

Lecture 5

Announcements

- New Open Lab Hours schedule:
fixed times each week
<https://goo.gl/iXFi1i>
- B230 is unavailable to us on Wednesdays,
from 9AM to 5PM
- Use the ticket system during times that
contact hours have been scheduled, staff can
also meet you in the public area outside the
labs: <https://autograder.ucsd.edu/login>

Cores and threads

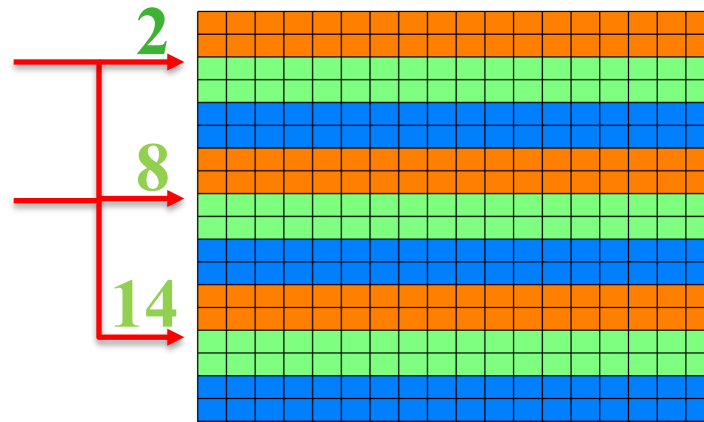
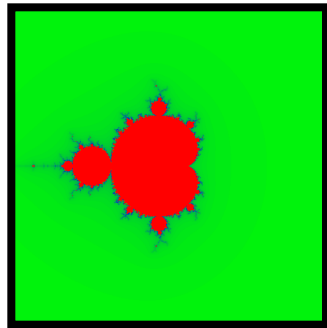
- In this course, we will get “close to the metal” in order to squeeze as much performance as we can out of the processor
- We will generally assume that once a thread is assigned to a core, that it remains there: this is approximation, but is good enough
- For us threads are really a means of an end: a unit for assigning work to cores
- More generally, threads are an abstraction that allow us to do other things: like multiplexing cores. We won't be doing that.

Today's lecture

- Processor Self-scheduling
- Cache memories

Non-uniform workloads

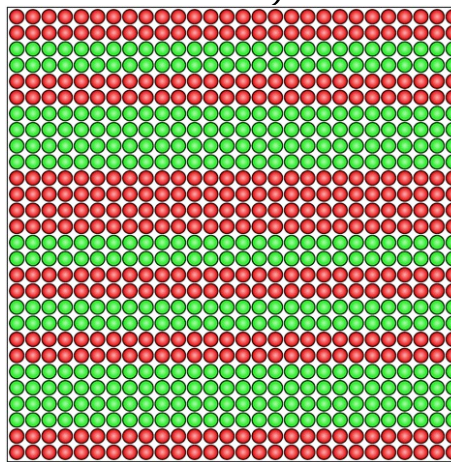
- In the Mandelbrot set assignment, some points take longer to complete than others, so the work is not uniformly distributed
- A block cyclic decomposition can balance the workload: core k gets strips starting at rows $\text{CHUNK} * k, \text{CHUNK} * k + 1 * \text{NT} * \text{CHUNK}, \text{CHUNK} * k + 2 * \text{NT} * \text{CHUNK}, \dots$
- Core 1 gets strips @ $2 * 1, 2 + 1 * 2 * 3, 2 + 2 * 2 * 3$



NT=3
CHUNK=2

Second approach: dynamic scheduling

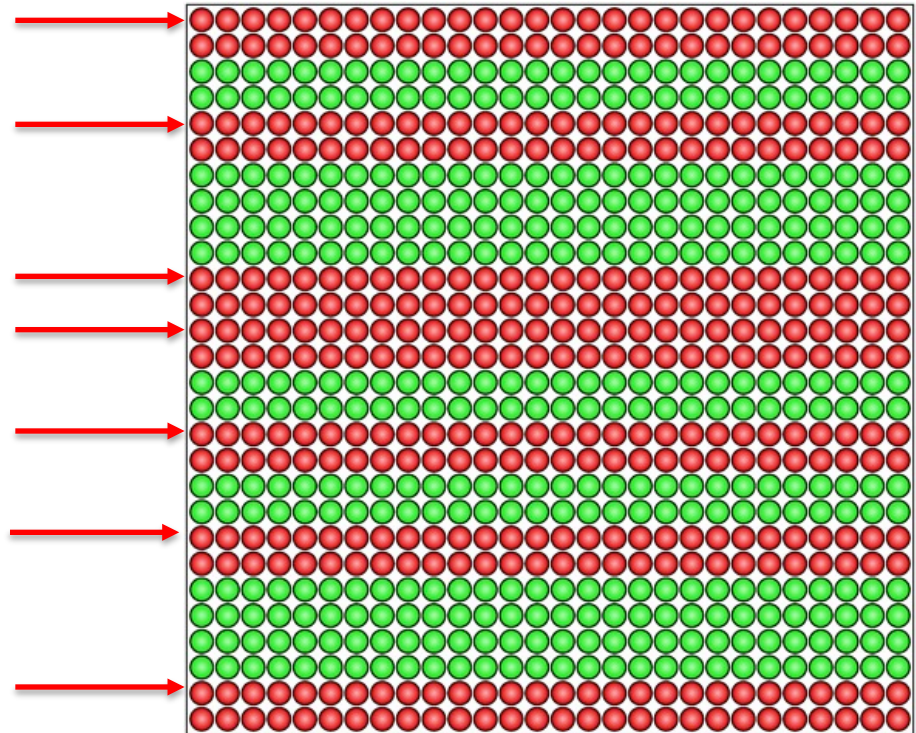
- Block cyclic decomposition is a static method
- The workload assignment can't respond to the workload distribution: it is the same for all
- Problem: what if the workload distribution correlates with the block cyclic thread assignment?
- *Processor self scheduling*: threads pick up work dynamically, on demand, with a chunk size



NT=2
CHUNK=2

How does self scheduling work?

```
SelfScheduler S(n, NT, Chunk);  
h=1/(n-1)  
while ( S.getChunk(startRow))  
    for i = startRow to startRow+ Chunk - 1  
        for j = 0; j < n; j++  
            z=complex(i*h,j*h)  
            do  
                 $z = z^2 + c$   
            while (|z| < 2 )  
        end for  
    end for  
end while
```



Implementation

- We increment a counter inside a critical section
- OMP and C++ implementations
- `_counter` is a protected member of `SelfScheduler`

```
boolean getChunk(int& startRow){
```

```
    Begin critical section
```

```
        k = _counter;
```

```
        _counter += _chunk;
```

```
    End critical section
```

```
    if ( k > (_n - _chunk)
```

```
        return false;
```

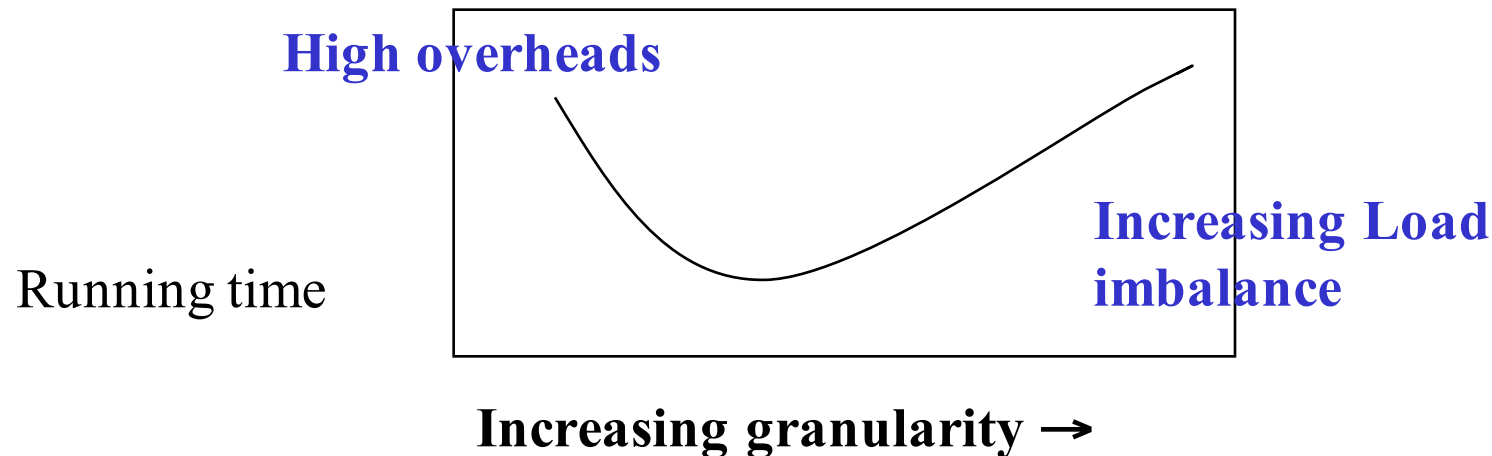
```
    startRow= k;
```

```
    return true;
```

```
}
```

Tradeoffs in dynamic scheduling

- Dynamic workload assignments
- Each thread samples a unique (and disjoint) set of indices, changes from run to run
- A shared counter or work queue represents the work
- User tunes work granularity (chunk size) trading off the overhead of workload assignment against increased load imbalance
 - Finest granularity: each point is a separate task
 - Coarsest granularity: one block per processor



When might decreasing the chunk size hurt performance?

- A. We will hand out more chunks of work
- B. Data from adjacent rows is less likely to reside in cache
- C. The workloads will be balanced less evenly
- D. A & B
- E. B & C

Iteration to thread mapping with block cyclic

for ($i = 0; i < N; i++$)

$\text{iters}[i] = \text{thread assigned to row } i$

$N = 9, \# \text{ of threads} = 3 \text{ (BLOCK)}$

0 0 0 1 1 1 2 2 2

$N = 16, \# \text{ threads} = 4, \text{BLOCK CYCLIC}(\text{Chunk} = 2)$

0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3

$N=9, \# \text{ threads} = 3, \text{BLOCK CYCLIC}(\text{Chunk} = 2)$

0 0 1 1 2 2 0 0 1

Which are plausible dynamic assignments of iterations to threads with, $N=16$,
 $NT=4$, $Chunk=2$

A. 3 3 0 0 1 1 2 2 3 3 3 3 3 3 3 3

B. 2 3 3 2 0 0 1 1 2 2 2 2 3 3 2 1

C. 2 2 3 3 0 0 1 1 2 2 2 2 0 0 2 2

D. A&B

E. A&C

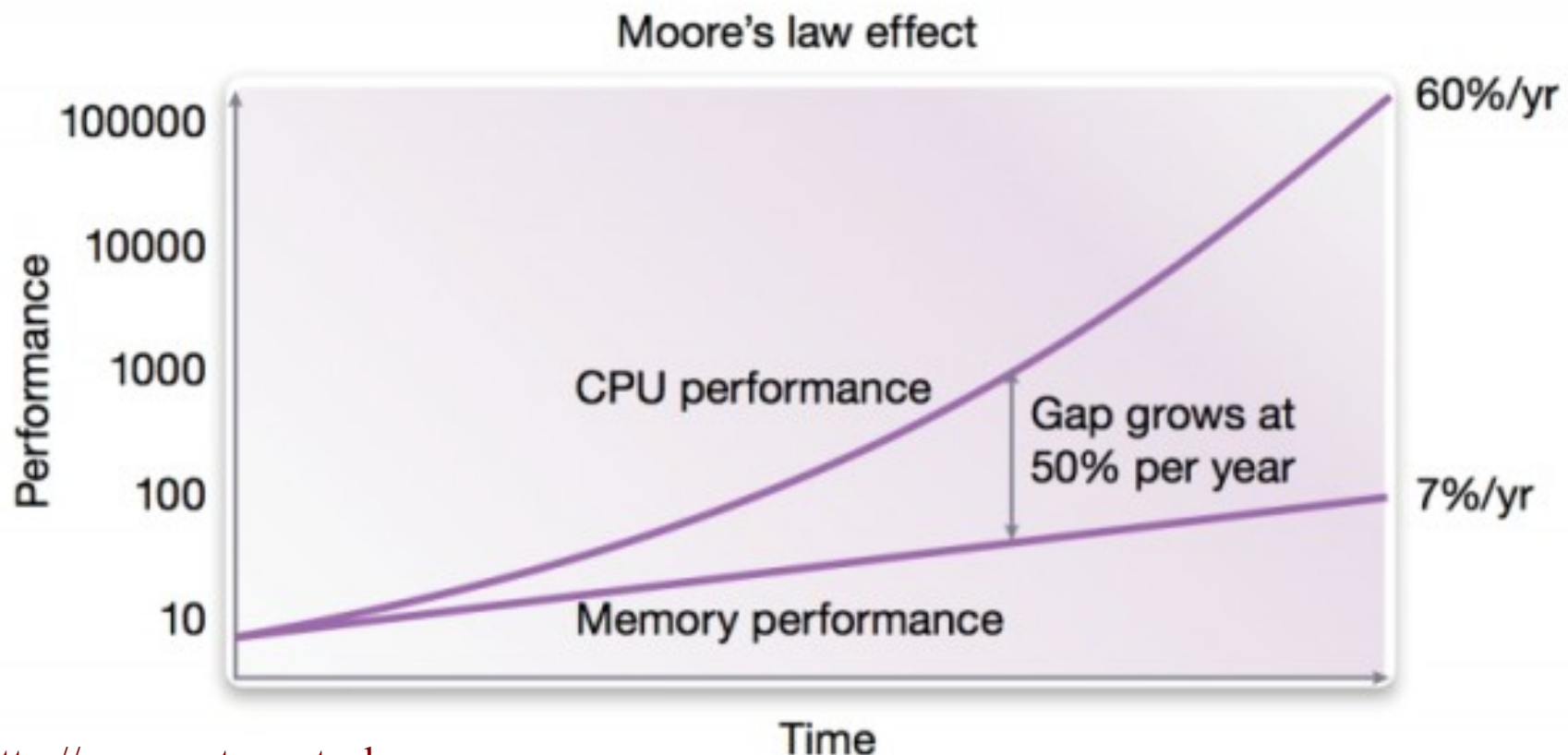
```
while ( S.getChunk(startrow))  
    iters[startRow:startRow+NT-1] = Thread ID();
```

Today's lecture

- Processor Self-scheduling
- Cache memories

The processor-memory gap

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



<http://www.extremetech.com>
(see <http://tinyurl.com/ohmbo7y>)

An important principle: locality

- Memory accesses exhibit two forms of locality
 - Processor likely to access a location again in the near future (Temporal locality (time))
 - Processor likely to access a location nearby to the current access (Spatial locality (space))
- Often involves loops
- Opportunities for reuse

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

Which of these loops exhibit spatial locality?

A. for i = 1 to N-2
 $u[i] = (u[i-1] + u[i+1]) / 2$

B. for i = N-2 to 1 by -1
 $u[i] = (u[i-1] + u[i+1]) / 2$

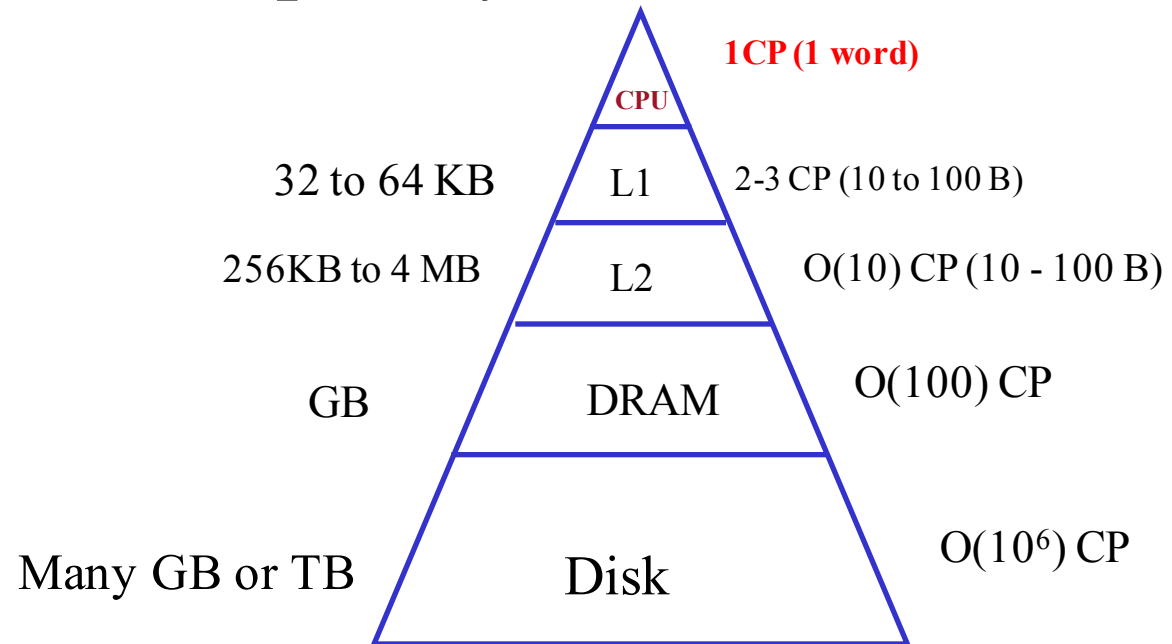
C. for i = 1 to N-2 by 1024
 $u[i] = (u[i-1] + u[i+1]) / 2$

D. for i = 1 to N-2 by 1024
 $u[J[i]] = \text{sqrt}(u[J[i]])$

E. None of them do

Memory hierarchies

- Enable reuse through a hierarchy of smaller but faster memories
- Put things in faster memory that we re-use frequently



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

Access latency,
throughput (clocks)

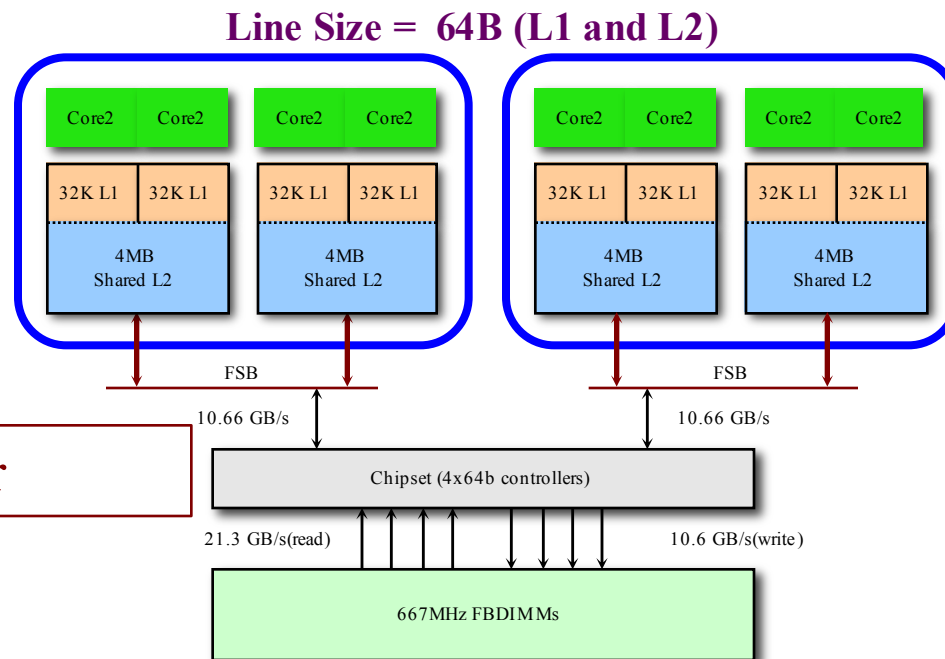
3, 1

14*, 2

150/600

<http://goo.gl/QxvSVr>

* Will vary depending on
access patterns and
other factors



Associativity

8

16

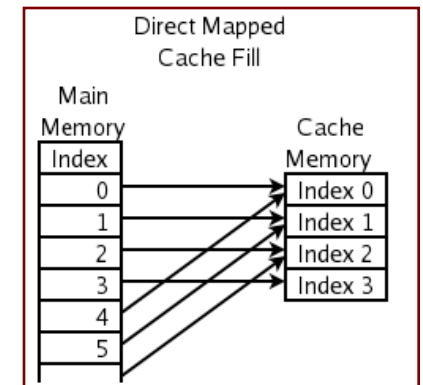
Line size (bytes):
64

Write update policy:
Writeback

Sam Williams et al.

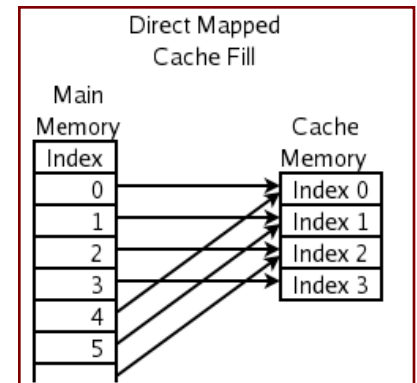
How does a cache work?

- Organize memory and cache into blocks called *lines* (typically 64 bytes)
- Simplest cache: a *direct mapped cache*
- We map each line in main memory to a unique line in cache: many-to-1 mapping
- The first time we access an address within a given line in main memory, we load the line into cache: a cache *miss*
- The next time we access any address within that line, we have a *cache hit*: we pay the cost of accessing the cache line only, not memory
- With a 1 level cache, the miss penalty is more or less the cost of accessing memory



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 (cache *hit* rate), access time approaches that of fast memory
- $T_{\text{access}} = 0.90 \times 1 + (1-0.9) \times 10 = 1.9$
- Memory appears to be 5 times faster
- We can have multiple levels of cache



If our L1 hit rate is 95%, L2 hit rate = 99%, what is the effective memory access time on Bang where L1 hit= 3CP, L2 = 12 CP, Memory = 160CP?

A. $0.95 \times 3 + (1 - 0.95) \times 0.99 \times 12 + 0.05 \times 0.01 \times 160 = 3.5$

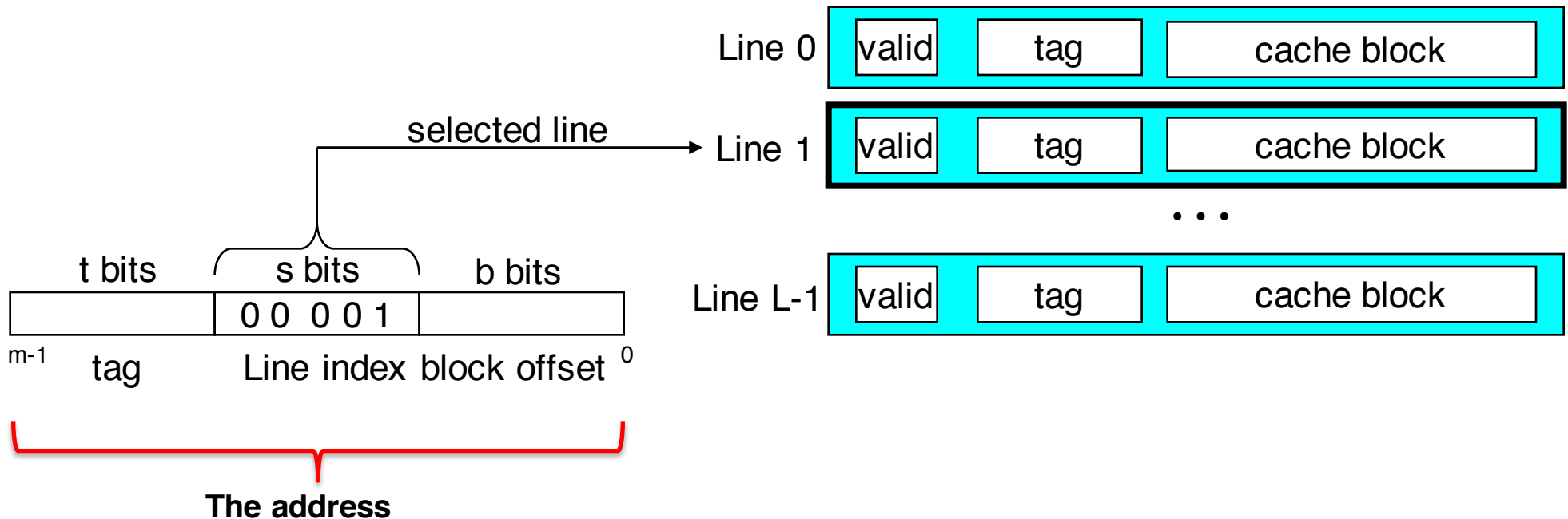
B. $0.95 \times 3 + 0.99 \times 12 + (1 - 0.99) \times 160 = 16.3$

C. Neither

Direct mapped cache



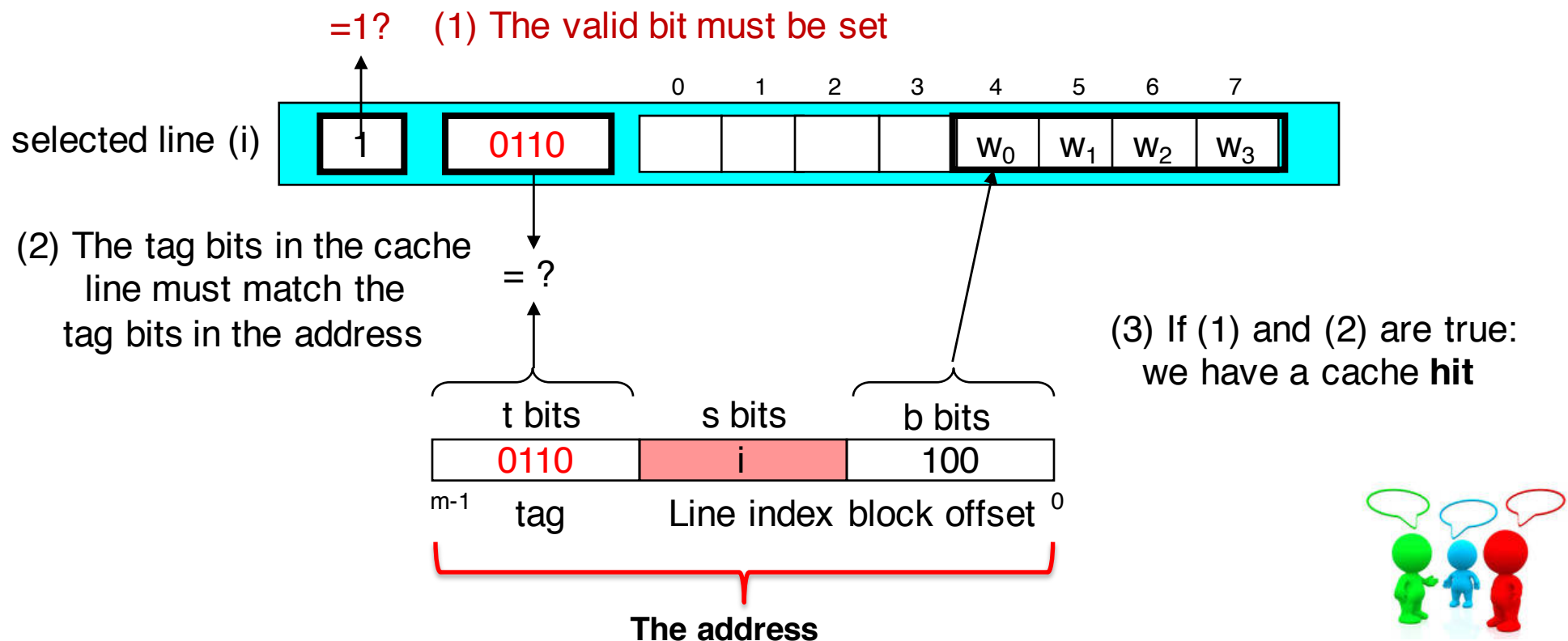
- Look up the line indexed by the line index
- Like a hash table: match the stored tag against the higher order address bits



Randal E. Bryant and David R. O

Issues in using a cache

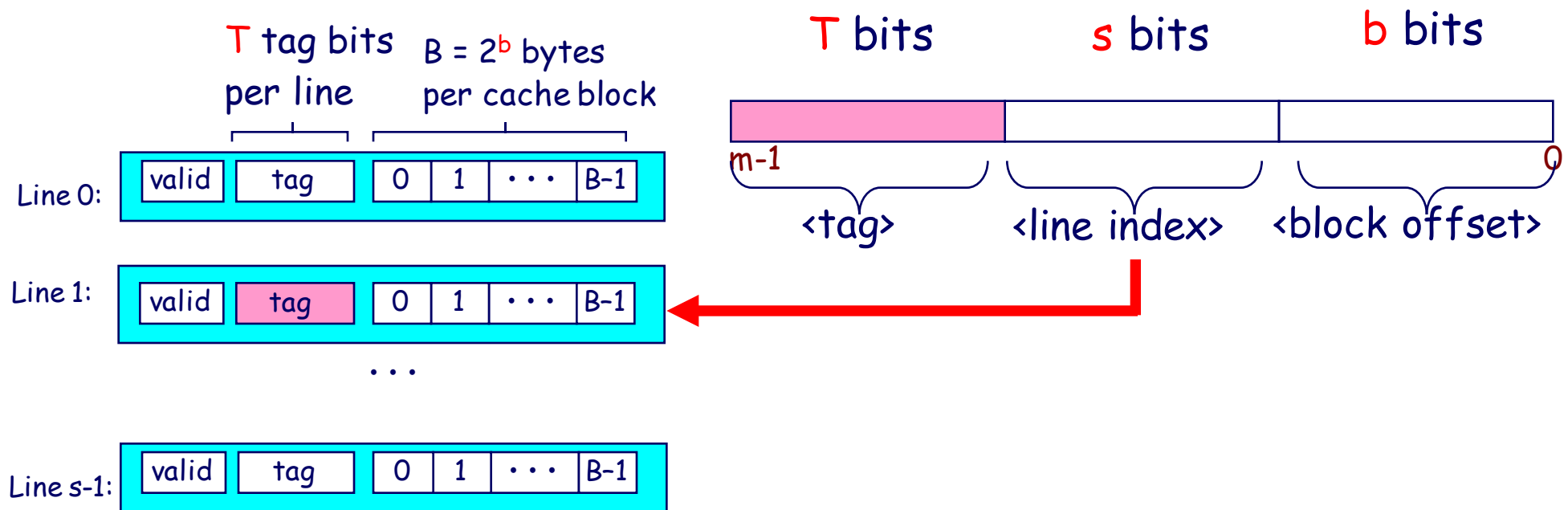
- Where can we place the block?
- How do we find the block?
- Which block should we on a miss?
- What happens on a write?



Why do we use the middle bits for indexing the cache rather than the higher order bits?

A. Consecutive memory lines map to different cache lines

B. Consecutive memory lines map to same cache entry



What happens on a write?

- Write Through / Write Back
 - Write through to memory
 - Write to cache only – when do we update memory?
- Allocate on Write / No Allocate on Write: whether or not we load cache on a miss, or write to main memory only

Why might a write-back cache be useful?

- A. We expect to write to that location several times
- B. We expect to read from that location several times
- C. A & B
- D. Neither A nor B

The 3 C's of cache misses

- Compulsory (AKA Cold Start)
- Capacity
- Conflict

What is the cache miss rate we'll observe if we have an infinite cache?

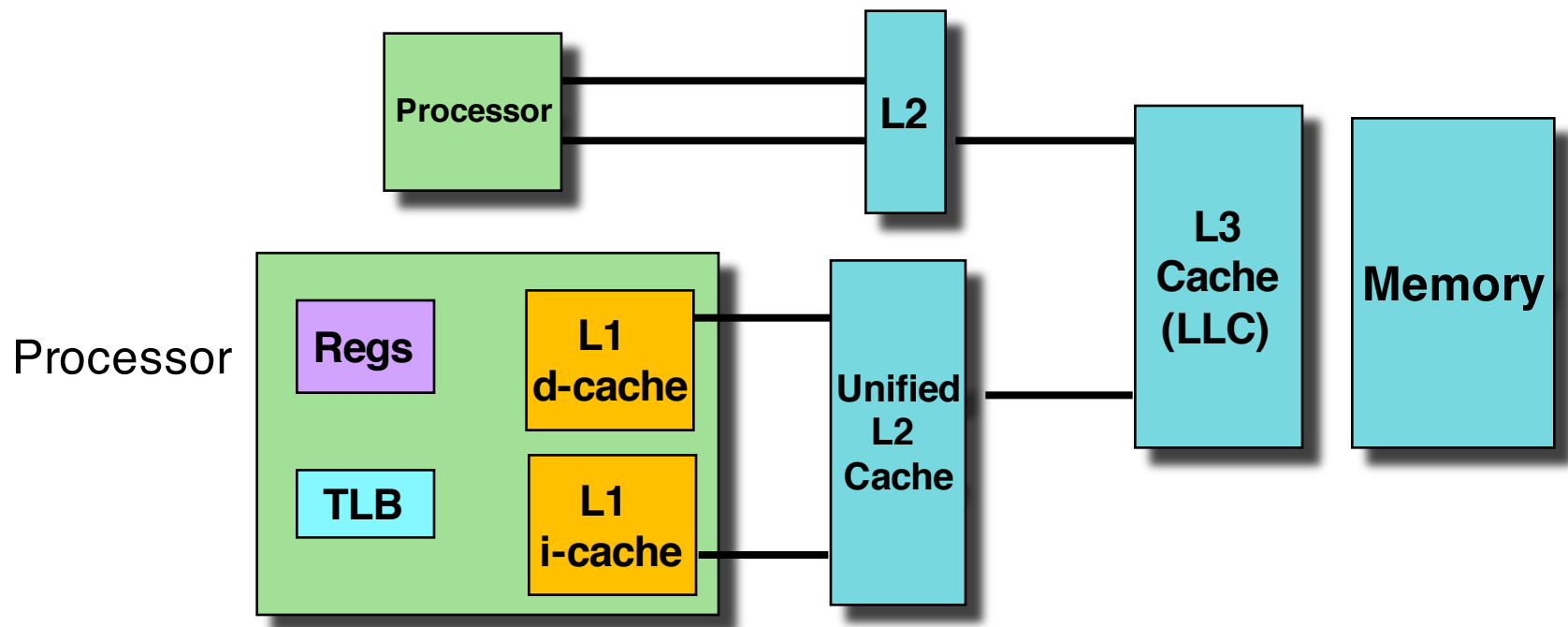
A. Compulsory

B. Capacity

C. Conflict

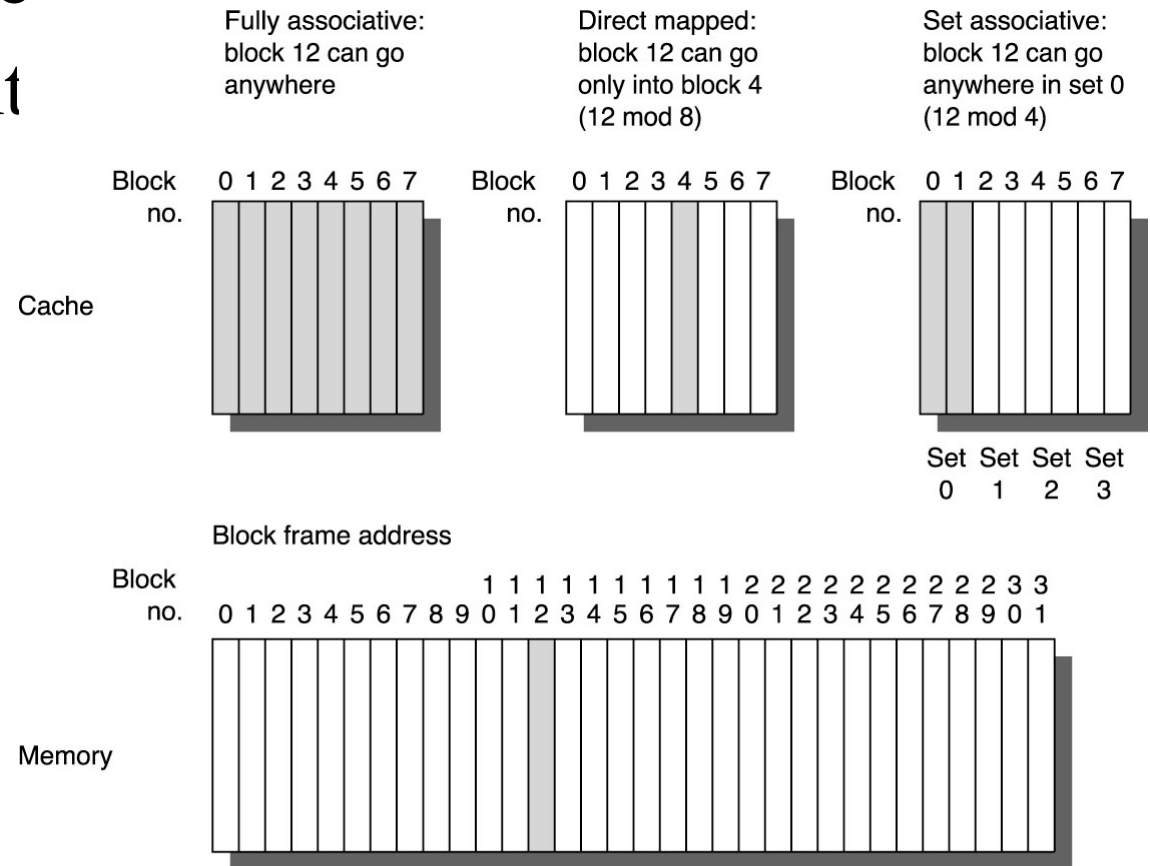
Other cache design issues

- Separate Instruction (I) and Data (D)
- Unified (I+D)
- Last Level Cache (LLC) – the slowest cache (furthest away from the processor)
- Translation Lookaside Buffer (TLB)



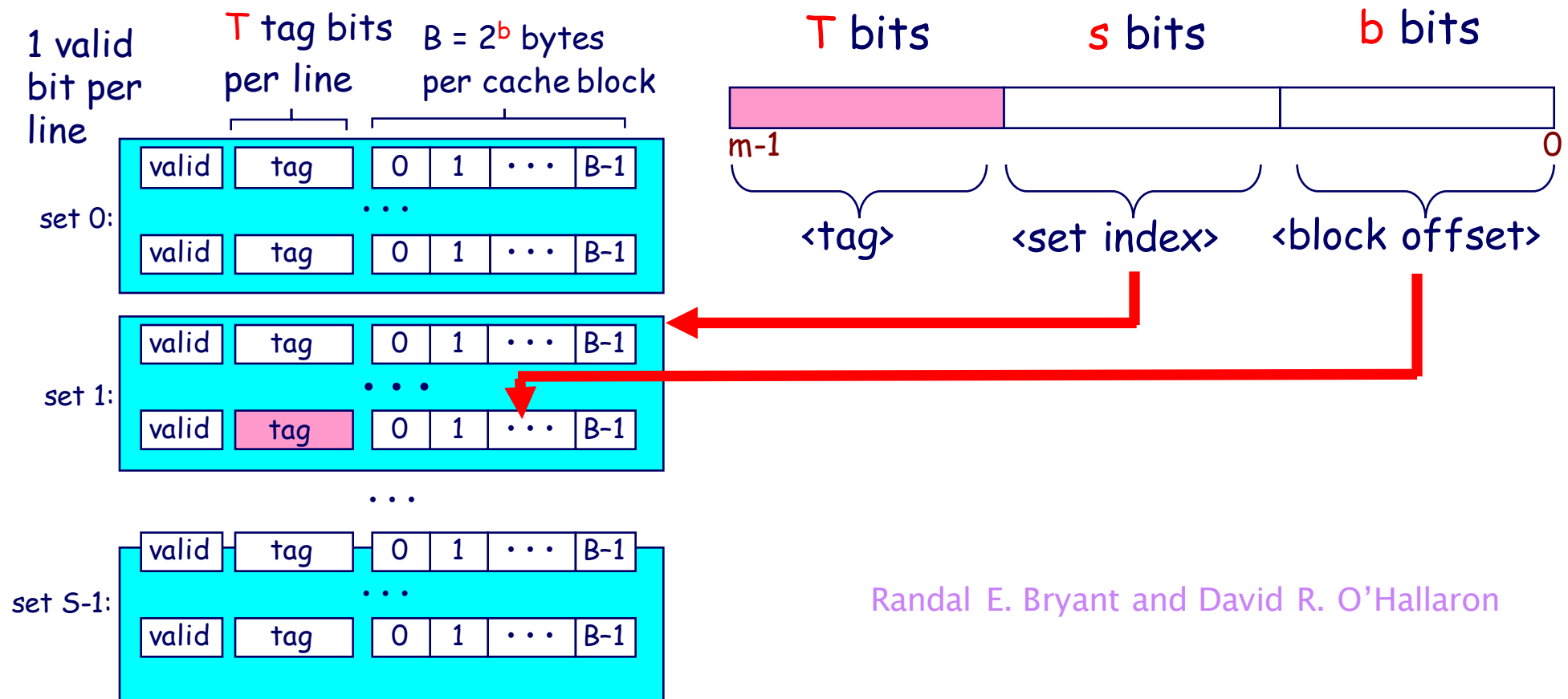
Two main types of caches

- Direct mapped
- Set Associative
 - Block placement
 - Eviction policy on a miss/write



Set associative cache

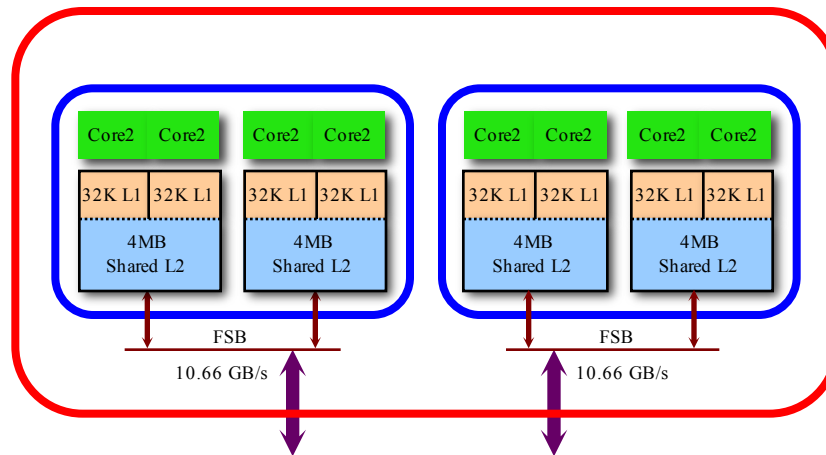
- More than 1 line per set
- Search each valid line for matching tag bits
- Miss/write: evict the least recently used (LRU)



Randal E. Bryant and David R. O'Hallaron

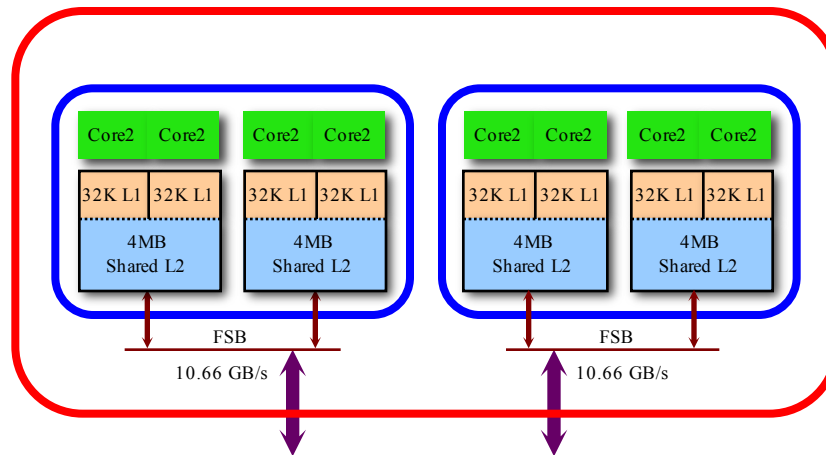
Examining Bang's Memory Hierarchy

- /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



How can we improve cache performance?

- A. Reduce the miss rate
- B. Reduce the miss penalty
- C. Reduce the hit time
- D. A, B and C

Optimizing for re-use

- The success of caching depends on the ability to *re-use* previously cached data
- Data access order affects re-use
- Assume a cache with 2 entries, each 2 words wide

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        a[i][j] += b[i][j];
```

```
for (j=0; j<N; j++)  
    for (i=0; i<N; i++)  
        a[i][j] += b[i][j];
```

0	1
4	5

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

The 3 C's
Compulsory
Capacity
Conflict

Testbed

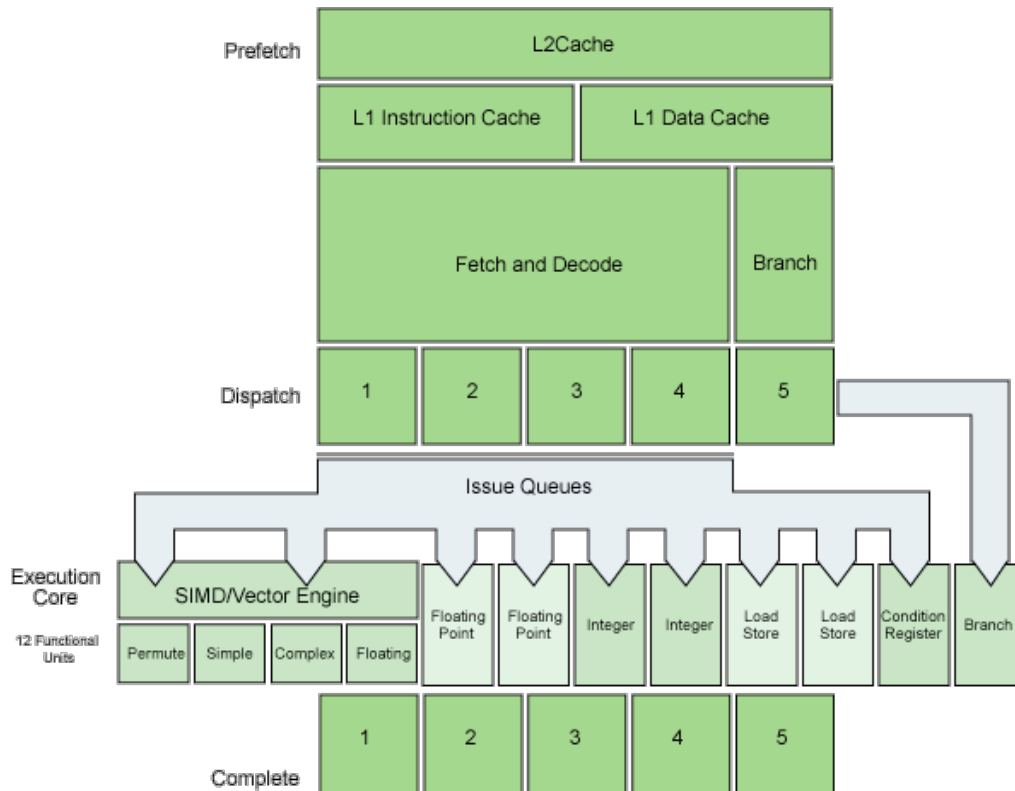
- 2.7GHz Power PC G5 (970fx)
- [http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/DC3D43B729FDAD2C00257419006FB955/\\$file/970FX_user_manual.v1.7.2008MAR14_pub.pdf](http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/DC3D43B729FDAD2C00257419006FB955/$file/970FX_user_manual.v1.7.2008MAR14_pub.pdf)
- Caches: 128 Byte line size
 - 512KB L2 (8-way, 12 CP hit time)
 - 32K L1 (2-way, 2 CP hit time)
- TLB: 1024 entries, 4-way
- gcc version 4.0.1
(Apple Computer, Inc. build 5370), -O2 optimization
- Single precision floating point

The results

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i][j] += b[i][j];
```

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++)
    a[i][j] += b[i][j];
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15



N	IJ (ms)	JI (ms)
64	0.007	0.007
128	0.027	0.083
512	1.1	37
1024	4.9	284
2048	18	2,090