

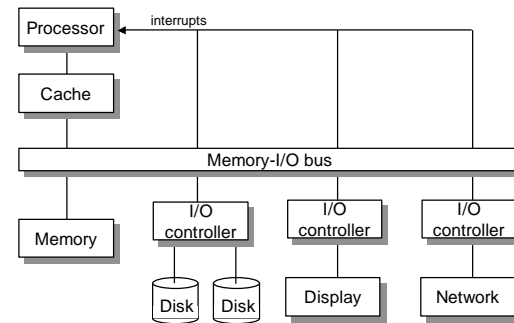
The Memory Hierarchy CS 740

Sept. 28, 2001

Topics

- The memory hierarchy
- Cache design

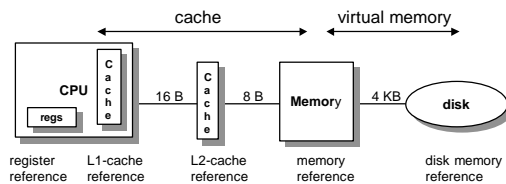
Computer System



- 2 -

CS 740 F'01

The Tradeoff



	register reference	L1-cache reference	L2-cache reference	memory reference	disk memory reference
size:	608 B	128k B	512kB - 4MB	128 MB	27GB
speed:	1.4 ns	4.2 ns	16.8 ns	112 ns	9 ms
\$/Mbyte:			\$90/MB	\$2-6/MB	\$0.01/MB
block size:	4 B	4 B	16 B	4-8 KB	

larger, slower, cheaper

(Numbers are for a 21264 at 700MHz)

- 3 -

CS 740 F'01

Why is bigger slower?

- Physics slows us down
- Racing the speed of light: ($3.0 \times 10^8 \text{ m/s}$)
 - clock = 500MHz
 - how far can I go in a clock cycle?
 - $(3.0 \times 10^8 \text{ m/s}) / (500 \times 10^6 \text{ cycles/s}) = 0.6 \text{ m/cycle}$
 - For comparison: 21264 is about 17mm across
- Capacitance:
 - long wires have more capacitance
 - either more powerful (bigger) transistors required, or slower
 - signal propagation speed proportional to capacitance
 - going "off chip" has an order of magnitude more capacitance

- 4 -

CS 740 F'01

Alpha 21164 Chip Photo

Microprocessor
Report 9/12/94

Caches:
L1 data
L1 instruction
L2 unified
+ L3 off-chip

- 5 -

CS 740 F01

Alpha 21164 Chip Caches

Caches:
L1 data
L1 instruction
L2 unified
+ L3 off-chip

- 6 -

CS 740 F01

Locality of Reference

Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently.
- Temporal locality:** recently referenced items are likely to be referenced in the near future.
- Spatial locality:** items with nearby addresses tend to be referenced close together in time.

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

Locality in Example:

- Data**
 - Reference array elements in succession (spatial)
- Instructions**
 - Reference instructions in sequence (spatial)
 - Cycle through loop repeatedly (temporal)

- 7 -

CS 740 F01

Caching: The Basic Idea

Main Memory

- Stores words
A-Z in example

Cache

- Stores subset of the words
4 in example
- Organized in lines
 - Multiple words
 - To exploit spatial locality

Access

- Word must be in cache for processor to access

Processor

Small, Fast Cache

A
B
G
H

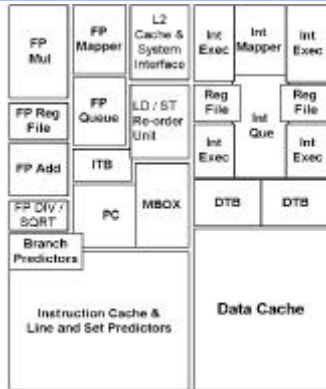
Big, Slow Memory

A
B
C
⋮
Y
Z

- 8 -

CS 740 F01

How important are caches?



- 21264 Floorplan
- Register files in middle of execution units
- 64k instr cache
- 64k data cache
- Caches take up a large fraction of the die

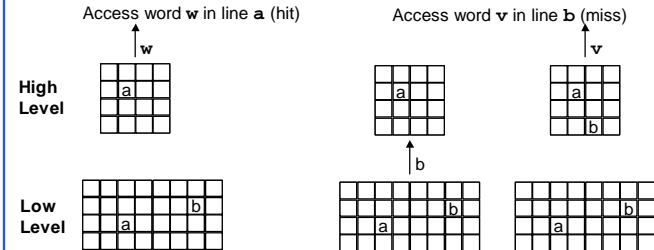
(Figure from Jim Keller, Compaq Corp.)

- 9 -

CS 740 F01

Accessing Data in Memory Hierarchy

- Between any two levels, memory is divided into *lines* (aka "blocks")
- Data moves between levels on demand, in line-sized chunks
- Invisible to application programmer
 - Hardware responsible for cache operation
- Upper-level lines a subset of lower-level lines



- 10 -

CS 740 F01

Design Issues for Caches

Key Questions:

- Where should a line be placed in the cache? (line placement)
- How is a line found in the cache? (line identification)
- Which line should be replaced on a miss? (line replacement)
- What happens on a write? (write strategy)

Constraints:

- Design must be very simple
 - Hardware realization
 - All decision making within nanosecond time scale
- Want to optimize performance for "typical" programs
 - Do extensive benchmarking and simulations
 - Many subtle engineering tradeoffs

- 11 -

CS 740 F01

Direct-Mapped Caches

Simplest Design

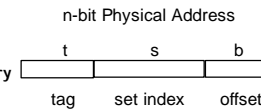
- Each memory line has a unique cache location

Parameters

- Line (aka block) size $B = 2^b$
 - Number of bytes in each line
 - Typically $2X-8X$ word size
- Number of Sets $S = 2^s$
 - Number of lines cache can hold
- Total Cache Size = $B * S = 2^{b+s}$

Physical Address

- Address used to reference main memory
- n bits to reference $N = 2^n$ total bytes
- Partition into fields
 - Offset: Lower b bits indicate which byte within line
 - Set: Next s bits indicate how to locate line within cache
 - Tag: t identifies this line when in cache

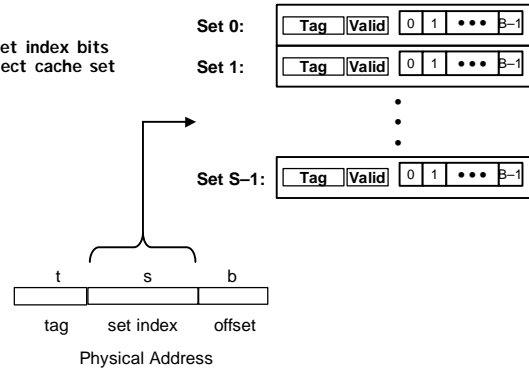


- 12 -

CS 740 F01

Indexing into Direct-Mapped Cache

- Use set index bits to select cache set



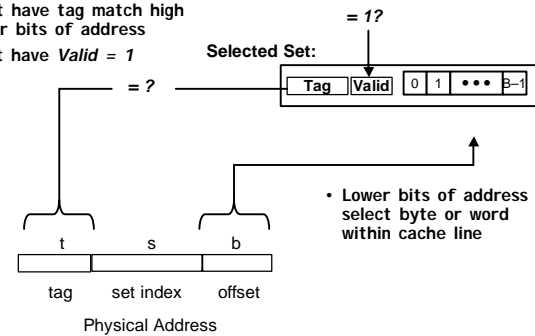
- 13 -

CS 740 F'01

Direct-Mapped Cache Tag Matching

Identifying Line

- Must have tag match high order bits of address
- Must have *Valid* = 1



- 14 -

CS 740 F'01

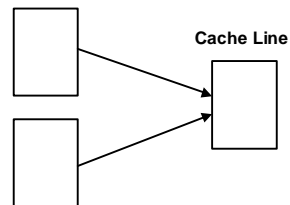
Properties of Direct Mapped Caches

Strength

- Minimal control hardware overhead
- Simple design
- (Relatively) easy to make fast

Weakness

- Vulnerable to thrashing
- Two heavily used lines have same cache index
- Repeatedly evict one to make room for other



- 15 -

CS 740 F'01

Vector Product Example

```
float dot_prod(float x[1024], y[1024])
{
    float sum = 0.0;
    int i;
    for (i = 0; i < 1024; i++)
        sum += x[i]*y[i];
    return sum;
}
```

Machine

- DECStation 5000
- MIPS Processor with 64KB direct-mapped cache, 16 B line size

Performance

- Good case: 24 cycles / element
- Bad case: 66 cycles / element

- 16 -

CS 740 F'01

Thrashing Example

x[0]
x[1]
x[2]
x[3]

⋮

x[1020]
x[1021]
x[1022]
x[1023]

y[0]
y[1]
y[2]
y[3]

⋮

y[1020]
y[1021]
y[1022]
y[1023]

• Access one element from each array per iteration

-17 - CS 740 F'01

Thrashing Example: Good Case

x[0]
x[1]
x[2]
x[3]

y[0]
y[1]
y[2]
y[3]

Cache Line

Access Sequence

- Read x[0]
 - x[0], x[1], x[2], x[3] loaded
- Read y[0]
 - y[0], y[1], y[2], y[3] loaded
- Read x[1]
 - Hit
- Read y[1]
 - Hit
- ⋮
- 2 misses / 8 reads

Analysis

- x[i] and y[i] map to different cache lines
- Miss rate = 25%
 - Two memory accesses / iteration
 - On every 4th iteration have two misses

Timing

- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration = $10 + 0.25 * 2 * 28$

-18 - CS 740 F'01

Thrashing Example: Bad Case

x[0]
x[1]
x[2]
x[3]

y[0]
y[1]
y[2]
y[3]

Cache Line

Access Pattern

- Read x[0]
 - x[0], x[1], x[2], x[3] loaded
- Read y[0]
 - y[0], y[1], y[2], y[3] loaded
- Read x[1]
 - x[0], x[1], x[2], x[3] loaded
- Read y[1]
 - y[0], y[1], y[2], y[3] loaded
- ⋮
- 8 misses / 8 reads

Analysis

- x[i] and y[i] map to same cache lines
- Miss rate = 100%
 - Two memory accesses / iteration
 - On every iteration have two misses

Timing

- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration = $10 + 1.0 * 2 * 28$

-19 - CS 740 F'01

Set Associative Cache

Mapping of Memory Lines

- Each set can hold E lines (usually E=2-8)
- Given memory line can map to any entry within its given set

Eviction Policy

- Which line gets kicked out when bring new line in
- Commonly either "Least Recently Used" (LRU) or pseudo-random
 - LRU: least-recently accessed (read or written) line gets evicted

LRU State

Set i:

Line 0: Tag Valid 0 1 ⋮ β-1

Line 1: Tag Valid 0 1 ⋮ β-1

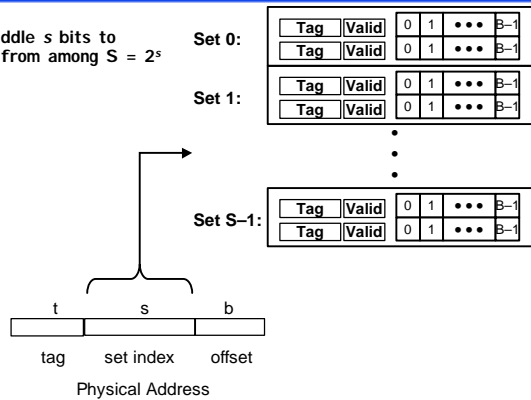
⋮

Line E-1: Tag Valid 0 1 ⋮ β-1

-20 - CS 740 F'01

Indexing into 2-Way Associative Cache

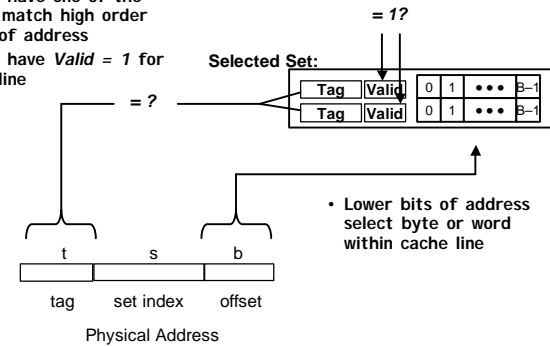
- Use middle s bits to select from among $S = 2^s$ sets



Associative Cache Tag Matching

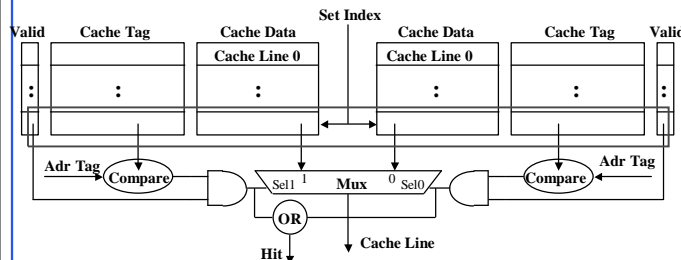
Identifying Line

- Must have one of the tags match high order bits of address
- Must have *Valid* = 1 for this line



Two-Way Set Associative Cache Implementation

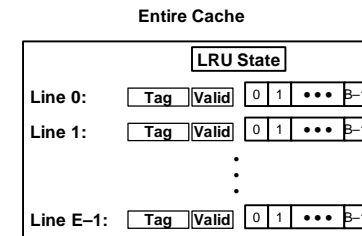
- Set index selects a set from the cache
- The two tags in the set are compared in parallel
- Data is selected based on the tag result



Fully Associative Cache

Mapping of Memory Lines

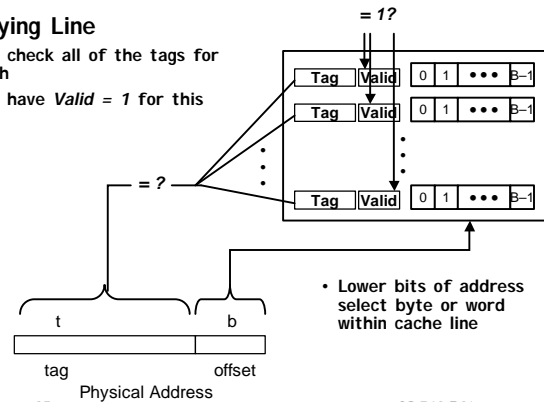
- Cache consists of single set holding E lines
- Given memory line can map to any line in set
- Only practical for small caches



Fully Associative Cache Tag Matching

Identifying Line

- Must check all of the tags for match
- Must have *Valid* = 1 for this line



- 25 -

CS 740 F'01

Replacement Algorithms

- *When a block is fetched, which block in the target set should be replaced?*

Optimal algorithm:

- replace the block that will not be used for the longest period of time
- must know the future

Usage based algorithms:

- **Least recently used (LRU)**
 - replace the block that has been referenced least recently
 - hard to implement

Non-usage based algorithms:

- **First-in First-out (FIFO)**
 - treat the set as a circular queue, replace block at head of queue.
 - easy to implement
- **Random (RAND)**
 - replace a random block in the set
 - even easier to implement

- 26 -

CS 740 F'01

Implementing RAND and FIFO

FIFO:

- maintain a modulo E counter for each set.
- counter in each set points to next block for replacement.
- increment counter with each replacement.

RAND:

- maintain a single modulo E counter.
- counter points to next block for replacement in any set.
- increment counter according to some schedule:
 - each clock cycle,
 - each memory reference, or
 - each replacement anywhere in the cache.

LRU

- Need state machine for each set
- Encodes usage ordering of each element in set
- E! possibilities ==> - E log E bits of state

- 27 -

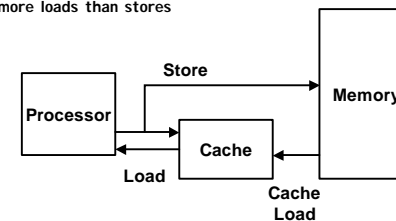
CS 740 F'01

Write Policy

- What happens when processor writes to the cache?
- Should memory be updated as well?

Write Through:

- Store by processor updates cache *and* memory
- Memory always consistent with cache
- Never need to store from cache to memory
- ~2X more loads than stores



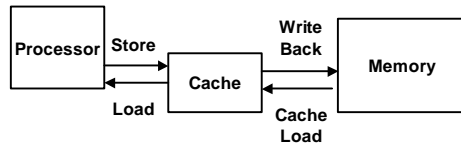
- 28 -

CS 740 F'01

Write Policy (Cont.)

Write Back:

- Store by processor only updates cache line
- Modified line written to memory only when it is evicted
 - Requires “dirty bit” for each line
 - » Set when line in cache is modified
 - » Indicates that line in memory is stale
- Memory not always consistent with cache



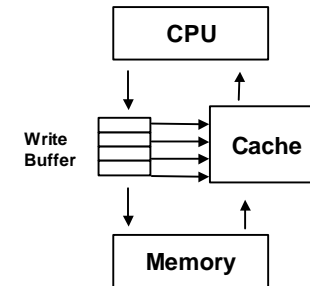
- 29 -

CS 740 F'01

Write Buffering

Write Buffer

- Common optimization for write-through caches
- Overlaps memory updates with processor execution
- Read operation must check write buffer for matching address

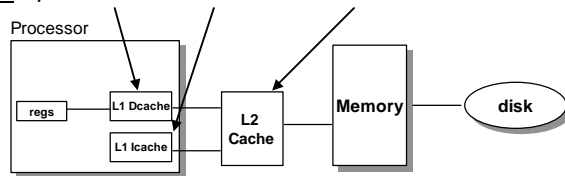


- 30 -

CS 740 F'01

Multi-Level Caches

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



How does this affect self modifying code?

- 31 -

CS 740 F'01

Bandwidth Matching

Challenge

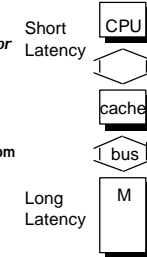
- CPU works with short cycle times
- DRAM (relatively) long cycle times
- *How can we provide enough bandwidth between processor & memory?*

Effect of Caching

- Caching greatly reduces amount of traffic to main memory
- But, sometimes need to move large amounts of data from memory into cache

Trends

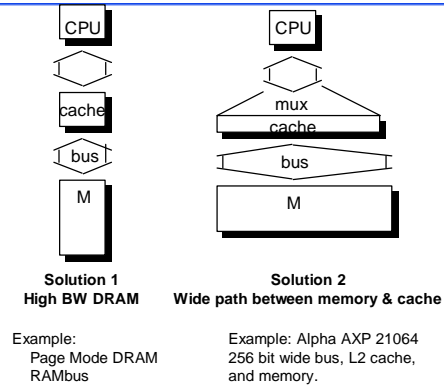
- Need for high bandwidth much greater for multimedia applications
 - Repeated operations on image data
- Recent generation machines (e.g., Pentium II) greatly improve on predecessors



- 32 -

CS 740 F'01

High Bandwidth Memory Systems



- 33 -

CS 740 F'01

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:
 - 1-3 clock cycles for L1
 - 3-12 clock cycles for L2

Miss Penalty

- additional time required because of a miss
 - Typically 25-100 cycles for main memory

- 34 -

CS 740 F'01

Impact of Cache and Block Size

Cache Size

- Effect on miss rate?
- Effect on hit time?

Block Size

- Effect on miss rate?
- Effect on miss penalty?

- 35 -

CS 740 F'01

Impact of Associativity

- Direct-mapped, set associative, or fully associative?

Total Cache Size (tags+data)?

Miss rate?

Hit time?

Miss Penalty?

- 36 -

CS 740 F'01

Impact of Replacement Strategy

- RAND, FIFO, or LRU?
- Total Cache Size (tags+data)?**

Miss Rate?

Miss Penalty?

- 37 -

CS 740 F'01

Impact of Write Strategy

- Write-through or write-back?
- Advantages of Write Through?**

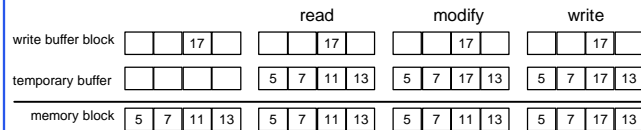
Advantages of Write Back?

- 38 -

CS 740 F'01

Allocation Strategies

- On a write miss, is the block loaded from memory into the cache?
- Write Allocate:**
- Block is loaded into cache on a write miss.
 - Usually used with write back
 - Otherwise, write-back requires read-modify-write to replace word within block



- But if you've gone to the trouble of reading the entire block, why not load it in cache?

- 39 -

CS 740 F'01

Allocation Strategies (Cont.)

- On a write miss, is the block loaded from memory into the cache?
- No-Write Allocate (Write Around):**
- Block is not loaded into cache on a write miss
 - Usually used with write through
 - Memory system directly handles word-level writes

- 40 -

CS 740 F'01

Qualitative Cache Performance Model

Miss Types

- **Compulsory ("Cold Start") Misses**
 - First access to line not in cache
- **Capacity Misses**
 - Active portion of memory exceeds cache size
- **Conflict Misses**
 - Active portion of address space fits in cache, but too many lines map to same cache entry
 - Direct mapped and set associative placement only
- **Validation Misses**
 - Block invalidated by multiprocessor cache coherence mechanism

Hit Types

- **Reuse hit**
 - Accessing same word that previously accessed
- **Line hit**
 - Accessing word spatially near previously accessed word

– 41 –

CS 740 F'01

Interactions Between Program & Cache

Major Cache Effects to Consider

- **Total cache size**
 - Try to keep heavily used data in highest level cache
- **Block size (sometimes referred to "line size")**
 - Exploit spatial locality

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;
  }
}
    
```

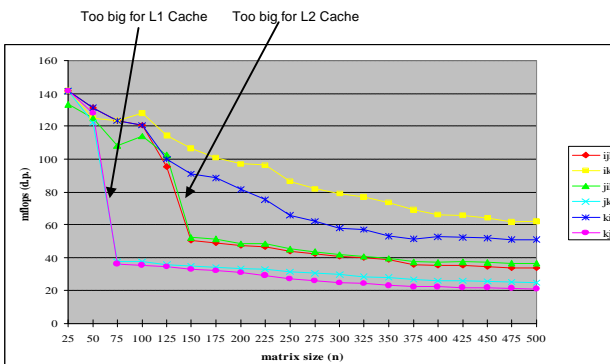
Example Application

- **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

– 42 –

CS 740 F'01

Matmult Performance (Alpha 21164)



– 43 –

CS 740 F'01

Block Matrix Multiplication

Example n=8, B = 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., A_{ij}) can be treated just like scalars.

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

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

– 44 –

CS 740 F'01

Blocked Matrix Multiply (bijk)

```

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

```

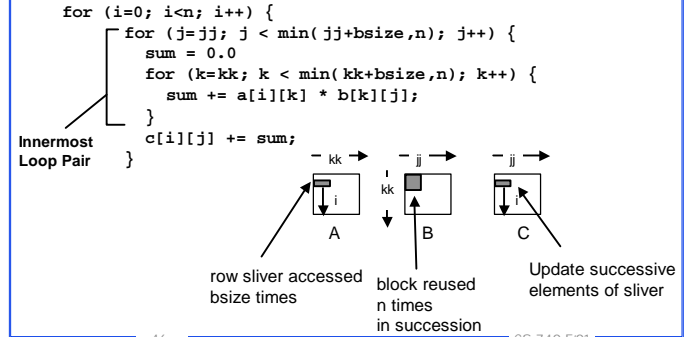
Warning: Code in H&P (p. 409) has bugs!

- 45 -

CS 740 F'01

Blocked Matrix Multiply Analysis

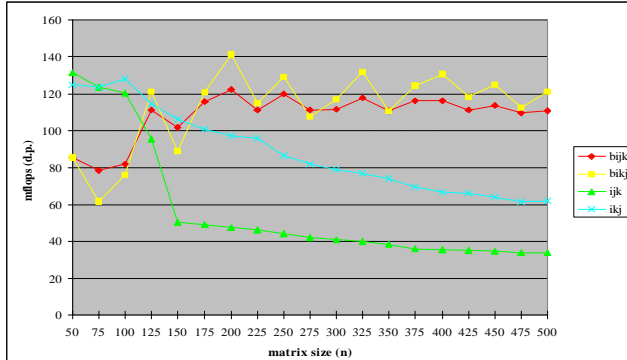
- Innermost loop pair multiplies 1 X bsize sliver of A times bsize X bsize block of B and accumulates into 1 X bsize sliver of C
- Loop over i steps through n row slivers of A & C, using same B



- 46 -

CS 740 F'01

Blocked matmult perf (Alpha 21164)



- 47 -

CS 740 F'01