Performance Tuning

CS 217

Principles

- Don't optimize your code
  - Your program might be fast enough already
  - Machines are getting faster and cheaper every year
  - Memory is getting denser and cheaper every year
  - Hand optimization may make the code less readable, less robust, and more difficult to test
- Performance tuning of bottlenecks
  - Identify performance bottlenecks
  - Machine independent algorithm improvements
  - Machine instruction dependent, but architecture dependent improvements
- Try not to sacrifice correctness, readability and robustness

Amdahl's Law: Only Bottlenecks Matter

- Definition of speedup:

$$Speedup = \frac{Original}{Enhanced} \qquad Enhanced = \frac{Original}{Speedup}$$

- Amdahl's law (1967):

$$OverallSpeedup = \frac{1}{(1-f) + \frac{f}{s}}$$

  - f is the fraction of program enhanced
  - s is the speedup of the enhanced portion

Examples

- Amdahl's law

$$OverallSpeedup = \frac{1}{(1-f) + \frac{f}{s}}$$

- What is the overall speedup if you make 10% of a program 90 times faster?

$$\frac{1}{(1-0.1) + \frac{0.1}{90}} \approx \frac{1}{0.9011} \approx 1.11$$

- What is the overall speedup if you make 90% of a program 10 times faster

$$\frac{1}{(1-0.9) + \frac{0.9}{10}} = \frac{1}{0.19} \approx 5.26$$

Identify Performance Bottlenecks

- Use tools such as gprof to learn where the time goes

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 76.21      3.46     3.46  6664590     0.00     0.00  partition
 16.74      4.22     0.76 54358002     0.00     0.00  swap
  3.74      4.39     0.17        1     0.17     0.17  fillArray
  2.86      4.52     0.13        1     0.13     4.35  quicksort
  0.44      4.54     0.02                             printArray
```

- More sophisticated tools
  - Tools that use performance counters to show cache miss/hit etc (e.g. VTune)
  - Tools for multiprocessor systems (for multi-threaded programs)
  - Tools to investigate where I/O operations take place

Strategies to Speedup

- Use a better algorithm
  - Complexity of the algorithm makes a big difference
- Simple code optimizations
  - Extract common expression: f(x*y + x*z) + g(x*y+x*z)
  - Loop unrolling:

```
for (i=0; i<N; i++)
    x[i]=y[i];

for (i=0; i<N; i+=4) {   /* if N is divisible by 4 */
    x[i]   = y[i];
    x[i+1] = y[i+1];
    x[i+2] = y[i+2];
    x[i+3] = y[i+3];
}
```

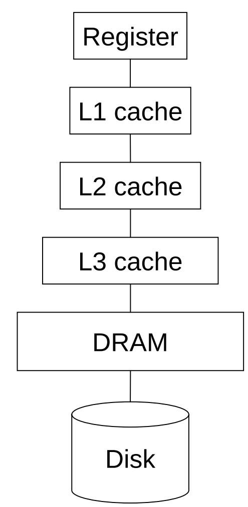
- Enable compiler optimizations
  - Modern compilers perform most of the above optimizations
  - Example: use level 3 optimization in gcc:
    gcc –O3 foo.c

Strategies to Speedup, con'd

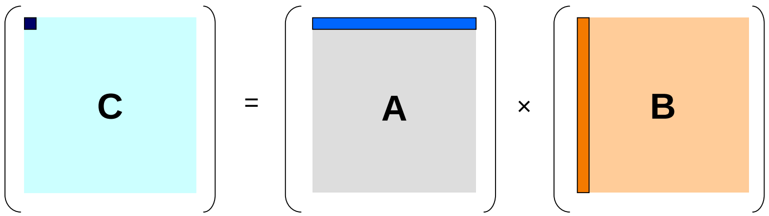
- Improve performance with deep memory hierarchy
  - Make the code cache-aware
  - Reduce the number of I/O operations
- Inline procedures
  - Remove the procedure call overhead (compilers can do this)
- Inline assembly
  - Almost never do this unless you deal with hardware directly
  - Or when the high-level language is in the way

Memory Hierarchy

- Hardware trends
  - CPU clock rate doubles every 18-24 months (50% per year)
  - DRAM and disk Access times improve at a rate about 10% per year
  - Memory hierarchy is getting deeper (L1, L2 and L3 caches)
- Software performance has become more sensitive to cache misses
  - Register: 1 cycle
  - L1 cache hit: 2-4 cycles
  - L2 cache hit: ~10 cycles
  - L3 cache hit: ~50 cycles
  - L3 miss: ~500 cycles
  - Disk I/O: ~30M cycles



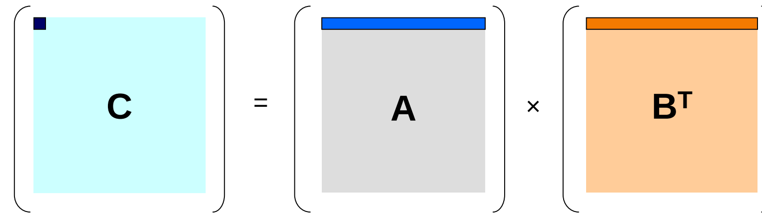
Example: Matrix Multiply




```
int i, j, k;
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      for (k=0; k<N; k++)
         C[i][j] += A[i][k] * B[k][j];
```

- How many cache misses?
- Execution time on tux (N=1000, -O3 with gcc): 18.5sec

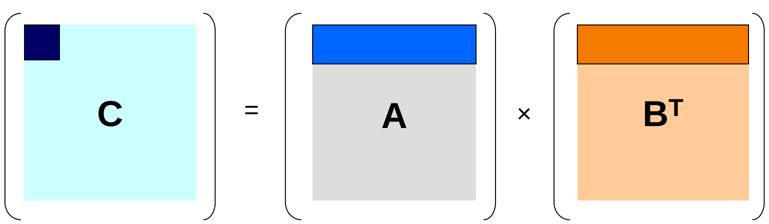
Transpose Matrix B First




```
int i, j, k;
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      for (k=0; k<N; k++)
         C[i][j] += A[i][k] * BT[j][k];
```

- What about the cache miss situation now?
- Execution time on tux (N=1000, -O3 with gcc): 13sec

A Blocked Matrix Multiply




```
int i, j, ii, jj, k, block;
block = 10;
for (ii=0; ii<N; ii+=block)
   for (jj=0; jj<N; jj+=block)
     for (i=ii; i<ii+block; i++)
        for (j=jj; j<jj+block; j++)
           for (k=0; k<N; k++)
              C[i][j] += A[i][k] * BT[j][k];
```

- Execution time on tux (N=1000, -O3 with gcc): 4.4sec

Inline Procedure

- To specify an inline procedure

```
static inline int plus5(int x)
{
   return x + 5;
}
```

- Is this better than using macro?

```
#define plus5(x)  (x+5)
```

Why Inline Assembly?

- For most system modules (>99%), programming in C delivers adequate performance
- It is more convenient to write system programs in C
  - Robust programming techniques apply to C better
  - Modular programming is easier
  - Testing is easier
- When do you have to use assembly?
  - You need to use certain instructions that the compiler don't generate (MMX, SSE, SSE2, and IA32 special instructions)
  - You need to access some hardware, which is not possible in a high-level language
- A compromise is to write most programs in C and as little as possible in assembly: inline assembly

Inline Assembly

- Basic format for gcc compiler

```
asm [volatile] ( "asm-instructions" );
__asm__ [volatile] ( "asm-instructions" );
```

  - "asm-instructions" will be inlined into where this statement is in the C program
  - The key word "volatile" is optional: telling the gcc compiler not to optimize away the instructions
  - Need to use "\n\t" to separate instructions. Otherwise, the strings will be concatenated without space in between.
- Example
  - `asm volatile( "cli" );`
  - `__asm__( "pushl %eax\n\t"`
    `"incl %eax" );`
- But, to integrate assembly with C programs, we need a contract on register and memory operands

Extended Inline Assembly

- Extended format

```
asm [volatile]
  ( "asm-instructions": out-regs: in-regs: used-
  regs);
```

  - Both "asm" and "volatile" can be enclosed by "__"
  - "volatile" is telling gcc compiler not to optimize away
  - "asm-instructions" are assembly instructions
  - "out-regs" provide output registers (optional)
  - "in-regs" provide input registers (optional)
  - "used-regs" list registers used in the assembly program (optional)

Register Allocation

- Use a single letter to specify register allocation constrain
- Example

```
int add2(int a, int b) {
    asm ("addl %0, %1"
      :
      : "r" (a), "r" (b) );
}
```

  - **gcc will save and load registers for you**
  - **If you use "a", "b", … "D", you will need to specify "%%eax", "%%ebx", …**

	Meaning
a	eax
b	ebx
c	ecx
d	edx
S	esi
D	edi
I	Constant value (0 to 31)
q	Allocate a register from eax, ebx, ecx, and edx
r	Allocate a register from eax, ebx, ecx, edx, esi, edi
g	eax, ebx, ecx, edx or variable in memory
A	eax and edx combined into a 64-bit integer

Compile with –O (Optimize)

C program

```
int add2(int a, int b) {
    asm ("addl %0, %1“
       :
       : "r" (a), "r" (b)
  );
}
```

gcc –S -O foo.c

```
        .text
.globl add2
        .type
add2:
        pushl   %ebp
        movl    %esp, %ebp
        leave
        ret
```

gcc optimized away the “asm” instructions!

Result Is Elsewhere

C program

```
int add2(int a, int b) {
   asm volatile
      ("addl %0, %1“
       :
       : "r" (a), "r" (b) );
}
```

gcc –S –O foo.c

```
        .text
.globl add2
add2:
        pushl %ebp
        movl  %esp, %ebp
        movl  8(%ebp), %edx
        movl  12(%ebp), %eax
#APP
        addl %eax, %edx
#NO_APP
        leave
        ret
```

The result is not in %eax.

Constrain Register Allocation

C program

```
int add2(int a, int b) {
   asm ("addl %1, %%eax“
      :
      : “a" (a), "r" (b) );
}
```

gcc –S –O foo.c

```
        .text
.globl add2
add2:
        pushl %ebp
        movl  %esp, %ebp
        movl  8(%ebp), %eax
        movl  12(%ebp), %edx
#APP
        addl %edx, %eax
#NO_APP
        leave
        ret
```


Summary

- Don’t optimize your code, unless it is really necessary
- Use a better algorithm is choice #1
- Then, tune the bottleneck first (Amdahl’s law)
  - Identify the bottlenecks by using tools
  - Make program cache aware
  - Reduce I/O operations
  - Inline procedures
  - Inline assembly (to access hardware including special instructions)
- Additional reading besides the textbook
  - Jon Bentley’s *Writing Efficient Programs* (Prentice-Hall, 1982), Programming Pearls and More Programming Pearls (Addision Wesley, 1986 and 1988)
  - John Hennessy and David Patterson’s Computer Organization and Design: The Hardware/Software Interface (Morgan Kaufman, 1997)

What's Covered in The Final Exam?

- Rephrase: What do I expect you all to know
  - Master the C language
  - Modules, interfaces and abstract data types
  - Memory allocation
  - Robust programming
  - Testing
  - Concept of computer architecture
  - Basic IA32 instruction set and assembly
  - How assemblers and linkers work
  - Use UNIX system services (signal, processes and interprocess communication)
  - How to write portable code
  - Performance tuning
- The final will be in COS 104, 1:30-3:30pm, 5/20
- Open book and open notes