[SWPP 202001 Project]

Team 1 Requirement & Specification Document

Team 1(이진우, 정덕인, 김도현, 임지규)

0. Project plan summary

1. GV Optimization & Other global-level preprocessing
   - (InterProcedural) Sparse Conditional Constant Propagation + Simplify CFG
   - GV optimizer
   - Dead Global Elimination
   - GV to malloc in main
   - Mem2Reg
   - Global Value Numbering
2. Function Call Optimization
   - Function Merging
   - Tail Call Elimination
   - Function Inline
3. Memory Operation Optimization
   - Vectorize Load/Store
   3-1. Stack/Heap Optimizations
      - Heap to Stack Lowering
      - Interprocedural Alloca Sinking
      - Stack/Heap Access Grouping
      - malloc in main to GV + Dead Argument Elimination
      - Mem2Reg
4. Computative Instruction Optimization Pipeline
   - Reassociate
   - Constant Propagation
   - Combine Instructions
   - Code Sinking
   - SCCP + Simplify CFG
5. Loop Optimization Pipelines
   5-1. LICM&Roting&Unswitching
      - Reassociate
      - Loop Simplify
      - Loop-Closed SSA Form(Terminal phi node)
      - Loop Invariant Code Motion

- Loop Rotate
- Loop Unswitching

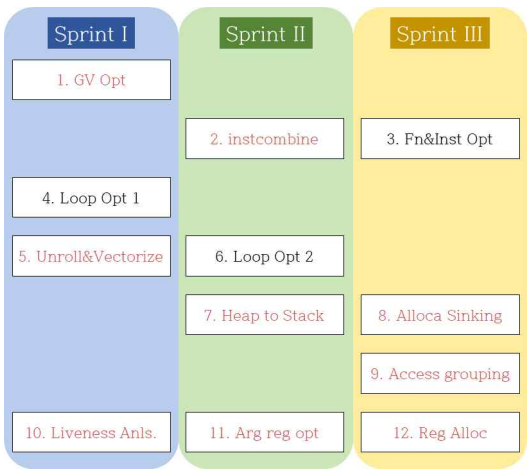
5-2. Loop vectorization

- Reassociate
- Loop Simplify
- Canonicalize induction variables
- Loop Unroll and Jam + *Vectorize Load & Store*

6. Backend Optimization

- Liveness Analysis
- Direct Access to Parent Local Registers(Parameter Reduction)
- Register Allocation
- LLVM IR to Assembly Static Translation

1	GV ⇔ malloc() + Embedding GV optimization
2	Modifying Combine Insturction
3	Embedding Function opt. + Instruction opt.
4	Embedding Loop Opt Pipeline 1
5	Loop unroll + Vectorized load&store
6	Embedding Loop Opt Pipeline 2
7	Heap to Stack&Argument lowering
8	Interprocedural Alloca Sinking
9	Heap/Stack grouping
10	Liveness Analysis + Translation
11	Direct Access to Parent Local Registers
12	Register Allocation



I. Sprint I

1. GV to malloc in main & malloc in main to GV

1-1. 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. If the time is enough, we also aim to implement this pass also in this sprint.

1-2. GV to malloc in main

1-2-1. Algorithm and Pseudocode

This task is to change Global variable to malloc heap allocation. In llvm, both of the declaration of GV and heap allocated variable returns the pointer of the variable. This means What we have to do is getting the GV and make new malloc variable set to the size of the original GV. And finally change all uses of the GV to new variable In main function. But In case of other functions which used the original GV, it is necessary to give same variables to the functions. According to our spec, main function always exists and it is starting function. So after we make new malloc heap allocated variable, we have to give those variables to functions as arguments. And later we will eliminate the unused arguments from functions by using passes like dead argument elimination.

The following is the pseudocode of this algorithm.

```
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)
```

To summarize the effect, these consequences can be observed in the source IR program.

1. Global value are replaced into malloc function inside the main.
2. The malloc variable was put as the argument of function foo.(this means other

functions except main function in real code.)

3. The use of the original GV is changed to the new malloc variable pointer.

It might be dangerous to increase the number of arguments a lot because our specification restricts the maximum number of arguments to 16. There are three significant steps which reduce the number of function arguments.

1. Interprocedural Alloca Sinking: sinks the malloc/alloca into subprocedure if the alloca is only used as a parameter in the function and is not passed to multiple functions. Calling Dead Argument Elimination(DAE) will easily reduce the function arguments.
2. malloc to GV pass: This pass(specifications written in 1-3) implements the exact inverse procedure of this pass. This transforms mallocs in main to GV, so that function arguments containing the malloc instruction in main() is not used anywhere. DAE pass will be applied to remove redundant dummy arguments.
3. Direct Access to Parent Local Register pass: this pass from the backend side moves arguments into normal registers. Invariant values like pointers are likely to sent through arg registers while other registers ride on normal ones.

1-2-2. IR transformation examples

1. LLVM IR input
@GV = global i32 1, align 4 define i32 @main() #0 {

```
  %1 = alloca i32, align 4
  store i32 0, i32* %1, 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

```
define i32 @main() #0 {
  %GV = call i32* @malloc(i64 4)
  store i32 1, i32* %GV
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 @foo(i32* %GV)
  ret i32 %2
}

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

1. LLVM IR input

```
@GV = global i32 2, align 4

define i32 @main() #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

```
define i32 @main() #0 {
  %GV = call i32* @malloc(i64 4)
  store i32 2, i32* %GV
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %3 = call i32 @foo(i32* %GV)
  store i32 %3, i32* %2, align 4
  %4 = load i32, i32* %GV, align 4  // @a -> %a
  %5 = load i32, i32* %2, align 4
  %6 = add nsw i32 %4, %5
  ret i32 %6
}

define i32 @foo(i32* %GV) #0 {  // add argument %a
  %1 = load i32, i32* %GV, align 4  // @GV -> %GV
  %2 = load i32, i32* %GV, align 4  // @GV -> %GV
  %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

```
define i32 @main() #0 {
  %GV = call i32* @malloc(i64 4)
  store i32 2, i32* %GV
  %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
}
```

1-3. malloc in main to GV

1-3-1. Algorithm and Pseudocode

After the GV to malloc pass is executed and optimizations like DAE, stack lowering happen, some malloc operations will still reside in the main function. For instance, allocating a large array in the stack may cause stack overflow, so it should remain on the heap memory. Since we pass transformed GV to all functions by parameters, it may be better to not pass GV and just treat them as global variables. If the malloc to GV pass is excuted, every malloc operation found in main() function are transformed into GV.

```
for mallocInst instM in main:
  create GV
  instS = find StoreInst(instM, (initial value)) in main
  perform Memory Alias Analysis with instS globally
  for all mem. inst. that alias with instS(c.f. loads from mallocInst):
    change ptr ⇒ GV
  add StoreInst(GV, (initial value)) to main
  delete instM, instS

Afterwards:
perform Dead Argument Elimination
```

This pass should be a exact inverse of the GV to malloc pass, so LLVM IR examples are omitted.

4. Verifying Loop Optimization Pipeline 1(김도현)

This chunk is to embed the loop optimization pipeline of LLVM standard to our project. The sequence of existing passes which we will bring from LLVM is as following. The order here is the same as the order we are going to apply. Dependencies between passes are found from various sources; LLVM documentations, clang -O1 tests, and the comments in the source code.

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

These optimizations reduce complexity inside the loop by moving invariant values outside the loop(LICM), restructuring the loop into if-nested do-while loops(Loop Rotate), or reducing the branch instruction in the loop(Loop Unswitching). The task is to create a pass which wraps all these passes, so a single run through the pass will optimize loops as desired. Also, we should ensure that no invalid IR components such as intrinsics appear after the optimization.

5. Loop unrolling + Vectorize Load & Stores(정덕인)

5-1. Basic Description of the task

The pass implemented here receives the loop-unrolled functions, 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 that consequent 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.

5-2. Algorithm and Pseudocode

As we mentioned before, we will use int64 type to vectorize. One 64-bit register can save 2 32-bit registers, 4 16-bit registers, and 8 8-bit registers. According to this, if we unroll loop in bunches of 8, we can handle every case easily. For example, if target IR has loop with consequent memory access by 16-bit, then it can be unrolled in bunches of 8 and every bunch of loop would

use two 64-bit registers, which have 4 16-bit values individually.

We can modify the parameters of the existing Loop Unrolling pass in LLVM standard to unroll loops by 8. Also, unrolling loops that number of iterations are not determined in the compile time is initially inactivated in the LLVM standard loop unrolling because it consumes more time while compiling. However, we will force the pass to perform unrolling even in these cases.

This is the pseudocode of this algorithm below.

```
Prerequisites:
all loop optimizers until Loop unrolling performed
Combine instructions performed(for easy comparing of GEP offsets)
  ⇒ (must perform add combination)

VectorizeLoadNStore::run():
set sequentialGEPs
for BB in Function
  if 8 GEP instructions are sequential:
    sequentialGEPs.add( { 8 GEP instructions GEP0~7} )
  for GEP0~7 in sequentialGEPs:
    for usage0~7 w.r.t. GEP0~7:
      type := inst.getType()
      if type == i32 then
        groupAndSplit<2>(usage0, usage1)
        groupAndSplit<2>(usage2, usage3)
        groupAndSplit<2>(usage4, usage5)
        groupAndSplit<2>(usage6, usage7)
      else if type == i16 then
        groupAndSplit<4>(usage0, usage1, usage2, usage3)
        groupAndSplit<4>(usage4, usage5, usage6, usage7)
      else if type == i8 then
        groupAndSplit<8>(usage0, usage1, usage2, usage3, usage4,
usage5, usage6, usage7)

groupAndSplit() performs the splitting of operations using urem+udiv /
or+umul.
- e.g. If 'load' operations comes in:
Adds a i64 load and spares it before the first usage, and distributes all
values to new registers. usages(loads) are all swapped to these new
distributed values.
```

First, LoopUnrollPass from LLVM standard is included to unroll the loop. We will assign a value to LoopUnrollOptions, so we can unroll loop with unknown

iteration. After loop unrolling, we check if there are any sequential loads inside the unrolled loop body. By the effects of loop simplification and unswitching, most body codes are going to be written in one basic block, so we restrict the searching scope into a basic block. Sequential memory accesses are defined as sequential if two pointers(should be *getelementptr*, the GEP instruction) are sequential. Two GEP instructions are considered to be sequential if,

- Types of *getelementptr* are equal.
- Base address pointers are equal.
- Offset instruction has 1 difference.

  e.g.) "%x = i32.." and "add i32 %x, 1" has 1 difference.

  "add i32 %x, 2" and "add i32 %x, 3" has 1 difference.

If there exists 8(=unroll factor) sequential GEPs in total, they are grouped and their uses are transformed into vectorized memory accesses.

5-3. IR transformation examples

We supposed unroll by 8 bunches, but this leads to unrealistic length of code to show here. Because of this reason, following examples are slightly different(unroll factor = 4, not 8) as we mentioned above.

```
define void @_Z1fv() {
  %1 = alloca [5 x i32], align 16
  %2 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  br label %3

3:
  %4 = load i32, i32* %2, align 4
  %5 = icmp slt i32 %4, 5
  br i1 %5, label %6, label %16

6:
  %7 = load i32, i32* %2, align 4
  %8 = load i32, i32* %2, align 4
  %9 = mul nsw i32 %7, %8
  %10 = load i32, i32* %2, align 4
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 %11
  store i32 %9, i32* %12, align 4
  br label %13
```

```
define void @_Z1fv() {
  %1 = alloca [5 x i32], align 16
  %2 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 0
  %3 = mul nsw i64 0, 4294967296
  %4 = add nsw i64 %3, 1
  store i64 %4, i32* %2, align 8
  %5 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 2
  %6 = mul nsw i64 4, 4294967296
  %7 = add nsw i64 %6, 9
  store i64 %7, i32* %5, align 8
  %8 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 4
  store i32 16, i32* %8, align 4
  ret void
}
```

```
13:
  %14 = load i32, i32* %2, align 4
  %15 = add nsw i32 %14, 1
  store i32 %15, i32* %2, align 4
  br label %3

16:
  ret void
}
```

define i32 @_Z1fv() { %1 = alloca [100 x i32], align 16 br label %2 2: %.04 = phi i32 [0, %0], [%6, %2] %3 = mul nsw i32 %.04, %.04 %4 = sext i32 %.04 to i64 %5 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %4 store i32 %3, i32* %5, align 4 %6 = add nuw nsw i32 %.04, 1 %7 = icmp ult i32 %6, 100 br i1 %7, label %2, label %.preheader .preheader: %.13 = phi i32 [%12, %.preheader], [0, %2] %.012 = phi i32 [%11, %.preheader], [0, %2] %8 = sext i32 %.13 to i64 %9 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %8 %10 = load i32, i32* %9, align 4 %11 = add nsw i32 %.012, %10 %12 = add nuw nsw i32 %.13, 1 %13 = icmp ult i32 %12, 100 br i1 %13, label %.preheader, label %14 14: %.01.lcssa = phi i32 [%11,	define i32 @_Z1fv() { %1 = alloca [100 x i32], align 16 br label %2 2: %.04 = phi i32 [0, %0], [%22, %2] %3 = mul nsw i32 %.04, %.04 %4 = sext i32 %.04 to i64 %5 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %4 %6 = add nuw nsw i32 %.04, 1 %7 = mul nsw i32 %6, %6 %8 = sext i32 %3 to i64 %9 = mul nsw i64 %8, 4294967296 %10 = sext i32 %7 to i64 %11 = add nuw nsw i64 %9, %10 store i64 %11, i64* %5, align 8 %12 = add nuw nsw i32 %6, 1 %13 = mul nsw i32 %12, %12 %14 = sext i32 %12 to i64 %15 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %14 %16 = add nuw nsw i32 %12, 1 %17 = mul nsw i32 %16, %16 %18 = sext i32 %13 to i64 %19 = mul nsw i64 %18, 4294967296 %20 = sext i32 %17 to i64 %21 = add nuw nsw i64 %18, %20 store i64 %21, i64* %15, align 8 %22 = add nuw nsw i32 %16, 1 %23 = icmp ult i32 %22, 100

%.preheader] ret i32 %.01.lcssa }	br i1 %23, label %2, label %.preheader.preheader .preheader.preheader: br label %.preheader .preheader: %.13 = phi i32 [0, %.preheader.preheader], [%39, %.preheader] %.012 = phi i32 [0, %.preheader.preheader], [%38, %.preheader] %24 = sext i32 %.13 to i64 %25 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %24 %26 = load i64, i64* %25, align 8 %27 = rem i32 %26, 4294967296 %28 = div i32 %26, 4294967296 %29 = rem i32 %28, 4294967296 %30 = add nsw i32 %.012, %27 %31 = add nsw i32 %30, %29 %32 = add nuw nsw i32 %.13, 2 %33 = sext i32 %32 to i64 %34 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %33 %35 = load i64, i64* %34, align 8 %36 = rem i32 %35, 4294967296 %37 = div i32 %36, 4294967296 %38 = rem i32 %37, 4294967296 %39 = add nsw i32 %31, %36 %40 = add nsw i32 %39, %38 %41 = add nuw nsw i32 %32, 2 %42 = icmp ult i32 %41, 100 br i1 %42, label %.preheader, label %43 43: %.01.lcssa = phi i32 [%38, %.preheader] ret i32 %.01.lcssa }

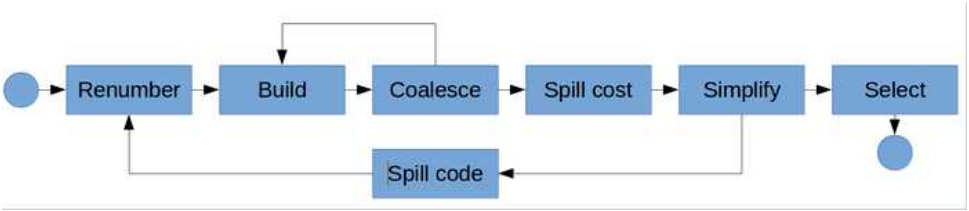
10. Liveness Analysis(이진우)

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. This assumption will break by the Direct Access to Parent Local Register pass (DAPLR) which is going to be implemented in sprint II.

Also, we plan to implement the interface that receives the result of Liveness Analysis and the pass for sprint II and generates the symbol table by graph coloring. According to Briggs, Cooper, & Torczon 1992, this tasks covers the Renumber~Coalesce in the whole register allocation procedure.

Several liveness analysis pass with LLVM IR are present online, however for the compatibility with the DAPLR pass, we decided to implement it again.



10-2. Interference Graph Construction

10-2-1 Algorithm & Pseudocode

All N symbols(arguments and instructions, but not labels) are mapped by a unsigned integer. All L instructions in the target LLVM source are mapped to the boolean array length of N. For each N symbols, the pass recursively traces every usage of it from the definition and marks the flag array true if the variable is alive. Postorder search is used to track the liveness in the measure of instructions, not basic blocks. The results are converted into the graph form(N*N adjacency matrix) and all results are preserved.

Pseudocode of this analysis pass can be written as follows.

```
LivenessAnalysis::run()

  //Counts all symbols
  map<unsigned int, argument+instructions> symbols
  N = 0
```

```
  for symbol in arguments + instructions:
    symbols[N++] = symbol
  //Recursively searches and marks liveness
  map liveness: instruction ↦ bool[N]
  for i in range(N):
    RecursiveLiveIntervalSearch(i, symbols[i], symbols, liveness)
  //Turn the result into graph
  bool[][] adjacencyMatrix
  for I in instructions:
    overlap = Liveness[instruction]
    for (i, j) from (0, 0) to (N, N), i < j:
      if liveness[I][i] && liveness[I][j] :
        adjacencyMatrix[i][j] = adjacencyMatrix[j][i] = true

LivenessAnalysis::RecursiveLiveIntervalSearch(curr, inst, &symbol, &liveness)

  currBlock = inst.parent
  boolean isAlive = false //is symbol[curr] alive in the current line?
  for succ in currBlock.dominatedSuccessors:
    //If symbol[curr] is alive in single descendant, isAlive is true
    isAlive = isAlive || LivenessAnalysis(curr, succ.firstInst, symbol, liveness)
  //When succ is used in the phi node, it should be preserved
  for succ in currBlock.successors(not dominated):
    if any succ.phiNode uses symbol[curr]:
      isAlive = true
  for I from currBlock.terminator to inst:
    if I uses symbol[curr]:
      isAlive = true
    if isAlive && I!=succ[curr]:
      symbol[I][curr] = true

  return isAlive
```

I!=succ[curr] condition is required to allow the user and the used to be overlapped. For instance, two registers '%x = load %ptr', '%y = add %x, 5' can be allocated together if %x is not reused afterwards so %x is dead and %y is alive in further instructions.

The pass only searches through the dominated blocks because the use of an instruction could not be dominated by itself if and only if when the value is brought with Phi instructions. Operands of Phi instructions are treated dead in the parent block of Phi. so coallocating the operand and the Phi instruction is

possible. This would much reduce register accumulation in loops.

Phi instruction will be removed(a.k.a SSA elimination) after the symbol table is completed by adding the swap instruction while translating.

We are not sure if this solution is mathematically flawless in every situation. Unit tests should be given to verify the RecursiveLive IntervalSearch.

10-2-2. LLVM IR examples

Liveness interval analysis results w.r.t. each LLVM IR examples are presented below. To be precise, the result of Liveness interval analysis is formatted as one-hot encoding, but to fit the space it was represented as an simple list which includes a register iff it is marked true in the original result.

When two registers are present(marked true) in the same array, they are considered to be connected in the Interference graph as each are pushed into each others' adjacency lists. This stage is rather obvious, so it is omitted in the following visualizations.

1. LLVM IR input

```
@GV = global i32 2, align 4

define i32 @main() #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. Liveness Intervals(brackets at the end of line indicates live variables)

```
@GV = global i32 2, align 4
```

```
define i32 @main() #0 {
  %1 = alloca i32, align 4                    ;[]
  %2 = alloca i32, align 4                    ;[1, ]
  store i32 0, i32* %1, align 4               ;[1, 2]
  %3 = call i32 @foo()                        ;[2]
  store i32 %3, i32* %2, align 4              ;[2, 3]
  %4 = load i32, i32* @GV, align 4            ;[2]
  %5 = load i32, i32* %2, align 4             ;[2, 4]
  %6 = add nsw i32 %4, %5                     ;[4, 5]
  ret i32 %6                                  ;[6]
}

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

1. LLVM IR input

```
define i32 @main() #0 {
  %1 = alloca [10 x i32], align 16
  br label %2
2:                                                ; preds = %8, %0
  %.0 = phi i32 [ 0, %0 ], [ %9, %8 ]
  %3 = icmp slt i32 %.0, 5
  br i1 %3, label %4, label %10
4:                                                ; preds = %2
  %5 = sub nsw i32 %.0, 2
  %6 = sext i32 %.0 to i64
  %7 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i64 0, i64 %6
  store i32 %5, i32* %7, align 4
  br label %8
8:                                                ; preds = %4
  %9 = add nsw i32 %.0, 1
  br label %2
10:                                               ; preds = %2
  %11 = add nsw i32 5, 3
  %12 = ashr i32 %11, 1
  %13 = sext i32 %12 to i64
  %14 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i64 0, i64 %13
```

```
  %15 = load i32, i32* %14, align 4
  ret i32 %15
}
```

2. Liveness Intervals(brackets at the end of line indicates live variables)

```
define i32 @main() #0 {
  %1 = alloca [10 x i32], align 16                    ;[]
  br label %2
2:                                                    ; preds = %8, %0
  %.0 = phi i32 [ 0, %0 ], [ %9, %8 ]                ;[1]
  %3 = icmp slt i32 %.0, 5                            ;[1, .0]
  br i1 %3, label %4, label %10                       ;[1, .0, 3]
4:                                                    ; preds = %2
  %5 = sub nsw i32 %.0, 2                             ;[1, .0]
  %6 = sext i32 %.0 to i64                            ;[1, .0, 5]
  %7 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i64 0, i64 %6
                                                      ;[1, .0, 5, 6]
  store i32 %5, i32* %7, align 4                      ;[.0, 5, 7]
  br label %8                                         ;[.0]
8:                                                    ; preds = %4
  %9 = add nsw i32 %.0, 1                             ;[.0]
  br label %2                                         ;[9] //used in phi ⇒ alive
10:                                                   ; preds = %2
  %11 = add nsw i32 5, 3                              ;[1]
  %12 = ashr i32 %11, 1                               ;[1, 11]
  %13 = sext i32 %12 to i64                           ;[1, 12]
  %14 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i64 0, i64 %13
                                                      ;[1, 13]
  %15 = load i32, i32* %14, align 4                   ;[14]
  ret i32 %15                                         ;[15]
}
```

1. LLVM IR input

```
define i32 @main() #0 {
  br label %1
1:                                                    ; preds = %11, %0
  %.01 = phi i32 [ 16, %0 ], [ %.1, %11 ]
  %.0 = phi i32 [ 0, %0 ], [ %12, %11 ]
  %2 = icmp sgt i32 %.01, 1
  br i1 %2, label %3, label %13
```

```
3:                                                ; preds = %1
  %4 = srem i32 %.01, 2
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %8
6:                                                ; preds = %3
  %7 = sdiv i32 %.01, 2
  br label %11
8:                                                ; preds = %3
  %9 = mul nsw i32 %.01, 3
  %10 = add nsw i32 %9, 1
  br label %11
11:                                               ; preds = %8, %6
  %.1 = phi i32 [ %7, %6 ], [ %10, %8 ]
  %12 = add nsw i32 %.0, 1
  br label %1
13:                                               ; preds = %1
  ret i32 %.0
}
```

2. Liveness Intervals(brackets at the end of line indicates live variables)

```
define i32 @main() #0 {
  br label %1                                     ;[]
1:                                                ; preds = %11, %0
  %.01 = phi i32 [ 16, %0 ], [ %.1, %11 ]        ;[]
  %.0 = phi i32 [ 0, %0 ], [ %12, %11 ]          ;[]
  %2 = icmp sgt i32 %.01, 1                       ;[.01, .0]
  br i1 %2, label %3, label %13                   ;[.01, .0, 2]
3:                                                ; preds = %1
  %4 = srem i32 %.01, 2                           ;[.01]
  %5 = icmp eq i32 %4, 0                          ;[.01, 4]
  br i1 %5, label %6, label %8                    ;[.01, 5]
6:                                                ; preds = %3
  %7 = sdiv i32 %.01, 2                           ;[.01]
  br label %11                                    ;[7]
8:                                                ; preds = %3
  %9 = mul nsw i32 %.01, 3                        ;[.01]
  %10 = add nsw i32 %9, 1                         ;[9]
  br label %11                                    ;[10]
11:                                               ; preds = %8, %6
  %.1 = phi i32 [ %7, %6 ], [ %10, %8 ]          ;[]
  %12 = add nsw i32 %.0, 1                        ;[.0, .1]
```

```
  br label %1                                    ;[.1, 12]
13:                                              ; preds = %1
  ret i32 %.0                                    ;[.0]
}
```

10-3. Graph Coloring

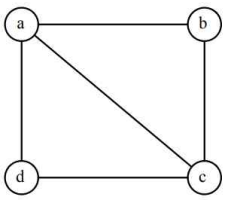
10-3-1. Algorithm

*Most mathematical and algorithmic approach of this section comes from here. this document includes pseudocodes as well as the algorithms, we would like to follow this document's guide.

Graph coloring with minimal set of colors is generally a NP-complete problem, so it would be wise to reduce the compile time with efficient algorithms. We can prove that in SSA form, interference graphs become 'chordal', so that no minimal loop larger than size 3 is formed. This proves that a Simplicial(Perfect) Elimination Ordering exists. Precise definition for SEO is that:

A SEO of the graph G is a permutation of every N vertices(V_1, ..., V_N) where for every V_i and the subgraph G_i of G induced from {V_1, V_2, ..., V_i}, neighbours of V_i in G_i form a clique.

For example, in the graph drawn below, A⇒C⇒B⇒D can be a instance for the SEO of G.



There are various linear time algorithms which find SEO from chordal graphs. Maximum Cardinality Search and Lexicographic BFS are two algorithms available for SEO search.

Also, when the graph has an SEO, it is proved that the graph can be colored with C colors with greedy algorithm in polynomial time where C, chromatic number of the graph, is smaller than (size of the biggest clique) + 1. Greedy algorithm is simple; it colors all nodes in order of the SEO. The color is the smallest number not used in the clique of the vertices' neighbours.

Because these are all well known algorithms and pseudocodes are easily found by web searching, pseudocode is omitted. Also, this coloring part is not a LLVM IR pass but purely algorithmic, so we do not present LLVM IR examples here.