CS 201
Computer Systems Programming II – Caching

Notes based on robboy’s notes and instructors notes from B&O’s web site: http://csapp.cs.cmu.edu

Review of last lecture…

- Program exhibit two types of locality
  - Spatial locality
  - Temporal locality

Review of last lecture…

- Caching
  - Cache hit
  - Cache miss
  - Placement / replacement policies

Review of last lecture…

- Caching problems
  - Cold miss
    - Another term?
  - Conflict miss
  - Capacity miss
Review of last lecture…

- Measuring cache performance
  - Miss Rate
  - Hit Rate
  - Hit Time
  - Miss Penalty

Review of last lecture…

- Set associative caches
  - Given address how does one map data to a set-associative cache?
  - How do you look up an address?

Direct-Mapped Cache

- Given address how does one map data to a direct-mapped cache?
- What is the tag for?

Set associative caches

- How do you look up an address?

Why is tag the upper bits?
Review of last lecture...

- Set associative caches
  - What is the drawback of large sets?

Review of last lecture...

- What is the purpose of a separate data and instruction cache?
  - Advantages?
  - Disadvantages?

In class exercise

- Suppose we have
  1. Physical address: 0x00073AFD
  2. Cache with:
     - direct-mapped cache
     - 8-bytes per row of cache
     - 1MB cache

Where does this address end up in the cache? Give another address that maps to the same line. How many bits in tag?

In class exercise

- A processor has 32-bit addresses. The L2 cache is 256 K-bytes in size, 8-way set associated with a block size of 64 bytes.

- How many block offset bits are there, how many set index bits, and how many tag bits?

- Given address 0x00284722
  - What is the block offset, what is the set index, and what is the tag?
Matrix Multiplication Example

- Major Cache Effects to Consider
  - Total cache size
    - Exploit temporal locality and keep the working set small (e.g., by using blocking)
  - Block size
    - Exploit spatial locality

- Description:
  - Multiply N x N matrices
  - O(N^3) total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

```c
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Line size = 32B (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows

- Analysis Method:
  - Look at access pattern of inner loop

Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations

- Stepping through columns in one row:
  - for (i = 0; i < N; i++)
    - accesses successive elements
    - if block size (B) > 4 bytes, exploit spatial locality
      - compulsory miss rate = 4 bytes / B

- Stepping through rows in one column:
  - for (i = 0; i < n; i++)
    - accesses distant elements
    - no spatial locality!
      - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

- Misses per Inner Loop
  - Iteration:
    - A: 0.25
    - B: 1.0
    - C: 0.0
Matrix Multiplication (jik)

```c
/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

- **Misses per Inner Loop**
- **Iteration:**
  - A: 0.25
  - B: 1.0
  - C: 0.0

Matrix Multiplication (kij)

```c
/* kij */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

- **Misses per Inner Loop**
- **Iteration:**
  - A: 0.0
  - B: 0.25
  - C: 0.25

Matrix Multiplication (ikj)

```c
/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

- **Misses per Inner Loop**
- **Iteration:**
  - A: 0.0
  - B: 0.25
  - C: 0.25

Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

- **Misses per Inner Loop**
- **Iteration:**
  - A: 1.0
  - B: 0.0
  - C: 1.0
Matrix Multiplication (kji)

```c
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

- Misses per Inner Loop
  - Iteration:
    - Column-wise: A
    - Fixed: B
    - Column-wise: C

Summary of Matrix Multiplication

- ijk (& jik):
  - 2 loads, 0 stores
  - misses/iter = 1.25

- kij (& ikj):
  - 2 loads, 1 store
  - misses/iter = 0.5

- jki (& kji):
  - 2 loads, 1 store
  - misses/iter = 2.0

Pentium Matrix Multiply Performance

- Miss rates are helpful but not perfect predictors.
  - Code scheduling matters, too.

Improving Temporal Locality by Blocking

- Example: Blocked matrix multiplication
  - “block” (in this context) does not mean “cache block”.
  - Instead, it means a sub-block within the matrix.
  - Example: N = 8; sub-block size = 4

<table>
<thead>
<tr>
<th>A_{11} A_{12}</th>
<th>B_{11} B_{12}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{21} A_{22}</td>
<td>B_{21} B_{22}</td>
</tr>
</tbody>
</table>

C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{12}B_{22}
C_{12} = A_{11}B_{12} + A_{12}B_{22}
C_{21} = A_{21}B_{11} + A_{22}B_{21}
C_{22} = A_{21}B_{12} + A_{22}B_{22}

Key idea: Sub-blocks (i.e., A_{xy}) can be treated just like scalars.
**Blocked Matrix Multiply (bijk)**

```c
for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;

    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++) {
            for (j=jj; j < min(jj+bsize,n); j++) {
                sum = 0.0
                for (k=kk; k < min(kk+bsize,n); k++) {
                    sum += a[i][k] * b[k][j];
                }
                c[i][j] += sum;
            }
        }
    }
}
```

**Blocked Matrix Multiply Analysis**

- Innermost loop pair multiplies a $1 \times \text{bsize}$ sliver of $A$ by a $\text{bsize} \times \text{bsize}$ block of $B$ and accumulates into a $1 \times \text{bsize}$ sliver of $C$
- Loop over $i$ steps through $n$ row slivers of $A$ & $C$, using same $B$

**Pentium Blocked Matrix Multiply Performance**

- Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)
- Relatively insensitive to array size.

**In class exercise**

- Suppose we have a direct mapped, 512-kbyte data cache with 32-byte cache blocks. Write a program that exhibits extremely poor cache locality.
- Can the program be extended to show extremely poor cache locality with a 2-way set associative cache?
Writing to memory

- What happens when you write to memory that is not in the cache?
  - Treated like a cache miss:
    - Bring block into the cache
    - Update the cache line
    - Subsequent reads and writes will have a cache hit

Cache write policies

- Advantages of write-through caches?
- Disadvantages of write-through caches?

In class exercise

- Suppose the time to read and write main memory is 6 times longer than reading or writing cache. What is the difference in performance to the program in the previous exercise?
Memory Management...

Building the Address Space

• Load time:
  • Allocate primary memory
  • Adjust addresses in address space
  • Copy address space from secondary to primary memory

Old school memory management

Limited versatility.

With multiple processes

Issue: Where do you load $p_1$’s address space into primary memory?
A System with Physical Memory Only

- Examples:
  - most Cray machines, early PCs, nearly all embedded systems

Addresses generated by the CPU correspond directly to bytes in physical memory

Some problems with physical memory only

- With multi-tasking, you have to dynamically relocate programs when loading them.
- If the stack overflows the area allocated for it, we’re in trouble.
- The same with the heap.
- With swapping, you have to dynamically relocate programs each time they are swapped in.
  - Is that even possible? How would you handle locally declared pointers?

VM address translation

A system with virtual memory

- Examples:
  - workstations, servers, modern PCs, etc.

Address Translation: Hardware converts virtual addresses to physical addresses via a lookup table (page table)
Example

- 32 bit addresses, page size is 4096 = 0x1000
- How many bits is the offset into a page?
- A page-aligned address has how many low order zero bits?

Example

- 32 bit addresses, page size is 4096 = 0x1000
- Consider some address: 0x3e80a123
- Low order 12 bits: offset within the page: 0x123
- Address with low order 12 bits masked out: address of the page: 0x3e80a000
- High order 20 bits alone are the page number: 0x3e80a