[SWPP] Team 7

배수민, 이상민, 김주연, 이철호

2024.04.25

List of Optimizations

1. Branch in Loop Optimization
2. Consecutive Branch Optimization
3. Loop Unrolling
4. Tail Call Optimization
5. Shift to Multiplication/Division
6. Add/Sub to Increment/Decrement
7. Vectorization

1. Branch in Loop Optimization

Conditional Branch	br <reg_cond> <true_bb> <false_bb>	90 for true_bb 30 for false_bb

↓

여러 번 실행되는 branch instruction에서
true로 가는 경우가 false로 가는 경우보다 많은 경우
두 경우를 바꿔준다.

1. Branch in Loop Optimization

```
define void @loop_example() {
entry:
    br label %loop_body

loop_body:
    %counter = phi i32 [ 0, %entry ], [ %new_counter, %loop_body ]

    ; 루프 본문
    ; …

    ; 루프 카운터 증가
    %new_counter = add i32 %counter, 1

    ; 반복이 100번 되었는지 검사
    %cmp = icmp slt i32 %counter, 100
    br i1 %cmp, label %loop_body, label %loop_exit

loop_exit:
    ret void
}
```

1. Branch in Loop Optimization

```
define void @loop_example() {
entry:
    br label %loop_body

loop_body:
    %counter = phi i32 [ 0, %entry ], [ %new_counter, %loop_body ]

    ; 루프 본문
    ; …

    ; 루프 카운터 증가
    %new_counter = add i32 %counter, 1

    ; 반복이 100번 되었는지 검사
    %cmp = icmp slt i32 %counter, 100
    br i1 %cmp, label %loop_body, label %loop_exit

loop_exit:
    ret void
}
```

→

```
%cmp = icmp sge i32 %counter, 100
br i1 %cmp, label %loop_exit, label %loop_body
```

2. Consecutive Branch Optimization

Conditional Branch	`br <reg_cond> <true_bb> <false_bb>`	90 for `true_bb` 30 for `false_bb`
Switch Instruction	`switch <reg_cond> <cst1> <bb1> .. <default_bb>`	60

Branch (equality check) instruction 이 연달아 있는 경우,
switch instruction 으로 수정하여
평균 cost 를 줄인다.

2. Consecutive Branch Optimization

```
define i32 @example(i32 %x) {
entry:
    %cmp1 = icmp eq i32 %x, 1
    br i1 %cmp1, label %case1, label %else_if1
case1:
    ret i32 10

else_if1:
    %cmp2 = icmp eq i32 %x, 2
    br i1 %cmp2, label %case2, label %else_if2
case2:
    ret i32 20

else_if2:
    %cmp3 = icmp eq i32 %x, 3
    br i1 %cmp3, label %case3, label %default
case3:
    ret i32 30

default:
    ret i32 -1
}
```

2. Consecutive Branch Optimization

```
define i32 @example(i32 %x) {
entry:
    %cmp1 = icmp eq i32 %x, 1
    br i1 %cmp1, label %case1, label %else_if1
case1:
    ret i32 10

else_if1:
    %cmp2 = icmp eq i32 %x, 2
    br i1 %cmp2, label %case2, label %else_if2
case2:
    ret i32 20

else_if2:
    %cmp3 = icmp eq i32 %x, 3
    br i1 %cmp3, label %case3, label %default
case3:
    ret i32 30

default:
    ret i32 -1
}
```

→

```
define i32 @example(i32 %x) {
entry:
    switch i32 %x, label %default [
        i32 1, label %case1
        i32 2, label %case2
        i32 3, label %case3
    ]

case1:
    ret i32 10

case2:
    ret i32 20

case3:
    ret i32 30

default:
    ret i32 -1
}
```

3. Loop Unrolling

Conditional Branch	`br <reg_cond> <true_bb> <false_bb>`	90 for `true_bb` 30 for `false_bb`

↓

For loop에서 반복되는 branch instruction으로 인한
cost를 줄이기 위해 loop unrolling을 통해
실행 중 불리는 branch instruction 수를 줄인다.

3. Loop Unrolling

```
define void @loop_example() {
entry:
    br label %for.cond

for.cond:
    %counter = phi i32 [ 0, %entry ], [ %new_counter, %for.body ]
    %cmp = icmp slt i32 %counter, 100
    br i1 %cmp, label %for.body, label %for.end

for.body:
    call void @print_int(i32 %counter)
    %new_counter = add i32 %counter, 1

    br label %for.cond

for.end:
    ret void
}
```

3. Loop Unrolling

```
define void @loop_example() {
entry:
    br label %for.cond

for.cond:
    %counter = phi i32 [ 0, %entry ], [ %new_counter2, %for.body ]
    %cmp = icmp slt i32 %counter, 50
    br i1 %cmp, label %for.body, label %for.end

for.body:
    call void @print_int(i32 %counter)
    %new_counter1 = add i32 %counter, 1
    call void @print_int(i32 %new_counter1)
    %new_counter2 = add i32 %new_counter1, 1

    br label %for.cond

for.end:
    ret void
}
```

100번 → 50번 반복

4. Tail Call Optimization

Function call	`call <fname> <reg1> .. <regN>` `<reg_ret> = call <fname> <reg1> .. <regN>`	30
Recursive function call	`rcall <reg1> .. <regN>` `<reg_ret> = rcall <reg1> .. <regN>`	10

↓

`rcall`의 cost가 더 낮으므로
tail recursive function에서 함수를 호출할 때
`call`이 아닌 `rcall`을 사용하도록 수정한다.

4. Tail Call Optimization

gcd.ll

```
define dso_local i64 @gcd(i64 noundef %x, i64 noundef %y) #0 {
…
if.end7:                                          ; preds = %if.else, %if.then5
     %x.addr.0 = phi i64 [ %rem, %if.then5 ], [ %x, %if.else ]
      %y.addr.0 = phi i64 [ %y, %if.then5 ], [ %rem6, %if.else ]
      %call = call i64 @gcd(i64 noundef %x.addr.0, i64 noundef %y.addr.0)
      br label %return
return:                                           ; preds = %if.end7, %if.then2, %if.then
  %retval.0 = phi i64 [ %y, %if.then ], [ %x, %if.then2 ], [ %call, %if.end7 ]
  ret i64 %retval.0
}
```

gcd.s

```
.if.end7:
r1 = call gcd r1 r2
br .return
.return:
ret r1
```

4. Tail Call Optimization

gcd.ll

```
define dso_local i64 @gcd(i64 noundef %x, i64 noundef %y) #0 {
…
if.end7:                                          ; preds = %if.else, %if.then5
     %x.addr.0 = phi i64 [ %rem, %if.then5 ], [ %x, %if.else ]
      %y.addr.0 = phi i64 [ %y, %if.then5 ], [ %rem6, %if.else ]
      %call = musttail call i64 @gcd(i64 noundef %x.addr.0, i64 noundef %y.addr.0)
      ret i64 %call
return:                                           ; preds = %if.end7, %if.then2, %if.then
  %retval.0 = phi i64 [ %y, %if.then ], [ %x, %if.then2 ]
  ret i64 %retval.0
}
```

gcd.s

```
.if.end7:
r1 = rcall r1 r2
ret r1
.return:
ret r1
end gcd
```

5. Shift to Multiplication/Division

scalar

Integer
Multiplication/Division
	`<reg> = udiv <reg1> <reg2> <bw>`
	`<reg> = sdiv <reg1> <reg2> <bw>`
	`<reg> = urem <reg1> <reg2> <bw>`
	`<reg> = srem <reg1> <reg2> <bw>`
	`<reg> = mul <reg1> <reg2> <bw>`
	`<bw> := 1|8|16|32|64`
		1

Integer Shift
- shl: shift-left
- lshr: logical shift-right
- ashr: arithmetic shift-right
	`<reg> = shl <reg1> <reg2> <bw>`
	`<reg> = lshr <reg1> <reg2> <bw>`
	`<reg> = ashr <reg1> <reg2> <bw>`
	`<bw> := 1|8|16|32|64`
		4

vector

Integer
Multiplication/Division
	`<vr> = vudiv <vr1> <vr2> <bw>`
	`<vr> = vsdiv <vr1> <vr2> <bw>`
	`<vr> = vurem <vr1> <vr2> <bw>`
	`<vr> = vsrem <vr1> <vr2> <bw>`
	`<vr> = vmul <vr1> <vr2> <bw>`
	`<bw> := 32|64`
		2

Integer Shift
- shl: shift-left
- lshr: logical shift-right
- ashr: arithmetic shift-right
	`<vr> = vshl <vr1> <vr2> <bw>`
	`<vr> = vlshr <vr1> <vr2> <bw>`
	`<vr> = vashr <vr1> <vr2> <bw>`
	`<bw> := 32|64`
		8

↓

Shift operation의 cost가 높으므로
multiplication/division operation으로 대체한다.

5. Shift to Multiplication/Division

```
define dso_local i32 @countSetBits(i32 noundef %n) #0 {
entry:
  br label %while.cond

while.cond:                                       ; preds = %while.body, %entry
  %n.addr.0 = phi i32 [ %n, %entry ], [ %shr, %while.body ]
  %count.0 = phi i32 [ 0, %entry ], [ %add, %while.body ]
  %tobool = icmp ne i32 %n.addr.0, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:                                       ; preds = %while.cond
  %and = and i32 %n.addr.0, 1
  %add = add i32 %count.0, %and
  %shr = lshr i32 %n.addr.0, 1
  br label %while.cond, !llvm.loop !5

while.end:                                        ; preds = %while.cond
  ret i32 %count.0
}
```

```
udiv i32 %n.addr.0, 2
```

6. Add/Sub to Increment/Decrement

Integer Add/Sub	`<reg> = add <reg1> <reg2> <bw>`
	`<reg> = sub <reg1> <reg2> <bw>`
	`<bw> := 1|8|16|32|64`
		5

Integer increment <reg> = <reg> + 1	`<reg> = incr <reg1> <bw>` `<bw> := 1|8|16|32|64`	1
Integer decrement <reg> = <reg> - 1	`<reg> = decr <reg1> <bw>` `<bw> := 1|8|16|32|64`	1

↓

Add/Sub의 cost가 높으므로
Increment/Decrement를 여러 번 하도록 수정한다.

6. Add/Sub to Increment/Decrement

```
define dso_local i32 @collatz(ptr noundef %iter, i32 noundef %n) #0 {
...<code>...
if.end6:                                         ; preds = %if.end
  %1 = load i16, ptr %iter, align 2
  %conv7 = sext i16 %1 to i32
  %add = add nsw i32 %conv7, 1
  %conv8 = trunc i32 %add to i16
  store i16 %conv8, ptr %iter, align 2
  %rem = urem i32 %n, 2
  %cmp9 = icmp eq i32 %rem, 0
  br i1 %cmp9, label %cond.true, label %cond.false
...<code>...
```

```
%add = call i32 @incr_i32(i32 %conv7)
```

7. Vectorization

`load`
`store`
scalar arithmetic
	→	`vload`
		`vstore`
		vector arithmetic

7. Vectorization

unroll 되어있는 loop
(load, add, store 4번씩 반복)

```
define void @vectorized_add(i64* %a, i64* %b, i64* %c, i64 %n) {
…. <code > …
for.body:                                         ; preds = %for.cond
  %mul = mul i64 %i.0, 4
  %arrayidx = getelementptr inbounds i64, ptr %a, i64 %mul
  %0 = load i64, ptr %arrayidx, align 8
  %mul1 = mul i64 %i.0, 4
  %arrayidx2 = getelementptr inbounds i64, ptr %b, i64 %mul1
  %1 = load i64, ptr %arrayidx2, align 8
  %add = add i64 %0, %1
  %mul3 = mul i64 %i.0, 4
  %arrayidx4 = getelementptr inbounds i64, ptr %c, i64 %mul3
  store i64 %add, ptr %arrayidx4, align 8
  %mul5 = mul i64 %i.0, 4
  %add6 = add i64 %mul5, 1
  %arrayidx7 = getelementptr inbounds i64, ptr %a, i64 %add6
  %2 = load i64, ptr %arrayidx7, align 8
  %mul8 = mul i64 %i.0, 4
  %add9 = add i64 %mul8, 1
  %arrayidx10 = getelementptr inbounds i64, ptr %b, i64 %add9
  %3 = load i64, ptr %arrayidx10, align 8
  %add11 = add i64 %2, %3
  %mul12 = mul i64 %i.0, 4
  %add13 = add i64 %mul12, 1
  %arrayidx14 = getelementptr inbounds i64, ptr %c, i64 %add13
  store i64 %add11, ptr %arrayidx14, align 8
  %mul15 = mul i64 %i.0, 4
  %add16 = add i64 %mul15, 2
  %arrayidx17 = getelementptr inbounds i64, ptr %a, i64 %add16
```

```
  %4 = load i64, ptr %arrayidx17, align 8
  %mul18 = mul i64 %i.0, 4
  %add19 = add i64 %mul18, 2
  %arrayidx20 = getelementptr inbounds i64, ptr %b, i64 %add19
  %5 = load i64, ptr %arrayidx20, align 8
  %add21 = add i64 %4, %5
  %mul22 = mul i64 %i.0, 4
  %add23 = add i64 %mul22, 2
  %arrayidx24 = getelementptr inbounds i64, ptr %c, i64 %add23
  store i64 %add21, ptr %arrayidx24, align 8
  %mul25 = mul i64 %i.0, 4
  %add26 = add i64 %mul25, 3
  %arrayidx27 = getelementptr inbounds i64, ptr %a, i64 %add26
  %6 = load i64, ptr %arrayidx27, align 8
  %mul28 = mul i64 %i.0, 4
  %add29 = add i64 %mul28, 3
  %arrayidx30 = getelementptr inbounds i64, ptr %b, i64 %add29
  %7 = load i64, ptr %arrayidx30, align 8
  %add31 = add i64 %6, %7
  %mul32 = mul i64 %i.0, 4
  %add33 = add i64 %mul32, 3
  %arrayidx34 = getelementptr inbounds i64, ptr %c, i64 %add33
  store i64 %add31, ptr %arrayidx34, align 8
  br label %for.inc
}
```

7. Vectorization

```
define void @vectorized_add(i64* %a, i64* %b, i64* %c, i64 %n) {
... <code> ...

vector_loop:
  %i = phi i64 [0, %entry], [%next_i, %vector_loop]
  %next_i = add i64 %i, 4

  %vec_a_ptr = getelementptr i64, i64* %a, i64 %i
  %vec_b_ptr = getelementptr i64, i64* %b, i64 %i
  %vec_c_ptr = getelementptr i64, i64* %c, i64 %i

  %vec_a = load <4 x i64>, <4 x i64>* %vec_a_ptr, align 32
  %vec_b = load <4 x i64>, <4 x i64>* %vec_b_ptr, align 32
  %vec_c = add <4 x i64> %vec_a, %vec_b

  store <4 x i64> %vec_c, <4 x i64>* %vec_c_ptr, align 32

  %loop_cond = icmp slt i64 %next_i, %n2
  br i1 %loop_cond, label %vector_loop, label %scalar_remainder

... <code> ...

}
```

4번의 load, store,
scalar arithmetic operation
→ 1번의 vload, vstore,
vector arithmetic operation

THANK YOU