IR optimization proposal

Team8 – 안재우, 고은, 조호연, 조휘준

Recursive call optimization

- Implementation
  attach `musttail` in front of `call` instruction
- Cost reduction
  140 costs reduced in Fibonacci(10) function call

Before optimization

```
define dso_local i32 @function(i32 noundef %n) #0 {
entry:
  %cmp = icmp slt i32 %n, 2
  br i1 %cmp, label %if.then, label %if.end
if.then:
  br label %return
if.end:
  %sub = sub nsw i32 %n, 1
  %call = call i32 @function(i32 noundef %sub)
  %mul = mul nsw i32 %n, %call
  br label %return
return:
  %retval.0 = phi i32 [ 1, %if.then ], [ %mul, %if.end ]
  ret i32 %retval.0
}
```

After optimization

```
define dso_local i32 @function(i32 noundef %n) #0 {
entry:
  %cmp = icmp slt i32 %n, 2
  br i1 %cmp, label %if.then, label %if.end
if.then:
  br label %return
if.end:
  %sub = sub nsw i32 %n, 1
  %call = musttail call i32 @function(i32 noundef %sub) ; <- musttail
  %mul = mul nsw i32 %n, %call
  br label %return
return:
  %retval.0 = phi i32 [ 1, %if.then ], [ %mul, %if.end ]
  ret i32 %retval.0
}
```

Shift operation optimization

- Implementation

  Replace logical shift right operation to unsigned division operation

  x >> y -> x / 2^y

  lshr x y -> udiv x 2^y

- Cost reduction

  95 costs reduced in bitcount program

  with 4,294,967,295 (maximum value of unsigned int)

Before optimization

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

After optimization

```
define dso_local i32 @countSetBits(i32 noundef %n) #0 {
entry:
  br label %while.cond
while.cond:
  %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:
  %and = and i32 %n.addr.0, 1
  %add = add i32 %count.0, %and
  %shr = udiv i32 %n.addr.0, 2 ; <----------
  br label %while.cond, !llvm.loop !5
while.end:
  ret i32 %count.0
}
```

Merge unconditional branch

- Implementation

  Copy & Paste body of a block to it's precedence

- Cost reduction

  30 per `if` branches, and extras

Before optimization

```
if.then:
  %x.0 = mul i32 %n, 3
  %x.1 = add i32 %x.0, 1
  br label %if.end

if.else:
  %x.2 = sdiv i32 %n, 2
  br label %if.end

if.end:
  %res.0 = phi i32 [%x.1, %if.then], [%x.2, %if.else]
  ret i32 %res.0
```

After optimization

```
if.then:
  %x.0 = mul i32 %n, 3
  %x.1 = add i32 %x.0, 1
  ret %x.1

if.else:
  %x.2 = sdiv i32 %n, 2
  ret %x.2
```

Loop unrolling

- Implementation
  merge 8 iterations of loop to once
- Cost reduction
  about 210 per 8 iterations

Before optimization

```
for.cond:
  %res.0 = phi i32 [ 0, %entry ], [ %add, %for.inc ]
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i32 %i.0, %sz
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %arrayidx = getelementptr inbounds i32, ptr %arr, i32 %i.0
  %0 = load i32, ptr %arrayidx, align 4
  %add = add i32 %res.0, %0
  br label %for.inc

for.inc:
  %inc = add i32 %i.0, 1
  br label %for.cond, !llvm.loop !5

for.end:
  ret i32 %res.0
```

After optimization

```
for.unrolled.cond:
  %res.0 = phi i32 [ 0, %entry ], [ %add.7, %for.unrolled.inc ]
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.unrolled.inc ]
  %cmp.0 = icmp ult i64 %i.0, %sz
  br i1 %cmp.0, label %for.unrolled.body, label %for.unrolled.rest.0

for.unrolled.body:
  %arrayidx.0 = getelementptr inbounds i32, ptr %arr, i64 %i.0
  %0 = load i32, ptr %arrayidx.0, align 4
  %add.0 = add i32 %res.0, %0

  %i.1 = add i64 %i.0, 1
  %arrayidx.1 = getelementptr inbounds i32, ptr %arr, i64 %i.1
  %1 = load i32, ptr %arrayidx.1, align 4
  %add.1 = add i32 %add.0, %1

  ...

  %i.7 = add i64 %i.6, 1
  %arrayidx.7 = getelementptr inbounds i32, ptr %arr, i64 %i.7
  %7 = load i32, ptr %arrayidx.7, align 4
  %add.7 = add i32 %add.6, %7

  br label %for.unrolled.inc

for.unrolled.inc:
  %inc = add i64 %i.7, 1
  br label %for.unrolled.cond, !llvm.loop !5
```

```
for.unrolled.rest.0:
  %cmp.1 = icmp ult i64 %i.0, %sz
  br i1 %cmp.1, label %for.unrolled.rest.1, label %for.unrolled.end

for.unrolled.rest.1:
  %arrayidx.rest.0 = getelementptr inbounds i32, ptr %arr, i64 %i.0
  %rest.0 = load i32, ptr %arrayidx.rest.0, align 4
  %res.1 = add i32 %res.0, %rest.0

  %i.rest.1 = add i64 %i.0, 1
  %cmp.2 = icmp ult i64 %i.rest.1, %sz
  br i1 %cmp.2, label %for.unrolled.rest.2, label %for.unrolled.end

...

for.unrolled.rest.7:
  %arrayidx.rest.6 = getelementptr inbounds i32, ptr %arr, i64 %i.rest.6
  %rest.6 = load i32, ptr %arrayidx.rest.6, align 4
  %res.7 = add i32 %res.6, %rest.6

  br label %for.unrolled.end

for.unrolled.end:
  %res = phi i32 [%res.0, %for.unrolled.rest.0], [%res.1, %for.unrolled.rest.1],
                 [%res.2, %for.unrolled.rest.2], [%res.3, %for.unrolled.rest.3],
                 [%res.4, %for.unrolled.rest.4], [%res.5, %for.unrolled.rest.5],
                 [%res.6, %for.unrolled.rest.6], [%res.7, %for.unrolled.rest.7]
  ret i32 %res
```

Load/Write vectorize

- Implementation
  replace consecutive memory load/write to vload/vwrite
- Cost reduction
  180 per 8 `i32` load (stack area)

Before optimize

```
for.unrolled.body:
  %arrayidx.0 = getelementptr inbounds i32, ptr %arr, i64 %i.0
  %0 = load i32, ptr %arrayidx.0, align 4

  %i.1 = add i64 %i.0, 1
  %arrayidx.1 = getelementptr inbounds i32, ptr %arr, i64 %i.1
  %1 = load i32, ptr %arrayidx.1, align 4

  ...

  %i.7 = add i64 %i.0, 7
  %arrayidx.7 = getelementptr inbounds i32, ptr %arr, i64 %i.7
  %7 = load i32, ptr %arrayidx.7, align 4
```

After optimize

```
for.unrolled.body:
  %arrayidx.0 = getelementptr inbounds i32, ptr %arr, i64 %i.0
  %v = load <8 x i32>, ptr %arrayidx.rest.0, align 4

  %0 = extractelement <8 x i32> %v, i32 0
  ...
  %7 = extractelement <8 x i32> %v, i32 7
```

Constant Folding & Trivial Conditioned branch elimination

- Implementation
  Remove instructions that take constants for both operands and replace the variable being assigned with a constant.

  Also remove instructions that returns constant values even if they don't take constants as operands.

Before Optimization

```
unfolded_block1:
    %a = add i32 1, i32
    %b = sub i32 %a, i32
    ret i32 %b

unfolded_block2:
    ; %a is declared before
    %b = icmp eq i32 %a, %a
    ret i1 %b
```

After Optimization

```
folded_block1:
    ret i32 2


folded_block2:
    ret i1 1
```

Reducing Free function call

- Implementation

  Remove free functions that appear after the last malloc of the program flow

Before Optimization

```
intermediate_branch:
  call void @free(ptr noundef %ptr)
  %ptr2 = call ptr @malloc(i64 noundef 24)
  br label %leaf_branch

leaf_branch:
  call void @free(ptr noundef %ptr2)
  ret void
```

After Optimization

```
intermediate_branch:
  call void @free(ptr noundef %ptr)
  %ptr2 = call ptr @malloc(i64 noundef 24)
  br label %leaf_branch

leaf_branch:
  ret void
```

If to ternary operation

- Implementation

  Convert assignment-only if-else branches to ternary operations
- Cost reduction

  119 if `true` is taken, 59 if `false` is taken

Before Optimization

```
entry:
  %call = call i64 (...) @read()
  %cmp = icmp eq i64 %call, 1
  br i1 %cmp, label %if.then, label %if.else

if.then:
  br label %if.end

if.else:
  br label %if.end

if.end:
  %r.0 = phi i64 [ 5, %if.then ], [ 6, %if.else ]
  call void @write(i64 noundef %r.0)
  ret i32 0
```

After Optimization

```
entry:
  %call = call i64 (...) @read() #2
  %cmp = icmp eq i64 %call, 1
  %. = select i1 %cmp, i64 5, i64 6
  call void @write(i64 noundef %.) #2
  ret i32 0
```

Flip conditional branch

- Implementation

  Since taking `false` branch is cheaper, flip the conditional branch that is likely to take `true` branch (e.g. loop conditions)
- Cost reduction

  -60 per successful prediction, +60 per failed prediction

Before Optimization

```
%cmp = icmp eq i32 %a, 0
br i1 %cmp, label %if.then, label %if.else
...
```

After Optimization

```
%cmp = icmp ne i32 %a, 0
br i1 %cmp, label %if.else, label %if.then
...
```

Convert if-else chain to switch

- Implementation
  Convert the form `if(v == const1) { ... } else if (v == const2) { ... }` to switch statement

Before optimization

```
entry:
  %call = call i64 (...) @read()
  %cmp = icmp eq i64 %call, 1
  br i1 %cmp, label %if.then, label %if.else

if.then:
  call void @write(i64 noundef 1)
  br label %if.end4

if.else:
  %cmp1 = icmp eq i64 %call, 2
  br i1 %cmp1, label %if.then2, label %if.else3

if.then2:
  call void @write(i64 noundef 2)
  br label %if.end

if.else3:
  call void @write(i64 noundef 0)
  br label %if.end

if.end:
  br label %if.end4

if.end4:
  ret i32 0
```

After Optimization

```
entry:
  %call = call i64 (...) @read()
  switch i64 %call, label %sw.default [
    i64 1, label %sw.bb
    i64 2, label %sw.bb1
  ]

sw.bb:
  call void @write(i64 noundef 1)
  br label %sw.epilog

sw.bb1:
  call void @write(i64 noundef 2)
  br label %sw.epilog

sw.default:
  call void @write(i64 noundef 0)
  br label %sw.epilog

sw.epilog:
  ret i32 0
```