CMPT 295
Unit – Code Optimization
Lecture 32 – Examples of Code Optimization
Last lectures

- Limitations on pipelined execution
  - Limitation 3 - Data Hazard
    - Data dependencies
      - Solution 1 – Stall (bubbles)
      - Solution 2 – Data forwarding
    - Load/Use Data Hazard
      - Solution: Load interlock
  - Limitation 4 – Structural Hazard
    - Solution: Stall the pipeline for the fourth instruction
  - Limitation 5 – Control Hazard
    - Branch misprediction
      - Solution: instruction smashing
Today’s Menu

- Code optimization
  - Examples of code optimization
    - Done by compilers
    - Done by software developers
  - More optimization strategies:
    - Loop unrolling
    - Enhancing parallelism
Goal for this unit

- So far, we have ...
  - Learnt what an instruction set architecture (ISA) is
  - Created a few simple ISA’s
  - Learnt quite a lot about the x86-64 ISA
  - Some understanding of how a simple processor executes such instructions
- Let’s now put all this together and investigate how this knowledge can help us make our programs *run faster* (*time efficient*) by exploring different program optimization strategies
Capabilities of optimizing compilers

- Employ sophisticated forms of analysis and optimization
  - -Og, -O1, -O2
- Examples:
  - Code Motion
    - Removing unnecessary function call
  - Reduction in Strength
Examples of Reduction in Strength optimization

1. \( y = \frac{x}{2.0} \Rightarrow y = x \times 0.5 \)
   - Division more expensive than multiplication
   - If compiler can tell that denominator is a constant, it may use multiplication instead

2. \( x = \text{pow}(y, 2.0) \Rightarrow x = y \times y \)
   - Raising one value to the power of another is more expensive than multiplying
   - If the compiler can tell that the power is a small integer, it may use multiplication instead
Limitations of optimizing compilers

- Ideally, compilers able to take whatever code we write and generate the most efficient possible machine-level program
  - Thwarted by *optimization blockers*:
    - Function calls
      - Compiler may not be able to determine whether a function is free of side effect – compiler may leave functions calls intact
    - Memory aliasing
      - Compiler may not be able to determine whether or not 2 pointers are aliased - pointing to same memory location - compiler may not optimize the use of the pointers
  - Why?
    - Compiler must produce a machine code version of our program with same behavior for all possible test cases
  - Solution: *Help* compiler by rewriting our code
Demo 1 – Turning ON compiler optimization

```c
int proc(long a1, int a2, int a3, long a4, char a5, char a6, char a7, long *alp)
{
    return a1+a2+a3+a4+a5+a6+a7; // 28
}

long call_proc()
{
    long x1 = 1;
    int x2 = 2;
    int x3 = 3;
    long x4 = 4;
    char x5 = 5;
    char x6 = 6;
    char x7 = 7;

    long result = proc(x1, x2, x3, x4, x5, x6, x7, &x1);
    return result;
}

void main()
{
    call_proc();

    return;
}
```
No Optimization

```assembly
.file "main.c"
.text
.globl proc
.type proc, @function
proc:
.LFB0:
    .size proc, .-proc
.globl call_proc
.type call_proc, @function
.call proc:
.LFB1:
    .size call_proc, .-call_proc
.globl main
.type main, @function
main:
.LFB2:
    .size main, .-main
.ident "GCC: (Ubuntu 7.5.0-3ubuntu1-18.04) 7.5.0"
.section .note.GNU-stack, "", @progbits
```
-Og -> Basic set of optimizations

Produces assembly code that mirrors the C code

```c
.globl proc
.type proc, @function
proc:
.LRN0:
  .cfi_startproc
  addl %edi, %esi
  addl %esi, %edx
  addl %edx, %ecx
  movsbl $r8b, %rdl
  addl $r8d, %ecx
  movsbl $r8b, %rdl
  addl $r9d, %ecx
  movsbl $(r8p), %eax
  addl %ecx, %eax
  ret
  .cfi_endproc
.LRN0:
.size proc, -.proc
.globl call_proc
.type call_proc, @function
.call_proc:
.LFB1:
  .cfi_startproc
  subq $24, %rsp
  .cfi_def_cfa_offset 32
  movq %fs:40, %rax
  movq %rax, @($r8p)
  xorl %eax, %eax
  movq %rsp, %rax
  pushq %rax
  .cfi_def_cfa_offset 40
  pushq $7
  .cfi_def_cfa_offset 48
  movl $6, %rdl
  movl $5, %r8d
  movl $4, %ecx
  movl $3, %edx
  movl $2, %esi
  movl $1, %edi
  call proc
  addq $16, %rsp
  .cfi_def_cfa_offset 32
  movq 8(%r8p), %rdx
  xorq %fs:40, %rdx
  jne .L5
  cltq
  addq $24, %rsp
  .cfi_def_cfa_offset 8
  ret
.L5:
  .cfi_restore_state
  call __stack_chk_fail@PLT
  .cfi_endproc
.LFB2:
.size call_proc, -.call_proc
.globl main
.type main, @function
main:
.LFB2:
  .cfi_startproc
...
-O1-> Removes unnecessary function call

Compiler has removed the call to `proc(...)` and replaced it with the return value of `proc(...)`, i.e., 28!
Compiler-Generated Code Motion (-O1)

void setRow(char *A, char *B, int i, int N) {
    int j;
    for (j = 0; j < N; j++)
        A[N*i+j] = B[j];
}

int j;
int Ni = N*i;
long size = B+N;
for (j = 0; B < size; B++) {
    *(A+Ni) = *B;
    Ni++;
}

setRow:
    testl %ecx, %ecx  # Test N
    jle done         # If 0, done
    imull %ecx, %edx  # Ni = N*i
    movq %rsi, %rax
    leal -1(%rcx), %ecx  # N-1
    leaq 1(%rsi,%rcx), %r8  # size = B+N
loop:
    movzbl (%rax), %esi  # t = M[B]
    movslq %edx, %rcx
    movb %sil, (%rdi, %rcx)  # M[A+Ni] = t
    addl $1, %edx  # Ni++
    addl $1, %rax  # B++
    cmpq %r8, %rax  # if B != B+N ...
    jne loop        # ... loop

done:
    ret
“Programmer-generated” optimization

- We must assist compiler by writing code that can be optimized readily

- Strategies:
  - Eliminating unnecessary work: function calls (with unchanged results - in loop), memory references, conditional statements (Lab 6), factoring
  - Replacing function calls using inline substitution
  - Exploiting capability of processors to provide instruction-level parallelism
    - Loop unrolling: reduce critical path length and reduce loop overhead by reducing number of loops
  - Cache-friendly code

- Issues
  - Transformation may break modularity, make program more difficult to read (hence understand)
Demo – str_alnum – 1 & 2

// Around 3409 clock cycles
int str_alnum1(char *s) {
    int i;
    for (i = 0; i < strlen(s); i++) {
        if (!isalnum(s[i])) {
            return 0;
        }
    }
    return 1;
}

// Around 1058 clock cycles
int str_alnum2(char *s) {
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++) {
        if (!isalnum(s[i])) {
            return 0;
        }
    }
    return 1;
}

Inefficiency in str_alnum1(...): calling strlen(s) at every iteration of the for loop

Solution: Because strlen(s) produces the same result at every iteration of the for loop, we can move strlen(s) outside (before) the for loop and assigned it constant result to a local variable, which is used inside the loop

Effect of this solution: Since strlen(s) has a time efficiency of O(n), it made str_alnum1(...) operate in O(n^2). Now that strlen(s) is outside the loop, str_alnum1(...) operates in O(n).

Possible inefficiency in str_alnum2(...): isalnum(s[i]) called inside the loop
Solution: We could “inline” `isalnum(s[i])` and save on the function call overhead.

Since `isalnum(...)` is a C function, we “inline” this function with our own implementation of it.

Why is this solution not working?

Reason:
1. When “inlining” the function `isalnum(...)`, we have introduced 6 CMP instructions (branches - in assembly code - which can be taken or not). Each of them can potentially produce misprediction, causing instructions smashing, which increases execution time.
2. We also have introduced 6 memory accesses.
Solution: We access memory once and assign its result to a local variable

But: We are still using 6 CMP instructions

```c
// Around 2917 lock cycles
int str_alnum4(char *s) {
    int i;
    int len = strlen(s);
    char aChar;
    for (i = 0; i < len; i++) {
        aChar = s[i];
        if (!( ((aChar >= 'a') && (aChar <= 'z')) ||
               ((aChar >= 'A') && (aChar <= 'Z')) ||
               ((aChar >= '0') && (aChar <= '9')) )) {
            return 0;
        }
    }
    return 1;
}
```
Solution: We use a hash “look up” table where each character of the string s is used as an index into this table. Also, the whole look up table may be cached in the fastest section of the memory -> Chapter 6
Summary

- Goal: Make programs time efficient by exploring different program optimization strategies
  - Optimization strategies: code motion, inline substitution, memory reference reduction, branching minimization, factoring, loop unrolling, cache-friendly code ...
- Compiler-generated optimization
  - Optimization blockers: function calls and memory accesses
- Programmer-generated optimization
  - Transform our program to help compiler optimize
- Issues
  - Code length, readability, modularity ...
Next lecture

- Code optimization
  - Examples of code optimization
    - Done by compilers
    - Done by software developers
  - More optimization strategies:
    - Enhancing parallelism
    - Loop unrolling