Course Reminder

► You have covered all of the material required to build an optimizing compiler!

► Let’s wrap up a few more topics
  ► Other optimizations
  ► Security in compilers
GCC’s Optimizations

GCC Architecture
General order of optimizations

A. Scalar replacement of array references
   Data-cache optimizations

B. Procedure integration
   Tail-call optimization, including
tail-recursion elimination
   Scalar replacement of aggregates
   Sparse conditional constant propagation
   Interprocedural constant propagation
   Procedure specialization and cloning
   Sparse conditional constant propagation

C1. Global value numbering
    Local and global copy propagation
    Sparse conditional constant propagation
    Dead-code elimination

C2. Local and global common-subexpression elimination
    Loop-invariant code motion

C3. Partial-redundancy elimination

C4. Dead-code elimination
    Code hoisting
    Induction-variable strength reduction
    Linear-function test replacement
    Induction-variable removal
    Unnecessary bounds-checking elimination
    Control-flow optimizations

(from box D)
General order of optimizations (2)

- In-line expansion
- Leaf-routine optimization
- Shrink wrapping
- Machine idioms
- Tail merging
- Branch optimizations and conditional moves
- Dead-code elimination
- Software pipelining, with loop unrolling, variable expansion, register renaming, and hierarchical reduction
- Basic-block and branch scheduling 1
- Register allocation by graph coloring
- Basic-block and branch scheduling 2
- Intraprocedural I-cache optimization
- Instruction prefetching
- Data prefetching
- Branch prediction
- Interprocedural register allocation
- Aggregation of global references
- Interprocedural I-cache optimization

(to constant folding, algebraic simplifications, and reassociation)
Optimizations

▶ We have covered dataflow analysis

▶ Dataflow analysis serves as the basis for making decisions about program transformations that can preserve semantics while improving some objective function

▶ But what if we’re okay with wrecking the program a little?
int stripe (float p) {
    return floor(p.s + 0.5) - floor(p.s);
}
AST Manipulation

- By randomly changing the AST, we can occasionally find more optimal programs
  - Highly effective bug fixing technique
  - If you have an optimization function, you can use a **genetic search** to find more optimal ASTs
- How might we guide some random changes?
Notes on Genetic Search

- Random changes approximate first year students
  - “Wow, let’s change this and see what happens.”

- By leveraging **domain knowledge**, we can relax strict correctness requirements in favor of making gigantic optimizations
  - Analogy: MP3’s are way smaller than WAV’s—they optimize space by making the sound “wrong”
Examples

65% lower energy
Examples (2)
Hardware

Designed in Electric
(c) dilawar@ee.iitb.ac.in
Caching

- Caches contain fixed-size *lines* of adjacent addresses
  - Must replace complete lines at a time
  - Fetches/loads across line boundaries may be slower
    - Slightly higher chance of a cache miss with two lines
  - Recent x86 (e.g., Haswell) CPUs use 64-byte lines

- Knowing cache properties can influence optimization decisions by compiler
  - Ordering of loops, changing structure layout... (alignment)
CPU-level optimizations

Diagram of pipeline stages: Fetch, Decode, Execute, Memory, Write-Back.

- PC (Program Counter) to Fetch
- Memory to Execute
- Registers to Execute
- FP (Floating Point) and ALU (Arithmetic Logic Unit) in Execute
- Memory to Write-Back

Memory at the bottom.
CPU Pipeline

Clock cycle

0 1 2 3 4 5 6 7 8

Waiting instructions

Stage 1: Fetch
Stage 2: Decode
Stage 3: Execute
Stage 4: Write-back

Completed instructions

Compiler Construction
Bubbles in Pipeline

What if the green instruction is a `jmp`?
Delay slots

- Sometimes, we know instructions will require clearing the pipeline or insert delays.

- If the cost of such an instruction is fixed, maybe we can agree with the compiler to be more efficient.
  - We can emit instructions into delay slots after certain instructions.

```
1  r0 = 0 ;
2  call func ;
3  r1 = 0 ;  -- 2x delay slots
4  r2 = 0 ;  -- always executes before func!
```
Out-of-order Execution

1  ld  rbx, [rsp+4]
2  ld  rcx, [rsp+4]
3  add rbx, 1
4  add rcx, 2

What will happen in the pipeline here?

- Considerations for the compiler:
  - Track with clever register allocation
  - Peephole optimizations (rename registers)
  - Move instructions around!
Optimizations in Hardware

- Out-of-order execution
  - Load up lots of instructions that need executing
  - Execute them in any order, so long as dependencies remain in order

Add writeback logic
Add reservation station
Register renaming

- Sometimes, we don’t care about specific register names until written back

1. `mov rax, [rsp+4]`
2. `add rax, 2`
3. `mov [rsp+8], rax`
4. `mov rax, [rsp+12]  -- rax here??`
5. `add rax, 4`
6. `mov [rsp+16], rax`
Optimizations in Hardware (3)

- Micro-op (\(\mu\)-op) instruction sequencing
  - What are common characteristics of CISC machines?

Variable length, variable delay, extremely complex

Historical artifacts... modern platforms prefer RISC-like

However, we need to support legacy code

Modern CISC compiles to RISC on the fly!

CPU caches contain RISC instructions

\(\mu\)-op sequencer holds RISC-like instructions

Execution cores work on simpler, fixed length RISC instructions that implement CISC behavior
Optimizations in Hardware (3)

➤ Micro-op (μ-op) instruction sequencing
  ➤ What are common characteristics of CISC machines?

➤ Variable length, variable delay, extremely complex
  ➤ Historical artifacts... modern platforms prefer RISC-like
  ➤ However, we need to support legacy code
Optimizations in Hardware (3)

▶ Micro-op (µ-op) instruction sequencing
  ▶ What are common characteristics of CISC machines?

▶ Variable length, variable delay, extremely complex
  ▶ Historical artifacts... modern platforms prefer RISC-like
  ▶ However, we need to support legacy code

▶ Modern CISC compiles to RISC on the fly!
  ▶ CPU caches contain RISC instructions
  ▶ a µ-op sequencer holds RISC-like instructions
  ▶ Execution cores work on simpler, fixed length RISC instructions that implement CISC behavior
ROP Attacks

- Return Oriented Programming

- Idea How do we hijack control without injecting code?

- Later How does this relate to compilers?1

Stack Frame and Stack Overflows

Diagram showing the layout of a stack frame with:
- **high address**
- **RBP + 24**: h
- **RBP + 16**: g
- **RBP + 8**: return address
- **RBP**: saved RBP
- **RBP - 8**: xx
- **RBP - 16**: yy
- **RBP - 24**: zz

And a snippet showing:
- **RDI**: a
- **RSI**: b
- **RDX**: c
- **RCX**: d
- **R8**: e
- **R9**: f

Additionally, there's a note indicating a "red zone" of 128 bytes.
ROP attacks

- Classic Stack overflow attack includes code to be executed that smashes the stack!
  - NX pages, stack canaries, and address randomization fix these easily!

- But what if we know other instructions in memory?
  - e.g., if we know the address of `printf` in memory, maybe we can use some instructions in that function to achieve malicious behavior?
Gadgets

- ROP attack collects addresses of gadgets
- Gadgets are benign individually
  - ROP attack orders them in a way that causes malicious behavior to emerge
  - Gadgets are short sequences of instructions in memory that terminate with return instructions
- a ROP attack smashes the stack with sequence of addresses of gadgets
  - critical: No code injection! CPU executes code that’s already there

- What does this have to do with compilers?
- How might we fix these?