<Project Documentation A>

SWPP 2020 Spring LLVM Project
Version 1 (2020.05.02)

Team 4
Hyoung Uk Sul
Ahyoung Oh
Jaewook Lee
Jaeeun Lee

A. Requirement and Specification

Sprint 1

1. Function Outlining

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
1. Simple case of optimization	
NormalBlock: %0 = ... %1 = ... %10 = ... br label Breakeven Breakeven: %11 = ... %20 = ... ret ... %20	NormalBlock: %0 = ... %1 = ... %10 = ... %r = call @newFunc(dependent regs) ret ... %r define ... newFunc(...): %11 = ... %20 = ... ret ... %20
2. 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
3. Branched optimization	
Block1:	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 Block4: (a lot of register usage 4) ret %reg	(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) %r = call @newFunc(dependent regs) ret %r Define ... newFunc(...): (a lot of register usage 4) ret %reg

2. 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.

For simplicity, we limit the target 'malloc operation' to those that are freed in the same function. 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.

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(heap);
	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.

Other possible optimizations that can be done in this pass:

1) malloc to global variable(in beginning of the heap)
The address of the global variable is the beginning of the heap. so you can easily access it without the cost of moving head significantly. You can simply reset the head.

2) global variable to local variable(in stack)
You do not have to access heap to get global variables. After the optimization, global variables that once used to be stored in the beginning of the heap will be stored in the stack and the cost of access is decreased.

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 && and I malloc size is lower than 2048 bytes )
            If ( malloc'd pointer is freed in F)
                Replace I with alloca instruction;
                Remove call to free;
                Replace all uses of malloc'd pointer in F to alloc'd pointer;
```

iii. 3 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,
	// 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
	…

```
i64 0
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() 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	define void G() entry: %p = alloca [4 x i32], align 4 ... br ... ... exit: ... ret void

3. Arithmetic Optimizations

i. Description

This optimization pass customizes [include/llvm/Transforms/InstCombine/InstCombine.h] which combines redundant instructions so that it fits our spec. 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. Therefore, shift/logical and add/sub operations need to be transformed to multiplication/division operations when possible. Also, in special cases, bitwise operator AND(0.8 cost) could be transformed into modulo operations.

To prevent overflow, we need to take care of signed integer limits.

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:
            If (I matches m_Shl(x,i))
                Replace I with m_Mul(x, 2^i)
```

```
If(I matches m_Add(x,x))
        Replace with m_Mul(x,2)

If(I matches m_And(x,i) && i == (7, 63, 2047, ...))
        Replace with m_uRem(x,8/64/2048/...)
```

iii. 3 IR Programs

Before Optimization	**After Optimization**
1. Replace addition with multiplication	
... %a = add i32 %x, %x ...	... %a = mul i32 %x, 2 ...
2. Replace shift with multiplication	
... %a = shl i32 %x, 2 ...	... %a = mul i32 %x, 4 ...
3. Replace logical “and” with urem, and constant-fold multiplying by 0.	
... %a = and i32 %x, 7 %b = mul i32 %a, 0 %c = ... i32 %a, %b ...	... %a = urem i32 %x, 8 %c = ... i32 %a, 0 ...
4. Eliminate meaningless operations	
... %a = add i32 %x, 0 %b = sub i32 %y, 0 ...	... Replace %a with %x Replace %b with %y ...

4. Packing Register Pass

i. Description

This pass reduces the cost of multiple load instructions by “packing” registers. Multiple **load** instructions with small values can be reduced to calling less loads, and instead calling **udiv** or **urem.** The cost of such

optimization can be reduced, as load instruction has the cost of 2(in stack) or 4(in heap) while udiv or urem instructions have a cost of 0.6.
This optimization requires an additional register, and we use the register as a buffer.

ii. Algorithm in pseudo-code

```
if (Loading 4 values from an array sequentially )
        reg.0 = Load 8 sp 0
        for (i=4; i>1; i--){
                reg.i = urem reg.0 65536
                reg.0= udiv  reg.0  65536
        }
```

Before optimization:	->	**After optimization:**
reg1 = load 2 sp 0		reg0 = load 8 sp 0
reg2= load 2 sp 1		reg4 = urem reg0 2^16
reg3= load 2 sp 2		reg0 = udiv reg0 2^16
reg4= load 2 sp 3		reg3 = urem reg0 2^16
		reg0 = udiv reg0 2^16
		reg2 = urem reg0 2^16
		reg0 = udiv reg0 2^16
		reg1 = urem reg0 2^16

iii. 3 IR Programs

Before Optimization	**After Optimization**
1. Simple array element addition	
define void @add(i16* %numbers) { entry: **%1 = load i16, i16* %numbers, align 2** %arrayidx1 = getelementptr inbounds i16, i16* %numbers, i64 1 **%2 = load i16, i16* %arrayidx1, align 2** %arrayidx2 = getelementptr inbounds i16, i16* %numbers, i64 2 **%3 = load i16, i16* %arrayidx2, align 2** %arrayidx3 = getelementptr inbounds i16, i16* %numbers, i64 3 **%4 = load i16, i16* %arrayidx3, align 2** %add = add i16 %2, %1	define void @add(i16* %numbers) { entry: **%0 = load i64, %sp 0** **%4 = urem i16 %0. 65536** **%0 = udiv i16 %0. 65536** **%3 = urem i16 %0. 65536** **%0 = udiv i16 %0. 65536** **%2 = urem i16 %0. 65536** **%0 = udiv i16 %0. 65536** **%1 = urem i16 %0. 65536** %add = add i16 %2, %1 %add6 = add i16 %add, %3 %add8 = add i16 %add6, %4

<table>
<tr><td>

```
 %add6 = add i16 %add, %3
 %add8 = add i16 %add6, %4
 store i16 %add8, i16* @answer, align 2
 ret void
}
```

</td><td>

```
 store i16 %add8, i16* @answer, align 2
 ret void
}
```

</td></tr>
<tr><td colspan="2">2. Load in unrolled loops</td></tr>
<tr><td>

```
define signext i16 @average(i16* %arr, i16 signext
%size){
entry:
          ...
 %indvars.iv = phi i64 [ %0, %for.body7.preheader
], [ %indvars.iv.next, %for.body7 ]
 %average.149 = phi i32 [ %average.0.lcssa,
%for.body7.preheader ], [ %add26, %for.body7 ]
 %arrayidx9 = getelementptr inbounds i16, i16*
%arr, i64 %indvars.iv
 %3 = load i16, i16* %arrayidx9, align 2
 %conv10 = sext i16 %3 to i32
 %4 = add nuw nsw i64 %indvars.iv, 1
 %arrayidx13 = getelementptr inbounds i16, i16*
%arr, i64 %4
 %5 = load i16, i16* %arrayidx13, align 2
 %conv14 = sext i16 %5 to i32
 %6 = add nuw nsw i64 %indvars.iv, 2
 %arrayidx18 = getelementptr inbounds i16, i16*
%arr, i64 %6
 %7 = load i16, i16* %arrayidx18, align 2
 %conv19 = sext i16 %7 to i32
 %8 = add nuw nsw i64 %indvars.iv, 3
 %arrayidx23 = getelementptr inbounds i16, i16*
%arr, i64 %8
 %9 = load i16, i16* %arrayidx23, align 2
 ...

}
```

</td><td>

```
define signext i16 @average(i16* %arr, i16
signext %size){

 // Only Relevant Instructions

//  Equivalent to the top  example, where the 4
loads turn into one load and 8 urem, udiv calls.

 ...
 %0 = load i64, %sp 0
 %9 = urem i16 %0. 65536
 %0 = udiv i16 %0. 65536
 %7 = urem i16 %0. 65536
 %0 = udiv i16 %0. 65536
 %5 = urem i16 %0. 65536
 %0 = udiv i16 %0. 65536
 %3 = urem i16 %0. 65536

...
```

</td></tr>
<tr><td colspan="2">3. Handling 8 load instructions</td></tr>
<tr><td>

```
define signext i16 @big_sum(i16* %arr)  {
entry:
 %1 = load i16, i16* %arr, align 2
 %arrayidx1 = getelementptr inbounds i16, i16*
%arr, i64 1
 %2 = load i16, i16* %arrayidx1, align 2
```

</td><td>

```
define signext i16 @big_sum(i16* %arr){

 // Only Relevant Instructions

//  After: Also equivalent to the top  example 1 ,
```

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

%arrayidx2 = getelementptr inbounds i16, i16* %arr, i64 2 **%3 = load i16, i16* %arrayidx2, align 2** %arrayidx3 = getelementptr inbounds i16, i16* %arr, i64 3 **%4 = load i16, i16* %arrayidx3, align 2** %arrayidx4 = getelementptr inbounds i16, i16* %arr, i64 4 **%6 = load i16, i16* %arrayidx4, align 2** %arrayidx5 = getelementptr inbounds i16, i16* %arr, i64 5 **%7 = load i16, i16* %arrayidx5, align 2** %arrayidx6 = getelementptr inbounds i16, i16* %arr, i64 6 **%8 = load i16, i16* %arrayidx6, align 2** %arrayidx7 = getelementptr inbounds i16, i16* %arr, i64 7 **%9 = load i16, i16* %arrayidx7, align 2** %add = add i16 %1, %0 %add10 = add i16 %add, %2 %add12 = add i16 %add10, %3 %add14 = add i16 %add12, %4 %add16 = add i16 %add14, %5 %add18 = add i16 %add16, %6 %add20 = add i16 %add18, %7 ret i16 %add20 }	where this time, the 8 loads turn into two loads, each with 8 urem, udiv calls. So this example will use 2 extra registers. **%0 = load i64, %sp 0** **%4 = urem i16 %0. 65536** **%0 = udiv i16 %0. 65536** **%3 = urem i16 %0. 65536** **%0 = udiv i16 %0. 65536** **%2 = urem i16 %0. 65536** **%0 = udiv i16 %0. 65536** **%1 = urem i16 %0. 65536** **%5 = load i64, %sp 0** **%9 = urem i16 %0. 65536** **%5 = udiv i16 %0. 65536** **%8 = urem i16 %0. 65536** **%5 = udiv i16 %0. 65536** **%7 = urem i16 %0. 65536** **%5 = udiv i16 %0. 65536** **%6 = urem i16 %0. 65536**

Sprint 2

5. Tail Call Elimination

i. Description

This optimization does 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 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.

6. 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;
	int f(int arg1) {
	return arg1;
	}
	
	void main() {
	int a = 3;
	int b = 5;

x = f(a, b); }	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.

7. 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 difference between desired address and previous address gets bigger when one is in the stack area and the other is in the heap area, alternation between heap accesses and stack accesses should be minimized and the allocations should be reordered in order to make traversal as linear as possible.

8. 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.

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. Register Allocation Pass

i. Description

This pass is to optimize the allocation of IR registers to the assembly registers. There are unlimited number of IR registers but only 16 registers and one stack register(sp) in our backend assembly language. Therefore, we need an algorithm to allocate the IR registers to assembly registers. Basically, first of all, all IR registers will be converted to alloca which means that all read/write instructions to the

IR registers become load/store instructions. Then, these instructions' names will be changed according to the allocation to assembly registers.

Currently, this allocation is performed by a naive algorithm. However, by using more efficient allocation algorithm, we can reduce the number of alloca instructions and convert them into the assembly registers which will result in lower cost. Our objective will be to assign the register names carefully so that we can cover more IR registers with less assembly registers.

11. Lowering Allocas to IR registers

i. Description

This optimization pass removes the local occurrence of store and load instructions. To be specific, when store and load instructions occur within identical function scope, we replace it with registers. By doing this, we can reduce costly memory operations.

To check whether the alloca meets the condition to be promoted to a register, we could use an existing function called isAllocaPromotable in [llvm/include/llvm/Transforms/Utils/PromoteMemToReg.h]. This function ensures that there are only loads, stores, and lifetime markers using this alloca and enforces that there is only ever one layer of bitcasts or GEPs between the alloca and the lifetime markers. If this function returns true, we could use another function named PromoteMemToReg to promote Alloca to register. This function would promote the specified list of alloca instructions into scalar registers, inserting PHI nodes as appropriate. This also makes uses of Dominance Frontier info. We need to note that CFG of the function is not modified at all.

Before Optimization	After Optimization
%0 = (some instruction) ... %ptr = alloca i32 store i32 %0, i32* %ptr ... %val = load i32, i32* %ptr %sum = add i32 %val, 10 ...	%a = 20 %res =add %a, 10 store %res

12. Loop Invariant Code Motion Pass (LICM Pass)

i. Description

Loop-invariant code consists of statements or expressions which can be moved outside the body of a loop without affecting the semantics of the program.

For example,

```
int i = 0;
while (i < n) {
   x = p + q;
   a[i] = 6 * i + x * x;
   ++i;
}
```

In the above code, two invariants, x=p+q and x*x is loop invariant.

We plan to find the loop invariant instructions using the dominance tree, and traverse (pre-order traversal to accumulate the invariants) the dominance tree to find out if an instruction is invariant or not.

After accumulating all the loop invariant instructions, we can delete the instructions and move it to the end of the preheader, before the branch. We can use the method **moveBefore** in Instruction.h.

We also plan to check if the instruction is safe to hoist, by checking if it has no side effects (exception, traps) and that the basic block containing the instruction dominates all exit blocks for the loop.