Cache Memories

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS2101 October 2012

Plan

1. Hierarchical memories and their impact on our programs

2. Cache Analysis in Practice

Plan

1. Hierarchical memories and their impact on our programs

2. Cache Analysis in Practice

Capacity
Access Time
Cost

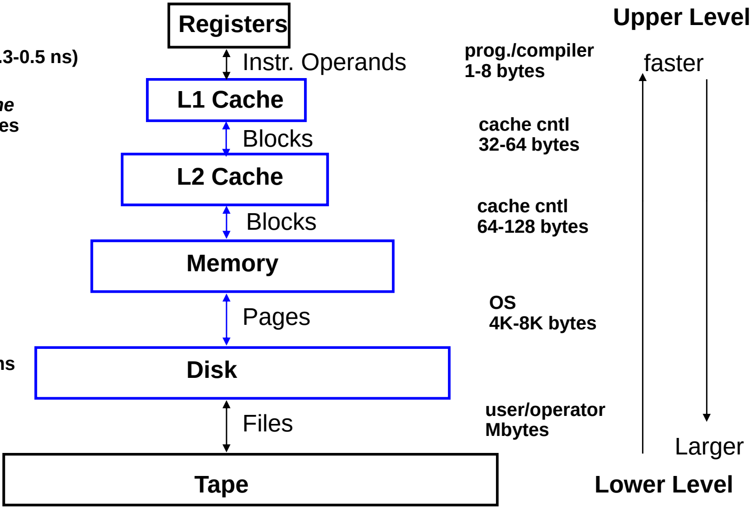
CPU Registers
100s Bytes
300 – 500 ps (0.3-0.5 ns)

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



Registers

Instr. Operands

L1 Cache

Blocks

L2 Cache

Blocks

Memory

Pages

Disk

Files

Tape

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

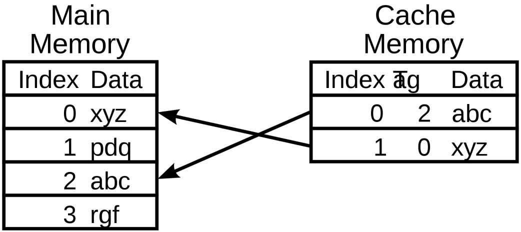
Upper Level

faster

Larger

Lower Level

CPU Cache (1/7)



Main Memory

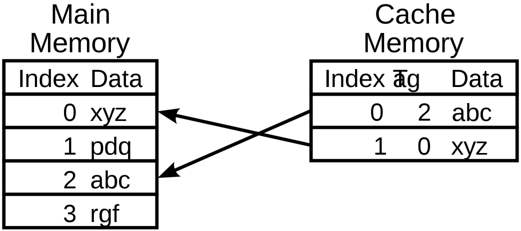
Index	Data
0	xyz
1	pdq
2	abc
3	rgf

Cache Memory

Index	Tag	Data
0	2	abc
1	0	xyz

- A CPU cache is an auxiliary memory which is smaller, faster memory than the main memory and which stores copies of the main memory locations that are expectedly frequently used.
- Most modern desktop and server CPUs have at least three independent caches: the data cache, the instruction cache and the translation look-aside buffer.

CPU Cache (2/7)



Main Memory

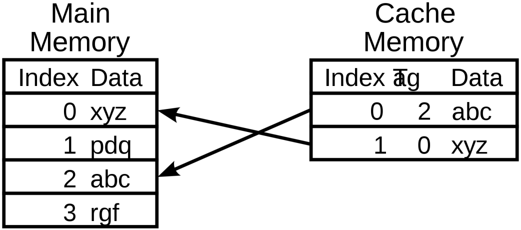
Index	Data
0	xyz
1	pdq
2	abc
3	rgf

Cache Memory

Index	Tag	Data
0	2	abc
1	0	xyz

- Each location in each memory (main or cache) has
  - a datum (cache line) which ranges between 8 and 512 bytes in size, while a datum requested by a CPU instruction ranges between 1 and 16.
  - a unique index (called address in the case of the main memory)
- In the cache, each location has also a tag (storing the address of the corresponding cached datum).

CPU Cache (3/7)



Main Memory

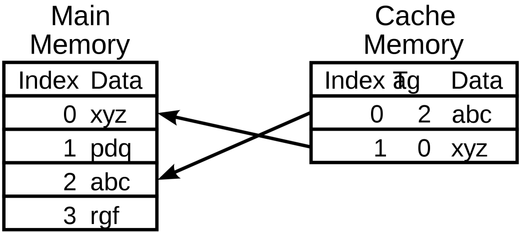
Index	Data
0	xyz
1	pdq
2	abc
3	rgf

Cache Memory

Index	Tag	Data
0	2	abc
1	0	xyz

- When the CPU needs to read or write a location, it checks the cache:
  - if it finds it there, we have a cache hit
  - if not, we have a cache miss and (in most cases) the processor needs to create a new entry in the cache.
- Making room for a new entry requires a replacement policy: the Least Recently Used (LRU) discards the least recently used items first; this requires to use age bits.

CPU Cache (4/7)



Main Memory

Index	Data
0	xyz
1	pdq
2	abc
3	rgf

Cache Memory

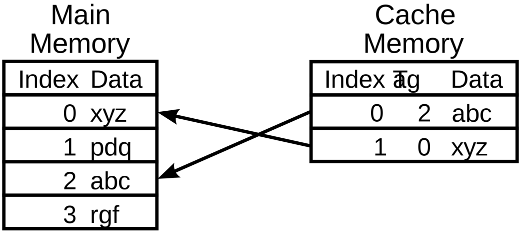
Index	Tag	Data
0	2	abc
1	0	xyz

- Read latency (time to read a datum from the main memory) requires to keep the CPU busy with something else:

  out-of-order execution: attempt to execute independent instructions arising after the instruction that is waiting due to the cache miss

  hyper-threading (HT): allows an alternate thread to use the CPU

CPU Cache (5/7)



Main Memory

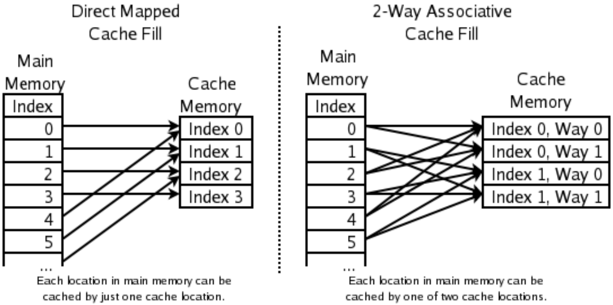
Index	Data
0	xyz
1	pdq
2	abc
3	rgf

Cache Memory

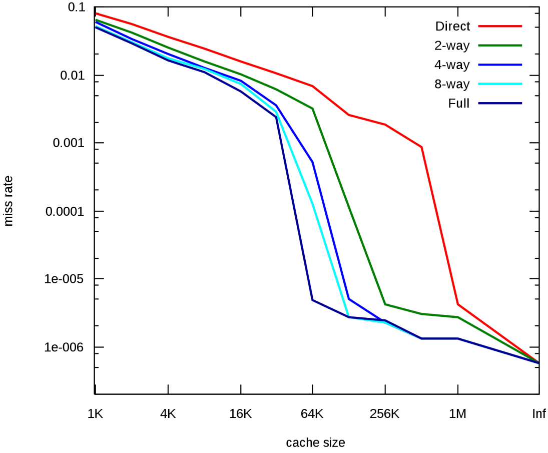
Index	Tag	Data
0	2	abc
1	0	xyz

- Modifying data in the cache requires a write policy for updating the main memory
  - write-through cache: writes are immediately mirrored to main memory
  - write-back cache: the main memory is mirrored when that data is evicted from the cache
- The cache copy may become out-of-date or stale, if other processors modify the original entry in the main memory.

CPU Cache (6/7)



- The replacement policy decides where in the cache a copy of a particular entry of main memory will go:
  - fully associative: any entry in the cache can hold it
  - direct mapped: only one possible entry in the cache can hold it
  - N-way set associative: N possible entries can hold it



- *Cache Performance for SPEC CPU2000* by J.F. Cantin and M.D. Hill.
- The SPEC CPU2000 suite is a collection of 26 compute-intensive, non-trivial programs used to evaluate the performance of a computer's CPU, memory system, and compilers (`http://www.spec.org/osg/cpu2000`).

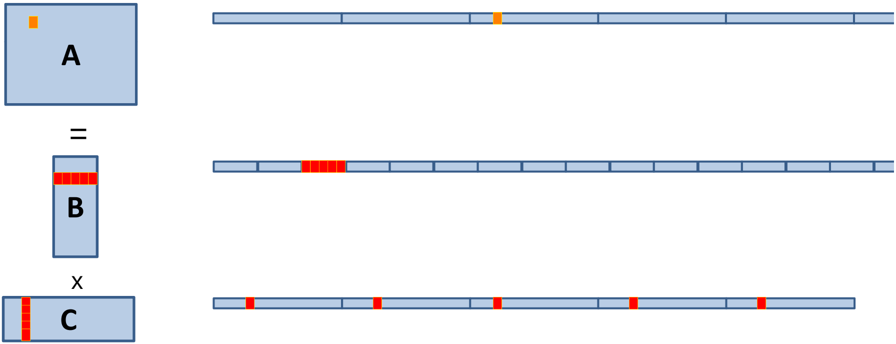
Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.
- **Conflict miss:** Multiple data items mapped to the same location with eviction before cache is full. Cure: Rearrange data and/or pad arrays.
- **True sharing miss:** Occurs when a thread in another processor wants the same data. Cure: Minimize sharing.
- **False sharing miss:** Occurs when another processor uses different data in the same cache line. Cure: Pad data.

A typical matrix multiplication C code

```
#define IND(A, x, y, d) A[(x)*(d)+(y)]
uint64_t testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C;
        long started, ended;
        float timeTaken;
        int i, j, k;
        srand(getSeed());
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
                    //  A[i][j] += B[i][k] + C[k][j];
                    IND(A,i,j,y) += IND(B,i,k,z) * IND(C,k,j,z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with matrix representation

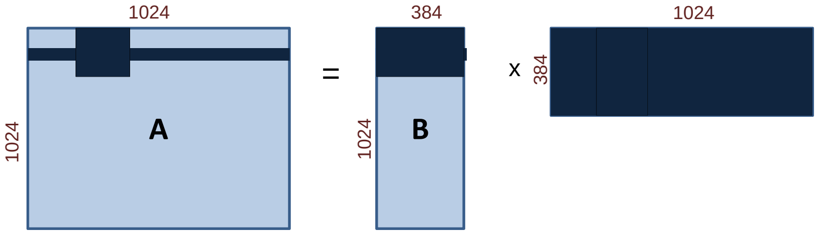


- Contiguous accesses are better:
  - Data fetch as cache line (Core 2 Duo 64 byte per cache line)
  - With contiguous data, a single cache fetch supports 8 reads of doubles.
  - Transposing the matrix `C` should reduce L1 cache misses!

Transposing for optimizing spatial locality

```
float testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C; double *Cx;
        long started, ended; float timeTaken; int i, j, k;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        Cx = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for(j =0; j < y; j++)
          for(k=0; k < z; k++)
            IND(Cx,j,k,z) = IND(C,k,j,y);
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
               IND(A, i, j, y) += IND(B, i, k, z) *IND(Cx, j, k, z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with data reuse



- Naive calculation of a row of `A`, so computing 1024 coefficients: 1024 accesses in `A`, 384 in `B` and $1024 \times 384 = 393,216$ in `C`. Total $= 394,524$.
- Computing a 32×32-block of `A`, so computing again 1024 coefficients: 1024 accesses in `A`, 384×32 in `B` and 32×384 in `C`. Total $= 25,600$.
- The iteration space is traversed so as to reduce memory accesses.

Blocking for optimizing temporal locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
                       IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,k0,j0,y);
         ended = example_get_time();
         timeTaken = (ended - started)/1.f;
   return timeTaken;
}
```

Transposing and blocking for optimizing data locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
                       IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,j0,k0,z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
        return timeTaken;
}
```

Experimental results

Computing the product of two $n \times n$ matrices on my laptop (Core2 Duo CPU P8600 @ 2.40GHz, L1 cache of 3072 KB, 4 GBytes of RAM)

n	naive	transposed	speedup	64×64-tiled	speedup	t. & t.	speedup
128	7	3		7		2	
256	26	43		155		23	
512	1805	265	6.81	1928	0.936	187	9.65
1024	24723	3730	6.62	14020	1.76	1490	16.59
2048	271446	29767	9.11	112298	2.41	11960	22.69
4096	2344594	238453	**9.83**	1009445	**2.32**	101264	**23.15**

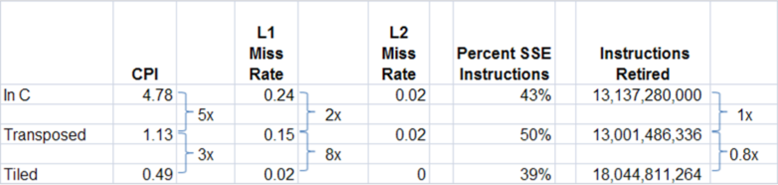
Timings are in milliseconds.

The cache-oblivious multiplication (more on this later) runs within 12978 and 106758 for $n = 2048$ and $n = 4096$ respectively.

Other performance counters

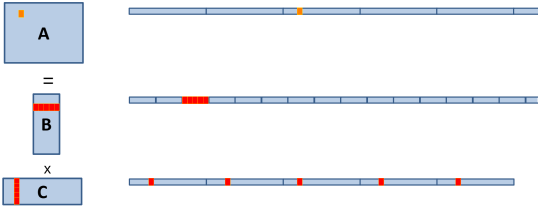
Hardware count events

- CPI Clock cycles Per Instruction: the number of clock cycles that happen when an instruction is being executed. With pipelining we can improve the CPI by exploiting instruction level parallelism
- L1 and L2 Cache Miss Rate.
- Instructions Retired: In the event of a misprediction, instructions that were scheduled to execute along the mispredicted path must be canceled.



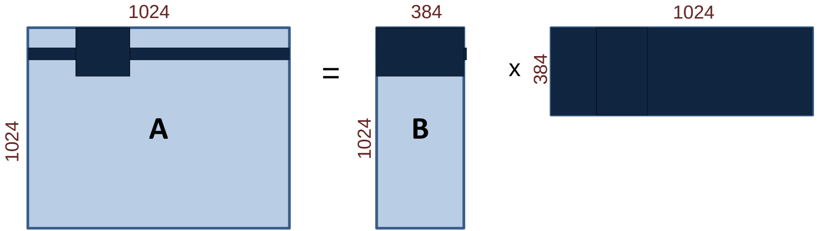
	CPI		L1 Miss Rate		L2 Miss Rate		Percent SSE Instructions		Instructions Retired	
In C	4.78		0.24		0.02		43%		13,137,280,000	
		5x		2x						1x
Transposed	1.13		0.15		0.02		50%		13,001,486,336	
		3x		8x						0.8x
Tiled	0.49		0.02		0		39%		18,044,811,264	

Analyzing cache misses in the naive and transposed multiplication



- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- A is scanned one, so mn/L cache misses if L is the number of coefficients per cache line.
- B is scanned n times, so mnp/L cache misses if the cache cannot hold a row.
- C is accessed “nearly randomly” (for m large enough) leading to mnp cache misses.
- Since $2m\,n\,p$ arithmetic operations are performed, this means roughly **one cache miss per flop!**
- If C is transposed, then the ratio improves to 1 for L.

Analyzing cache misses in the tiled multiplication



- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- Assume all tiles are square of order B and three fit in cache.
- If C is transposed, then loading three blocks in cache cost $3B^2/L$.
- This process happens n^3/B^3 times, leading to $3n^3/(BL)$ cache misses.
- Three blocks fit in cache for $3B^2 < Z$, if Z is the cache size.
- So $O(n^3/(\sqrt{Z}L))$ cache misses, if B is well chosen, which is optimal.

Counting sort: the algorithm

- *Counting sort* takes as input a collection of n items, each of which known by a key in the range $0 \cdots k$.
- The algorithm computes a *histogram* of the number of times each key occurs.
- Then performs a *prefix sum* to compute positions in the output.

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

Counting sort: cache complexity analysis

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

1. n/L to compute k.
2. k/L cache misses to initialize Count.
3. $n/L + n$ cache misses for the histogram (worst case).
4. k/L cache misses for the prefix sum.
5. $n/L + n + n$ cache misses for building Output (worst case).

Total: $3n + 3n/L + 2k/L$ cache misses (worst case).

Counting sort: cache complexity analysis: explanations

1. n/L to compute k: this can be done by traversing the `items` linearly.
2. k/L cache misses to initialize `Count`: this can be done by traversing the `Count` linearly.
3. $n/L + n$ cache misses for the histogram (worst case): `items` accesses are linear but `Count` accesses are potentially random.
4. k/L cache misses for the prefix sum: `Count` accesses are linear.
5. $n/L + n + n$ cache misses for building `Output` (worst case): `items` accesses are linear but `Output` and `Count` accesses are potentially random.

   Total: $3n$+$3n/L + 2k/L$ cache misses (worst case).

How to fix the poor data locality of counting sort?

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- Recall that our worst case is $3n+3n/L+2k/L$ cache misses.
- The troubles come from the irregular which experience capacity misses and conflict misses.
- To solve this problem, we preprocess the input so that counting sort is applied in succession to several smaller input item sets with smaller value ranges.
- To put it simply, so that k and n are small enough for `Output` and

Counting sort: bukecting the input

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

- Goal: after preprocessing, `Count` and `Output` incur **cold misses only**.
- To this end we choose a parameter m (more on this later) such that
  1. a key in the range $[ih, (i+1)h-1]$ is always before a key in the range $[(i+1)h, (i+2)h-1]$, for $i = 0 \cdots m-2$, with $h = k/m$,
  2. `bucketsize` and m cache-lines from `bucketedinput` all fit in cache. That is, counting cache-lines, $m/L + m \leq Z/L$, that is, $m + m/L \leq Z$.

Counting sort: cache complexity with bukecting

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

1. $3m/L + n/L$ caches misses to compute `bucketsize`
2. **Key observation:** `bucketedinput` is traversed regularly by segment.
3. Hence, $2n/L + m + m/L$ caches misses to compute `bucketedinput`

   Preprocessing: $3n/L + 3m/L + m$ cache misses.

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute bucketsize:
   - m/L to set each cell of bucketsize to zero,
   - $m/L + n/L$ for the first for loop,
   - m/L for the second for loop.
2. **Key observation:** bucketedinput is traversed regularly by segment:
   - So writing bucketedinput means writing (in a linear traversal) m consecutive arrays, of possibly different sizes, but with total size n.
   - Thus, because of possible misalignments between those arrays and their cache-lines, this writing procedure can yield $n/L + m$ cache misses (and not just n/L).
3. Hence, $2n/L + m + m/L$ caches misses to compute bucketedinput:
   - n/L to read the items,
   - $n/L + m$ to write bucketedinput,
   - m/L to load bucketsize.

Cache friendly counting sort: complete cache complexity analysis

- **Assumption:** the preprocessing creates buckets of average size n/m.
- After preprocessing, counting sort is applied to each bucket whose values are in a range $[ih, (i+1)h-1]$, for $i = 0 \cdots m-1$.
- To be cache-friendly, this requires, for $i = 0 \cdots m-1$, $h + |\{\texttt{key} \in [ih, (i+1)h-1]\}| < Z$ and $m < Z/(1+L)$. These two are very realistic assumption considering today's cache size.
- And the total complexity becomes;

$$\begin{array}{rcl} Q_{\rm total} & = & Q_{\rm preprocessing} + Q_{\rm sorting} \\ & = & Q_{\rm preprocessing} + m\,Q_{\rm sortingofonebucket} \\ & = & Q_{\rm preprocessing} + m\,(3\frac{n}{m\,\mathbf{L}} + 3\frac{n}{m} + 2\frac{k}{mL}) \\ & = & Q_{\rm preprocessing} + 6n/L + 2k/L \\ & = & 3n/L + 3m/L + m + 6n/L + 2k/L \\ & = & 9n/L + 3m/L + m + 2k/L \end{array}$$

Instead of $3n + 3n/L + 2k/L$ for the naive counting sort.

Cache friendly counting sort: experimental results

- Experimentation on an *Intel(R) Core(TM) i7 CPU @ 2.93GHz*. It has L2 cache of 8MB.
- CPU times in seconds for both classical and cache-friendly counting sort algorithm.
- The keys are random machine integers in the range $[0, n]$.

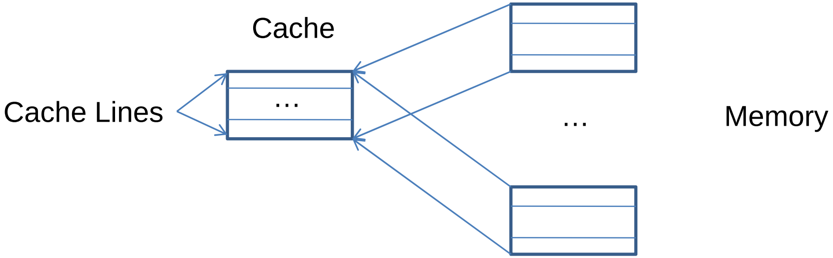
n	classical counting sort	cache-oblivious counting sort (preprocessing + sorting)
100000000	13.74	4.66 (3.04 + 1.62)
200000000	30.20	9.93 (6.16 + 3.77)
300000000	50.19	16.02 (9.32 + 6.70)
400000000	71.55	22.13 (12.50 +9.63)
500000000	94.32	28.37 (15.71 + 12.66)
600000000	116.74	34.61 (18.95 + 15.66)

Plan

1. Hierarchical memories and their impact on our programs

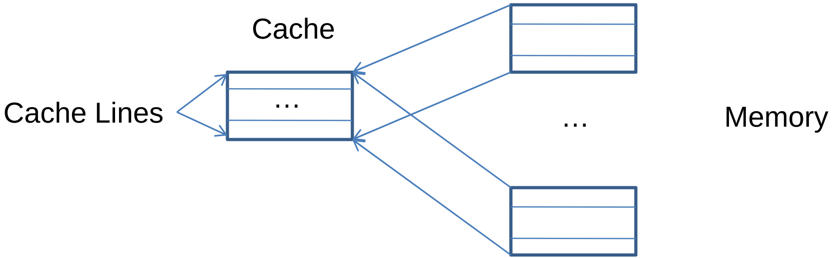
2. Cache Analysis in Practice

Basic idea of a cache memory (review)



- A cache is a smaller memory, faster to access
- Using smaller memory to cache contents of larger memory provides the illusion of fast larger memory
- Key reason why this works: **temporal locality** and **spatial locality**.

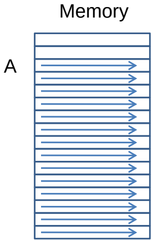
A simple cache example



- Byte addressable memory
- Size of 32Kbyte with direct mapping and 64 byte lines (512 lines) so the cache can fit $2^9 \times 2^4 = 2^{13}$ `int`.
- A cache access costs 1 cycle while a memory access costs 100 cycles.
- How addresses map into cache
    - Bottom 6 bits are used as offset in a cache line,
    - Next 9 bits determine the cache line

Exercise 1 (1/2)

```
// sizeof(int) = 4 and Array laid out sequentially in memory
#define S ((1<<20)*sizeof(int))
int A[S];
// Thus size of A is 2^(20) x 4 bytes
for (i = 0; i < S; i++) {
        read A[i];
}
```



Total access time? What kind of locality? What kind of misses?

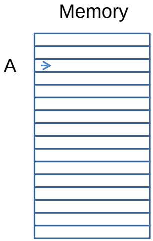
Exercise 1 (2/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[i];
}
```

- S reads to A.
- 16 elements of A per cache line
- 15 of every 16 hit in cache.
- Total access time: $15(S/16) + 100(S/16)$.
- spatial locality, cold misses.

Exercise 2 (1/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[0];
}
```



Total access time? What kind of locality? What kind of misses?

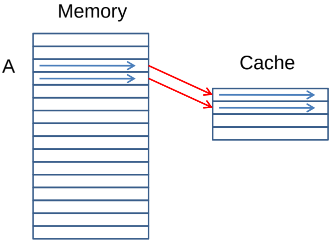
Exercise 2 (2/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[0];
}
```

- S reads to A
- All except the first one hit in cache.
- Total access time: $100 + (S - 1)$.
- Temporal locality
- Cold misses.

Exercise 3 (1/2)

```
// Assume 4 <= N <= 13
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[i % (1<<N)];
}
```



Total access time? What kind of locality? What kind of misses?

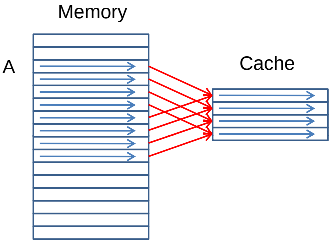
Exercise 3 (2/2)

```
// Assume 4 <= N <= 13
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
         read A[i % (1<<N)];
}
```

- `S` reads to `A`
- One miss for each accessed line, rest hit in cache.
- Number of accessed lines: 2^{N-4}.
- Total access time: $2^{N-4}100 + (S - 2^{N-4})$.
- Temporal and spatial locality
- Cold misses.

Exercise 4 (1/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
 read A[i % (1<<N)];
}
```



Total access time? What kind of locality? What kind of misses?

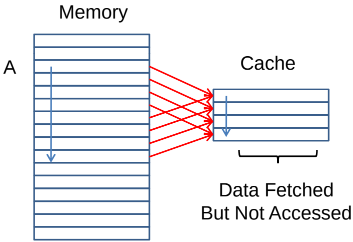
Exercise 4 (2/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
 read A[i % (1<<N)];
}
```

- S reads to A.
- First access to each line misses
- Rest accesses to that line hit.
- Total access time: $15(S/16) + 100(S/16)$.
- Spatial locality
- Cold and capacity misses.

Exercise 5 (1/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
read A[(i*16) % (1<<N)];
}
```



Total access time? What kind of locality? What kind of misses?

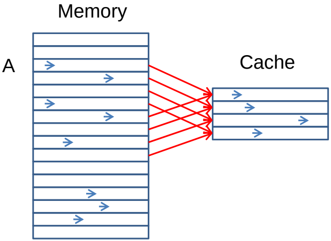
Exercise 5 (2/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
read A[(i*16) % (1<<N)];
}
```

- `S` reads to `A`.
- First access to each line misses
- One access per line.
- Total access time: $100S$.
- No locality!
- Cold and conflict misses.

Exercise 6 (1/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[random()%S];
}
```



Total access time? What kind of locality? What kind of misses?

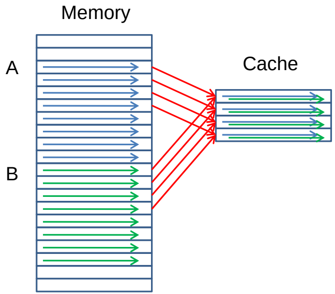
Exercise 6 (2/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[random()%S];
}
```

- S reads to A.
- After N iterations, for some N, the cache is full and holds 2^9 cache lines from S.
- S consists of $2^{20-4} = 2^{16}$ cache lines.
- Them the chance of hitting in cache is $2^9/2^{16} = 1/128$
- Estimated total access time: $S\,((127/128)100 + (1/128))$.
- Almost no locality!
- Cold, capacity conflict misses.

Exercise 7 (1/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```



Total access time? What kind of locality? What kind of misses?

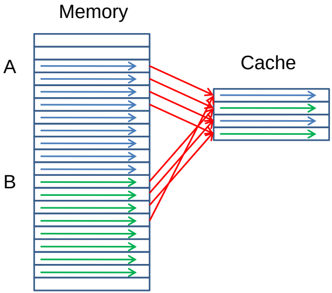
Exercise 7 (2/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```

- S reads to A and B.
- A and B interfere in cache: indeed two cache lines whose addresses differ by a multiple of 2^9 have the *same way to cache*.
- Total access time: $2 \times 100 \times S$.
- Spatial locality but the cache cannot exploit it.
- Cold and conflict misses.

Exercise 8 (1/2)

```
#define S ((1<<19+4)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```



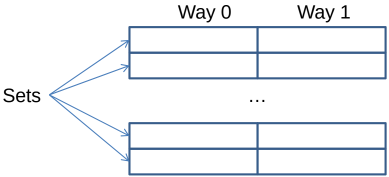
Total access time? What kind of locality? What kind of misses?

Exercise 8 (2/2)

```
#define S ((1<<19+4)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```

- S reads to A and B.
- A and B almost do not interfere in cache.
- Total access time: $2(15S/16 + 100S/16)$.
- Spatial locality.
- Cold misses.

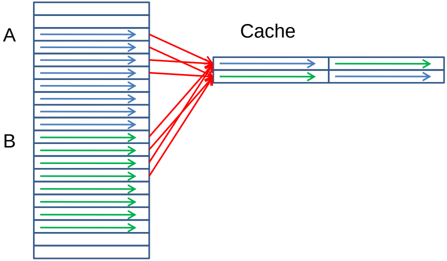
Set Associative Caches



- **Set associative caches** have sets with multiple lines per set.
- Each line in a set is called a way
- Each memory line maps to a specific set and can be put into any cache line in its set
- In our example, we assume a 32 Kbyte cache, with 64 byte lines, 2-way associative. Hence we have:
    - 256 sets
    - Bottom six bits determine offset in cache line
    - Next 8 bits determine the set.

Exercise 9 (1/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```



Total access time? What kind of locality? What kind of misses?

Exercise 9 (2/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```

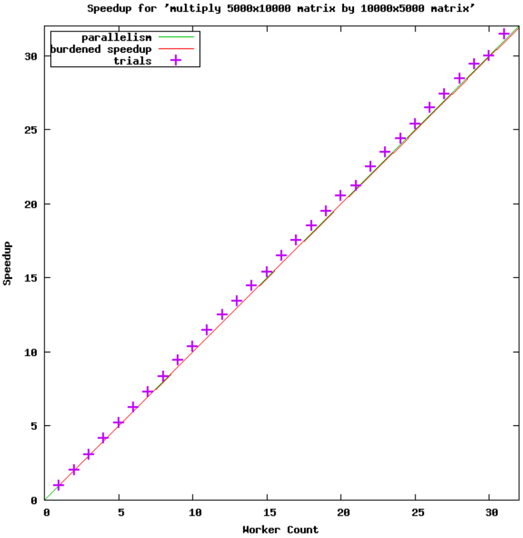
- S reads to A and B.
- A and B lines hit same set, but enough lines in a set.
- Total access time: $2(15S/16 + 100S/16)$.
- Spatial locality.
- Cold misses.

Tuned cache-oblivious matrix transposition benchmarks

size	Naive	Cache-oblivious	ratio
5000x5000	126	79	1.59
10000x10000	627	311	2.02
20000x20000	4373	1244	3.52
30000x30000	23603	2734	8.63
40000x40000	62432	4963	12.58

- `Intel(R) Xeon(R) CPU E7340 @ 2.40GHz`
- L1 data 32 KB, L2 4096 KB, cache line size 64bytes
- **Both codes run on 1 core**
- The ration comes simply from an **optimal memory access pattern.**

Tuned cache-oblivious parallel matrix multiplication



Acknowledgments and references

Acknowledgments.

- Charles E. Leiserson (MIT) and Matteo Frigo (Intel) for providing me with the sources of their article *Cache-Oblivious Algorithms*.
- Charles E. Leiserson (MIT) and Saman P. Amarasinghe (MIT) for sharing with me the sources of their course notes and other documents.

References.

- *Cache-Oblivious Algorithms* by Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran.
- *Cache-Oblivious Algorithms and Data Structures* by Erik D. Demaine.