Lecture #15
Chap. 5: Program Performance Optimization

Prof. Soraya Abad-Mota, PhD

Chapter 5: Optimizing Program Performance

1. Realities and compiler optimizations.
2. Optimization blockers + practice
3. Measures of performance: CPE (cycles per element) and a benchmark
4. Exploiting instruction parallelism: effect of all basic optimizations; the bounds: latency and throughput.
5. Loop unrolling, reassociation, separate accumulators.
6. Branches

Computer Programs?

Working correctly under all possible conditions

or

Running fast

How to write an efficient program

1. Select appropriate data structures and algorithms for the tasks at hand. Programs must be easily understood to allow a “smooth” maintenance.

How to write an efficient program (2)

2. Write source code that the compiler can effectively optimize to transform into "efficient executable code" ➔ learn capabilities and limitations of optimizing compilers. *C* has some challenges.

How to write an efficient program (3)

3. “Divide a task into portions that can be computed in parallel, on some combination of multiple cores and multiple processors.”

Performance Realities (1/2)

- *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 (specific) levels:
    - algorithm, data representations, procedures, and loops

Performance Realities (2/2)

- 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.

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

Note: we have seen some examples, i.e. using shifts and additions instead of multiplication, using leaq for some arithmetic computations.

We will see more examples in this chapter.

Optimizing Compilers (2)

- 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

Note: we will see how the programmer can avoid these.

Can we rely solely on the optimizer?

No,

in addition to the characteristics described,
they have other limitations.

Limitations of Optimizing Compilers

1. Operate under fundamental constraints
   - 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.

Limitations of Optimizing Compilers (2)

2. 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

3. Most analysis is based only on *static* information
   - Compiler has difficulty anticipating run-time inputs

Limitations of Optimizing Compilers (3)

4. Most analysis is performed only within procedures
   - Whole-program analysis is too expensive in most cases
   - Newer versions of GCC do inter-procedural analysis within individual files
     - But, not between code in different files (.c source code)

5. When in doubt, the compiler must be conservative

What we can do:

- Common Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- Optimization Blockers (avoid them to help the optimizer)
  - Procedure calls
  - Memory aliasing

Summary

When Optimizing program performance, be aware of:

- Performance realities: constant factors matter, must understand the system to optimize performance.
- Limitations of the optimizer (inside the compiler)
  - helps but has limitations, e.g. cannot overcome optimization blockers, cannot change program behavior;
  - most analysis is of procedures and optimizations occur on loops;
  - when in doubt it must be conservative.

Generally Useful Optimizations

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 a loop

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

→

n and i are not changing
do n*i only once

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
    a[ni+j] = b[j];
```

Compiler-Generated Code Motion (-O1)

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

```
    long j;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
    *rowp++ = b[j];
```

```
set_row:
    testq    %rcx, %rcx        # Test n
    jle  .L1           # If 0, goto done
    imulq    %rcx, %rdx        # ni = n*i
    leaq (%rdi,%rdx,8), %rdx   # rowp = A + ni*8
    movl $0, %eax                  # j = 0
.L3:                       # loop:
    movsd    (%rsi,%rax,8), %xmm0      # t = b[j]
    movsd    %xmm0, (%rdx,%rax,8)      # M[A+ni*8 + j*8] = t
    addq $1, %rax          # j++
    cmpq %rcx, %rax        # j:n
    jne  .L3           # if !=, goto loop
.L1:                       # done:
    rep ; ret
```

Reduction 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

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Share Common Subexpressions

- 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;
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

1 multiplication: i*n

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

Optimization Blocker #1: Procedure Calls
(section 5.1)

- Procedure to Convert String to Lower Case

```
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');
}
```

  - Extracted from 213 lab submissions, Fall, 1998

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance

Convert Loop To Goto Form

```
void lower1(char *s)
{
   size_t i = 0;
   if (i >= strlen(s))
     goto done;
 loop:
   if (s[i] >= 'A' && s[i] <= 'Z')
       s[i] -= ('A' - 'a');
   i++;
   if (i < strlen(s))
     goto loop;
 done:
}
```

- `strlen` executed every iteration

Calling Strlen

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
   s++;
   length++;
    }
    return length;
}
```

- Strlen performance
  - Only way to determine length of string is to scan its entire length, looking for null character.
- Overall performance, string of length N
  - N calls to strlen
  - Require times N, N-1, N-2, …, 1
  - Overall $O(N^2)$ performance

Improving Performance

```
void lower2(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- 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

Optimization Blocker: Procedure Calls

- *Why couldn't compiler move* `strlen` *out of inner loop?*
  - Procedure may have side effects
    - Alters global state each time called
  - 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

```
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

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
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];
    }
}
```

```
# sum_rows1 inner loop
.L4:
        movsd   (%rsi,%rax,8), %xmm0   # FP load
        addsd   (%rdi), %xmm0      # FP add
        movsd   %xmm0, (%rsi,%rax,8)   # FP store
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

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

Summarize the approach to performance optimization

- Compiler optimizes but it is conservative
- Programmers should avoid optimization blockers:
  - #1: procedure calls (the optimizer will not optimize them)
  - #2: memory aliasing
- Program performance metric: cycles per element (CPE)
- Benchmark: using a program example will show the effect on performance of some simple transformations that lead to great savings
- Understand modern processors: the bounds
- Loop unrolling (5.8) and parallelism (5.9)

Optimization Blocker: Procedure Calls

- *Why couldn't compiler move* `strlen` *out of inner loop?*
  - Procedure may have side effects
    - Alters global state each time called
  - 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 (p. 501)
    - GCC does this with –O1
      - Within single file
  - Do your own code motion

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

Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
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];
    }
}
```

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

double B[3] = A+3;

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]
```

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

Removing Aliasing

```
/* 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;
    }
}
```

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0 # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

- No need to store intermediate results (in b[i])

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references (pointers) may designate a single location (the same location)
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - May lead to different results
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

Practice problems (do them on your own)

- Problem 5.1 p. 500 (implications of memory aliasing)
- Problem 5.3 p. 512 (procedure calls)

Memory aliasing example (Prob. 5.1)

```
void swap (long *xp, long *yp) {
    *xp = *xp + *yp;   /* x + y */
    *yp = *xp - *yp;   /* x + y – y = x */
    *xp = *xp - *yp;   /* x + y – x = y */
}
```

What if swap is called with xp equal to yp?

Do problem 5.3, p. 512

- for (init; condition; increment)

  t += function4 call;

init, condition, and increment call 3 different functions.

Three different for loops with calls to 4 functions in those four places.

The whole point of the problem is how many times each function is called. Depending on where it is called from. If there is not a different value on each call, it should not be called more than one time, so code motion is recommended.

Working in performance optimization

involves:

- having measures to quantify the performance.
  - cycles per element
- using benchmarks to verify which optimizations work best and produce substantial improvements in performance

Cycles Per Element (CPE) (section 5.2)

- Convenient way to express performance of program that operates on vectors or lists. psnum1 (1 elem per iteration) psnum2 (2 elem per it.) pp.502-504
- Length = n
- In our case: CPE = cycles per OP
- T = CPE*n + Overhead
  - CPE is slope of line

Applicable to procedures containing loop that iterates over set of elem.

psnum2 1st e.g. loop unrolling

A benchmark

Description of the benchmark used in the book

- Uses a data structure that involves a struct which points to a vector (array).
- First: how is the data represented in the benchmark?

The image part with relationship ID rId13 was not found in the file.

Benchmark Example: Data Type for Vectors

(section 5.3) Program combine, series of optimizations

```
/* data structure for vectors */
typedef struct{
    size_t len;
    data_t *data;
} vec;
```

- Data Types
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`

```
/* retrieve vector element
   and store at val */
int get_vec_element
  (*vec v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

The image part with relationship ID rId13 was not found in the file.

A benchmark (2)

- The combine1 function operates on all the elements of the vector to compute either the summation or the product of all the elements.
- Notice that this function has a for loop in which a function is called in the condition of the loop (vec_length) and in the body of the loop (get_vec_elem).

(This is why it is relevant to do problem 5.3)

The image part with relationship ID rId13 was not found in the file.

Benchmark Computation

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

- Data Types
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`

The image part with relationship ID rId13 was not found in the file.

- Operations
  - Use different definitions of `OP` and `IDENT`
  - `+ / 0`
  - `* / 1`

Benchmark Performance

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

Basic Optimizations (series of changes)

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- Move vec_length out of loop (code motion to get combine2)
- Avoid bounds check on each cycle (combine 3, sec 5.5)
- Accumulate in temporary (combine 4)

Effect of Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

- Eliminates sources of overhead in loop

Exploiting Instruction-Level Parallelism

- To move beyond basic optimizations
- 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 (for example)

Modern CPU Design (Section 5.7)

Fig. 5.11
p. 519

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)

The image part with relationship ID rId13 was not found in the file.

Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```

	Time						
	1	2	3	4	5	6	7
Stage 1	`a*b`	`a*c`			`p1*p2`		
Stage 2		`a*b`	`a*c`			`p1*p2`	
Stage 3			`a*b`	`a*c`			`p1*p2`

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

Operation characterization

- *latency:* total time required to perform the operation
- *issue time:* minimum number of clock cycles between two independent operations of same type (due to pipelining)
- *capacity*: number of functional units (FU) that can perform the operation

1. FU with issue time = 1 are *fully pipelined*, can start a new operation every clock cycle
2. capacity > 1 -> more than 1 FU can perform op.

The image part with relationship ID rId13 was not found in the file.

Haswell CPU (FU performance sec. 5.7.2)

8 Total Functional Units

- Multiple instructions can execute in parallel

  2 load, with address computation

  1 store, with address computation

  4 integer

  2 FP multiply

  1 FP add

  1 FP divide

- Some instructions take > 1 cycle, but can be pipelined

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

Example: Floating-point adder

- Typically has three stages and therefore latency = 3
  - one to process the exponent values
  - one to add the fractions, and
  - one to round the results

Divider (a special case)

- Used for integer and FP division, and FP square root
- is not pipelined ➔ issue time = latency
- must perform a complete division before beginning a new one.
- Divisions measures are provided in ranges:
  - some combinations of dividend and divisor require more steps than others
- Division is a costly operation

Maximum Throughput of an FU

Another way of expressing the *issue time.*

For an operation with capacity *C* and issue time *I*

the processor can achieve a throughput of

C/I operations per clock cycle

e.g.

the reference machine can perform

2 floating point multiplications per cycle (2/1)

Summary of Performance Optimization until now

- Common Optimizations
  - Code motion/precomputation
  - Strength reduction (substitute * with shifts)
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- Optimization Blockers (avoid them to help the optimizer)
  - Procedure calls
  - Memory aliasing