Computer Systems Architecture CMSC 411 Unit 5 – Memory Hierarchy

Alan Sussman
April 15, 2003

Administrivia

- Quiz next Tuesday, April 22
- Homework 4b due Thursday
- Project out by tomorrow
  - due May 9
- Read Chapter 5
  - concentrate on 5.1-5.12
- Midterm questions?

Last time

- Global code scheduling
  - to move instructions across branches, and preserve data and control dependences, and exception behavior
  - trace scheduling – *selection* and *compaction*
  - superblocks – like trace, but only 1 entry point
- Hardware support
  - predicated instructions – convert control into data dependences
  - help with exception handling for global code motion
  - address conflict resolution for reordering loads/stores
    - speculative loads

Cache Memory

Issues to consider

- How big should the fastest memory (cache memory) be?
- How do we decide what to put in cache memory?
- If the cache is full, how do we decide what to remove?
- How do we find something in cache?
- How do we handle writes?

First, there is main memory

- Jargon:
  - frame address – which page?
  - block number – which cache block?
  - contents – the data

Then add a cache

- Jargon: Each address of a memory location is partitioned into
  - block address
    - tag
    - index
  - block offset

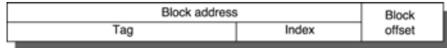


Fig. 5.5

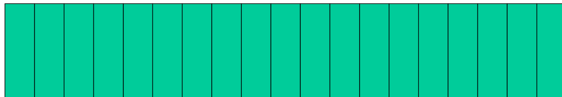
© 2003 Elsevier Science (USA). All rights reserved.

How does cache memory work?

- The following slides discuss:
  - what cache memory is
  - three organizations for cache memory
    - direct mapped.
    - set associative
    - fully associative
  - how the bookkeeping is done
- **Important note**: All addresses shown are in octal. Addresses in the book are usually decimal.

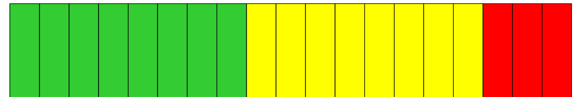
What is cache memory? Main memory first

Main memory is divided into (cache) blocks.
Each block contains many words (16-64 common now).



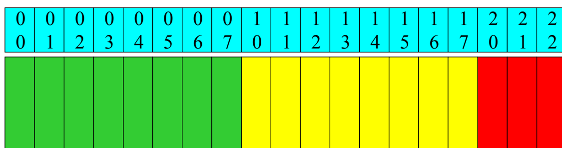
Main memory

Blocks are grouped into frames (pages), 3 frames in this picture.



Main memory (cont.)

Blocks are addressed by their frame number, and their block number within the frame.



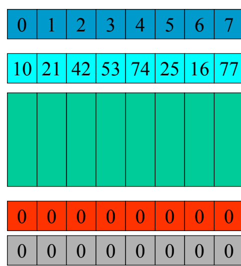
Cache memory

Cache has many, MANY fewer blocks than main memory, each with

a **block number**,	0	1	2	3	4	5	6	7
a **memory address**,	10	21	42	53	74	25	16	77
data,								
a **valid** bit,	0	0	0	0	0	0	0	0
a **dirty** bit.	0	0	0	0	0	0	0	0

Cache memory (cont.)

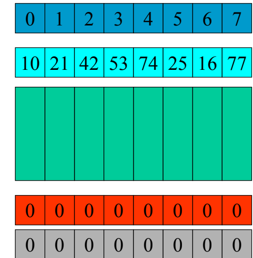
Initially, all the **valid** bits set to zero.



Cache memory (cont.)

Suppose want to load block 14 (octal) from memory into cache.

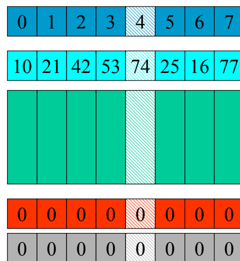
Three ways to organize cache
- direct mapped
- set associative
- fully associative



Direct mapped cache

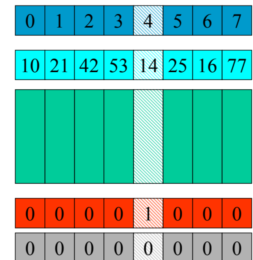
In **direct mapped cache,** block 14 can only be put in the cache block with address 4.

So the cache will no longer hold the block with memory address 74.



Direct mapped cache (cont.)

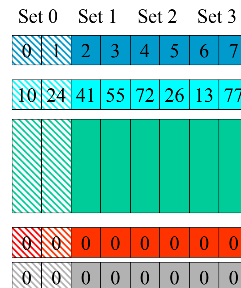
After the load, the contents look like this.



Set associative cache

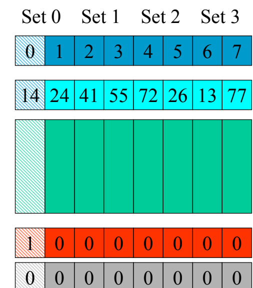
In **set associative cache**, each memory block can be put in any of a set of possible blocks in cache.

For example, if divide cache into 4 sets, block 14 can be put in any block in Set 0 (since last two bits of 14 octal are zero).



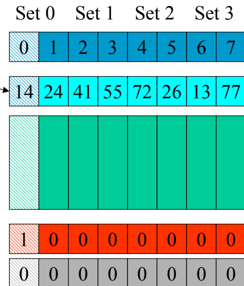
Set associative cache (cont.)

So after loading the block, cache memory might look like this.



Set associative cache (cont.)

Note that the last two bits of the memory block's address always match the set number, so do not need to be stored. This part of the address is called the **index**. The higher order bits are stored, and are called the **tag.**
In these pictures, both index and tag shown.



Set associative cache replacement

- Which entry in the set to replace?
- Three common choices:
  - Replace an eligible *random* block
  - Replace the least recently used (LRU) block
    - can be hard to keep track of, so often only approximated
  - Replace the oldest eligible block (First In, First Out, or FIFO)

Data cache replacement – Fig. 5.6

SPEC2000, in misses per 1000 instructions

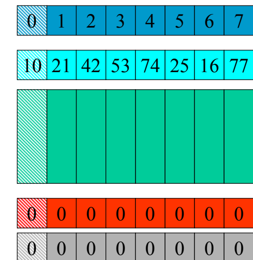
Set associativity

	Two-way			Four-way			Eight-Way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Fully associative cache

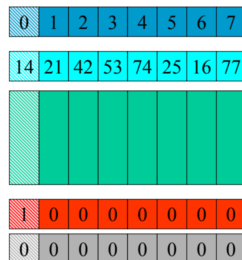
In **fully associative** cache, memory blocks may be stored anywhere.

So block 14 might be put in the first available block -- one with **valid** = 0.



Fully associative cache (cont.)

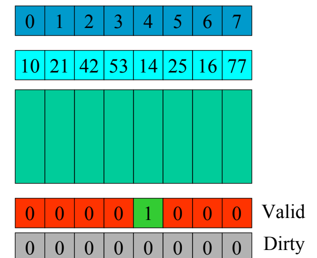
With this result.



Managing cache

Use direct mapped cache as an example.

After first read operation, cache memory looked like this.



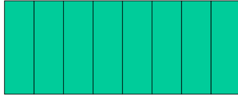
Managing cache (cont.)

If all other memory references involved block 14, no other blocks would need to be fetched from memory.

But suppose eventually need to fetch blocks 10, 31 and 66.

Need to fetch all three, because don't have valid versions of them.

0	1	2	3	4	5	6	7
10	21	42	53	14	25	16	77



0	0	0	0	1	0	0	0	Valid
0	0	0	0	0	0	0	0	Dirty

Computer Systems Architecture CMSC 411 Unit 5 – Memory Hierarchy

Alan Sussman

April 17, 2003

Administrivia

- Quiz Tuesday, April 22
- Homework 4b due today
- HW 5 out soon
- Project out - due May 9
  - questions?

Last time

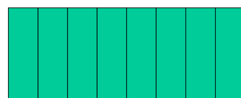
- Cache memory
  - Partition an address into a block address (tag and index) and a block offset
  - Cache holds blocks, located via the tag, with a valid and dirty bit for each block
  - Direct mapped cache has exactly one cache location for each block
  - Set associative cache has a set of possible locations for a block (*N* for *N*-way set associative – index part of address determines which set)
    - replacement choices: random, LRU, FIFO
  - Fully associative cache allows a block to go into any cache location

Managing cache

Use direct mapped cache as an example.

After first read operation, cache memory looked like this.

0	1	2	3	4	5	6	7
10	21	42	53	14	25	16	77



0	0	0	0	1	0	0	0	Valid
0	0	0	0	0	0	0	0	Dirty

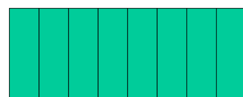
Managing cache (cont.)

If all other memory references involved block 14, no other blocks would need to be fetched from memory.

But suppose eventually need to fetch blocks 10, 31 and 66.

Need to fetch all three, because don't have valid versions of them.

0	1	2	3	4	5	6	7
10	21	42	53	14	25	16	77

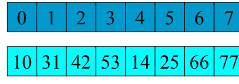


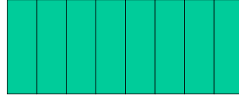
0	0	0	0	1	0	0	0	Valid
0	0	0	0	0	0	0	0	Dirty

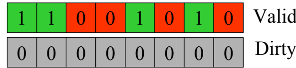
Managing cache (cont.)

The result looks like this.

Now suppose **write** to block 66.



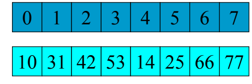


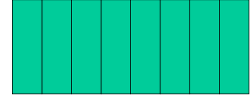


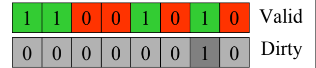
Managing cache (cont.)

The block is valid in cache, so don't need to fetch.

But the **write** operation sets the **dirty** bit for that block.



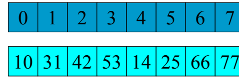


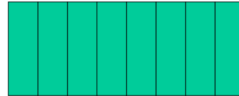


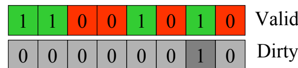
Managing cache (cont.)

The **write** operation sets the **dirty** bit for that block.

That means that the cached block is different from the memory block, so must eventually be written back.



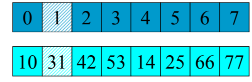


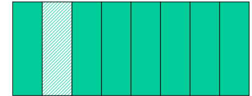


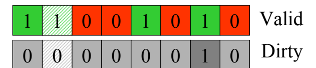
Managing cache (cont.)

Now suppose need to **read** from a block not in cache.

If it is block 41, then must overwrite block 31.



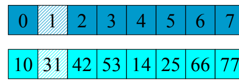


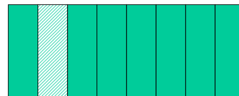


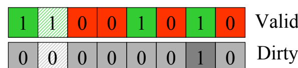
Write-through cache

In **write-through** caches, every **write** causes an *immediate* change both to cache and to main memory.

So the **read** just involves fetching the block.



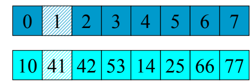


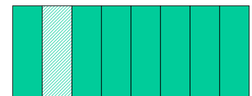


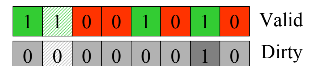
Write-back cache

In **write-back** caches, every **write** causes a change only to cache.

So the **read** involves writing block 31 back to memory if its **dirty** bit is set, then fetching block 41.







Reads easy, writes are not

- Most memory access is read, not write, because read both data and instructions but write only data
- If the data requested is not in cache, call that a *cache miss*
- It's easy to make most reads from cache fast: just pull the data into a register as soon as it is accessed, while checking whether the address matches the tag. If not, that is a cache miss, so load a block from main memory to cache.
- Can't do this with write:
  - must verify the address *before* changing the value of the cache location

Write through vs. write back

- Which is better?
  - Write back gives faster writes, since don't have to wait for main memory
  - Write back is very efficient if want to modify many bytes in a given block
  - But write back can slow down some reads, since a cache miss might cause a write back
  - In multiprocessors, write through might be the only correct solution. Why?

Cache summary

- Cache memory can be organized as direct mapped, set associative, or fully associative
- Can be write-through or write-back
- Extra bits such as **valid** and **dirty** bits help keep track of the status of the cache

How much do memory stalls slow down a machine?

- Suppose that on pipelined MIPS, each instruction takes, on average, 2 clock cycles, not counting cache faults/misses
- Suppose, on average, there are 1.33 memory references per instruction, memory access time is 50 cycles, and the miss rate is 2%
- Then each instruction takes, on average:

$$2 + (0 \times .98) + (1.33 \times .02 \times 50) = 3.33 \text{ clock cycles}$$

Memory stalls (cont.)

- To reduce the impact of cache misses, can reduce any of three parameters:
  - main memory access time (miss penalty)
  - miss rate
  - cache access (hit) time

Reducing cache miss penalty

- 5 strategies:
  - Give priority to *read* misses over *write* misses
  - Don't wait for the whole block
  - Use a nonblocking cache
  - Multi-level cache
  - Victim caches
- First 4 used in most desktop and server machines

Give priority to read misses over write misses

- But need to be careful
- Example:
  - Suppose have a direct mapped cache, with room for 8 blocks of 16 bytes each
  - Then **M[512]** and **M[1024]** both get stored in block 0, so can't be in cache at the same time
- Consider the following instructions:

```
SD   R3, 512(R0)
LD   R1, 1024(R0)
LD   R2, 512(R0)
```


Example (cont.)

- If the cache is write-through, the **SD** will cause memory location 512 to be changed
- The first **LW** will cause block 0 to be replaced, so that the contents **M[512]** are no longer available in cache
  - If the system is write-back, this is when memory location 512 will be changed
- Physically, the contents of block 0 will be put into temporary storage (a *write buffer*) while the new block is loaded, then the write back proceeds
- The second **LW** again replaces block 0, but this time no write-back is necessary
- But get a RAW hazard if don't ensure that the write-through or write-back completes before the second **LW** reads memory

Example (cont.)

- To avoid such RAW hazards:
  - Can force the read miss to always wait until the write buffer is empty
  - Or can force the hardware to check the write buffer before read and only wait if there is a potential hazard

Another write buffer optimization

- Write buffer mechanics, with *merging*
  - An entry may contain multiple words (maybe even a whole cache block)
  - If there's an empty entry, the data and address are written to the buffer, and the CPU is done with the write
  - If buffer contains other modified blocks, check to see if new address matches one already in the buffer – if so, combine the new data with that entry
  - If buffer full and no address match, cache and CPU wait for an empty entry to appear (some entry has been written to main memory)
  - Merging improves memory efficiency, since multi-word writes usually faster than one word at a time

Don't wait for the whole block

- Two ways to do this – suppose need the 10th word in a block:
  - *Early restart*: access the required word as soon as it is fetched, instead of waiting for the whole block
  - *Critical word first*: start the fetch with word 10, and fill in the first few later

Use a nonblocking cache

- With this optimization, the cache doesn't stop for a miss, but continues to process later requests if possible, even though an earlier one is not yet fulfilled
  - Introduces significant complexity into cache architecture – have to allow multiple outstanding cache requests (maybe even multiple misses)

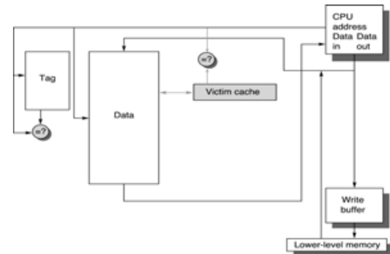
Multi-level cache

- For example, if cache takes 1 clock cycle, and memory takes 50, might be a good idea to add a larger (but necessarily slower) secondary cache in between, perhaps capable of 10 clock cycle access
- Complicates performance analysis (see H&P), but 2nd level cache captures many of 1st level cache misses, lowering effective miss penalty

Victim caches

- To remember a cache block that has recently been replaced (*evicted*)
  - use a small, *fully associative* cache between a cache and where it gets data from
  - check the victim cache on a cache miss, before going to next lower-level memory
    - if found, swap victim block and cache block
  - reduces *conflict* misses, soon to be discussed

Victim caches (cont.) – Fig. 5.13



© 2003 Elsevier Science (USA). All rights reserved.

Computer Systems Architecture CMSC 411 Unit 5 – Memory Hierarchy

Alan Sussman

April 22, 2003

Administrivia

- Quiz today
- HW 5 out, mostly
- Project questions?

Last time

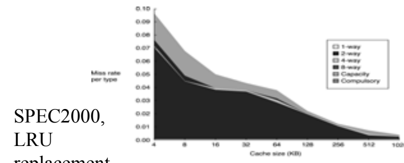
- Managing cache
  - direct mapped, set associative, fully associative
  - write-through vs. write-back
  - valid and dirty bits to keep track of cache state
- Reducing cache miss penalty
  - Give priority to read over write misses, but maintain correctness – write buffer problems
  - Write buffer merging
  - Don't wait for whole block
    - Early restart or critical word first
  - Nonblocking cache
  - Multilevel cache
  - Victim cache

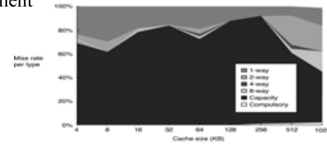
Reducing the miss rate

- Sometimes cache misses are inevitable:
  - The first time a block is used, need to bring it into cache (a *compulsory miss*)
  - If need to use more blocks at once than can fit into cache, some will bounce in and out (*capacity miss*)
  - In direct mapped or set associative caches, there are certain combinations of addresses that cannot be in cache at the same time (*conflict miss*)

Miss rate – Fig. 5.15

SPEC2000, LRU replacement





© 2003 Elsevier Science (USA). All rights reserved.

How to reduce the miss rate?

- Use larger blocks
- Use more associativity, to reduce conflict misses
- Victim cache
- Pseudo-associative caches
- Prefetch (hardware controlled)
- Prefetch (compiler controlled)
- Compiler optimizations

Increasing block size

- Want the block size **large** so don't have to stop so often to load blocks
- Want the block size **small** so that blocks load quickly

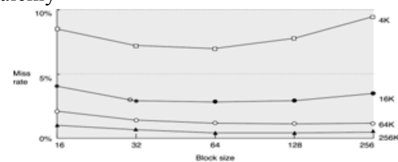


Fig. 5.16 – SPEC92

© 2003 Elsevier Science (USA). All rights reserved.

Increasing block size (cont.)

- So large block size reduces miss rates, but...
- Example:
  - Suppose that loading a block takes 80 cycles (overhead) plus 2 clock cycles for each 16 bytes
  - A block of size 64 bytes can be loaded in 80 + 2*64/16 cycles = 88 cycles (miss penalty)
  - If the miss rate is 7%, then the average memory access time is
    1 + .07 * 88 = 7.16 cycles

Memory Access Times – Fig. 5.18

		Cache size			
Block size	Miss penalty	4K	16K	64K	256K
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

Higher associativity

- A direct-mapped cache of size *N* has about the same miss rate as a 2-way set-associative cache of size *N/2*
  - 2:1 cache rule of thumb (seems to work up to 128KB caches)
- But associative cache is slower than direct-mapped, so the clock may need to run slower
- Example:
  - Suppose that the clock for 2-way memory needs to run at a factor of 1.1 times the clock for 1-way memory
    - the **hit** time increases with higher associativity
  - Then the average memory access time for 2-way is 1.10 + miss rate × 50 (assuming that the miss penalty is 50)

CMSC 411 - Alan Sussman 61

Memory access time – Fig. 5.19

Cache size (KB)	Associativity			
	One-way	Two-way	Four-way	Eight-way
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

CMSC 411 - Alan Sussman 62

Pseudo-associative cache

- Uses the technique of *chaining*, with a series of cache locations to check if the block is not found in the first location
  - e.g., invert most significant bit of index part of address (as if it were a set associative cache)
- The idea:
  - Check the direct mapped address
  - Until the block is found or the chain of addresses ends, check the next alternate address
  - If the block has not been found, bring it in from memory
- Three different delays generated, depending on which step succeeds

CMSC 411 - Alan Sussman 63

Computer Systems Architecture CMSC 411 Unit 5 – Memory Hierarchy

Alan Sussman

April 24, 2003

Administrivia

- Quiz questions?
- HW 5
- Project questions?

CMSC 411 - Alan Sussman 65

Hardware prefetch

- Idea: If read page *k* of a book, the next page read is most likely page *k+1*
- So, when a block is read from memory, read the next block too
  - maybe into a separate buffer that is accessed on a cache miss before going to memory
- Advantage:
  - if use blocks sequentially, will need to fetch only half as often from memory
- Disadvantages:
  - more information to move
  - may fill the cache with useless blocks
  - may compete with demand misses for memory bandwidth

CMSC 411 - Alan Sussman 66

Compiler-controlled prefetch

- Idea: The compiler has a better idea than the hardware does of when blocks are being use sequentially
- Want the prefetch to be nonblocking:
  - don't slow the pipeline waiting for it
- Usually want the prefetch to fail quietly:
  - if ask for an illegal block (one that generates a page fault or protection exception), don't generate an exception; just continue as if the fetch wasn't requested
  - called a *non-binding* cache prefetch

Compiler optimizations to reduce cache miss rate

Four compiler techniques

- 4 techniques to improve cache locality:
  - merging arrays
  - loop interchange
  - loop fusion
  - blocking

Technique 1: merging arrays

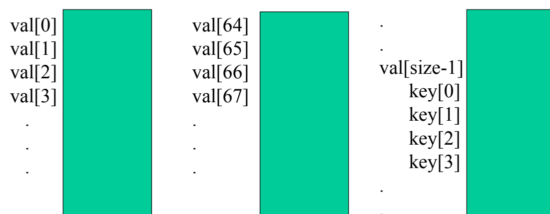
- Suppose have two arrays:

```
int val[size];
int key[size];
```

- and that usually use both of them together

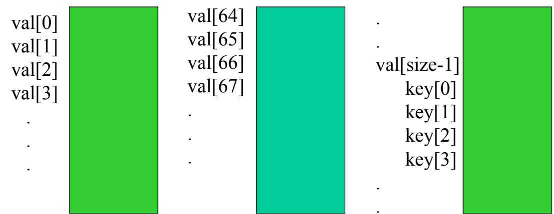
Merging arrays (cont.)

This is how they would be stored if cache blocksize is 64 words:



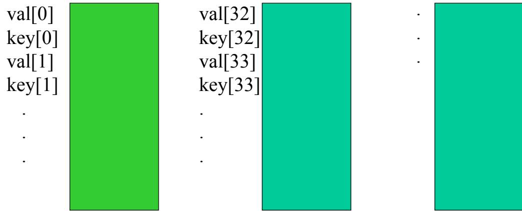
Merging arrays (cont.)

Means that at least 2 blocks must be in cache to begin using the arrays.



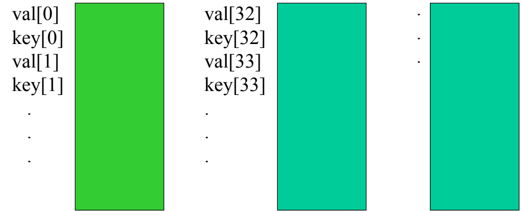
Merging arrays (cont.)

More efficient, especially if more than two arrays are coupled this way, to store them together.



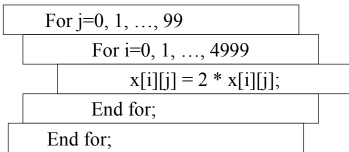
Merging arrays (cont.)

Can do this by making the two arrays part of a structure.



Technique 2: interchanging loops

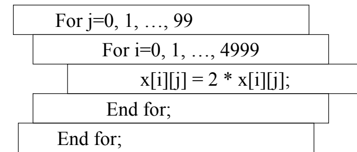
Example:



Interchanging loops (cont.)

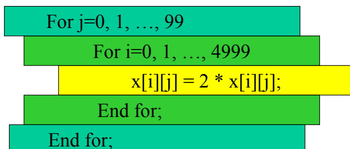
Notice that accesses are by columns, so the elements are spaced 100 words apart.

Blocks are bouncing in and out of cache.



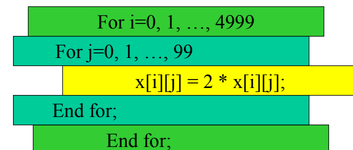
Interchanging loops (cont.)

First color the loops:



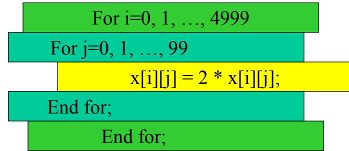
Interchanging loops (cont.)

Notice that the program has the same effect if the two loops are interchanged:



Interchanging loops (cont.)

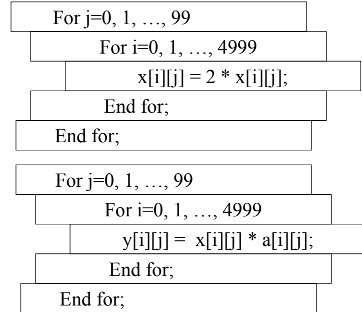
But with this ordering, use every element in a cache block before needing another block!



CMSC 411 - Alan Sussman 79

Technique 3: loop fusion

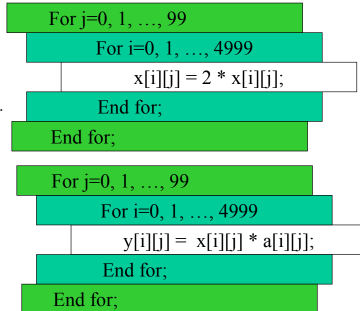
Example:



CMSC 411 - Alan Sussman 80

Loop fusion (cont.)

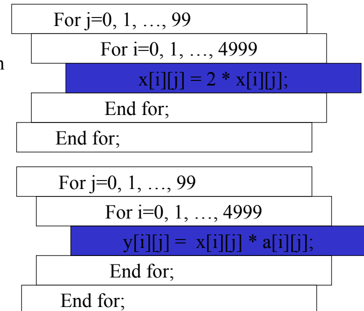
Note that the loop control is the same for both sets of loops.



CMSC 411 - Alan Sussman 81

Loop fusion (cont.)

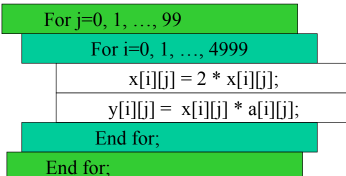
And note that the array x is used in each, so probably needs to be loaded into cache twice, which wastes cycles.



CMSC 411 - Alan Sussman 82

Loop fusion (cont.)

So combine,or **fuse**, the loops to improve efficiency.



CMSC 411 - Alan Sussman 83

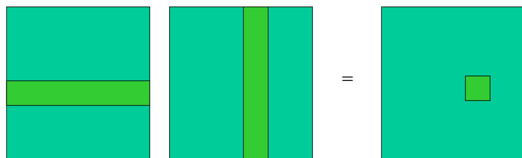
Technique 4: blocking access to arrays

- *Blocking* does not mean *preventing*. Means gathering the accesses into blocks that optimize cache use (access order, block size, etc.)

CMSC 411 - Alan Sussman 84

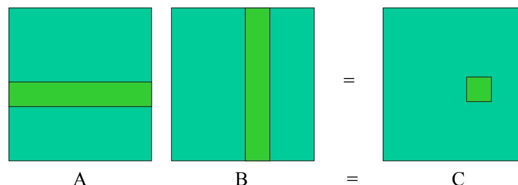
Blocking access to arrays (cont.)

Example: Matrix-matrix multiplication



Blocking access to arrays (cont.)

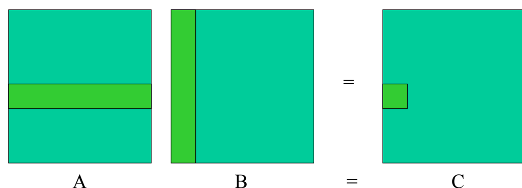
Trouble: Easy to get rows of A;
not so efficient to get columns of B.



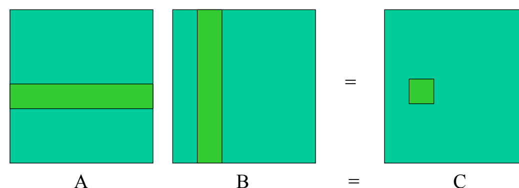
Blocking access to arrays (cont.)

And if cycle through rows of A, end up loading all of B *m* times, where *m* is the number of rows of A.

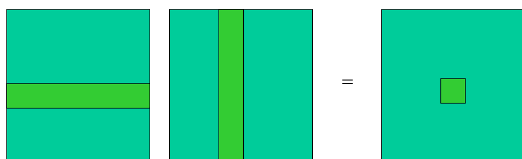
Computing the elements in the columns of C:



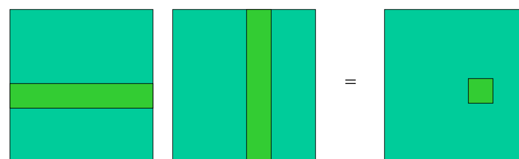
Blocking access to arrays (cont.)



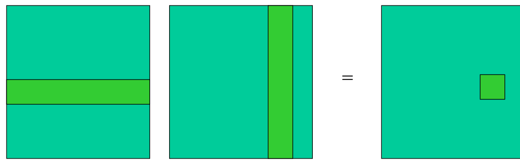
Blocking access to arrays (cont.)



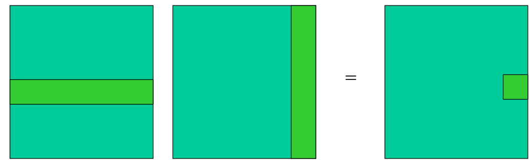
Blocking access to arrays (cont.)



Blocking access to arrays (cont.)

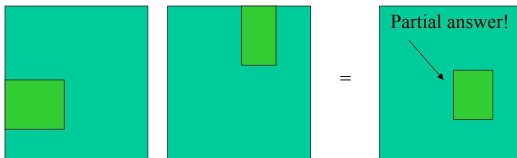


Blocking access to arrays (cont.)



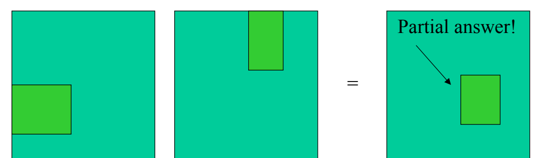
Blocking access to arrays (cont.)

Instead, order the computation using rectangular blocks of A and B.



Blocking access to arrays (cont.)

If the block of A has *k* rows, then only need to load B *m/k* times.



Blocking access to arrays (cont.)

Improves *temporal locality*

```
/* Before */
for (i=0; i<N; i++)
   for (j=0; j<N; j++) {
      r=0;
      for (k=0; k<N; k++)
         r=r+y[i][k]*z[k][j];
      x[i][j]=r;
   }
```

```
/* After */
for (jj=0; jj<N; jj=jj+B)
for (kk=0; kk<N; kk=kk+B)
for (i=0; i<N; i++)
   for (j=jj; j<min(jj+B,N); j++) {
      r=0;
      for (k=kk; k<min(kk+B,N); k++)
         r= r+y[i][k]*z[k][j];
      x[i][j]=x[i][j]+r;
   }
```


Reducing the time for cache hits

- K.I.S.S.
- Use *virtual addresses* rather than *physical addresses* in the cache.
- Pipeline cache accesses
- Trace caches

K.I.S.S.

- Cache should be small enough to fit on the processor chip
- Direct mapped is faster than associative, especially on *read*
  - overlap tag check with transmitting data
- For current processors, small L1 caches to keep fast clock cycle time, hide L1 misses with dynamic scheduling, and use L2 caches to avoid main memory accesses

Simulated cache access times

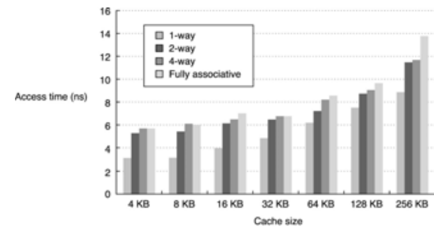


Fig. 5.24 – 0.8-micron feature size, 1 R/W port, 32 address & 64 data bits, 32-byte blocks

© 2003 Elsevier Science (USA). All rights reserved.

Use virtual addresses

- Each user has his/her own *address space*, and no addresses outside that space can be accessed
- To keep address length small, each user addresses by offsets relative to some physical address in memory (pages)
- For example:

Physical address	Virtual address
5400	00
5412	12
5500	100

Virtual addresses (cont.)

- Since instructions use virtual addresses, use them for index and tag in cache, to save the time of translating to physical address space (the subject of the next part of this unit)
- Note that it is important to flush the cache and set all blocks invalid when switch to a new user in the OS (a *context switch*), since the same virtual address then may refer to a different physical address
  - or use the process/user ID as part of the tag in cache
- Aliases are another problem
  - when two different virtual addresses map to the same physical address – can get 2 copies in cache
    - what happens when one copy is modified?

Pipelined cache access

- Latency to first level cache is more than one cycle
  - we've already seen this in Units 3 & 4
- Benefit is fast cycle time
- Penalty is slower hits
  - also more clock cycles between a load and the use of the data (maybe more pipeline stalls)

Trace cache

- Find a dynamic sequence of instructions to load into a cache block, including *taken* branches
  - instead of statically, from how the instructions are laid out in memory
  - branch prediction needed for loading cache
- One penalty is complicated address mapping, since addresses not always aligned to cache block size
  - can also end up storing same instructions multiple times
- Benefit is only caching instructions that will actually be used (if branch prediction is right), not all instructions that happen to be in the same cache block

Computer Systems Architecture CMSC 411 Unit 5 – Memory Hierarchy

Alan Sussman

April 29, 2003

Administrivia

- Quiz 3
  - Average: 48 Median: 49 25%: 34 75%: 59
  - questions?
- Quiz 4 moved to May 8
- HW 5 posted – due May 6
- Project questions?

Last time

- Reducing cache misses
  - hardware prefetch – for sequential access
  - software prefetch – by compiler
    - want it to be non-blocking and fail quietly
  - compiler techniques – merging arrays, loop interchange, loop fusion, blocking array accesses
- Reducing cache hit times
  - small, fast caches backed by slower, bigger ones
  - use direct mapping instead of associative
  - use virtual addresses – to avoid address translation
  - pipeline accesses – faster cycle time, slower hits
  - trace cache

Main Memory

Main memory management

- Questions:
  - How big should main memory be?
  - How to handle reads and writes?
  - How to find something in main memory?
  - How to decide what to put in main memory?
  - If main memory is full, how to decide what to replace?

The scale of things

- Typically (as of 2000):
  - *Registers*: < 1 KB, access time .25 - .5 ns
  - *Cache*: < 8 MB, access time .5 - 25 ns
  - *Main Memory*: < 4 GB, access time 150 - 250 ns
  - *Disk Storage*: > 30 GB, access time 5,000,000 ns (5ms)
- Memory Technology: CMOS (Complementary Metal Oxide Semiconductor)
  - uses a combination of n- and p-doped semiconductor material to achieve low power dissipation.

Memory hardware

- *DRAM*: dynamic random access memory, typically used for main memory
  - one transistor per data bit
  - each bit must be refreshed periodically (e.g., every 8 milliseconds), so maybe 5% of time is spent in refresh
  - access time < cycle time
  - address sent in two halves so that fewer pins are needed on chip (row and column access)

Memory hardware (cont.)

- *SRAM*: static random access, typically used for cache memory
  - 4-6 transistors per data bit
  - no need for refresh
  - access time = cycle time
  - address sent all at once, for speed

Bottleneck

- Main memory access will slow down the CPU unless the hardware designer is careful
- Some techniques can improve *memory bandwidth*, the amount of data that can be delivered from memory in a given amount of time:
  - *wider* main memory
  - interleaved memory
  - independent memory banks
  - avoiding memory bank conflicts

Wider main memory

- Cache miss: If a cache block contains k words, then each cache miss involves these steps repeated k times:
  - Send the address to main memory
  - Access the word (i.e., locate it)
  - Send the word to cache, with the bits transmitted in parallel
- Idea behind wider memory: the user thinks about 32 bit words, but physical memory can have longer words
- Then the operations above are done only k/n times, where n is the number of 32 bit words in a physical word

Wider main memory (cont.)

- Extra costs:
  - a wider *memory bus*: hardware to deliver 32n bits in parallel, instead of 32 bits
  - a multiplexor to choose the correct 32 bits to transmit from the cache to the CPU

Interleaved memory

- Partition memory into banks, with each bank able to access a word and send it to cache in parallel
- Organize address space so that adjacent words live in different banks - called *interleaving*
- For example, 4 banks might have words with the following octal addresses:

Bank 0	Bank 1	Bank 2	Bank3
00	01	02	03
04	05	06	07
10	11	12	13
…	…	…	…

Interleaved memory (cont.)

- Note how nice interleaving is for write-through
- Also helps speed read and write-back
- *Note*: Interleaved memory acts like wide memory, except that words are transmitted through the bus sequentially, not in parallel

CMSC 411 - Alan Sussman 115

Independent memory banks

- Each *bank* of memory has its own address lines and (usually) a bus
- Can have several independent banks: perhaps
  - one for instructions
  - one for data
- Banks can operate independently without slowing others

CMSC 411 - Alan Sussman 116

Avoid memory bank conflicts

- By having a prime number of memory banks
- Since arrays frequently have even dimension sizes - and often dimension sizes that are a power of 2 - strides that match the number of banks (or a multiple) give very slow access

CMSC 411 - Alan Sussman 117

Example

```
int x[256][512];

for (j=0; j<512; j=j+1)
for (i=0; i<256; i=i+1)
   x[i][j] = 2 * x[i][j];
```

- First access the first column of **x**:
  - x[0][0], x[1][0], x[2][0], ... x[255][0,
- with addresses
  - K, K+512*4, K+512*8, ... K+512*something
- With 4 memory banks, all of the elements live in the same memory bank, so the CPU will stall in the worst possible way

CMSC 411 - Alan Sussman 118

Number theory to the rescue!

- Subtitle: One reason why computer scientists need math
- *Fact 1*: It is easy to compute **mod**, if the base is a prime number that is one less than a power of 2
- *Fact 2*: The Chinese remainder theorem says that it is safe to do a rather bizarre, but convenient, mapping of words to memory banks
- Idea: Suppose have 3 memory banks with 8 words each
  - Map word *k* to memory bank *k mod 3*, word *k mod 8*
  - For example, word 17 (decimal) goes to bank 2, address 1
  - Word 9 goes to bank 0, word 1

CMSC 411 - Alan Sussman 119

Number theory (cont.)

- That mapping gives the following arrangement of words:

Address within bank	Memory bank					
	Sequentially interleaved			Modulo interleaved		
	0	1	2	0	1	2
0	0	1	2	0	16	8
1	3	4	5	9	1	17
2	6	7	8	18	10	2
3	9	10	11	3	19	11
4	12	13	14	12	4	20
5	15	16	17	21	13	5
6	18	19	20	6	22	14
7	21	22	23	15	7	23

CMSC 411 - Alan Sussman 120

How much good do these techniques do?

- Example: Assume a cache block of 4 words, and
  - 4 cycles to send address to main memory
  - 24 cycles to access a word, once the address arrives
  - 4 cycles to send a word back to cache
- *Basic miss penalty*: 4*32 = 128 cycles, since each of 4 words has the full 32 cycle penalty
- *Memory with a 2-word width*: 2*32 = 64 cycle miss penalty
- *Simple interleaved memory*: address can be sent to each bank simultaneously, so miss penalty is 4 + 24 + 4*4 (for sending words) = 44 cycles
- *Independent memory banks*: 32 cycle miss penalty, as long as the words are in different banks, since each has its own address lines and bus

A confession: We've been lying

- The lie: User's don't really use physical addresses in their programs
- Instead, they use *virtual addresses*, where *virtual* means just what it does in virtual reality!
- This idea is over 40 years old, invented by the designers of the Atlas computer (Section 5.18)
  - Cache memory is just a little newer, discussed in print in 1965
- So when the user addresses word 450, the computer provides the illusion that the data is actually in this address, when really the data is somewhere else
- This *address translation* or *memory mapping* is invisible to the user

Why virtual addressing

- Computers are designed so that multiple users can be active at the same time
- At the time a program is compiled, the compiler has to assign addresses to each data item. But how can it know what memory addresses are being used by other users?
- Instead, the compiler assigns virtual addresses, and expects the loader to provide the means to map these into physical addresses

In the olden days …

- The loader would locate an unused set of main memory addresses and load the program and data there
- There would be a special register called the *relocation register*, and all addresses that the program used would be interpreted as addresses relative to the base address in that register
- So if the program jumped to location 54, the jump would really be to 54 + contents of relocation register. A similar thing, perhaps with a second register, would happen for data references

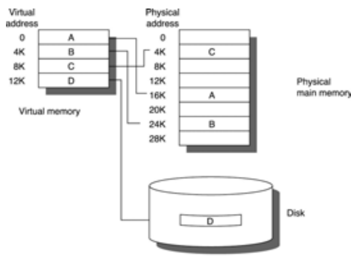
In the less-olden days ...

- It became difficult to find a contiguous segment of memory big enough to hold program and data, so the program was divided into *pages*, with each page stored contiguously, but different pages in any available spot, either in main memory or on *disk*
- This is the *virtual addressing* scheme
  - to the user, memory looks like a contiguous segment, but actually, data is scattered in main memory and perhaps on disk

But we know all about this!

- Already know that a program and data can be scattered between cache memory and main memory
- Now add the reality that its location in main memory is also determined in a scattered way, and some pages may also be located on disk
- So each page has its own relocation value

Virtual Memory – Fig. 5.31



© 2003 Elsevier Science (USA). All rights reserved.

Parameters – Fig. 5.32

Parameter	First-level cache	Virtual memory
Block (page) size	16-128 bytes	4096-65,536 bytes
Hit time	1-3 clock cycles	50-150 cc
Miss penalty	8-150 cc	10^6-10^7 cc
(access time)	(6-130 cc)	(.8-8*10^6 cc)
(transfer time)	(2-20 cc)	(.2-2*10^6 cc)
Miss rate	0.1-10%	0.00001-0.001%
Address mapping	25-45 bit physical address to 14-20 bit cache address	32-64 bit virtual address to 25-45 bit physical address

Cache vs. virtual memory

Cache	Virtual memory
Cache miss handled by hardware	Page faults handled by operating system
Cache size fixed for a particular machine	Virtual memory size fixed for a particular program
Fundamental unit is a block	Fundamental unit is a fixed-length page or a variable-length segment
cache fault	page fault

Paging vs. segmentation – Fig. 5.34

	Page	Segment
Words per address	One	Two (segment/offset)
Programmer visible?	Invisible to app programmer	May be visible to app programmer
Replacing a block	Trivial (all blocks same size)	Hard (must find contiguous, variable-sized chunk)
Memory use inefficiency	Internal fragmentation (within page)	External fragmentation (in unused memory)
Efficient disk traffic	Yes (can adjust page size)	Not always (small segment problem)

Managing main memory

- Fully-associative mapping, because page faults are *really, really* expensive
- Page is located using a *page table*, one entry per page in the virtual address space
  - Size is sometimes reduced by hashing, to make one entry per physical page in main memory – an *inverted* page table
- Since locality says that a page will be used multiple times, address translation usually tests the address of the last-referenced page before looking in other places
- So address translation information is held in the *translation look-aside buffer* (TLB)

Computer Systems Architecture CMSC 411 Unit 5 – Memory Hierarchy

Alan Sussman

May 1, 2003

Administrivia

- Quiz 4 next Thursday, on Memory Hierarchy
- HW 5 due Tuesday
  - note a small change to solutions to HW 4B, question 4.10a
- Read Chapter 7 in H&P – Storage Systems
  - except Section 7.8
- Project due next Friday
  - for Part II, unroll only the innermost loop
  - questions?

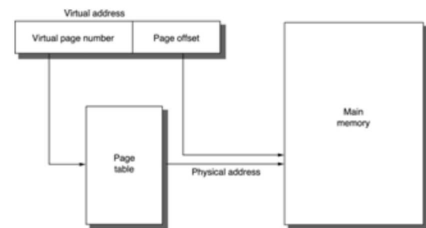
Last time

- Improving main memory bandwidth
  - wider main memory – *k* words at at time
  - interleaved memory – separate memory banks
  - independent memory banks – each with address lines and a bus to the processor
  - avoiding memory bank conflicts – for common access patterns (e.g., sequential, strided)
    - example was of modulo interleaved pattern – map word *k* to memory bank *k* ***mod*** *#banks*, word *k* ***mod*** *#words per bank*

Last time

- Virtual memory
  - transparent address translation from *virtual* address in program to *physical* address in memory
  - each *page* has a relocation value stored in a page table
  - fully associative mapping of pages to *page frames*
  - page faults handled by OS – compare to cache faults
  - TLB caches address translation information

Page tables



© 2003 Elsevier Science (USA). All rights reserved.

Managing main memory (cont.)

- Most machines replace the LRU page
  - moving pages between memory and disk is so slow that it's worth doing something close to real LRU
- Disks are *so slow* that machines use write-back, not write-through, and keep a *dirty bit* for each page

Choosing page size

- A large page size
  - keeps page table small.
  - reduces cache miss times, if accesses have locality
  - reduces start-up overhead in moving data from disk to memory
  - means fewer TLB misses
- but also
  - wastes memory (internal fragmentation)
  - increases the time to start up a program

Memory protection

- Each program “lives” in its own virtual space, called its *process*
- When the CPU is working on one process, others may be partially completed or waiting for attention
- The CPU is *time shared* among the processes, working on each in turn
- And main memory is also shared among processes

Protection modes

- User processes need to be protected from each other
- Two registers, *base* and *bound* test whether this virtual address belongs to this process
- If not, a *memory protection violation* exception is raised
- Users cannot change the base and bound registers

Who can change them?

- The operating system needs access to the base and bound registers
- So a process that is labeled *kernel* (also called *supervisor* or *executive*) can access any memory location and change the registers
- Kernel processes are accessed through *system calls*, and a return to user mode is like a subroutine return, restoring the state of the user process

The bookkeeping

- Each process has
  - base and bound values, denoting its virtual address space
  - page tables, giving mapping to accessible pages in memory
  - mode bit (user or kernel)
- A user process cannot modify any of this information

Examples

- The book uses examples to illustrate these concepts:
  - Alpha and Intel Pentium
- Read them if you are curious
  - and to see how ugly Pentium segments are

Fallacies & Pitfalls

- *Biggest pitfall in computer design*: If the addresses are too short, the length of programs and the amount of data they can use will be too limited
- *Other pitfalls/fallacies*:
  - using benchmarks to predict memory performance
  - emphasizing DRAM bandwidth vs. latency
  - reducing memory efficiency by a poorly designed operating system
  - making the write buffer too small