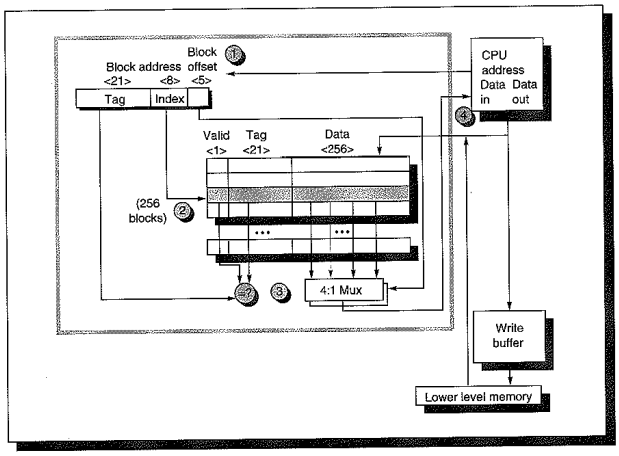
Memory Hierarchy 2
(Cache Optimizations)

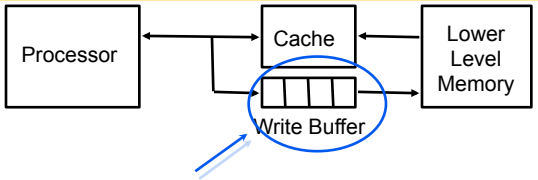
So far….

- Fully associative cache
  - Memory block can be stored in any cache block
- Write-through cache
  - Write (store) changes both cache and main memory right away
  - Reads only require getting block on cache miss
- Write-back cache
  - Write changes only cache
  - Read causes write of dirty block to memory on a replace
- Reads easy to make fast, writes harder
  - Read data from cache in parallel with checking address against tag of cache block
  - Write must verify address against tag before update

Example: Alpha 21064



Write buffers for write-through caches



Holds data awaiting write-through to lower level memory

Q. Why a write buffer ?

A. So CPU doesn't stall

Q. Why a buffer, why not just one register ?

A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer?

A. Yes! Drain buffer before next read, or send read 1st after check write buffers.

How much do stalls slow 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:

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

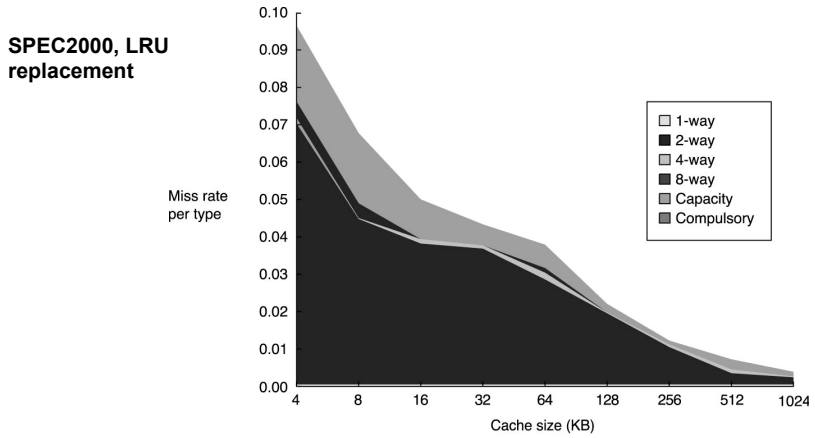
Memory stalls (cont.)

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

Cache miss terminology

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

Miss rate



SPEC2000 cache miss rate vs cache size and associativity (LRU)

5 Basic cache optimizations

- Reducing Miss Rate
  1. Larger Block size (compulsory misses)
  2. Larger Cache size (capacity misses)
  3. Higher Associativity (conflict misses)
- Reducing Miss Penalty
  4. Multilevel Caches
- Reducing hit time
  5. Giving Reads Priority over Writes
     - » E.g., Read completes before earlier writes in write buffer

More terminology

- 'write-allocate'
  - Ensure block in cache before performing a write operation
- 'write-no-allocate'
  - Don't allocate block in cache if not already there

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 (meaning 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 whole block on cache miss

- 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
    - » Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called wrapped fetch and requested word first

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)
  - but this is what's done in modern processors

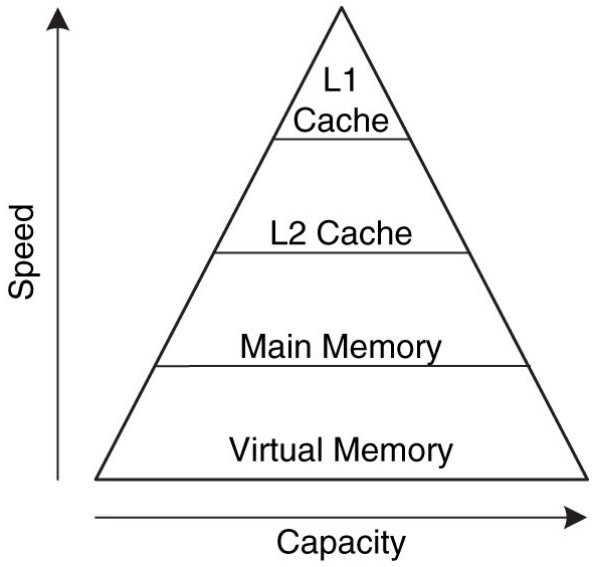
So far (cont.)

- Reducing memory stalls
  - Reduce miss penalty, miss rate, cache hit time
- Reducing miss penalty
  - Give priority to read over write misses
  - Don't wait for the whole block
  - Use a non-blocking cache

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
  - and 3rd level cache has same benefits for 2nd level cache
- Most modern machines have separate 1st level instruction and data caches, shared 2nd level cache
  - and off processor chip shared 3rd level cache

Multi-level cache (cont.)



Example: Apple iMac G5 (2004)

Managed by compiler

Managed by hardware

Managed by OS, hardware, application

	Reg	L1 Inst	L1 Data	L2	DRAM	Disk
Size	1K	64K	32K	512K	256M	80G
Latency Cycles, Time	1, 0.6 ns	3, 1.9 ns	3, 1.9 ns	11, 6.9 ns	88, 55 ns	10^7, 12 ms

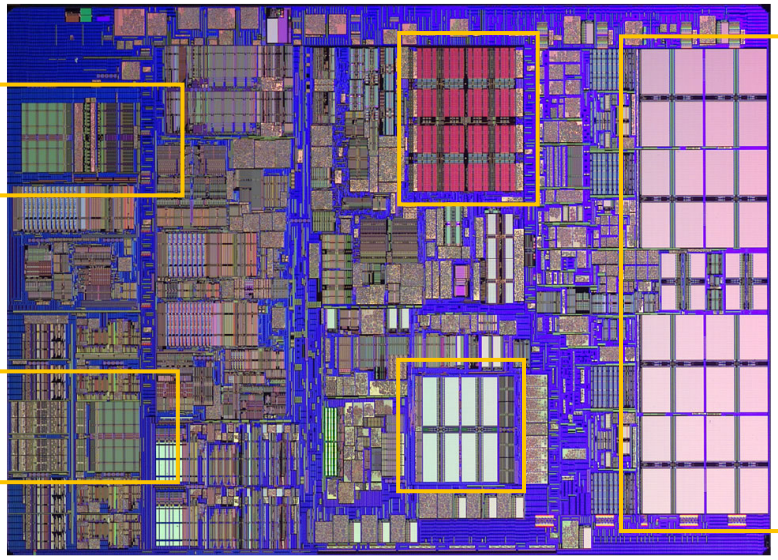
iMac G5
1.6 GHz

Goal: Illusion of large, fast, cheap memory

Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access

iMac's PowerPC 970: All caches on-chip

L1 (64K Instruction)



Registers

512K L2

(1K)

L1 (32K Data)

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

Victim caches (cont.)

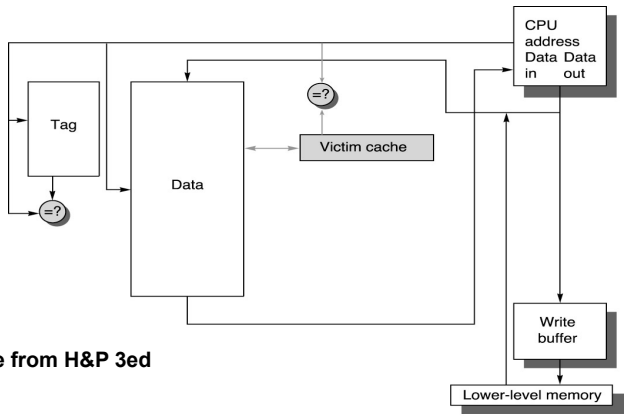


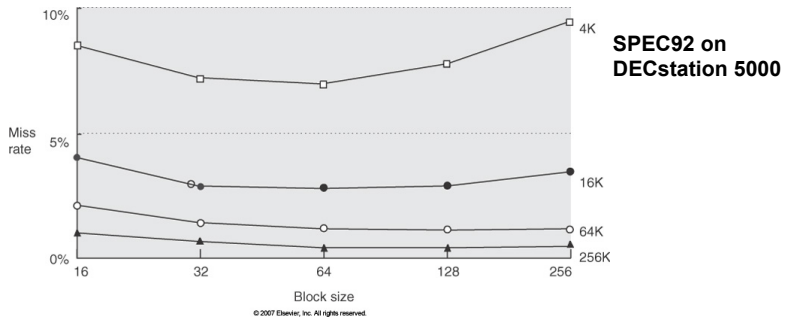
Figure from H&P 3ed

How to reduce the miss rate?

- Use larger blocks
- Use more associativity, to reduce conflict misses
- Victim cache
- Pseudo-associative caches (won't talk about this)
- 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



SPEC92 on DECstation 5000

Increasing block size (cont.)

- So large block size may reduce 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 vs. block size

SPEC92 on DECstation 5000

		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 cache needs to run at a factor of 1.1 times the clock for 1-way cache
    - » 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)

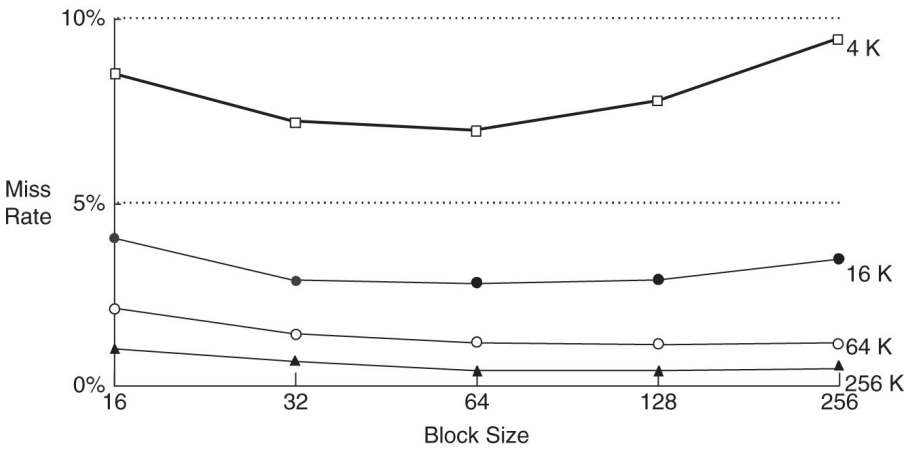
Memory access time

Fig. C.13 - SPEC92 on DECstation 5000

	Associativity			
Cache size (KB)	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**

- If cache big enough, slowdown in access time hurts mem performance

Miss rate vs. cache size & associativity



SPEC2000 benchmark

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

How to reduce the cache miss rate?

- Use larger blocks
- Use more associativity, to reduce conflict misses
- Victim cache
- Pseudo-associative caches (won't talk about this)
- Prefetch (hardware controlled)
- Prefetch (compiler controlled)
- Compiler optimizations

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 access blocks sequentially, will need to fetch only half as often from memory
- Disadvantages:
  - More data to move
  - May fill the cache with useless blocks
  - May compete with demand misses for memory bandwidth

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

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, relatively small L1 caches to keep fast clock cycle time, hide L1 misses with dynamic scheduling, and use L2 and L3 caches to avoid main memory accesses

Use virtual addresses

- Each process has its 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 a later 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 Unit 3
- 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

Compiler optimizations to reduce cache miss rate

Four compiler techniques

- 4 techniques to improve cache locality:
  - Merging arrays
  - Loop interchange
  - Loop fusion
  - Loop blocking / tiling

Technique 1: merging arrays

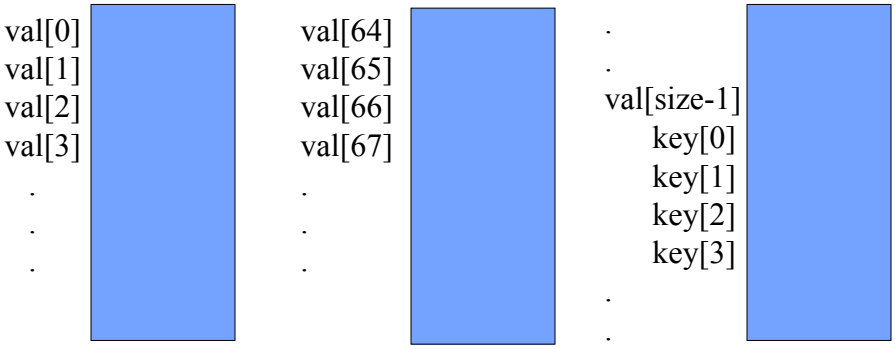
- Suppose have two arrays:

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

and usually use both of them together

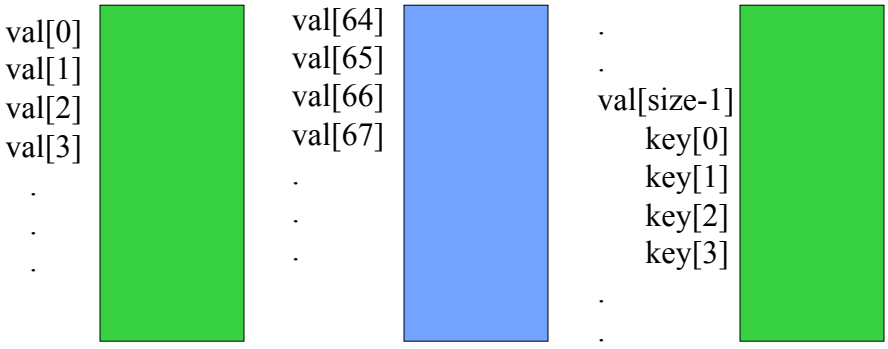
Merging arrays (cont.)

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



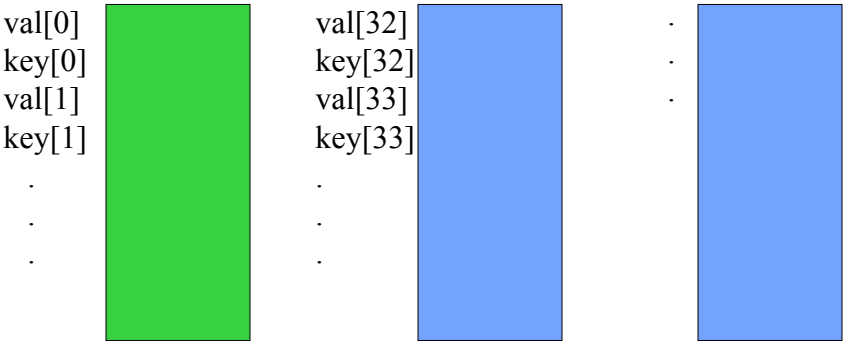
Merging arrays (cont.)

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.



Technique 2: Loop interchange

Example:

C uses row-major storage, so x[i][j] is adjacent to x[i][j+1] in memory, while x[i][j] and x[i+1][j] are one row apart.

```
int x[1000][1000];
For j=0, 1, …, 999
    For i=0, 1, …, 999
        x[i][j] = 2 * x[i][j];
    End for;
End for;
```

Loop interchange (cont.)

Notice that accesses are by columns, so the elements are spaced 1000 words apart.

Blocks are bouncing in and out of cache.

```
For j=0, 1, …, 999
    For i=0, 1, …, 999
        x[i][j] = 2 * x[i][j];
    End for;
End for;
```


Loop interchange (cont.)

First color the loops:

```
For j=0, 1, …, 999
    For i=0, 1, …, 999
        x[i][j] = 2 * x[i][j];
    End for;
End for;
```

Loop interchange (cont.)

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

```
For i=0, 1, …, 999
    For j=0, 1, …, 999
        x[i][j] = 2 * x[i][j];
    End for;
End for;
```

No data dependences across loop iterations

Loop interchange (cont.)

With new ordering, can exploit spatial locality to use every element in a cache block before needing another block!

```
For i=0, 1, …, 999
    For j=0, 1, …, 999
        x[i][j] = 2 * x[i][j];
    End for;
End for;
```

Technique 3: loop fusion

Example:

```
For j=0, 1, …, 999
    For i=0, 1, …, 999
        x[i][j] = 2 * x[i][j];
    End for;
End for;

For j=0, 1, …, 999
    For i=0, 1, …, 999
        y[i][j] =  x[i][j] * a[i][j];
    End for;
End for;
```


Loop fusion (cont.)

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

```
For j=0, 1, …, 999
    For i=0, 1, …, 999
        x[i][j] = 2 * x[i][j];
    End for;
End for;

For j=0, 1, …, 999
    For i=0, 1, …, 999
        y[i][j] =  x[i][j] * a[i][j];
    End for;
End for;
```

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.

```
For j=0, 1, …, 999
    For i=0, 1, …, 999
        x[i][j] = 2 * x[i][j];
    End for;
End for;

For j=0, 1, …, 999
    For i=0, 1, …, 999
        y[i][j] = x[i][j] * a[i][j];
    End for;
End for;
```


Loop fusion (cont.)

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

```
For j=0, 1, …, 999
    For i=0, 1, …, 999
        x[i][j] = 2 * x[i][j];
        y[i][j] = x[i][j] * a[i][j];
    End for;
End for;
```

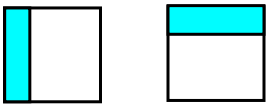
Need to avoid introducing new capacity misses with fused loop.

Technique 4: loop blocking / tiling

Example:

```
For j=0, 1, …, 999
    For i=0, 1, …, 999
        x[i][j] = y[j][i];
    End for;
End for;
```

Notice spatial reuse for both i & j loops (for x & y)



Loop interchange would exploit spatial reuse for either x or y, depending on which loop is outermost.

Loop blocking / tiling (cont.)

```
For jj=0, 50, 100, …, 950
    For ii=0, 50, 100, …, 950
        For j=jj, …, jj+49
            For i=ii, …, ii+49
                x[i][j] = y[j][i];
            End for;
        End for;
    End for;
End for;
```

50x50 blocks

Blocking / tiling breaks loops into strips, then interchanges to form blocks

Block sizes are selected so that they are small enough to exploit locality carried on both loops.

Multi-level inclusion…

- If all data in level *n* is also in level *n+1*
  - Each bigger part of the memory hierarchy contains all data (addresses) in smaller parts
  - Not always the same data because of delayed writeback
- Why useful?
  - I/O…
- May be problematic
  - If block sizes differ between levels