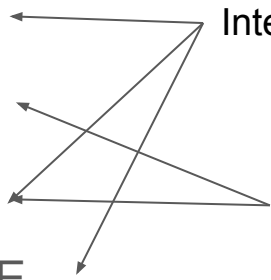


SWPP 2023 spring

Team members

- Choi Mingi, 19 CSE
 - Sungbin Jo, 21 CSE
 - Kijun Shin, 21 CSE
 - Wonseok Hur, 21 CSE
- 
- Interested in double majoring in math
- Interested in low-level compsci,
took the Compilers course
- The diagram consists of a central point from which four arrows originate. One arrow points to 'Choi Mingi, 19 CSE', another to 'Sungbin Jo, 21 CSE', a third to 'Kijun Shin, 21 CSE', and a fourth to 'Wonseok Hur, 21 CSE'. Additionally, there are two text labels on the right. The label 'Interested in double majoring in math' has an arrow pointing to 'Choi Mingi, 19 CSE'. The label 'Interested in low-level compsci, took the Compilers course' has an arrow pointing to 'Kijun Shin, 21 CSE'.

High level architecture cost analysis

- Idiotic costs for arith operations
 - e.g) bit operations are more expensive than multiplication?
- Super cheap function calls, with 'automatic' register management
 - e.g) effectively 32 registers for every function
 - throwing registers at everything seems like a viable strategy
 - inlining becomes detrimental if it increases register pressure
- Memory costs expected to dominate
- Global oracle function, unique to each program

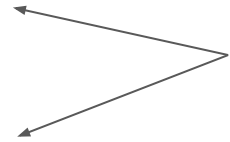
Reusing (basic) LLVM passes

- SimplifyCFG: canonicalize IR, run on every significant IR change
- SROA (scalar replacement of aggregates): smart register promotion
- CSE (common subexpression elimination)
- InstCombine: various peephole optimizations
 - probably needs to update according to our idiotic cost model
 - e.g. shifts are better expressed as multiplication

Trivial optimizations 1: branching costs

- Premise: Most branches have an expected result in the common case
- e.g) loops are (usually) expected to run multiple times
→ check how LLVM detects loops (& how the assembly emitter works)

```
entry:  
  br i1 %cond0, label %while.end, label %while.body  
while.body:  
  ;; computation  
  br i1 %cond1, label %while.end, label %while.body  
while.end:  
  ;; exit block
```



Expected to take truthy branch in the common case

Trivial optimizations 2: arithmetic optimization

- Premise: we have a nonsensical cost model for arithmetic operations

```
add x x      → mul x 2
shl x c      → mul x (1<<c)
ashr x c     → sdiv x (1<<c)
lshr x c     → udiv x (1<<c)
and x (1<<c-1) → urem x (1<<c)
```

load2aload: moving aload upwards

- Premise: asynchronously starting loads as soon as we can will increase compute/fetch overlap (at the cost of losing 1 reg)

while.body:

```
%idx = phi i64 [ %idx.next, %while.body ], [ 0, %entry ]
```

```
%ptr = getelementptr i64, i64* %arr, i64 %idx
```

```
;; computation (without stores)
```

```
%val = load i64, i64* %ptr
```

```
;; computation (using %val)
```

while.body:

```
%idx = phi i64 [ %idx.next, %while.body ], [ 0, %entry ]
```

```
%ptr = getelementptr i64, i64* %arr, i64 %idx
```

```
%val = aload i64, i64* %ptr
```

```
;; computation (without stores)
```

```
;; computation (using %val)
```

load2aload: moving aload upwards

- Alias analysis: single threaded memory model
 - can basically push until the last store we're not sure
 - Requires influence on register allocation
 - i.e., so that aloaded registers don't get spilled to the stack
 - Hopefully) Aload registers that get spilled to the stack
 - (check register pressure if this is a meaningful optimization)
- We'd like the ability to touch register allocation


```
preexecutor: precompute code paths
```

- Premise: our execution environment only have few side-effect operations
→ precompute deterministic code paths on compile time
- e.g.) function calls without pointer arguments → cannot touch memory

```
define i64 @somefunc(i8 %arg0, i1 %arg1, i1 %arg2) {  
  ;; computation (no read/write calls)  
  ret i64 %val  
}  
  
define i64 @switch(arg)
```

```
define i64 @somefunc(i8 %arg0, i1 %arg1, i1 %arg2) {
→ switch (arg0, arg1, arg2) {
    ;; 256*2*2=1024 cases
    }
}
```

| | | |
|--------------------|------------------------------------|---|
| Switch Instruction | switch <cond val> <val1> <bb1> ... | 4 |
|--------------------|------------------------------------|---|

| | | |
|--|---|---|
| | | |
| Switch Instruction - <val1>, ... should be constant integers. | switch <cond_val> <val1> <bb1> .. <div style="text-align: right;"><default_bb></div> | 4 |

(because switches are cheap
for some reason)

Other ideas: exploiting the sum operation

- Peephole optimization: convert multiple adds into a sum
- If we can do smart-enough loop analysis, hopefully we might be able to unroll 8 loop iterations? (I doubt it.)

```
res = 0;
for (i=0; i<25; ++i) {
    res += /*i-dependent computation*/
}
```

(hopefully)

| | | |
|-------------|---|----|
| Integer Sum | $\langle \text{reg} \rangle = \text{sum } \langle \text{val1} \rangle \dots \langle \text{val8} \rangle \langle \text{bw} \rangle$ $\langle \text{bw} \rangle := 1 8 16 32 64$ | 10 |
|-------------|---|----|

```
res = 0;
for (i=0; i<25;) {
    res0 = /*i-dependent computation*/
    res1 = /**/
    ...
    res7 = /**/
    res += asm![sum res0...res7]
}
```

Other ideas: usage of the oracle function

- Oracle function is unique to whole program
 - Optimizing for oracle requires whole program analysis
- Determining the usage of oracle dynamically is non-trivial
- Considering using oracle as a batch-storing operation

Other ideas: reduce memory cost

- Premise: stack memory use is significantly cheaper than the heap
 - detect memory that we can safely allocate in the stack
- e.g.) malloc-ed memory that gets freed in the same function
 - replace malloc with an alloca on the stack
- Different (hack) idea: using our own custom memory allocator? Maaaybe...

Gathering test cases

- Hoping that the TAs might provide test cases
- If not, write plain C-programs & compile them with clang -O0
- Adversarial test cases by hand
- General test cases from some real programs