[SWPP 202001 Project]

# Team 1 Planning Document

Team 1(이진우, 정덕인, 김도현, 임지규)

## I. Pipeline for LLVM IR Optimization & Assembler

Blueprint of our optimizer pass pipeline is based on *clang -O1*. However, we removed some transform passes which are not necessary or has insignificant effect in our project specification while adding some passes which are expected to reduce the cost significantly. Custom passes we will introduce in our projects are highlighted pink below. The Basic information of each custom passes are introduced below, while details are described in the 『Requirements and Specification』 document. Simple references for pre-existing passes could be found on llvm.org/docs/Passes.html.

1 ⇒ 2 ⇒ 3-1 ⇒ 4 ⇒ 5-1 ⇒ 4 ⇒ 5-2 ⇒ 4 ⇒ 6

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
2. Function Call Optimization
   - Function Merging
   - Tail Call Elimination
   - Function Inline
3. Memory Operation Optimization
   - Vectorized Load/Store

   3-1. Stack/Heap Optimizations
      - Local Heap Access Elimination
      - Heap to Stack+Argument Lowering
      - Stack/Heap Access Grouping
      - Mem2Reg
4. Calculation Operation Optimization Pipeline
   - Reassociate
   - Constant Propagation

- Combine Instructions
- Code Sinking
- SCCP + Simplify CFG

5. Loop Optimization Pipelines
   5-1. LICM&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
      ⇒ call *Vectorized Load/Store* pass
6. Optimized Assembler
   - Direct Access to Parent Local Registers(Parameter Reduction)
   - Register Usage Analysis
   - Register Allocation

## II. Basic Information about Custom Passes

This section contains the basic description for custom passes we plan to implement in our project,

1. GV to malloc 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.

2. Vectorized Load&Store

*Highly inspired from the presentation done in 04/28 by Anonymous.
This pass 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. This reduces the cost of load&store especially when the data type is small. However, sequential memory operations do not happen frequently in normal codes, so we decided to tie this pass with the Loop Unrolling pass(already present in LLVM standard, but need to be modified for compatibility with custom Vectorizing pass).

3. Local Heap Access Elimination

This pass changes *malloc* into *alloca* or *register* in specific cases. If the usage of *call malloc* instruction is totally local within the scope, we can lower the variable into the stack. Also, if the address is never used, we can even lower the variable into the register. Case-specific criteria will not be given here.

4. Heap to Stack&Argument Lowering

Accessing the heap is not costworthy in most cases, so lowering the variable to stack or register is highly recommended. This pass will refer to the call graph and find the usage of the heap variable across the functions. If the size of the heap allocation is fairly small and the number of any function arguments does not exceed the maximum by this transformation, this pass will bring the heap variable to the local stack of the common antecedents of all user functions and send this variable via arguments. Loss caused by increased function arguments will soon be resolved by the Direct Access to Parent Local Registers pass. Mem2Reg pass should be followed to clean up the *alloca* operations.

5. Stack/Heap Access grouping

This pass will reorder the memory operations to group stack & heap access as much as it can. In result, we can reduce the number of *reset* operation, and also can save cost of head movement.

6. Combine Instructions

This pass is basically a derivation from *instcombine* pass in LLVM. However, project-specific features will be introduced for higher performance. Some examples would be:

- add %x, %x ⇒ mul %x, 2
- and i1 %x, %y ⇒ mul i1 %x, %y
- add %x, 0 ⇒ mul %x, 1 (move instruction)

7. Direct Access to Parent Local Registers

This pass will eventually transform the function parameters to the local register variables. Exploiting the property that *r1~r16* are not discarded but just saved after the function call, functions may freely use the remaining value on the registers. If we use these registers for the arguments, we can save much time in calling & restoring the arguments. This pass is very complicated because the input should be LLVM IR(to modify the function signature easily) while the output should be an assembly code(to assign the same hardware register to the local variable of caller and the callee), but it is not the key component of the assembler. We planned to implement this as an analysis pass which indicates that these registers should be allocated the same, so that the Register Allocation Pass would catch the signal.

c.f. It is pretty obvious that using argX register is slower than this method.

8. Register Usage Analysis + Register Allocation

These passes are for register allocation. The goal is simple; it should minimize the cost spent for the memory operation. It checks how much the register is called in the scope, and rate them proportional to their potential load&store costs. IR registers are then selected to maximize the rate. Two IR registers may be assigned to the same hardware register if they are not interfering each other according to the CFG. Ones that are not assigned return to the stack. DP(Dynamic Programming), heuristic or other approaches could be used to find the (almost) best allocation in reasonable time.