CIS 450 – Computer Architecture and Organization

Lecture 18: Cache Memories (cont.)

Mitch Neilsen

(neilsen@ksu.edu)

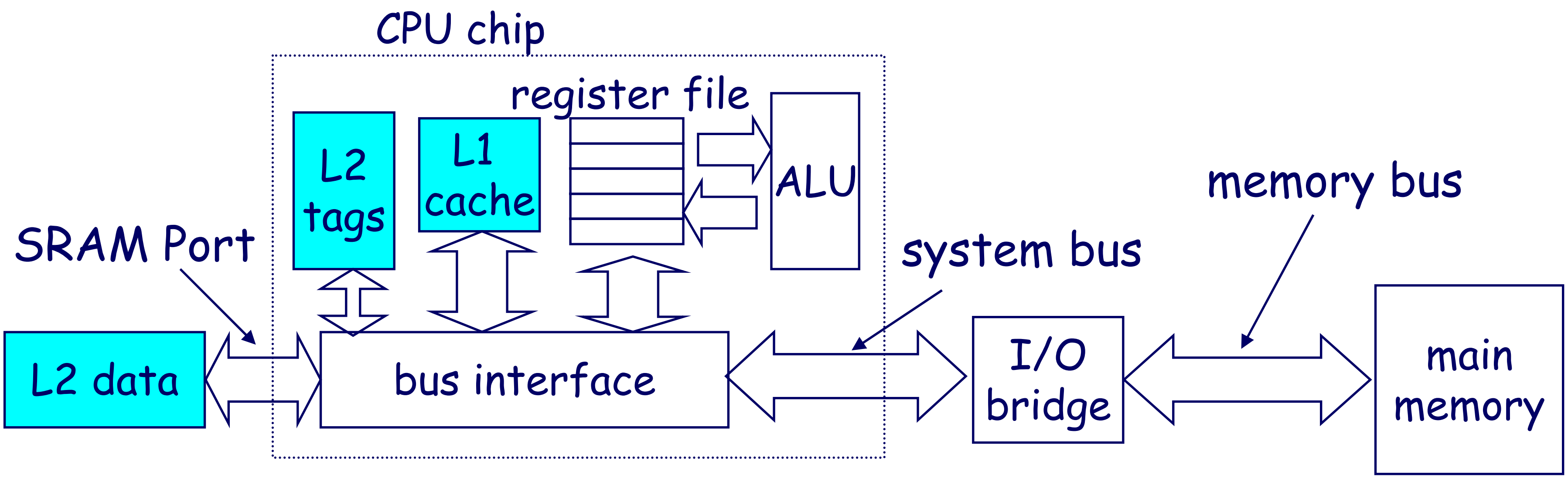
219D Nichols Hall

Topics

- **Impact of caches on performance**
- **The memory mountain**

Cache Memories

- **Cache memories are small, fast SRAM-based memories managed automatically in hardware.**
  - **Hold frequently accessed blocks of main memory**
- **CPU looks first for data in L1, then in L2, then in main memory.**
- **Typical system structure:**



Writing Cache Friendly Code

- **Repeated references to variables are good (temporal locality)**
- **Stride-1 reference patterns are good (spatial locality)**
- **Examples:**
  - **cold cache, 4-byte words, 4-word cache blocks**

```
int sum_array_rows(int a[M][N])
{
  int i, j, sum = 0;

  for (i = 0; i < M; i++)
     for (j = 0; j < N; j++)
        sum += a[i][j];
  return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
  int i, j, sum = 0;

  for (j = 0; j < N; j++)
     for (i = 0; i < M; i++)
        sum += a[i][j];
  return sum;
}
```

Miss rate = 1/4 = 25%

Miss rate = 100%

The Memory Mountain

Read throughput (read bandwidth)

- **Number of bytes read from memory per second (MB/s)**

Memory mountain

- **Measured read throughput as a function of spatial and temporal locality.**
- **Compact way to characterize memory system performance.**

Memory Mountain Test Function

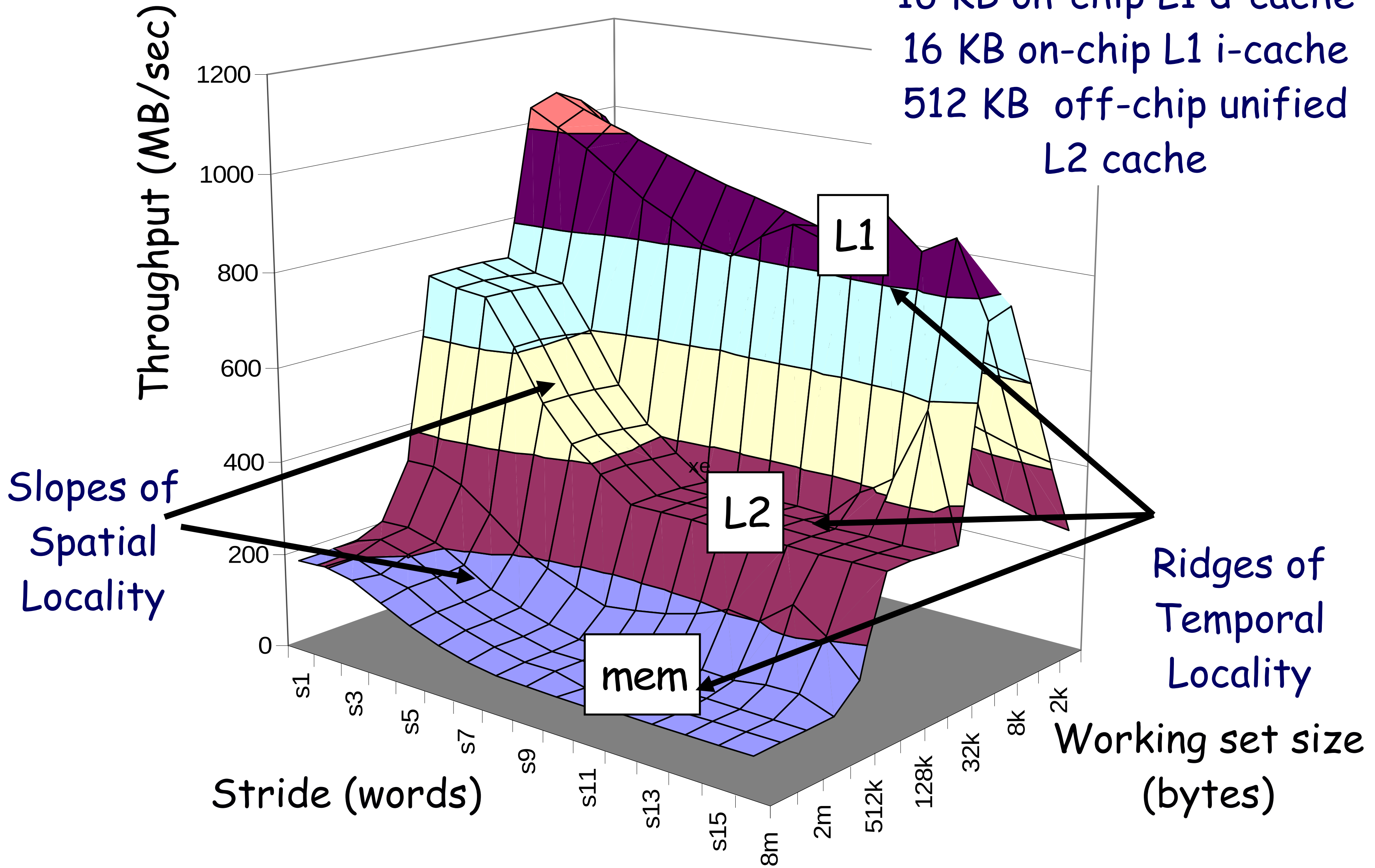
```
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    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 */
}
```

Memory Mountain Main Routine

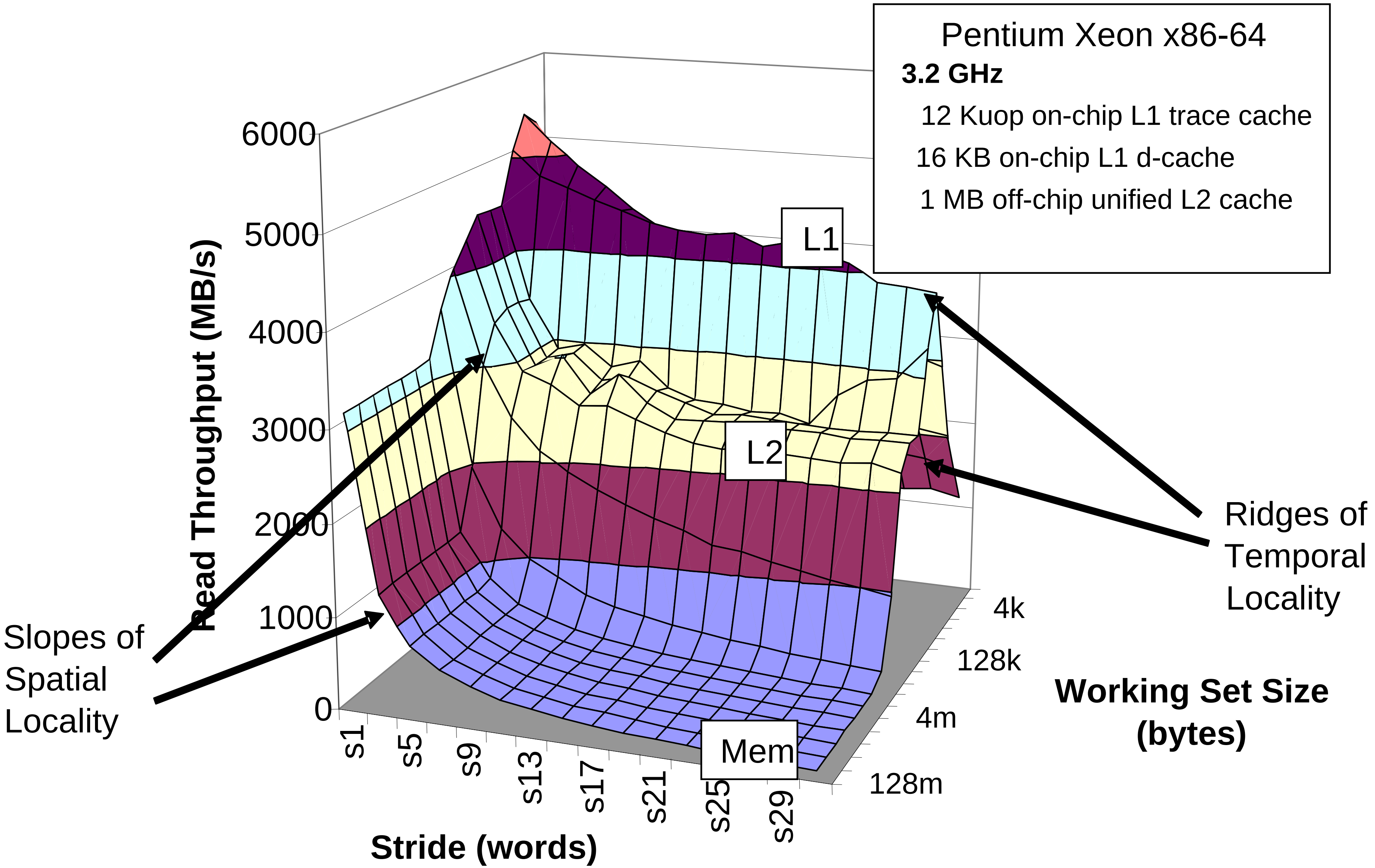
```
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

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);
}
```

The Memory Mountain

Pentium III
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 cache

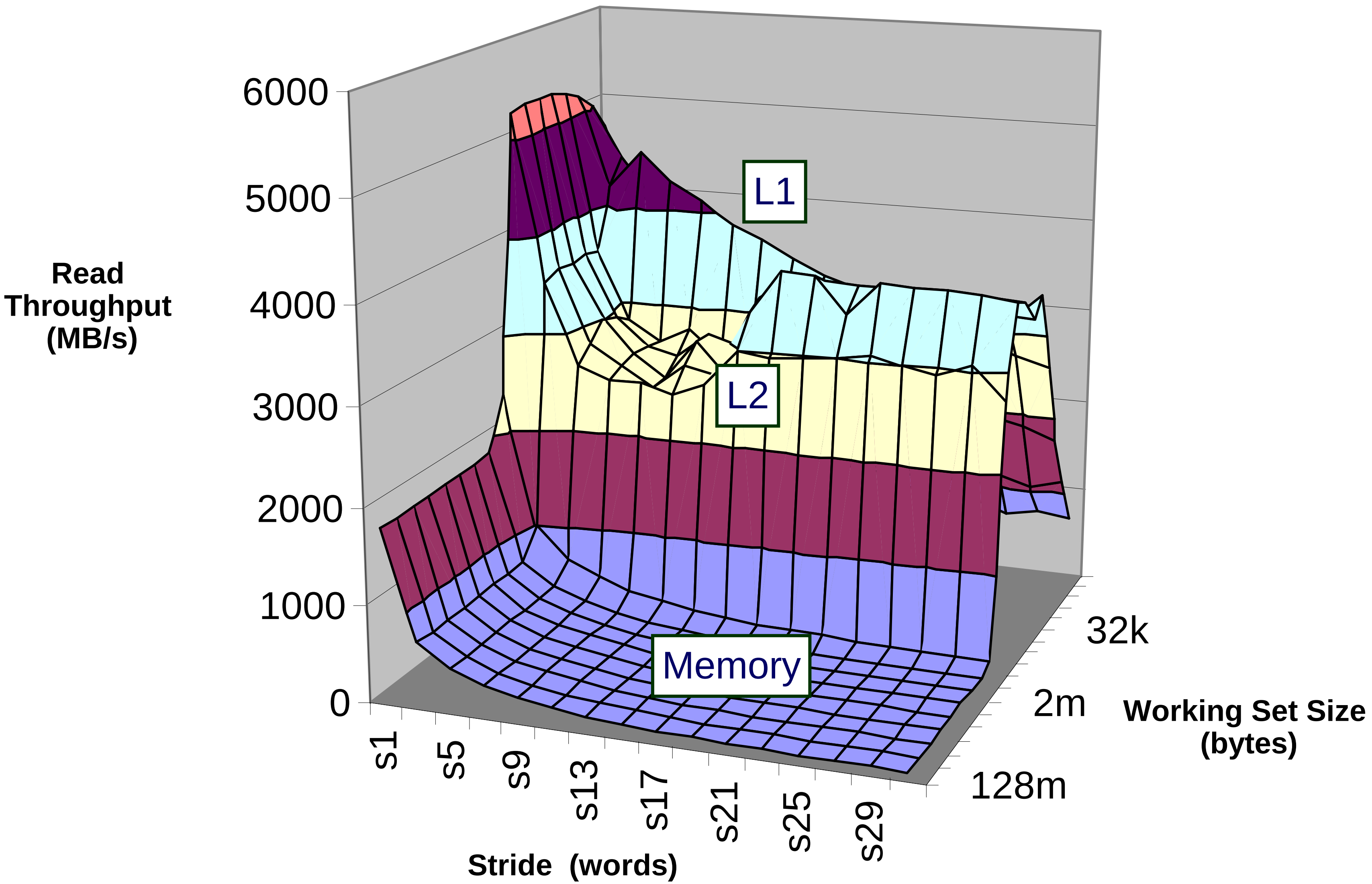


X86-64 Memory Mountain

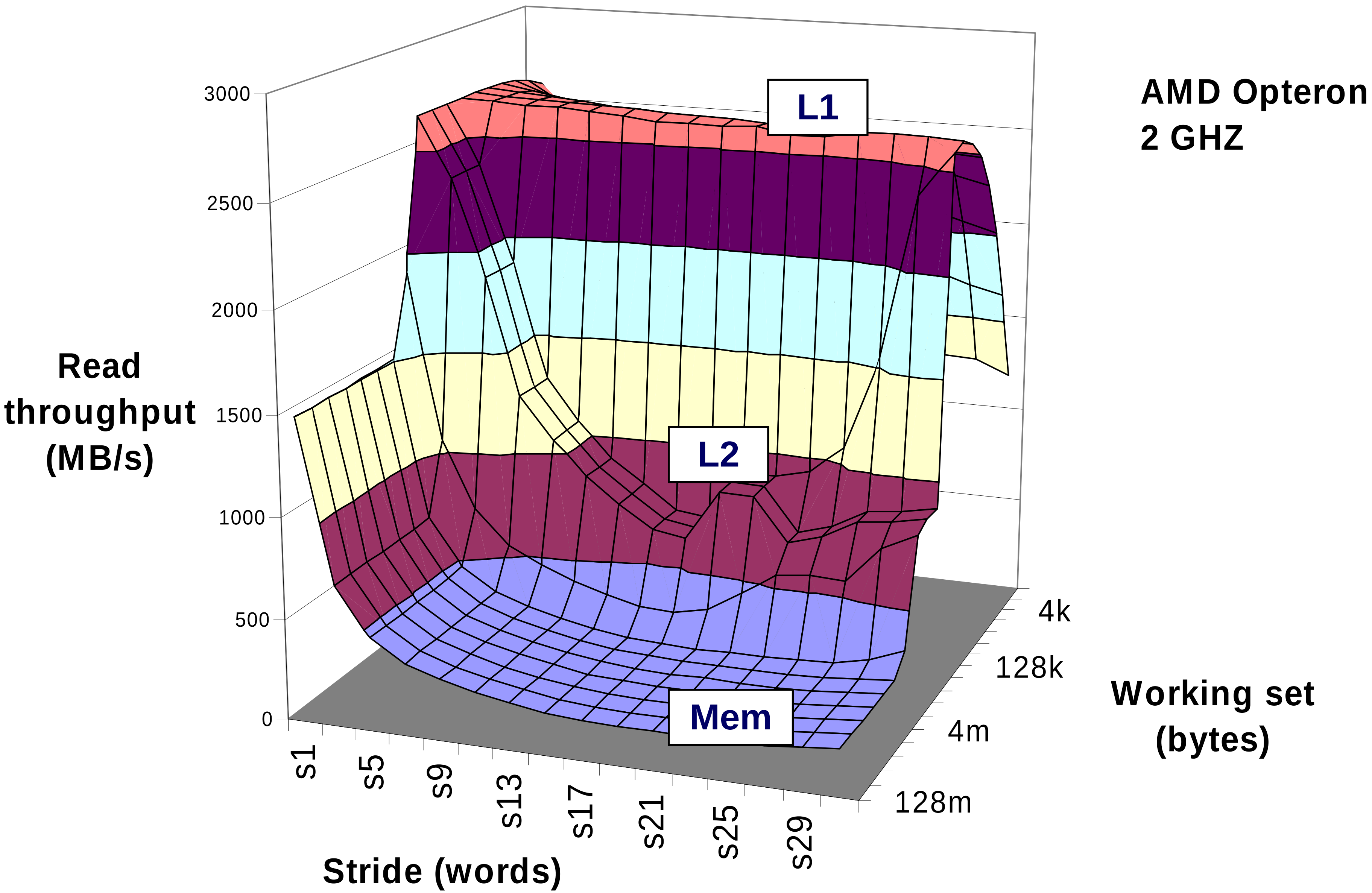


Camaro.cis.ksu.edu Memory Mountain

(Intel(R) Xeon(TM) CPU 2.66GHz)



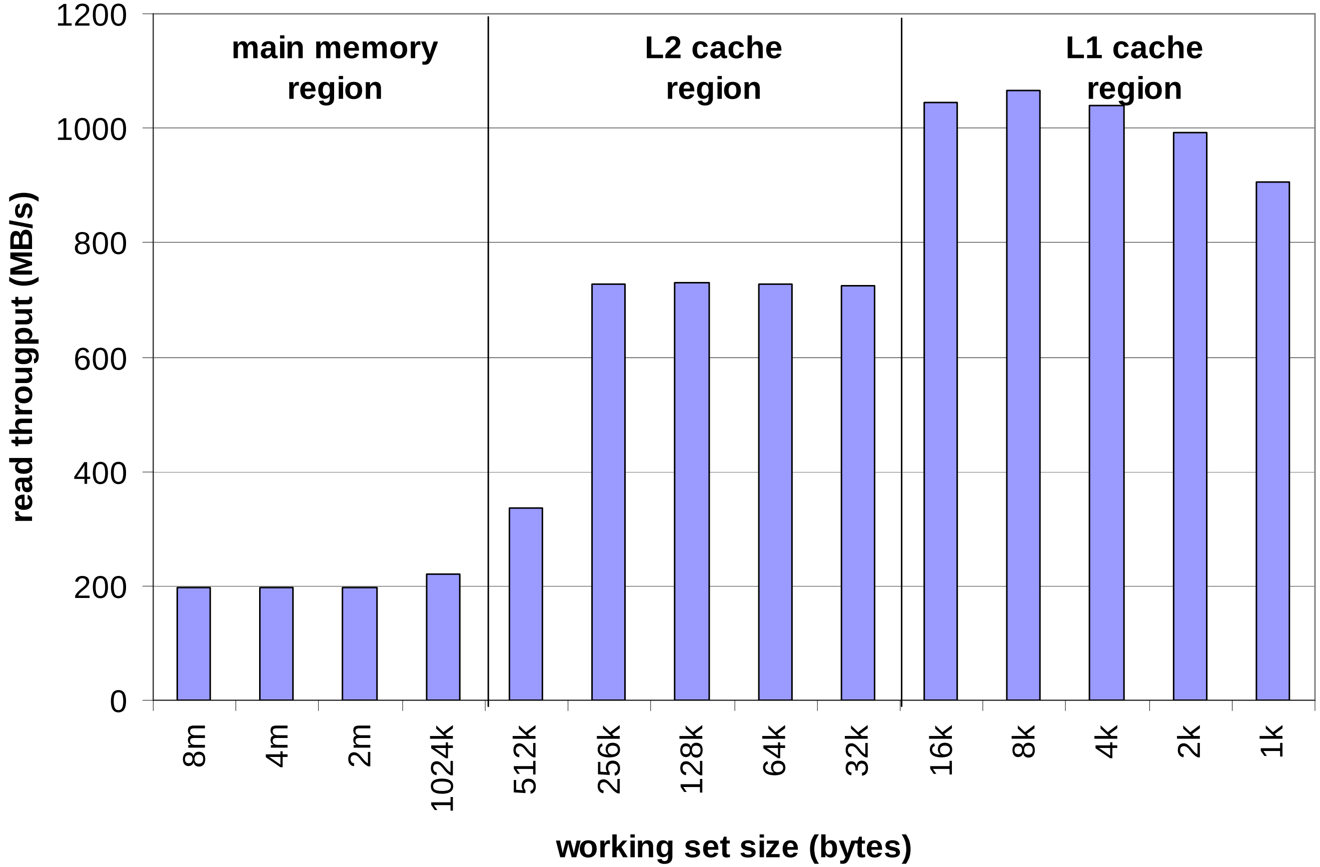
Opteron Memory Mountain



Ridges of Temporal Locality

Slice through the memory mountain with stride=1

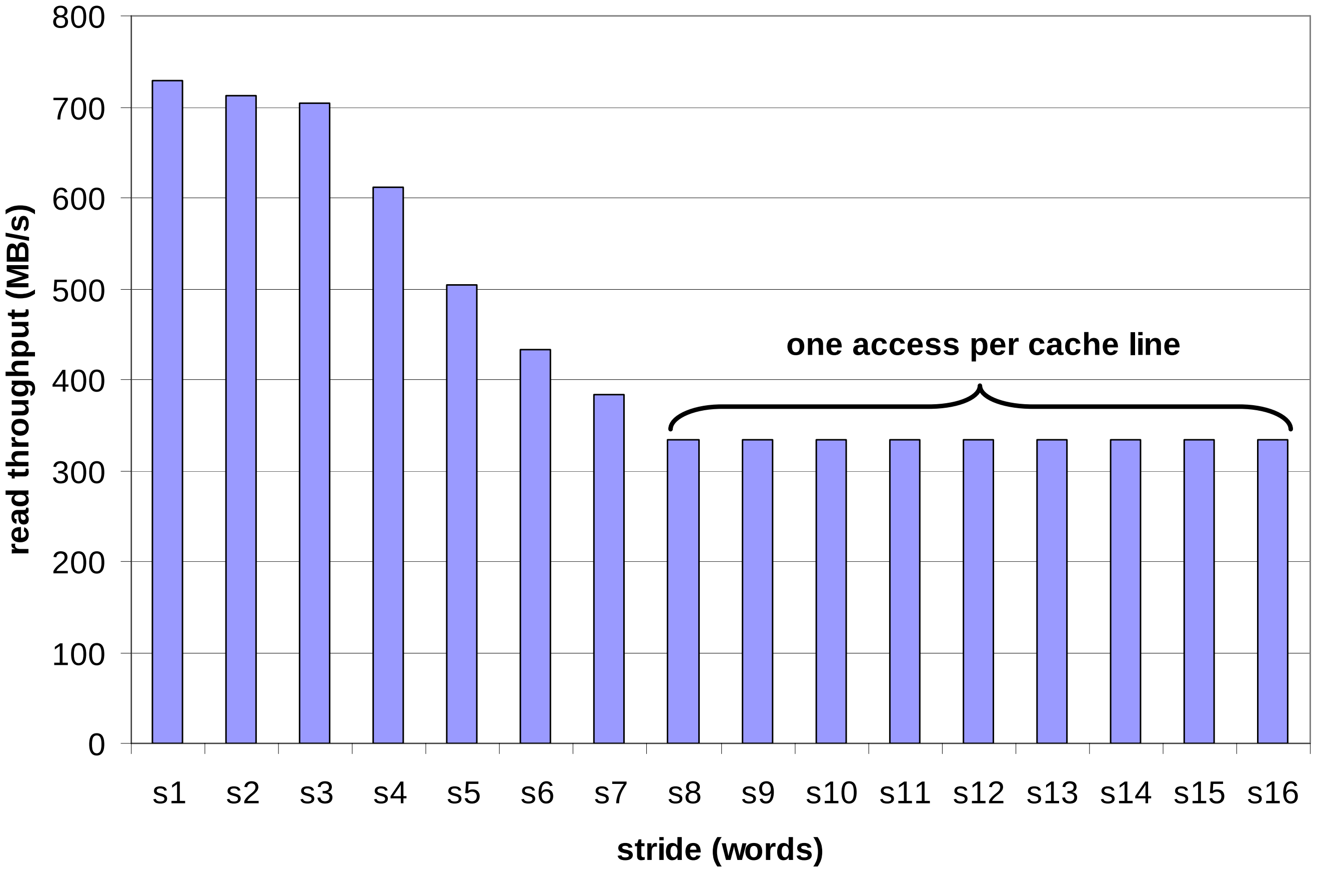
- **illuminates read throughputs of different caches and memory**



A Slope of Spatial Locality

Slice through memory mountain with size=256KB

- shows cache block size.



Matrix Multiplication Example

Major Cache Effects to Consider

- Total cache size
  - Exploit temporal locality and keep the working set small (e.g., use blocking)
- Block size
  - Exploit spatial locality

Description:

- Multiply N x N matrices
- $O(N^3)$ total operations
- Accesses
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

Variable ***sum*** *held in register*

```
/* 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;
  }
}
```

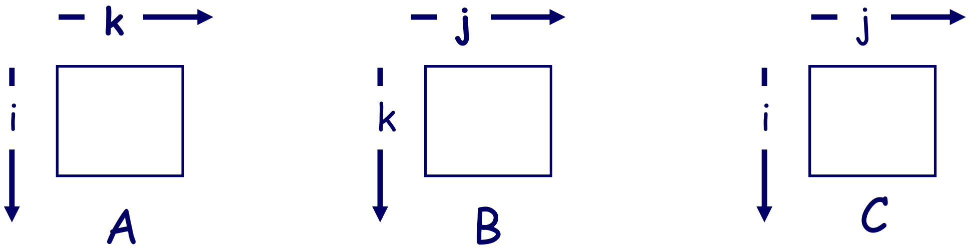
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for four 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:

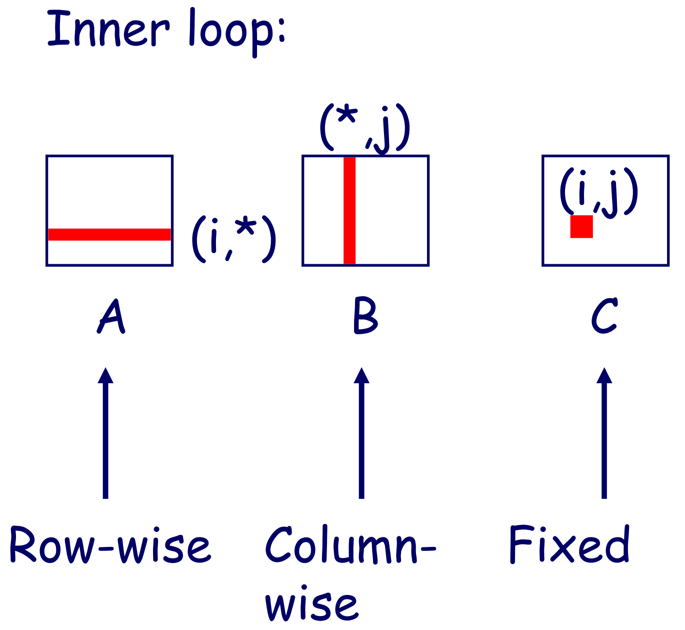
- ```
  for (i = 0; i < N; i++)
    sum += a[0][i];
  ```
- **accesses successive elements**
- **if block size (B) > 4 bytes, exploit spatial locality**
  - **compulsory miss rate = 4 bytes / B**

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;
  }
}
```



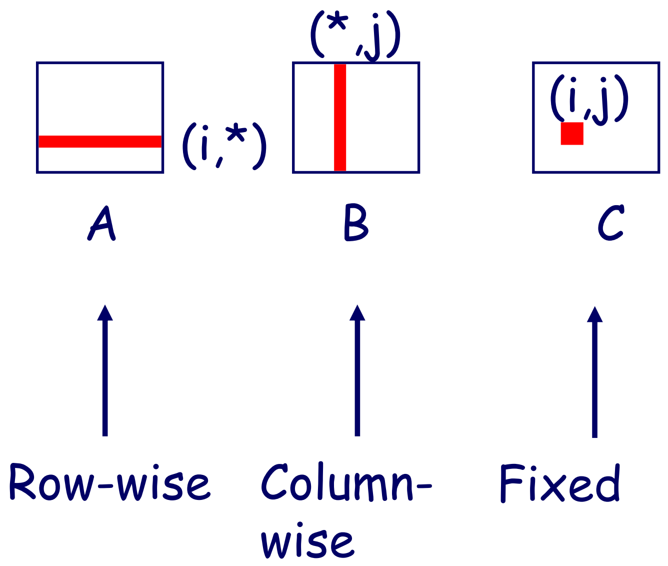
Misses per Inner Loop Iteration:

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:

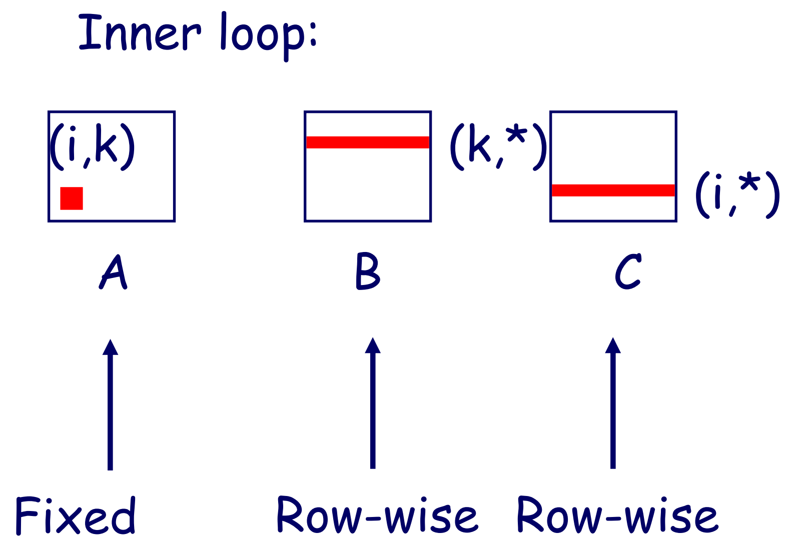


Misses per Inner Loop Iteration:

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];
  }
}
```

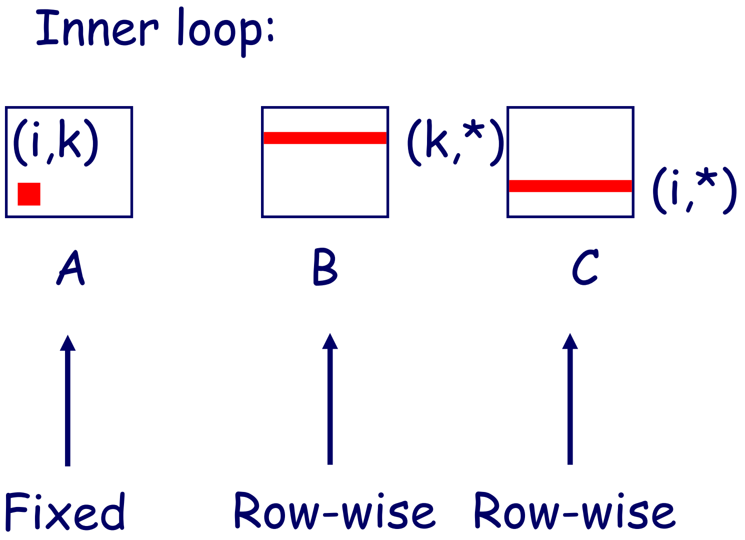


Misses per Inner Loop Iteration:

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];
  }
}
```

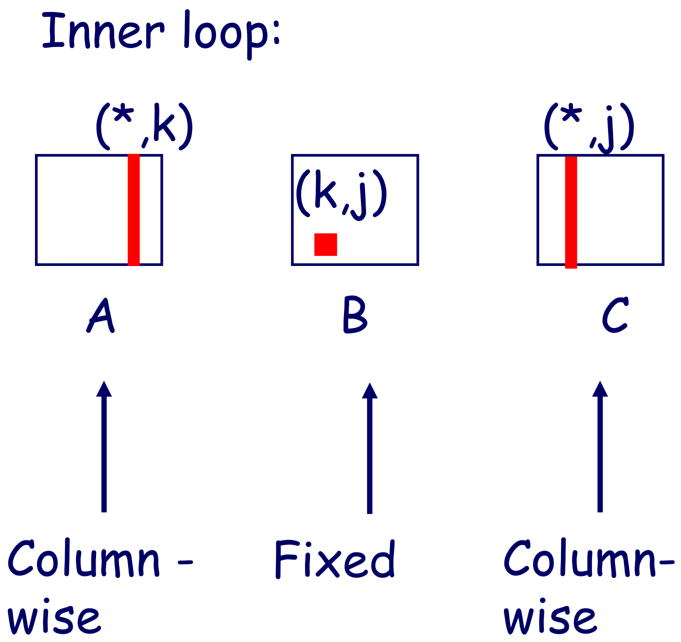


Misses per Inner Loop Iteration:

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;
  }
}
```

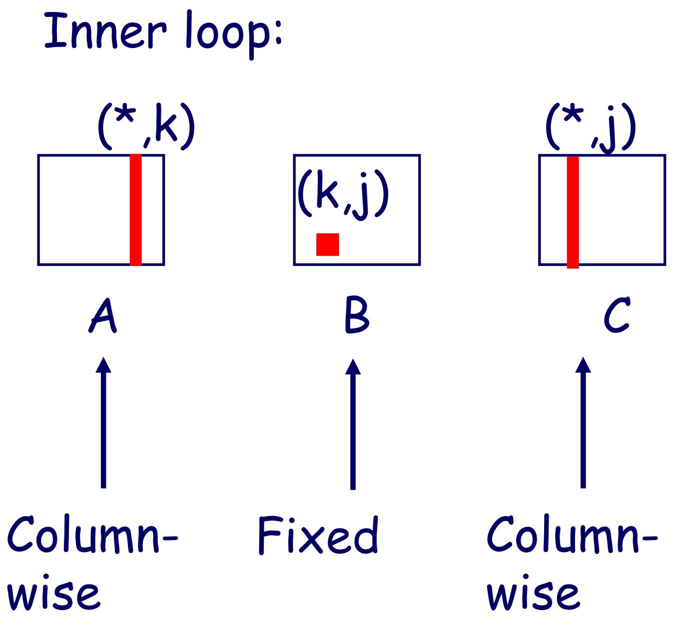


Misses per Inner Loop Iteration:

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;
  }
}
```



Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
 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;
 }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```
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];
 }
}
```

kij (& ikj):

- 2 loads, 1 store
- misses/iter = **0.5**

```
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;
 }
}
```

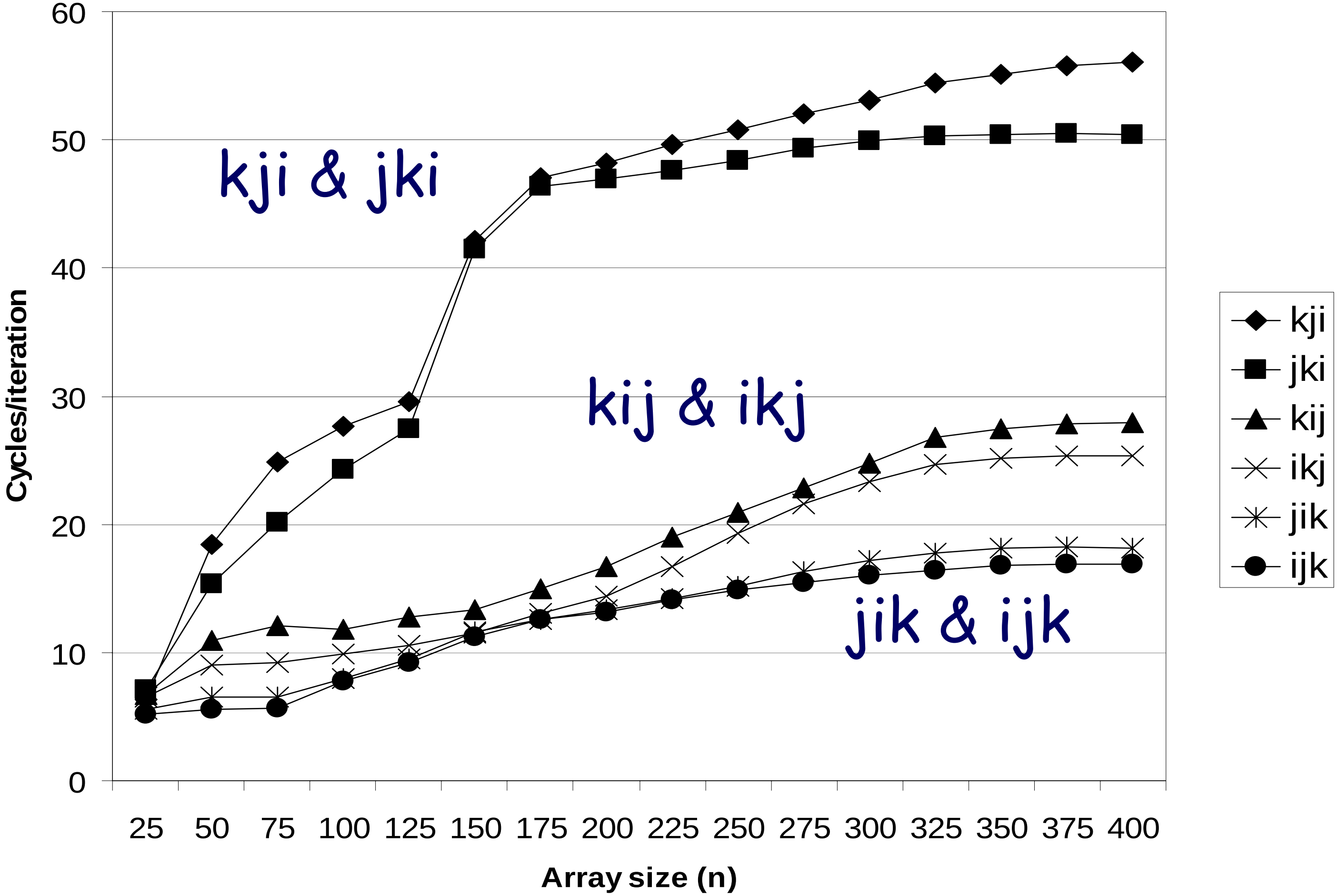
jki (& kji):

- 2 loads, 1 store
- misses/iter = **2.0**

Pentium Matrix Multiply Performance

Miss rates are helpful but not perfect predictors.

- Code scheduling matters, too.



Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

- **"block" (in this context) does not mean "cache block".**
- **Instead, it mean a sub-block within the matrix.**
- **Example: N = 8; sub-block size = 4**

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Key idea: Sub-blocks (i.e., $\mathbf{A_{xy}}$) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Blocked Matrix Multiply (bijk)

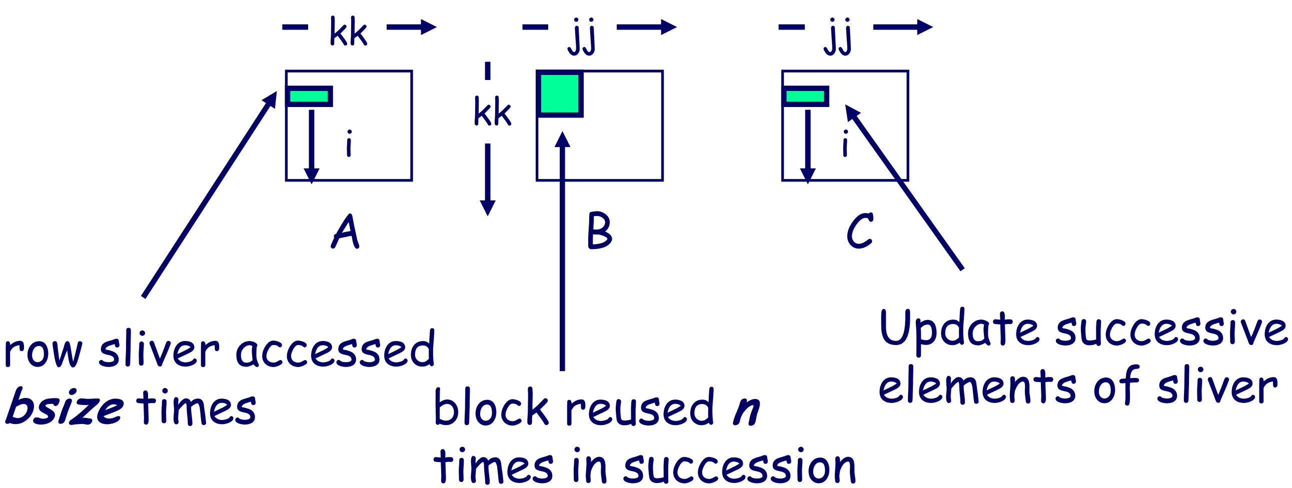
```
for (jj=0; jj<n; jj+=bsize) {

  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;

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

Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a *1 X bsize* sliver of *A* by a *bsize X bsize* block of *B* and accumulates into *1 X bsize* sliver of *C*
- Loop over *i* steps through *n* row slivers of *A* & *C*, using same *B*

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

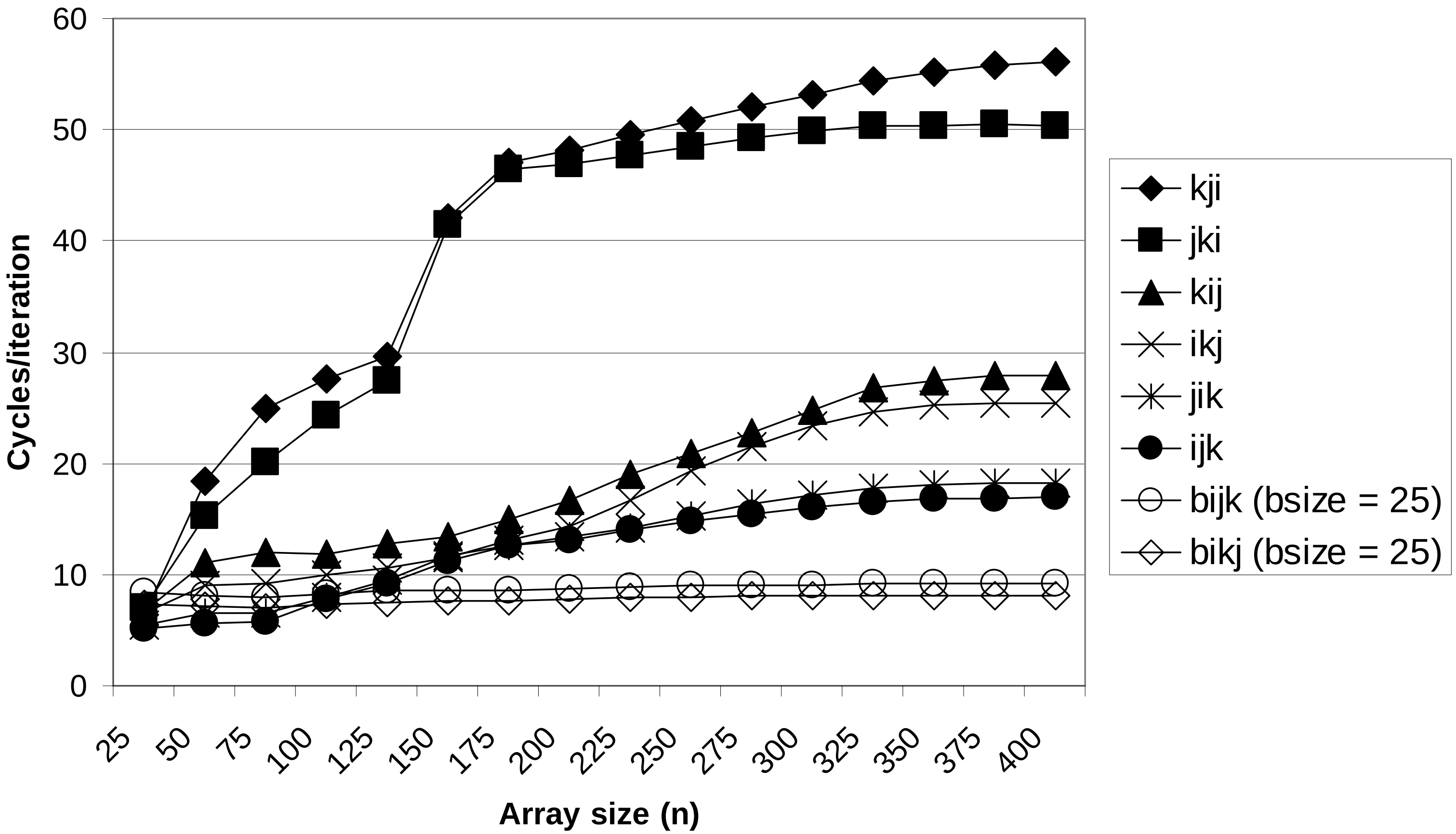
Innermost Loop Pair



Pentium Blocked Matrix Multiply Performance

Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)

- **relatively insensitive to array size.**



Concluding Observations

Programmer can optimize for cache performance

- **How data structures are organized**
- **How data are accessed**
  - **Nested loop structure**
  - **Blocking is a general technique**

All systems favor "cache friendly code"

- **Getting absolute optimum performance is very platform specific**
  - **Cache sizes, line sizes, associativities, etc.**
- **Can get most of the advantage with generic code**
  - **Keep working set reasonably small (temporal locality)**
  - **Use small strides (spatial locality)**

Cycle Counters

- Most modern systems have built in registers that are incremented every clock cycle
  - Very fine grained
  - Maintained as part of process state
    - In Linux, counts elapsed global time
- Special assembly code instruction to access
- On (recent model) Intel machines:
  - 64 bit counter.
  - RDTSC instruction sets %edx to high order 32-bits, %eax to low order 32-bits
- Aside: Is this a security issue?

Cycle Counter Period

Wrap Around Times for 550 MHz machine

- **Low order 32 bits wrap around every 2^{32} / (550 * 10^6) = 7.8 seconds**
- **High order 64 bits wrap around every 2^{64} / (550 * 10^6) = 33539534679 seconds**
  - **1065 years**

For 2 GHz machine

- **Low order 32 bits every 2.1 seconds**
- **High order 64 bits every 293 years**

Measuring with Cycle Counter

Idea

- Get current value of cycle counter
  - store as pair of unsigned's `cyc_hi` and `cyc_lo`
- Compute something
- Get new value of cycle counter
- Perform double precision subtraction to get elapsed cycles

```
/* Keep track of most recent reading of cycle counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

void start_counter()
{
  /* Get current value of cycle counter */
  access_counter(&cyc_hi, &cyc_lo);
}
```

Accessing the Cycle Counter

- **GCC allows inline assembly code with mechanism for matching registers with program variables**
- **Code only works on x86 machine compiling with GCC**

```
void access_counter(unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- **Emit assembly with `rdtsc` and two `movl` instructions**

Closer Look at Extended ASM

```
asm(“Instruction String"
       : Output List
       : Input List
       : Clobbers List);
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

Instruction String

- Series of assembly commands
  - Separated by “;” or “\n”
  - Use “%%” where normally would use “%”

Closer Look at Extended ASM

```
asm("Instruction String"
      : Output List
      : Input List
      : Clobbers L
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

Output List

- Expressions indicating destinations for values `%0`, `%1`, …, `%j`
  - Enclosed in parentheses
  - Must be *lvalue*
    - Value that can appear on LHS of assignment
- Tag `"=r"` indicates that symbolic value (`%0`, etc.), should be replaced by a register

Closer Look at Extended ASM

```
asm("Instruction String"
        : Output List
        : Input List
        : Clobbers L
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

Input List

- Series of expressions indicating sources for values `%j+1`, `%j+2`, …
  - Enclosed in parentheses
  - Any expression returning value
- Tag `"r"` indicates that symbolic value (`%0`, etc.) will come from register

Closer Look at Extended ASM

```
asm("Instruction String"
       : Output List
       : Input List
       : Clobbers L
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

Clobbers List

- List of register names that get altered by assembly instruction
- Compiler will make sure doesn't store something in one of these registers that must be preserved across `asm`
  - Value set before & used after

Completing Measurement

- **Get new value of cycle counter**
- **Perform double precision subtraction to get elapsed cycles**
- **Express as `double` to avoid overflow problems**

```
double get_counter()
{
  unsigned ncyc_hi, ncyc_lo
  unsigned hi, lo, borrow;
  /* Get cycle counter */
  access_counter(&ncyc_hi, &ncyc_lo);
  /* Do double precision subtraction */
  lo = ncyc_lo - cyc_lo;
  borrow = lo > ncyc_lo;
  hi = ncyc_hi - cyc_hi - borrow;
  return (double) hi * (1 << 30) * 4 + lo;
}
```

Timing With Cycle Counter

Determine Clock Rate of Processor

- Count number of cycles required for some fixed number of seconds

```
double MHZ;
int sleep_time = 10;
start_counter();
sleep(sleep_time);
MHZ = get_counter()/(sleep_time * 1e6);
```

Time Function P

- First attempt: Simply count cycles for one execution of P

```
double tsecs;
start_counter();
P();
tsecs = get_counter() / (MHZ * 1e6);
```

Example – testClock.c

```
#include <stdio.h>
#include "clock.h"

int main()
{

  double cycles, Mhz;

  Mhz = mhz(1);
  start_counter();
  sleep(2);
  cycles = get_counter();

  printf("cycles = %f, MHz = %f, cycles/Mhz = %f\n", cycles, Mhz, cycles/Mhz);
  printf("elapsed time = %f seconds \n", cycles/(1.0e6*Mhz));

  return 0;
}
```

```
Processor Clock Rate ~= 2673.5 MHz
cycles = 5343976388.000000, MHz = 2673.526339, cycles/Mhz = 1998849.351153
elapsed time = 1.998849 seconds
```

Measurement Pitfalls

Overhead

- **Calling `get_counter()` incurs small amount of overhead**
- **Want to measure long enough code sequence to compensate**

Dealing with Overhead & Cache Effects

- **Always execute function once to "warm up" cache**
- **Keep doubling number of times execute P() until reach some threshold**
  - **Used CMIN = 50000**

```
int cnt = 1;
double cmeas = 0;
double cycles;
do  {
  int c = cnt;
  P();               /* Warm up cache */
  get_counter();
  while (c-- > 0)
  P();
  cmeas = get_counter();
  cycles = cmeas / cnt;
  cnt += cnt;
} while (cmeas < CMIN);  /* Make sure have enough */
return cycles / (1e6 * MHZ);
```

Summary

- **Cache Memory can be used to improve performance**
- **Programmers can write code that takes advantage of caching**
- **Read Ch. 6 (Cache Memory 6.4)**