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

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

General organization of a cache memory

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

Cache size: $C = B \times E \times S$ data bytes
### Addressing caches

**Address A:**  
- \( t \) bits  
- \( s \) bits  
- \( b \) bits

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.

### Accessing direct-mapped caches

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

**Line matching and word selection**  
- Find a valid line in the selected set with a matching tag (line matching)  
- Then extract the word (word selection)
**Direct-mapped cache simulation**

- M = 16 byte addresses, B = 2 bytes/block, S = 4 sets, E = 1 entry/set
- Address trace (reads):
  - 0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]

**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

**Accessing set associative caches**

- Set selection
  - identical to direct-mapped cache
Accessing set associative caches

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

\[
\begin{array}{ccccccc}
\text{line} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\text{tag} & 0110 & & & & & & & \\
\text{valid bit} & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
\text{block offset} & W_0 & W_1 & W_2 & W_3 & & & & \\
\end{array}
\]

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
- TLB
- L1 Data Cache
- L1 Instruction Cache
- L2 Unified Cache
- Memory

size: 200 B 8-64 KB 1-4 MB SRAM 128 MB DRAM 30 GB
speed: 3 ns 3 ns 6 ns 60 ns 8 ms
$/\text{MB}: \quad$100/MB $1.50/MB $0.05/MB
line size: 8 B 32 B 32 B 8 KB

greater line size, higher associativity, more likely to write back

Intel Pentium cache hierarchy

Processor Chip
- Main Memory Up to 4 GB
- L1 Instruction 16 KB, 4-way 32 B lines
- L1 Data 16 KB, 4-way assoc Write-through 32 B lines
- L2 Unified 128 KB–2 MB 4-way assoc Write-back Write allocate 32 B lines
- Regs

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 (e.g., < 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:
  - 1 clock cycle for L1
  - 3-8 clock cycles for L2

Miss Penalty
- additional time required because of a miss
  - Typically 25-100 cycles for main memory
Writing cache friendly code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Example

- 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;
}
```

```c
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;
}
```

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 */
}
```

```c
/* 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 = fcyck(test, elems, stride, 0); /* call test(elems, stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

Memory mountain main routine

```c
/* main.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 */
    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(); /* 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

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
  » but may be able to hold in register

/* 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

Analysis Method:
- Look at access pattern of inner loop

Layout of arrays in memory

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[0][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++)
  sum += a[i][0];
- accesses distant elements
  - no spatial locality!
  - compulsory miss rate = 1 (i.e. 100%)

Matrix multiplication (ijk)

/* 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;
  }
}

/* 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;
  }
}

Inner loop:

Misses per Inner Loop Iteration:

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

Matrix multiplication (jik)

/* 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;
  }
}

/* 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;
  }
}

Inner loop:

Misses per Inner Loop Iteration:

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

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:

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>

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:

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

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:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
### Summary of matrix multiplication

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

```c
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;
    }
}
```

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

```c
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];
    }
}
```

**jki (& kji):**
- 2 loads, 1 store
- misses/iter = 0.5

```c
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;
    }
}
```

### Pentium matrix multiply performance

Notice that miss rates are helpful but not perfect predictors.

![Chips vs. Array size (n)](image)

### 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

```
\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\times
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
= 
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]
```

**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 analysis

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

```
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 $bsize$ times
- Block reused $n$ times in succession

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 accessed
  - Nested loop structure
  - Blocking (see text) is a general technique

All machines like “cache friendly code”

- Getting absolute optimum performance 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)