15-213

Code Optimization
November 3, 1998

Topics

- **Basic optimizations**
  - Reduction in strength
  - Code motion
  - Common subexpression sharing
- **Optimization blockers**
- **Advanced optimizations**
  - Code scheduling
  - Unrolling & pipelining
- **Advice**

Great Reality #4

There's more to performance than asymptotic complexity

Constant factors matter too!

- **Easily see 10:1 performance range depending on how code written**
- **Must optimize at multiple levels: algorithm, data representations, procedures, and loops**

Must understand system to optimize performance

- **How programs compiled and executed**
- **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**
- **Eliminating minor inefficiencies**

Don't (Usually) Improve Asymptotic Efficiency

- **Up to programmer to select best overall algorithm**
- **Big-O savings more important than constant factors**
  - But constant factors count, too.

Have Difficulty Overcoming "Optimization Blockers"

- **Potential memory aliasing**
- **Potential procedure side-effects**

Limitations of Optimizing Compilers

Work under Tight Restriction

- **Cannot perform optimization if changes behavior under any realizable circumstance**
- **Even if circumstances seem quite bizarre**

Have No Understanding of Application

- **Limited information about data ranges**
- **Don't always make best trade-offs**

Some Don't Try Very Hard

- **Increase cost of compilation**
- **More chances for compiler errors**

Basic Optimizations

Reduction in Strength

- **Replace costly operation with simpler one**
- **Shift, add instead of multiply or divide**
  - Integer multiplication requires 8-16 cycles on the Alpha 21164
- **Procedure with no stack frame**
- **Keep data in registers rather than memory**
- **Pointer arithmetic**

Code Motion

- **Reduce frequency with which computation performed**
  - If it will always produce same result
  - Especially moving code out of loop

Share Common Subexpressions

- **Reuse portions of expressions**

Optimizing Multiply / Divide

Optimize Handling of Constant Factors

- **Exploit properties of binary number representation**
- **Several shifts & adds cheaper than multiply**

Multiplication

- $x * (2^{w_1} + 2^{w_2} + \ldots + 2^{w_k}) = (x << w_1) + (x << w_2) + \ldots + (x << w_k)$
- **Both signed and unsigned**
- **Special trick for groups of 1's**

  $(2^{w+k-1} + 2^{w+k-2} + \ldots + 2^{w}) = 2^{w+k} - 2^{w}$

Division

- $x / 2^w = x >> w$
- $x \% 2^w = x \& (2^w - 1)$
- **Special considerations if *x* can be negative**
  - Arithmetic rather than logical shift
  - Want remainder to have same sign as dividend

Multiply / Divide Example #1

Unsigned integers, power of 2

- **Most possible optimizations**

```
void
uweight4(unsigned long x,
         unsigned long *dest)
{
  dest[0] = 4*x;
  dest[1] = 4*4*x;
  dest[2] = -4 * x;
  dest[3] = x / 4;
  dest[4] = x % 4;
}
```

Code Sequences

```
s4addq  $16,0,$1     # $1 = 4x
stq     $1,0($17)    # dest[0] = 4x

sll     $16,4,$1     # $1 = 16x
stq     $1,8($17)    # dest[1] = 16x

lda     $1,-4        # $1 = -4
mulq    $16,$1,$1    # $1 = -4x
stq     $1,16($17)   # dest[2] = -4x

srl     $16,2,$1     # $1 = x / 4
stq     $1,24($17)   # dest[3] = x / 4

and     $16,3,$16    # $16 = x % 4
stq     $16,32($17)  # dest[4] = x % 4
```


Multiply / Divide Example #2

Signed integers, power of 2

- **Multiplication same as for unsigned**
- **Correct rounding of negatives for division**
- **Shift / And combination would produce positive remainder**

```
void
weight4(long int x,
        long int *dest)
{
  • • •
  dest[3] = x / 4;
  dest[4] = x % 4;
}
```

Division Code

```
addq    $16,3,$1     # $1 = x + 3
cmovge  $16,$16,$1   # if (x >= 0), $1 = x
sra     $1,2,$1      # $1 = x / 4
stq     $1,24($17)   # dest[3] = x / 4

s4addq  $1,0,$1      # $1 = 4 * (x / 4)
subq    $16,$1,$16   # $16 = x - (4 * (x / 4))
stq     $16,32($17)  # dest[4] = x % 4
```

Multiply / Divide Example #3

Non-power of 2

- Only optimize multiplication
- 5x = 4x + x
- 25x = 16x + 8x + x
  = 4(4x + x) + (4x + x)

```
void
uweight5(unsigned long x,
         unsigned long *dest)
{
  dest[0] = 5 * x;
  dest[1] = 5 * 5 * x;
  dest[2] = -5 * x;
}
```

```
s4addq  $16,$16,$1    # 5x
stq     $1,0($17)     # dest[0] = 5x

s4addq  $1,$1,$1      # 25x
stq     $1,8($17)     # dest[1] = 25x

lda     $1,-5         # $1 = -5
mulq    $16,$1,$1     # $1 = -5x
stq     $1,16($17)    # dest[2] = -5x
```


Omitting Stack Frame

- Reduces strength of general procedure call

Leaf Procedure

- Does not call any other procedures

All Local Variables Can be Held in Registers

- Not too many
- No local structures or arrays
  - Suppose allocate array `int a[6]` as registers `$16–$21`
    - » How would you generate code for `a[i]`?
- No address operations
  - `&x` cannot be generated if x is in register

Performance Improvements

- Minor saving in stack space
- Eliminates time to setup and undo stack frame

Keeping Data in Registers

Computing Integer Sum `z = x + y`

- Integer data stored in registers `r1`, `r2`, `r3`
  `addq $1, $2, $3`
  – 1 clock cycle
- Data addresses stored in registers `r1`, `r2`, `r3`
  `ldq $4, 0($1)`
  `ldq $5, 0($2)`
  `addq $4, $5, $6`
  `stq $6, 0($3)`
  – 4 clock cycles

Computing Double Precision Sum `z = x + y`

- Register data: 4 clock cycles
- Memory data: 7 clock cycles

Memory Optimization Example

Procedure `product1`

- Compute product of array elements and store at `*dest`
- Each iteration requires ~11 cycles (assuming simplified pipeline)

```
void
product1(double vals[],
    double *dest, long int cnt)
{
  long int i;
  *dest = 1.0;
  for (i = 0; i < cnt; i++)
    *dest = *dest * vals[i];
}
```

```
Loop:
  s8addq $2,$16,$1
  ldt $f1,0($17)         <- Load *dest
  ldt $f10,0($1)
  mult $f1,$f10,$f1
  stt $f1,0($17)         <- Store *dest
  addq $2,1,$2
  cmplt $2,$18,$1
  bne $1,Loop
```

4 cycle delay

Memory Optimization Example (Cont.)

Procedure `product2`

- Compute product of array elements and store at `*dest`
- Accumulate in register
- Each iteration takes ~6 cycles (roughly twice as fast)

```
void
product2(double vals[],
  double *dest, long int cnt)
{
  int i;
  double prod = 1.0;
  for (i = 0; i < cnt; i++)
    prod = prod * vals[i];
  *dest = prod;
}
```

```
# $2 = i, $16 = vals, $18 = cnt
# $f10 = prod
Loop:
  s8addq $2,$16,$1     # $1 = &vals[i]
  ldt $f1,0($1)        # $f1 = vals[i]
  mult $f10,$f1,$f10   # prod *= vals[i]
  addq $2,1,$2         # i++
  cmplt $2,$18,$1      # if (i<cnt) then
  bne $1,Loop          #  continue looping
```

But why didn't the compiler generate this code for `product1`?

Blocker #1: Memory Aliasing

Aliasing

- Two different memory references specify single location

Example

- `double a[3] = { 3.0, 2.0, 5.0 };`
- `product1(a, &a[2], 3) -->    { 3.0, 2.0,      }`
- `product2(a, &a[2], 3) -->    { 3.0, 2.0,      }`

Observations

- 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

Pointer Code

All C arrays indexed by address arithmetic

- `a[i]` same as `*(a+i)`
  - Access at location a + K*i, for object of size K
- Just converting array references to pointer references has no effect

Pointer code can *sometimes* reduce overhead associated with array addressing and loop testing

e.g.

```
int i = 0;
do {
  a[i] = 0;
  i++;
} while (i < 100);
```

```
int *ptr = &a[0];
int *end_ptr = &a[100];
do {
  *ptr = 0;
  ptr++;
} while (ptr != end_ptr);
```

BUT:

- Signficantly less readable
- ***Often inhibits key loop optimizations in good compilers!!!***

Pointer Code Example

Procedure `product3`

- Compute product of array elements and store at `*dest`
- Each iteration takes ~5 cycles
  - Can't do much better (in this case), since mult takes 4 cycles
- With more functional units or lower mult latency, we could do better:
  - requires loop unrolling or software pipelining (discussed later)

```
void product3(double vals[],
  double *dest, long int cnt)
{
  double *val_end = vals+cnt;
  double prod = 1.0;
  if (cnt <= 0) {
    *dest = prod;
    return;
  }
  while (vals != val_end)
    prod = prod * *vals++;
  *dest = prod;
}
```

```
# $16 = vals,  $2 = val_end
# $f10 = prod
Loop:
  ldt $f1,0($16)       # $f1 = *vals
  mult $f10,$f1,$f10   # prod *= vals
  addq $16,8,$16       # vals++
  subq $16,$2,$1       # if (vals != val_end)
  bne $1,Loop          #  continue looping
```

Code Motion

Move Computation out of Frequently Executed Section

- if guaranteed to always give same result
- out of loop

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

→

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


What's in a Loop?

For Loop Form

```
for (Init; Test; Update)
  Body
```

While Loop Equivalent

```
Init;
while(Test) {
  Body;
  Update;
}
```

Machine Code Translation

```
  Init
  t = Test
  beq t Done
Loop:
  Body
  Update
  t = Test
  bne t Loop
Done:
```

- *Update* and *Test* are part of the loop!
- *Init* is not

Code Motion Examples

- Sum Integers from 1 to n!

Bad

```
sum = 0;
for (i = 0; i <= fact(n); i++)
  sum += i;
```

Better

```
sum = 0;
fn = fact(n);
for (i = 0; i <= fn; i++)
  sum += i;
```

```
sum = 0;
for (i = fact(n); i > 0; i--)
  sum += i;
```

Best

```
fn = fact(n);
sum = fn * (fn + 1) / 2;
```


Blocker #2: Procedure Calls

Why couldn't the compiler move fact(n) out of the inner loop?

Procedure May Have Side Effects

- i.e, alters global state each time called

Function May Not Return Same Value for Given Arguments

- Depends on other parts of global state

Why doesn't compiler look at code for fact(n)?

- Linker may overload with different version
  - Unless declared static
- Interprocedural optimization is not used extensively due to cost

Warning

- Compiler treats procedure call as a black box
- Weak optimizations in and around them

Common Subexpressions

Detect Repeated Computation in Two Expressions

Multiply Ex #3

```
...
dest[0] = 5 * x;
dest[1] = 5 * 5 * x;
...
```

```
s4addq   $16,$16,$1        # $1 = 5x
stq      $1,0($17)         # dest[0] = 5x
s4addq   $1,$1,$1          # $1 = 25x
stq      $1,8($17)         # dest[1] = 25x
```

Compilers Make Limited Use of Algebraic Properties

```
/* 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;
```

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

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

Basic Optimization Summary

Reduction in Strength

- **Shift, add instead of multiply or divide**
  - ☞ *Compilers are good at this*
- **Procedure with no stack frame**
  - ☞ *Compilers are good at this*
- **Keep data in registers rather than memory**
  - ☞ *Compilers are not good at this, since concerned with aliasing*
- **Pointer arithmetic**
  - ☞ *Some compilers are good at this* (e.g., CC on Alpha, SGI)

Code Motion

☞ *Compilers are not very good at this, especially when procedure calls*

Share Common Subexpressions

☞ *Compilers have limited algebraic reasoning capabilities*

Advanced Optimizations

Code Scheduling

- **Reorder operations to improve performance**
  - Especially for multi-cycle operations

Loop Unrolling

- **Combine bodies of several iterations**
- **Optimize across iteration boundaries**
- **amortize loop overhead**
- **Improve code scheduling**

Software Pipelining

- **Spread code for iteration over multiple loop executions**
- **Improve code scheduling**

Warning

- **Benefits depend heavily on particular machine**
- **Best if performed by compiler**

Multicycle Operations

Alpha Instructions Requiring > 1 Cycle (partial list)

`mull`	*(32-bit integer multiply)*	**8**
`mulq`	*(64-bit integer multiply)*	**16**
`addt`	*(fp multiply)*	**4**
`mult`	*(fp add)*	**4**
`divs`	*(fp single-precision divide)*	**10**
`divt`	*(fp double-precision divide)*	**23**

Operation

- **Instruction initiates multicycle operation**
- **Successive operations can potentially execute without delay**
  - as long as they don't require the result of the multicycle operation
  - and sufficient hardware resources are available
- **If there is a problem, stall the processor until operation completed**

Code Scheduling Strategy

Get Resources Operating in Parallel

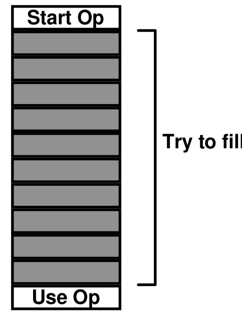
- Integer data path
- Integer multiply / divide hardware
- FP adder, multiplier, divider

Method

- Fill space between operation initiation and operation use
- With computations that do not require result or same hardware resources

Drawbacks

- Highly hardware dependent
  - Even among processor models
- Tricky to get maximum performance



Code Scheduling Example

Attempt to hide the long latency of division

- compiled using Digital's `cc` (`gcc` is not so great at scheduling)

```
double fpdiv(double f1,
 double f2, long int a[])
{
  a[0] = 0;
  a[1] = 1;
  a[2] = 2;
  a[3] = 3;
  a[4] = 4;
  a[5] = 5;
  a[6] = 6;
  a[7] = 7;
  a[8] = 8;
  a[9] = 9;
  a[10] = 10;
  a[11] = 11;
  return f1/f2;
}
```

```
divt     $f16,$f17,$f0
bis      r31, 0x1, r2
stq      r31, 0(r18)
bis      r31, 0x2, r3
stq      r2, 8(r18)
bis      r31, 0x3, r4
stq      r3, 16(r18)
bis      r31, 0x4, r5
stq      r4, 24(r18)
bis      r31, 0x5, r6
stq      r5, 32(r18)
bis      r31, 0x6, r7
stq      r6, 40(r18)
...
stq      r20, 88(r18)
ret      r31, (r26), 1
```

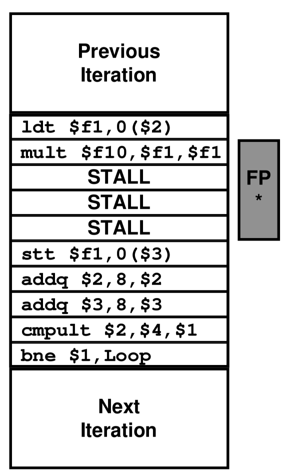
23 instructions between `divt` and return

Code Scheduling Example #2

- Multiply elements of vector b by scalar c and store in vector a
- Compiles into 7 instructions (`gcc -O`)
- Requires ~10 cycles
  - 3 cycle stall from FP multiply

```
#define CNT 256
static double a[CNT], b[CNT];
static double c;

void loop1(void)
{
  double *anext = a;
  double *bnext = b;
  double *bdone = b+CNT;
  double tc = c;
  while (bnext < bdone)
    *anext++ = *bnext++ * tc;
}
```



Loop Unrolling

Advanced Optimization

- Combine loop iterations
- Reduce loop overhead
- Expose optimizations across iterations
  - e.g., common subexpressions
- More opportunities for clever scheduling

```
for (i=0; i<n; i++)
  Body_i
```

Unroll by k

```
for (i=0; i<n%k; i++)
  Body_i
for (; i<n; i+=k) {
  Body_i
  Body_i+1
  •••
  Body_i+k–1
}
```

Loop Unrolling Example

- **Unrolling by 2 gives ~16 cycle loop**
  – 8 cycles / element
- **4 overhead operations spread over 2 elements**
- **Roughly a 25% speedup over original code**

```
void loop2(void)
{
  double *anext = a;
  double *bnext = b;
  double *bdone = b+CNT;
  double tc = c;
  while (bnext < bdone) {
    double b0 = bnext[0];
    double b1 = bnext[1];
    bnext += 2;
    anext[0] = b0 * tc;
    anext[1] = b1 * tc;
    anext += 2;
  }
}
```

ldt $f1,0($2)	
ldt $f10,8($2)	
addq $2,16,$2	
mult $f1,$f11,$f1	FP *
STALL	
STALL	
STALL	
stt $f1,0($3)	
mult $f10,$f11,$f10	FP *
STALL	
STALL	
STALL	
stt $f10,8($3)	
addq $3,16,$3	
cmpult $2,$4,$1	
bne $1,Loop	

Software Pipelining

Advanced Optimization

- **Spread code for single iteration over multiple loops**
- **Tends to stretch out data dependent operations**
- **Allows more effective code scheduling**

```
for (i=0; i<n; i++) {
  Ai ; Bi ; Ci
}
```

3-way pipeline

```
A0 ; B0 ; A1
for (i=2; i<n; i++) {
  Ci–2 ; Bi–1 ; Ai
}
Cn–2 ; Bn–1 ; Cn–1 ;
```


Software Pipelining Example

```
void loop1(void)
{
  double *anext = a;
  double *bnext = b;
  double *bdone = b+CNT;
  double tc = c;
  while (bnext < bdone) {
    double load = *bnext++;
    double prod = load * tc;
    *anext++ = prod;
  }
}
```

- **Operations**
  – A: load element from b
  – B: multiply by c
  – C: store in a
- **7 cycles / iteration**
  – No stalls!

3-Way Pipelined

```
void pipe(void)
{
  double tc = c;
  double prod = b[0] * tc; /* A0, B0 */
  double load = b[1];      /* A1 */
  double *anext = a;
  double *bnext = b+2;
  double *bdone = b+CNT;
  while (bnext < bdone) {
    *anext++ = prod;       /* Ci-2 */
    prod = load * tc;      /* Bi-1 */
    load = *bnext++;       /* Ai */
  }
  a[CNT-2] = prod;         /* Cn-2 */
  a[CNT-1] = load * tc;    /* Bn-1, Cn-1 */
}
```


Advanced Optimization Summary

Code Scheduling

☞ *Compilers getting good at this* (e.g., CC on Alpha, SGI)

Loop Unrolling

☞ *Compilers getting good at this* (e.g., CC on Alpha, SGI)

» e.g., `bubbleUp2` in Homework H3

Software Pipelining

☞ *Compilers getting good at this* (e.g., CC on Alpha, SGI)

Warning

- **Benefits depend heavily on particular machine**
- **Best if performed by compiler**

Role of Programmer

How should I write my programs, given that I have a good, optimizing compiler?

Don’t: Smash Code into Oblivion

- **Hard to read, maintain, & assure correctness**

Do:

- **Select best algorithm**
- **Write code that’s readable & maintainable**
  - Procedures, recursion, without built-in constant limits
  - Even though these factors can slow down code
- **Eliminate optimization blockers**
  - Allows compiler to do its job

Focus on Inner Loops

- **Do detailed optimizations where code will be executed repeatedly**
- **Will get most performance gain here**