Memory Systems and Performance Engineering

Fall 2011
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

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Access Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>100s Bytes</td>
<td>300 – 500 ps (0.3-0.5 ns)</td>
</tr>
<tr>
<td>L1 and L2 Cache</td>
<td>10s-100s K Bytes</td>
<td>~1 ns - ~10 ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~ $1000s/ GByte</td>
</tr>
<tr>
<td>Main Memory</td>
<td>G Bytes</td>
<td>80ns- 200ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~ $100/ GByte</td>
</tr>
<tr>
<td>Disk</td>
<td>10s T Bytes, 10 ms</td>
<td>(10,000,000 ns)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~ $1 / GByte</td>
</tr>
<tr>
<td>Tape</td>
<td>infinite</td>
<td>sec-min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~$1 / GByte</td>
</tr>
</tbody>
</table>

Upper Level

Faster

L2 Cache

Cache ctrl
32-64 bytes

Cache ctrl
64-128 bytes

OS
4K-8K bytes

User/operator
Mbytes

Lower Level

Larger

Memory

Blocks

Instr. Operands

Registers

Disk

Pages

Files

Tape

Staging Xfer Unit

Prog./compiler
1-8 bytes

Cache ctrl

32-64 bytes

64-128 bytes

OS
4K-8K bytes

User/operator
Mbytes
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
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];
}
```

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*1*(S/16) + 1*100*(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
Analytically Model Access Patterns

```c
#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

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

| Cache |
#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\textsuperscript{th} element of initial segment of A, repeatedly

S reads to A
Assume 14 <= N
First access to each line misses
One access per line
Total access time: 
100 * S

What kind of locality? 
None
What kind of misses? 
Cold, conflict
Analytically Model Access Patterns

```c
#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): $S(\frac{255}{256})\times 100 + S \times (\frac{1}{256})$

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)
Analytically Model Access Patterns

#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:
  \[ 2\left(\frac{15}{16}S + \frac{1}{6}S \times 100\right) \]

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
```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
  \[2 \times (\frac{15}{16}S + \frac{1}{6}S \times 100)\]

What kind of locality?
- Spatial locality

What kind of misses?
- Cold
Analytically Model Access Patterns

```
#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)
\[3*S*100\]

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;
};

sizeof(struct node) = 16

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

- **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

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];
}

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

Disadvantages:
  Code distortion
  Maximum size has to be known or
  Must manage own memory

int data[MAXDATA];
int more_data[MAXDATA];
int even_more_data[MAXDATA];
int yet_more_data[MAXDATA];
int flags[MAXDATA];
int next[MAXDATA];
### Managing Code Distortion

```c
typedef struct node *list;

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

typedef int list;

int data(list l) {
    return data[l];
}
int more_data(list l) {
    return more_data[l];
}
...
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
   
   ```
   double A[4][4];
   A[i][j];
   
   Or
   
   double A[16];
   A[i*4+j]
   ```

C. What if you want column-major storage?

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

B

C

© Martin Rinard 2008
Memory Access Pattern

Scanning the memory

A

=  

B

X

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

- In C: Multiplied by 5x and 2x
- Transposed: Multiplied by 2x and 1x
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)

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

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

```
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

\[ A \times \text{transpose}(B) = C \]
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>ln 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>
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
Call Stack and Locality

What does call stack look like?

fib(4)

How deep does it go?

What kind of locality?

```c
int fib(int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;
    return (fib(n-1) + fib(n-2));
}
```
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></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>3 cycles</td>
<td>8-way</td>
</tr>
<tr>
<td><strong>L1 Instruction Cache</strong></td>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td>6 MB</td>
<td>64 bytes</td>
<td>14 cycles</td>
<td>24-way</td>
</tr>
</tbody>
</table>
## Intel® Nehalem™ Microarchitecture – Memory Sub-system

### Intel 6 Core 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>4 ns</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>4 ns</td>
<td>4-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>256 KB</td>
<td>64 bytes</td>
<td>10 ns</td>
<td>8-way</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L3 Cache</th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 MB</td>
<td>64 bytes</td>
<td>50 ns</td>
<td>16-way</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Memory</th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64 bytes</td>
<td>75 ns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Intel Core 2 Quad Processor

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

```
<table>
<thead>
<tr>
<th>N</th>
<th>L1</th>
<th>L2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32768</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>131072</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Runtime per access: 0
© Saman Amarasinghe 2009
Intel® Nehalem™ Processor

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

© Saman Amarasinghe 2009
Prefetching

• **Speculatively bring in the data**
  • Can hide the long memory latencies

• **Issues**
  • When to prefetch?
    • Too early → replaces useful data
    • Too late → pipeline already stalled
  • What to prefetch
    • Too conservative → cache misses for prefetchable data
    • Too aggressive → pollute the cache with unused data
  • Who should prefetch
    • Software based: Additional instructions given by programmer/compiler
    • Hardware based: dynamically look for patterns

• **Prefetching patterns**
  • Array prefetching (unit and stride access patterns)
  • Pointer prefetching (identify delinquent accesses + complex predictors)
Random access within an address range
Address range → working set
Defeat the prefetcher

mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
mask = \((1 << n) - 1\);

```c
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
```

What is going on???

Let's look at performance counters to explain.
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}

© Saman Amarasinghe 2009

L1 Cache Miss

Performance
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
Intel® Nehalem™ Processor

```c
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;
}

L1 Cache Miss
L2 Cache Miss
L3 Cache Miss
TLB misses
Performance
Virtual Memory

• Virtual and real memory
  • Programmers operate as we have access to all the memory in the address space. i.e. $2^{64}$ bytes.
  • However, machines have limited real memory (ex: 64 GB is $2^{36}$)
  • Need to translate from virtual memory to physical memory
    • Pages: Granularly of the translation chunks
    • TLBs: Translation Lookaside Buffer

• Pages
  • Memory is chunked into pages, and a page is mapped from virtual to physical memory
  • Page size is historically 4K (small pages).
    • Modern machines support up to 2M, but support lacking in OSs

• TLBs
  • Hardware structure next the cache that’ll do the virtual to physical mapping
  • OS sets a table with all the translations, TLB is a cache for these entries
  • The translation table needs to be changed on context switch and flush the TLB
TLB in Intel® Nehalem™ / Linux

- **Page size is 4 KB (in Linux)**
- **I-TLB**
  - 128 entries 4-way set associative
- **D-TLB**
  - 64 entries 4-way set associative
- **Unified L2 TLB**
  - 256 entries 4 way set associative

- **On a Cache miss**
  - Even if the data is on the next level of the cache
  - Address has no TLB entry → TLB miss + cache miss
Explaining the Performance Anomaly

- **Assume working set is < 32K**
  - Fits in L1 cache → No capacity misses
- **Assume working set is > 32K but < 256K**
  - Don’t fit in L1 cache → L1 capacity miss
  - Fits in L2 cache, so no L2 capacity misses
  - 256K → 64 pages, 64 entries in D-TLB, so no TLB capacity misses
- **Assume working set is > 256K but < 1M**
  - Don’t fit in L1, L2 cache → L1, L2 capacity misses
  - Fits in L3 cache, so no L3 capacity misses
  - 1M → 256 pages, D-TLB miss, but 256 entries in L2-TLB, so no TLB capacity misses
  - Small D-TLB miss penalty
- **Assume working set is > 1M but < 8M**
  - Don’t fit in L1, L2 cache → L1, L2 capacity misses
  - Fits in L3 cache, so no L3 capacity misses
  - 8M > 256 pages, L2-TLB will incur capacity misses
  - TLB miss is a big pipeline stall and extra work
  - Need to read the page table from memory to replenish the TLB

© Saman Amarasinghe 2009
Multicores and Multiprocessors

**Multicores share resources**

**Bandwidth does not scale linearly**

2 → 1 does not mean 2x bandwidth

**A lot more bottlenecks on shared resources**

Scalability limitations

Each thread does:

for(rep=0; rep < REP; rep++)
  for(a=0; a < N ; a++)
Non Uniform Memory Access

- Each core has its own L1/L2 caches
- Each Processor shares L3 cache
- Each processor has its own DRAM
- Processors communicate via an interconnect (QPI)
- Multiple processors are connected via point-to-point links
- Different access times depending on where the memory is