CPU performance increases are slowing

Amdahl’s Law

Given:
- Parallelisable fraction \( P \)
- Number of processors \( N \)
- Speed up \( S \)

\[
S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \\
S(\infty) = \frac{1}{1 - P}
\]

Parallel computing takeaway:
- Useful for small numbers of CPUs \( (N) \)
- Or, high values of \( P \)
  - Aim for high \( P \) values by design

Multiprocessor System

A single CPU can only go so fast
- Use more than one CPU to improve performance
- Assumes
  - Workload can be parallelised
  - Workload is not I/O-bound or memory-bound

Types of Multiprocessors (MPs)

Classic symmetric multiprocessor (SMP)
- Uniform Memory Access
  - Access to all memory occurs at the same speed for all processors.
- Processors with local caches
  - Separate cache hierarchy
    - Cache coherency issues
**Cache Coherency**

What happens if one CPU writes to address 0x1234 (and it is stored in its cache) and another CPU reads from the same address (and gets what is in its cache)?

- Can be thought of as managing replication and migration of data between CPUs

**Memory Model**

A read produces the result of the last write to a particular memory location?

- Approaches that avoid the issue in software also avoid exploiting replication for cooperative parallelism
  - E.g., no mutable shared data.
- For classic SMP a hardware solution is used
  - Write-through caches
  - Each CPU snoops bus activity to invalidate stale lines
  - Reduces cache effectiveness – all writes go out to the bus.

- Longer write latency
- Reduced bandwidth

**Types of Multiprocessors (MPs)**

NUMA MP

- Non-uniform memory access
  - Access to some parts of memory is faster for some processors than other parts of memory
- Provides high-local bandwidth and reduces bus contention
  - Assuming locality of access

**Cache Coherence**

Snooping caches assume

- write-through caches
- cheap "broadcast" to all CPUs

Many alternative cache coherency models

- They improve performance by tackling above assumptions
- They will examine MESI (four state)
- ‘Memory bus’ becomes message passing system between caches

**Example Coherence Protocol MESI**

Each cache line is in one of four states

**Invalid (I)**

- This state indicates that the addressed line is not resident in the cache and/or any data contained is considered not useful.

**Exclusive (E)**

- The addressed line is in this cache only.
- The data in this line is consistent with system memory.

**Shared (S)**

- The addressed line is valid in the cache and in at least one other cache.
- A shared line is always consistent with system memory. That is, a shared state is shared-unmodified; there is no shared-modified state.

**Modified (M)**

- The line is valid in the cache and in only this cache.
- The line is modified with respect to system memory—that is, the modified data in the line has not been written back to memory.
Example

Directory-based coherence

Each memory block has a home node

Home node keeps directory of caches that have a copy

- E.g., a bitmap of processes per cache line

Pro

- Invalidation/update messages can be directed explicitly
  - No longer rely on broadcast/snooping

Con

- Requires more storage to keep directory
  - E.g. each 256 bits of memory (cache line) requires 32 bits (processor mask) of directory

Chip Multiprocessor (CMP)

- per-core L1 caches
- shared lower on-chip caches
- usually called “multicore”
- “reduced” cache coherency issues
  - Between L1, L2 shared.

MESI (with snooping/broadcast)

Events:

- RH = Read Hit
- RWS = Read miss, shared
- RW = Read miss, exclusive
- WHit = Write hit
- WM = Write miss
- SHR = Snoop hit on read
- SIR = Snoop hit on invalidate
- LRU = LRU replacement

Bus Transactions:

- Push = Write cache line back to memory
- Invalidate = Broadcast invalidate
- Read = Read cache line from memory

Performance improvement via write-back caching

- Less bus traffic

Pro

- Invalidation/update messages can be directed explicitly
  - No longer rely on broadcast/snooping

Con

- Requires more storage to keep directory
  - E.g. each 256 bits of memory (cache line) requires 32 bits (processor mask) of directory

Chip Multiprocessor (CMP)

- per-core L1 caches
- shared lower on-chip caches
- usually called “multicore”
- “reduced” cache coherency issues
  - Between L1, L2 shared.

ARM MCore: Cache-to-Cache Transfers

- Cache lines can migrate between L1 caches belonging to different cores without involving the L2
- Clean lines – DDI (Direct Data Intervention)
- Dirty Lines – ML (Migratory Lines)
Cache to Cache Latency

- Significant benefits achievable if the working set of the application partitioned between the cores can be contained within the sum of their caches
- Helpful for streaming data between cores
  - may be used in conjunction with interrupts between cores

Though dirty lines have higher latency they still have ≈ 50% performance benefit

Simultaneous multithreading (SMT)


- Replicated functional units, register state
- Interleaved execution of several threads
- Opposite to extracting limited parallelism from instruction streams
- Tightly shared cache hierarchy
- No cache coherency issues
  (called hyperthreading on x86)

Summary

Hardware-based cache coherency:
- Provide a consistent view of memory across the machine.
- Read will get the result of the last write to the memory hierarchy

Memory Ordering

Example: a tail of a critical section
```c
/* assuming lock already held */
/* counter++ */
load r1, counter
add r1, r1, 1
store r1, counter
/* unlock(mutex) */
store zero, mutex
```
Relies on all CPUs seeing update of counter before update of mutex
Depends on assumptions about ordering of stores to memory

Memory Models: Strong Ordering

Sequential consistency
- the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program

Traditionally used by many architectures
Assume X = Y = 0 initially

Potential interleavings

At least one CPU must load the other’s new value
- Forbidden result: X=0,Y=0

```
store 1, X
load r2, Y
store 1, Y
load r2, X
X=1, Y=0
```

```
store 1, X
load r2, Y
store 1, Y
load r2, X
X=0, Y=1
```

```
store 1, X
load r2, Y
store 1, Y
load r2, X
X=1, Y=1
```

```
store 1, X
load r2, Y
store 1, Y
load r2, X
X=0, Y=1
```

```
store 1, X
load r2, Y
store 1, Y
load r2, X
X=1, Y=1
```
Realistic Memory Models

Modern hardware features can interfere with store order:
• write buffer (or store buffer or write-behind buffer)
• instruction reordering (out-of-order execution)
• superscalar execution and pipelining

Each CPU/core keeps its own data consistent, but how is it viewed by others?

Write-buffers and SMP

Stores go to write buffer to hide memory latency
• And cache invalidates

Loads read from write buffer if possible

CPU 0
store r1, A
store r2, B
store r3, C
load r4, A

CPU 0
store r1, A
store r2, B
store r3, C

CPU 1
store r1, A
store r2, B
store r3, C

Write-buffers and SMP

When the buffer eventually drains, what order does CPU1 see CPU0’s memory updates?

What happens in our example?

CPU 0
store 1, A
store 2, B
store 3, C

CPU 0
store 1, A
store 2, B
store 3, C

CPU 1 sees
A=1
B=2
C=3

CPU 1 sees
count updated
mutex = 0

/* counter++ */
load r1, count
add r1, r1, 1
store r1, count
/* unlock(mutex) */
store zero, mutex

Total Store Ordering (e.g. x86)

Stores are guaranteed to occur in FIFO order

CPU 0
store 1, A
store 2, B
store 3, C

CPU 1 sees
A=1
B=2
C=3

CPU 1 sees
count updated
mutex = 0

/* counter++ */
load r1, count
add r1, r1, 1
store r1, count
/* unlock(mutex) */
store zero, mutex
**Total Store Ordering (e.g. x86)**

Assume X = Y = 0 initially

CPU 0
store 1, X
load r2, Y

CPU 1
store 1, Y
load r2, X

What is the problem here?

Stores are buffered in write-buffer and don't appear on other CPU in time. Can get X=0, Y=0!!!

Loads can "appear" re-ordered with preceding stores

**Memory “fences” or “barriers”**

The provide a "fence" between instructions to prevent apparent re-ordering

Effectively, they drain the local CPU’s write-buffer before proceeding.

CPU 0
store 1, X
fence
load r2, Y

CPU 1
store 1, Y
fence
load r2, X

**Partial Store Ordering (e.g. ARM MPcore)**

All stores go to write buffer
Loads read from write buffer if possible

Redundant stores are cancelled or merged

CPU 0
store BUSY, addr1
store VAL, addr2
store IDLE, addr3

CPU 1 sees
addr2 = VAL
addr1 = IDLE

- Stores can appear to overtake (be re-ordered) other stores
- Need to use memory barrier

**Total Store Ordering (e.g. ARM MPcore)**

The barriers prevent preceding stores appearing after successive stores

- Note: Reality is a little more complex (read barriers, write barriers), but principle similar
  - load r1, counter
  - add r1, r1, 1
  - store r1, counter barrier
  - store zero, mutex

- Store to counter can overtake store to mutex
  - i.e. update move outside the lock
- Need to use memory barrier
- Failure to do so will introduce subtle bugs:
  - Critical section “leaking” outside the lock
MP Hardware Take Away

Each core/cpu sees sequential execution of own code

Other cores see execution affected by
- Store order and write buffers
- Cache coherence model
- Out-of-order execution

Systems software needs understand:
- Specific system (cache, coherence, etc.)
- Synch mechanisms (barriers, test_n_set, load_linked — store_cond).

...to build cooperative, correct, and scalable parallel code

Memory ordering for various Architectures

Concurrency Observations

Locking primitives require exclusive access to the "lock"

- Care required to avoid excessive bus/interconnect traffic

Kernel Locking

Several CPUs can be executing kernel code concurrently.

Need mutual exclusion on shared kernel data.

Issues:
- Lock implementation
- Granularity of locking (i.e. parallelism)

Mutual Exclusion Techniques

Disabling interrupts (CLI — STI).
- Unsuitable for multiprocessor systems.

Spin locks.
- Busy-waiting wastes cycles.

Lock objects (locks, semaphores).
- Flag (or a particular state) indicates object is locked.
- Manipulating lock requires mutual exclusion.
Hardware Provided Locking Primitives

```c
int test_and_set(lock *);
int compare_and_swap(int c, int v, lock *);
int exchange(int v, lock *)
int atomic_inc(lock *)
```

`v = load_linked(lock *) / bool
store_conditional(int, lock *)`

- LL/SC can be used to implement all of the above

Spin Lock Busy-waits Until Lock Is Released

Stupid on uniprocessors, as nothing will change while spinning.
- Should release (block) thread on CPU immediately.
- Maybe ok on SMPs: locker may execute on other CPU.
  - Minimal overhead (if contention low).
  - Still, should only spin for short time.
Generally restrict spin locking to:
- short critical sections,
- unlikely to (or preferably can’t) be contend by thread on same CPU.
- local contention can be prevented
  - by design
  - by turning off interrupts

Spinning versus Switching

The general approaches taken are
- Spin forever
- Spin for some period of time, if the lock is not acquired, block and switch
  - The spin time can be
    - Fixed (related to the switch overhead)
    - Dynamic
    - Based on previous observations of the lock acquisition time

Spin locks

```c
void lock (volatile lock_t *l) {
  while (test_and_set(l)) ;
}
void unlock (volatile lock_t *l) {
  *l = 0;
}
```

Busy waits. Good idea?

Interrupt Disabling

Assume no local contention by design, is disabling interrupt important?

Hint: What happens if a lock holder is preempted (e.g., at end of its timeslice)?

All other processors spin until the lock holder is re-scheduled
Alternative to spinning: Conditional Lock (TryLock)

```c
bool cond_lock (volatile lock t *l) {
    if (test_and_set(l))
        return FALSE; // couldn’t lock
    else
        return TRUE; // acquired lock
}
```

Can do useful work if fail to acquire lock.

**But** may not have much else to do.

Livelock: May never get lock!

Another alternative to spinning.

```c
void mutex lock (volatile lock t *l) {
    while (1) {
        for (int i=0; i<MUTEX N; i++)
            if (!test_and_set(l))
                return;
        yield();
    }
}
```

Spins for limited time only
- assumes enough for other CPU to exit critical section

Useful if critical section is shorter than N iterations.

Starvation possible.

Common Multiprocessor Spin Lock

```c
void mp_spinlock (volatile lock t *l) {
    cli(); // prevent preemption
    while (test_and_set(l)); // lock
}

void mp_unlock (volatile lock t *l) {
    *l = 0;
    sti();
}
```

Only good for short critical sections
Does not scale for large number of processors
Relies on bus-arbitrator for fairness
Not appropriate for user-level
Used in practice in small SMP systems

Need a more systematic analysis


Compares Simple Spinlocks

**Test and Set**

```c
void lock (volatile lock_t *l) {
    while (test_and_set(l));
}
```

**Test and Test and Set**

```c
void lock (volatile lock_t *l) {
    while (*l == BUSY || test_and_set(l));
}
```

does not scale for large number of processors
Relies on bus-arbitrator for fairness
Not appropriate for user-level
Used in practice in small SMP systems

**test_and_test_and_set LOCK**

Avoid bus traffic contention caused by test_and_set until it is likely to succeed
Normal read spins in cache
Can starve in pathological cases
Benchmark

for i = 1 .. 1,000,000 {
    lock()
    crit_section()
    unlock()
    compute()
}

Compute chosen from uniform random distribution of mean 5 times critical section
Measure elapsed time on Sequent Symmetry (20 CPU 30386, coherent write-back invalidate caches)

Results

Test and set performs poorly once there is enough CPUs to cause contention for lock
• Expected

Test and Test and Set performs better
• Performance less than expected
  • Still significant contention on lock when CPUs notice release and all attempt acquisition

Critical section performance degenerates
• Critical section requires bus traffic to modify shared structure
  • Lock holder competes with CPU that missed as they test and set
    – lock holder is slower
  • Slower lock holder results in more contention

Idea

Can inserting delays reduce bus traffic and improve performance
Explore 2 dimensions
• Location of delay
  – Insert a delay after release prior to attempting acquire
  – Insert a delay after each memory reference
• Delay is static or dynamic
  – Static – assign delay “slots” to processors
    – Issue delay tuned for expected contention level
  – Dynamic – use a back-off scheme to estimate contention
    – Similar to ethernet
    – Degrades to static case in worst case.

Examining Inserting Delays

<table>
<thead>
<tr>
<th>Location</th>
<th>Delay (usec)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>6.5</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>6.0</td>
</tr>
</tbody>
</table>

Queue Based Locking

Each processor inserts itself into a waiting queue
• It waits for the lock to free by spinning on its own separate cache line
• Lock holder frees the lock by “freeing” the next processors cache line.
Results

Static backoff has higher overhead when backoff is inappropriate
Dynamic backoff has higher overheads when static delay is appropriate
• as collisions are still required to tune the backoff time
Queue is better when contention occurs, but has higher overhead when it does not.
• Issue: Preemption of queued CPU blocks rest of queue (worse than simple spin locks)

MCS Locks
Each CPU enqueues its own private lock variable into a queue and spins on it
• No contention
On lock release, the releaser unlocks the next lock in the queue
• Only have bus contention on actual unlock
• No livelock (order of lock acquisitions defined by the list)

MCS Lock
Requires
• compare_and_swap()
• exchange()
  – Also called fetch_and_store()


© Kevin Elphinstone. Distributed under Creative Commons Attribution License
Sample MCS code for ARM MPCore

```c
void mcs_acquire(mcs_lock *L, mcs_qnode_ptr I)
{
    I->next = NULL;
    MEM_BARRIER;
    mcs_qnode_ptr pred = (mcs_qnode*) SWAP_PTR( L, (void *)I);
    if (pred == NULL) {
        /* lock was free */
        MEM_BARRIER;
        return;
    }
    I->waiting = 1; // word on which to spin
    MEM_BARRIER;
    pred->next = I; // make pred point to me
}
```

Selected Benchmark

Compared
- test and test and set
- Anderson’s array based queue
- test and set with exponential back-off
- MCS

Confirmed Trade-off

Queue locks scale well but have higher overhead
Spin Locks have low overhead but don’t scale well
What do we use?

Beng-Hong Lim and Anant Agarwal, “Reactive Synchronization Algorithms for Multiprocessors”, ASPLOS VI, 1994
Idea
Can we dynamically switch locking methods to suit the current contention level???

Issues
How do we determine which protocol to use?
• Must not add significant cost
How do we correctly and efficiently switch protocols?
How do we determine when to switch protocols?

Protocol Selection
Keep a “hint”
Ensure both TTS and MCS lock a never free at the same time
• Only correct selection will get the lock
• Choosing the wrong lock will result in retry which can get it right next time
• Assumption: Lock mode changes infrequently
  – hint cached read-only
  – infrequent protocol mismatch retries

Changing Protocol
Only lock holder can switch to avoid race conditions
• It chooses which lock to free, TTS or MCS.
When to change protocol

Use threshold scheme
- Repeated acquisition failures will switch mode to queue
- Repeated immediate acquisition will switch mode to TTS

Results

The multicore evolution and operating systems

Frans Kaashoek

Joint work with: Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, Robert Morris, and Nickolai Zeldovich

MIT

How well does Linux scale?

- Experiment:
  - Linux 2.6.35-rc5 (relatively old, but problems are representative of issues in recent kernels too)
  - Select a few inherent parallel system applications
  - Measure throughput on different # of cores
  - Use tmpfs to avoid disk bottlenecks

- Insight 1: Short critical sections can lead to sharp performance collapse

Non-scalable locks are dangerous.


Off-the-shelf 48-core server (AMD)

- Cache-coherent and non-uniform access
- An approximation of a future 48-core chip
Poor scaling on stock Linux kernel

Y-axis: (throughput with 48 cores) / (throughput with one core)

Exim on stock Linux: collapse

Oprofile shows an obvious problem

Exim on stock Linux: collapse

Oprofile shows an obvious problem
Oprofile shows an obvious problem

Bottleneck: reading mount table
- Delivering an email calls sys_open
- sys_open calls

Bottleneck: reading mount table
- sys_open calls:

What causes the sharp performance collapse?
- Linux uses ticket spin locks, which are non-scalable
- So we should expect collapse [Anderson 90]
- But why so sudden, and so sharp, for a short section?
  - Is spin lock/unlock implemented incorrectly?
  - Is hardware cache-coherence protocol at fault?

Scalability collapse caused by non-scalable locks [Anderson 90]
Scalability collapse caused by non-scalable locks [Anderson 90]

```c
void spin_lock(spinlock_t *lock) {
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket);
} /* Spin */
}

void spin_unlock(spinlock_t *lock) {
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```
Scalability collapse caused by non-scalable locks [Anderson 90]

```c
void spin_lock(spinlock_t *lock) {
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket);
} /* Spin */
}

void spin_unlock(spinlock_t *lock) {
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Why collapse with short sections?

- Arrival rate is proportional to # non-waiting cores
- Service time is proportional to # cores waiting ($k$)
  - As $k$ increases, waiting time goes up
  - As waiting time goes up, $k$ increases
  - System gets stuck in states with many waiting cores

Short sections result in collapse

- Experiment: 2% of time spent in critical section
- Critical sections become “longer” with more cores
- Lesson: non-scalable locks fine for long sections

Avoiding lock collapse

- Unscalable locks are fine for long sections
- Unscalable locks collapse for short sections
  - Sudden sharp collapse due to “snowball” effect
- Scalable locks avoid collapse altogether
  - But requires interface change

Scalable lock scalability

- It doesn’t matter much which one
- But all slower in terms of latency
Avoiding lock collapse is not enough to scale

- “Scalable” locks don’t make the kernel scalable
- Main benefit is avoiding collapse: total throughput will not be lower with more cores
- But, usually want throughput to keep increasing with more cores