Memory Systems and Performance Engineering

Fall 2010
Basic Caching Idea

A. Smaller memory faster to access
B. Use smaller memory to cache contents of larger memory
C. Provide illusion of fast larger memory
D. Key reason why this works: locality
   1. Temporal
   2. Spatial
Levels of the Memory Hierarchy

**Capacity**
- **CPU Registers**
  - 100s Bytes
  - 300 – 500 ps (0.3-0.5 ns)
- **L1 and L2 Cache**
  - 10s-100s K Bytes
  - ~1 ns - ~10 ns
  - $1000s/ GByte
- **Main Memory**
  - G Bytes
  - 80ns- 200ns
  - ~ $100/ GByte
- **Disk**
  - 10s T Bytes, 10 ms (10,000,000 ns)
  - ~ $1 / GByte
- **Tape**
  - infinite
  - sec-min
  - ~$1 / GByte

**Access Time**
- ** Registers**
  - Instr. Operands
  - Blocks
- **L1 Cache**
  - Blocks
- **L2 Cache**
  - Blocks
- **Memory**
  - Pages
- **Disk**
  - Files
- **Tape**

**Cost**
- **Staging Xfer Unit**
  - prog./compiler
  - 1-8 bytes
  - cache cntl
  - 32-64 bytes
  - cache cntl
  - 64-128 bytes
  - OS
  - 4K-8K bytes
  - user/operator
  - Mbytes

**Upper Level**
- faster

**Lower Level**
- Larger
Cache Issues

**Cold Miss**
- The first time the data is available
- Prefetching may be able to reduce the cost

**Capacity Miss**
- The previous access has been evicted because too much data touched in between
- “Working Set” too large
- Reorganize the data access so reuse occurs before getting evicted.
- Prefetch otherwise

**Conflict Miss**
- Multiple data items mapped to the same location. Evicted even before cache is full
- Rearrange data and/or pad arrays

**True Sharing Miss**
- Thread in another processor wanted the data, it got moved to the other cache
- Minimize sharing/locks

**False Sharing Miss**
- Other processor used different data in the same cache line. So the line got moved
- Pad data and make sure structures such as locks don’t get into the same cache line
Simple Cache

A. 32Kbyte, direct mapped, 64 byte lines (512 lines)
B. Cache access = single cycle
C. Memory access = 100 cycles
D. Byte addressable memory
E. How addresses map into cache
   1. Bottom 6 bits are offset in cache line
   2. Next 9 bits determine cache line
F. Successive 32Kbyte memory blocks line up in cache
## Analytically Model Access Patterns

```c
#define S ((1<<20)*sizeof(int))
int A[S];

for (i = 0; i < S; i++) {
    read A[i];
}
```

Array laid out sequentially in memory

![Memory Layout](image)

### Access Pattern Summary
- Read in new line
- Read rest of line
- Move on to next line

### Assume `sizeof(int) = 4`
- `S` reads to `A`
- 16 elements of `A` per line
- 15 of every 16 hit in cache

**Total access time:**

\[
15 \times \frac{S}{16} + 100 \times \frac{S}{16}
\]

### What kind of locality?
- Spatial

### What kind of misses?
- Cold
Analytically Model Access Patterns

```c
#define S ((1<<20)*sizeof(int))
int A[S];

for (i = 0; i < S; i++) {
    read A[0];
}
```

**Access Pattern Summary**
- Read A[0] every time
- S reads to A
- All (except first) hit in cache
- Total access time: \( 100 + (S-1) \)

**What kind of locality?**
Temporal

**What kind of misses?**
Cold
#define S ((1<<20)*sizeof(int))
int A[S];

for (i = 0; i < S; i++) {
    read A[i % (1<<N)];
}

Access Pattern Summary
Read initial segment of A repeatedly

S reads to A
Assume 4 <= N <= 13
One miss for each accessed line
Rest hit in cache
How many accessed lines?
2^{(N-4)}

Total Access Time
2^{(N-4)}*100 + (S - 2^{(N-4)})

What kind of locality?
Spatial, Temporal

What kind of misses?
Cold
#define S ((1<<20)*sizeof(int))

int A[S];

for (i = 0; i < S; i++) {
    read A[i % (1<<N)];
}

---

**Access Pattern Summary**

Read initial segment of A repeatedly

S reads to A
Assume 14 <= N
First access to each line misses
Rest accesses to that line hit

**Total Access Time**

(16 elements of A per line)
(15 of every 16 hit in cache)

15*(S/16) + 100*(S/16)

What kind of locality?
Spatial

What kind of misses?
Cold, capacity
#define S ((1<<20)*sizeof(int))

int A[S];

for (i = 0; i < S; i++) {
    read A[(i*16) % (1<<N)];
}

---

Access Pattern Summary

Read every $16^{th}$ element of initial segment of $A$, repeatedly

$S$ reads to $A$
Assume $14 \leq N$
First access to each line misses
One access per line
Total access time:
$$100 \times S$$

What kind of locality?
None

What kind of misses?
Cold, conflict
#define S ((1<<20)*sizeof(int))
int A[S];

for (i = 0; i < S; i++) {
    read A[random()%S];
}

**Access Pattern Summary**

Read random elements of A

S reads to A

Chance of hitting in cache (roughly) = 8K/1G = 1/256

Total access time (roughly):

S(255/256)*100 + S*(1/256)

What kind of locality?
Almost none

What kind of misses?
Cold, Capacity, Conflict
Basic Cache Access Modes

A. No locality – no locality in computation
B. Streaming – spatial locality, no temporal locality
C. In Cache – most accesses to cache
D. Mode shift for repeated sequential access
   1. Working set fits in cache – in cache mode
   2. Working set too big for cache – streaming mode
E. Optimizations for streaming mode
   1. Prefetching
   2. Bandwidth provisioning (can buy bandwidth)
#define S ((1<<19)*sizeof(int))

int A[S];
int B[S];

for (i = 0; i < S; i++) {
    read A[i], B[i];
}

Access Pattern Summary
Read A and B sequentially

S reads to A, B
A and B interfere in cache
Total access time
2*100 * S

What kind of locality?
Spatial locality, but cache can’t exploit it...

What kind of misses?
Cold, Conflict
Analytically Model Access Patterns

```c
#define S ((1<<19+16)*sizeof(int))
int A[S];
int B[S];

for (i = 0; i < S; i++) {
    read A[i], B[i];
}
```

### Access Pattern Summary

- **Read** A and B sequentially

- S reads to A, B
- A and B almost, but don't, interfere in cache
- **Total access time**
  \[ 2\left(\frac{15}{16}S + \frac{1}{6}S*100\right) \]

### What kind of locality?
- Spatial locality

### What kind of misses?
- Cold
Set Associative Caches

A. Have sets with multiple lines per set
B. Each line in cache called a way
C. Each memory line maps to a specific set
D. Can be put into any cache line in its set
E. 32 Kbyte cache, 64 byte lines, 2-way associative
   1. 256 sets
   2. Bottom six bits determine offset in cache line
   3. Next 8 bits determine set
Analytically Model Access Patterns

```c
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];

for (i = 0; i < S; i++) {
    read A[i], B[i];
}
```

**Access Pattern Summary**
- Read A and B sequentially
- S reads to A, B
- A and B lines hit same way, but enough lines in way
- Total access time
  \[2\times \left(\frac{15}{16}S + \frac{1}{6}S\times 100\right)\]

**What kind of locality?**
- Spatial locality

**What kind of misses?**
- Cold
# Analytically Model Access Patterns

```c
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];

for (i = 0; i < S; i++) {
  read A[i], B[i], C[i];
}
```

**Access Pattern Summary**

- Read A and B sequentially
- S reads to A, B, C
- A, B, C lines hit same way, but NOT enough lines in way
- Total access time (with LRU replacement)
  
  \[ 3 \times S \times 100 \]

**What kind of locality?**

- Spatial locality
  
  (but cache can’t exploit it)

**What kind of misses?**

- Cold, conflict
Associativity

A. How much associativity do modern machines have?

B. Why don’t they have more?
Linked Lists and the Cache

```c
struct node {
    int data;
    struct node *next;
};
```

```c
sizeof(struct node) = 16
```

```c
for (c = l; c != NULL; c = c->next++) {
    read c->data;
}
```

Struct layout puts
4 struct node per cache line (alignment, space for padding)

Assume list of length S
Access Pattern Summary
  Depends on allocation/use

Best case - everything in cache
total access time = S
Next best – adjacent (streaming)
total access time = (3/4*S + 1/4*S*100)
Worst – random (no locality)
total access time =100*S

Concept of effective cache size
(4 times less for lists than for arrays)
Structs and the Cache

```c
struct node {
    int data;
    int more_data;
    int even_more_data;
    int yet_more_data;
    int flags;
    struct node *next;
};
```

```c
sizeof(struct node) = 32
```

```c
for (c = l; c != NULL; 
    c = c->next++) {
    read c->data;
}
```

2 struct node per cache line
(alignment, space for padding)

Assume list of length S

Access Pattern Summary
Depends on allocation/use

Best case - everything in cache
total access time = S

Next best – adjacent (streaming)
total access time = \((1/2*S + 1/2*S*100)\)

Worst – random (no locality)
total access time =\(100*S\)

Concept of effective cache size
(8 times less for lists than for arrays)
Parallel Array Conversion

```c
struct node {
    int data;
    int more_data;
    int even_more_data;
    int yet_more_data;
    int flags;
    struct node *next;
};
for (c - 1; c != -1;
    c = next[c]) {
    read data[c];
}
```

```
int data[MAXDATA];
int more_data[MAXDATA];
int even_more_data[MAXDATA];
int yet_more_data[MAXDATA];
int flags[MAXDATA];
int next(MAXDATA];
```

**Advantages:**
- Better cache behavior
  (more working data fits in cache)

**Disadvantages:**
- Code distortion
- Maximum size has to be known or
  Must manage own memory
Managing Code Distortion

typedef struct node *list;

typedef int list;

int data(list l) {
  return l->data;
}

int data(list l) {
  return data[l];
}

int more_data(list l) {
  return l->more_data;
}

int more_data(list l) {
  return more_data[l];
}

... list next(list l) {
  return l->next;
}

... list next(list l) {
  return next[l];
}

This version supports only one list
Can extend to support multiple lists (need a list object)
Matrix Multiply

A. Representing matrix in memory

B. Row-major storage

```c
double A[4][4];
A[i][j];
```

Or

```c
double A[16];
A[i*4+j];
```

C. What if you want column-major storage?

```c
double A[16];
A[j*4+i];
```
Standard Matrix Multiply Code

for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
        for (k = 0; k < SIZE; k++) {
            C[i*SIZE+j] += A[i*SIZE+k]*B[k*SIZE+j];
        }
    }
}

Look at inner loop only:

Only first access to C misses (temporal locality)
A accesses have streaming pattern (spatial locality)
B has no temporal or spatial locality
Access Patterns for A, B, and C

A \times B = C

Courtesy of Martin Rinard. Used with permission.
Memory Access Pattern

Scanning the memory

A = B

x
How to get spatial locality for B

A. Transpose B first, then multiply. New code:

```c
for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
        for (k = 0; k < SIZE; k++) {
            C[i*SIZE+j] += A[i*SIZE+k]*B[j*SIZE+k];
        }
    }
}
```

Overall effect on execution time?

11620 ms (original)

2356 ms (after transpose)
A \times \text{transpose}(B) = C

Courtesy of Martin Rinard. Used with permission.
## Profile Data

<table>
<thead>
<tr>
<th></th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Percent SSE Instructions</th>
<th>Instructions Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td>In C</td>
<td>4.78</td>
<td>0.24</td>
<td>0.02</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>0.15</td>
<td>0.02</td>
<td>50%</td>
<td>13,001,486,336</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
How to get temporal locality?

A. How much temporal locality should there be?
B. How many times is each element accessed? SIZE times.
C. Can we rearrange computation to get better cache performance?
D. Yes, we can block it!
E. Key equation (here $A_{11}, B_{11}$ are submatrices)

$$A_{11}...A_{1N} \times B_{11}...B_{1N} = \sum_k A_{1k} * B_{k1} ... \sum_k A_{1k} * B_{kN}$$

$$A_{N1}...A_{NN} \times B_{N1}...B_{NN} = \sum_k A_{Nk} * B_{k1} ... \sum_k A_{Nk} * B_{kN}$$
Blocked Matrix Multiply

```c
for (j = 0; j < SIZE; j += BLOCK) {
    for (k = 0; k < SIZE; k += BLOCK) {
        for (i = 0; i < SIZE; i+=BLOCK) {
            for (ii = i; ii < i+BLOCK; ii++) {
                for (jj = j; jj < j+BLOCK; jj++) {
                    for (kk = k; kk < k+BLOCK; kk++) {
                        C[ii*SIZE+jj] += A[ii*SIZE+kk]*B[jj*SIZE+kk];
                    }
                }
            }
        }
    }
}
(Warning – SIZE must be a multiple of BLOCK)
```

Overall effect on execution time?

11620 ms (original), 2356 ms (after transpose),
631 ms (after transpose and blocking)
After Blocking

\[ A \times \text{transpose}(B) = C \]

Courtesy of Martin Rinard. Used with permission.
Data Reuse in Blocking

Data reuse

- Change of computation order can reduce the # of loads to cache
- Calculating a row (1024 values of A)
  - A: $1024 \times 1 = 1024$ + B: $384 \times 1 = 394$ + C: $1024 \times 384 = 393,216 = 394,524$
- Blocked Matrix Multiply ($32^2 = 1024$ values of A)
  - A: $32 \times 32 = 1024$ + B: $384 \times 32 = 12,284$ + C: $32 \times 384 = 12,284 = 25,600$
# Profile Data

<table>
<thead>
<tr>
<th></th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Percent SSE Instructions</th>
<th>Instructions Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td>In C</td>
<td>4.78</td>
<td>0.24</td>
<td>0.02</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>0.15</td>
<td>0.02</td>
<td>50%</td>
<td>13,001,486,336</td>
</tr>
<tr>
<td>Tiled</td>
<td>0.49</td>
<td>0.02</td>
<td>0</td>
<td>39%</td>
<td>18,044,811,264</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Blocking for Multiple Levels

A. Can block for registers, L1 cache, L2 cache, etc.
B. Really nasty code to write by hand
C. Automated by compiler community
D. Divide and conquer an alternative (coming up)
int fib(int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;
    return (fib(n-1) + fib(n-2));
}

What does call stack look like?
fib(4)

How deep does it go?

What kind of locality?
Stages and locality

A. Staged computational pattern
   1. Read in lots of data
   2. Process through Stage1, …, StageN
   3. Produce results

B. Improving cache performance
   1. For all cache-sized chunks of data
      a. Read in chunk
      b. Process chunk through Stage1, …, StageN
      c. Produce results for chunk
   2. Merge chunks to produce results
Basic Concepts

A. Cache concepts
   1. Lines, associativity, sets
   2. How addresses map to cache

B. Access pattern concepts
   1. Streaming versus in-cache versus no locality
   2. Mode switches when working set no longer fits in cache

C. Data structure transformations
   1. Segregate and pack frequently accessed data
      a. Replace structs with parallel arrays, other split data structures
      b. Pack data to get smaller cache footprint
   2. Modify representation; Compute; Restore representation
      a. Transpose; Multiply; Transpose
      b. Copy In; Compute; Copy Out

D. Computation transformations
   1. Reorder data accesses for reuse
   2. Recast computation stages when possible
Intel® Core™ Microarchitecture – Memory Sub-system

Intel Core 2 Quad Processor

<table>
<thead>
<tr>
<th>L1 Data Cache</th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L1 Instruction Cache</th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2 Cache</th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 MB</td>
<td>64 bytes</td>
<td>14 cycles</td>
<td>24-way</td>
<td></td>
</tr>
</tbody>
</table>
Intel® Nehalem™ Microarchitecture – Memory Sub-system

Intel 6 Core Processor

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 Data Cache</strong></td>
<td>32 KB</td>
<td>64 bytes</td>
<td>4 ns</td>
<td>8-way</td>
</tr>
<tr>
<td><strong>L1 Instruction Cache</strong></td>
<td>32 KB</td>
<td>64 bytes</td>
<td>4 ns</td>
<td>4-way</td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td>256 KB</td>
<td>64 bytes</td>
<td>10 ns</td>
<td>8-way</td>
</tr>
<tr>
<td><strong>L3 Cache</strong></td>
<td>12 MB</td>
<td>64 bytes</td>
<td>50 ns</td>
<td>16-way</td>
</tr>
<tr>
<td><strong>Main Memory</strong></td>
<td>64 bytes</td>
<td>75 ns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Intel Core 2 Quad Processor

for(rep=0; rep < REP; rep++)
    for(a=0; a < N ; a++)

© Saman Amarasinghe 2009
Intel Core 2 Quad Processor

```c
for(r=0; r < REP; r++)
    for(a=0; a < 64*1024; a++)
```

© Saman Amarasinghe 2009
for(rep=0; rep < REP; rep++)
    for(a=0; a < N; a++)

© Saman Amarasinghe 2009
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
**TLB**

- **Page size is 4 KB**
- **Number of TLB entries is 512**

- **So, total memory that can be mapped by TLB is 2 MB**
- **L3 cache is 12 MB!**

- **TLB misses before L3 cache misses!**
Other issues

- Multiple outstanding memory references
- Hardware Prefetching
- Prefetching instructions
- TLBs
- Paging
6.172 Performance Engineering of Software Systems
Fall 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.