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

Locality

- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - Temporal locality: Recently referenced items are likely to be referenced in the near future.
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time.

Locality Example:

- Data
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference sum each iteration: **Temporal locality**

- Instructions
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Does this function have good locality? Why or why not?

```c
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

- Temporal locality

- Spatial locality

Question: Does this function have good locality? Why or why not?

```c
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            sum += a[i][j];
    return sum;
}
```
Locality Example

**Question:** Can you permute the loops so that the function scans the 3-d array $a[\cdot]$ with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sumarray3d(int a[M][N][N])
{
    int i, j, k, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum
}
```

Caching in a Memory Hierarchy

- Larger, slower, cheaper storage device at level $k+1$ is partitioned into blocks.
- Data is copied between levels in block-sized transfer units.

General Caching Concepts

- **Program needs object** $d$, which is stored in some block $b$.
- **Cache hit**
  - Program finds $b$ in the cache at level $k$. E.g., block 14.
- **Cache miss**
  - $b$ is not at level $k$, so level $k$ cache must fetch it from level $k+1$. E.g., block 12.
  - If level $k$ cache is full, then some current block must be replaced (evicted). Which one is the “victim”? Place the new block $b$ mod 4
    - Replacement policy: which block should be evicted? E.g., LRU

General Caching Concepts

- **Types of cache misses:**
  - **Cold (compulsary) miss**
  - Cold misses occur because the cache is empty.
  - **Conflict miss**
  - Most caches limit blocks at level $k+1$ to a small subset (sometimes a singleton) of the block positions at level $k$.
  - E.g. Block $i$ at level $k+1$ must be placed in block $(i \mod 4)$ at level $k+1$.
  - Conflict misses occur when the level $k$ cache is large enough, but multiple data objects all map to the same level $k$ block.
  - E.g. Referencing blocks $0, 0, 8, 8, 0, 8, \ldots$ would miss every time.
  - **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
### Examples of Caching in the Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What Cached</th>
<th>Where Cached</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-byte word</td>
<td>CPU registers</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32-byte block</td>
<td>On-Chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>32-byte block</td>
<td>Off-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware+ OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Network buffer</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>AFS/NFS</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server</td>
<td>1,000,000,000</td>
<td>Web proxy</td>
</tr>
</tbody>
</table>

### Memory Cache

- Stores values from main memory, recently accessed by the processor.
- Controlled by the hardware – completely invisible to software
  - except for the performance

### Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses/references)
  - Typical numbers:
    - 3-10% for L1
    - can be quite small (< 1%) for L2, depending on size, etc.
- **Hit Time**
  - Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
  - Typical numbers:
    - 2 clock cycles for L1
    - ~20 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
    - Typically ~100 cycles for main memory

### Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
- Hold frequently accessed blocks of main memory
- CPU looks first for data in L1, then in L2, then in main memory.
- Typical bus structure:
Inserting an L1 Cache Between the CPU and Main Memory

The transfer unit between the CPU register file and the cache is a 4-byte block.

The transfer unit between the cache and main memory is a 4-word block (16 bytes).

The tiny, very fast CPU register file has room for four 4-byte words.

The small fast L1 cache has room for two 4-word blocks.

The big slow main memory has room for many 4-word blocks.

The big slow main memory has room for many 4-word blocks.

General Org of a Cache Memory

Cache is an array of sets. Each set contains one or more lines. Each line holds a block of data.

 addressed caches

Addressing Caches

Address A:  t bits  s bits  b bits

<tag> <set index> <block offset>

The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.

Direct-Mapped Cache

Simplest kind of cache

Characterized by exactly one line per set.

Cache size:  \( C = B \times E \times S \) data bytes
Accessing Direct-Mapped Caches

- Set selection
  - Use the set index bits to determine the set of interest.

Line matching and word selection

- Line matching: Find a valid line in the selected set with a matching tag
- Word selection: Then extract the word

Direct-Mapped Cache Simulation

| M=16 byte addresses, B=2 bytes/block, S=4 sets, E=1 entry/set |
|-------------------|-------------------|-------------------|
| t=1               | s=2               | b=1               |
| x                 | xx                | x                 |

Address trace (reads):
0 [0000], 1 [0001], 13 [1101], 8 [1000], 0 [0000]

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>M(0-1)</td>
</tr>
<tr>
<td>(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>data</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>M(0-1)</td>
</tr>
<tr>
<td>(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>data</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>M(1-13)</td>
</tr>
<tr>
<td>(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>data</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>M(0-1)</td>
</tr>
<tr>
<td>(5)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Why Use Middle Bits as Index?

- High-Order Bit Indexing
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality
- Middle-Order Bit Indexing
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time
Set Associative Caches

- Characterized by more than one line per set

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Cache Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$E=2$ lines per set

Accessing Set Associative Caches

- Set selection
  - identical to direct-mapped cache

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Cache Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Accessing Set Associative Caches

- Line matching and word selection
  - must compare the tag in each valid line in the selected set.

- 1. The valid bit must be set.

- 2. The tag bits in one of the cache lines must match the tag bits in the address.

- 3. If (1) and (2), then cache hit, and block offset selects starting byte.

Multi-Level Caches

- Options: separate data and instruction caches, or a unified cache

Processor

- Regs
- L1 d-cache
- L2 Cache
- Unified L2 Cache
- Memory
- Disk

Options:
- size: 200 B, 8-64 KB, 1-4 MB SRAM, 128 MB DRAM, 30 GB
- speed: 3 ns, 6 ns, 60 ns, 8 ms
- $S/\text{Byte}$: $100/MB, $1.50/MB, $0.05/MB
- line size: 8 B, 32 B, 8 KB
Intel Pentium Cache Hierarchy

- **L1 Data**
  - 1 cycle latency
  - 16 KB
  - 4-way associativity
  - Write-through
  - 32B lines

- **L1 Instruction**
  - 16 KB, 4-way
  - 32B lines

- **L2 Unified**
  - 128KB–2 MB
  - 4-way associativity
  - Write-back
  - Write allocate
  - 32B lines

- **Main Memory**
  - Up to 4GB

Pentium-4

- L1 data cache
  - 8K bytes, on the cpu
  - 2 cycle latency
- L1 instruction cache replaced with “execution trace cache”
  - Optimized for fetching and decoding instructions
- L2 cache (unified? data only?)
  - 256K - 1 MB, also on the cpu
  - 8-way set associative, 64-byte cache line size
  - 18 cycle latency, even though it’s on the CPU
- 2 MB L3 cache off chip
- Memory has around 92 cycle latency

In class exercise

- Suppose we have
  1. Physical address: 0x00043B7A
  2. Cache with:
     - 2-way set associativity
     - 8-bytes per row of cache
     - 1MB cache

Where does this address end up in the cache?

Writing Cache Friendly Code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
  - cold cache, 4-byte words, 4-word cache blocks

```c
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}

int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = \( \frac{1}{4} = 25\% \)  
Miss rate = 100%
The Memory Mountain

- Read throughput (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

- Memory mountain
  - Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.

Memory Mountain Test Function

```c
/* The test function */
void test(int elems, int stride) {
  int i, result = 0;
  volatile int sink;
  for (i = 0; i < elems; i += stride)
    result += data[i];
  sink = result; /* So compiler doesn't optimize away the loop */
}
/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz) {
  double cycles;
  int elems = size / sizeof(int);
  test(elems, stride); /* warm up the cache */
  cycles = fcyc2(test, elems, stride, 0); /* call test(elems,stride) */
  return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

Memory Mountain Main Routine

```c
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10) /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23) /* ... up to 8 MB */
#define MAXSTRIDE 16 /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)
int data[MAXELEMS]; /* The array we'll be traversing */
int main() {
  int size; /* Working set size (in bytes) */
  int stride; /* Stride in array elements */
  double Mhz; /* Clock frequency */
  Datinit_data(data, MAXELEMS); /* Initialize each element in data to 1 */
  Mhz = mhz(0); /* Estimate the clock frequency */
  for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
    for (stride = 1; stride <= MAXSTRIDE; stride++)
      printf("%.1f	", run(size, stride, Mhz));
    printf("\n");
  }
  exit(0);
}
```

The Memory Mountain

- **Pentium III Xeon 550 MHz**
  - 16 KB on-chip L1 d-cache
  - 16 KB on-chip L1 i-cache
  - 512 KB off-chip unified L2 cache

- **Ridges of Temporal Locality**

- **Slopes of Spatial Locality**
Ridges of Temporal Locality

- Slice through the memory mountain with stride=1
  - Illuminates read throughputs of different caches and memory

A Slope of Spatial Locality

- Slice through memory mountain with size=256KB
  - Shows cache block size.

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

```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;
    }
}
```

Variable `sum` held in register

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

A: Access pattern of inner loop
B: Cache is not big enough
C: Matrix dimension is very large
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++)
    - sum += a[i][j];
  - 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++)
    - sum += a[i][0];
  - 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:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

Matrix Multiplication (kij)

```c
/* kij */
for (k=0; k<n; k++) {
  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:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
<td></td>
</tr>
</tbody>
</table>
Matrix Multiplication (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)

```
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)

```
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:**
    - A: 1.0
    - B: 0.0
    - C: 1.0

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
Miss rates are helpful but not perfect predictors.

- Code scheduling matters, too.

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

\[
\begin{bmatrix}
A_{i1} & A_{i2} \\
A_{j1} & A_{j2}
\end{bmatrix}
\times
\begin{bmatrix}
B_{k1} & B_{k2} \\
B_{l1} & B_{l2}
\end{bmatrix}
= 
\begin{bmatrix}
C_{i1} & C_{i2} \\
C_{j1} & C_{j2}
\end{bmatrix}
\]

Key idea: Sub-blocks (i.e., \(A_{xy}\)) can be treated just like scalars.

\[
\begin{align*}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{align*}
\]

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 \(1 \times \text{bsize}\) sliver of C
- Loop over \(i\) steps through \(n\) row slivers of A & C, using same B

```c
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;
    }
}
```

Innermost Loop Pair

- row sliver accessed \(\text{bsize}\) times
- block reused \(n\) times in succession
- Update successive elements of sliver
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.

Concluding Observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)