lecture 24

- discuss Assignment 4 Question 3

- more on cache design

April 11, 2016
Recursive Solution

```c
findRoot(a,c) {
    b = (a + c) / 2
    // calculate p(a) and p(b) i.e. call evaluate() twice
    if ( p(b) == 0 )
        return b
    else if ( abs(a - c) < epsilon )
        return b
    else if ( p(a)*p(b) > 0 )
        a = b
    else
        c = b
    return findRoot(a,c)
}
```

Note: The stack frame for evaluate() is 16 bytes.
Iterative Solution

```c
findRoot(a,c) {

    while (true){
        b = (a + c) / 2
        // calculate p(a) and p(b) i.e. call evaluate( ) twice
        if ( p(b) == 0 )
            return b
        else if ( abs(a - c) < epsilon)
            return b
        else if ( p(a)*p(b) > 0 )
            a = b
        else
            c = b
    }
}
```
Q3b)

For my solutions to findRoot, and using constants in the starter code, the number of instructions at runtime was:

- Iterative: 5900
- Recursive: 6257 (only 357 more than iterative)

My recursive code uses more instructions, because there were more stack operations required. The difference is not that big though. Why? Because most of the time was spent in evaluate() and power() and other instructions that are the same in the two versions.
Q: How many recursive calls?
   (How many times through while loop?)

A: 22.

Q: How can this value be explained?

INTERVAL: .float -1.53, 0.51 // about $2^1$
EPSILON:  .float 0.000001 // about $2^{-20}$

$$b = (a + c) / 2$$ so we shrink interval size by half each time.

A: $2^{22}$ is roughly $2^1 / 2^{-20}$
Q3a) Compare cache hit rates.

Number of Memory accesses:

- Iterative: 613
- Recursive: 908 (only 295 more than iterative)

You may have found greater or lesser differences between Iterative and Recursive, depending on how many save vs temporary registers you used.
We fix the cache size to be 1024 bytes, so the blocksize is inversely related to the number of blocks.

Here I compare the *number of misses* (rather than hit rates).

<table>
<thead>
<tr>
<th>numblocks</th>
<th>size of block in bytes</th>
<th>#misses (recursive)</th>
<th>#misses (iterative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>8</td>
<td>76</td>
<td>13</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>39</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1024</td>
<td>93</td>
<td>93</td>
</tr>
</tbody>
</table>
Observation 1: (both for recursive and iterative)

The miss rate falls steadily as the size of each block increases from 8 up to 256 bytes.

Q: Why?

A: The bigger each block, the more stack frames it can hold.

Note: for the iterative method, the increase in block size matters only up to about 64 bytes since the stack doesn't grow more than that.
For my recursive solution, each stack frame had 24 bytes ($ra, a, c, epsilon, and 2 save registers). Thus, the stack can grow to $22 \times 24 = 528$ bytes.

This suggests that using a big block size gives good performance.
Observation 2:

More cache misses for recursive than iterative.

Q: Why?

A:

<table>
<thead>
<tr>
<th>numblocks</th>
<th>block size (bytes)</th>
<th>#misses (recursive)</th>
<th>#misses (iterative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>8</td>
<td>76</td>
<td>13</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>39</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1024</td>
<td>93</td>
<td>93</td>
</tr>
</tbody>
</table>
findRoot() stack frame is 24 bytes.
evaluate() stack frame is 16 bytes.
power() doesn't use the stack at all.

recursive

iterative
Observation 3:

When the number of blocks decreases to 1, both recursive and iterative have big rise in cache misses. Why?

<table>
<thead>
<tr>
<th>numblocks</th>
<th>block size (bytes)</th>
<th>#misses (recursive)</th>
<th>#misses (iterative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>8</td>
<td>76</td>
<td>13</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>39</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1024</td>
<td>93</td>
<td>93</td>
</tr>
</tbody>
</table>
If each block has 1024 bytes (0x400), then block boundaries will be at addresses whose last three hex digits are 0x000, 0x400, 0x800, 0xc00.

Q: So why should one block be a problem?
.data

POLYORDER: .word 3
COEFFICIENTS: .word 1, -2, -1, 2

evaluate:

    addi $sp, $sp, -16
    sw $ra, 0($sp)
    swc1 $f20, 4($sp)
    sw $s0, 8($sp)
    sw $s1, 12($sp)

    lw $s0, POLYORDER
    la $s1, COEFFICIENTS

Stack
0x 7fff effc

data
0x 10010000
For a block size of 1024, each call to evaluate() will cause two cache refills for both recursive and iterative case.
Observation 4:

When the number of blocks decreases to 2, the recursive algorithm has an increase in cache misses. Why?

<table>
<thead>
<tr>
<th>numblocks</th>
<th>block size (bytes)</th>
<th>#misses (recursive)</th>
<th>#misses (iterative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>8</td>
<td>76</td>
<td>13</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>39</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1024</td>
<td>93</td>
<td>93</td>
</tr>
</tbody>
</table>
For a block size of 512, iterative code has very few cache misses. The recursive code has more cache misses, namely when stack pointer is near a block boundary.

The reason is that three blocks are being used but the cache only can hold $2 \times 512$ each.
WARNING: This example might suggest to you that the problem is due to having big blocks. That would be an incorrect conclusion.

Here is a more general way to state this problem situation:

We have two memory addresses (data or instructions) that:

- are accessed frequently
- lie in different blocks
- have the same cache (or TLB) index.

In this situation, there will be many cache (or TLB) misses.

Such a situation can easily arise. How can we design caches differently to avoid such situations?
Recall the direct mapping method from lecture 18. For simplicity, consider just the case of one word per block.

The problem occurs if two commonly occurring addresses have the same cache index (same block) but different tags.
Solution 1: "Fully Associative Cache"

Allow *any block* in main memory to be stored *at any line* in the cache.

The tag now contains the entire block number, including the cache index bits which were previously used to say which line of the cache we mapped to.

This requires checking every line of the cache every time. We would need a much more complicated circuit -- details not shown below.
Fully associate caches are sometimes used for TLB's. (Sketch on previous slide would be changed to use virtual address rather than physical address, and data word would be the physical word address.)

The circuit is much more complicated than what was shown on the previous slide, since the equality test needs to be combined with a large multiplexor to select one of the words.

A key issue:

When there is a TLB refill, how to choose which entry to remove?
Solution 2: "N way Set Associative Cache"

Data and instruction caches typically do not use fully associative scheme.

Instead, they are a compromise between direct mapping method that you learned and fully associative.

Partition cache into N fully associative caches.

A direct mapping (lower order bits of block number) maps an address to one of these N caches.

Details omitted! I mention these for your interest only.
Announcements

- Wednesday: Java Virtual Machine (connection to MIPS)

- Final exam is multiple choice (see end of lecture 21)

  Bring a pencil!

- Course Evaluations