Performance Tuning

CS 217

Principles

• Don’t optimize your code
  o Your program might be fast enough already
  o Machines are getting faster and cheaper every year
  o Memory is getting denser and cheaper every year
  o Hand optimization may make the code less readable, less robust, and more difficult to test

• Performance tuning of bottlenecks
  o Identify performance bottlenecks
  o Machine independent algorithm improvements
  o Machine instruction dependent, but architecture dependent improvements

• Try not to sacrifice correctness, readability and robustness

Amdahl’s Law: Only Bottlenecks Matter

• Definition of speedup:
  
  \[
  \text{Speedup} = \frac{\text{Original}}{\text{Enhanced}} \quad \text{Enhanced} = \frac{\text{Original}}{\text{Speedup}}
  \]

• Amdahl’s law (1967):
  
  \[
  \text{OverallSpeedup} = \frac{1}{(1 - f) + \frac{f}{s}}
  \]

  o \( f \) is the fraction of program enhanced
  o \( s \) is the speedup of the enhanced portion

Examples

• Amdahl’s law
  
  \[
  \text{OverallSpeedup} = \frac{1}{(1 - 0.1) + \frac{0.1}{90}} \approx 1.11
  \]

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

• What is the overall speedup if you make 90% of a program 10 times faster?
Identify Performance Bottlenecks

• Use tools such as gprof to learn where the time goes
  Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>time</th>
<th>cumulative</th>
<th>self</th>
<th>seconds</th>
<th>calls</th>
<th>s/call</th>
<th>total</th>
<th>s/call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>76.21</td>
<td>3.46</td>
<td>3.46</td>
<td>6664590</td>
<td>0.00</td>
<td>0.00</td>
<td>partition</td>
<td></td>
</tr>
<tr>
<td>16.74</td>
<td>4.22</td>
<td>0.76</td>
<td>54358002</td>
<td>0.00</td>
<td>0.00</td>
<td>swap</td>
<td></td>
</tr>
<tr>
<td>3.74</td>
<td>4.39</td>
<td>0.17</td>
<td>1</td>
<td>0.17</td>
<td>0.17</td>
<td>fillArray</td>
<td></td>
</tr>
<tr>
<td>2.86</td>
<td>4.52</td>
<td>0.13</td>
<td>1</td>
<td>0.13</td>
<td>4.35</td>
<td>quicksort</td>
<td></td>
</tr>
<tr>
<td>0.44</td>
<td>4.54</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td>printArray</td>
<td></td>
</tr>
</tbody>
</table>

• 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:
    ```c
    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:
    ```bash
    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

\[
\begin{bmatrix}
C
\end{bmatrix}
= \begin{bmatrix}
A
\end{bmatrix} \times \begin{bmatrix}
B
\end{bmatrix}
\]

```c
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

\[
\begin{bmatrix}
C
\end{bmatrix}
= \begin{bmatrix}
A
\end{bmatrix} \times \begin{bmatrix}
B^T
\end{bmatrix}
\]

```c
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[j][k];
```

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

A Blocked Matrix Multiply

\[
\begin{bmatrix}
C
\end{bmatrix}
= \begin{bmatrix}
A
\end{bmatrix} \times \begin{bmatrix}
B^T
\end{bmatrix}
\]

```c
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
  ```c
  static inline int plus5(int x)
  {
    return x + 5;
  }
  ```
- Is this better than using macro?
  ```c
  #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-regns: in-regns: used-regns);
  ```
  - Both “asm” and “volatile” can be enclosed by “__”
  - “volatile” is telling gcc compiler not to optimize away
  - “asm-instructions” are assembly instructions
  - “out-regns” provide output registers (optional)
  - “in-regns” provide input registers (optional)
  - “used-regns” 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  |
| l         | 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

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

```
gcc –S -O foo.c
```

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

```
gcc −S −O foo.c
```

Result Is Elsewhere

C program

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

```
gcc –S −O foo.c
```

```
.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

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

```
gcc –S –O foo.c
```

```
.globl add2
.add2:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
#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)
  o Identify the bottlenecks by using tools
  o Make program cache aware
  o Reduce I/O operations
  o Inline procedures
  o Inline assembly (to access hardware including special instructions)
• Additional reading besides the textbook
  o 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