Program Optimization

Readings: Chapter 5
Coming Up: Assignment 5

Performance Realities

- There’s more to performance than asymptotic complexity
- Constant factors matter too!
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs are compiled and executed
  - How modern processors + memory systems operate
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Generally Useful Optimizations (Machine-Independent)

- Optimizations that you or the compiler should do regardless of processor / compiler
- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b, long l, long n) {
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

Compiler-Generated Code Motion (-O1)

```
void set_row(double *a, double *b, long l, long n) {
    long j;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
        *rowp++ = b[j];
}
```

```
set_row:
    testq %rcx, %rcx
    jle .L1  # Test n
    imulq %rcx, %rdx
    leaq (%rdi,%rdx,8), %rdx
    movl $0, %eax
    .L3:
        movsd (%rsi,%rax,8), %xmm0
        movsd %xmm0, (%rdx,%rax,8)
        # M[Arn*i+B+8] = t
        addq $1, %rax
        cmpq %rcx, %rax
        # j<n
        jne .L3  # if !=, goto loop
        jmp .L1:  # done.
```

```
void set_row(double *a, double *b, long l, long n) {
    long j;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
        *rowp++ = b[j];
}
```
Share Common Subexpressions (Machine-Independent)

- Reuse portions of expressions
- GCC will do this with -O1

/* Sum neighbors of i, j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;

// Reducing in strength
- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  16*x --> x << 4
  • Utility machine dependent
  • Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

leaq 1(%rsi), %rax  # i+1
leaq -1(%rsi), %r8  # i-1
imulq %rcx, %r8  # i*n
imulq %rcx, %rax  # (i+1)*n
addq %rdx, %r8  # i*n+j
addq %rdx, %rax  # (i+1)*n+j
imulq %rcx, %rsi  # i*n
addq %rdx, %rsi  # i*n+j
movq %rsi, %rax  # i*n+j
subq %rcx, %rax  # i*n+j-n
leaq (%rsi,%rcx), %rcx # i*n+j+n

Make Use of Registers

- Reading and writing registers much faster than reading/writing memory
- Limitation
  - Compiler not always able to determine whether variable can be held in register
  - Possibility of Aliasing
  - See example later

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Don’t (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Have difficulty overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects
Limitations of Optimizing Compilers

- Operate under fundamental constraint
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
    - Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - E.g., Data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural analysis within individual files
  - But, not between code in different files
- Most analysis is based only on static information
  - Compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```c
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance
  - N calls to strlen, each O(N) ➞ O(N^2) performance

```
<table>
<thead>
<tr>
<th>String length</th>
<th>CPU seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>50000</td>
<td>0.1</td>
</tr>
<tr>
<td>100000</td>
<td>0.2</td>
</tr>
<tr>
<td>150000</td>
<td>0.4</td>
</tr>
<tr>
<td>200000</td>
<td>0.8</td>
</tr>
<tr>
<td>250000</td>
<td>1.6</td>
</tr>
<tr>
<td>300000</td>
<td>3.2</td>
</tr>
<tr>
<td>350000</td>
<td>6.4</td>
</tr>
<tr>
<td>400000</td>
<td>12.8</td>
</tr>
<tr>
<td>450000</td>
<td>25.6</td>
</tr>
<tr>
<td>500000</td>
<td>51.2</td>
</tr>
</tbody>
</table>
```

Improving Performance

- Move call to strlen outside of loop
- Since result does not change from one iteration to another
- Form of code motion
Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2

![Graph showing CPU seconds vs string length for lower1 and lower2](image)

Optimization Blocker: Procedure Calls

- Why couldn't compiler move strlen out of inner loop?
  - Procedure may have side effects
  - Function may not return same value for given arguments
  - Depends on other parts of global state
  - Procedure lower could interact with strlen

- Warning:
  - Compiler treats procedure call as a black box
  - Weak optimizations near them

- Remedies:
  - Use of inline functions
  - GCC does this with -O1
  - Within single file
  - Do your own code motion

```c
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0')
    {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

Memory Matters

- Code updates b[i] on every iteration
- Why couldn't compiler optimize this away?

```c
void sum_rows1(double *a, double *b, long n)
{
    long i, j;
    for (i = 0; i < n; i++)
    {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Memory Aliasing

- Code updates b[i] on every iteration
- Must consider possibility that these updates will affect program behavior

```c
double A[9] = { 0,   1,   2,
                4,   8,  16,
                32,  64, 128};


sum_rows1(A, B, 3);
```

Value of B:

```
init:  [4, 8, 16]
i = 0:  [3, 8, 16]
i = 1:  [3, 22, 16]
i = 2:  [3, 22, 224]
```
Removing Aliasing

```c
/* Sum rows is of n X n matrix a
 and store in vector b */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

- No need to store intermediate results

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

Advanced Compiler Optimizations: Loop Unrolling

```assembly
irmovl $5, %edx
irmovl $80, %ebx
Loop:
    mrmovl array_base(%ebx), %eax
    addl %edx, %eax
    rmmovl %eax, array_base(%ebx)
    rmovl %ebx, %esi
    addl $-4, %esi
    mrmovl array_base(%esi), %eax
    addl %edx, %eax
    rmmovl %eax, array_base(%esi)
    addl $-8, %ebx
    jne Loop
...```

Comparing the two code sequences

```assembly
irmovl $5, %edx
irmovl $80, %ebx
Loop:
    mrmovl array_base(%ebx), %eax
    addl %edx, %eax
    rmmovl %eax, array_base(%ebx)
    rmovl %ebx, %esi
    addl $-4, %esi
    mrmovl array_base(%esi), %eax
    addl %edx, %eax
    mrmovl %eax, array_base(%esi)
    addl $-8, %ebx
    jne Loop
...```
**Superscalar Processor**

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- **Benefit:** without programming effort, superscalar processor can take advantage of the *instruction-level parallelism* that most programs have.

- Most modern CPUs are superscalar
- Intel: since Pentium (1993)

**Types of Data Hazards**

- RAW – true dependence
- WAR – anti-dependence
- WAW – output dependence

**Instruction-level Parallelism**

- Pipelining/super-pipelining
- Out-of-order execution
- Super-scalar
  - multiple instructions per pipeline stage
  - Dependencies handled in hardware
- Very Large Instruction Word (VLIW)
  - Multiple instructions per pipeline stage
  - Dependencies taken care of by compiler
Enabling Out-Of-Order Execution

- Tomasulo’s generalized data forwarding algorithm in the IBM360
  - Register renaming
  - Reservation stations
  - Common data bus to broadcast data to reservation stations

Instruction Control

- Grabs Instruction Bytes From Memory
  - Based on Current PC + Predicted Targets for Predicted Branches
  - Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target
- Translates Instructions into Operations
  - Primitive steps required to perform instruction
  - Typical instruction requires 1–3 operations
- Converts Register References into Tags
  - Abstract identifier linking destination of one operation with sources of later operations

Execution Unit

- Multiple functional units
  - Each can operate independently
  - Operations performed as soon as operands available
    - Not necessarily in program order
    - Within limits of functional units
  - Control logic
    - Ensures behavior equivalent to sequential program execution

Advanced Processor Design Techniques

- Trace caches
- Register renaming – eliminate WAW, WAR hazards
  - Register map table
  - Free list
  - Active list
- Speculative execution
- Value prediction
- Branch prediction
  - Branch history tables for prediction
  - Branch stack to save state prior to branch
  - Branch mask to determine instructions that must be squashed
Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic