[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 written in red below. The basic information of each custom passes are introduced below, while details will be described in the 『Requirements and Specification』 document as they are implemented. 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
   - Heap to Stack+Argument Lowering
   - Interprocedural Alloca Sinking
   - Stack/Heap Access Grouping
   - 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

   ⇒ call *Vectorized Load/Store* pass

6. Optimized Assembler
   - Direct Access to Parent Local Registers(Parameter Reduction)
   - Register Packing
   - 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. 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. If the size of the heap allocation is fairly small and the number of any function arguments does not exceed the maximum(16) by this transformation, this pass will bring the heap variable to the stack and send this variable via arguments. Loss caused by increased function arguments will soon be reoptimized by the Interprocedural alloca Sinking and Direct Access to Parent Local Registers passes. Mem2Reg pass should be followed to clean up the *alloca* operations.
c.f. This is the general version of the Local malloc() to alloca pass.

4. Interprocedural Alloca Sinking

This pass gets the result of the Heap to Stack&Argument pass, and sinks the *alloca* instruction to the head of the dominating function of all uses, referring to the call graph. This combined with the Dead Argument Elimination reduces the number of function arguments of ancestor functions which is increased by the Heap to Stack&Argument pass.

5. Stack/Heap Access grouping

This pass will reorder the memory operations to group stack & heap access as much as it can while not damaging the semantics. In result, we can reduce the number of *reset* operation, and also can save cost for head movement.

6. Combine Instructions

This pass will basically be a derivation from *instcombine* pass in LLVM standard. However, project-specific features will be introduced for higher performance. Some potential examples would be:
- add %x, %x ⇒ mul %x, 2
- and i1 %x, %y ⇒ mul i1 %x, %y

- add %x, 0 ⇒ mul %x, 1 (reg-to-reg 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 because to perform operation using arguments we should obviously bring them to the normal registers(*r1~r16*).

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. However, loading arguments by this method may cause register shortage (registers are not enough to compute in a reasonable speed), so 'rarely accessed read only variables' can be passed with the *argX* register.

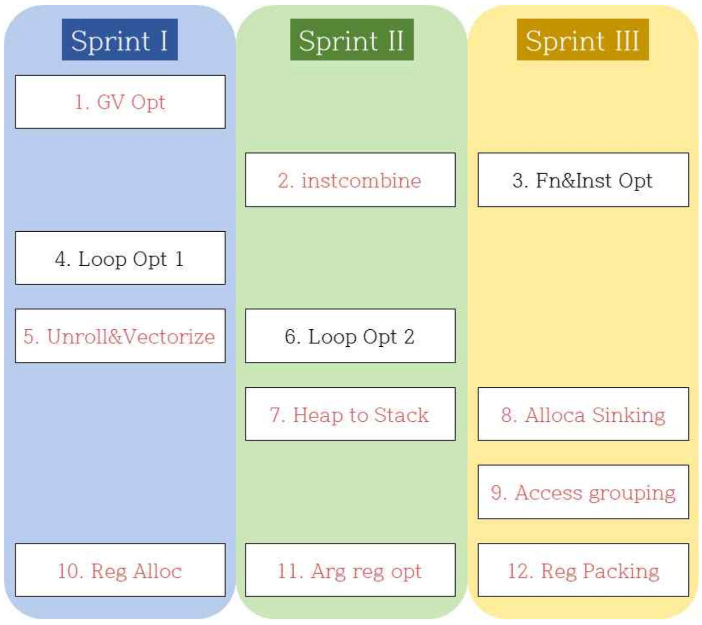
8. Register Packing Analysis + Register Allocation

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 algorithmic approaches could be used to find the (almost) best allocation in reasonable time.

III. Things to do: 12 Chunks of Tasks and Project Timeline

We divided the whole project into 12 chunks so that workloads are distributed flatly. Task chunks include subtasks as following. Red fonts indicate our custom pass. The dependency between these tasks is described in the following figure. Rows in the diagram indicate the procedure at the left should be completed before the right task for compatibility.

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



Our team members will each implement and achieve a single task per sprint, total three during the whole project. We distributed these 12 chunks to all members as fairly as possible. Two criteria was applied to this decision.

- When two tasks are dependent to each other, one person should both implement those when possible.
- Estimated workloads should be distributed fairly enough.

Name	Sprint I	Sprint II	Sprint III
임지규	1	2	12
김도현	4	7	8
정덕인	5	6	9
이진우	10	11	3