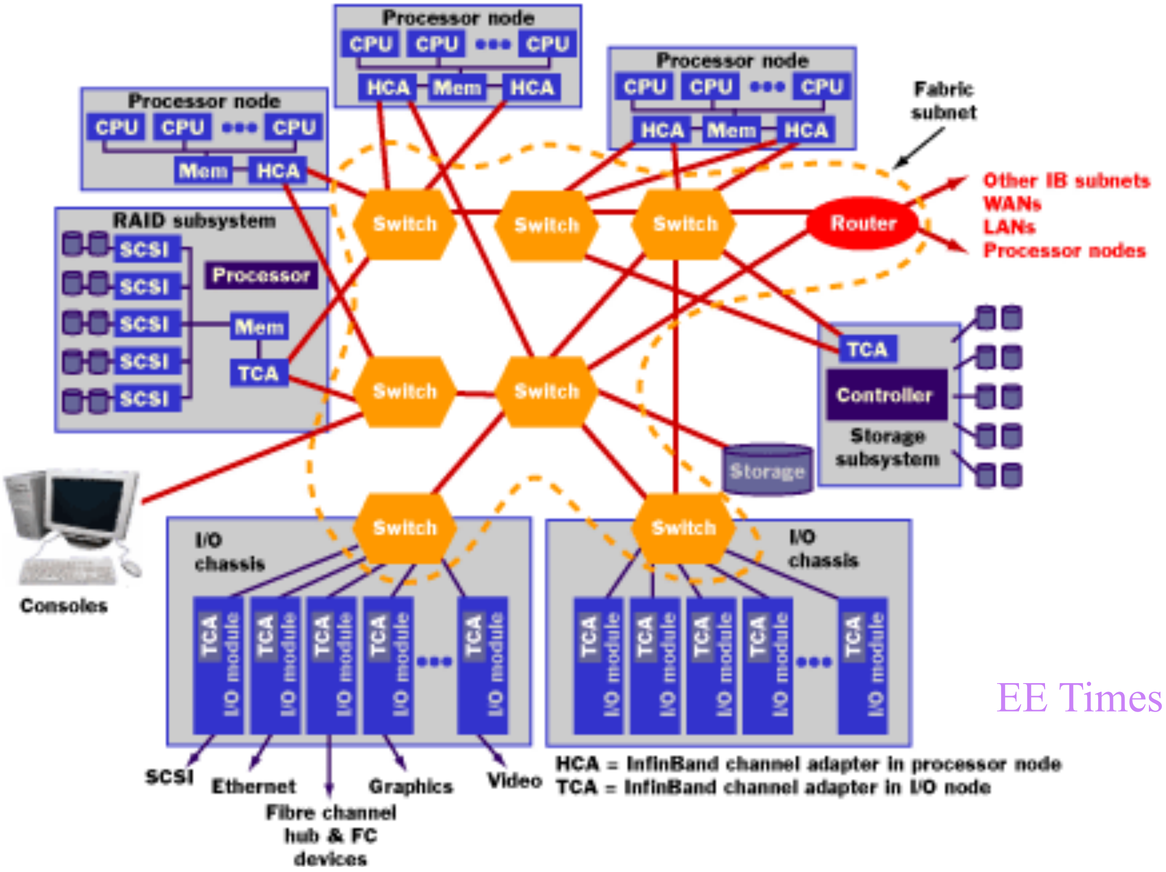
CSE 160
Lecture 5

The Memory Hierarchy
False Sharing
Cache Coherence and Consistency

Scott B. Baden

Using Bang – coming down the home stretch

- Do **not** use Bang's *front end* for running mergeSort
- Use batch, or interactive nodes, via *qlogin*
- Use the front end for editing & compiling only

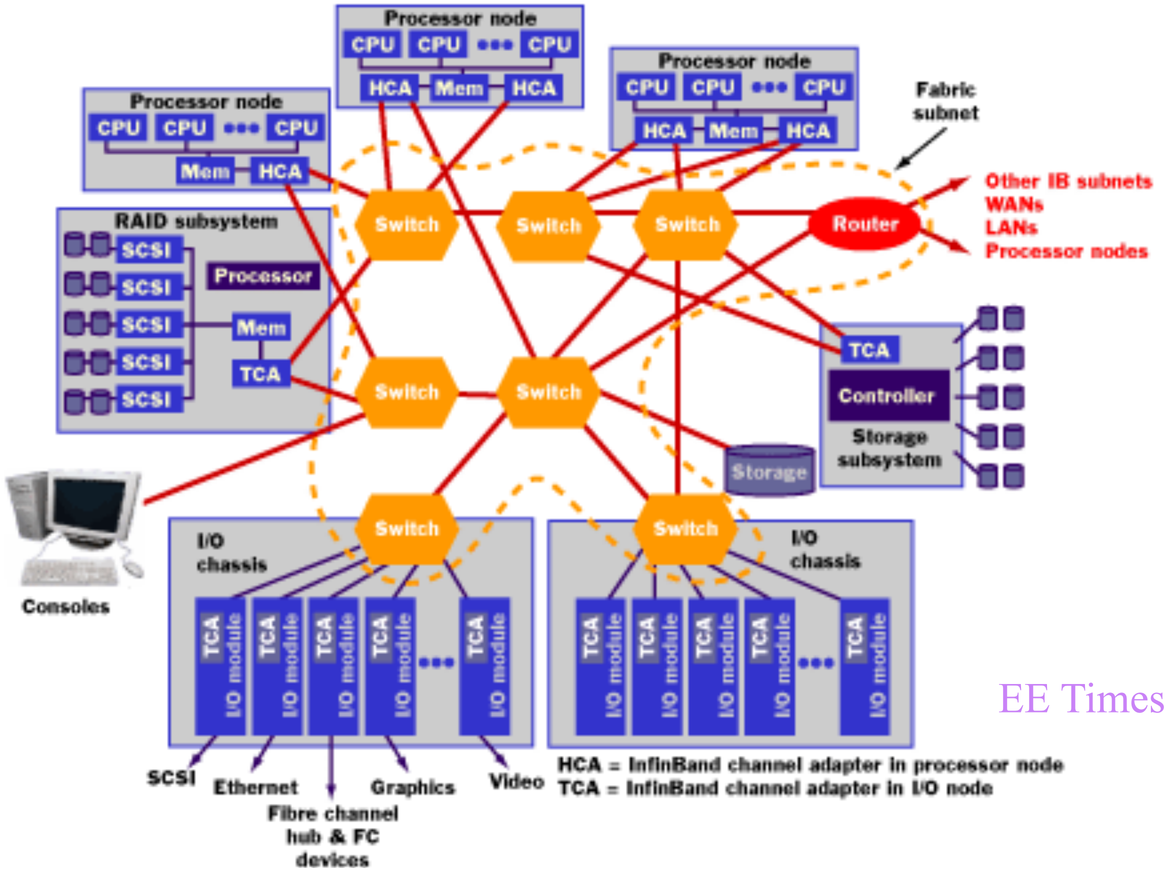


EE Times

10% penalty for using the login nodes improperly, doubles with each incident!

Announcements

- SDSC Tour on Friday 11/1



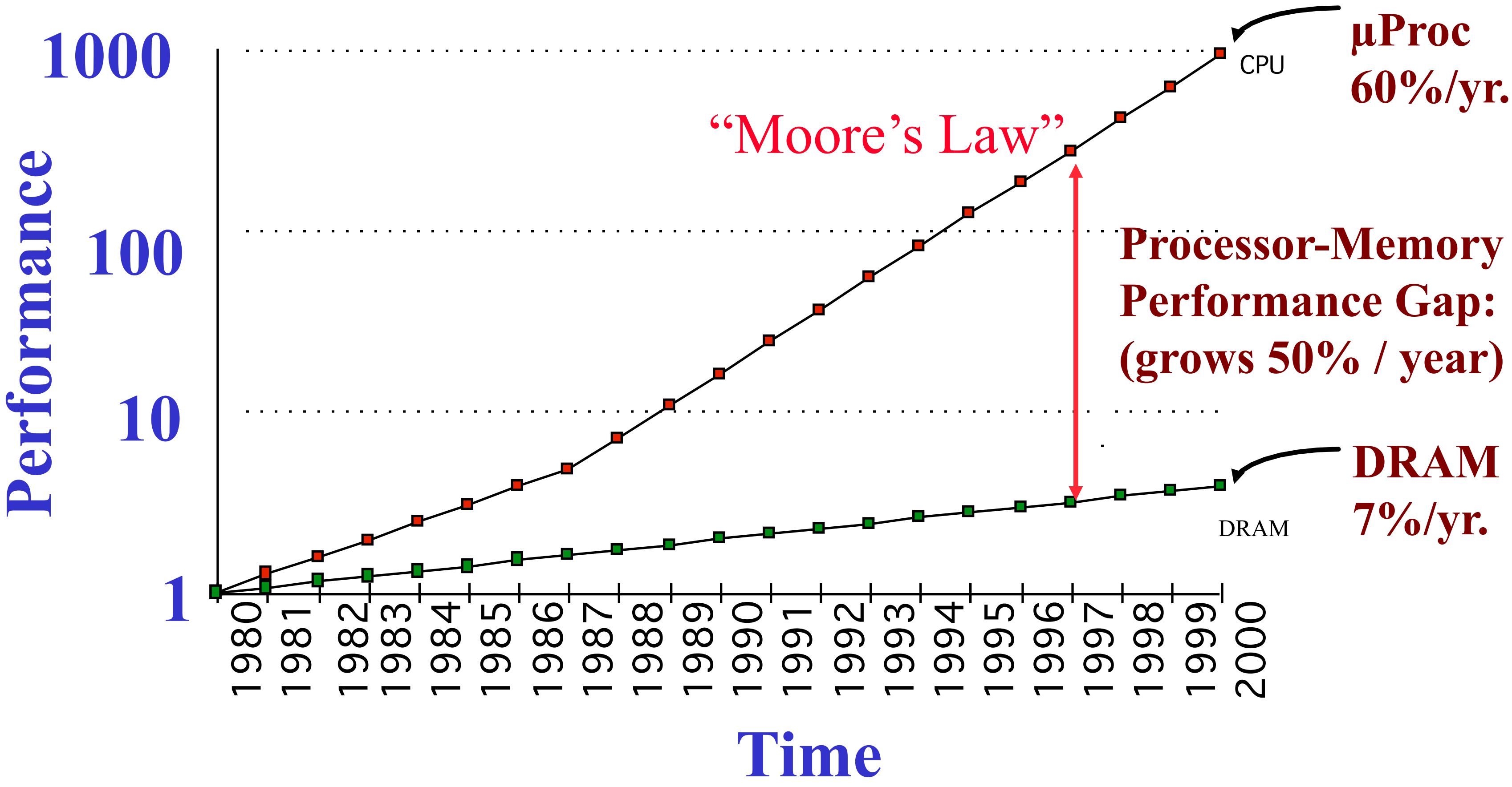
EE Times

Today’s lecture

- The memory hierarchy
- Cache Coherence and Consistency
- Implementing synchronization
- False sharing

The processor-memory gap

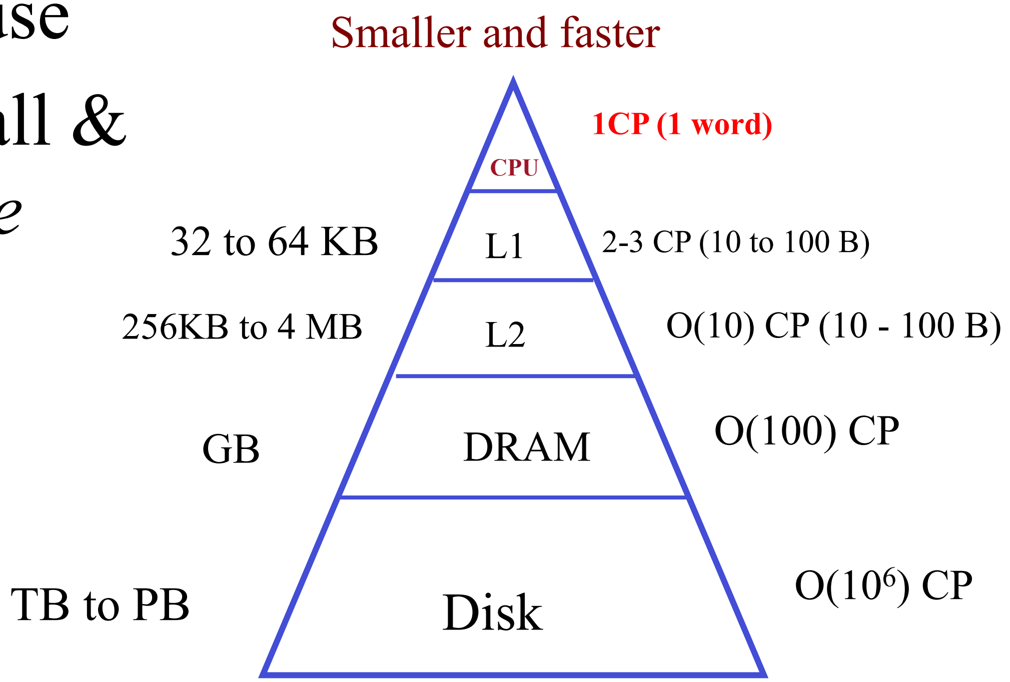
- The result of technological trends
- Difference in processing and memory speeds growing exponentially over time



An important principle: locality

- Memory accesses exhibit two forms of locality
  - Temporal locality (time)
  - Spatial locality (space)
- Often involves loops
- Opportunities for reuse
- Idea: construct a small & fast memory to *cache* re-used data

```
for t=0 to T-1
   for i = 1 to N-2
      u[i]=(u[i-1] + u[i+1])/2
```



Smaller and faster

CPU — 1CP (1 word)

32 to 64 KB — L1 — 2-3 CP (10 to 100 B)

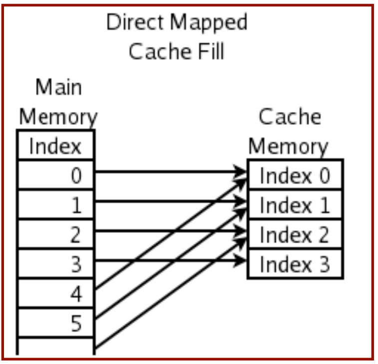
256KB to 4 MB — L2 — O(10) CP (10 - 100 B)

GB — DRAM — O(100) CP

TB to PB — Disk — O(10^6) CP

The Benefits of Cache Memory

- Let say that we have a small fast memory that is 10 times faster (access time) than main memory …
- If we find what we are looking for 90% of the time (a **hit**), the access time approaches that of fast memory



- $T_{access} = 0.90 \times 1 + (1\text{-}0.9) \times 10 = 1.9$
- Memory appears to be 5 times faster
- We organize the references by **blocks**
- We can have multiple levels of cache

Sidebar

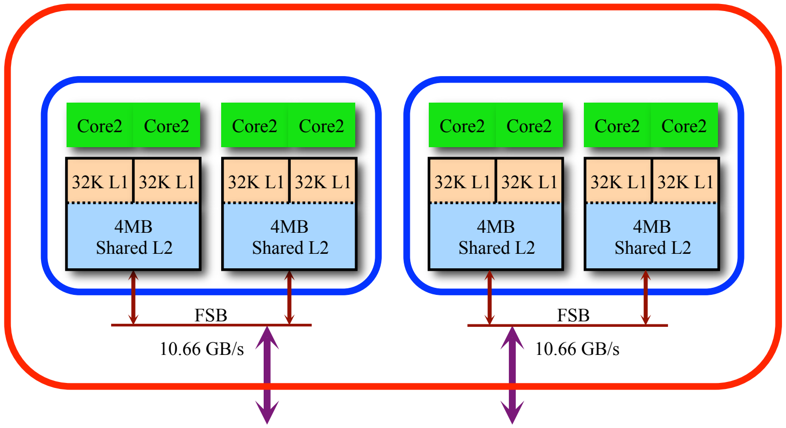
- If cache memory access time is 10 times faster than main memory …
- Cache "hit time" $T^{cache} = T^{main} / 10$
- T^{main} is the *cache miss penalty*
- And if we find what we are looking for f × 100% of the time ("cache hit rate") …
- Access time $= f \times T^{cache} + (1- f) \times T^{main}$

  $= f \times T^{main} /10 + (1- f) \times T^{main}$

  $= (1-(9f/10)) \times T^{main}$
- We are now $1/(1-(9f/10))$ times faster
- To simplify, we use $T^{cache} = 1$, $T^{main} = 10$

Different types of caches

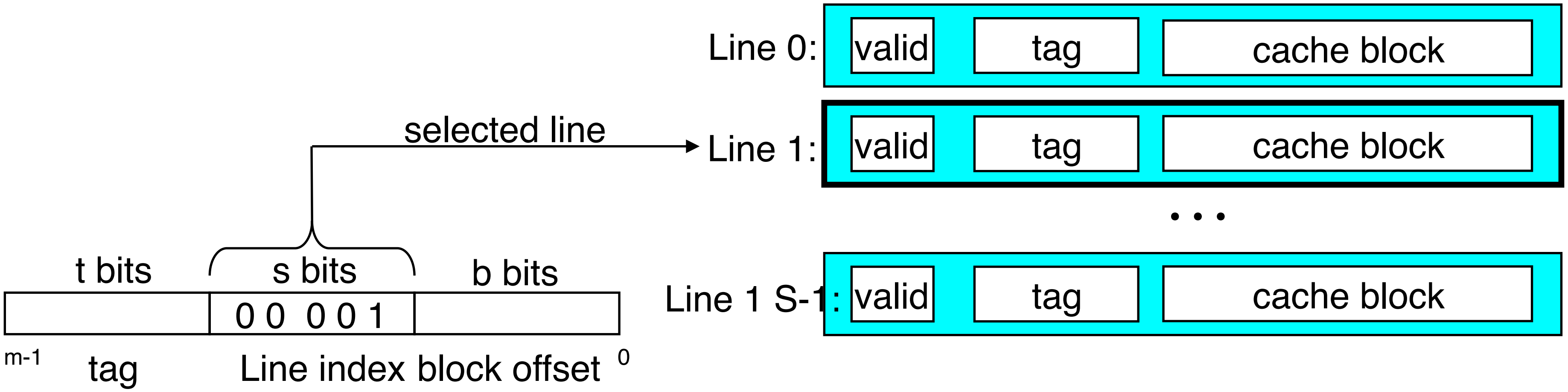
- Separate Instruction (I) and Data (D)
- Unified (I+D)
- Direct mapped / Set associative
- Write Through / Write Back
- Allocate on Write / No Allocate on Write
- Last Level Cache (LLC)
- Translation Lookaside Buffer (TLB)



Sam Williams et al.

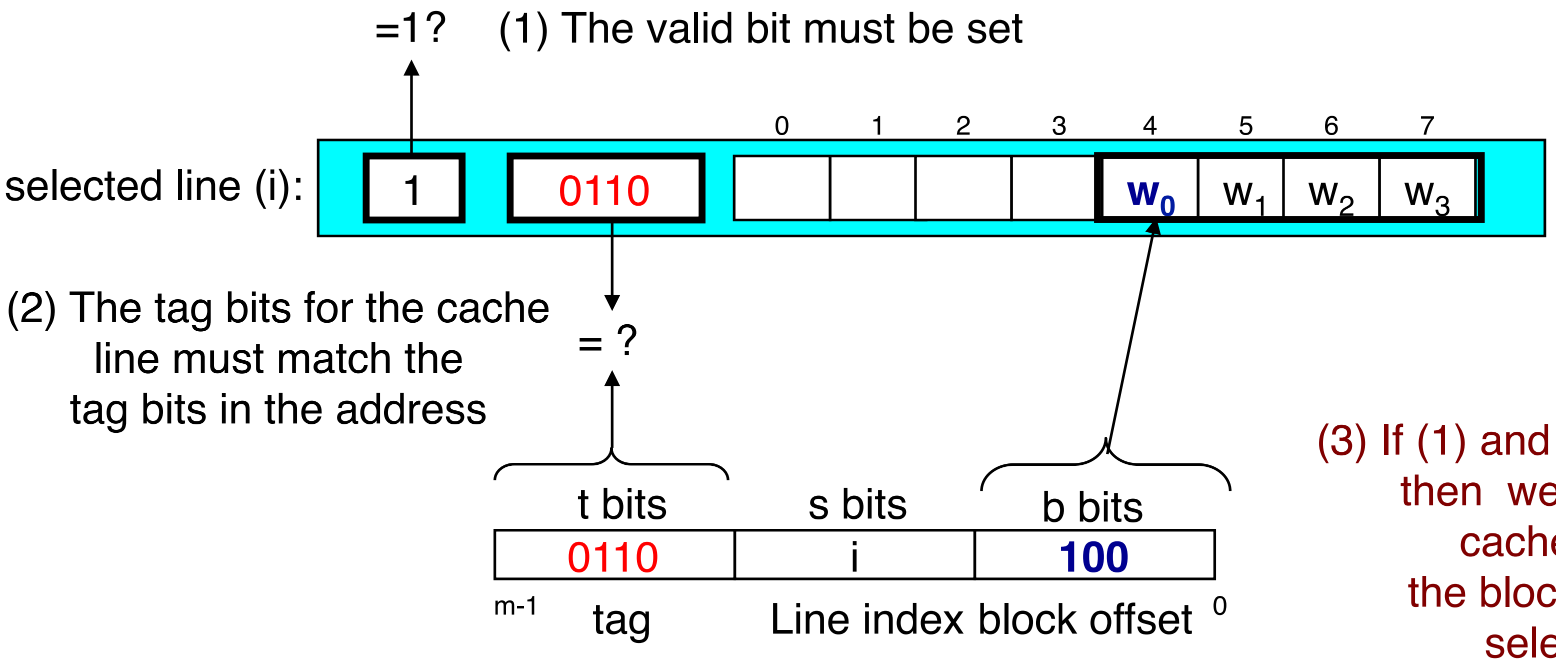
Direct mapped cache

- Simplest cache



Randal E. Bryant and
David R. O

Accessing a Direct mapped cache



=1? (1) The valid bit must be set

selected line (i):

(2) The tag bits for the cache line must match the tag bits in the address

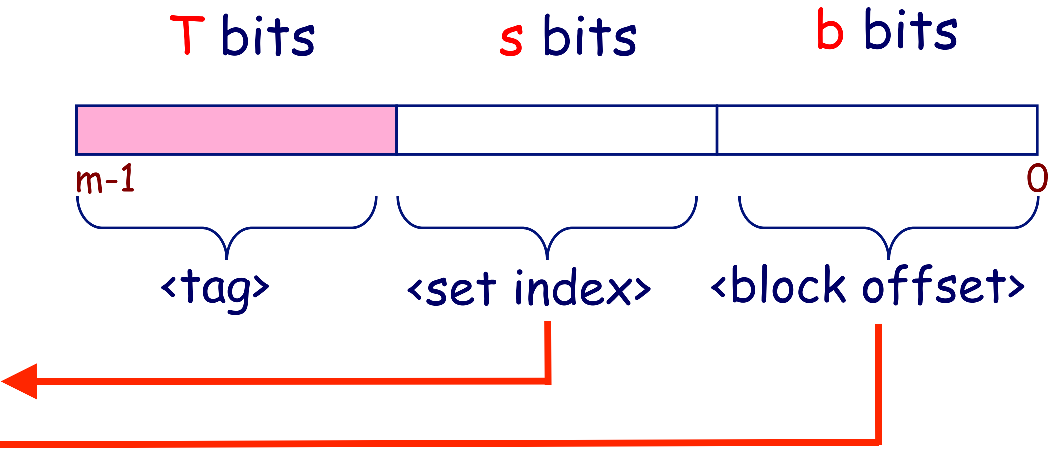
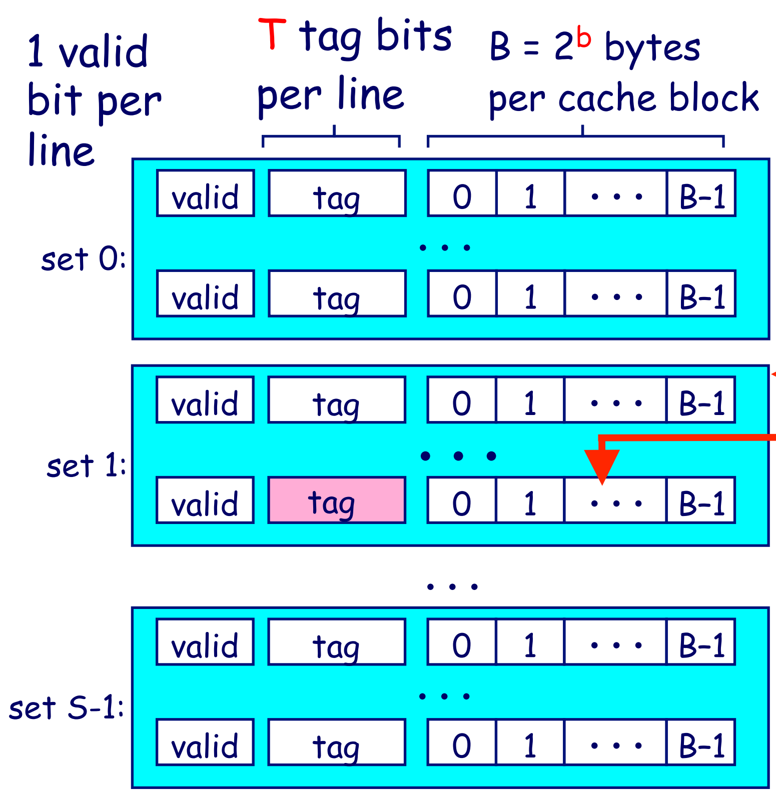
= ?

(3) If (1) and (2) are true, then we have a cache hit; the block offset selects starting byte.

Randal E. Bryant and David R. O

Set associative cache

- Why use the middle bits for the index?

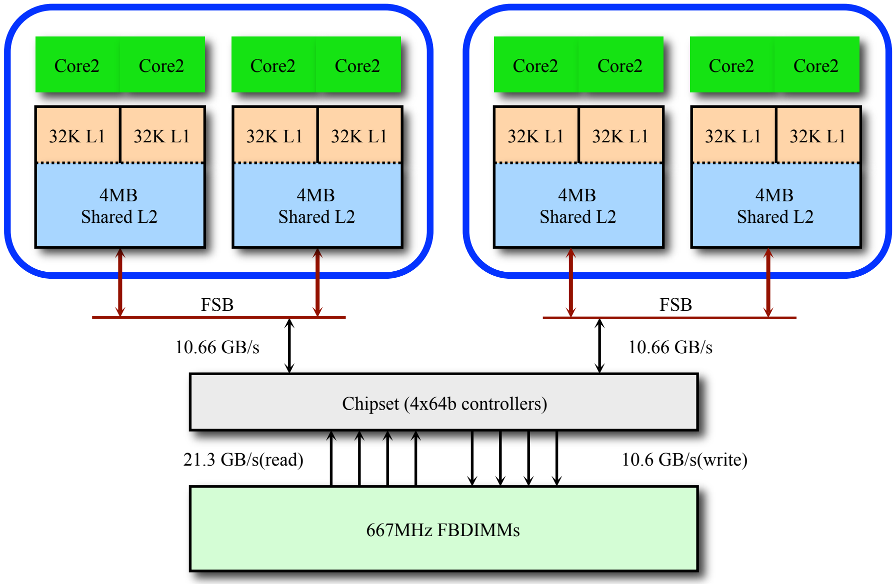



Randal E. Bryant and David R. O

The 3 C's of cache misses

- Cold Start
- Capacity
- Conflict

Line Size = 64B (L1 and L2)



Sam Williams et al.

Bang's Memory Hierarchy

- Intel "Clovertown" processor
- Intel Xeon E5355 (Introduced: 2006)
- Two "Woodcrest" dies (Core2) on a multichip module
- Two "sockets"
- *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Tab 2.16

techreport.com/articles.x/10021/2

Line Size = 64B (L1 and L2)

Access latency, throughput (clocks)

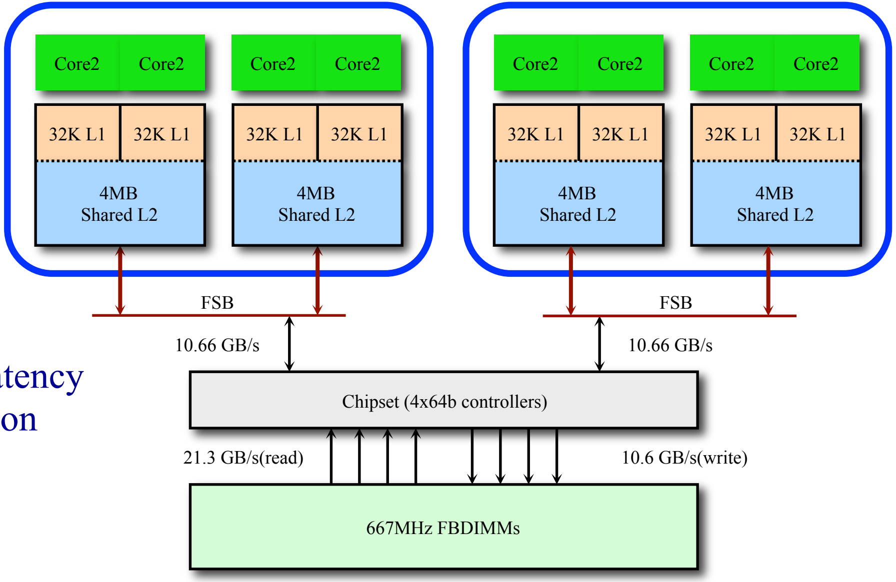
3, 1

14*, 2

Associativity

8

16



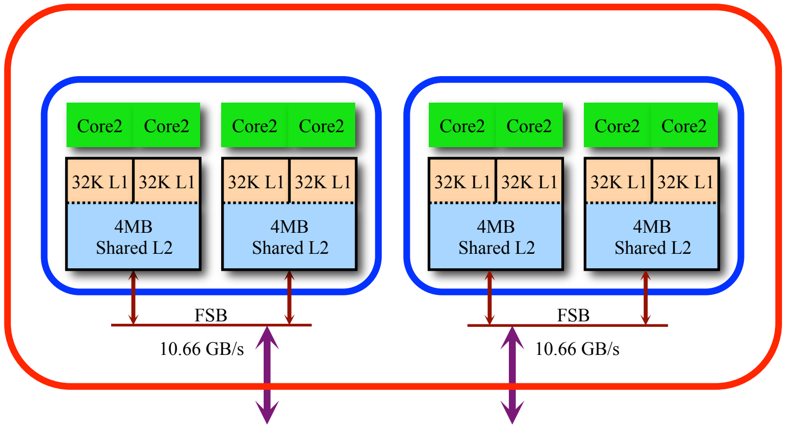
* Software-visible latency will vary depending on access patterns and other factors

Write update policy: Writeback

Sam Williams et al.

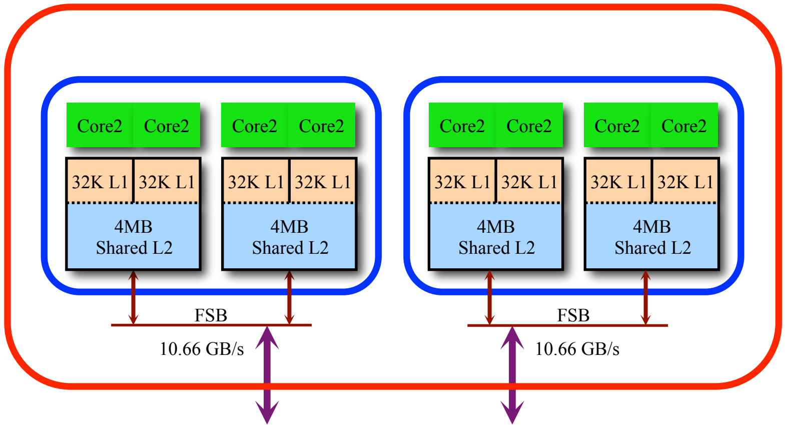
Examining Bang’s Memory Hierarchy

- /proc/cpuinfo summarizes the processor
  - vendor_id : GenuineIntel
  - model name : Intel(R) Xeon(R) CPU E5345 @2.33GHz
  - cache size : 4096 KB
  - cpu cores : 4
- processor : 0 through processor : 7



Detailed memory hierarchy information

- /sys/devices/system/cpu/cpu*/cache/index*/*
- Login to bang and view the files



Today's lecture

- The memory hierarchy
- Cache Coherence and Consistency
- Implementing synchronization
- False sharing

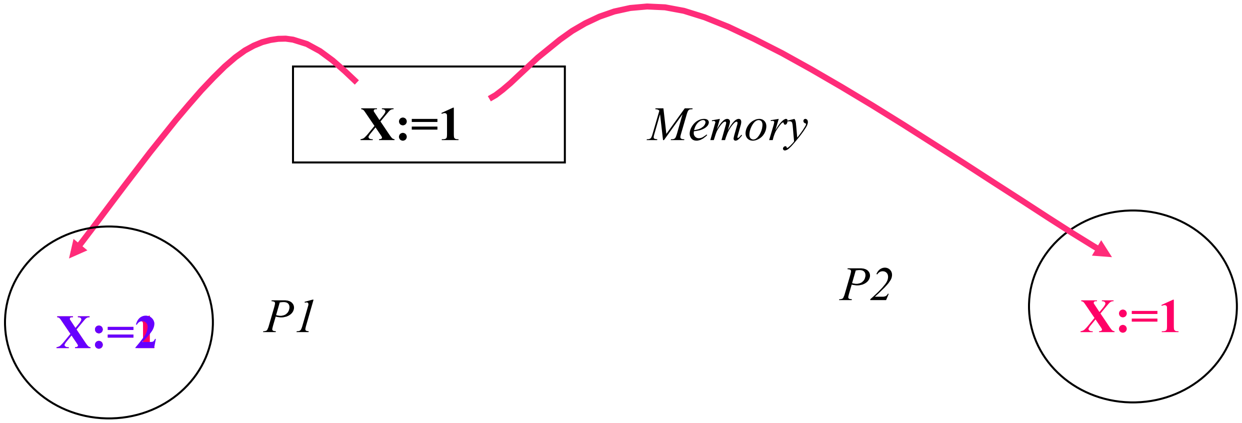
Cache Coherence

- A central design issue in shared memory architectures
- Processors may read and write the same cached memory location
- If one processor writes to the location, *all* others must *eventually* see the write

X:=1 *Memory*

Cache Coherence

- P1 & P2 load X from main memory into cache
- P1 stores 2 into X
- The memory system doesn't have a coherent value for X



X:=1

Memory

X:=2

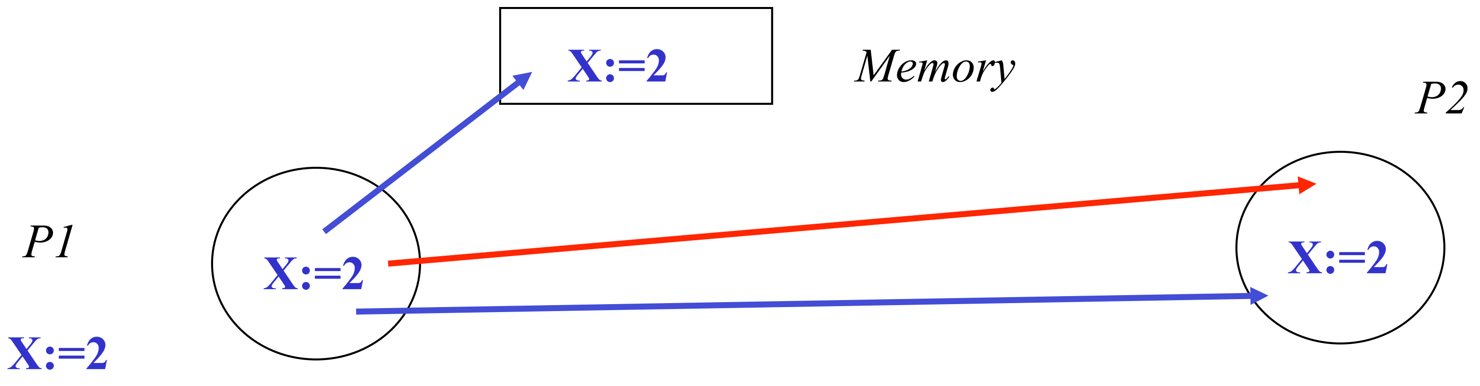
P1

P2

X:=1

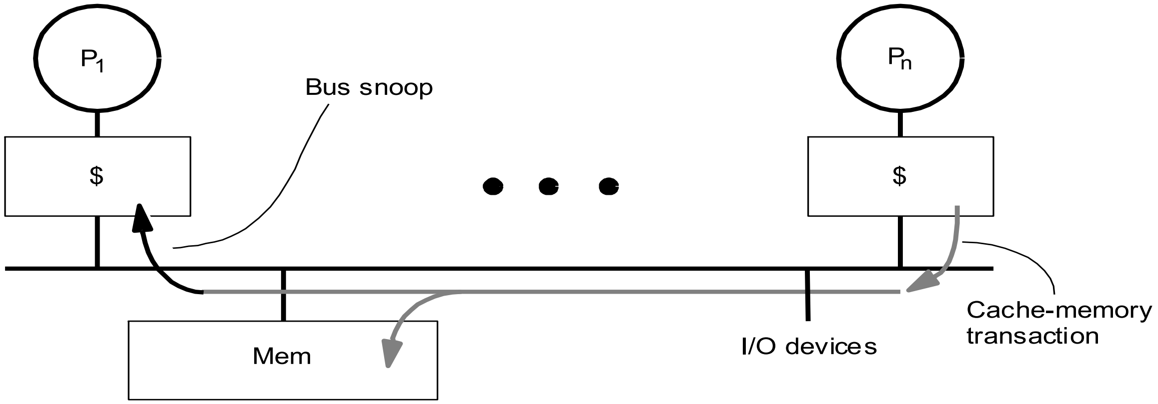
Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value
- Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write



SMP architectures

- Employ a *snooping protocol* to ensure coherence
- Cache controllers listen to bus activity updating or invalidating cache as needed



Patterson & Hennessey

Memory consistency and correctness

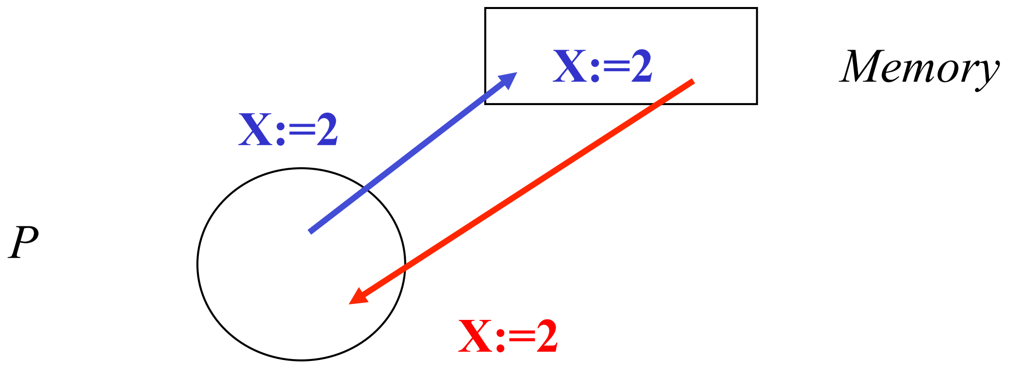
- Cache coherence tells us that memory will *eventually* be consistent
- The memory consistency policy tells us *when* this will happen
- Even if memory is consistent, changes don't propagate instantaneously
- These give rise to correctness issues involving program behavior

Memory consistency

- A memory system is consistent if the following 3 conditions hold
  - Program order (you read what you wrote)
  - Definition of a coherent view of memory (“eventually”)
  - Serialization of writes (a single frame of reference)

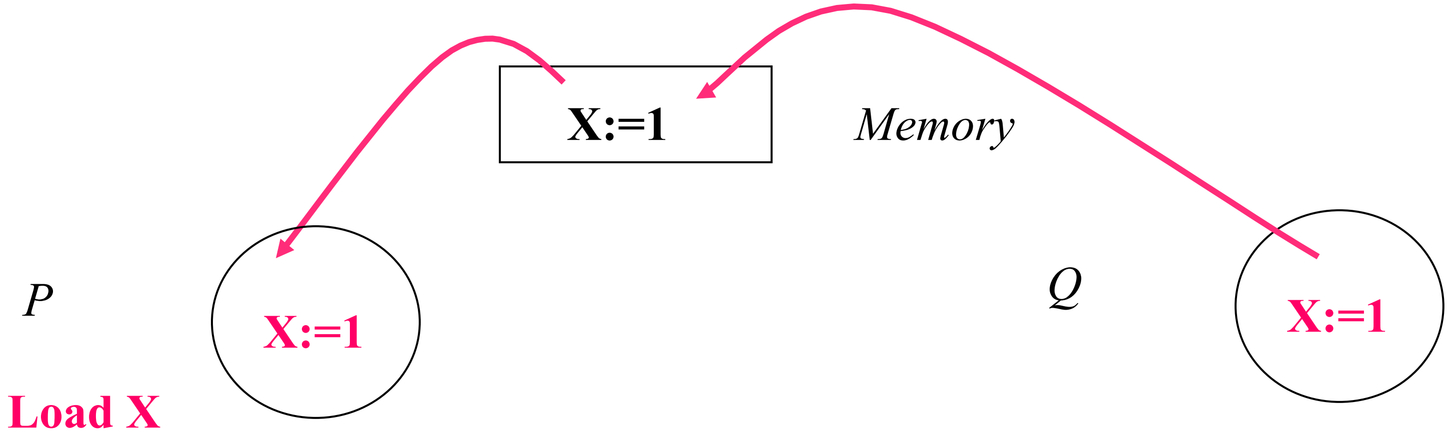
Program order

- If a processor writes and then reads the same location X, and there are no other intervening writes by other processors to X , then the read will always return the value previously written.



Definition of a coherent view of memory

- If a processor P reads from location X that was previously written by a processor Q , then the read will return the value previously written, if a sufficient amount of time has elapsed between the read and the write.



Serialization of writes

- If two processors write to the same location X, then other processors reading X will observe the same the sequence of values in the order written
- If 10 and then 20 is written into X, then no processor can read 20 and then 10

Memory consistency models

- Should it be impossible for both **if** statements to evaluate to true?
- With sequential consistency the results should always be the same provide that
  - Each processor keeps its access in the order made
  - We can't say anything about the ordering across different processors: access are interleaved arbitrarily

Processor 1	Processor 2
A=0	B=0
…	…
A=1	B=1
if (B==0) …	**if** (A==0) …

Undefined behavior in C++11

Global

```
int  x, y;
```

Thread 1	Thread 2
`x =17`	`cout << y << " ";`
`y = 37;`	`cout << x << endl;`

- Compiler may rearrange statements to improve performance
- Processor may rearrange order of instructions
- Memory system may rearrange order that writes are committed
- Memory might not get updated; “eventually can be a long time” (though in practice it’s often not)

Undefined behavior in earlier versions of C++

Global

```
int  x, y;
```

Thread 1	Thread 2
`char c;`	`char b;`
`c=1;`	`b =1;`
`int x=c;`	`int y=b;`

- In C++11, x=1 and y=1;
  they are "separate memory locations"
- But in earlier dialects you might get 1&0, 0&1, 1&1
- The linker could allocate b and c next to each other in the same word of memory
- Modern processors can't write a single byte, so they have to do read-modify-write

Today's lecture

- The memory hierarchy
- Cache Coherence and Consistency
- Implementing synchronization
- False sharing

Implementing Synchronization

- We build mutex and other synchronization primitives with special atomic operations, implemented with a single machine instruction, e.g. CMPXCHG
- Do atomically: compare contents of memory location loc to expected; if they are the same, modify the location with newval

```
CAS (*loc ,  expected ,  newval  ) {
    if (*loc ==  expected ) {
        *loc =  newval;
         return 0;
    }
    else
      return 1
```

- We can then build mutexes with CAS

```
Lock( *mutex ) {
   while (CAS ( *mutex , 1, 0)) ;
}

Unlock( *mutex ) { *mutex =  1; }
```

Memory fences

- How are we assured that a value updated within a critical section becomes visible to all other threads?
- With a *fence* instruction, e.g. MFENCE
- "A serializing operation guaranteeing that every load and store instruction that precedes, *in program order,* the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction is globally visible." [Intel 64 & IA32 architectures software developer manual]
- Also see www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

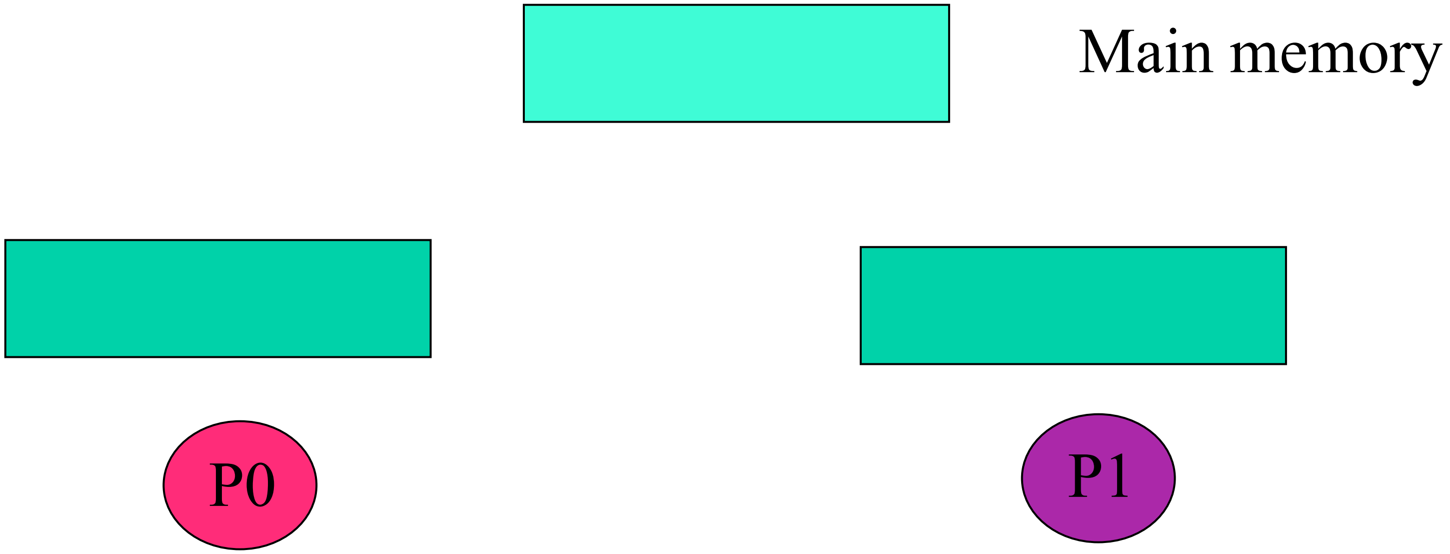
```
mutex mtx;
…
mutex.mtx.lock();
sum += local sum;
mutex.mtx.unlock();
```

Today's lecture

- The memory hierarchy
- Cache Coherence and Consistency
- Implementing synchronization
- False sharing

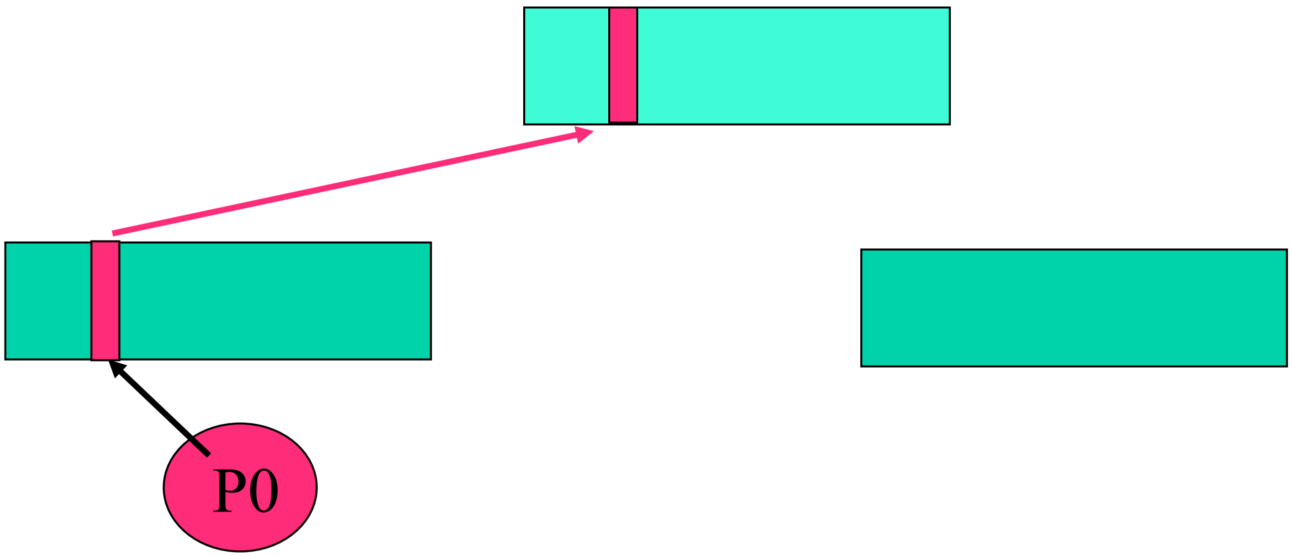
False sharing

- Consider two processors that write to different locations mapping to different parts of the same cache line



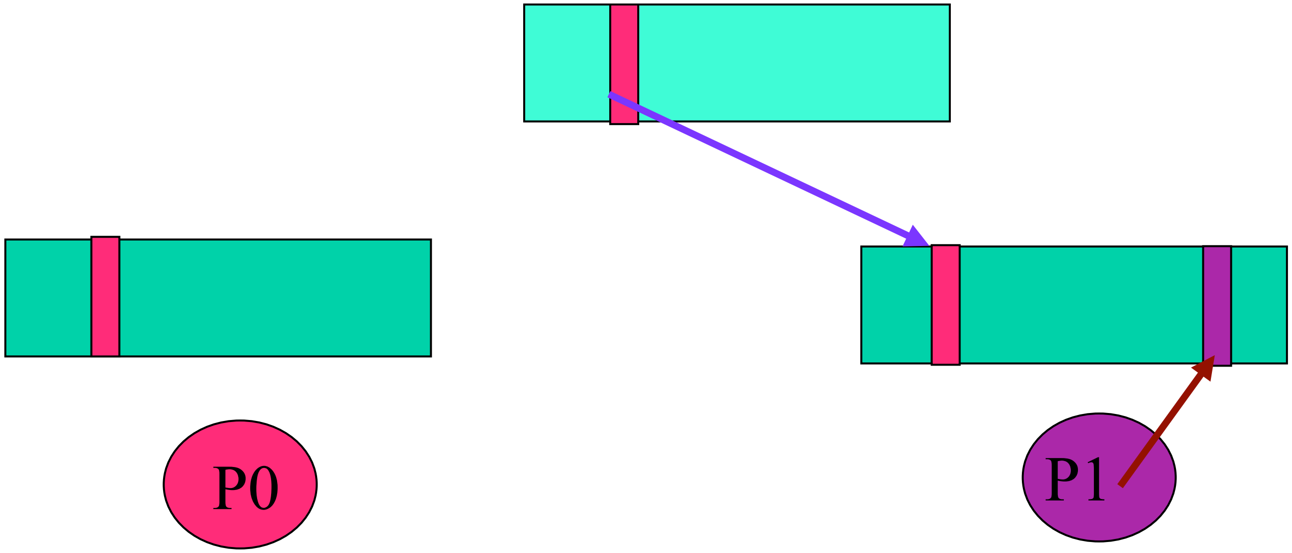
False sharing

- P0 writes a location
- Assuming we have a write-through cache, memory is updated



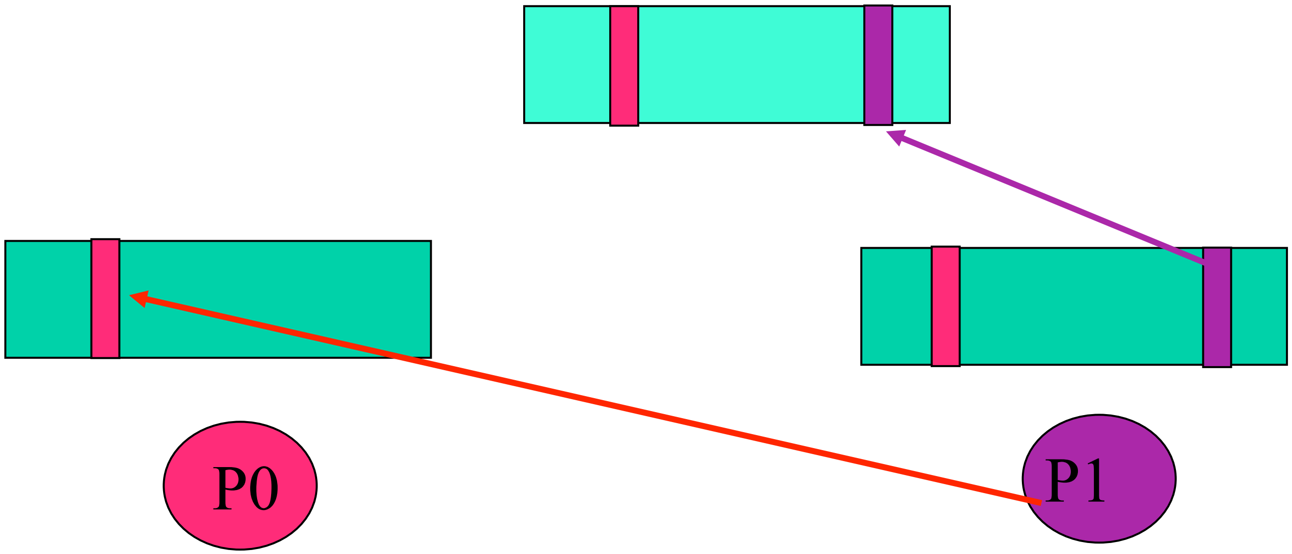
False sharing

- P1 reads the location written by P0
- P1 then writes a different location in the same block of memory



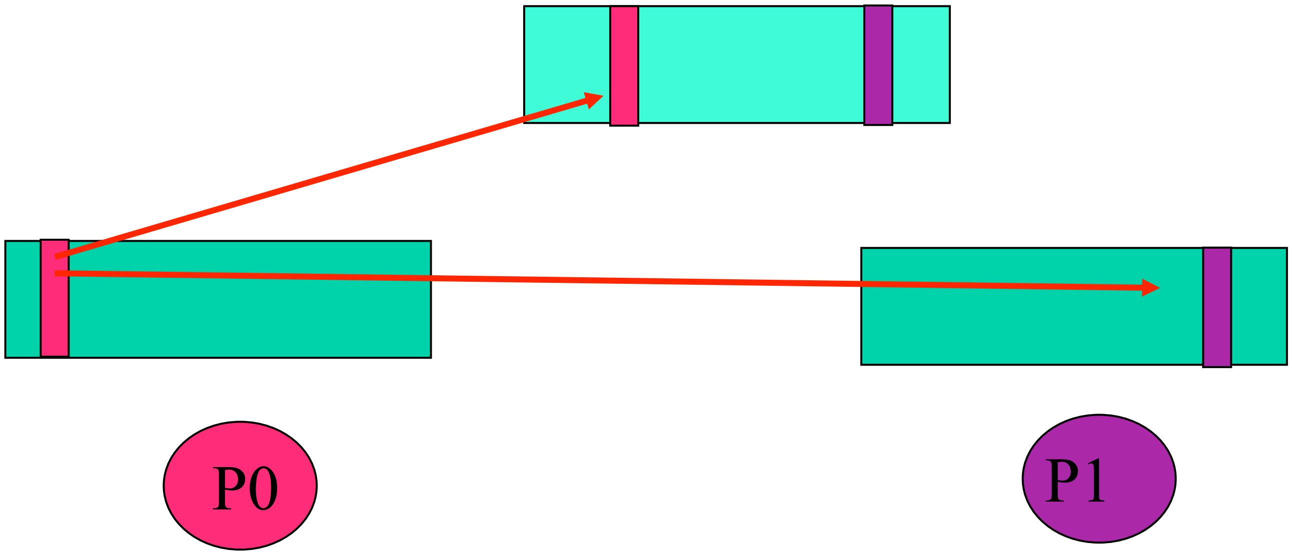
False sharing

- P1's write updates main memory
- Snooping protocol invalidates the corresponding block in P0's cache



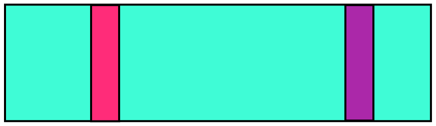
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another's cache



Eliminating false sharing

- Cleanly separate locations updated by different processors
  - Manually assign scalars to a pre-allocated region of memory using pointers
  - Spread out the values to coincide with a cache line boundaries



How to avoid false sharing

- Reduce number of accesses to shared state
- False sharing occurs a small fixed number of times

```
static int counts[];
for (int k = 0; k<reps; k++)
      for (int r = first; r <= last; ++ r)
            if ((values[r] % 2) == 1)
                  counts[TID]++;
```

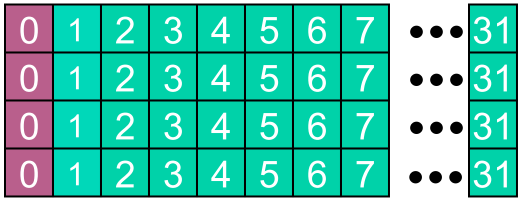
```
int _count = 0;
for (int k = 0; k<reps; k++){
     for (int r = first; r <= last; ++ r)
            if ((values[r] % 2) == 1)
                  _count++;
     counts[TID] = _count;
}
```

4.7s, 6.3s, 7.9s, 10.4 [NT=1,2,4,8]

3.4s, 1.7s, 0.83, 0.43 [NT=1,2,4,8]

Spreading

- Put each counter in its own cache line



0	1	2	3	4	5	6	7	•••	31
0	1	2	3	4	5	6	7	•••	31
0	1	2	3	4	5	6	7	•••	31
0	1	2	3	4	5	6	7	•••	31

```
static int counts[];
for (int k = 0; k<reps; k++)
        for (int r = first; r <= last; ++ r)
                if ((values[r] % 2) == 1)
                        counts[TID]++;
```

```
static int counts[][LINE_SIZE];
for (int k = 0; k<reps; k++)
        for (int r = first; r <= last; ++ r)
                if ((values[r] % 2) == 1)
                counts[TID][0]++;
```

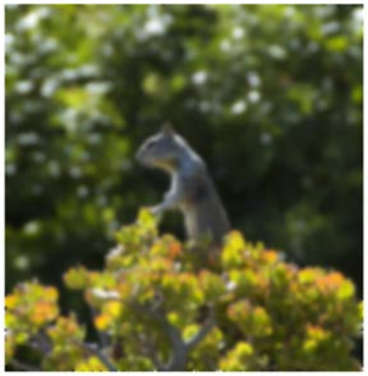
	NT=1	NT=2	NT=4	NT=8
Unoptimized	4.7 sec	6.3	7.9	10.4
Optimized	4.7	5.3	1.2	1.3

Cache performance bottlenecks in nearest neighbor computations

- Recall the image smoothing algorithm

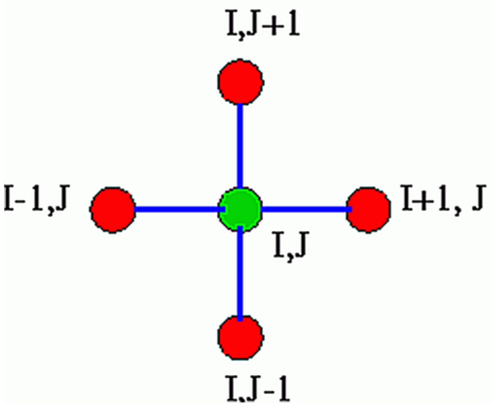
```
for (i,j) in 0:N-1 x 0:N-1
  $I^{new}[i,j] = ( I[i-1,j] + I[i+1,j]+ I[i,j-1]+ I[i, j+1])/4$
```

Original



100 iter

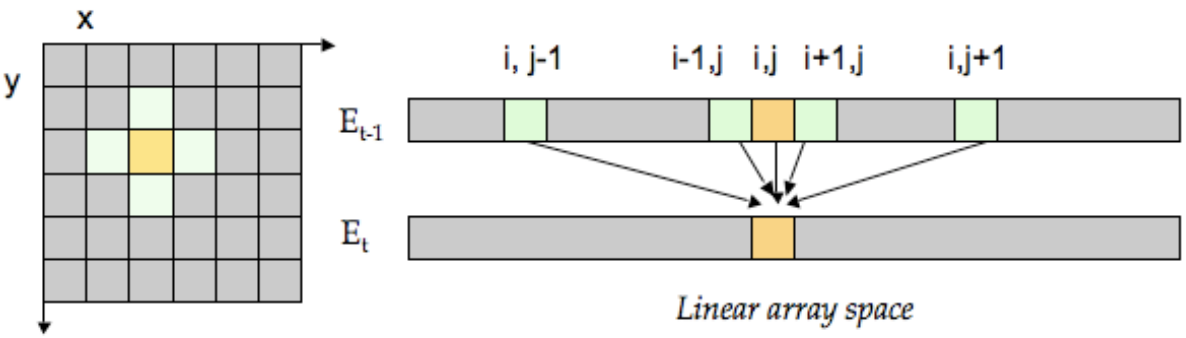
1000 iter



Memory access pattern

- Some nearest neighbors in space are far apart in memory
- Stride = N along the vertical dimension

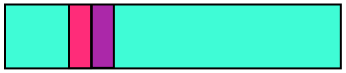
```
for (i,j) in 0:N-1 x 0:N-1
  $I^{new}[i,j] = ( I[i-1,j] + I[i+1,j]+ I[i,j-1]+ I[i, j+1])/4$
```



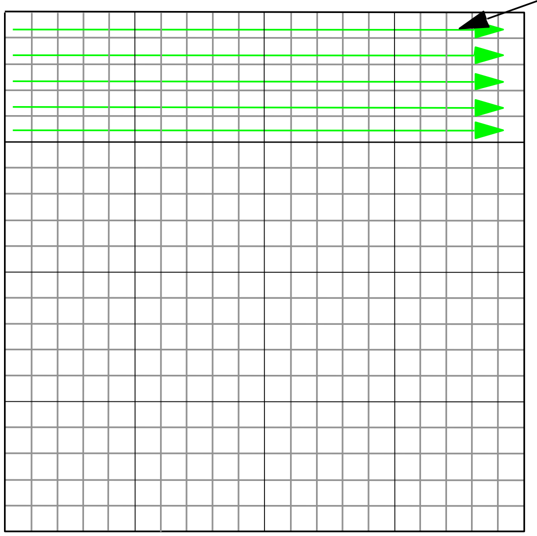
Linear array space

False sharing and conflict misses

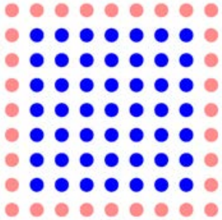
- False sharing involves internal boundaries, poor spatial locality, cache line internally fragmented
- Large memory access strides: conflict misses, poor cache locality
- Even worse in 3D: large strides of N^2
- Contiguous access on a single processor

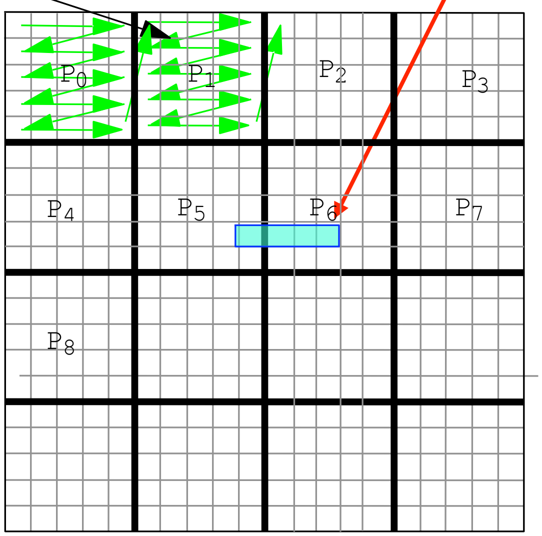


Cache block straddles boundary



On a single processor





On multiple processors

Parallel Computer Architecture, Culler, Singh, & Gupta