2. Register coloring result

```
main 5 colors needed
color 0 : f0 d0 g0
color 1 : e0
color 2 : c0
color 3 : a0
```

```
color 4 : b0

foo 2 colors needed
color 0 : e1 h1 i1
color 1 : f1 g1

bar 2 colors needed
color 0 : f2 g2
color 1 : e2
```

Results are proved by hand calculation of the reviewers and the author together. The algorithms and its implementations are proved to be mathematically accurate.

The calculation speed of PerfectEliminationOrdering() can be improved using different implementation of disjoint sets. Currently we use std::set which is a tree based structure. Also, we have to linear search the sets to find the desired value. This increases the time complexity higher than the union find implementation. However we did not have enough time to fix the implementation since the standard library does not provides the union set.