15-213

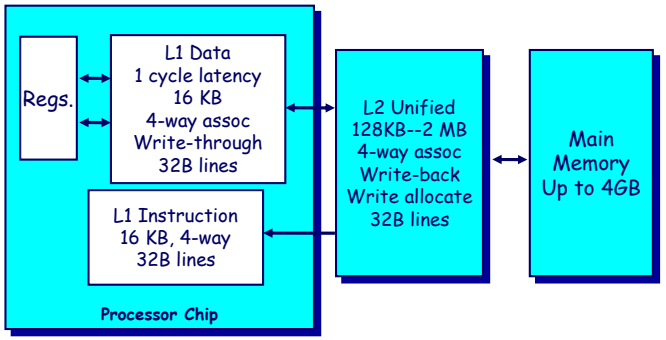
"The course that gives CMU its Zip!"

Cache Performance
October 3, 2007

Topics

- Impact of caches on performance
- The memory mountain

Intel Pentium III Cache Hierarchy



Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in cache (misses / references)
- Typical numbers:
  - 3-10% for L1
  - can be quite small (e.g., < 1%) for L2, depending on size, etc.

Hit Time

- Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- Typical numbers:
  - 1-2 clock cycle for L1
  - 5-20 clock cycles for L2

Aside for architects:
-Increasing cache size?
-Increasing block size?
-Increasing associativity?

Miss Penalty

- Additional time required because of a miss
  - Typically 50-200 cycles for main memory (Trend: increasing!)

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;
}
```

Miss rate = 1/4 = 25%

```
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 = 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.

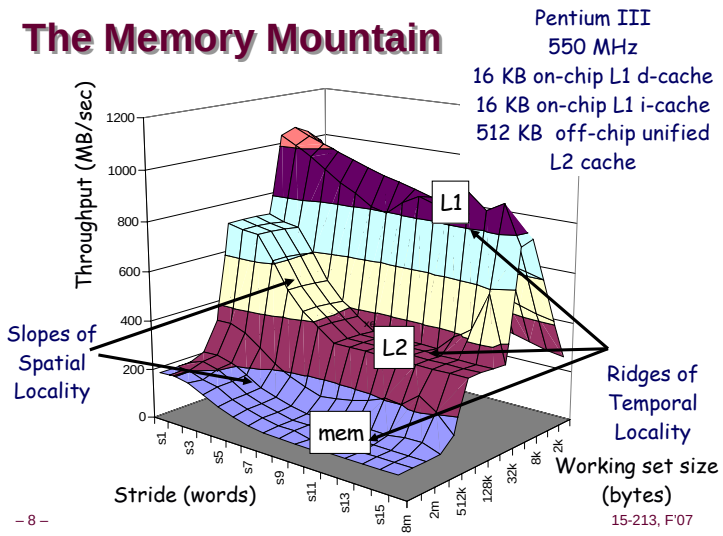
Memory Mountain Test Function

```
/* 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 */
}
```

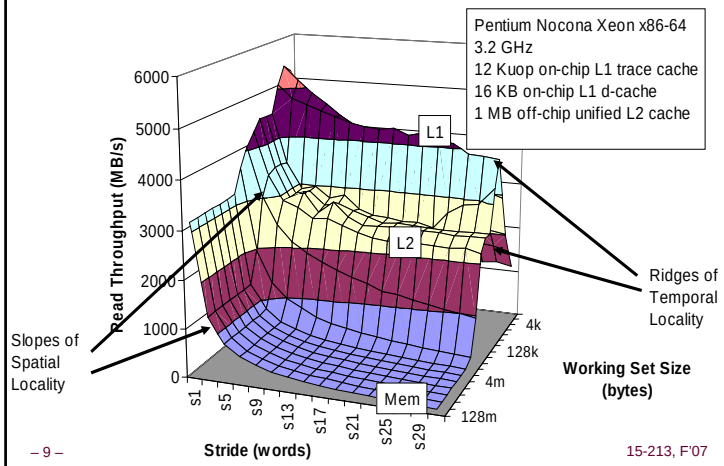

Memory Mountain Main Routine

```
/* 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);
}
```

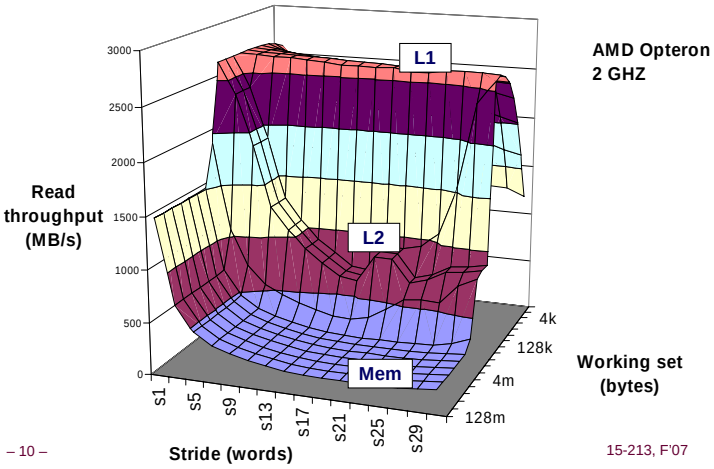

The Memory Mountain



X86-64 Memory Mountain



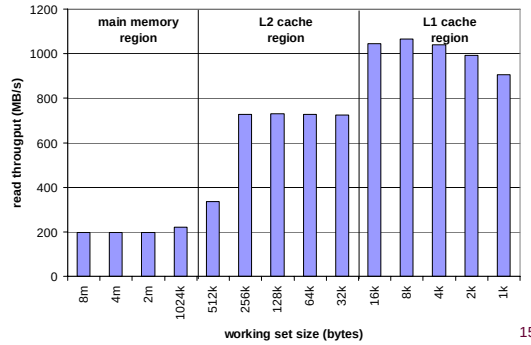
Opteron Memory Mountain



Ridges of Temporal Locality

Slice through the memory mountain with stride=1

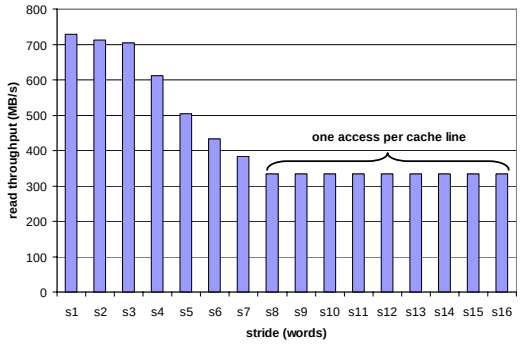
- **illuminates read throughputs of different caches and memory**



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(N3) total operations
- Accesses
  - N reads per source element
  - N values summed per destination
    - » but may be able to hold 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;
  }
}
```

Variable **sum** *held in register*

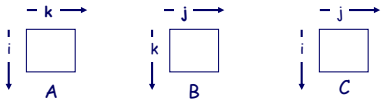
Miss Rate Analysis for Matrix Multiply

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:

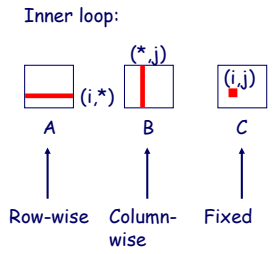
- `for (i = 0; i < N; i++)`
  `sum += a[0][i];`
- accesses successive elements
- if block size (B) > 4 bytes, exploit spatial locality
  - compulsory miss rate = 4 bytes / B

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;
  }
}
```

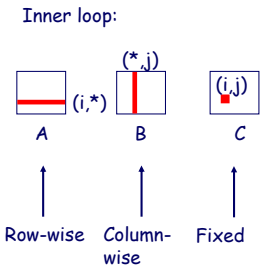


Misses per Inner Loop Iteration:

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
  }
}
```

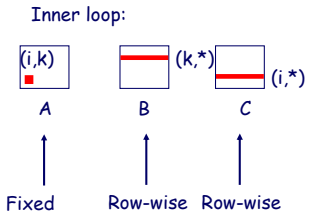


Misses per Inner Loop Iteration:

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];
  }
}
```

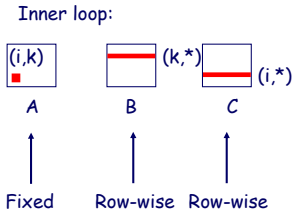


Misses per Inner Loop Iteration:

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];
  }
}
```

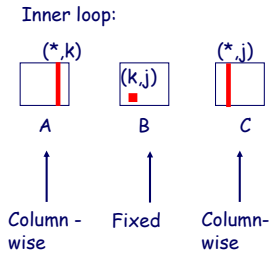


Misses per Inner Loop Iteration:

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;
  }
}
```

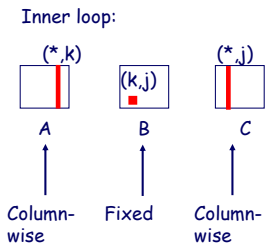


Misses per Inner Loop Iteration:

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**

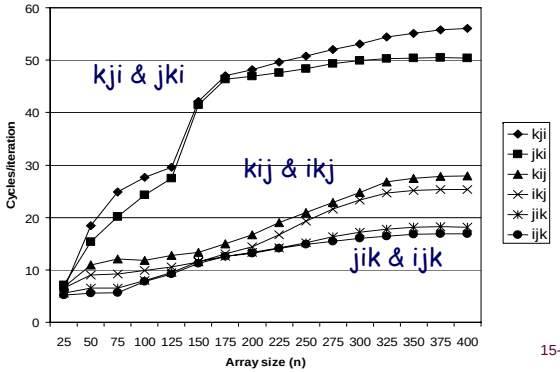
```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

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)

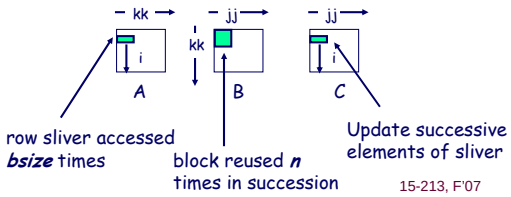
```
for (jj=0; jj<n; jj+=bsize) {

  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;
  }
```

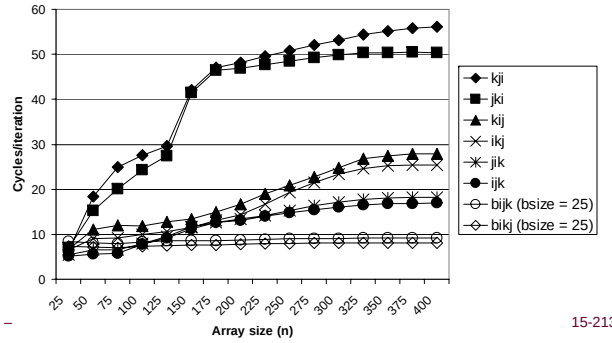
Innermost Loop Pair



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"

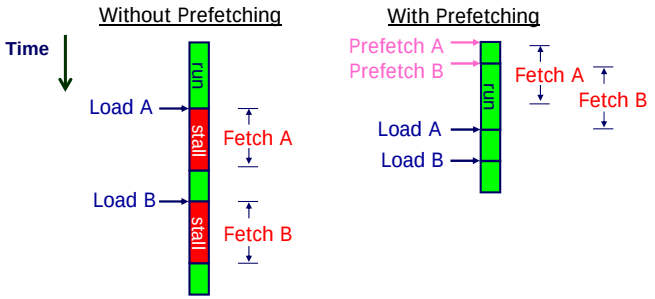
- Getting absolute optimum performance is very platform specific
  - Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)

Tolerating Cache Miss Latencies

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    sum += A[i][j];
```

What can the hardware do to avoid miss penalties in this case?

Prefetching: Start moving data close to the processor before it is needed



Prefetching allows cache misses to be overlapped with:

- computation, and
- other cache misses.

Prefetches can be issued either by Software or by Hardware

Software-Controlled Prefetching:

- programmer or compiler inserts explicit prefetch instructions
  - e.g., `prefetch 8(%eax)`

Hardware-Controlled Prefetching:

- the hardware tries to recognize strided memory accesses

Our fish machines support both forms of prefetching

Advantages/disadvantages of each approach?

Prefetches vs. Memory Loads

Similarities:

- both are given a data address as an argument
- if that location is not in the L1 data cache, then a cache miss is triggered, and the data is moved into the cache

Differences:

- prefetches do not have a register destination
  - Hence they are "non-binding"
- prefetches are non-blocking
  - i.e. the processor does not stall: it keeps executing
- prefetches do not trigger memory exceptions
  - it is ok to prefetch invalid memory addresses

Software Pipelining Prefetches

Original Loop:

```
for (i = 0; i < 1024; i++)
    a[i][0] = 0;
```

Software Pipelined Loop (prefetching 16 iters ahead):

```
for (i = 0; i < 16; i++)          /* Prolog */
    prefetch(&a[i][0]);

for (i = 0; i < 1008; i++) {      /* Steady State */
    prefetch(&a[i+16][0]);
    a[i][0] = 0;
}

for (i = 1008; i < 1024; i++)     /* Epilog */
    a[i][0] = 0;
```


Reducing Software Overhead

Original Loop:

```
for (i = 0; i < 1024; i++)
    a[i] = 0;
```

← Spatial Locality (4B elements, 32B lines)

Unrolled Steady-State Loop:

```
for (i = 0; i < 1008; i+=8) {
    prefetch(&a[i+16][0]);
    a[i][0]   = 0;
    a[i+1][0] = 0;
    a[i+2][0] = 0;
    a[i+3][0] = 0;
    a[i+4][0] = 0;
    a[i+5][0] = 0;
    a[i+6][0] = 0;
    a[i+7][0] = 0;
}
```


Reducing Software Overhead

Original Loop:

```
for (i = 0; i < 1024; i++)
    a[i] = 0;
```

← Spatial Locality (4B elements, 32B lines)

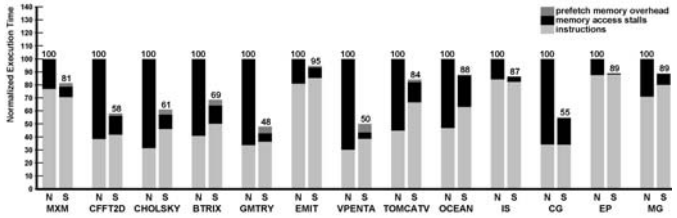
"Strip-Mined" Steady-State Loop:

```
for (i = 0; i < 1008; i+=8) {
    prefetch(&a[i+16][0]);
    for (i2 = i; i2 < i+8; i2++)
        a[i2][0] = 0;
}
```


More on Software Prefetching

Compiler algorithm for inserting prefetches:

- Todd C. Mowry, Monica Lam and Anoop Gupta. *"Design and Evaluation of a Compiler Algorithm for Prefetching"*, in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1992.



Info on prefetch support in gcc can be found here:

- http://gcc.gnu.org/projects/prefetch.html

Hardware Prefetching on the Pentium 4

The hardware attempts to detect repeated patterns of misses to consecutive caches lines

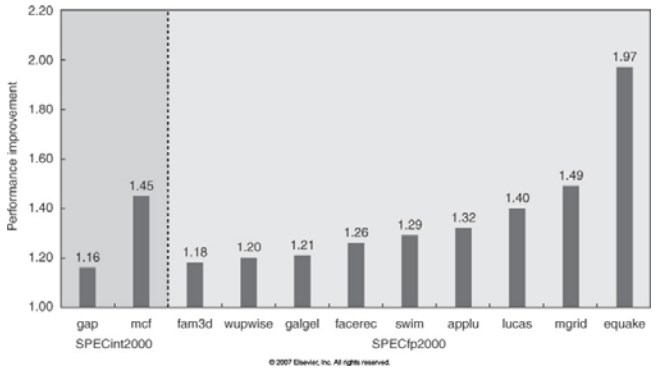
- the hardware must see several iterations first before it recognizes this pattern
- if the loop contains too many different memory accesses, it may not recognize the pattern

Once the pattern is recognized, the hardware then attempts to prefetch 2 cache lines ahead

- it will not prefetch beyond 4KB page boundaries
- it will continue prefetching beyond the end of the loop

Word of caution: hardware and software prefetching may interfere with each other

Hardware Prefetching Speedup on an Intel Pentium 4



(from Hennessy & Patterson, CA:AQQ, 4th Edition)

Prefetching Summary

Limitations of Prefetching:

- requires that access patterns be predictable
  - easy for array accesses, more difficult for pointer chasing
- performance is ultimately limited by memory hierarchy bandwidth
  - but it is easier to increase bandwidth than to reduce latency!

Prefetching is generally more applicable than "cache blocking"

Many commercial compilers now insert software prefetches, and many processors support some form of stride-1 hardware prefetching

Prefetching is a valuable technique for hiding large memory access latencies