<Project Documentation A>

SWPP 2020 Spring LLVM Project
Version 2 (2020.05.16)

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 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. Furthermore, it will not allocate stack for every single instruction, and rather only use backend registers for frequently used values. Overall, it 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)

<table>
<tr><th>Before Optimization</th><th>After Optimization</th></tr>
<tr><td colspan="2">1. Consecutives Add Instructions</td></tr>
<tr><td>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</td><td>r1 = call read
r2 = add r1 0 64
r4 = add r1 1 64</td></tr>
<tr><td colspan="2">2. Conditional Instructions</td></tr>
<tr><td>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</td><td>r1 = load 8 sp 0
r2 = icmp sge r1 0 64
br r2 bb1 bb2</td></tr>
<tr><td colspan="2">3. Write Instructions with Permanent Registers</td></tr>
<tr><td>r1 = load 8 sp 8
call write r1
r1 = load 8 sp 24
call write r1
r1 = load 8 sp 40
call write r1
r1 = load 8 sp 56</td><td>call write r10
call write r11
call write r12
call write r13
call write r14</td></tr>
</table>

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: %0 = …	NormalBlock: %0 = …

<table>
<tr>
<td>%1 = …
…
%10 = …
br label Breakeven
Breakeven:
%11 = ….
…
%20 = …
ret … %20</td>
<td>%1 = …
…
%10 = …
%r = call @newFunc(dependent regs)
ret … %r

define … newFunc(...):
%11 = …
…
%20 = …
ret … %20</td>
</tr>
<tr>
<td colspan="2">5. Recursive application of optimization</td>
</tr>
<tr>
<td>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</td>
<td>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</td>
</tr>
<tr>
<td colspan="2">6. Branched optimization</td>
</tr>
<tr>
<td>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)
br label %Block4</td>
<td>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:
(a lot of register usage 3)</td>
</tr>
</table>

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

3. Malloc to Alloca Conversion

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()

<table>
<tr>
<td>

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

</td>
<td>

entry:
  %p = alloca [4 x i32], align 4
  …
  br …

…

exit:
  …
  ret void

</td>
</tr>
</table>

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 decently improved the backend's register utilization, leading to a good decrease in running cost. 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 can be implemented to further up optimization from part 1.

1. Current backend (after optimization from part 1) inserts alloca instructions for temporary register users at the entry block regardless of whether the stack area allocated by this

instruction is actually used. Hence, such alloca instructions should be deleted to decrease stack usage and overhead of moving the head.

2. Current backend 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 overhead of the eviction algorithm.
3. Current backend does not apply new criterion for a tie on number of usages upon deciding what values to permanently station to registers. However, this should be more carefully decided as this could decrease the running cost from the way memory access is ordered.
4. The bitcast instruction does not need to take up registers as they appear to introduce unnecessary instruction in the resulting assembly code: simply multiplying a value by one.

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 there exists a tie on number of usages:
    give priority to instruction whose usage is closer to definition

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. Giving priority to better the memory access order	
(r1 is temporary, r10 is permanent) sp = sub sp X 64 r10 = call read r1 = call read store 8 r1 sp 0 r1 = (other value) ... r1 = load 8 sp 0 ... = add r1 0 64 store 8 r1 sp 0 r1 = (other value) ... r1 = load 8 sp 0 ... ... = add r10 0 64	(r1 is permanent, r10 is temporary) sp = sub sp X 64 r10 = call read r1 = call read store 8 r10 sp 0 r10 = (other value) ... ... = add r1 0 64 store r10 sp (some other area) ... r10 = load 8 sp 0 ... = add r10 0 64

6. Reordering memory accesses

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.

If possible, in the later sprint, I will try to reorder the allocations of heap accesses and stack accesses in order to make traversal as linear as possible.

ii. Algorithm in Pseudo-code

for every instruction in each function F:
    If I accesses to memory put it in the vector <memAccessInst>

for every instruction in the vector <memAccessInst>
    If I's memory access is different from the previous -> tag
    (**If** I is call to malloc and the next I is allocaInst -> tag
    If I is allocaInst and the next I is call to malloc -> tag)

then in AssemblyEmitter.cpp,
    for all tagged instructions -> reset

iii. Three IR / Assembly Programs

<table>
<tr><th>Before Optimization</th><th>After Optimization</th></tr>
<tr><td colspan="2">1. Global variables -> stack access</td></tr>
<tr><td>@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</td><td>@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</td></tr>
</table>

<table>
<tr><td>...
store i32 %sum, i32* @var
; store instruction to write to global variable
ret i32 %sum
}</td><td>%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
}</td></tr>
<tr><td colspan="2">2. Travel between stack and heap</td></tr>
<tr><td>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
}</td><td>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 [heap] (added in assembly)
store i32 %elem.1 i32* %tmp
…
ret i64 0
}</td></tr>
<tr><td colspan="2">3. Travel between stack and heap (Assembly -> Assembly)</td></tr>
<tr><td>…
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
…</td><td>…
r1 = mul arg1 8 64
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
…</td></tr>
</table>

7. Runtime Garbage Collector Pass

i. Description

This pass is for the efficient usage of the heap. If the memory in the heap is allocated, but no users are using or making reference to this allocated block after the allocation, or after a particular time in the code, then this block can be deallocated. I will call this useless block as a ‘garbage’. I am planning to check the malloc’d block before the function finishes and returns. If the heap-allocated memory is not captured (meaning that the pointer to the block is not returned or not passed on to the other function), but is not freed by explicit ‘free’ call, then this pass will automatically deallocate the memory in the heap.

For example,

```
void foo(int n) {
        int *a = malloc(n*sizeof(int));
        for(int i = 0; i<10; i++) {
                a[i] = i;
        }
}
Int main() {
        foo();
        (not used)
        int *x = malloc(100 * sizeof(int));
        return 0;
}
```

In this case, n * 4 bytes are allocated in foo by malloc, but it is not used again after the ‘foo’ function, thus being a garbage. Because of this garbage, not only the max heap usage will be bigger, but also the head movement will cost more because of the useless allocation of the n*4 bytes in the heap.

Condition for garbage :

1. Heap-allocated memory should not be used or be referenced outside the function.
   - If the heap-allocated memory is used or referenced outside the function, then deallocating it at the end of the function will obviously cause the error. Actually, this is basic condition of the garbage.
2. Heap-allocated memory should not be freed in the same function.
   - If it is already freed in the same function, it is neither possible to deallocate, nor needed to be deallocated.

Expected decrease in the cost :

1. max heap usage
   - ❑ memory allocated in heap will decrease because not captured memory will be automatically removed instead of remaining
2. Head movement Cost
   - ❑ cost of moving head between heap and stack, or moving head in stack will decrease.

3. Increase in cost due to additional deallocation instructions
    - ❑ keep in mind that due to the additional deallocation instruction, cost will increase by 1 * (# of newly added free inst)
    - ❑ But we can dramatically save the heap usage.

ii. Algorithm in Pseudo-code

```
for every function F in given program:
    for every instruction I in F:
        If ( I is call to malloc ) {
            ptr = pointer to heap-allocated malloc
            Check if ptr is  freed in the function
            Check if ptr is returned from F or used in other functions in the module
            If (ptr is not freed && not returned && not used in other functions)
                    Add IR free instruction in the successor exit blocks  after all uses of ptr
        }
```

iii. Three IR Programs

<table>
<tr><th>Before Optimization</th><th>After Optimization</th></tr>
<tr><td colspan="2">1. Completely not-freed malloc</td></tr>
<tr><td><pre>
define i32 foo(i64 %n) {
BB1 :
 %call = call i8* @malloc(i64 %n)
 %0 = bitcast i8* %call to i64*
 br label %BB2
BB2 :
 ...

BBexit :
 %retval = i64 0
 ret i64 %retval
}

define i32 main() {
 ...
 %a = call @foo(i64 %n)
 (no use of malloc-ed memory)
 ...
 ret i64 0
}
</pre></td><td><pre>
define i32 foo(i64 %n) {
BB1 :
 %call = call i8* @malloc(i64 %n)
 %0 = bitcast i8* %call to i64*
 br label %BB2
BB2 :
 ...

BBexit :
 %retval = i64 0
 %10 = bitcast i64* %0 to i8*
 call void @free(i8* %10)
 ret i64 %retval
}

define i32 main() {
 ...
 %a = call @foo(i64 %n)
 ...
 ret i64 0
}
</pre></td></tr>
</table>

<table>
<tr><td></td><td></td></tr>
<tr><td colspan="2">2. Partly not-freed malloc</td></tr>
<tr><td>define void foo() {
BB1 :
%call = call i8* @malloc(i64 32)
%0 = bitcast i8* %call to i32*
…
br i1 %cond, label %if.then, %labe %if.else
if.then :
%2 = bitcast i32* %0 to i8*
call void @free(i8* %2)
br label %if.end
if.else :
br label %if.end
if.end :
ret void
}

define i32 main() {
…
%a = call @foo(i64 %n)
(no use of malloc-ed memory)
...
ret i64 0
}</td><td>define void foo() {
BB1 :
%call = call i8* @malloc(i64 32)
%0 = bitcast i8* %call to i32*
…
br i1 %cond, label %if.then, %labe %if.else
if.then :
%2 = bitcast i32* %0 to i8*
call void @free(i8* %2)
br label %if.end
if.else :
%3 = bitcast i32* %0 to i8*
call void @free(i8* %3)
br label %if.end
if.end :
ret void
}

define i32 main() {
…
%a = call @foo(i64 %n)
(no use of malloc-ed memory)
...
ret i64 0
}</td></tr>
<tr><td colspan="2">3. Consequent malloc but not freed</td></tr>
<tr><td>define void malloc1() {
BB1 :
%call.1 = call i8* @malloc(i64 2000)
…
(not freed)
ret i64 0
}

define void malloc2() {
BB1 :
%call.2 = call i8* @malloc(i64 2000)
…
(not freed)
ret i64 0
}</td><td>define void malloc1() {
BB1 :
%call.1 = call i8* @malloc(i64 2000)
…
call void @free(i8* %call.1)
ret i64 0
}

define void malloc2() {
BB1 :
%call.2 = call i8* @malloc(i64 2000)
…
call void @free(i8* %call.2)
ret i64 0
}</td></tr>
</table>

define void malloc3() {
BB1 :
 %call.3 = call i8* @malloc(i64 2000)
 ...
 (not freed)
 ret i64 0
}
 	
define i32 main() {
 ...
 call @malloc1( )
 call @malloc2( )
 call @malloc3()
 ret i64 0
}
	define void malloc3() {
	BB1 :
	 %call.3 = call i8* @malloc(i64 2000)
	 ...
	 call void @free(i8* %call.3)
	 ret i64 0
	}
	
	define i32 main() {
	 ...
	 call @malloc1( )
	 call @malloc2( )
	 call @malloc3()
	 ret i64 0
	}

8. 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 part1).

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

<table>
<tr><td colspan="2" align="center">1. Case of extracting the next block</td></tr>
<tr><td>Entry:
    (a lot of register usage)
    %br next_block
Next_block:
    (a lot of register usage)</td><td>Entry:
    (a lot of register usage)
    %r = call @newFunc(dependent regs)
    ret %r
Define … newFunc(...):
    (a lot of register usage 4)
    ret %reg</td></tr>
<tr><td colspan="2" align="center">2. Case of extracting two blocks (br branches)</td></tr>
<tr><td>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)

    ....</td><td>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</td></tr>
<tr><td colspan="2" align="center">3. Extracting block into multiple blocks</td></tr>
</table>

Big block:	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	(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

9. 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;
}
```

10. 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.

11. Dead Argument Elimination Pass

i. Description

Dead Argument is an argument that is passed into the function but is never referred to in the function. Specifically, if the function does not read the argument nor change the argument, that argument can be considered as a dead argument. We can eliminate the dead argument from the function and thereby reduce the cost by 2 per every call to the function.

For simplicity, we are going to consider only the argument that is not mentioned in the function itself as a dead argument. For example,

Before Optimization	After Optimization

int f(int arg1, int arg2) {
    return arg1;
}
 	
void main() {
    int a = 3;
    int b = 5;
 	
    x = f(a, b);
}
	int f(int arg1) {
	    return arg1;
	}
	
	void main() {
	    int a = 3;
	    int b = 5;
	
	    x = f(a);
	}

In this case, ‘arg2’ of function f is not mentioned at all in the function and thus is treated as a dead argument. This can be eliminated.

For the implementation, we are going to make use of DeadArgumentEliminationPass which is already implemented, but revise it to fit better in our spec.

12. Induction Variable Strength Reduction Pass (IVSR Pass)

i. Description

Induction Variables (a.k.a loop counters) are variables in a loop, whose value is a function of the loop iteration number. It can be detected in LLVM IR using LoopInfo::isAuxiliaryInductionVariable. In this special project compiler case, integer multiplication and division are the cheapest, cheaper than integer add and subtraction and integer shift/ logical operations. The object of this pass is to optimize loops by rewriting the induction variables to use cheaper operation. (strength reduction)

For simplicity, in this pass we will **replace expensive additions in loops to cheaper multiplications using invariant variables.**

Addition in loops can be optimized to induction variable multiplication. Such cases can seem rather unnatural, but they often happen after other optimizations or in more complex codes.