SWPP Team 7

고승혁, 원영빈, 윤재석, 홍재희

Our Approach

- A. Reduce the **cost** of instructions
  - Use cost-efficient instructions

- B. Reduce the **length** of instructions
  - Use less instructions

Reducing the cost – 1. *Oracle*

- *Oracle*
  - Initial Cost: 40
  - Reduces cost of *load* or *store* by – 18 or 27

- More than **2 or 3 calls** of *load* or *store* -> **Use *Oracle***

Reducing the cost – 1. *Oracle*

w/o Oracle

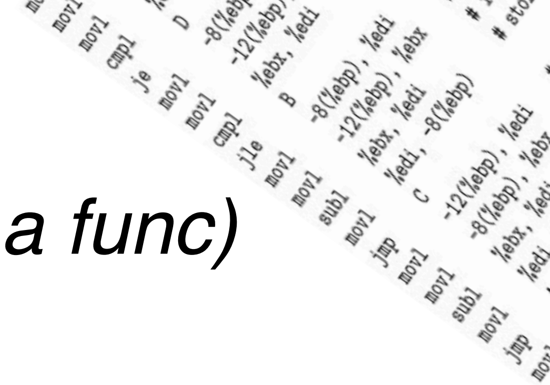
Cost: 60

```
store 8 1 r1 // 20
store 8 2 r2 // 20
store 8 3 r3 // 20
```

w/ Oracle

Cost: 40 + 6 = 46

```
start oracle r1, r2, r3: // 40
    store 8 1 r1 // 2
    store 8 2 r2 // 2
    store 8 3 r3 // 2
end oracle
```



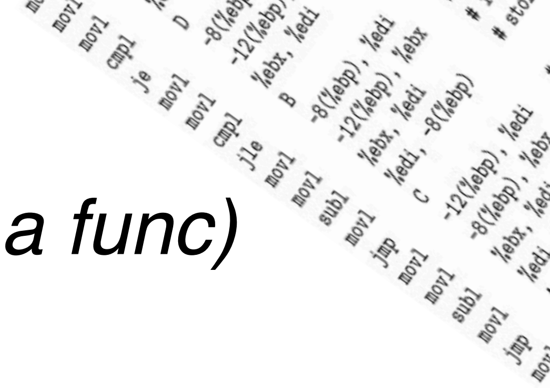
Reducing the cost – 2. *load* v. *aload* *(in a func)*

- *load*
  - - Stack: 20
  - - Heap: 30

- *aload*
  - - *aload* execution -> instructions with **cost m** -> *aload* resolved
  - - Stack: 1 + 24 (Resolve) = 25 (m <= 24)
  - - Heap: 1 + 34 (Resolve) = 35 (m <= 34)

- When m > 5, **use *aload*** instead of *load*

Reducing the cost – 2. *load* v. *aload* *(in a func)*



load

Cost: 20 + 10 + 5 = 35

```
start foo r1:
    r2 = load 8 r1 // 20
    r3 = sum 1 3 5 7 8 //
10
    r4 = add r2 r3 8 // 5
    ...
```

aload

Cost: 25 + 5 = 30

```
start foo r1:
    r2 = aload 8 r1 //
25-10
    r3 = sum 1 3 5 7 8 //
10
    r4 = add r2 r3 8 // 5
    ...
```

Reducing the cost – 3. *shift/logic* v. *mul/div*

- *shift/logic*
  - Cost: 4

- *mul/div*
  - Cost: 1

- If applicable, **favor *mul/div*** over *shift/logic*

Reducing the cost – 3. *shift/logic* v. *mul/div*

shift/logic

Cost: 4

```
r2 = shl r1 4 8
```

mul/div

Cost: 1

```
r2 = mul r1 16 8
```

Reducing the cost – 4. *add/sub* v. *mul/div*

- *add/sub*
  - Cost: 5

- *mul/div*
  - Cost: 1

- If applicable, **favor *mul/div*** over *add/sub*

- We can substitute **`a = a + a -> a = 2 * a`**

Reducing the cost – 4. *add/sub* v. *mul/div*

Original Code

```
if (a == b) {
    a = a + b
}
```

GVN + *mul/div* optimization

```
if (a == b) {
    a = a + a
// optimize -> a = 2 * a
}
```

Reducing the cost – 5. Conditional Branch

- Different cost for each leaf of conditional branch:

  - true_bb: 6

  - false_bb: 1 <- blocks most likely to be executed goes here

- Cost of conditional branch: 3.5 (or approx. 3 through false_bb optimization)

  - For **nested conditional branches**, *switch* might be more efficient

- If applicable, substitute to **ternary operation**

Reducing the cost – 5. Conditional Branch

Original Code

```
unsigned int foo;
if (foo > 2) {
    foo--;
} else {
    foo++;
}
```

false_bb optimization

```
// a1 = foo
br a1 <= 2 true_bb false_bb
true_bb:
    a1 = incr foo 8 // foo++
    br cont
false_bb:
    a1 = decr foo 8 // foo--
cont:
    ...
```

Reducing the length – 1. Dead Code Elimination

- Remove unused code
  - Find and prune unreachable BasicBlocks

Reducing the length – 2. Common Subexpression Elimination

- Reuse previously computed expressions
- Find and remove duplicate expressions

- Original Code

```
foo = a * b * x;
bar = a * b - y;
```

- CSE

```
tmp = a * b
foo = tmp * x;
bar = tmp - y;
```

Reducing the length – 3. Constant Propagation

- Substitute the values of known constants

- Original Code

```
int x = 420;
int y = x * 3;
int z = x + 42;
```

- Constant Propagation

```
int x = 420;
int y = 420 * 3;
int z = 420 + 42;
```

Reducing the length – 4. Function Inline

- Substitute function call -> function body
- Reduce overhead of function call & chance for additional optimization

Reducing the length – 5. Global Value Numbering

- Find and remove duplicate computaitons

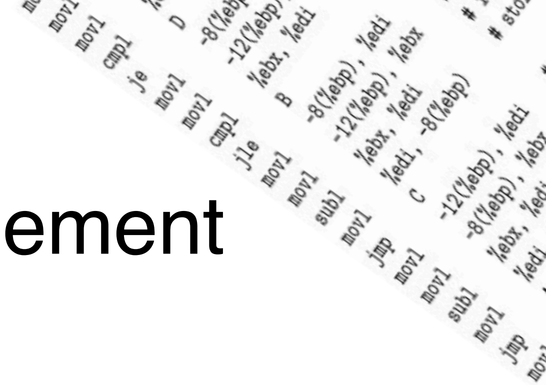
- Original Code

```
foo = 420;
bar = 420;

tmp1 = foo + 42;
tmp2 = bar + 42;
```

- GVN

```
foo = 420;
bar = foo;

tmp1 = foo + 42;
tmp2 = tmp1;
```

Reducing the length – 6. Heap Cost Management



- Similar to Garbage Collection (mark and sweep, etc.)

- Free allocated area in advance, if not referenced after certain point