CPU Cacheline False Sharing

- What is it?
- How it can impact performance.
- How to find it? (new tool)

Joe Mario
Oct 26, 2016
Senior Principal Engineer
Red Hat Performance Engineering

Where is the concern applicable?

1) An application with multiple threads accessing memory from different nodes*.

2) Multiple processes accessing shared memory from different nodes*.

* accesses from same node relevant but less painful.

Start with simple example

First, basic data structure

```
struct foo {
  int w;
  int x;
  int y;
  int z;
};
static struct foo f;
```

Add some code:

4 threads running in parallel on a 4 socket numa system

```
/* Thread 1 on node 0 */
        for (i = 0; i < 1000000; ++i)
          s += f.x;

/* Thread 2 on node 1 */
        for (i = 0; i < 1000000; ++i)
          ++f.y;

/* Thread 3 on node 2 */
        for (i = 0; i < 1000000; ++i)
          ++f.z;

/*Thread 4 on node 3 */
        for (i = 0; i < 1000000; ++i)
          ++f.zz;
```

Average time for each thread to execute its loop = **120 machine cycles**

Now modify the struct

1) Place each member in its own 64 byte aligned cacheline

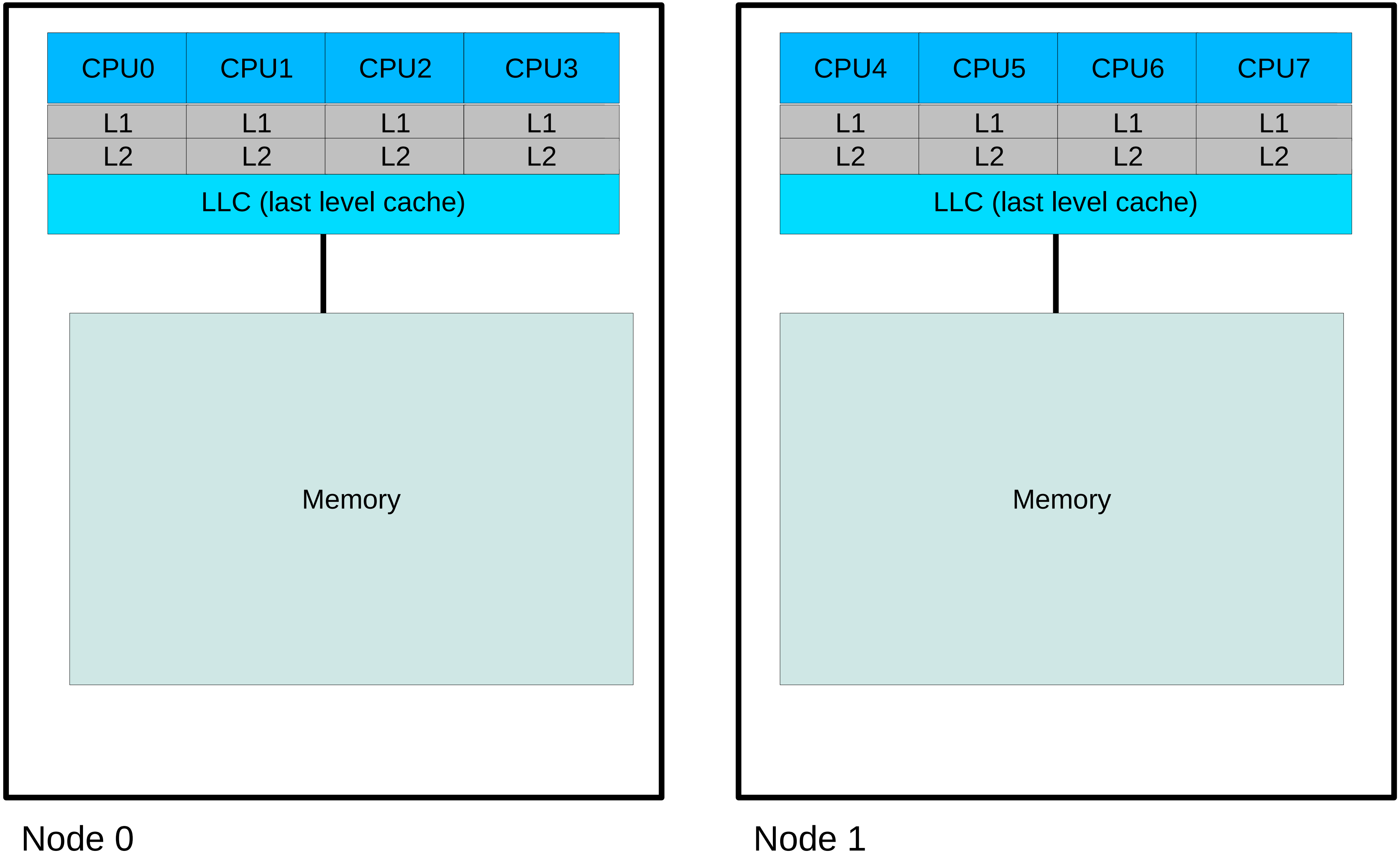
```
typedef int __attribute__((aligned (64))) aligned_int;

struct foo {
  aligned_int  w;
  aligned_int  x;
  aligned_int  y;
  aligned_int  z;
};
```

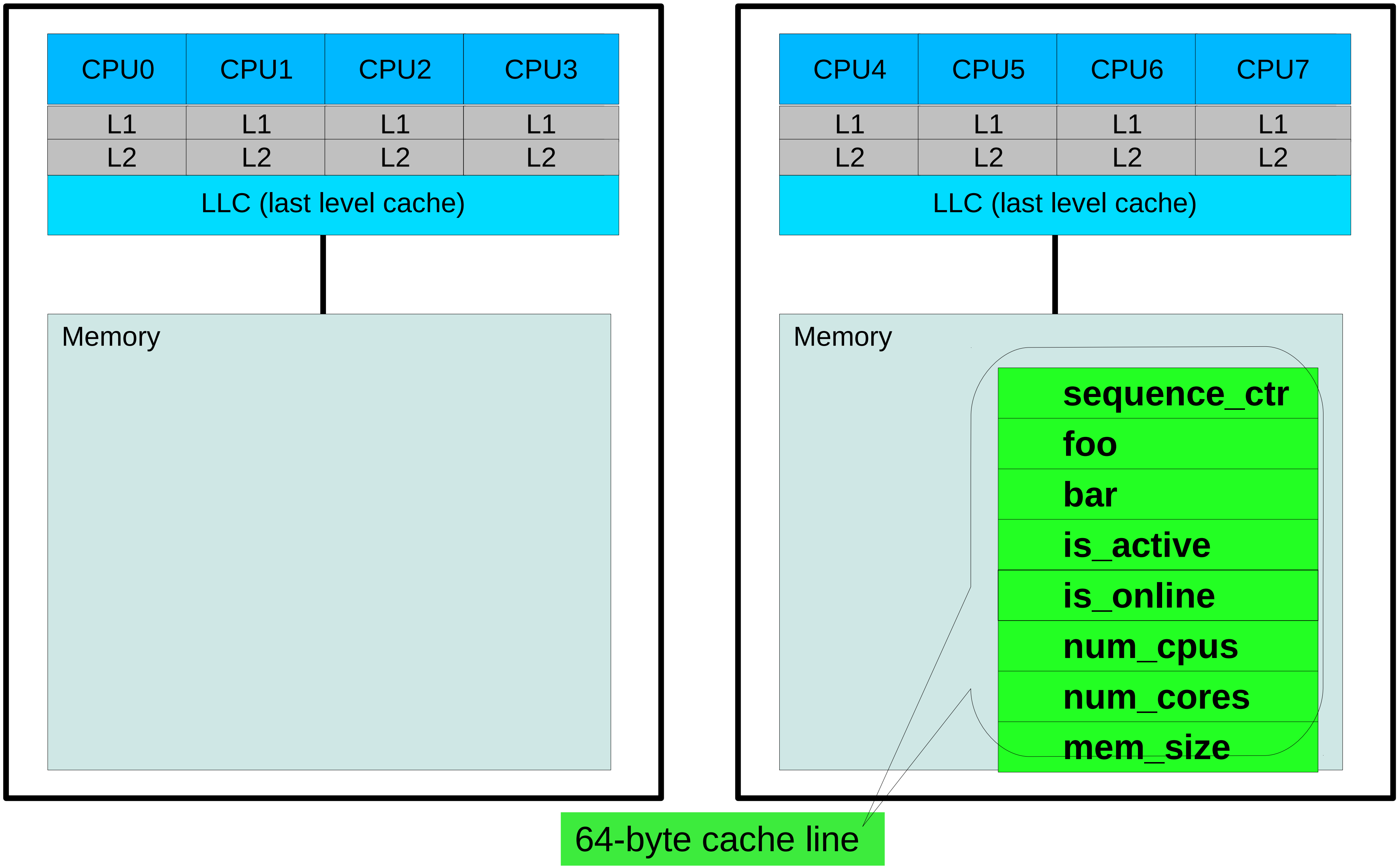
Average time for each thread to execute its loop drops by 2/3, from **120** machine cycles to **35 machine cycles**

Joe Mario

Basic false sharing – 2 socket system

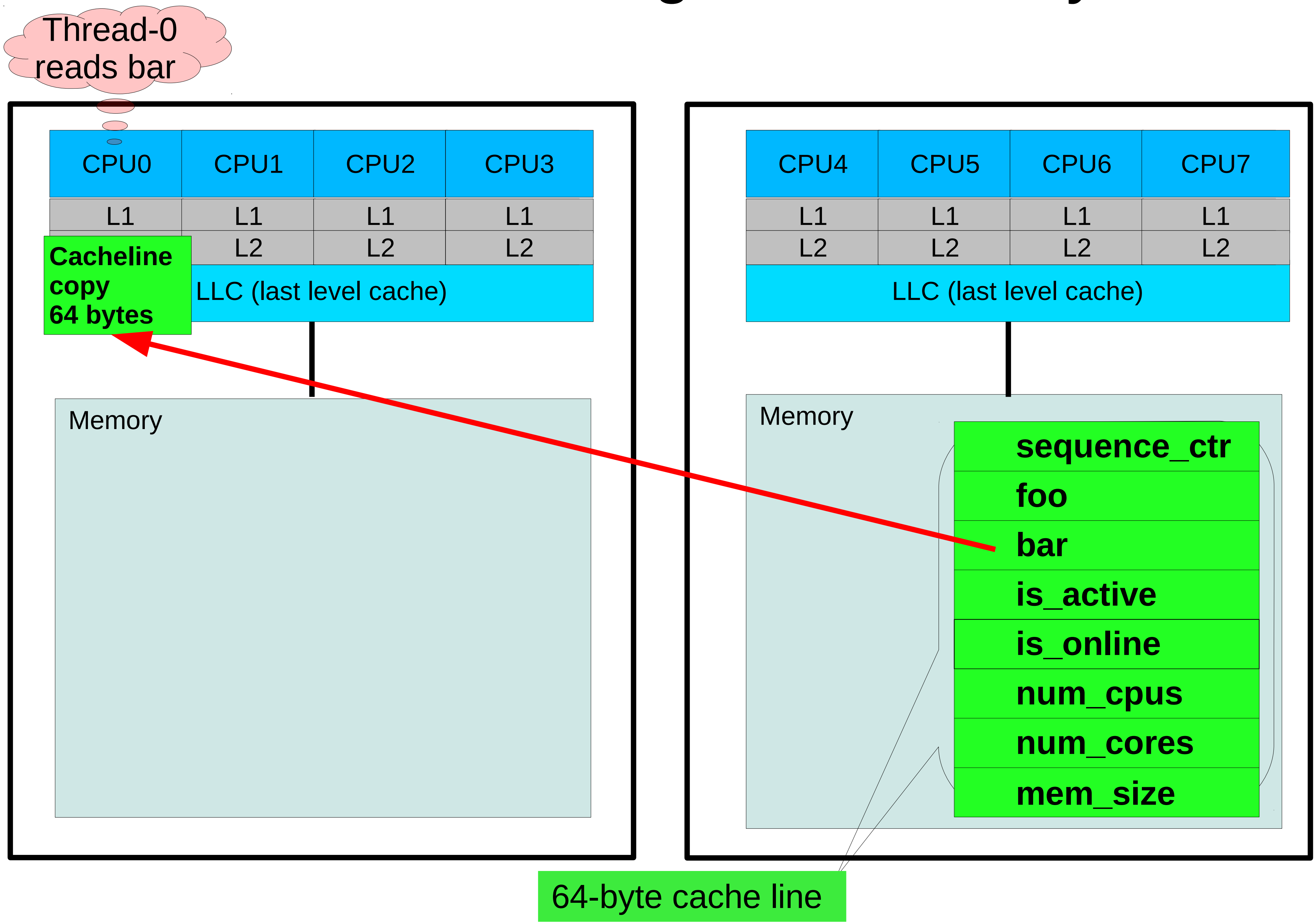


Basic false sharing – 2 socket system



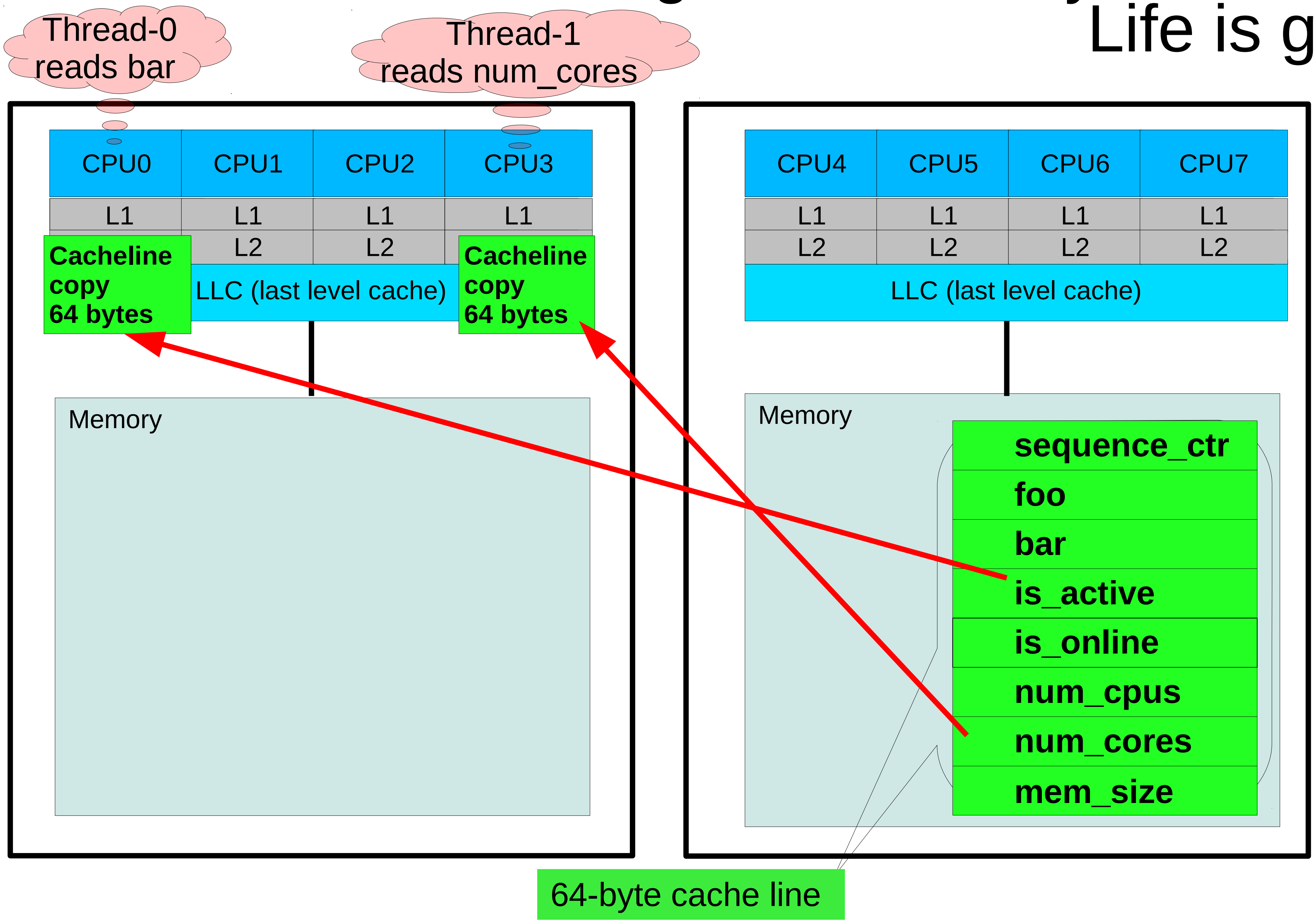
Joe Mario

Basic false sharing – 2 socket system



Basic false sharing – 2 socket system

Life is good



Joe Mario

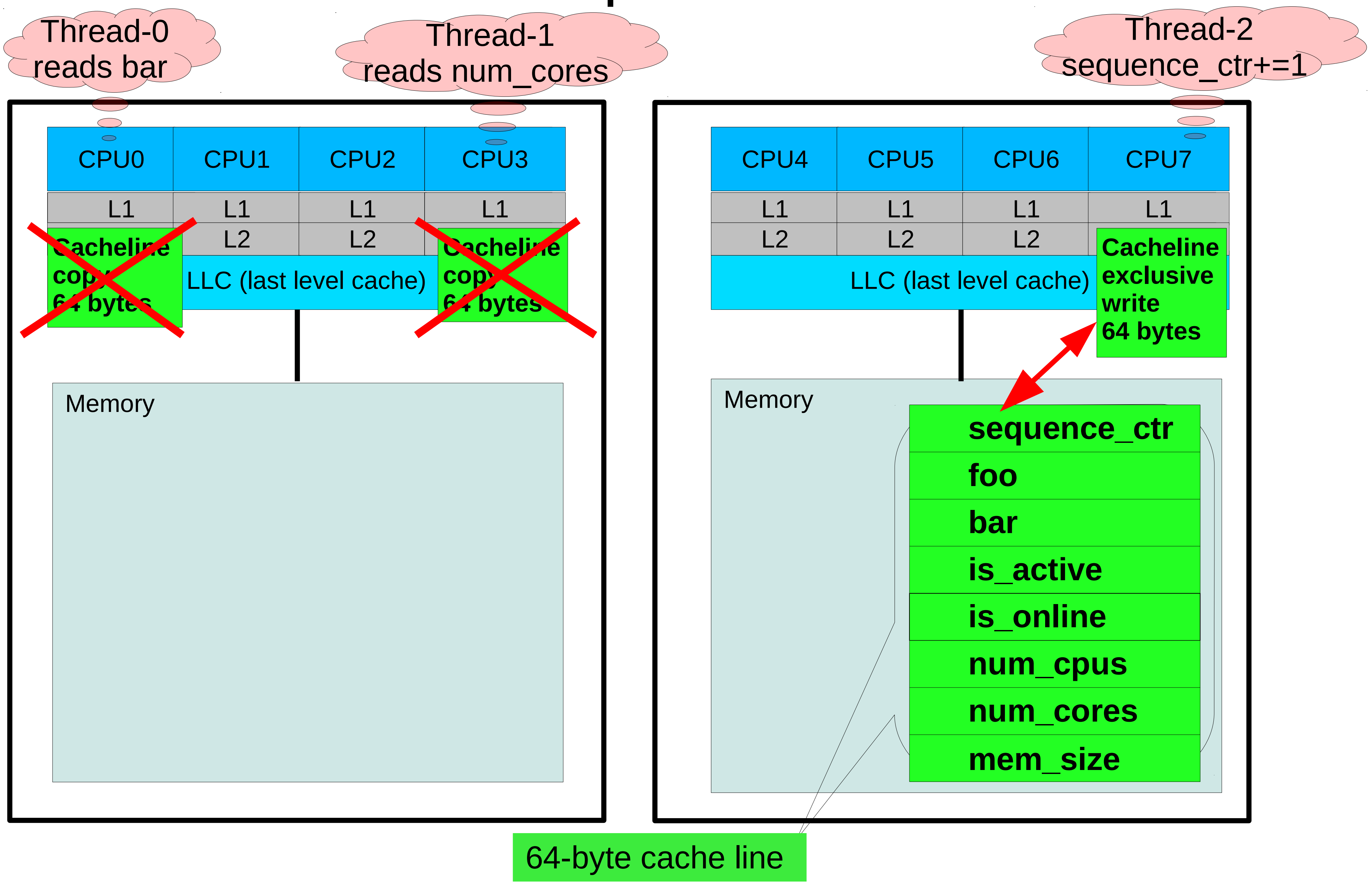
Where the trouble begins…

Now the program starts another thread which frequently modifies "*sequence_ctr*"

```
void random_func() {
   while (true) {
        do_work();
        do_more_work();
        sequence_ctr += 1;
   }
}
```


Joe Mario

The problem



Joe Mario

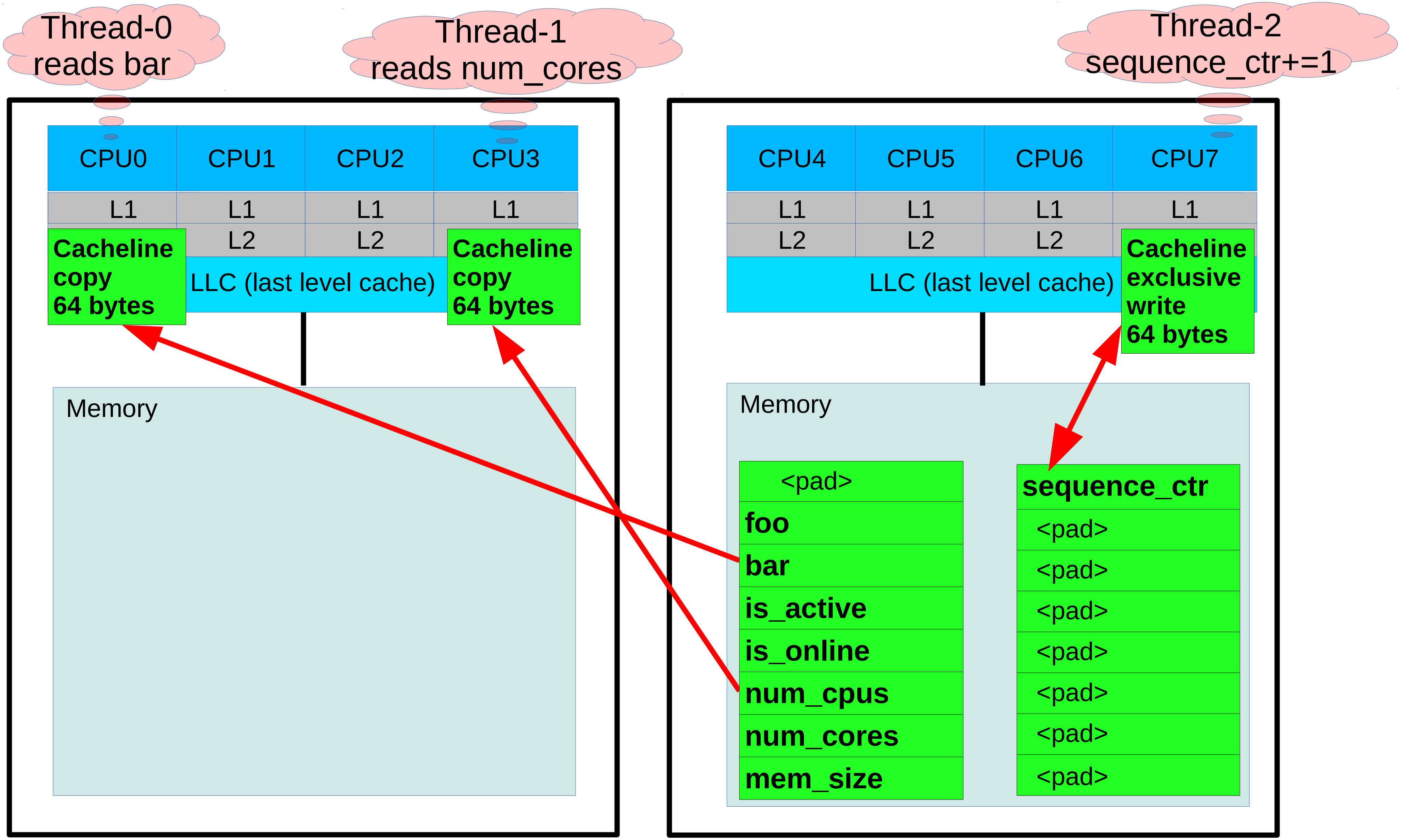
Looking a little closer:

- Read latencies for thread-0 and thread-1 read just got much longer.

- Every time sequence_ctr is modified:
  - threads 0 and 1 need to throw away their cacheline copies
  - Get back in line for an updated cacheline

- Wait for memory controller to synchronize
  - Ex: suppose sequence_ctr's value is 37 in memory and 38 in the modified cacheline

Ans: Put hot modified variable in own cacheline.



Conditions that aggravate false sharing

- Multiple threads writing to same cacheline.
- Multiple processes writing to same cacheline in shared memory.
- Remote accesses across numa nodes.
- Atomic memory operations. ex: _sync_fetch_and_add
    - atomic ops lock the cacheline
- Larger systems (8 and 16 numa nodes)
- On busy systems, (>= 4 sockets), false sharing load latencies peaking over 60,000 machine cycles are not uncommon

How to detect and find this?

New addition to the Linux perf tool:
perf c2c

"*c2c*" stands for "*cache to cache*"

Just got pulled into upstream

Look for it in a future RHEL 7.x (*use on Intel IVB or newer*)
Awesome feedback so far on it.

Prototype copy available at:
http://people.redhat.com/jmario/rhel7_c2c/perf.rhel7.c2c

Extensive usage info in blog at:
https://joemario.github.io/

Joe Mario

At a high level, “perf c2c” provides:

1) The cachelines virtual addr where false sharing was detected.
2) The readers and writers to those cachelines.
3) The offsets into the cachelines for those accesses.
4) The pid, tid, instruction addr, function name, filename.
5) The source file and line numbers.

At a high level, “perf c2c” provides (continued):

1) The average load latency for the loads.
2) The numa nodes and cpus involved.
3) Ability to see when hot variables are sharing a cacheline.
4) Ability to see unaligned hot data structs spilling into multiple cachelines.

Joe Mario

Steps to help minimize contention:

1) Pack read-only/read-mostly variables together.
2) Place the hottest written variables in their own cacheline.
3) Pad cachelines as a small tradeoff for reducing contention.
4) Align your data/buffers/structs on cacheline boundaries.
5) Lower the granularity of locks (lock smaller chunks of data to reduce contention).
6) Use compile-time asserts to guarantee struct member alignment:


Questions ?