Optimization idea

Team 11

정원, 여나경, 오규혁, 조영훈, 후슬렝

Optimization specific to asmspec

Load	`<reg> = load <size> <ptr>` `<size> := 1|2|4|8`	Stack area: 20 Heap area: 30 Cost reduced by 90% inside oracle
Store	`store <size> <val> <ptr>` `<size> := 1|2|4|8`	Stack area: 20 Heap area: 30 Cost reduced by 90% inside oracle
Async Load	`<reg> = aload <size> <ptr>` `<size> := 1|2|4|8`	Stack area: 1 Heap area: 1 Cost to resolve Stack area: 24 Heap area: 34 Cannot use inside oracle

Kind	Name	Cost

Integer Multiplication/Division	`<reg> = udiv <val1> <val2> <bw>`
	`<reg> = sdiv <val1> <val2> <bw>`
	`<reg> = urem <val1> <val2> <bw>`
	`<reg> = srem <val1> <val2> <bw>`
	`<reg> = mul  <val1> <val2> <bw>`
	`<bw> := 1|8|16|32|64`
		1

Integer Shift/Logical Operations
- shl: shift-left
- lshr: logical shift-right
- ashr: arithmetic shift-right
	`<reg> = shl  <val1> <val2> <bw>`
	`<reg> = lshr <val1> <val2> <bw>`
	`<reg> = ashr <val1> <val2> <bw>`
	`<reg> = and  <val1> <val2> <bw>`
	`<reg> = or   <val1> <val2> <bw>`
	`<reg> = xor  <val1> <val2> <bw>`
	`<bw> := 1|8|16|32|64`
		4

Integer Add/Sub	`<reg> = add <val1> <val2> <bw>`
	`<reg> = sub <val1> <val2> <bw>`
	`<bw> := 1|8|16|32|64`
		5

Integer Sum	`<reg> = sum <val1> ... <val8> <bw>` `<bw> := 1|8|16|32|64`	10
Integer increment `<reg>` = `<val>` + 1	`<reg> = incr <val> <bw>`	1
Integer decrement `<reg>` = `<val>` - 1	`<reg> = decr <val> <bw>`	1
Comparison - `<cond>` is equivalent to the cond of LLVM IR's icmp	`<reg> = icmp <cond> <val1> <val2> <bw>` `<bw> := 1|8|16|32|64`	1

Unconditional Branch	`br <bbname>`	1
Conditional Branch	`br <condition> <true_bb> <false_bb>`	6 for `true_bb` 1 for `false_bb`
Switch Instruction - `<val1>`, … should be constant integers.	`switch <cond_val> <val1> <bb1> .. <default_bb>`	4

Replace load with aload

load를 aload로 대체하고, load와 사용되는 부분 사이에 관계없는 instruction을 넣고, 가능하다면 aload를 하는 순서와 사용되는 순서가 같게 한다.

```
r1 = load 8 ... ;30
r2 = add r1 ... 64;35
r3 = add ... ... ;40
```

```
r1 = aload 8 ... ;1
r2 = add r1 ... 64;6
r3 = add ... ... ;35
```

```
r2 = load 8 ... ;30
r1 = load 8 ... ;60
write r1 ;63
write r2 ;66
```

```
r1 = aload 8 ... ;1
r2 = aload 8 ... ;2
write r1 ;38
write r2 ;41
```

Replace load with oracle

load를 aload로 최적화했을 때의 cost감소보다 oracle 함수를 사용했을 때의 cost 감소가 더 크다면 oracle 함수를 사용한다.

```
r1 = load 8 ...
…
r2 = load (associated with r1)
…
r3 = load (associated with r2)
```

```
start oracle :
    r1 = load 8 ...
    …
    r2 = load (associated with r1)
    …
    r3 = load (associated with r2)
end oracle
```

Change the exit condition of the loop

conditional branch에서 true cost > false cost이므로

false일 때 loop으로 다시 돌아가고 true일 때 loop을 exit한다.

```
loop:
...
%cmp = icmp slt i32 $1, $n
br il %cmp, label %loop, label %exit
```

```
loop:
...
%cmp = icmp sge i32 $1, $n
br il %cmp, label %exit, label %loop
```

Optimize for some instructions

더 낮은 **cost**로 같은 연산을 제공할 수 있는 경우에 교체한다.

```
r1 = add val1 1 64      ;5      r1 = incr val1 64       ;1
r2 = sub 0 val2 64      ;5      r2 = sub 0 val2 64      ;1
r3 = add val3 val3 64   ;5      r3 = add val3 val3 64   ;1

r4 = shl val4 1 64      ;4      r4 = mul val4 2 64      ;1
r5 = and val5 0 64      ;4      r5 = mul 0 0 64         ;1
r6 = or val6 0 64       ;4      r6 = mul 1 1 64         ;1
```

Optimization general

- `-adce`: Aggressive Dead Code Elimination
- `-always-inline`: Inliner for `always_inline` functions
- `-argpromotion`: Promote 'by reference' arguments to scalars
- `-bb-vectorize`: Basic-Block Vectorization
- `-block-placement`: Profile Guided Basic Block Placement
- `-break-crit-edges`: Break critical edges in CFG
- `-codegenprepare`: Optimize for code generation
- `-constmerge`: Merge Duplicate Global Constants
- `-dce`: Dead Code Elimination
- `-deadargelim`: Dead Argument Elimination
- `-deadtypeelim`: Dead Type Elimination
- `-die`: Dead Instruction Elimination
- `-dse`: Dead Store Elimination
- `-function-attrs`: Deduce function attributes
- `-globaldce`: Dead Global Elimination
- `-globalopt`: Global Variable Optimizer
- `-gvn`: Global Value Numbering
- `-indvars`: Canonicalize Induction Variables
- `-inline`: Function Integration/Inlining
- `-instcombine`: Combine redundant instructions
- `-aggressive-instcombine`: Combine expression patterns
- `-internalize`: Internalize Global Symbols
- `-ipsccp`: Interprocedural Sparse Conditional Constant Propagation
- `-jump-threading`: Jump Threading
- `-lcssa`: Loop-Closed SSA Form Pass
- `-licm`: Loop Invariant Code Motion
- `-loop-deletion`: Delete dead loops
- `-loop-extract`: Extract loops into new functions
- `-loop-extract-single`: Extract at most one loop into a new function
- `-loop-reduce`: Loop Strength Reduction
- `-loop-rotate`: Rotate Loops
- `-loop-simplify`: Canonicalize natural loops
- `-loop-unroll`: Unroll loops
- `-loop-unroll-and-jam`: Unroll and Jam loops
- `-loop-unswitch`: Unswitch loops
- `-lower-global-dtors`: Lower global destructors
- `-loweratomic`: Lower atomic intrinsics to non-atomic form
- `-lowerinvoke`: Lower invokes to calls, for unwindless code generators
- `-lowerswitch`: Lower `SwitchInst`s to branches
- `-mem2reg`: Promote Memory to Register
- `-memcpyopt`: MemCpy Optimization
- `-mergefunc`: Merge Functions
- `-mergereturn`: Unify function exit nodes
- `-partial-inliner`: Partial Inliner
- `-prune-eh`: Remove unused exception handling info
- `-reassociate`: Reassociate expressions
- `-rel-lookup-table-converter`: Relative lookup table converter
- `-reg2mem`: Demote all values to stack slots
- `-sroa`: Scalar Replacement of Aggregates
- `-sccp`: Sparse Conditional Constant Propagation
- `-simplifycfg`: Simplify the CFG
- `-sink`: Code sinking
- `-strip`: Strip all symbols from a module
- `-strip-dead-debug-info`: Strip debug info for unused symbols
- `-strip-dead-prototypes`: Strip Unused Function Prototypes
- `-strip-debug-declare`: Strip all `llvm.dbg.declare` intrinsics
- `-strip-nondebug`: Strip all symbols, except dbg symbols, from a module
- `-tailcallelim`: Tail Call Elimination

https://llvm.org/docs/Passes.html#strip-nondebug-strip-all-symbols-except-dbg-symbols-from-a-module

Remove useless function arguments

cost of function call: 2 + Narg

function에서 실제로 사용되는 argument 수를 X라 했을 때 X < Narg 라면

Narg = X로 바꾸면서, arg # 도 변경

```
start func 5:
BB:
    r1 = add arg1 arg2 64
    r2 = add arg3 arg4 64
    ...; never used arg5
end func
```

```
start func 4:
BB:
    r1 = add arg1 arg2 64
    r2 = add arg3 arg4 64
    ...; never used arg5
end func
```

Dead code elimination

사용하지 않는 코드를 제거

```
start func 2:
BB:
    r1 = add arg1 3 64
    r2 = add arg1 3 64
    r3 = add 4 3 64
    r4 = mul r1 r2 64
    ret 64 r4
```

```
start func 2:
BB:
    r1 = add arg1 3 64
    r3 = add 4 3 64
    r4 = mul r1 r2 64
    ret 64 r4
```

Unswitch loop

loop문의 비교 연산의 수를 줄이거나 없앰

```
function slow (x, y) {
    for (let i = 0; i < 8; i++) {
        if (condition) {
            ...
        }
    }
    }
}
```

```
function slow (x, y) {
    if (condition) {
        for (let i = 0; i < 8; i++) {
            ...
        }
    }
}
```

Unroll Loop and jam

loop문의 비교 연산의 수를 줄이거나 없앰

```
function slow (x, y) {
 for (i = 0; i < 8; i++)
        x += y
    }
    return x
}
```

```
function slow (x, y) {
for(i = 0; i < 8; i = i + 4)
{
      x += y
      x += y
      x += y
      x += y
   }
  return x
}
```

```
function slow (x, y) {
      x += y
      x += y
      x += y
      x += y
      x += y
      x += y
      x += y
      x += y
      return x
}
```

Unused Optimization

sroa (scalar replacement of aggregates transformation)

- no structure or array

tailcallelim (tail call elimination)

- no recursive call