Cache Lines

- Data flows among the CPU's caches and memory in fixed-length blocks called cache lines
- usually a power of 2, ranging from 16 to 256 bytes
- both the Xeon and Xeon Phi CPU Cores have a cache line size of 64 bytes

Cache miss

- on first access of a data item by a CPU it will be absent of this CPUs cache: warmup miss
- then the CPU is stalled and waits hundreds of cycles until it is fetched from main memory
- however: the item will be in the CPUs cache
- subsequent accesses to the same data item will be served faster (temporal locality) and the CPU will run at full speed

Cache Miss (2)

- after some time the cache will fill up
- once full, a new item will cause a capacity miss
- however a miss can occur way before the cache is completely full
- large caches are implemented as hardware hash tables with fixed size buckets (sets)
- the hash function is rather simple: look at the last significant n bits, where 2^n = set size

Intel Xeon Phi Cache

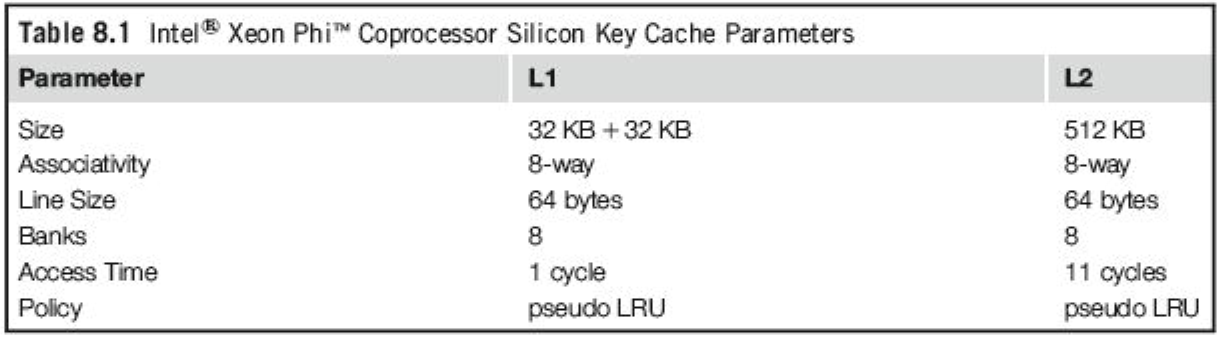
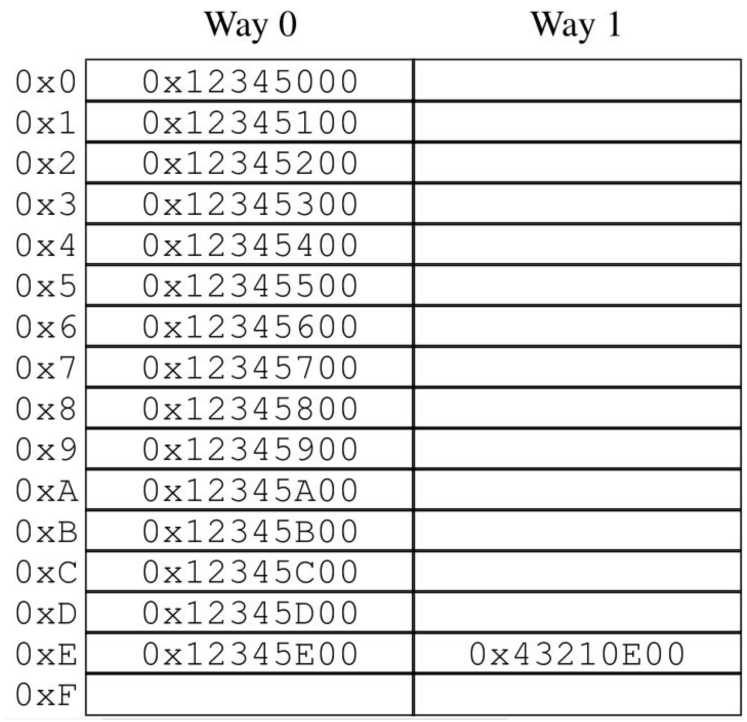


Table 8.1 Intel® Xeon Phi™ Coprocessor Silicon Key Cache Parameters

Parameter	**L1**	**L2**
Size	32 KB + 32 KB	512 KB
Associativity	8-way	8-way
Line Size	64 bytes	64 bytes
Banks	8	8
Access Time	1 cycle	11 cycles
Policy	pseudo LRU	pseudo LRU

Example

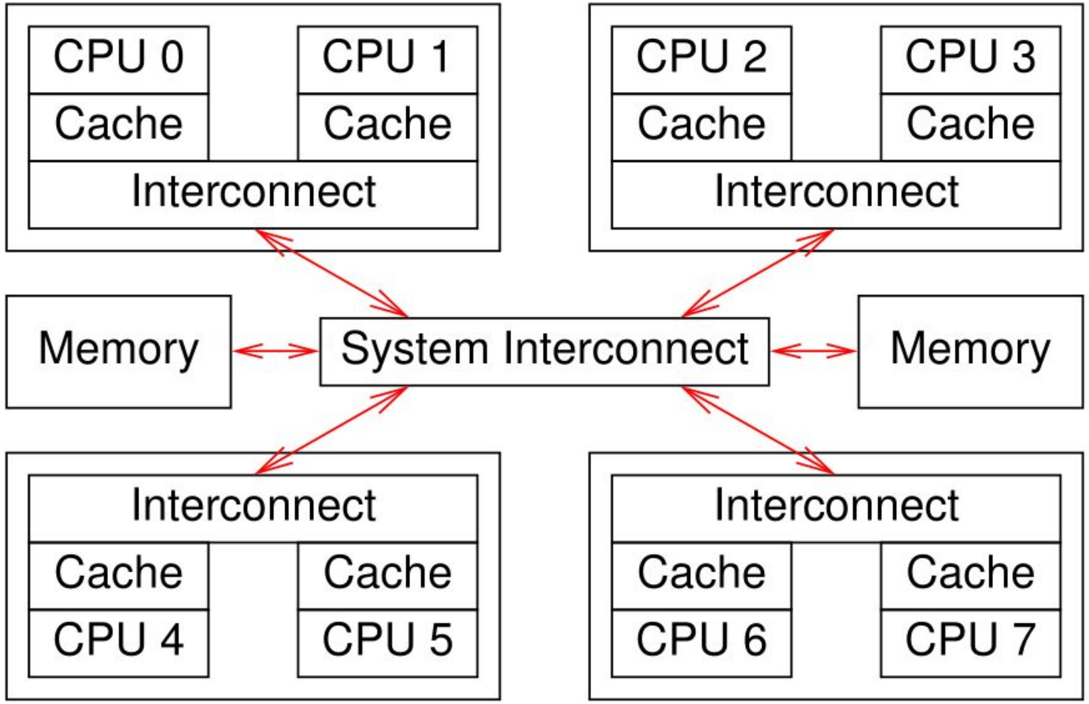


	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		

Writing to Cache

- All Cores need to agree on the value of a particular data item (cache coherence)
- When attempting to write, a core needs to be sure to be the only one containing the cache line of the data item
- to assure coherence, this cache line needs to be evicted from other core's caches
- if later another core wants to read this cache line a miss occurs: communication miss

Example: Writing a data item



Communication (simplified)

1. CPU 0 wants to write to a memory location **x**
2. it *asks* on the interconnect C0-C1 if the cache line containing x is present on any other core → no
3. request is forwarded to system interconnect → reports the cache line is present on interconnect C6-C7
4. request is forwarded to interconnect C6-C7 → C7 contains cache line containing x is its cache
5. C7 evicts cache line from its cache and transfers ownership to interconnect C6-C7
6. ownership is transferred all the way back to C0 cache → now C0 can write to memory location **x**

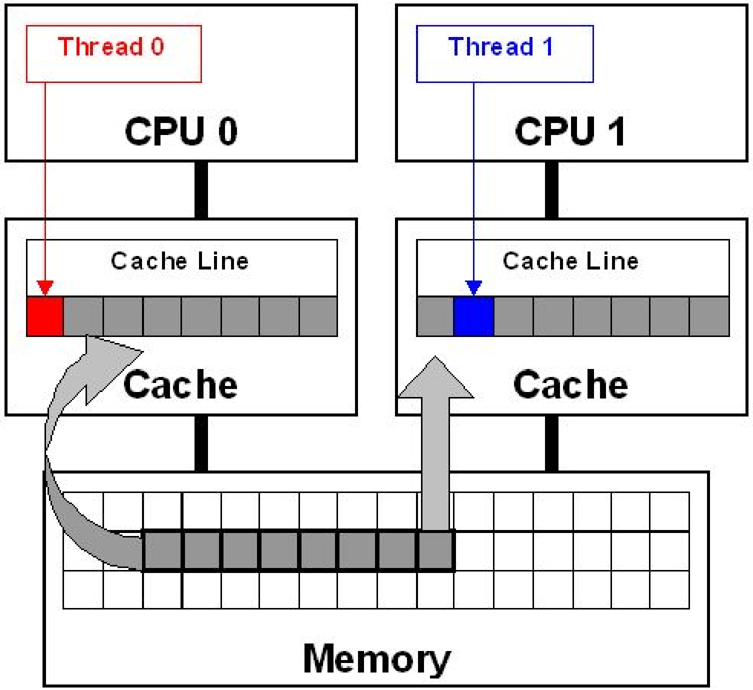
Cache Coherency

- assured by coherence protocols (most common:MESI)
- if multiple threads communicate over a single memory location, this can lead to a cache line bouncing between the cores' caches
- transferring a cache line is expensive (1 order of magnitude with respect to executing a single instruction) $\rightarrow$ destroys any chance of a scaling algorithm

False sharing

- an artifact of cache coherency, that hurts performance on multicore systems
- n threads on n cores work on n distinct memory locations → no logical sharing
- however, if these memory locations happen to be on the same cache line in memory, and at least 1 thread is writing to its location → cache line bouncing
- typically with (dynamic) arrays, but also distinct members within a class/struct → anything laid out contiguously in memory

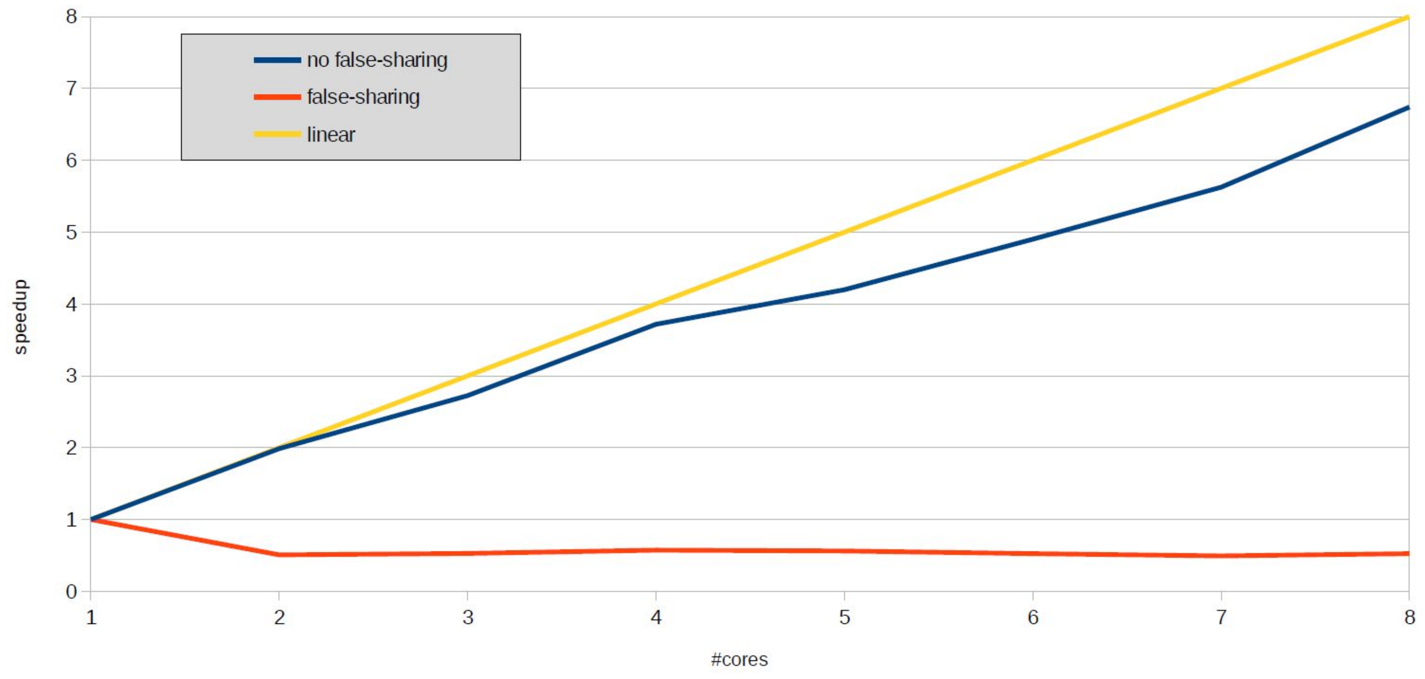
False-Sharing



Example

```
int odds = 0;
std::vector<int> v(N);
std::iota(v.begin(), v.end(), 0);
std::array<int, NUM_THREADS> t_odds;
std::fill(t_odds.begin(), t_odds.end(), 0);
#pragma omp parallel for
for (std::int64_t i = 0; i < N; ++i)
  if (1 == v[i] % 2) ++t_odds[CURRENT_THREAD_ID];
odds = std::accumulate(t_odds.begin(), t_odds.end(), 0);
```

Scaling



no false-sharing
false-sharing
linear
speedup
#cores
0
1
2
3
4
5
6
7
8

How to avoid False Sharing

- use thread-private local copies, modify locally, and write back when done
- use thread individual variables and align them to the cache line boundary
- use padding, i.e. extend the accessed memory location such that subsequently accessed memory is stored on a different cache line
- trade-off decision: too much padding trashes cash and reduces net-worth cache size

Summary

- access to main memory is expensive compared to today's multicore compute power
- multiple levels of CPU caches *can* help hiding that latency and reduce pressure on the memory bus (more net-worth bandwidth)
- software must be developed with temporal locality of reference in mind
- caches can even reduce performance in a multicore system due to cache coherence
- special care must be taken to avoid false sharing