6.172 PERFORMANCE ENGINEERING OF SOFTWARE SYSTEMS

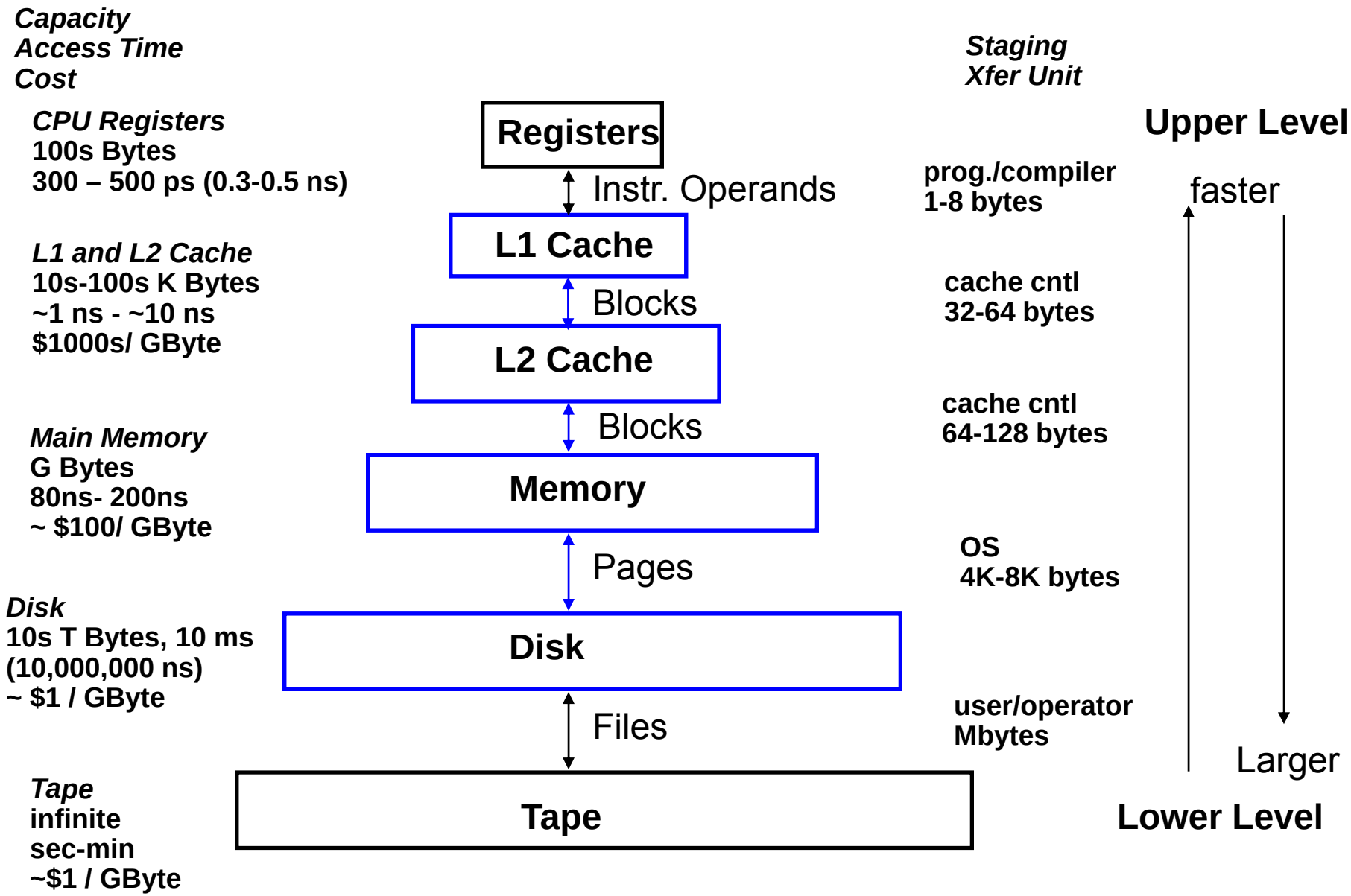
Memory Systems and Performance Engineering

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



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

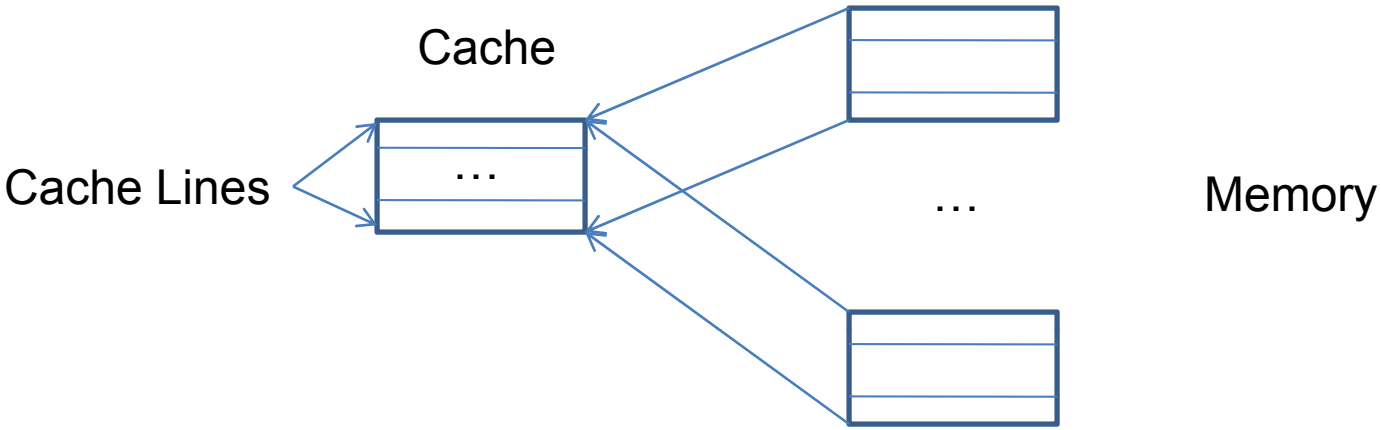
C. Memory access = 100 cycles

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

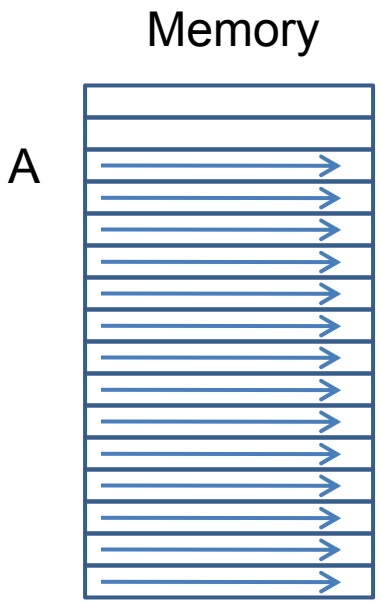


Analytically Model Access Patterns

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

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

Array laid out sequentially in memory



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

What kind of locality?

Spatial

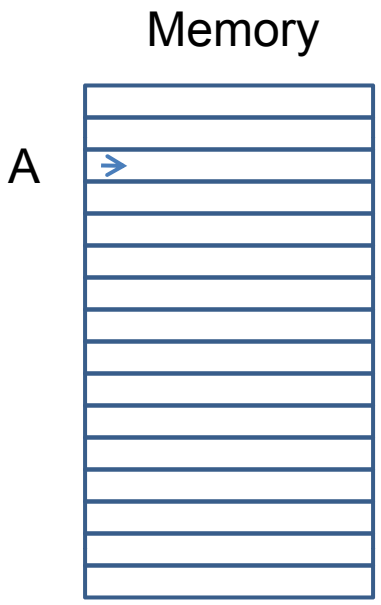
What kind of misses?

Cold

Analytically Model Access Patterns

```
#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

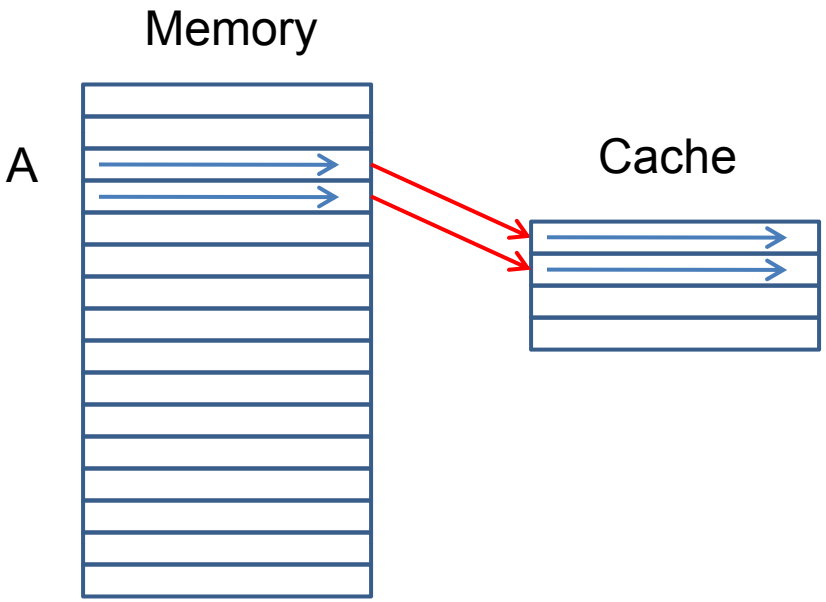
What kind of misses?

Cold

Analytically Model Access Patterns

```
#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

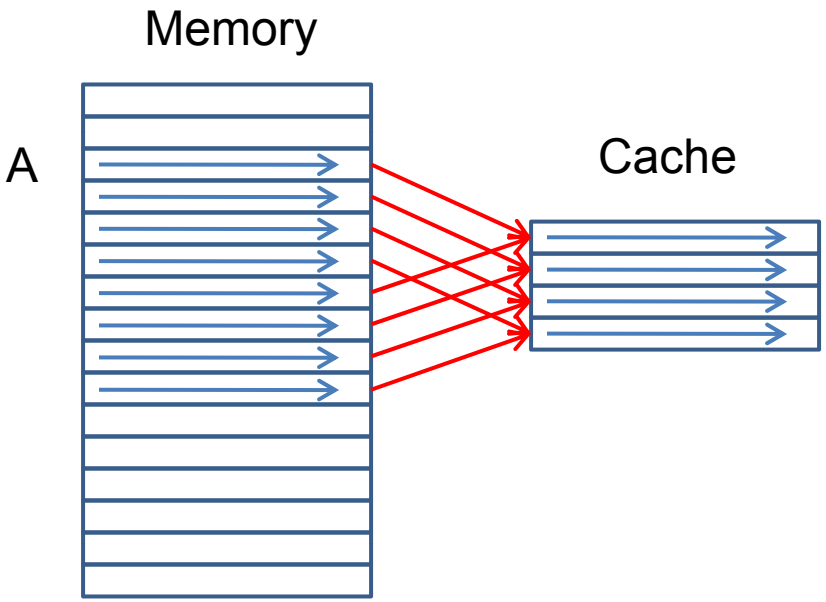
What kind of misses?

Cold

Analytically Model Access Patterns

```
#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

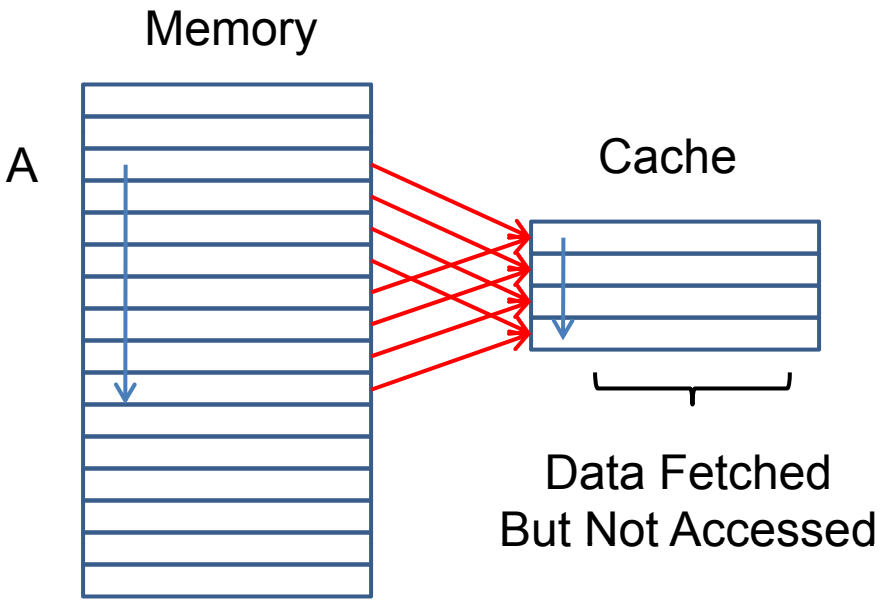
What kind of misses?

Cold, capacity

Analytically Model Access Patterns

```
#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 16th 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

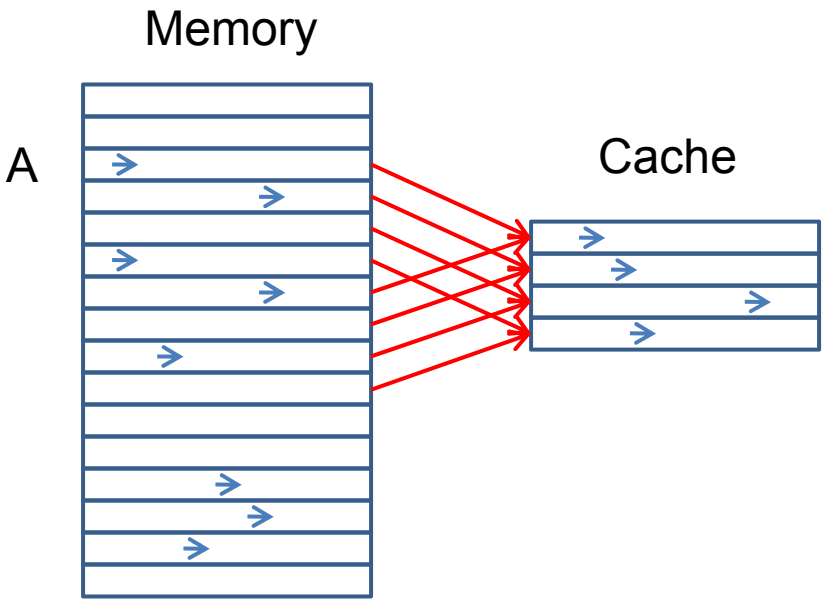
What kind of misses?

Cold, conflict

Analytically Model Access Patterns

```
#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

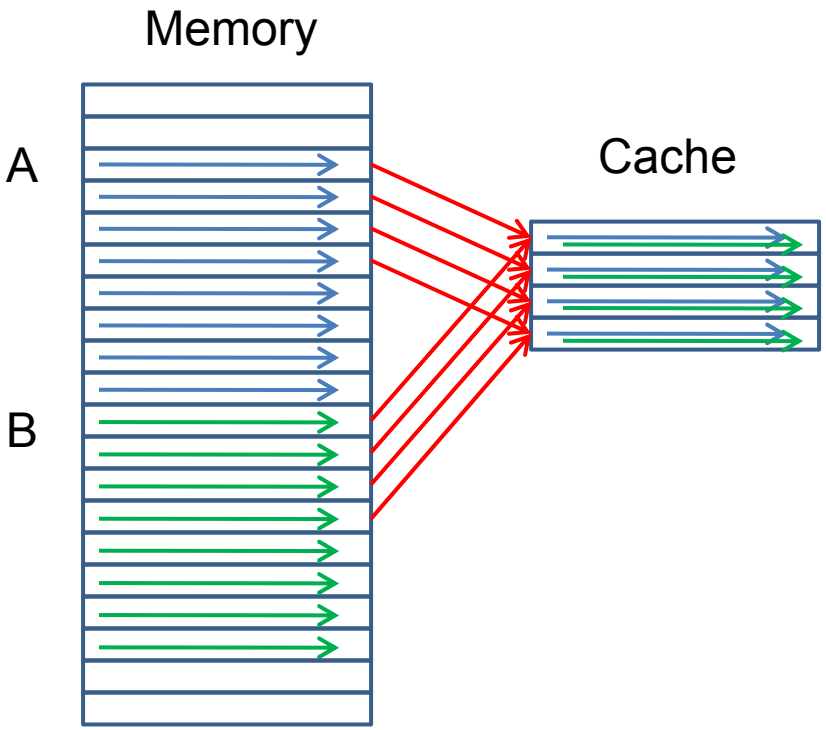
Basic Cache Access Modes

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)

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…

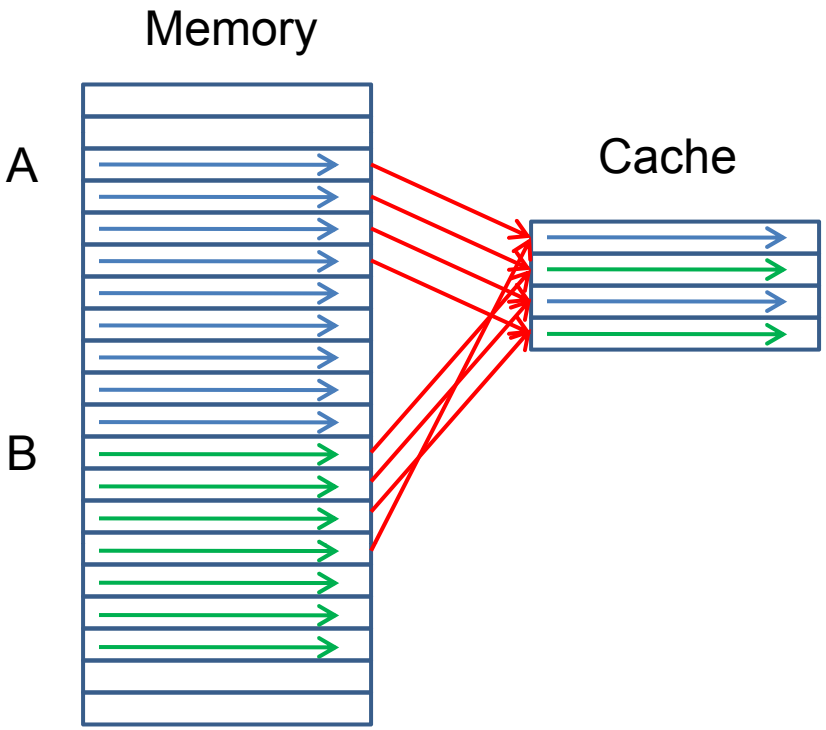
What kind of misses?

Cold, Conflict

Analytically Model Access Patterns

```
#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(15/16*S + 1/6*S*100)

What kind of locality?

Spatial locality

What kind of misses?

Cold

Set Associative Caches

A. Have sets with multiple lines per set

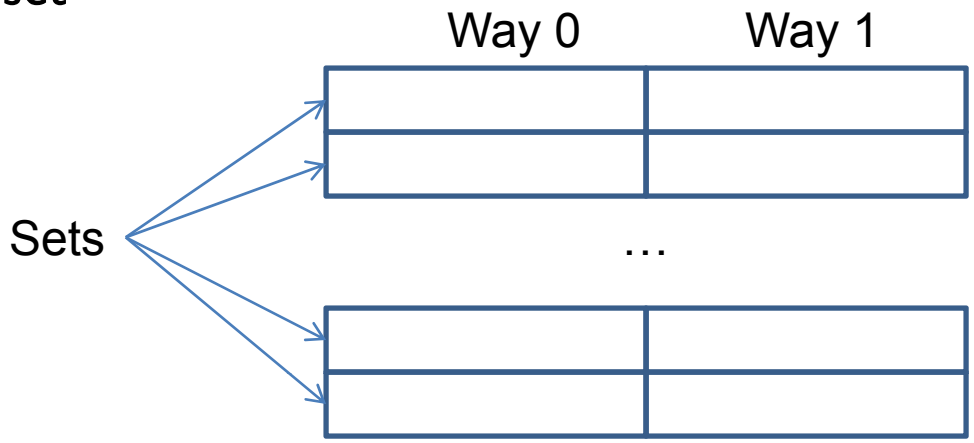
B. Each line in cache called a way

C. Each memory¡¡¨ne 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

```
#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

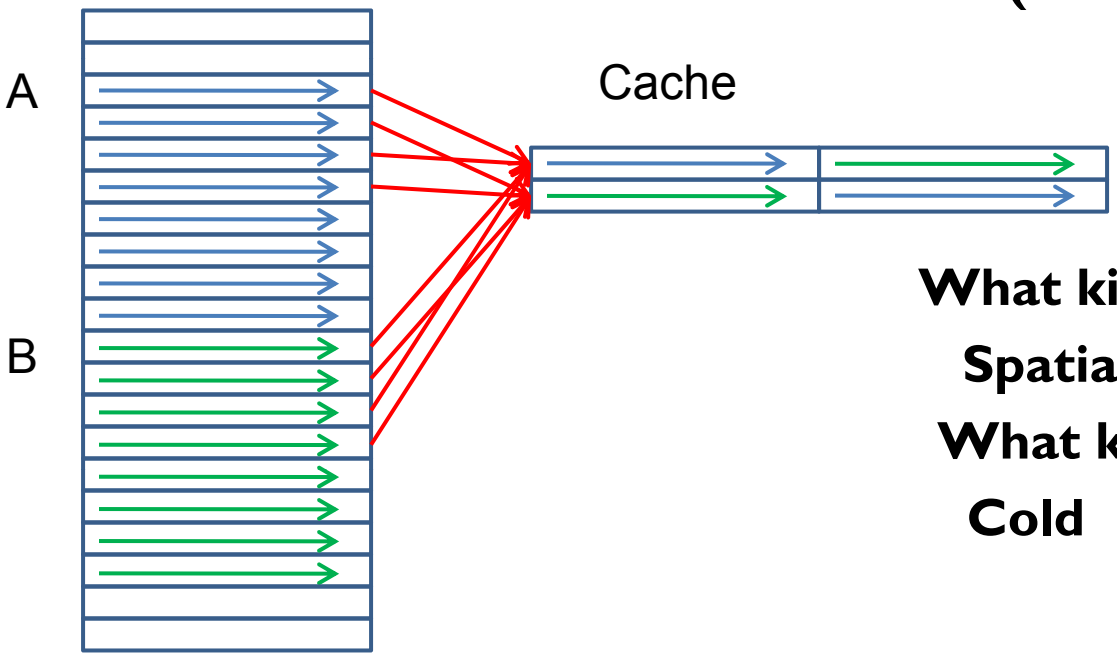
Read A and B sequentially

S reads to A, B

A and B lines hit same way, but enough lines in way

Total access time

2*(15/16*S + 1/6*S*100)



What kind of locality?

Spatial locality

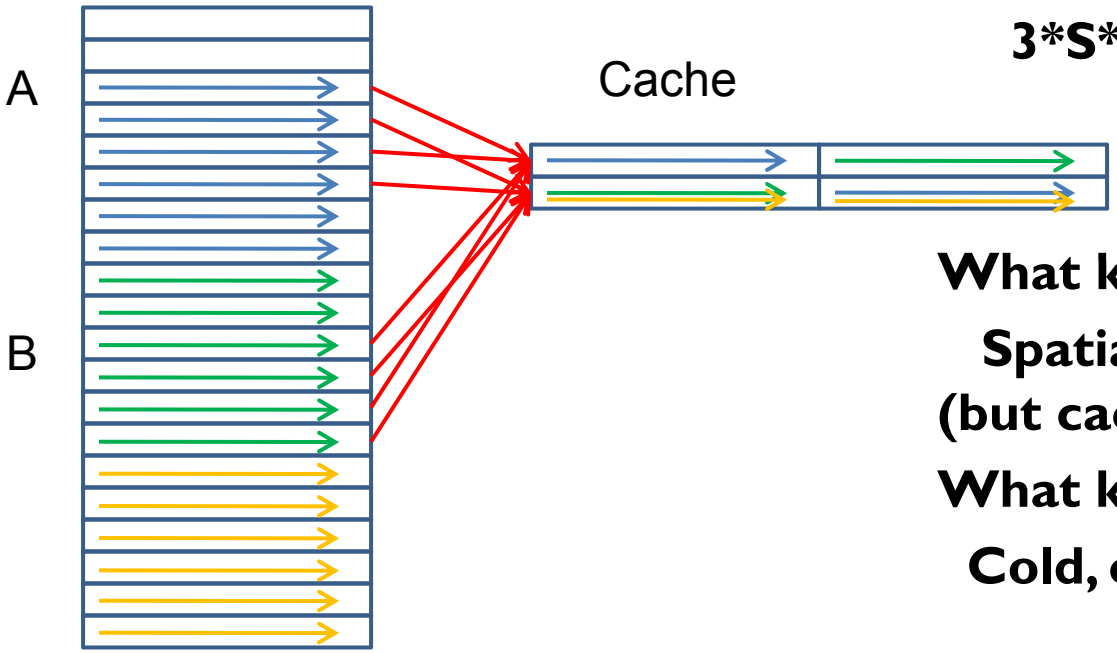
What kind of misses?

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

```
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

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

sizeof(struct node) = 32

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

```
struct node {
 int data;
 int more_data;
 int even_more_data;
 int yet_more_data;
 int flags;
 struct node *next;
};
for (c = l; 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;

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]
```

A_{00} A_{01} A_{02} A_{03} A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23} A_{30} A_{31} A_{32} A_{33}

C. What if you want column-major storage?

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

A_{00} A_{10} A_{20} A_{30} A_{01} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{32} A_{03} A_{13} A_{23} A_{33}

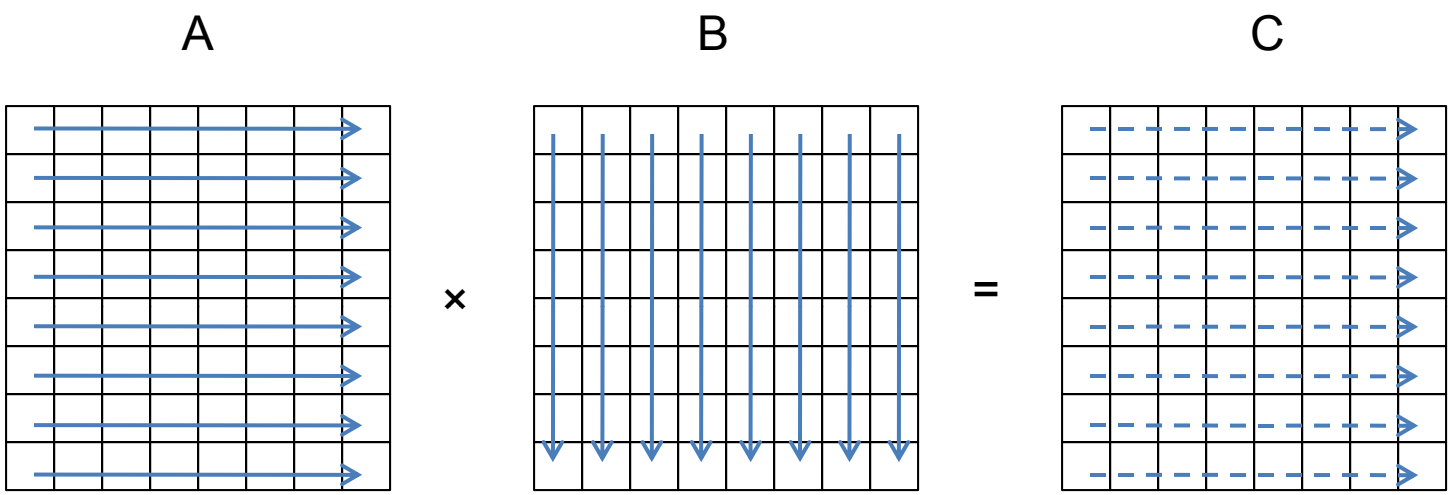
Standard Matrix Multiply Code

```
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**

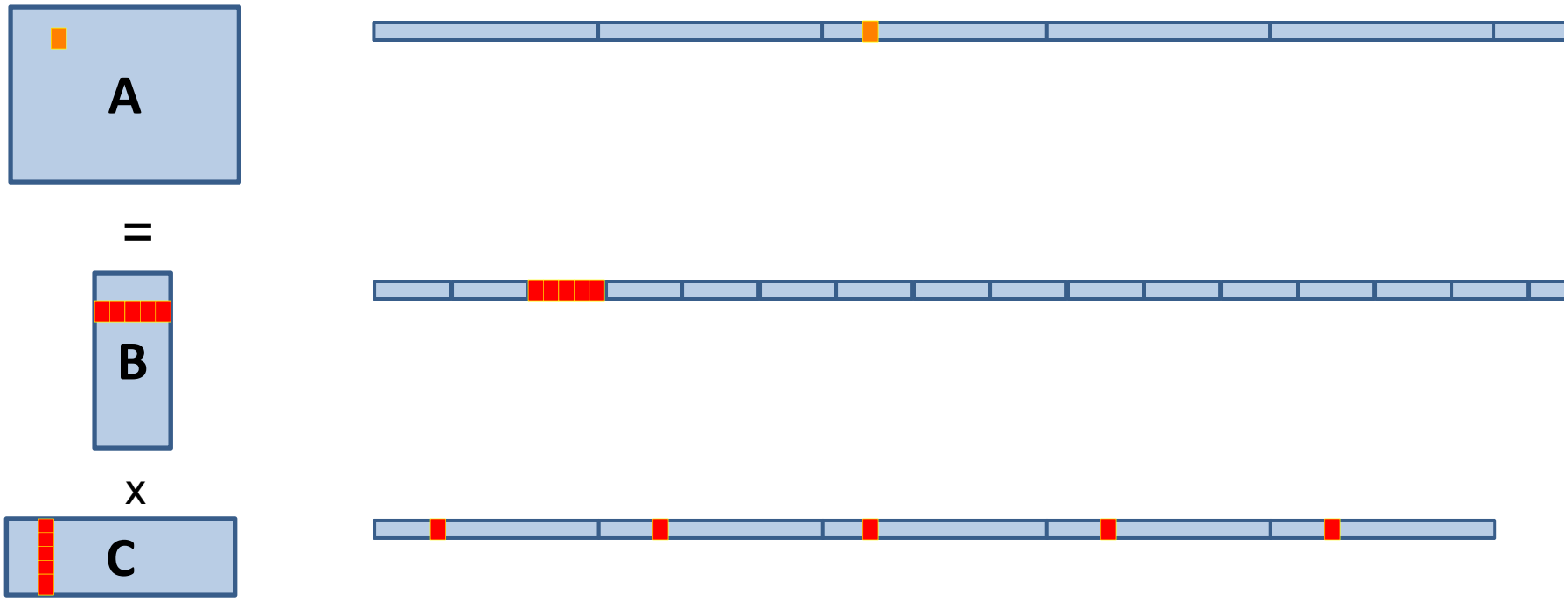
Access Patterns for A, B, and C



Courtesy of Martin Rinard. Used with permission.

Memory Access Pattern

Scanning the memory



How to get spatial locality for B

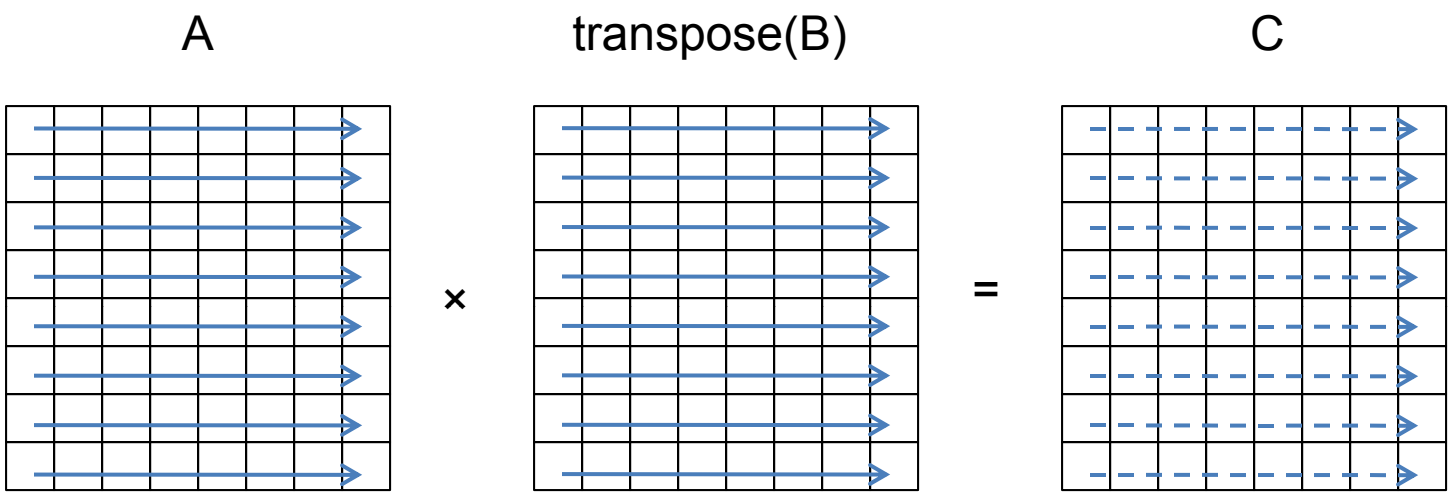
A. Transpose B first, then multiply. New code:

```
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[j*SIZE+k];
      }
    }
  }
```

Overall effect on execution time?

11620 ms (original)

2356 ms (after transpose)



Courtesy of Martin Rinard. Used with permission.

Profile Data

	CPI		L1 Miss Rate		L2 Miss Rate		Percent SSE Instructions		Instructions Retired	
In C	4.78	5x	0.24	2x	0.02		43%		13,137,280,000	1x
Transposed	1.13		0.15		0.02		50%		13,001,486,336	

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)**

$$\begin{pmatrix} A_{11} \cdots A_{1N} \\ \cdots \\ A_{N1} \cdots A_{NN} \end{pmatrix} \times \begin{pmatrix} B_{11} \cdots B_{1N} \\ \cdots \\ B_{N1} \cdots B_{NN} \end{pmatrix} = \begin{pmatrix} \sum_k A_{1k} * B_{k1} \cdots \sum_k A_{1k} * B_{kN} \\ \cdots \\ \sum_k A_{Nk} * B_{k1} \cdots \sum_k A_{Nk} * B_{kN} \end{pmatrix}$$

Blocked Matrix Multiply

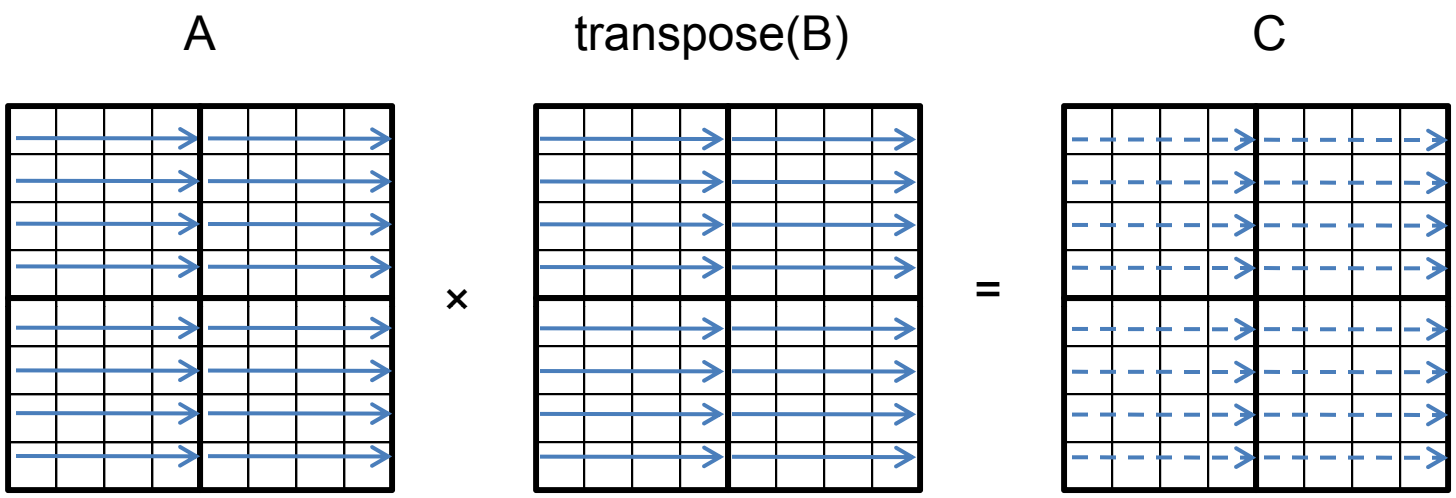
```
for (j = 0; j < SIZE; j += BLOCK) {
  for (k = 0; k < SIZE; k += BLOCK) {
    for (i = 0; i < SIZE; i+=BLOCK) {
      for (ii = i; ii < i+BLOCK; ii++) {
        for (jj = j; jj < j+BLOCK; jj++) {
          for (kk = k; kk < k+BLOCK; kk++) {
              C[ii*SIZE+jj] += A[ii*SIZE+kk]*B[jj*SIZE+kk];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

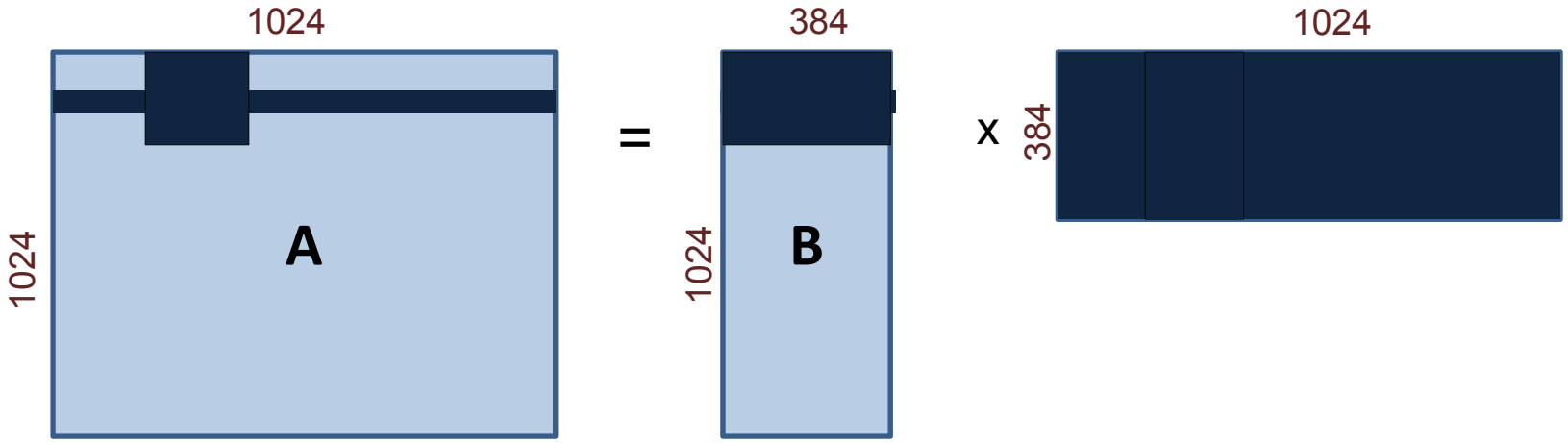


Courtesy of Martin Rinard. Used with permission.

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*1=1024 + B: 384*1=394 + C: 1024*384=393,216 = 394,524
- Blocked Matrix Multiply (32^2 = 1024 values of A)
  - A: 32*32=1024 + B: 384*32 =12,284 + C: 32*384=12,284 = 25,600



Profile Data

	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	

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)

Call Stack and Locality

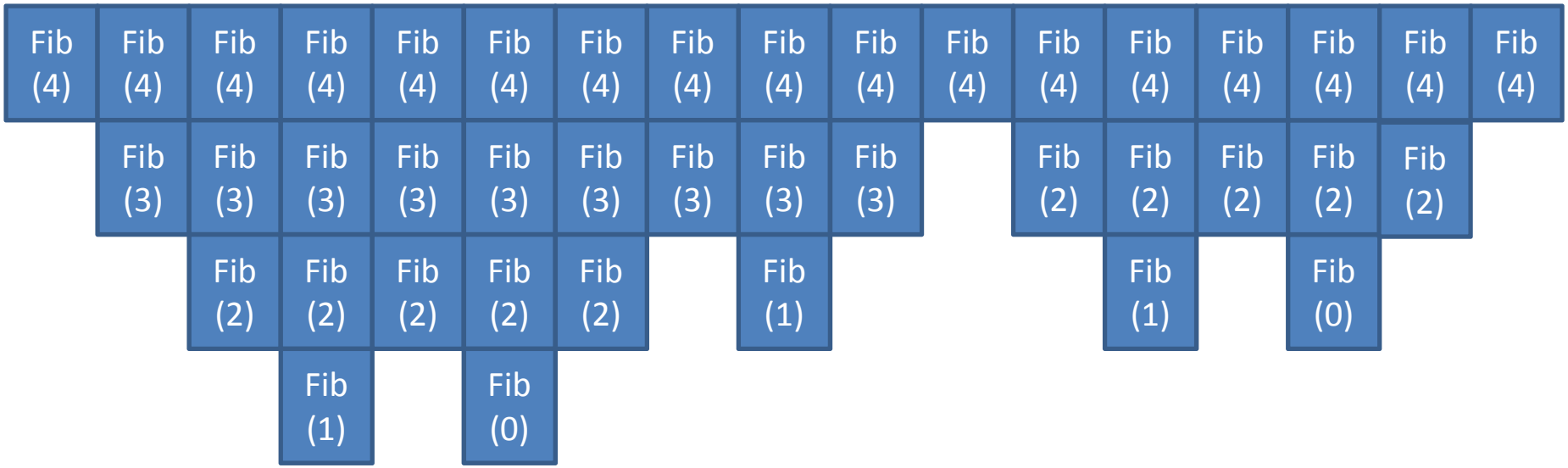
```
int fib(int n) {
  if (n == 0) return 1;
  if (n == 1) return 1;
  return (fib(n-1) + fib(n-2));
}
```

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

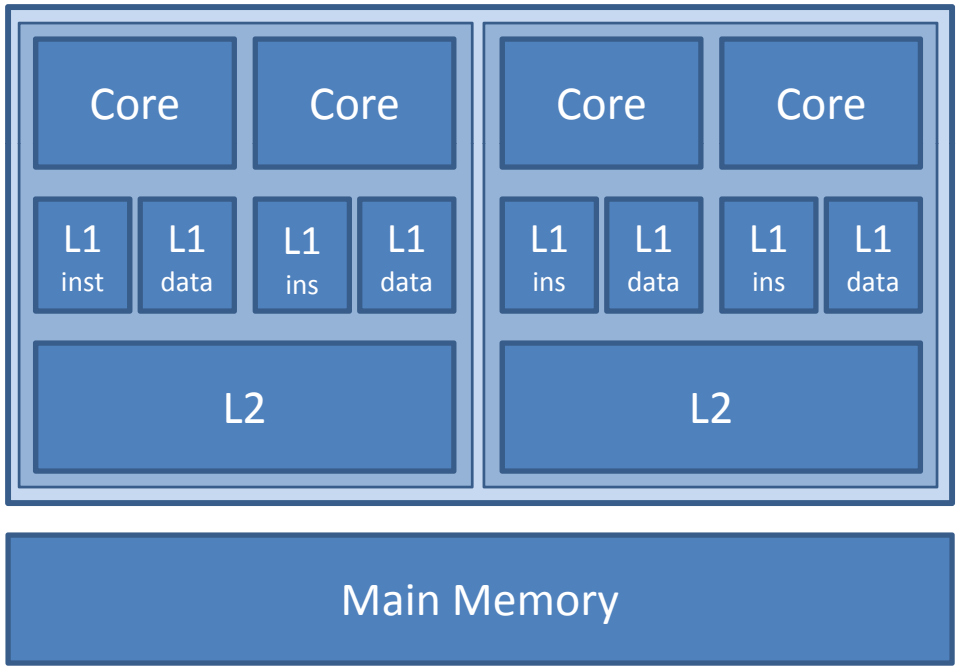
1. Segregate and pack frequently accessed data
    a. Replace structs with parallel arrays, other split data structures
    b. Pack data to get smaller cache footprint
2. Modify representation; Compute; Restore representation
    a. Transpose; Multiply; Transpose
    b. Copy In; Compute; Copy Out

D. Computation transformations

1. Reorder data accesses for reuse
2. Recast computation stages when possible

Intel® Core™ Microarchitecture – Memory Sub-system

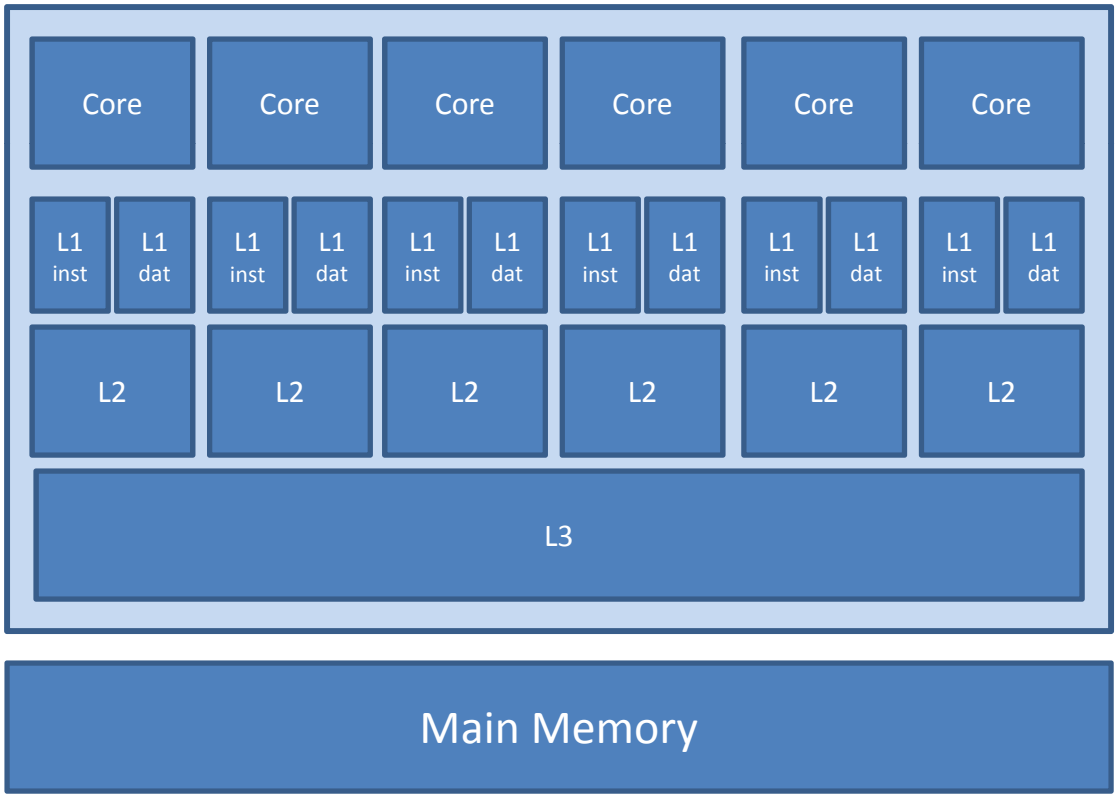
Intel Core 2 Quad Processor



L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L2 Cache			
Size	Line Size	Latency	Associativty
6 MB	64 bytes	14 cycles	24-way

Intel® Nehalem™ Microarchitecture – Memory Sub-system

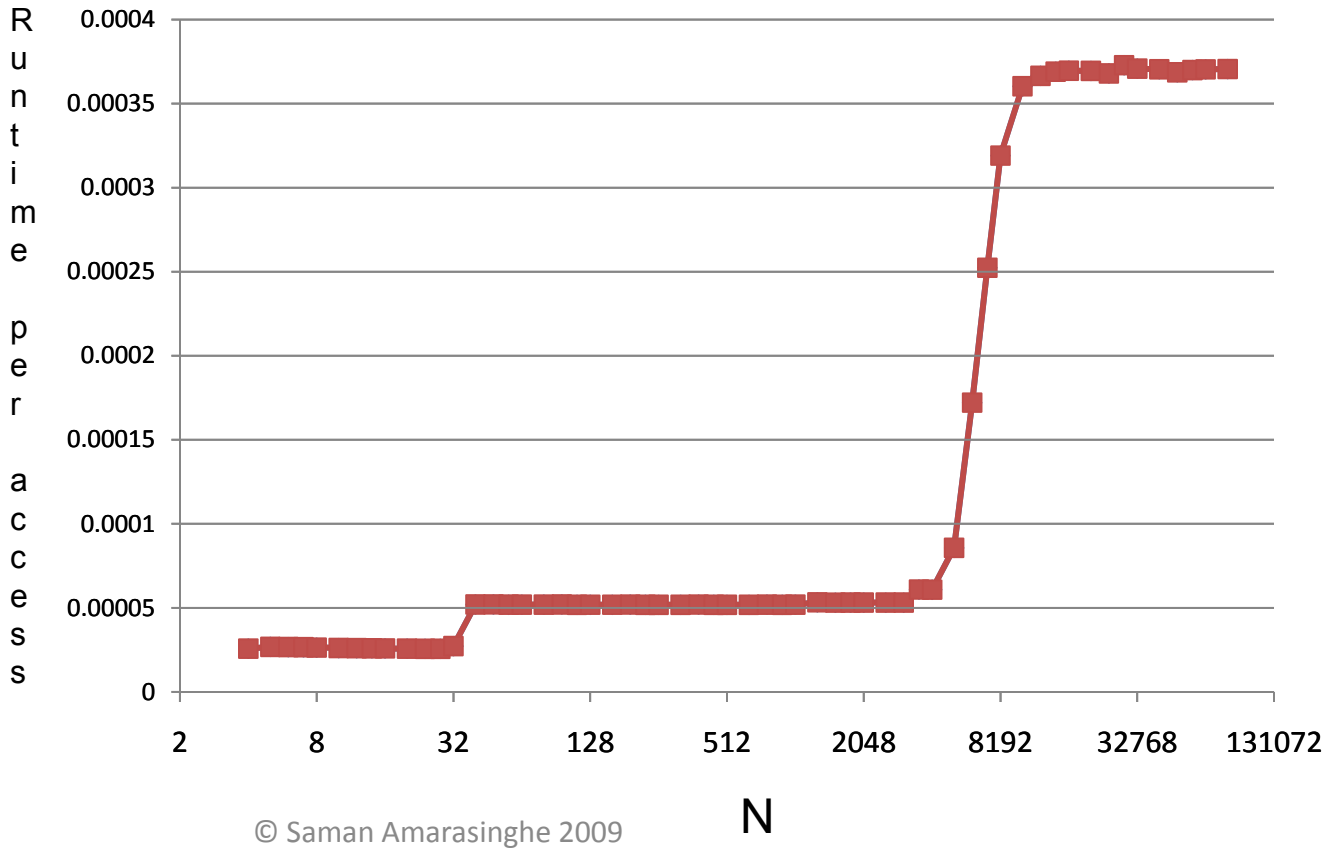
Intel 6 Core Processor



L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	4 ns	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	4 ns	4-way
L2 Cache			
Size	Line Size	Latency	Associativty
256 KB	64 bytes	10 ns	8-way
L3 Cache			
Size	Line Size	Latency	Associativty
12 MB	64 bytes	50 ns	16-way
Main Memory			
Size	Line Size	Latency	Associativty
	64 bytes	75 ns	

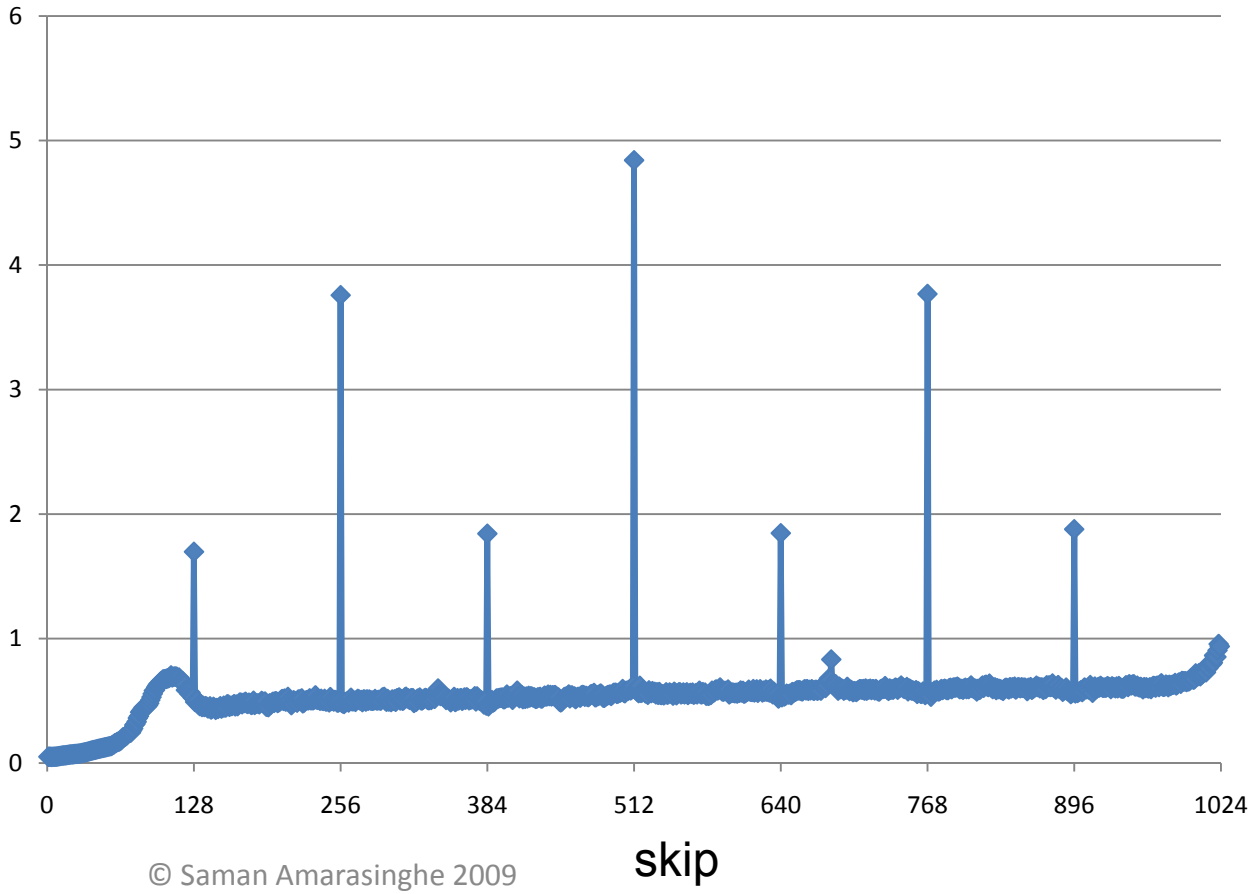
Intel Core 2 Quad Processor

```
for(rep=0; rep < REP; rep++)
    for(a=0; a <  N ; a++)
        A[a] = A[a] + 1;
```



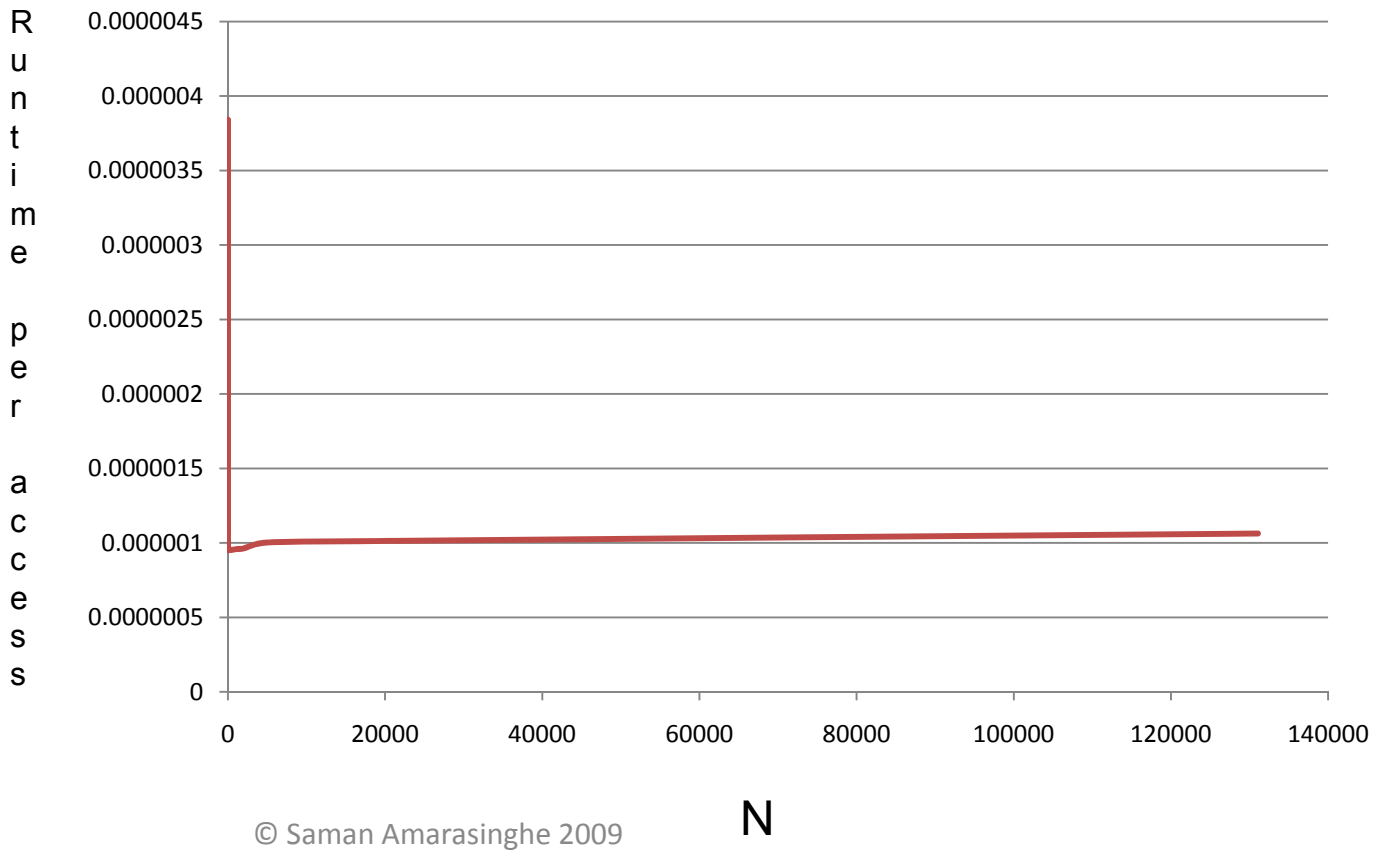
Intel Core 2 Quad Processor

```
for(r=0; r < REP; r++)
    for(a=0; a <64*1024; a++)
        A[a*skip] = A[a*skip] + 1;
```



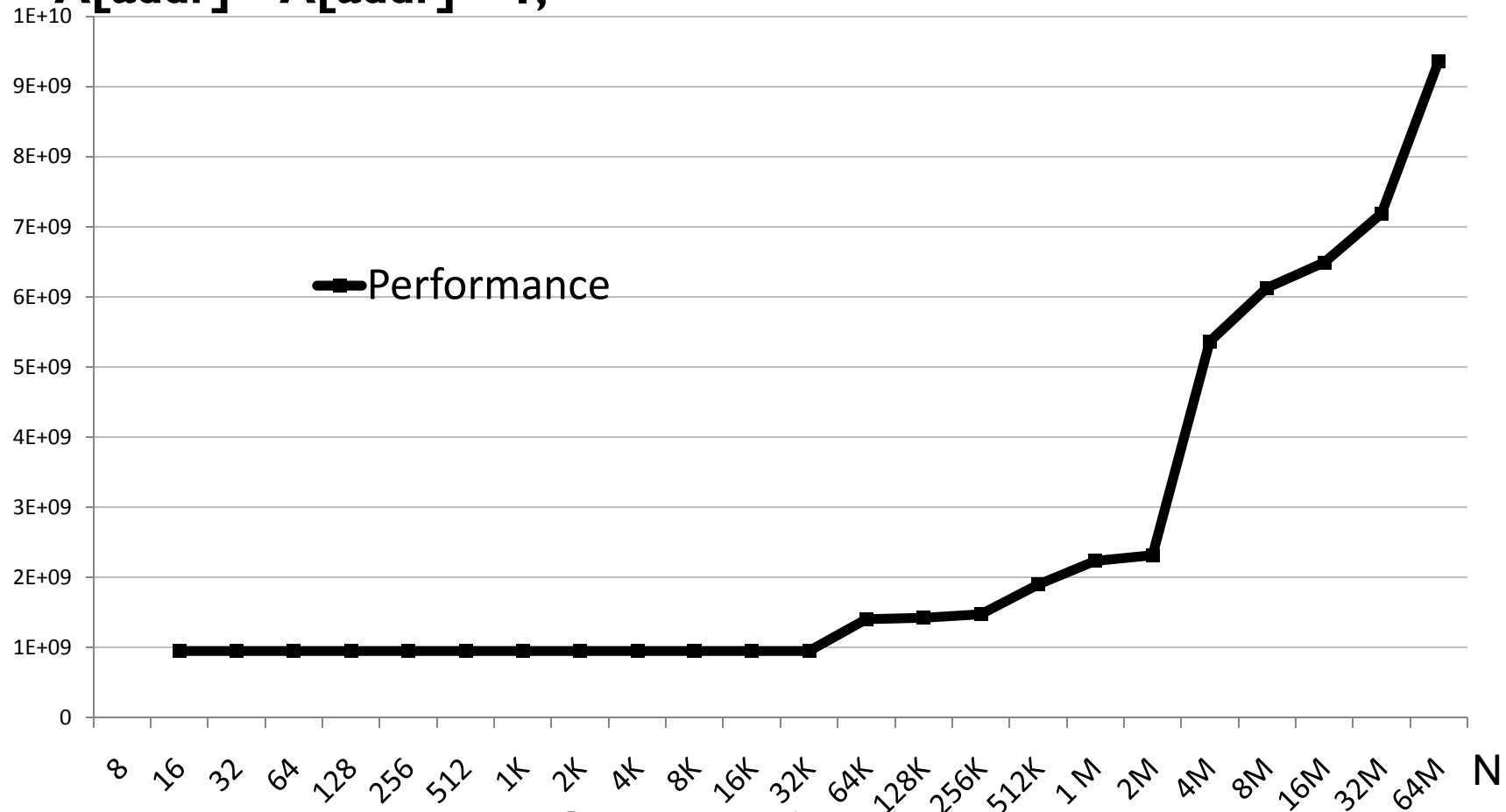
Intel® Nehalem™ Processor

```
for(rep=0; rep < REP; rep++)
    for(a=0; a <  N ; a++)
        A[a] = A[a] + 1;
```



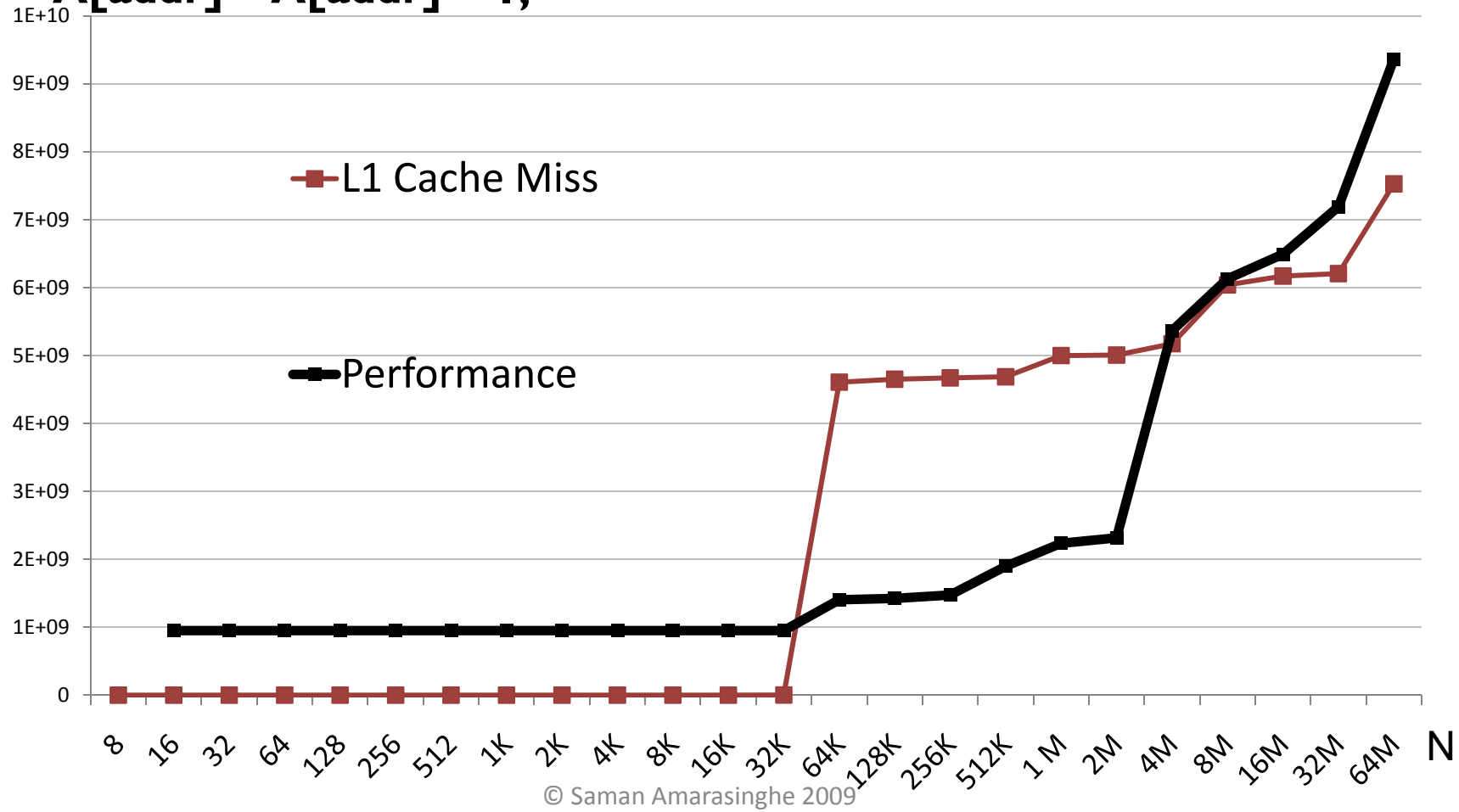
Intel® Nehalem™ Processor

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



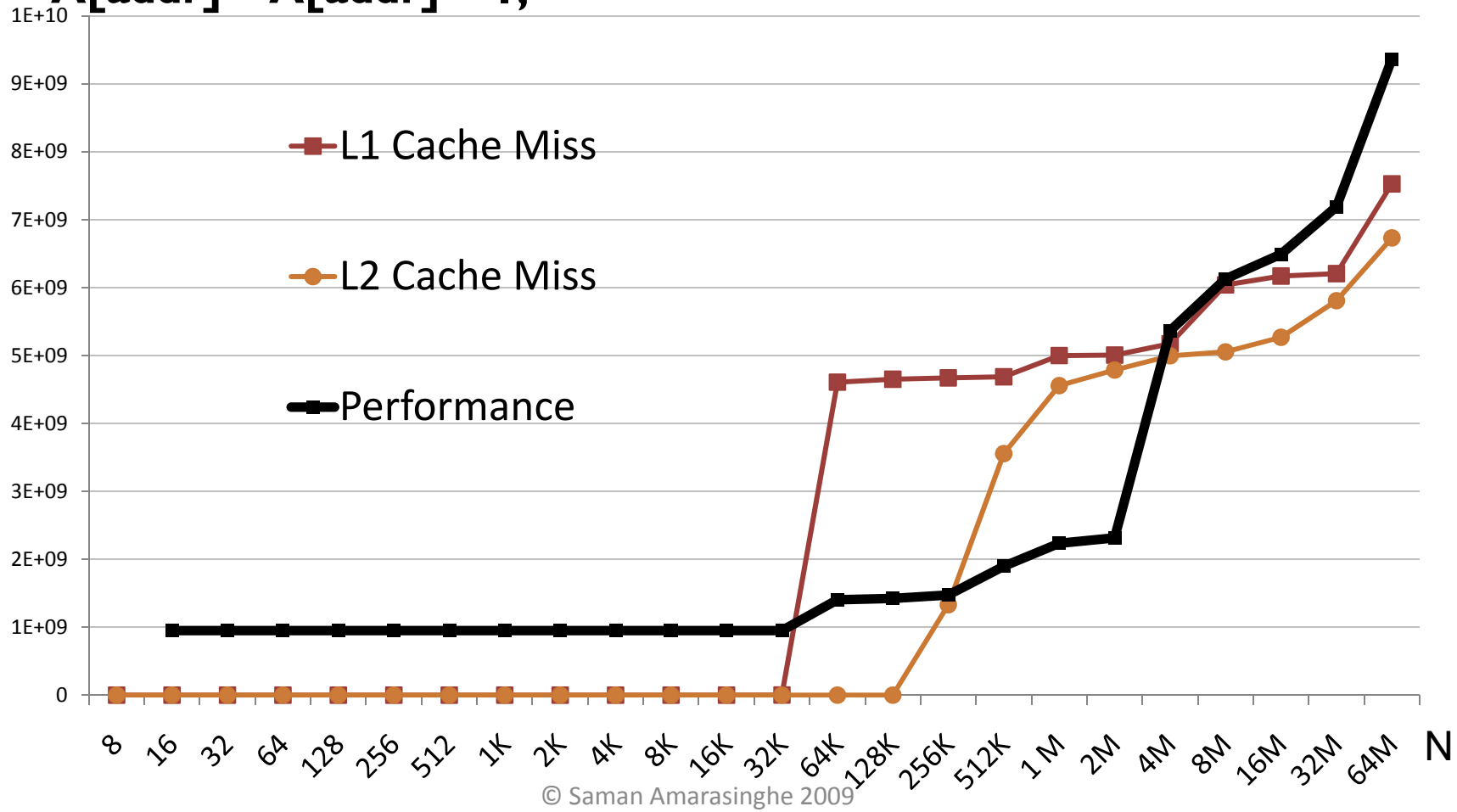
Intel® Nehalem™ Processor

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



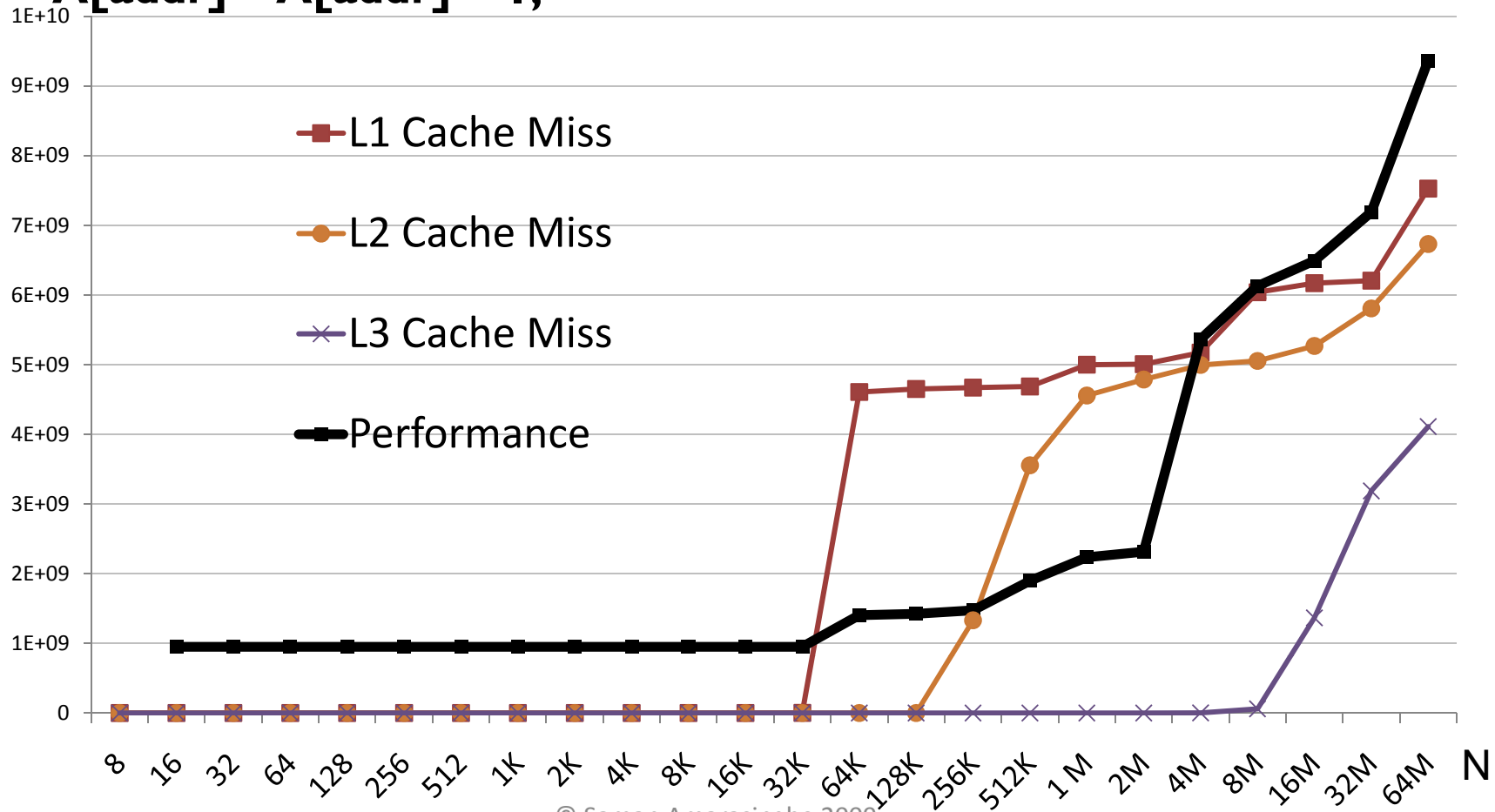
Intel® Nehalem™ Processor

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



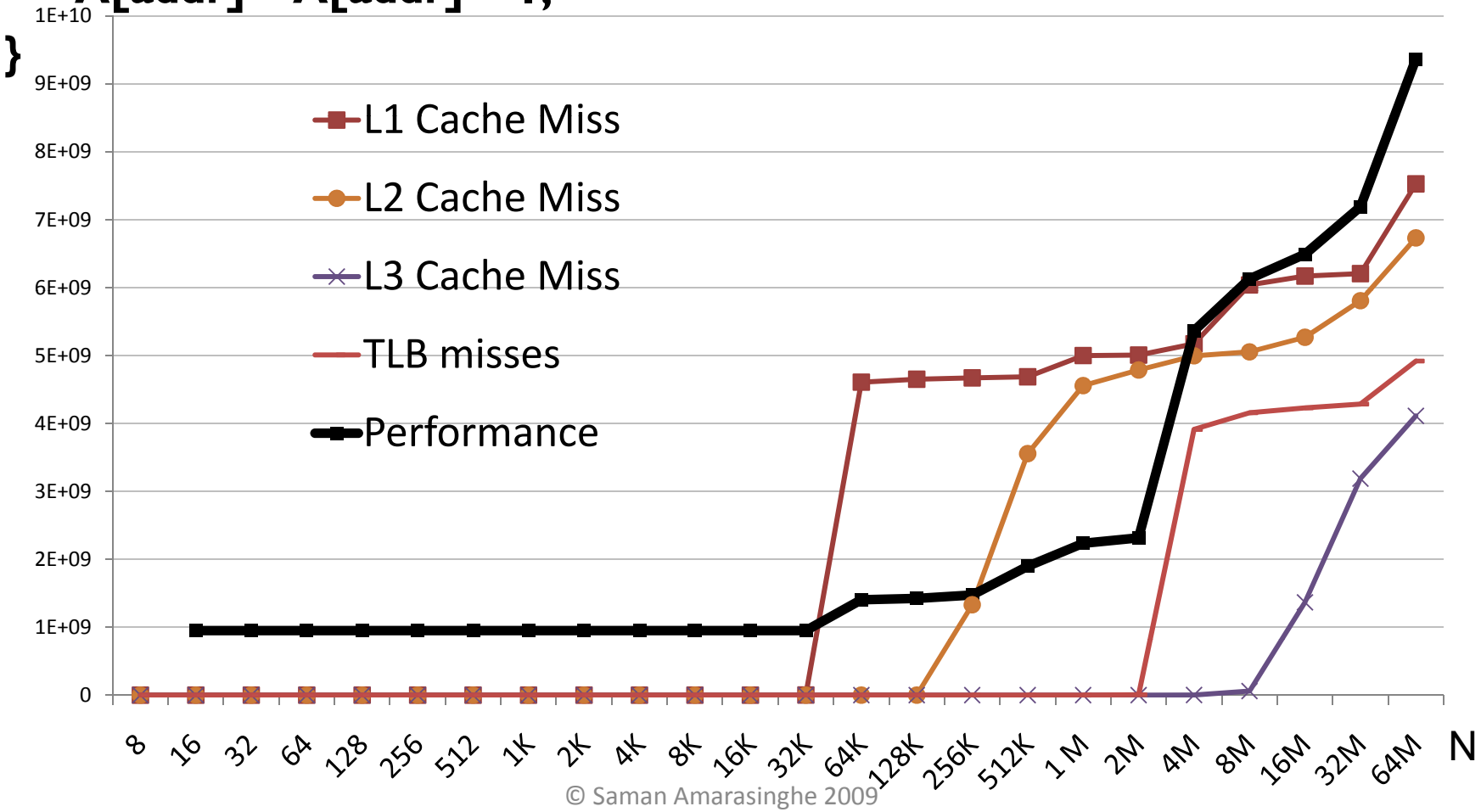
Intel® Nehalem™ Processor

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



Intel® Nehalem™ Processor

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



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