<Project Documentation A>

SWPP 2020 Spring LLVM Project
Version 4 (2020.06.13)

Team 4
Hyoung Uk Sul
Ahyoung Oh
Jaewook Lee
Jaeeun Lee

A. Requirement and Specification

Sprint 1

1. SimpleBackend Register Allocation Optimization (Part 1)

i. Description

This optimization unit aims to optimize the allocation of IR registers to the assembly registers. There are an unlimited number of LLVM IR registers, but the given backend assembly only supports 16 registers and one stack register(sp). Therefore, we need an algorithm to allocate the IR registers to assembly registers in the most efficient way possible. Currently, the given backend converts all IR registers to alloca which means that all definition and usage of LLVM IR instructions involve load and store instructions. This causes a huge increase in running cost and does not utilize 16 registers that are available in the backend.

After this optimization is implemented, the backend will no longer use store-load sequence for every instruction, rather, it will only use store and load instruction only when it runs out of registers to store different values, evicting existing values by LRU policy. Furthermore, it will not allocate stack for every single instruction, and rather only use backend registers for frequently used values. Overall, this optimization will significantly reduce the number of store and load instructions introduced in the output assembly code, and hence reduce the running cost to a good extent.

ii. Algorithm in Pseudo-code

```
for every instruction in each function F:
    count the total number of usages for each instruction
    sort the instructions by their usages
    decide the total number of permanent register users by looking at instructions that require
temporary registers, such as call instruction
    assign permanent right to use registers for certain frequently used instructions

for every instruction in each function F:
    if it is a permanent register user:
      give the register it is assigned to and continue
    else if a temporary register is available:
      give the temporary register and continue
    else
      evict the value that was stationed on temporary register for the longest (LRU)
      insert store instruction for evicted value (it will be loaded when used again later)
      give the new available register to this instruction and continue
```

iii. Three Assembly Programs (IR programs are not feasible here since this is backend optimization)

Before Optimization	**After Optimization**
1. Consecutives Add Instructions	

r1 = call read
store 8 r1 sp 0
r1 = load 8 sp 0
r1 = add r1 0 64
store 8 r1 sp 8
r1 = load 8 sp 0
r1 = add r1 1 64
	r1 = call read
	r2 = add r1 0 64
	r4 = add r1 1 64

2. Conditional Instructions	

r1 = load 8 sp 0
r1 = icmp sge r1 0 64
store 8 r1 sp 16
r1 = load 8 sp 16
br r1 bb1 bb2
	r1 = load 8 sp 0
	r2 = icmp sge r1 0 64
	br r2 bb1 bb2

3. Write Instructions with Permanent Registers	

r1 = load 8 sp 8
call write r1
r1 = load 8 sp 24
call write r1
r1 = load 8 sp 40
call write r1
	call write r10
	call write r11
	call write r12
	call write r13
	call write r14

r1 = load 8 sp 56
call write r1
r1 = load 8 sp 72
call write r1
r1 = load 8 sp 88
call write r1	

2. Function Outlining (Part 1)

i. Description

This optimization pass will put a certain group of basic blocks within an IR function into a wholy new function. The purpose of such optimization is to take advantage of the fact that memory access is more costly operation compared with function call in given backend assembly. This optimization will result in much more function calls compared with pre-optimized code, but will have less stack or heap memory access counts. Hence, the total cost of optimized code in the backend machine will be reduced.

In order to implement this functionality, a notion of **post-domination** must be denoted. A block B1 is said to be post-dominating another block B2 if and only if control flow graph from B2 has to reach B1 eventually. Hence, this becomes the case where function-outlining is possible.

ii. Algorithm in Pseudo-code

```
define funcOutline(IRProgram P):
  for every function F in given program P:
    if total number of registers used exceed the maximum physical registers:
      search for block B where breakeven point occurs
      from block B and to its successors, search for block B'  where its successors
                  post-dominate every predecessors AND register usage do not overlap
      if such block B' exists, make a new function F' from block B
        call F' in F where it originally calls B'
```

iii. Three IR Programs

Before Optimization	**After Optimization**
4. Simple case of optimization	
NormalBlock:	NormalBlock:

%0 = … %1 = … … %10 = … br label Breakeven Breakeven: %11 = …. … %20 = … ret … %20	%0 = … %1 = … … %10 = … %r = call @newFunc(dependent regs) ret … %r define … newFunc(...): %11 = … … %20 = … ret … %20
5. Recursive application of optimization	
Block1: (a lot of register usage 1) br label %Block2 Block2: (a lot of register usage 2) br label %Block3 Block3: (a lot of register usage 3) br label %Block4 Block4: (a lot of register usage 4) ret %reg	Block1: (a lot of register usage 1) %r = call @newFunc1(dependent regs) ret %r Define … newFunc1(...): (a lot of register usage 2) %r = call @newFunc2(dependent regs) ret %r Define … newFun2(...): (a lot of register usage 3) %r = call @newFunc3(dependent regs) ret %r Define … newFunc4(...): (a lot of register usage 4) ret %reg
6. Branched optimization	
Block1: (a lot of register usage 1) %c = icmp eq ... br i1 %c, label %Block2, label %Block2 Block2: (a lot of register usage 2) br label %Block4 Block3: (a lot of register usage 3)	Block1: (a lot of register usage 1) %c = icmp eq ... br i1 %c, label %Block2, label %Block2 Block2: (a lot of register usage 2) %r = call @newFunc(dependent regs) ret %r Block3:

```
    br label %Block4

Block4:
    (a lot of register usage 4)
    ret %reg
```
	```
	    (a lot of register usage 3)
	    %r = call @newFunc(dependent regs)
	    ret %r

	Define … newFunc(...):
	    (a lot of register usage 4)
	    ret %reg
	```

3. Malloc to Alloca Conversion (Part 1)

i. Description

This pass will convert allocation in heap to allocation in the stack. In our spec, storing at or loading from the heap costs 4 and storing at or loading from the stack costs 2. This means that there is a chance of optimization by using stack instead of heap. We are going to convert the 'local malloc' to the allocation in the stack and by that way we will be able to reduce cost of 2 whenever the load/store occurs. Also this will be able to reduce the maximum usage of the heap since memory allocated in the heap will move to the stack.

For example,

Allocation in Heap	Allocation in Stack

```
void foo()
{
        int *a = malloc(10*sizeof(int));
        for(int i = 0; i<10; i++) {
                a[i] = i;
        }
        free(a);
}
```
	```
	void foo()
	{
	        int a[10];
	        for(int i = 0; i<10; i++) {
	                a[i] = i;
	        }
	}
	```

The code on the left is assigning 40 bytes in the heap. However, after the optimization it will be allocated in the stack as you can see in the code on the right side.

Condition for discovering replaceable malloc instruction :

1. Malloc-ed memory should be freed in the same function.
    - The malloc-ed memory should not be captured and exist after the function is finished. If not, there is a possibility that other functions will need access to that heap memory and we should not move it to the stack.
2. Malloc should not be dynamic.

- If malloc is dynamic, size of the allocation will be determined during runtime. This is not possible for 2 reasons. First, all alloca instructions should be located in EntryBlock of the function, but we cannot decide the size of the allocation in the EntryBlock. Second, we have to know how big the allocated size is in order to prevent stack overflow.

3. The size of the allocated memory in the heap should be less than 2048 bytes.
    - This is to prevent unlimited usage of stack, causing stack overflow.

Expected decrease in the cost :

1. Store / Load instructions
    - ❏ store/load to heap : 4 → store/load to stack : 2
    - ❏ additional cost for moving head between heap & stack
2. heap allocation / deallocation
    - ❏ malloc : 1, free : 1 → removed
3. max heap usage
    - ❏ memory allocated in heap will move to the stack

ii. Simple algorithm in pseudo code

```
for every function F in given program:
    for every instruction I in F:
        If ( I is call to malloc && I is not dynamic && allocation size is lower than 2048 bytes )
            If ( malloc'd pointer is freed in F){
                Replace I with alloca instruction;
                Remove all corresponding call to free;
                Replace all uses of malloc'd pointer in F to alloc'd pointer;
            }
```

iii. Three IR Programs

Before Optimization	**After Optimization**
1. Function that allocates data in heap	

%p = alloca i32*, align 8
// p = malloc(8);
%call = **call i8* @malloc(i64 8)**
%0 = bitcast i8* %call to i32*
store i32* %0, i32** %p, align 8
%1 = **load i32*, i32** %p, align 8**
%arrayidx = getelementptr inbounds i32, i32* %1, i64 0
	// int p[2];
	%p = alloca [2 x i32], align 4
	// p[0] = 1;
	%arrayidx = getelementptr inbounds [2 x i32], [2 x i32]* %p, i64 0, i64 0
	store i32 1, i32* %arrayidx, align 4
	…

store i32 1, i32* %arrayidx, align 4 ... %3 = load i32*, i32** %p, align 8 %4 = bitcast i32* %3 to i8* // free(p) **call void @free(i8* %4)** ...	
2. Function which store its argument in heap, load the value from heap, and return it.	

define i32 @F(i32 %a) {
entry:
  %a.addr = alloca i32, align 4
  %ret = alloca i32, align 4
  %p = alloca i32*, align 8
  store i32 %a, i32* %a.addr, align 4
  %call = call i8* @malloc(i64 4)
  %0 = bitcast i8* %call to i32*
  store i32* %0, i32** %p, align 8
  %1 = load i32, i32* %a.addr, align 4
  %2 = load i32*, i32** %p, align 8
  store i32 %1, i32* %2, align 4
  %3 = load i32*, i32** %p, align 8
  %4 = load i32, i32* %3, align 4
  store i32 %4, i32* %ret, align 4
  %5 = load i32*, i32** %p, align 8
  %6 = bitcast i32* %5 to i8*
  call void @free(i8* %6)
  %7 = load i32, i32* %ret, align 4
  ret i32 %7
}
	define i32 @G(i32 %a) {
	entry:
	  %a.addr = alloca i32, align 4
	  %ret = alloca i32, align 4
	  %p = alloca i32, align 4
	  store i32 %a, i32* %a.addr, align 4
	  %0 = load i32, i32* %a.addr, align 4
	  store i32 %0, i32* %p, align 4
	  %1 = load i32, i32* %p, align 4
	  store i32 %1, i32* %ret, align 4
	  %2 = load i32, i32* %ret, align 4
	  ret i32 %2
	}

3. Function with multiple BasicBlocks	
define void F()	define void G()

entry:
  %p = alloca i32*, align 8
  store i32 %a, i32* %a.addr, align 4
  %call = call i8* @malloc(i64 4)
   …
   br …
 	
...
 	
exit:
  call void @free(i8* %6)
  …
  ret void
	entry:
	  %p = alloca [4 x i32], align 4
	  …
	  br …
	
	…
	
	exit:
	  …
	  ret void

4. Arithmetic Optimizations

i. Description

This optimization pass first does integer equality propagation, and then optimizes arithmetic operations by pattern-matching instructions that we are interested in. According to our spec, different from normal processors, Integer multiplication/division costs 0.6, integer shift/logical costs 0.8 and integer add/sub costs 1.2. As Mul/Div is cheaper than Add/Sub, shift operations and logical operations, I transformed other operations to Mul/Div if possible.

ii. Simple algorithm in pseudo code

```
using PatternMatch;

for every function F in given program:
    for every basicblock BB in F:
        for every instruction in BB:
                do integer equality propagation
                If(I matches  icm eq (Constant,Instruction or Argument))
                        Propagate Constant

                If(I matches  icm eq (Argument,Instruction))
                        Propagate Argument
                If(rest of the cases)
                    Propagate the former


for every function F in given program:
    for every basicblock BB in F:
        for every instruction in BB:
                If (I matches m_Shl(x,i))
                    Replace I with Mul(x, 2^i)

                If(I matches m_Add(x,x))
                    Replace with Mul(x,2)

                If(I matches m_Shr(x,i))
                    Replace with uDiv(x, 2^i)

                If(I matches m_Sub(x,x))
                    Replace with 0

                If(I matches m_Sub(x,0))
                    Replace with x

                If(I matches m_Add(x,0))
                    Replace with x

                If(I matches m_Mul(x,0))
                    Replace with 0
```

iii. Three IR Programs

Before Optimization	**After Optimization**
1. Replace addition with multiplication	
… %a = add i32 %x, %x …	… %a = mul i32 %x, 2 …
2. Replace shift-left with multiplication	
…	…

%a = shl i32 %x, 2 …	%a = mul i32 %x, 4 …
3. Replace shift-right with unsigned division	
… %a = ashr i32 %x, 2 …	… %a = urem i32 %x, 4 …
4. Eliminate meaningless operations	
… %a = add i32 %x, 0 %b = sub i32 %y, 0 …	… Replace %a with %x Replace %b with %y …
5. Replace subtraction with zero	
… %a = sub i32 %x, %x …	… Replace %a with 0 …
6. Replace multiplication with zero	
… %a = mul i32 %x, 0 …	… Replace %a with 0 …

Sprint 2

5. SimpleBackend Register Allocation Optimization (Part 2)

i. Description

The part 1 of this optimization greatly improved the backend's register utilization, leading to an almost twofold decrease in running cost for most test cases. However, there are still more optimizations to be done in order to ensure that the given backend fully utilizes register allocation and there is no overhead from overusing stack area. Specifically, the following are implemented to further up optimization from part 1.

1. After optimization from part 1, the depromotion algorithm inserts alloca instructions for temporary register users at the entry block regardless of whether the stack area allocated by this instruction is actually used. If a temporary value is never evicted back to stack, alloca instruction for this value should be deleted to decrease stack usage and overhead of moving the head.
2. After optimization from part 1, the depromotion algorithm inserts store instruction whenever a temporary register user is evicted from the list of register users. However, if such value is never used again, the register does not need to be stored. Such store instructions should be removed to decrease the overall cost and overhead of the eviction algorithm.
3. Phi nodes and values in loops are more likely to be used much frequently than those values that are not. Hence, they must prioritized on deciding permanent register users.
4. The bitcast/truncate/extension instructions do not need to take up registers as they appear to introduce unnecessary instruction in the resulting assembly code: simply multiplying the value by one. Hence, they can be removed if their removal does not introduce any dependency issue.
5. If an instruction uses an operand that is already saved on a register, and this operand is never used again after this instruction, then this operand no longer needs to take up its register space. Therefore, an instruction will take its operand's register ID if the operand is used only once in this instruction.

ii. Algorithm in Pseudo-code

```
for every alloca instruction in each function F from Module M, after initial optimization:
    if there does not exist any access to stack addressed by this alloca:
        remove the alloca instruction
for every store instruction in each function F from Module M, after initial optimization:
    if there does not exist any loads to the same location after the store:
        remove the store instruction
Upon deciding permanent register users, from part 1:
    if a value is a phi node, or is in a loop:
        give more priority towards becoming a permanent register user
for every bitcast/truncate/extension instruction:
    if its removal does not affect the functionality of a module:
        remove the cast instruction (by equalizing the value name with operand name)
for every instruction that uses another instruction as its operand:
    if the operand is not used elsewhere:
        take the operand's register ID
```

iii. Three Assembly Programs (IR programs are not feasible here since this is backend optimization)

Before Optimization	**After Optimization**
1. Removing unnecessary allocas	
sp = sub sp 800 64 store 8 0 sp 0 store 8 0 sp 8 store 8 0 sp 16 store 8 0 sp 24 store 8 0 sp 32 store 8 0 sp 40 store 8 0 sp 56 store 8 0 sp 64 … store 8 0 sp 792 (sp+48 is never used)	sp = sub sp 792 64 store 8 0 sp 0 store 8 0 sp 8 store 8 0 sp 16 store 8 0 sp 24 store 8 0 sp 32 store 8 0 sp 40 store 8 0 sp 48 store 8 0 sp 56 store 8 0 sp 64 … store 8 0 sp 784
2. Removing unnecessary stores	

sp = sub sp 344 64
r9 = call read
r2 = add r9 0 64
r4 = add r9 1 64
r6 = add r9 2 64
r8 = add r9 3 64
r3 = add r9 4 64
...
r7 = add r9 5 64
r5 = add r9 6 64
store 8 r2 sp 0
r2 = add r9 7 64
(sp+0 is never loaded again)
	sp = sub sp 344 64
	r9 = call read
	r2 = add r9 0 64
	r4 = add r9 1 64
	r6 = add r9 2 64
	r8 = add r9 3 64
	r3 = add r9 4 64
	...
	r7 = add r9 5 64
	r5 = add r9 6 64
	r2 = add r9 7 64

3. Using operand's register ID	

r1 = add r2 r3
r4 = add r1 r3
r5 = add r4 r3
...
	r1 = add r2 r3
	r1 = add r1 r3
	r1 = add r1 r3
	...

6. Reordering memory accesses (Part 1)

i. Description

The main idea of this pass is to minimize the head's traveling cost. According to our spec, the cost for moving the head to the desired address is 0.0004 * |desired address - previous address|. On the other hand, the cost for 'reset' (i.e. moving the head to the beginning of stack or heap) is fixed to 2. This implies that when the cost for moving the head is bigger than 2 + 0.0004 * |desired address - beginning of stack/heap|, it is more efficient to move the head to the beginning of stack or heap.

For example, let's suppose we've accessed address 10232 (stack area) and then accessed address 20488 (heap area). If we just move head, it would cost 0.0004 * |20488 - 10232| = 4.1024, while it would cost 2 + 0.0004 * |20488 - 20480| = 2.0032 if we reset the head.

Since the cost gets larger proportional to the distance between the addresses, it would be very important to well use the 'reset' instruction. I am going to use 'reset' when the head needs to travel from stack to heap or from heap to stack.

ii. Algorithm in Pseudo-code

```
in AssemblyEmitter.cpp,

    Make emitReset function

    Initialize accessHeap var as unknown when visited basicblock

    If visited CallInst and that instruction calls malloc or
    If visited AllocaInst
        Put it in the memAllocation vector

    If visited LoadInst or StoreInst, if it's the user of AllocaInst or MallocCallInst
        Set accessHeap var as 0(stack), 1(heap) respectively
```

iii. Three IR / Assembly Programs

Before Optimization	**After Optimization**
1. Global variables -> stack access	

@var = **global i32** 21
 	
define i32 @main() {
  %1 = **load i32**, **i32*** @var
  ; load the global variable
  %2 = **mul i32** %1, 2
  ...
  %ptr = alloca i32
  store i32 %2, i32* %ptr
   ...
  %val = load i32, i32* %ptr
  %sum = add i32 %val, 10
   ...
   store i32 %sum, **i32*** @var
   ; store instruction to write to global variable
   ret i32 %sum
}
	@var = **global i32** 21
	
	define i32 @main() {
	  %1 = **load i32**, **i32*** @var
	  ; load the global variable
	  %2 = **mul i32** %1, 2
	  ...
	  reset [stack] (added in assembly)
	  %ptr = alloca i32
	  store i32 %2, i32* %ptr
	   ...
	  %val = load i32, i32* %ptr
	  %sum = add i32 %val, 10
	   ...
	   reset [heap] (added in assembly)
	   store i32 %sum, **i32*** @var
	   ; store instruction to write to global variable
	   ret i32 %sum
	}

2. Travel between stack and heap	

define void foo() {
    BB1 :
     …
     %tmp = alloca i32
     %call = call i8* @malloc(i64 32)
     %0 = bitcast i8* %call to i32*
     store i32 1 i32* %0
     %elem.1 = load i32* %0
     store i32 %elem.1 i32* %tmp
     ...
     ret i64 0
    }
	define void foo() {
	    BB1 :
	     …
	     %tmp = alloca i32
	     %call = call i8* @malloc(i64 32)
	     %0 = bitcast i8* %call to i32*
	     store i32 1 i32* %0
	     %elem.1 = load i32* %0
	     reset [stack] (added in assembly)
	     store i32 %elem.1 i32* %tmp
	     …
	     ret i64 0
	    }

3. Travel between stack and heap (Assembly -> Assembly)	
… r1 = mul arg1 8 64	… r1 = mul arg1 8 64

store 8 r1 sp 8 r1 = load 8 sp 8 r1 = malloc r1 store 8 0 r1 0 store 8 r1 sp 16 r2 = load 8 r1 0 sub sp 800 64 store 8 0 sp 0 …	store 8 r1 sp 8 r1 = load 8 sp 8 r1 = malloc r1 store 8 0 r1 0 reset [stack] (added in assembly) store 8 r1 sp 16 reset [heap] (added in assembly) r2 = load 8 r1 0 sub sp 800 64 store 8 0 sp 0 …

7. Loop Optimization

i. Description

This optimization is for the general optimization for loop-operations. There are many existing LLVM passes that handle the loop operation, and I am going to analyze each loop-related pass and see which ones can be added to our compiler without any problems (such as increasing the cost or causing conflicts with added passes.) Then, I will carefully control the order of the adding the passes so that the cost can be reduced by the biggest rate.

In this optimization, I added 5 additional loops, but there are 3 main passes that will reduce our cost: LICMPass, LoopDeletionPass, LoopUnrollPass.

LICMPass : Loop Invariant Code Motion

for(i=0; i<100; i++) {
  a[i] = i - n*n;
}
	temp = n*n
	for(i=0; i<100; i++) {
	   a[i] = i - temp;
	 }

This pass will get the loop-invariant instructions out of the loop body so that it does not have to be repeated many times. This will especially show big cost reduction when moving loop-invariant store/load instructions because these instructions costs a lot.

LoopDeletionPass

```
entry:
  br label %for.cond

for.cond:
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i64 %i.0, 100
  br i1 %cmp, label %for.body, label %for.end

for.body
  br label %for.inc

for.inc:
  %inc = add nsw i64 %i.0, 1
  br label %for.cond

for.end:
  ret void
```

```
entry:
  ret void
```

The code to the left is actually the empty loop that does not do anything, and does not affect the code outside. This pass will delete such dead loop so that its cost can be ignored.

LoopUnrollPass

```
for(i=0; i<10; i++) {
  write(n);
}
```

```
write(n);
write(n);
write(n);
....
write(n);
(10 times)
```

It is literally unrolling the simple instruction in the loop if it can be unrolled.

Expected decrease in the cost :

1. Instruction repetition cost within the loop (especially store/load)
   - ❑ If the loop invariant code is not repeated in every loop-cycle, the cost will be reduced.
2. Induction variable calculating cost
   - ❑ cost of calculating the induction variable will be reduced if the loop is unrolled or deleted.

ii. Algorithm in Pseudo-code

```
For all instructions I in this loop L:
        Sink I to the exit blocks if all users of the I is outside the loop
        Hoist I to the preheader block if all operands of I are loop-invariant

If Loop L has preheader && is in simplified form
        If L does not have its subloops
          If L is never reached from EntryBlock
```

```
            delete L.
        If L has only one exit block and all the instructions inside is invariant
            delete L

For all loops in this function:
    Convert the loop into the simplified form
    By the inner loop -> outer loop order:
        If the given conditions(PreserveLCSSA, Threshold,etc.) are satisfied:
            Unroll the loop
```

iii. Three IR Programs

Before Optimization	**After Optimization**
1. LICM	
entry: %a = alloca [100 x i64], align 16 br label %for.cond for.cond: %i = phi i64 [0, %entry], [%inc, %for.inc] %cmp = icmp slt i64 %i, 100 br i1 %cmp, label %for.body, label %for.end for.body: ... %mul = mul nsw i64 %n, %n ... br label %for.inc for.inc: %inc = add nsw i64 %i.0, 1 br label %for.cond for.end: ret void	entry: %a = alloca [100 x i64], align 16 %mul = mul nsw i64 %n, %n br label %for.cond for.cond: %i = phi i64 [0, %entry], [%inc, %for.inc] %cmp = icmp slt i64 %i, 100 br i1 %cmp, label %for.body, label %for.end for.body: ... (use %mul) ... br label %for.inc for.inc: %inc = add nsw i64 %i.0, 1 br label %for.cond for.end: ret void
2. Loop Deletion	
entry: br label %for.cond for.cond:	entry: ret void

```
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i64 %i.0, 100
  br i1 %cmp, label %for.body, label %for.end

for.body
  br label %for.inc

for.inc:
  %inc = add nsw i64 %i.0, 1
  br label %for.cond

for.end:
  ret void
```

3. Loop Unroll

```
entry:
  br label %for.cond

for.cond:
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i64 %i.0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:
  call void @write(i64 %n)
  br label %for.inc

for.inc:
  %inc = add nsw i64 %i.0, 1
  br label %for.cond

for.end:
  ret void
```

```
entry:
  call void @write(i64 %n)
  call void @write(i64 %n)
  call void @write(i64 %n)
   .....
  call void @write(i64 %n)
  ret void
```

8. Malloc to Alloca Conversion (Part 2) : Add GVN Pass

i. Description

This optimization is in the extension of the previous pass I added : Malloc2AllocPass. As I mentioned in sprint 1, this pass is only tracking the freed pointer through bitcast operations, so if the malloc-ed pointer is stored and then loaded again, it will not be removed. Thus to convert more malloc into alloca, unnecessary store/load instructions should be removed. That is why I am adding GVN Pass as a extension of the Malloc2AllocPass.

GVN Passes does two things :

1. tracks the load/store instructions or getelementptr instructions(which calculates the pointer value to store or load), and if the same instruction is used again, it stores the value of the first instruction and uses it for the second instruction.

2. if the program is loading the value that is previously stored in memory, and there is guarantee that the stored value is not changed, it just uses the value used in the store instruction instead of loading it again.

After implementing GVN Pass, Malloc2Alloc Pass will show better performance because it can be applied to more cases.

ii. Algorithm in Pseudo-code

```
Merge unconditional branches
For basic blocks BB in Function F
    For instruction I in BB:
        Check if I is redundant
        If I is redundant
            Delete I from BB
```

After applying the GVN, Malloc2Alloc will be performed like following code

```
Remove overlapping/unnecessary store/load/getelementptr instructions (for better performance)
If (freed pointer = malloc-ed pointer)
    Remove free inst & convert malloc to alloca
```

iii. Three IR Programs

Before Optimization	**After Optimization**
1. Same load-store to the stack	

BB:
  %a = alloca i64*
  %call = call noalias i8* @malloc(i64 80) #3
  %0 = bitcast i8* %call to i64*
  store i64* %0, i64** %a
  %1 = load i64*, i64** %a
  %2 = bitcast i64* %1 to i8*
  call void @free(i8* %2) #3
  ret void
	BB:
	  %a = alloca i64*
	  %call = call noalias i8* @malloc(i64 80) #3
	  %0 = bitcast i8* %call to i64*
	  store i64* %0, i64** %a
	   ...
	  %1 = bitcast i64* %0 to i8*
	  call void @free(i8* %1) #3
	  ret void

2. Same load-store to the heap	
BB:	BB:
%0 = load i64, i64* @num, align 8	%0 = load i64, i64* @num, align 8
%inc = add nsw i64 %0, 1	%inc = add nsw i64 %0, 1
store i64 %inc, i64* @num, align 8	store i64 %inc, i64* @num, align 8
%1 = load i64, i64* @num, align 8	...
%call = call i32 @foo(i64 %1)	%call = call i32 @foo(i64 %inc)
...	...
3. Same getelementptr calculation	
BB1:	BB1:
...	...
%idxprom = sext i32 %i.0 to i64	%idxprom = sext i32 %i.0 to i64
%arrayidx = getelementptr inbounds i64,	%arrayidx = getelementptr inbounds i64,
i64* %ptr, i64 %idxprom	i64* %ptr, i64 %idxprom
%0 = load i64, i64* %arrayidx, align 8	%0 = load i64, i64* %arrayidx, align 8
...	...
BB2:	BB2:
...	...
%idxprom4 = sext i32 %i.0 to i64	store i64 %1, i64* %arrayidx, align 8
%arrayidx5 = getelementptr inbounds i64,	...
i64* %ptr, i64 %idxprom4	
store i64 %1, i64* %arrayidx5, align 8	
...	

9. Function Outlining (Part 2)

Outlining whole 'blocks' and splitting big blocks into multiple blocks (more than two) , unlike part 1 where we outline big blocks into smaller blocks.

i. Description

This optimization pass will put a certain group of basic blocks within an IR function into a wholy new function. The purpose of such optimization is to take advantage of the fact that memory access is a more costly operation compared with function call in given backend assembly. This optimization will result in much more function calls compared with pre-optimized code, but will have less stack or heap memory access counts. Hence, the total cost of optimized code in the backend machine will be reduced.

In order to implement this functionality, a notion of **post-domination** must be denoted. A block B1 is said to be post-dominating another block B2 if and only if the control flow graph from B2 has to reach B1 eventually. Hence, this becomes the case where function-outlining is possible.

Also, we handle the example where we split a big block into smaller blocks (more than 2 blocks, which we handled in part 1).

ii. Rough Algorithm

```
Def Function Outlinepass(Block level):

    Iterate through all of the functions in the module:
        Iterate through all of the blocks in a function:
            If total number of registers used exceed the maximum physical registers:
                Keep track of the block, and push them into a vector

        After pushing all the blocks, then Iterate through the vector:
            Outline the blocks using CodeExtractor
```

Iii. Three IR examples

1. Case of extracting the next block	
Entry:	Entry:
(a lot of register usage)	(a lot of register usage)
%br next_block	%r = call @newFunc(dependent regs)
Next_block:	ret %r
(a lot of register usage)	Define … newFunc(...):
	(a lot of register usage 4)
	ret %reg
2. Case of extracting two blocks (br branches)	

```
Block1:
        (a lot of register usage 1)
        %c = icmp eq ...
        br i1 %c, label %Block2, label %Block3

Block2:
        (a lot of register usage 2)
        br label %Block4

Block3:
        (a lot of register usage 3)


        ....
```

```
Block1:
        (a lot of register usage 1)
        %c = icmp eq ...
        br i1 %c, label %Block2, label %Block3
Block2:
        (a lot of register usage 2)
        %r = call @newFunc(dependent regs)
        ret %r
Block3:
        (a lot of register usage 3)
        %r = call @newFunc(dependent regs)
        ret %r

Define … newFunc(...):
        (a lot of register usage 4)
        ret %reg
```

3. Extracting block into multiple blocks

```
Big block:
            (a lot of register usage)
            (a lot of register usage)
            (a lot of register usage)
            (a lot of register usage)
            Br next block
```

```
Big block:
      (a lot of register usage)
      %r = call @splitblock1(dependent regs)
      Ret &r

Define … splitblock1(...):
        (a lot of register usage)
        %r = call @splitblock2(dependent regs)
Define splitblock2 () :
      (a lot of register usage)
      %r = call @splitblock3(dependent regs)
Define splitblock3 () :
      (a lot of register usage)
      ret %reg
```

Sprint 3

10. SimpleBackend Register Allocation Optimization (Part 3)

i. Description

The part 1 and 2 of register allocation optimization greatly reduced the overall cost of compiled programs by efficiently managing assignment of register IDs, preventing unnecessary stack access and memory management. Although many possible optimizations were successfully accomplished, the depromotion algorithm still has some bits to improve on. Furthermore, because optimization from part 1 and 2 mostly rely on heuristics in the intention of speeding up compiled programs, they are prone to errors from corner cases. Even though the implementation successfully passes benchmarks and test cases, there might still exist some corner cases that this fails to catch.

The main goal of this part is to reduce costs by preventing depromotion algorithm from creating unnecessary loads and stores, and also increase the safety of algorithm in general. Specifically, unlike implementation from the previous sprint, this implementation will now attempt to track every possible flow of control in CFG of a given program to ensure that every load and store is, in fact, necessary. For instance, if the previous implementation put a load in front of a value's every use because the value is evicted somewhere previous to its location, this implementation will not put loads everywhere; rather, it will put load instruction at a location where it can dominate the maximum number of its use. Something similar will go for store instructions as well; if there is a store, and every possible route after this store does not load from the location that this store instruction stores to, then the store instruction will be removed.

ii. Algorithm in Pseudo-code

```
for a load instruction created during depromotion:
    Track every possible route on CFG that can reach to this point:
        if every possible route already contains load/definition of this value:
            this load is unnecessary; remove it
for a store instruction created during depromotion:
    Track every possible route on CFG that can be reached after this point:
        if every possible route does not contain load of store location:
            this store is unnecessary; remove it
```

iii. Three Assembly Programs (IR programs are not feasible here since this is backend optimization)

Before Optimization	After Optimization
1. Removing unnecessary loads	

```
blockEviction:
   ...
   store reg reg_slot
   reg = some_other_value
   ...
   br blockA

blockA:
   ...
   reg = load reg_slot
   ... = add reg, ....
   ...
   br cond if.true if.false

If.true:
   ...
   reg = load reg_slot
   ... = add reg, ....
   ...
   br blockB

if.false:
   ...
   reg = load reg_slot
   ... = add reg, ....
   ...
   br blockB

blockB:
   ...
   reg = load reg_slot
   ... = add reg, ....
   ...
   br blockC
....
```

(because reg is evicted in blockEviction, algorithm will insert load for every use after eviction)

```
blockEviction:
   ...
   store reg reg_slot
   reg = some_other_value
   ...
   br blockA

blockA:
   ...
   reg = load reg_slot
   ... = add reg, ....
   ...
   br cond if.true if.false
If.true:
   ...
   ... = add reg, ....
   ...
   br blockB
if.false:
   ...
   ... = add reg, ....
   ...
   br blockB
blockB:
   ...
   ... = add reg, ....
   ...
   br blockC
....
```

2. Removing unnecessary loads

```
blockEviction:
   ...
   store reg reg_slot
   reg = some_other_value
   ...
```

```
blockEviction:
   ...
   store reg reg_slot
   reg = some_other_value
   ...
```

```
    br blockCond

  blockCond:
    ...
    reg = load reg_slot
    ... = add reg, ....
    ...
    br cond blockBody blockEnd

  blockBody:
    ...
    reg = load reg_slot
    ... = add reg, ....
    ...
    br blockCond

  blockEnd:
    ...
    reg = load reg_slot
    ... = add reg, ....
    ...
    br blockA
  ...
```

```
    reg = load reg_slot
    br blockCond

  blockCond:
    ...
    ... = add reg, ....
    ...
    br cond blockBody blockEnd

  blockBody:
    ...
    ... = add reg, ....
    ...
    br blockCond

  blockEnd:
    ...
    ... = add reg, ....
    ...
    br blockA
  ...
```

(choose load location that incurs minimum cost)

3. Removing unnecessary store

```
  blockA:
    ...
    reg = some_value
    ...
    br blockCond

  blockCond:
    ...
    ... = add reg, ....
    ...
    br cond blockBody blockEnd

  blockBody:
    ...
    ... = add reg, ....
    ...
    br blockCond

  blockEnd:
    ...
```

```
  blockA:
    ...
    reg = some_value
    ...
    br blockCond

  blockCond:
    ...
    ... = add reg, ....
    ...
    br cond blockBody blockEnd

  blockBody:
    ...
    ... = add reg, ....
    ...
    br blockCond

  blockEnd:
    ...
```

store reg reg_slot (eviction) reg = some_other_value … ret i32 0	reg = some_other_value … ret i32 0 (store is not needed, so remove it)

11. Function Inlining

i. Description

Function inlining is a classic technique in program optimization in which a function call is simply replaced with an actual function body. The purpose of this optimization is to reduce function call overhead and to preserve program state, that is, stack and register status, which naturally lead to further overhead reduction. For instance:

```
int max(int a, int b) {
   return a > b ? a : b;
}

int main() {
   max(a, b);
}
```

Above code does not need to go through additional routine overhead. Hence, it can be optimized to:

```
int main() {
   a > b ? a : b;
}
```

>> What is important in this pass, is to be aware of the fact that in sprint 1 and sprint 2, there is a function outlining pass that outlines blocks if the function uses a lot of registers. To make both function inlining and outlining cost-effective, it is important to run the inlining pass before running the outline pass and also trying to target cases only when inlining is cheaper. Thereby, we decided to not just 'run' the existing function inlining pass but implement one on our own, only targeting cases where the function is 'small enough' to inline.

ii. Algorithm in Pseudo-code

So the key point in this algorithm is to detect which functions to inline and which to not inline. For the actually inlining process, I decided to use the method

```
InlineResult InlineFunction(CallBase *CB, InlineFunctionInfo &IFI,
                            AAResults *CalleeAAR = nullptr,
                            bool InsertLifetime = true);
```

The algorithm for detection of the need-to-be inlined functions is the following.

```
For Functions in Module:
    For Basic Blocks in a Function:
        For Instructions in Basic Blocks:
            Stuff all 'Call Instructions' in a vector

For CallInsts in vector:
    Analyze the called function of the called instruction:
    Analyze: Check if the inlining will save costs and not cause allocas or will be a target of the function outlining pass (Done heuristically by experimenting with costs)
    After analysis -> if worthy of inlining, stuff it into a new vector.

For Callinsts in the final vector:
    Call the InlineFunction method, and inline these functions
```

iii. Three Assembly Programs

1. Inline small functions with single block	
Entry: (...) %r = call @newFunc(dependent regs) ret %r Define … someFunc(...): .entry: (...little usage of regs..) ret %reg	Entry: (...) Delete call inst .inlined-func-block (...does what the function does) ret %reg
2. Inline functions with multiple blocks, but that do not use many regs	

Block1: %c = icmp eq ... br i1 %c, label %Block2, label %Block3 Block2: %r = call @someFunc(dependent regs) ret %r Define … someFunc(...): .entry: (...not many regs…) br label %for entry .for entry: (...not many regs…) …	Block1: %c = icmp eq ... br i1 %c, label %Block2, label %Block3 Block2: Delete call inst .inlined-func-entry: (...not many regs…) br label %inlined- func-for entry .inlined-func-for entry: (...not many regs…)
3. Recursively inlining as much as possible just before it needs outlining	
Block: (some register usage) %r = call @splitblock1(dependent regs) Ret &r Define …func1(...): (some regs) %r = call @func2(dependent regs) Define func2 () : (some regs) %r = call @func3(dependent regs) Define func3 () : (some regs) ret %reg	Block: (some register usage) Delete call inst .inlined-func1-into-blocks: (....insts…..) .inlined-func2-into-blocks: (....insts…..) .inlined-func3-into-blocks: (....insts…..) Ret &r

12. Function Optimization

1) SCCP Pass

i. Description

Sparse Conditional Constant Propagation : SCCP pass does sparse conditional constant propagation and merging, like following:

- Assumes values are constant unless proven otherwise
- Assumes BasicBlocks are dead unless proven otherwise
- Proves values to be constant, and replaces them with constants
- Proves conditional branches to be unconditional
  (from llvm.org)
  This pass works well with DCE pass so I've changed the order of the passes.

2) Unreachable Block Elimination

i. Description

This pass is a simplified version of SimplifyCFG pass. This pass eliminates LLVM basic blocks that are not reachable from the entry node by performing depth first traversal of the CFG and detecting unvisited nodes. This pass may help in case we have inputs that contain unreachable blocks.

3) Tail Call Elimination

i. Description

This optimization is the famous tail call optimization, in which the compiler utilizes tail recursive function call by removing additional stack overhead from recursive calls. Tail recursion occurs in the case where the very last operation a function does is a call to another function. In this case, since the caller function simply returns whatever callee returns, caller function's stack is no longer needed. Hence, tail optimization will simply reuse the environment of the tail recursive caller and jump straight to callee function. Through this method, the overhead cost of function call and additional stack allocation can be saved. For example:

```
void print(int n) {
   printf("%d\n", n);
   print(n - 1);
}
```

Above code will cause a myriad of recursive function stack frames if left as it is. However, using the fact that above function involves tail recursion, it can be optimized to following:

```
void print(int n) {
start:
   printf("%d\n", n);
   goto start;
}
```

Although arguably dreaded, above code can significantly reduce cost from function calls.

=> This optimization will be done by implementing the existing pass.

13. Reordering memory accesses (Part 2)

i. Description

For “Reordering memory accesses optimization Part 2”, the optimization that could be done by modifying the AssemblyEmitter.cpp code will be implemented.

The cases to emit reset will be generalized to the function level. In sprint2, in part 1 of this optimization, a reset instruction was emitted only when it’s 100% sure that the memory access has changed(either from heap to stack or from stack to heap) within the same block. However, it’d be more efficient if the memory access change could be detected within the same function. If implemented, more reset instructions will be emitted as there could be cases like the head remaining in the stack for the BB1 and then the head moving to the heap for the next block.

ii. Algorithm in Pseudo-code (in AssemblyEmitter.cpp)

In AssemblyEmitter.cpp,

```
When visited LoadInst or StoreInst
    See if its pointer operand is the user of instructions in the MemAllocation vector
    See if its pointer operand is BitCast Inst or PtrtoInt Inst, etc.. and track if it’s the user of
    the instructions in the MemAllocation vector

This time, don’t initialize the heapAccess as unknown(2) when we visit the basic block

Following the control flow, detect the memory access change
```

iii. Three Assembly Programs (IR programs are not feasible here since this is backend optimization)

Before Optimization	**After Optimization**
1. Emit reset	

.if.then:
  r6 = load 8 sp 0
  free r6
  br .if.end
 	
.if.end:
  r4 = malloc 64
  r2 = mul r4 1 64
  r6 = add r2 40 64
  store 4 16 r6 0
  r5 = add r2 44 64
  store 4 0 r5 0
  r7 = icmp ne r2 0 64
  br r7 .if.then1 .if.end1
…
	.if.then:
	  r6 = load 8 sp 0
	  free r6
	  br .if.end
	
	.if.end:
	  r4 = malloc 64
	  r2 = mul r4 1 64
	  r6 = add r2 40 64
	  store 4 16 r6 0
	  r5 = add r2 44 64
	  reset [heap] (added in assembly)
	  store 4 0 r5 0
	  r7 = icmp ne r2 0 64
	  br r7 .if.then1 .if.end1…

2. Emit reset	

define void foo() {
    BB1 :
      …
      %tmp = alloca i32
      %call = call i8* @malloc(i64 32)
      %0 = bitcast i8* %call to i32*
      store i32 1 i32* %0
      %elem.1 = load i32* %0
 	
    BB2 :
      store i32 %elem.1 i32* %tmp
      …
      ret i64 0
    }
	define void foo() {
	    BB1 :
	      …
	      %tmp = alloca i32
	      %call = call i8* @malloc(i64 32)
	      %0 = bitcast i8* %call to i32*
	      store i32 1 i32* %0
	      %elem.1 = load i32* %0
	
	    BB2 :
	      reset [stack] (added in assembly)
	      store i32 %elem.1 i32* %tmp
	      …
	      ret i64 0
	    }

3. Emit reset	

…
r1 = mul arg1 8 64
store 8 r1 sp 8
r1 = load 8 sp 8
r1 = malloc r1
	…
	r1 = mul arg1 8 64
	store 8 r1 sp 8
	r1 = load 8 sp 8
	r1 = malloc r1

```
store 8 0 r1 0
store 8 r1 sp 16
r2 = load 8 r1 0
sub sp 800 64
store 8 0 sp 0
...
```

```
store 8 0 r1 0
reset [stack] (added in assembly)
store 8 r1 sp 16
reset [heap] (added in assembly)
r2 = load 8 r1 0
sub sp 800 64
store 8 0 sp 0
...
```

14. Global Variable Load/Store Optimization

i. Description

This pass is the extension of the loop optimization I added in sprint 2. Although I added existing LICM Pass and other loop passes, I could still find some ways to reduce the cost more. In this pass, I will focus on the loop-invariant instruction that is loading the value from the global variable.

```
extern int64_t N;
extern int64_t *array;

void main() {
  N = 100;
  array = malloc(8*1000);
  for (int i = 0; i < 1000; i++) {
     array[i] = N*i
  }
}
```

For example, in this case, if we look at the IR program(as below shows), global variable N and array will be continuously being loaded every time it enters the loop body.

```
entry:
  ...
  br label %for.cond

for.cond:
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, 1000
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %1 = load i64, i64* @N, align 8
  ...
  %2 = load i64*, i64** @array, align 8
   ...(use %1, %2)...
  store i64 %mul, i64* %arrayidx, align 8
```

```
entry:
  ...
  %1 = load i64, i64* @N, align 8
  %2 = load i64*, i64** @array, align 8
  br label %for.cond

for.cond:
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, 1000
  br i1 %cmp, label %for.body, label %for.end

for.body:
  ...
  (use %1, %2)
   ...
```

br label %for.inc for.inc: %inc = add nsw i32 %i.0, 1 br label %for.cond for.end: ret i32 0	store i64 %mul, i64* %arrayidx, align 8 br label %for.inc for.inc: %inc = add nsw i32 %i.0, 1 br label %for.cond for.end: ret i32 0

However, this can be avoided by hoisting the load instruction outside the loop. Actually, total 2000 times of loading can be reduced to only 2 times of loading. Also because global variables will be stored in the heap, store/load cost will be high and thus, this pass might be able to reduce the cost a lot.

In addition, I will further reduce the cost by 1) removing repeated load instructions if they are loading same global variables, and 2) replacing the use of loaded value with stored value if store instruction to same global variable is located in the same preheader block.

ii. Algorithm in Pseudo-code (in AssemblyEmitter.cpp)

```
For Loop L in Function F : (from inside loop(nexted) -> to outside loop)
    For instruction I in the L :
        If (I is load to the globalVar GV && there is no store instruction to GV in L)
            Choose the location to hoist I : preheader block of L + after definition
            Hoist I chosen location
```

iii. Three Assembly Programs (IR programs are not feasible here since this is backend optimization)

Before Optimization	**After Optimization**
1. Hoist the invariant GV load instruction out of the loop	
@N =external dso_local global i64, align 8 @array = external dso_local global i64*, align 8 main() { entry: ... br label %for.cond for.cond: %i.0 = phi i32 [0, %entry], [%inc, %for.inc] %cmp = icmp slt i32 %i.0, 1000 br i1 %cmp, label %for.body, label %for.end for.body: %1 = load i64, i64* @N, align 8 ... %2 = load i64*, i64** @array, align 8 ...(use %1, %2)... store i64 %mul, i64* %arrayidx, align 8	@N =external dso_local global i64, align 8 @array = external dso_local global i64*, align 8 main() { entry: ... %1 = load i64, i64* @N, align 8 %2 = load i64*, i64** @array, align 8 br label %for.cond for.cond: %i.0 = phi i32 [0, %entry], [%inc, %for.inc] %cmp = icmp slt i32 %i.0, 1000 br i1 %cmp, label %for.body, label %for.end for.body: ... (use %1, %2) ...

br label %for.inc
 	
for.inc:
  %inc = add nsw i32 %i.0, 1
  br label %for.cond
 	
for.end:
  ret i32 0
}
	store i64 %mul, i64* %arrayidx, align 8
	  br label %for.inc
	
	for.inc:
	  %inc = add nsw i32 %i.0, 1
	  br label %for.cond
	
	for.end:
	  ret i32 0
	}

2. GV store inst is in the loop -> do not hoist (@N is GV)	

entry:
  ...
  br label %for.cond
 	
for.cond:
  ...
 	
for.body:
  %1 = load i64, i64* @N, align 8
   ...(use %1)...
  store i64 %a, i64* @N
  ...
  store i64 %mul, i64* %arrayidx, align 8
  br label %for.inc
 	
for.inc:
  ...
 	
for.end:
  ret i32 0
	entry:
	  ...
	  br label %for.cond
	
	for.cond:
	  ...
	
	for.body:
	  %1 = load i64, i64* @N, align 8
	   ...(use %1)...
	  store i64 %a, i64* @N
	  ...
	  store i64 %mul, i64* %arrayidx, align 8
	  br label %for.inc
	
	for.inc:
	  ...
	
	for.end:
	  ret i32 0

3. Nested loop -> consider the location of corresponding store inst.	

@N =external dso_local global i64, align 8
@array = external dso_local global i64*, align 8
define dso_local i32 @main() #0 {
entry:
  ...
 	
for.cond:
  ...
  br i1 %cmp, label %for.body, label %for.end9
 	
for.body:
  br label %for.cond1
 	
for.cond1:
  ...
  br i1 %cmp2, label %for.body4, label %for.end
 	
for.body4:
  %1 = load i64, i64* @N, align 8
  ...
  %2 = load i64*, i64** @array, align 8
  ... (use of %1, %2) ...
  br label %for.inc
	@N =external dso_local global i64, align 8
	@array = external dso_local global i64*, align 8
	define dso_local i32 @main() #0 {
	entry:
	  ...
	  %2 = load i64*, i64** @array, align 8 (inst nameshould be changed)
	
	for.cond:
	  ...
	  %1 = load i64, i64* @N, align 8 (inst nameshould be changed)
	  br i1 %cmp, label %for.body, label %for.end9
	
	for.body:
	  br label %for.cond1
	
	for.cond1:
	  ...
	  br i1 %cmp2, label %for.body4, label %for.end
	
	for.body4:
	  ... (use of %1, %2) ...
	  br label %for.inc

```
for.inc:
  ...
  br label %for.cond1

for.end:
  ...
  store i64 %conv6, i64* @N, align 8
  br label %for.inc7

for.inc7:
  ...
  br label %for.cond

for.end9:
  ret i32 0
}
```

```
for.inc:
  ...
  br label %for.cond1

for.end:
  ...
  store i64 %conv6, i64* @N, align 8
  br label %for.inc7

for.inc7:
  ...
  br label %for.cond

for.end9:
  ret i32 0
}
```