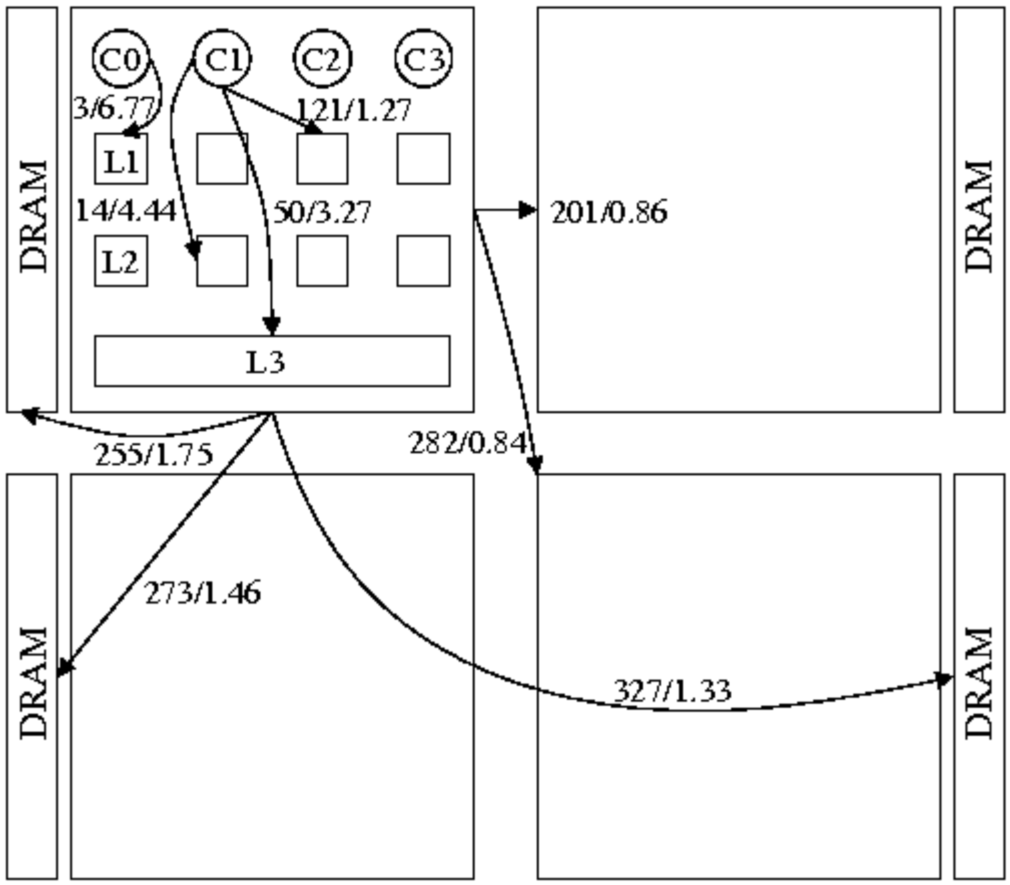


The AMD 16-core system topology. Memory access latency is in cycles and listed before the backslash. Memory bandwidth is in bytes per cycle and listed after the backslash. The measurements reflect the latency and bandwidth achieved by a core issuing load instructions. The measurements for accessing the L1 or L2 caches of a different core on the same chip are the same. The measurements for accessing any cache on a different chip are the same. Each cache line is 64 bytes, L1 caches are 64 Kbytes 8-way set associative, L2 caches are 512 Kbytes 16-way set associative, and L3 caches are 2 Mbytes 32-way set associative.

[Reprinted with permission from S. Boyd-Wickizer et al. Corey: An Operating System for Many Cores. Proceedings of Usenix Symposium on Operating Systems Design and Implementation (OSDI), December 2008.]

```
Handout for CS 372H
Class 18
21 March 2013

1. Recall implementation of acquire() and release() in spinlocks
context:

    [this item is fully review.]

    It uses an atomic instruction on the CPU. For example, on the
    x86, doing
            "xchg addr, %eax"
    does the following:

    (i)   freeze all CPUs' memory activity for address addr
    (ii)  temp = *addr
    (iii) *addr = %eax
    (iv)  %eax = temp
    (v)   un-freeze memory activity

    /* pseudocode */
    int xchg_val(addr, value) {
        %eax = value;
        xchg (*addr), %eax
    }

    struct Lock {
      int locked;
    }

    /* bare-bones version of acquire */
    void acquire (Lock *lock) {
      pushcli();    /* what does this do? */
      while (1) {
        if (xchg_val(&lock->locked, 1) == 0)
          break;
      }
    }

    /* optimization in acquire; call xchg_val() less frequently */
    void acquire(Lock* lock) {
        pushcli();
        while (xchg_val(&lock->locked, 1) == 1) {
            while (lock->locked) ;
        }
    }

    void release(Lock *lock){
       xchg_val(&lock->locked, 0);
       popcli();    /* what does this do? */
    }

    The above is called a *spinlock* because acquire() spins.

```

```
2. Here's an alternative.....

    Instead of using the XCHG instruction, it uses CMPXCHG.

A. CAS / CMPXCHG

    Useful operation: compare-and-swap, known as CAS. Says: "atomically
    check whether a given memory cell contains a given value, and if it
    does, then replace the contents of the memory cell with this other
    value; in either case, return the original value in the memory
    location".

    On the X86, we implement CAS with the CMPXCHG instruction, but note
    that this instruction is not atomic by default, so we need the LOCK
    prefix.

    Here's pseudocode:

        int cmpxchg_val(int* addr, int oldval, int newval) {
            LOCK: // remember, this is pseudocode
            int was = *addr;
            if (*addr == oldval)
                *addr = newval;
            return was;
        }

    Here's inline assembly:

        uint32_t cmpxchg_val(uint32_t* addr, uint32_t oldval, uint32_t newval) {
            uint32_t was;
            asm volatile("lock cmpxchg %3, %0"
                            : "+m" (*addr), "=a" (was)
                            : "a" (oldval), "r" (newval)
                            : "cc");
            return was;
        }

B. MCS locks

    Citation: Mellor-Crummey, J. M. and M. L. Scott.  Algorithms for
    Scalable Synchronization on Shared-Memory Multiprocessors, ACM
    Transactions on Computer Systems, Vol. 9, No.  1, February, 1991,
    pp.21-65.

    Each CPU has a qnode structure in *local* memory. Here, local can
    mean local memory in NUMA machine or its own cache line that other
    CPUs are not allowed to cache (i.e., the cache line is in exclusive
    mode):

    typedef struct qnode {
        struct qnode* next;
        bool someoneelse_locked;
    } qnode;

    typedef qnode* lock;  // a lock is a pointer to a qnode

    --The lock itself is literally the *tail* of the list of CPUs holding
    or waiting for the lock.

    --While waiting, a CPU spins on its local "locked" flag. Here's the
    code for acquire:

        // lockp is a qnode**. I points to our local qnode.
        void acquire(lock* lockp, qnode* I) {

            I->next = NULL;
            qnode* predecessor;

            // next line makes lockp point to I (that is, it sets *lockp <-- I)
            // and returns the old value of *lockp. Uses atomic operation
            // XCHG. see earlier in handout (or earlier handouts)
            // for implementation of xchg_val.

            predecessor = xchg_val(lockp, I);    // "A"
            if (predecessor != NULL) { // queue was non-empty
                I->someoneelse_locked = true;

                predecessor->next = I;           // "B"
                while (I->someoneelse_locked) ;    // spin
            }
            // we hold the lock!
        }
```

What’s going on?

--If the lock is unlocked, then *lockp == NULL.

--If the lock is locked, and there are no waiters, then *lockp points to the qnode of the owner

--If the lock is locked, and there are waiters, then *lockp points to the qnode at the tail of the waiter list.

--Here’s the code for release:

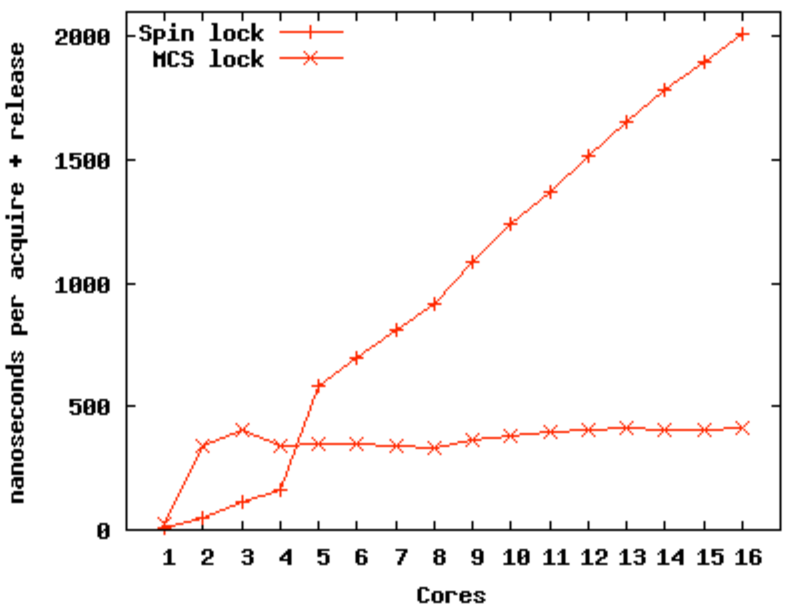
```
        void release(lock* lockp, qnode* I) {
            if (!I->next)   { // no known successor
                if (cmpxchg_val(lockp, I, NULL) == I) {     // "C"
                    // swap successful: lockp was pointing to I, so now
                    // *lockp == NULL, and the lock is unlocked. we can
                    // go home now.
                    return;
                }
                // if we get here, then there was a timing issue: we had
                // no known successor when we first checked, but now we
                // have a successor: some CPU executed the line "A"
                // above. Wait for that CPU to execute line "B" above.
                while (!I->next) ;
            }

            // handing the lock off to the next waiter is as simple as
            // just setting that waiter’s "someoneelse_locked" flag to false
            I->next->someoneelse_locked = false;
        }
```

What’s going on?

--If I->next == NULL and *lockp == I, then no one else is waiting for the lock. So we set *lockp == NULL.

--If I->next == NULL and *lockp != I, then another CPU is in acquire (specifically, it executed its atomic operation, namely line "A", before we executed ours, namely line "C"). So wait for the other CPU to put the list in a sane state, and then drop down to the next case:

--If I->next != NULL, then we know that there is a spinning waiter (the oldest one). Hand it the lock by setting its flag to false.



Time required to acquire and release a lock on a 16-core AMD machine when varying number of cores contend for the lock. The two lines show Linux kernel spin locks and MCS locks (on Corey). A spin lock with one core takes about 11 nanoseconds; an MCS lock about 26 nanoseconds.

[Reprinted with permission from S. Boyd-Wickizer et al. Corey: An Operating System for Many Cores. Proceedings of Symposium on Operating Systems Design and Implementation (OSDI), December 2008.]