Reading Response

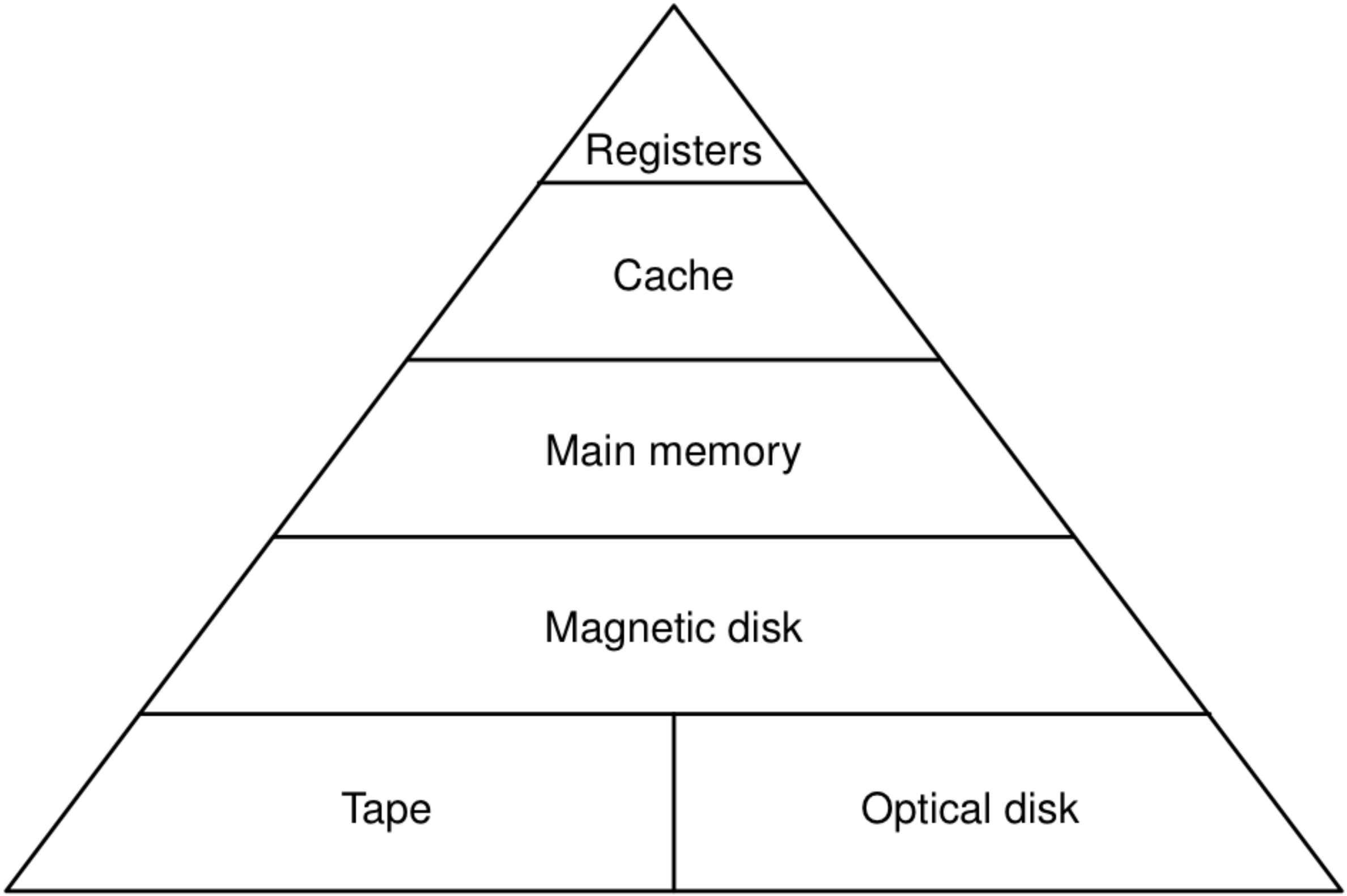
- Due Friday, December 10, 6am

SHARK: Architectural support for autonomic protection against stealth by rootkit exploits

http://portal.acm.org/citation.cfm?id=1521783

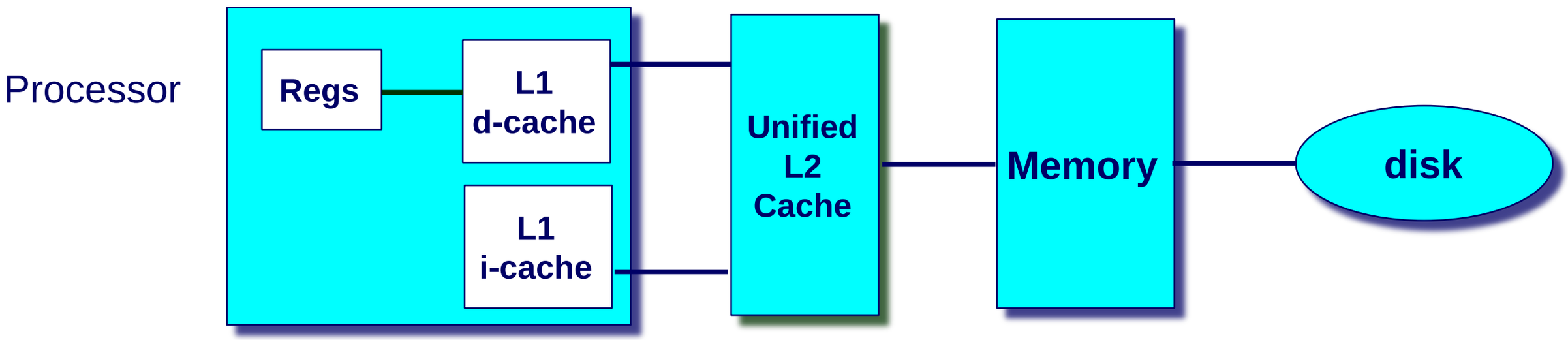
- New lab on Monday (quick & easy)
- Wednesday Lab: Reserve for lecture or review

Memory Hierarchy



Multi-Level Caches ($'s circa 2007)

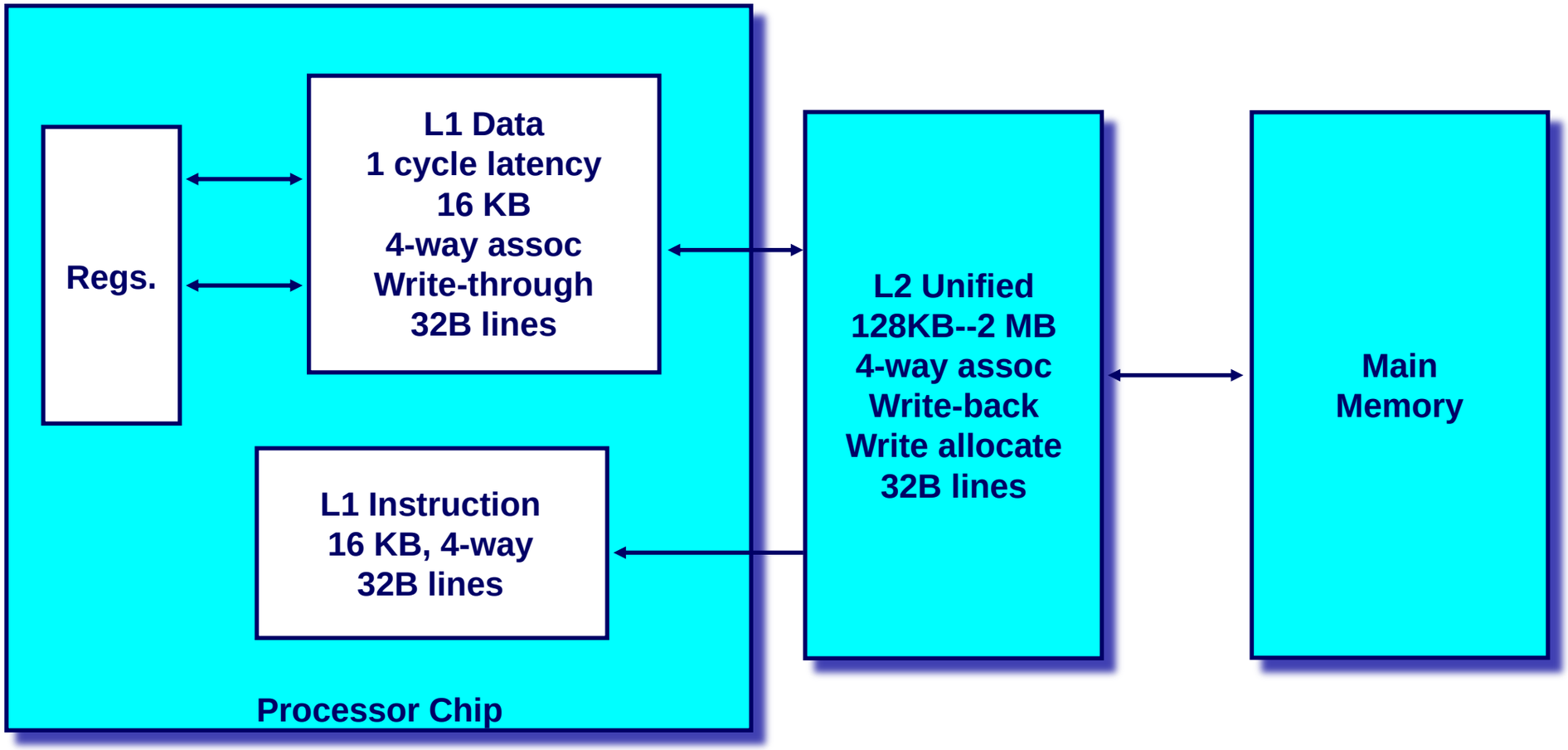
- Options: separate data and instruction caches, or a unified cache



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

larger, slower, cheaper

Intel Pentium Cache Hierarchy



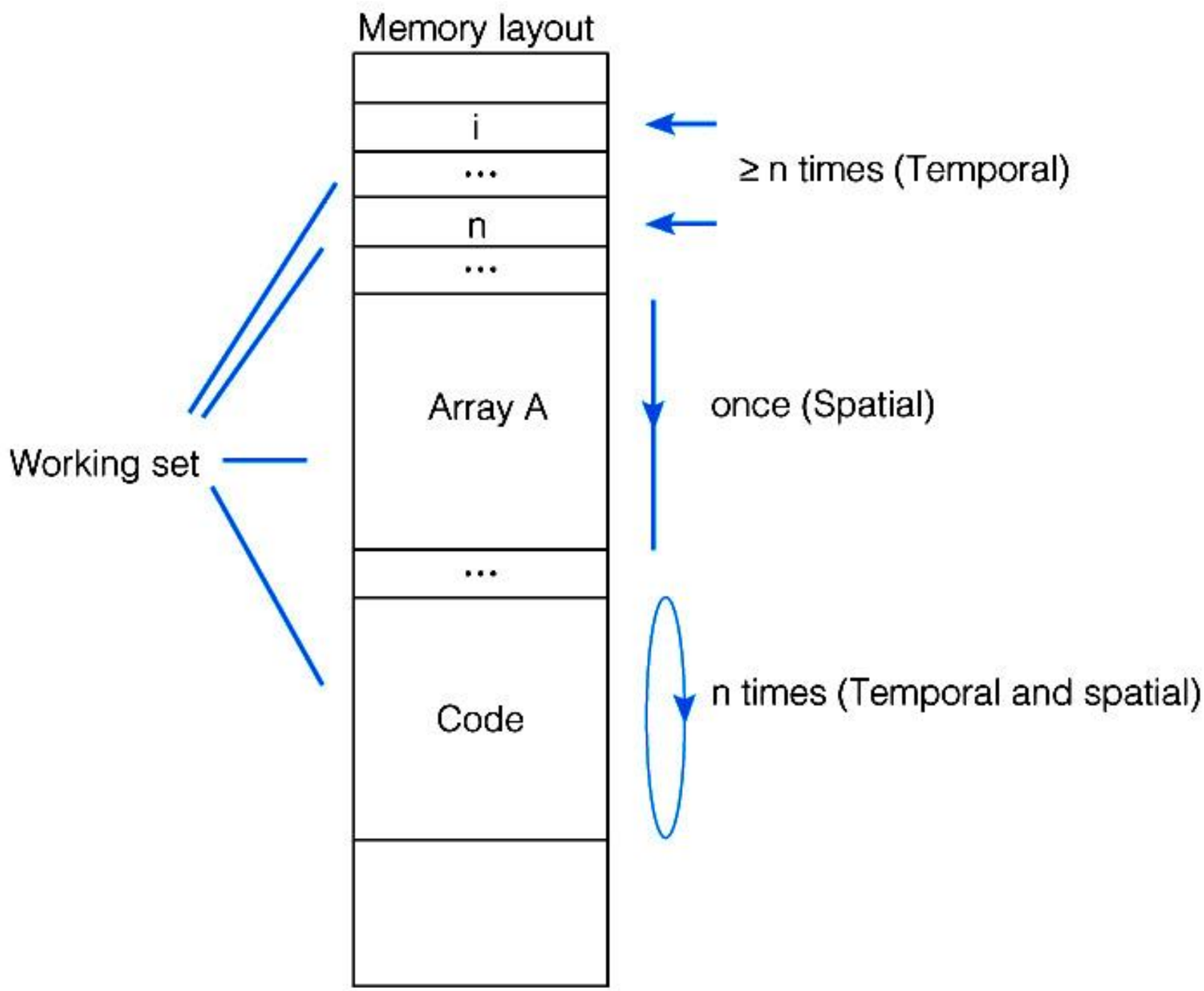
Locality

- **Principle of Locality** states with in a given period of time, programs tend to reference a relatively confined area of memory repeated.
- This Principle is a measurable phenomena and produces a **working set.**
- The working set is the set of memory locations referenced over a fixed period of time, or window, extending from the current time into the past.

Locality (2)

Spatial Locality, or the fact that when a given address has been referenced, it is likely that addresses near it will be referenced within a short period of time.

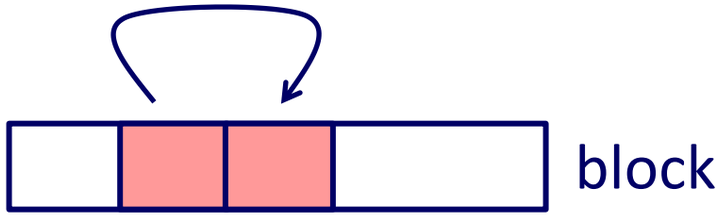
Temporal Locality, or the fact that once a particular memory item has been referenced, it is likely that it will be referenced again within a short period of time.



Consider a “block” in a cache

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future

- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time



2-Level Hierarchy

- A two level hierarchy suggests that memory can be structured in two layers, or two interacting layers.
  - **Primary level** -- small and fast
  - **Secondary level** -- large and slow
- This relationship indicates a trade-off in memory technologies between processing time expense and hardware purchasing expense.

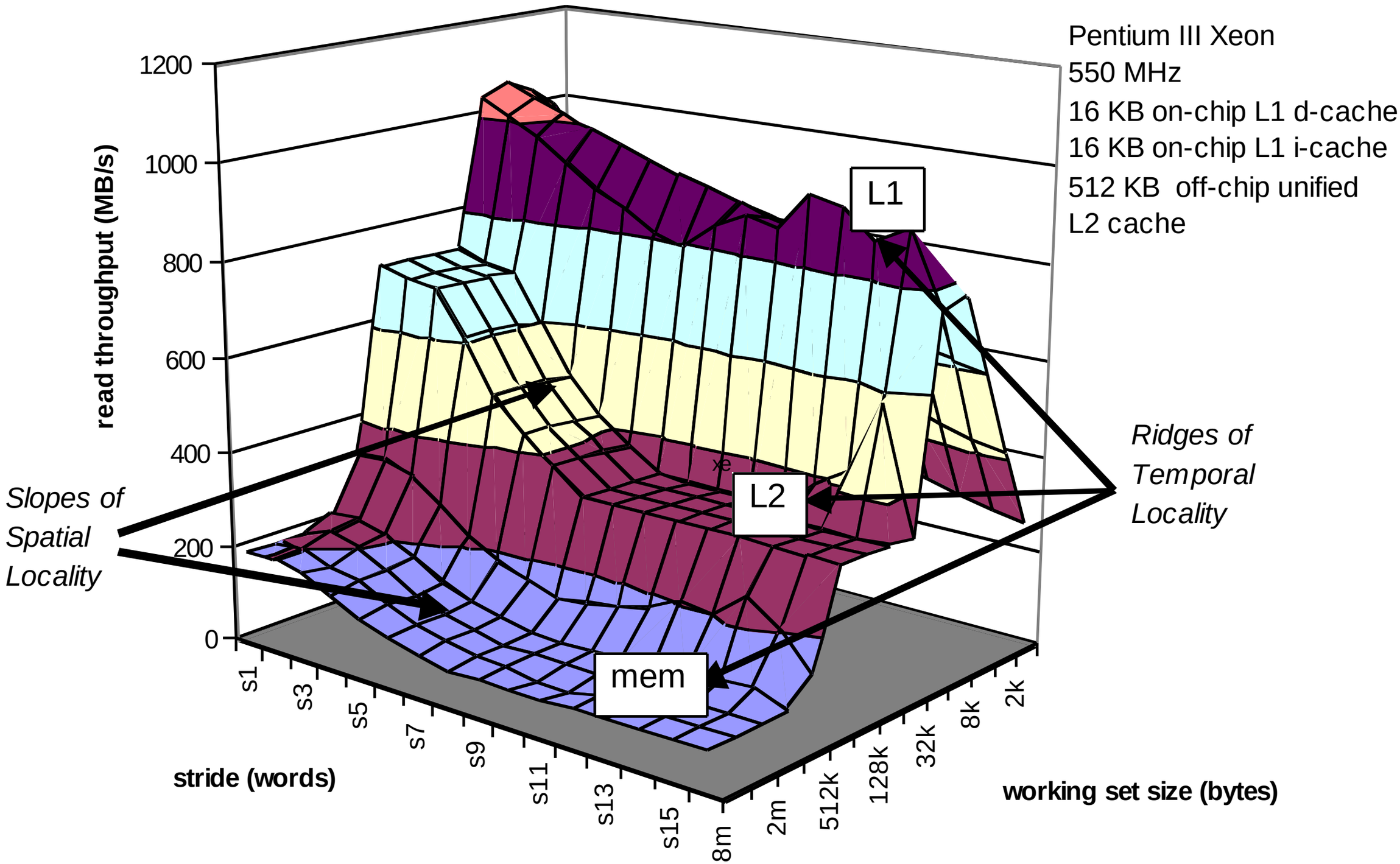
Memory Mountain Main Routine

```
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS];         /* The array we'll be traversing */

int main()
{
    int size;         /* Working set size (in bytes) */
    int stride;       /* Stride (in array elements) */
    double Mhz;       /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0);              /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
      for (stride = 1; stride <= MAXSTRIDE; stride++)
          printf("%.1f\t", run(size, stride, Mhz));
      printf("\n");
    }
    exit(0);
}
```

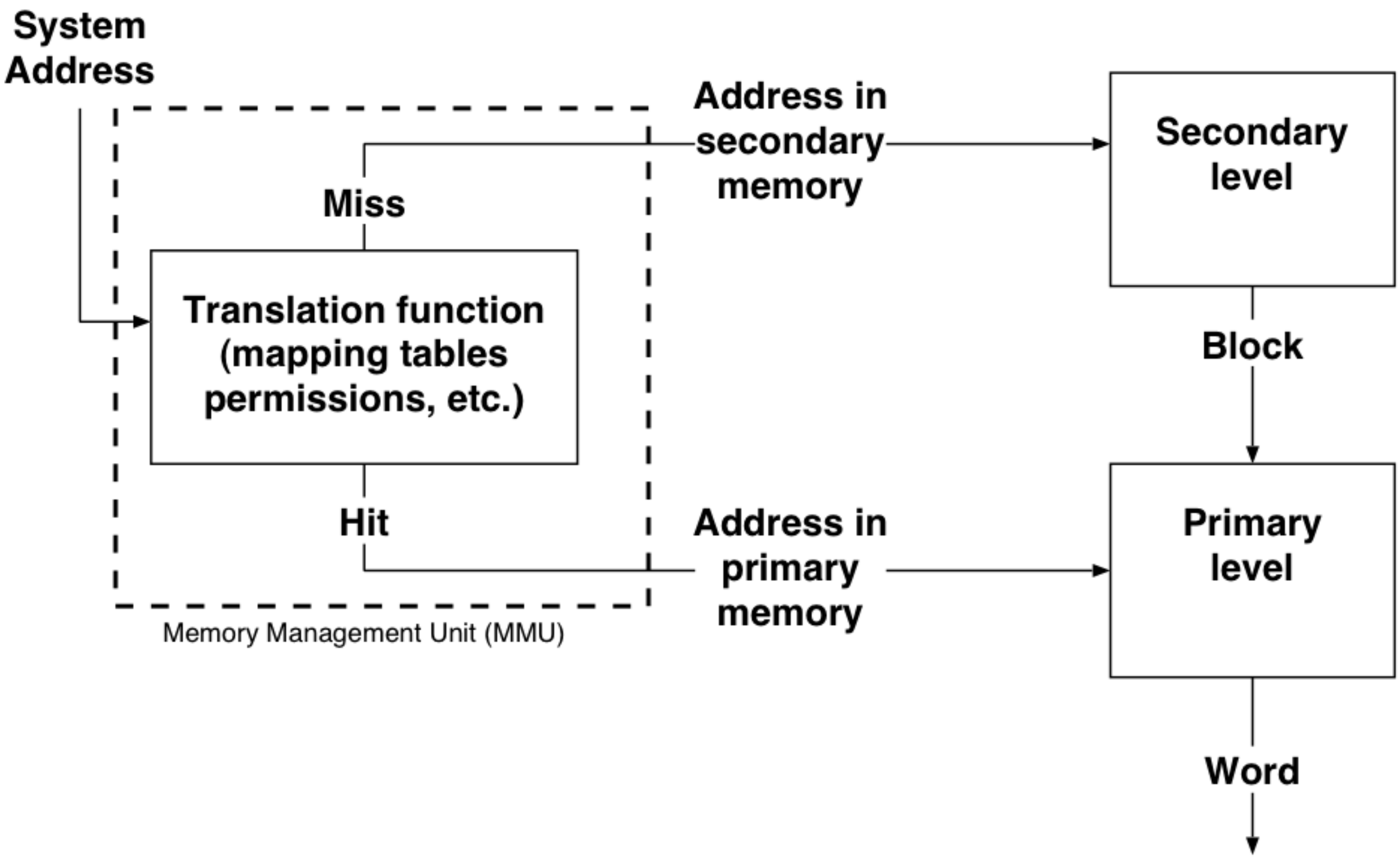
Memory Mountain Test Function

```
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
      result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);                      /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

The Memory Mountain



Has Mathematical Notation



Hits vs misses

$$t_a = ht_p + (1 - h)t_s$$

t_a	actual access time
t_p	primary memory access time
t_s	secondary memory access time
h	hit ratio, or hits per memory reference

Hits vs misses

$$t_a = ht_p + (1 - h)t_s$$

- **Assume:**

t_p -- primary memory access time is 10 ns.
t_s -- secondary memory access time is 100 ns.

Hits vs misses

$$t_a = ht_p + (1-h)t_s$$

h	ht_p	$(1-h)t_s$	t_a
0	0	100	100
0.2	2	80	82
0.4	4	60	64
0.5	5	50	55
0.6	6	40	46
0.8	8	20	28
1.0	10	0	10

Matrix Multiplication Example

- **Major Cache Effects to Consider**
  - Total cache size
    - Exploit temporal locality and keep the working set small (e.g., by using blocking)
  - Block size
    - Exploit spatial locality

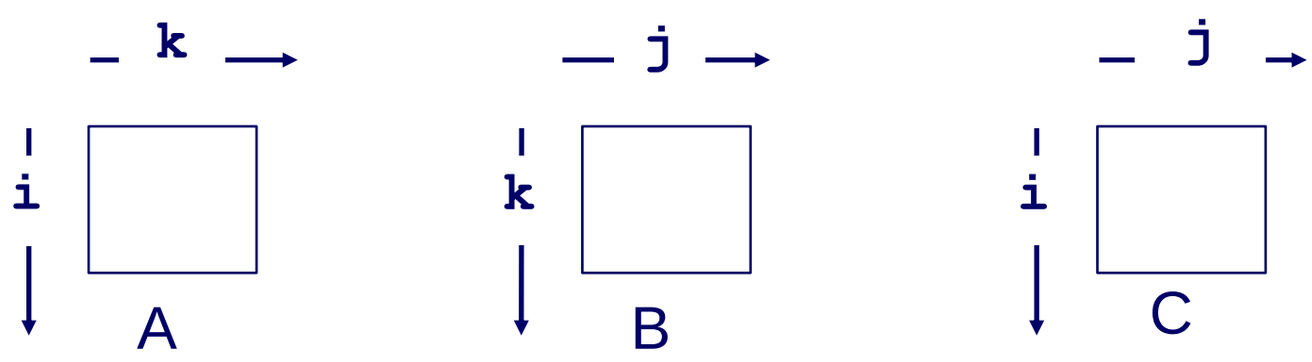
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

*Variable **sum** held in register*

- **Description:**
  - Multiply N x N matrices
  - O(N3) total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - **Cache Line (Block) size = 32 bytes (fits 4 64-bit words)**
  - **Matrix dimension (N) is very large**
    - **Approximate 1/N as 0.0**
  - **Cache is not even big enough to hold multiple rows**
- **Analysis Method:**
  - **Look at access pattern of inner loop**



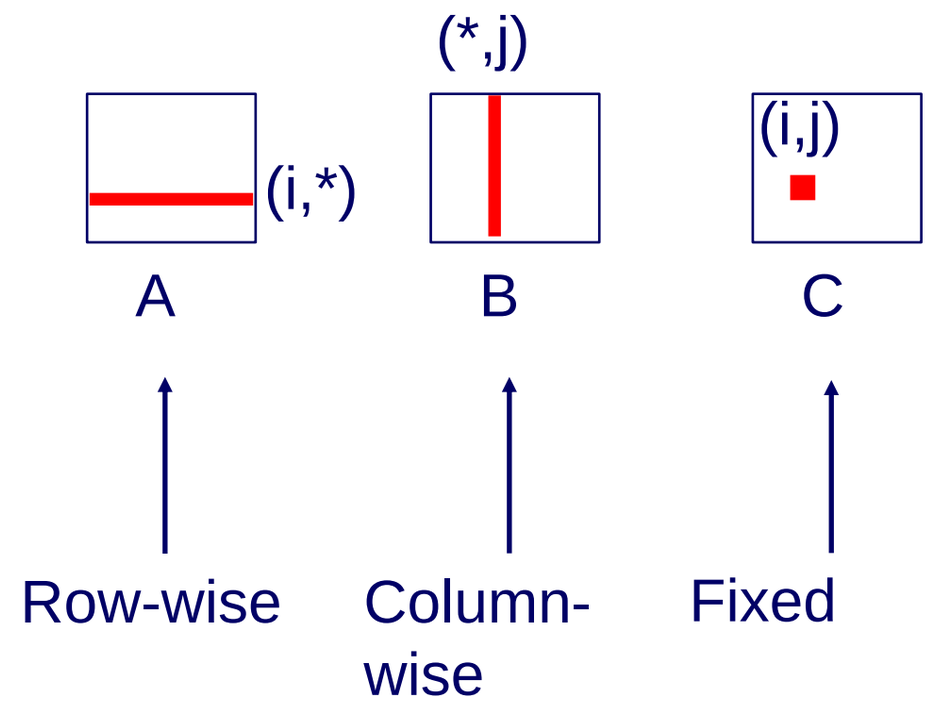
Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    - `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`
    - `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:



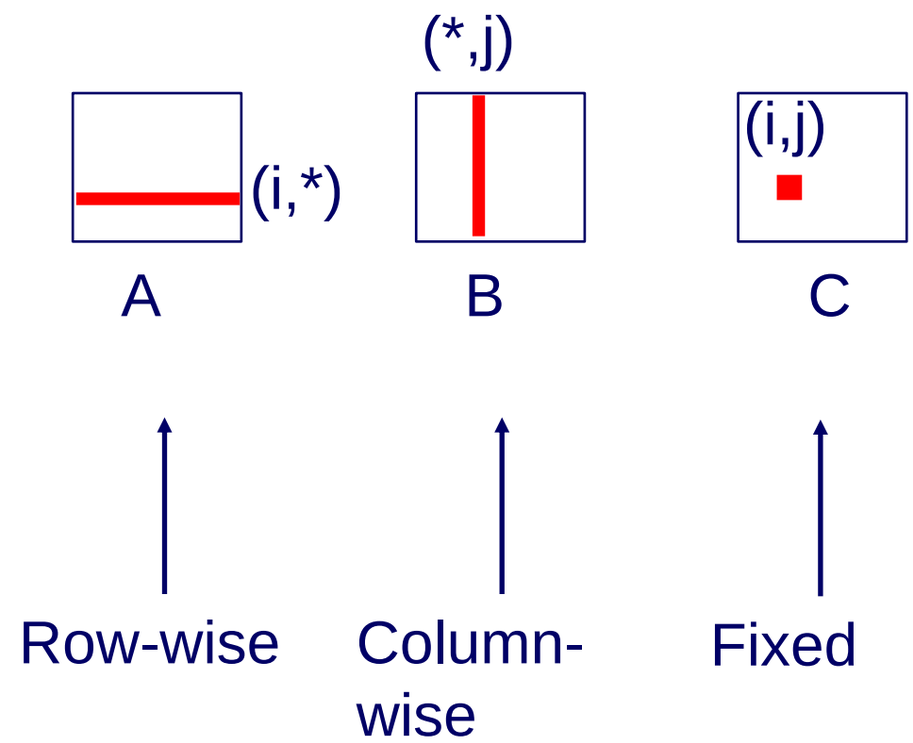
Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

Inner loop:



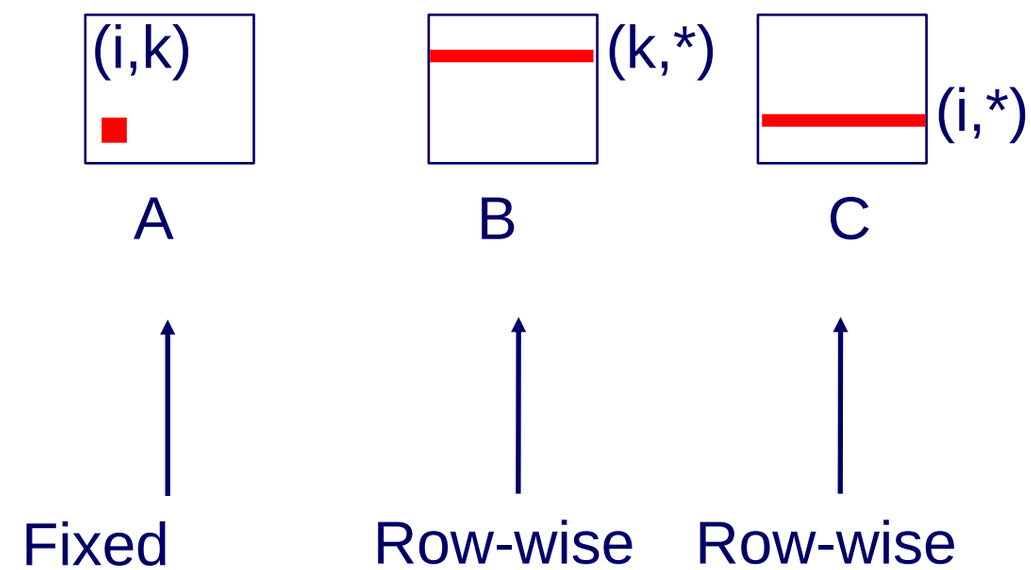
Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Inner loop:



Misses per Inner Loop Iteration:

A **B** **C**

Approx speedup relative to 1.25?

Cache = 10ms and memory = 100ms

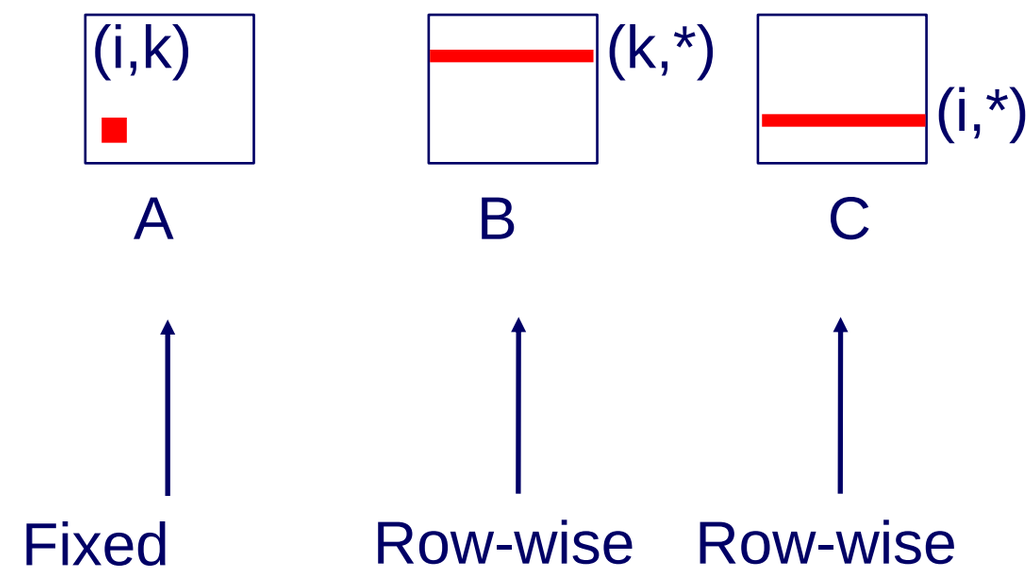
n is BIG

$$t_a = ht_p + (1-h)t_s$$

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Inner loop:



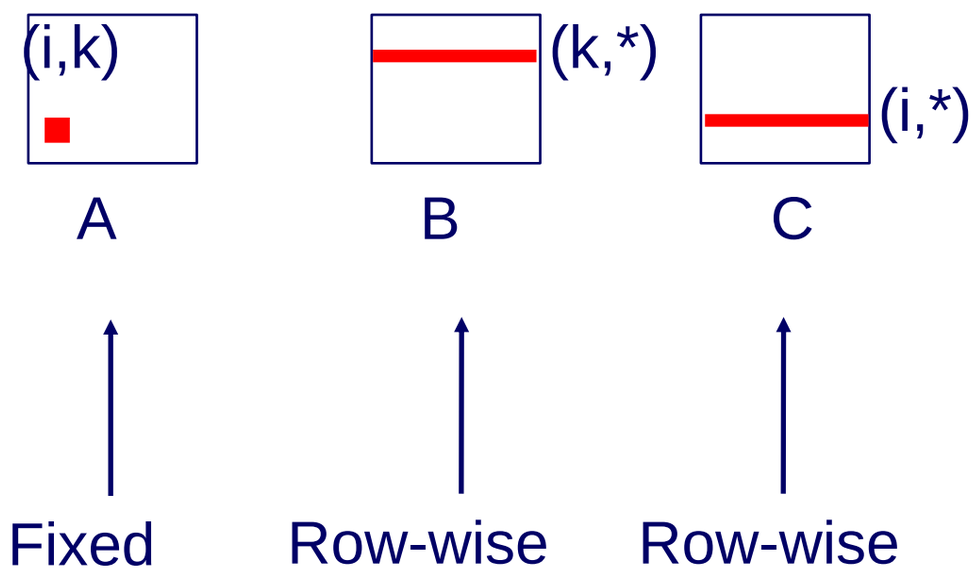
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Inner loop:



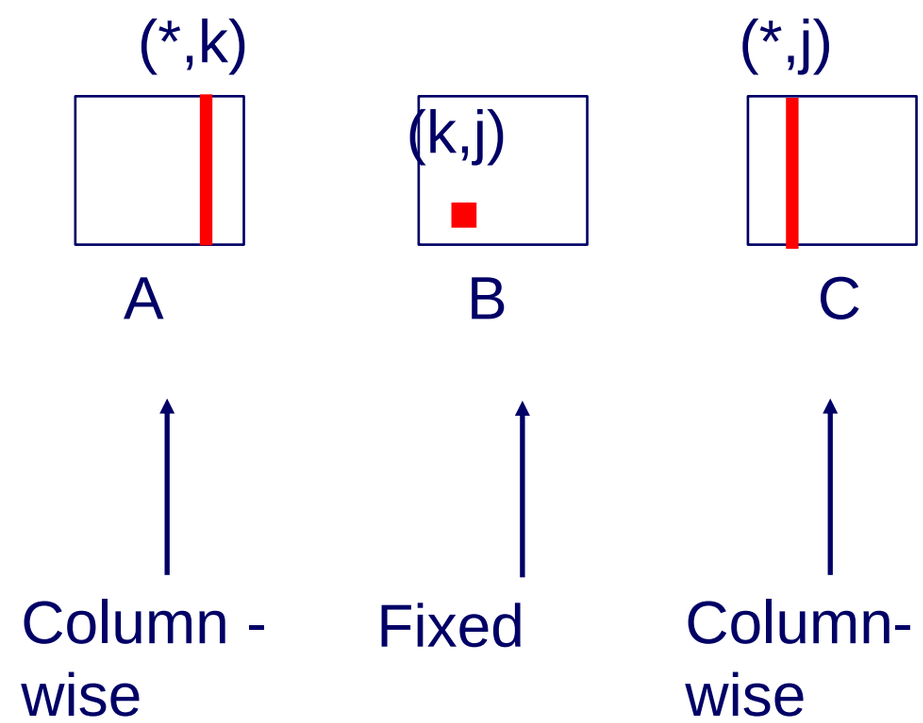
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:



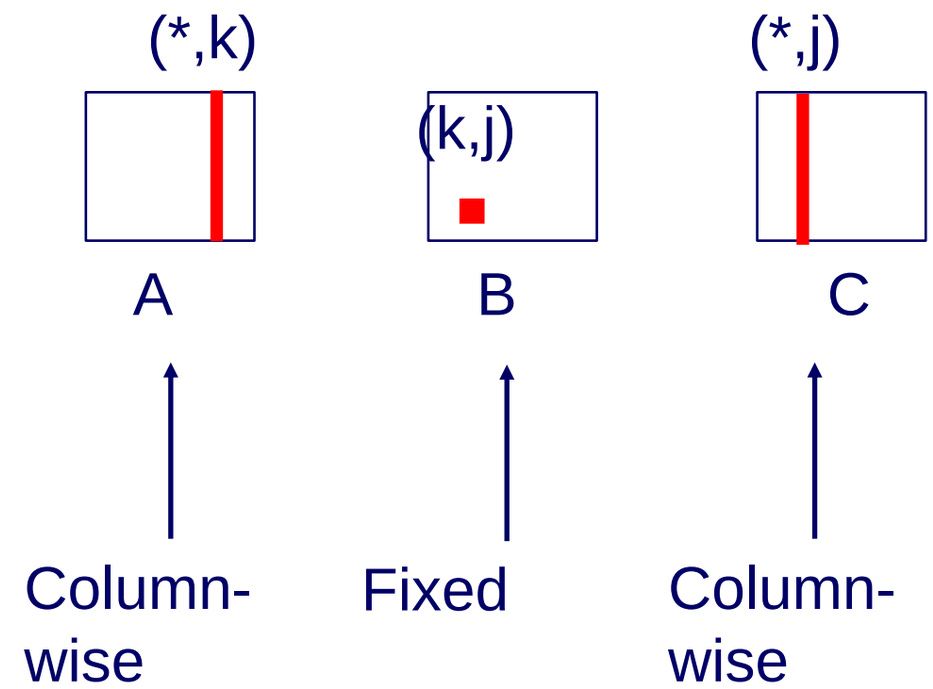
Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:



Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

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

```
for (i=0; i<n; i++)  {
   for (j=0; j<n; j++) {
      sum = 0.0;
      for (k=0; k<n; k++)
         sum += a[i][k] * b[k][j];
      c[i][j] = sum;
   }
}
```

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

```
for (k=0; k<n; k++) {
   for (i=0; i<n; i++) {
      r = a[i][k];
      for (j=0; j<n; j++)
         c[i][j] += r * b[k][j];
   }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
for (j=0; j<n; j++) {
   for (k=0; k<n; k++) {
      r = b[k][j];
      for (i=0; i<n; i++)
         c[i][j] += a[i][k] * r;
   }
}
```

Pentium Matrix Multiply Performance

- Miss rates are helpful but not perfect predictors.