



MULTIPROCESSING, LOCKING AND SCALABILITY

Peter Chubb and Ihor Kuz
first.last@nicta.com.au



Australian Government
Department of Broadband, Communications
and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners



1. Multiprocessors
2. Cache Coherency
3. Locking
4. Scalability

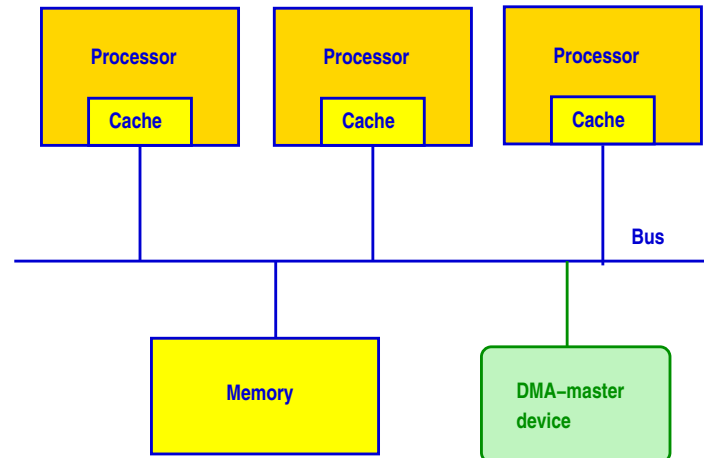
Multiprocessors:

- Moore's law running out of steam
- So scale out instead of up.
- Works well only for some applications!

For a long time people have been attempting 'scale-out' instead of 'scale-up' solutions to lack of processing power. The problem is that for a uniprocessor, system speed increases (including I/O and memory bandwidth) are linear for a geometric increase in cost, so it's cheaper to buy two machines than to buy one more expensive machine with twice the actual performance. As a half-way point, have become popular — especially as the limits to Moore's Law scalability are beginning to be felt.

Classical symmetric Multiprocessor (SMP)

- Processors with local caches
- Connected by bus
- Separated cache hierarchy
- \Rightarrow cache coherency issues

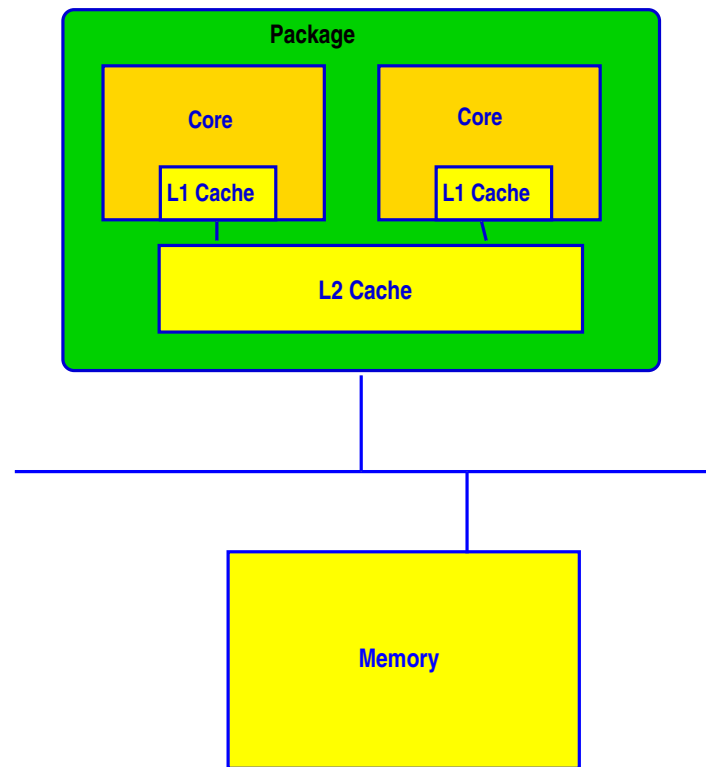


In the classical multiprocessor, each processor connects to shared main memory and I/O buses. Each processor has its own cache hierarchy. Typically, caches are *write-through*; each cache *snoops* the shared bus to invalidate its own cache entries.

There are also non-symmetric multiprocessor designs. In these, some of the processors are designated as having a special purpose. They may have different architectures (e.g., the IBM Cell processors) or may be the same (e.g., the early m68k multiprocessors, where to avoid problems in the architecture, one processor handled page faults, and the other(s) ran normal code). In addition, the operating system can be structured to be asymmetric: one or a few cores can be dedicated to running OS code, the rest, user code.

Multicore (Chip Multiprocessor, CMP)

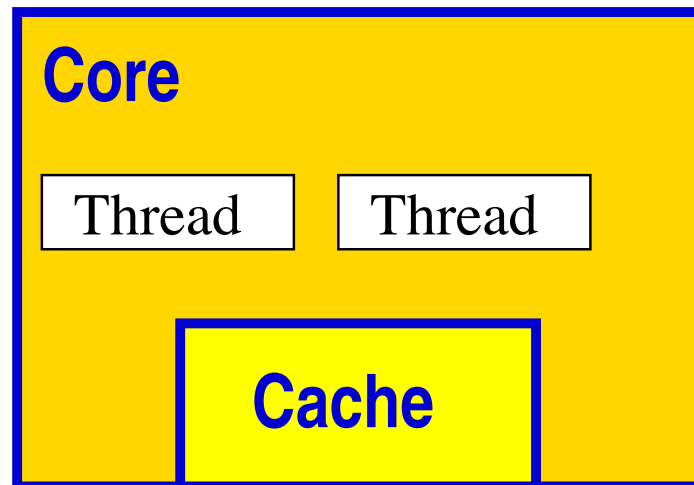
- per-core L1 caches
- Other caches shared
- Cache consistency addressed in h/w



It has become common for manufacturers to put more than one processor in a package. The exact cache levels that are shared varies from architecture to architecture; L1 cache is almost always shared; L2 sometimes, and L3 almost never, although the small number of cores per package mean that broadcast cache-coherency policies can be made to work.

Symmetric Multithreading (SMT)

- Multiple functional units
- Interleaved execution of several threads
- Fully shared cache hierarchy



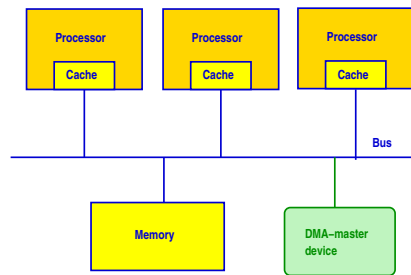
Almost every modern architecture has multiple functional units. As instruction-level parallelism often isn't enough to keep all of the units busy, some architectures offer some kind of *symmetric multithreading*, (one variant of this is called *hyperthreading*). The main difference between architectures is whether threads are truly concurrent (x86 family), interleaved (Niagara) or switched according to event (Itanium switches on L3 cache miss). These don't make a lot of difference from the OS point of view, however.

Cache Coherency:

Processor A writes a value to address x *then...*

Processor B reads from address x

Does Processor B see the value Processor A wrote?



Whenever you have more than one cache for the same memory, there's the issue of coherency between those caches.

Snoopy caches:

- Each cache watches bus write traffic, and invalidates cache lines written to
- Requires *write-through* caches.

On a sufficiently small system, all processors see all bus operations. This obviously requires that write operations make it to the bus (or they would not be seen).

- Having to go to the bus every time is **s l o w**.
- Out-of-order execution on a single core becomes problematic on multiple cores
 - *barriers*

As the level of parallelism gets higher, broadcast traffic from every processor doing a write can clog the system, so instead, caches are *directory-based*.

In addition, although from a single-core perspective loads and stores happen in program order, when viewed from another core they can be out-of-order. More later...

- There are many different coherency models used.
- We'll cover MESI only (four states).
 - Some consistency protocols have more states (up to ten!)
- 'memory bus' actually allows complex message passing between caches.

'Under the hood' that simple bus architecture is a complex message-passing system. Each cache line can be in one of a number of states; cache line states in different caches are coordinated by message passing.

This material is adapted from the book chapter by McKenney (2010).

MESI:

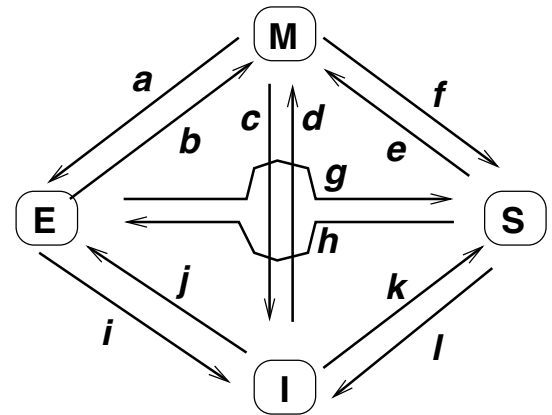
Each cache line is in one of four states:

M Modified

E Exclusive

S Shared

I Invalid



One commonly used protocol is the MESI protocol. Each cache line (containing more than one datum) is in one of four states. In the *Modified* state, the cache line is present only in the current cache, and it has been modified locally.

In the *Exclusive* state, the cache line is present only in the current cache, but it has not been modified locally.

In the *Shared* state, the data is read-only, and possibly present in other caches with the same values.

The *Invalid* state means that the cache line is out-of-date, or doesn't match the 'real' value of the data, or similar. In any event, it is not to be used.

MESI protocol messages:

Caches maintain consistency by passing messages:

Read

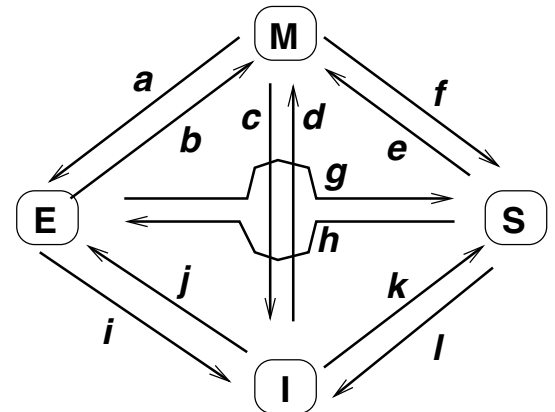
Read Response

Invalidate

Invalidate acknowledge

Read invalidate

Writeback

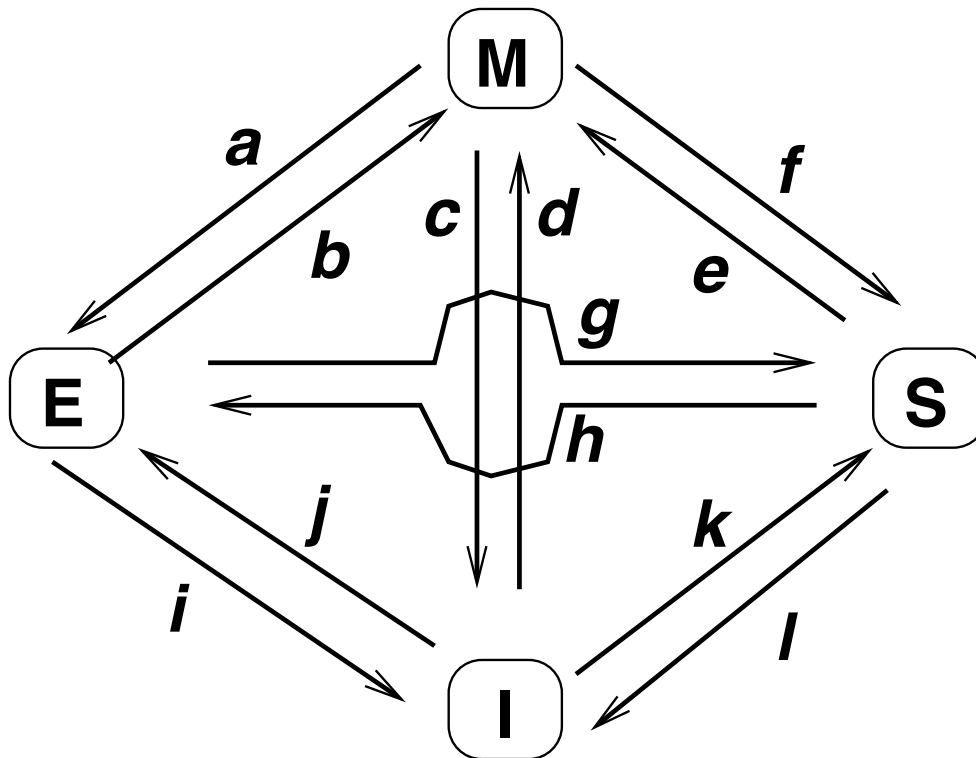


A *Read* message contains a physical address. A *read response* message contains the data held in that address: it can be provided either by main memory or by another processor's cache. An *invalidate* message contains a physical address. It says to mark the cache line containing that address as invalid. Each cache that contains the line must generate an *invalidate acknowledge* message; on small systems that do not use directory-based caching, *all* caches have to generate an *invalidate acknowledge*.

Read invalidate combines both *read* and *invalidate* messages into one: presumably the processor is getting a cache line to write to it. It requires both *read response* and *Invalidate Acknowledge* messages in return.

The *Writeback* message contains a physical address, and data to be written to that address in main memory. Other processors'

caches may *snoop* the data on the way, too. This message allows caches to eject lines in the *M* state to make room for more data.



- a** $M \rightarrow E$ A cache line is written back to memory (*Writeback* message) but the processor maintains the right to modify the cacheline
- b** $E \rightarrow M$ The processor writes to a cache line it already had exclusive access to. No messages are needed.
- c** $M \rightarrow I$ The processor receives a *read invalidate* message for a cacheline it had modified. The processor must invalidate its local copy, then respond with both a *Read Response* and an *Invalidate Acknowledge* message.
- d** $I \rightarrow M$ The processor is doing an atomic operation (read-modify-write) on data not in its cache. It sends a *Read Invalidate* message; it can complete its transition when

it has received a full set of *Invalidate acknowledge* responses.

- e $S \rightarrow M$ The processor is doing an atomic operation on a data item that it had a read-only shared copy of in its cache. It cannot complete the state transition until all *Invalidate acknowledge* responses have been received.
- f $M \rightarrow S$ Some other processor reads (with a *Read* message) the cache line, and it is supplied (with a *Read Response*) from this cache. The data may also be written to main memory.
- g $E \rightarrow S$ Some other processor reads data from this cache line, and it is supplied either from this processor's cache

or from main memory. Either way, this cache retains a read-only copy.

- h $S \rightarrow E$ There can be two causes for this transition: either all other processors have moved the cacheline to *Invalid* state, so this is the last copy; or this processor has decided it wants to write fairly soon to this cacheline, and has transmitted an *Invalidate* message. In the second case, it must wait for a full set of *Invalidate Acknowledge* responses before completing the transition.
- i $E \rightarrow I$ Some other processor does an atomic operation on a datum held only by this processor's cache. The transition is initiated by a *Read Invalidate* message; this processor responds with both *Read Response* and *Invalidate Acknowledge*.

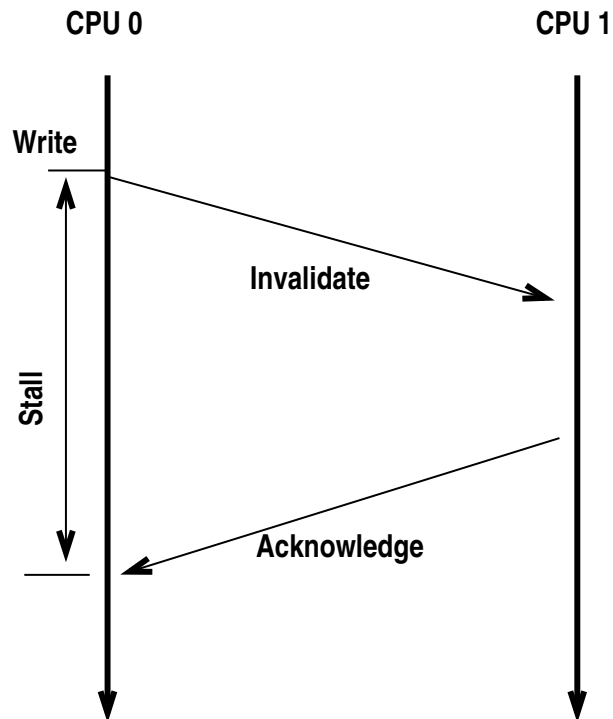
- j** $I \rightarrow E$ This processor is attempting to store to an address not currently cached. It will transmit a *Read Invalidate* message; and will complete the transition only when a *Read Response* and a full set of *Invalidate Acknowledge* messages have been received. The cacheline will usually move $E \rightarrow M$ soon afterwards, when the store actually happens.

- k** $I \rightarrow S$ This processor wants to get some data, It sends *Read* and receives *Read Response*

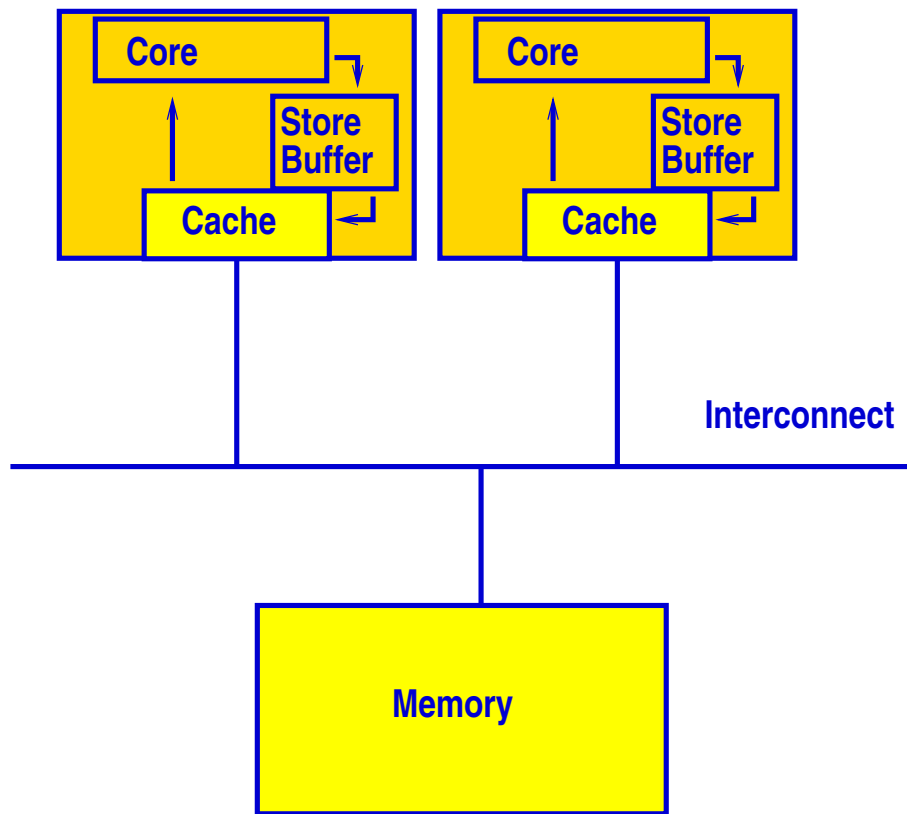
- l** $S \rightarrow I$ Some other processor is attempting to modify the cache line. An *Invalidate* message is received; an *Invalidate Acknowledge* is sent.

- Why don't *Invalidate Acknowledge* storms saturate interconnect?
 - ⇒ simple bus doesn't scale; add *directory* to each cache that tracks who holds what cache line.

With all this bus traffic, cache line bouncing can be a major concern tying up the interconnect for relatively long periods of time. In addition, if a cache is busy, because of the necessity to wait for a remote transaction to complete, the current processor is stalled

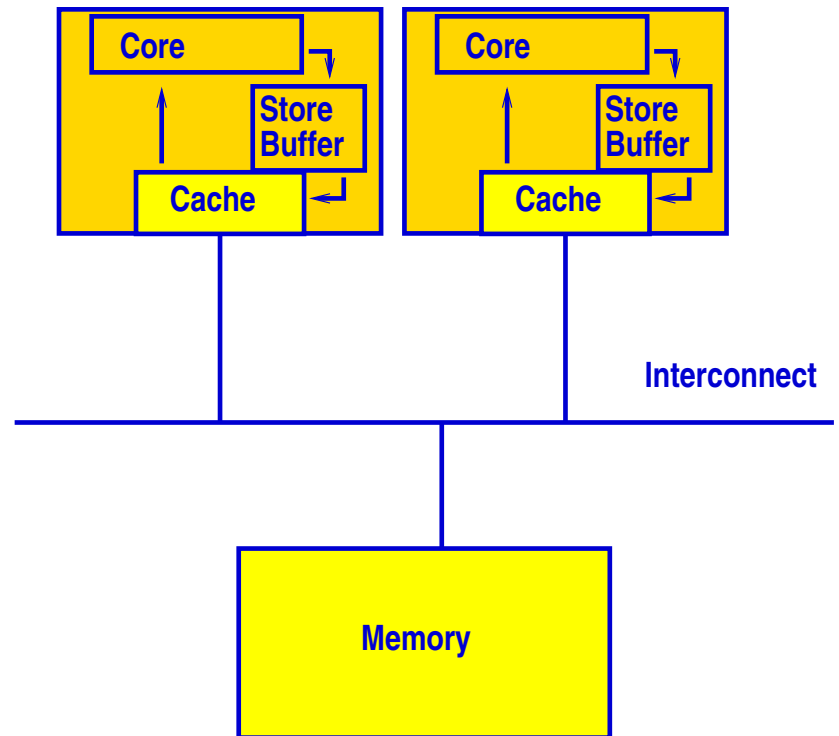


While waiting for all the invalidate-acknowledgements, the processor can make no forward progress.



In most architectures this latency is hidden by queueing up stores in a *store buffer*. When the processor does a write to a cache-line in *Invalid* or *Shared* states, it sends a read-invalidate or a invalidate message, and then queues the write to its store buffer. It can then continue with its next operation without stalling.

Problems:



`a = 1`

`b = a + 1`

`assert(b == 2)`

If this is all it did, there would be problems. Imagine `a` is not in the current cache (state *Invalid*).

1. `a` not in cache, sends *Read Invalidate*
2. $a \leftarrow 1$ in store buffer
3. starts executing `b=a+1`, needs to read `a`
4. gets *Read Response* with `a==0`
5. loads `a` from cache
6. applies store from store buffer writing 1 to cache
7. adds one to the loaded value of `a` and stores it into `b`
8. Assertion now fails because `b` is 1.

Solution is *Store Forwarding*

Processors snoop their store buffers as well as their caches on loads.

Local ops seen in program order

Insufficient in a Multiprocessor.

CPU 0	CPU 1
a = 1 b = 1	while (b==0) continue; assert(a == 1);

Start with a in CPU 1's cache; b in CPU 0.

1. CPU0: $a \leftarrow 1$. New value of a to store buffer, send *Read Invalidate*.
2. CPU1: reads b, sends *Read* message.
3. CPU0: executes $b \leftarrow 1$. It owns this cache line, so no messages sent.
4. CPU0 gets *Read*; sends value of b (now 1), marks it Shared.
5. CPU1 receives *Read Response*, breaks out of while loop, and the assertion fails.
6. CPU1 gets the *Read Invalidate* message and sends the cache line containing a to CPU 0.
7. CPU0 finally gets the *Read Response*

The hardware cannot know about data-dependencies like these, and needs help from the programmer. Almost every MP-capable architecture has some form of *barrier* or *fence* instruction, that waits until anything in the store buffer is visible to all other processors, and also tweaks the out-of-order engine (if any) to control whether stores and loads before this instruction can be occur afterwards, and vice versa.

Invalidate Queues:

- Invalidates take too long (busy caches, lots of other processors sending Invalidates)
- So buffer them:
 - Send Invalidate Acknowledge immediately
 - Guarantee not to send any other MESI message about this cache line until Invalidate completes.
- Can give *more* memory ordering problems (McKenney 2010)

Busy caches can delay *Invalidate Acknowledge* messages for a long time. This latency is hidden using an 'Invalidate Queue'. A processor with an invalidate queue can send an *Invalidate Acknowledge* as soon as it receives the *Invalidate Request*. The *Invalidate Request* is queued instead of being actioned immediately.

Placing an Invalidate Request in the queue is a promise not to send any MESI protocol messages that relate to that cache line until the line has been invalidated.

Barriers: (also called *fences*)

Write barrier Waits for store buffer to drain

Read barrier Waits for Invalidate Queue to drain

Memory barrier Waits for both

All barriers also tweak out-of-order engine.

In addition to waiting for queues to drain, barriers tell the out-of-order execution engine (on OOO processors) not to move writes past a write barrier, or reads before a read barrier. This ensures in a critical section that instructions don't 'leak'.

Guarantees:

1. Each processor sees its own memory accesses in program order
2. Processors may reorder writes only if they are to different memory locations
3. All of a processor's loads before a read barrier will be perceived by all processors to precede any load after that read barrier
4. Likewise for stores and a write barrier, or loads and stores and a full barrier

Particular processors may give stronger guarantees than these. In particular, X86 is fairly strongly ordered, and sometimes you can get away without a barrier that would be needed, say, on Alpha.

Another example:

```
void foo() {          void bar(void) {
    a = 1;            while (!b)
    mb();              ;
    b = 1;            assert (a == 1);
}                    }
```

Assume $a == 0$ in *Shared* state; $b == 0$ in *Exclusive* state in CPU 0; CPU0 does `foo()`, CPU1 does `bar()`

CACHE COHERENCY

1. CPU0 puts $a \leftarrow 1$ into store buffer; sends *Invalidate*.
2. CPU 1 starts `while (!b)`; sends *Read* and stalls.
3. CPU 1 gets *Invalidate*, puts it into queue, and responds.
4. CPU 0 gets *Invalidate Acknowledge*, completes $a \leftarrow 1$, moves past barrier.
5. CPU 0 does $b \leftarrow 1$; no need for store buffer.
6. CPU 0 gets *Read* for `b`, sends *Read Response*, transition to *Shared*
7. CPU 1 gets *Read Response* with $b == 1$, breaks out of `while(...)`.

8. CPU 1 reads a from cache, it's still 0 so assertion fails.
9. CPU 1 completes *Invalidate* on a **Too late!**

Fix:

```
void foo() {
    a = 1;
    wmb();
    b = 1;
}

void bar(void) {
    while (!b)
        ;
    rmb();
    assert (a == 1);
}
```

Adding a read barrier (or a full barrier) after the loop makes sure that the read of a cannot happen before the read of b — so (after a lot of coherency traffic) this will cause the assertion not to trigger.

DMA and I/O consistency:

- PCI MMIO writes are *posted* (i.e., queued) and can occur *out of order* WRT other I/Os.
- PCI I/O writes are *strongly ordered* and are effectively a barrier. wrt MMIO writes.
- Depending on architecture, CPU memory barriers may or may not be enough to serialise I/O reads/writes.
 - DMA not necessarily coherent with CPU cache: some bus-mastering devices need cache flushes.

We've talked a lot about memory consistency between processors – what about DMA? What of IPI?

Different I/O devices differ; some respect cache coherency, others do not. Read the docs!

In particular note that MMIO writes are 'posted' — they can be queued. The PCI spec says that any read from a device must force all writes to the device to complete.

Other buses have different memory-ordering issues. Be aware of them!

Some cases (from (McKenney 2010)): a processor is very busy, and holds onto a cache line, so that when a device's DMA complete interrupt arrives, it still has old data in its cache.

Context switching also needs appropriate memory barriers, so if a thread is migrated from one processor to another, it sees its current data.

- Shared data an issue
- How to maintain data consistency?
- Critical Sections
- Lock-free algorithms

In general to get good scalability, you need to avoid sharing data. However, sometimes shared data is essential.

Single word reads and writes are atomic, always; the visibility of such operations can be controlled with barriers as above.

When updating some data requires more than one cycle, there's the possibility of problems, when two processors try to update related data items at the same time. For instance, consider a shared counter. If it has the value 2 to start with, and CPU0 wants to add 3 and CPU1 wants to add 1, the result could be 3, 5 or 6. Only 6 is the answer we want!

To solve this problem, we identify *critical sections* of code, and lock the data during those sections of code. However, locks have problems (in particular heavily contended locks cause much cache coherency traffic) and so for some common cases it's possible to use lock-free algorithms to ensure consistency.

Lock Granularity:

- Coarse grained: 'Big Kernel Lock'
 - Kernel becomes a *monitor* (Hoare 1974)
 - At most one process/processor in kernel at a time
 - Limits kernel scalability
- Finer grained locking
 - Each group of related data has a lock
 - If carefully designed, can be efficient

Good discussion of trade-offs in ch 10 Schimmel (1994),

NICTA Copyright © 2011

From Imagination to Impact

30

The simplest locking is to treat all kernel data as shared, and use a single Big Kernel Lock to protect it. The Kernel then essentially becomes a Hoare Monitor.

The main problem with this is that it doesn't scale to very many processors. For systems that do not spend much time in the kernel, and do not have very many processors, this is a good solution though. It has also been used as a first step in converting from a uniprocessor OS to a multiprocessor OS.

The next step is generally identifying large chunks of data (e.g., the process table, the namei cache etc.) and providing a single lock for each.

As systems have grown, it's been noticed that finer-grain access to these things is desirable, and locks have become finer over time.

Lock Data not Code!

The one thing to remember, is that we're locking data, not code. Be very clear over what data items need to be kept consistent, and insert appropriate locks into the code that manipulates that data.

Uniprocessor Considerations:

- Just need to protect against preemption
- (and interrupt handlers)
 - disable/enable interrupts is sufficient
 - some processors have multiple interrupt levels
sp10...sp17

On a uniprocessor, the only way for multiple threads of control to access the same data at a time is by a context switch, either to interrupt context, or to a different thread. As context switches happen only in two circumstances: voluntarily, or in response to an interrupt, it suffices to disable interrupts.

Some processors have multiple interrupt levels (e.g., M68K); operating systems for such architectures can selectively disable interrupts.

Simple spin locks:

```
spinlock(volatile int *lp)
{
    while (test_and_set(*lp))
        ;
}
spinunlock(volatile int *lp)
{
    *lp = 0;
}
```

The simplest way to implement a spinlock, is to use an atomic test-and-set instruction sequence. On x86, this is usually a compare-exchange sequence; on m68k a `tas` instruction; ARM instead uses a send and receive event pair.

Such atomic operations usually have implicit memory barriers.

Issues with simple Spinlocks:

- Need to disable interrupts as well
- Wasted time spinning
- Too much cache coherency traffic
- Unfairness

Spinlocks have problems. To prevent context switching, you need to disable interrupts as well as hold the spinlock. While spinning, the processor uses power and wastes CPU cycles. Every test-and-set instruction sends a *Read Invalidate* message and gets all the *Invalidate Acknowledge* results if there's any contention at all. And if there's a code sequence where a processor releases the lock, then reacquires it, it's likely not to let any other processor in (because the lock is locally cache-hot).

Ameliorating the problem:

- Spin on read (Segall & Rudolph 1984)

```
while (*lp || test_and_set(*lp))  
    ;
```

- Most processors spin on `while(*lp)`
- The `test_and_set()` invalidates all locks, causes coherency traffic.
- Better than plain `test_and_set()` but not perfect.
- **Unfairness still an issue**

Segall & Rudolph (1984) suggested guarding the test-and-set with a read. That way, the spinning processors all have the cache-line in *Shared* state, and spin on their own local caches until the lock is released. This reduces coherency traffic, although there'll still be a thundering horde at unlock time. And the resulting lock is still unfair.

- Ticket locks (Anderson 1990, Corbet 2008)
 - Queue of waiting processes,
 - Always know which one to let in next —
Deterministic latencies run to run
 - As implemented in Linux, coherency traffic still an issue.

If you've ever bought Cheese at David Jones, you'll be familiar with a ticket lock: each person takes a ticket with a number on it; when the global counter (controlled by the server) matches the number on your ticket, you get served. Nick Piggin put these into the Linux kernel in 2008; his contribution was to put the ticket and the lock into a single word so that an optimal code sequence could be written to take the lock and wait for it.

```
lock(lp)
{
    x = atomic_inc(lp->next);
    while (lp->current != x)
        ;
}
```

```
unlock(lp)
{
    lp->current++;
}
```

Logically, the code sequence is as in the slide (I've omitted the necessary read barriers). It's assumed that the hardware provides a way to atomically increment a variable, do a write memory barrier and return its old value. If the lock was heavily contested it would make sense to have the ticket counter and the current ticket in separate cache lines, but this makes the lock larger.

- Multi-reader locks
 - Count readers, allow more than one
 - Single writer, blocks readers
- BRlocks

There are other locks used. There are many data structures that are read-mostly. Concurrent reads can allow forward progress for all readers. Multi-reader locks are slightly more expensive to implement than simple spin locks or ticket locks, but can provide good scalability enhancements.

Another lock used is the so-called brlock, or local-global lock. These use per-cpu spinlocks for read access. A process attempting write access needs to acquire all the per-cpu spinlocks. This can be *very* slow.

Other locking issues:

- Deadlocking
- Convoying
- Priority inversion
- Composibility issues

All locks have other issues. Lock ordering is important: if two processes each have to take two locks, and they do not take them in the same order, then they can deadlock.

Because locks serialise access to a data structure, they can cause processes to start proceeding in lock step — rather like cars after they're released from traffic lights.

It's possible for a process to be blocked trying to get a lock held by a lower priority process. This is usually only a problem with sleeping locks — while holding a spinlock, a process is always running. There have been many solutions proposed for this (usually involving some way to boost the priority of the process holding the lock) but none are entirely satisfactory.

Finally there are issues with operation composability: how do you operate on several data structures at once, when each is protected by a different lock? The simplest way is always to take

all the locks, but to watch for deadlocks.

Sleeping locks:

semaphore (Dijkstra 1965)

mutex just a binary semaphore

adaptive spinlock

In 1965, Edsger Dijkstra proposed a way for processes to co-operate using a *semaphore*. The idea is that there is a variable which has a value. Any attempt to reduce the value below zero causes the process making the attempt to sleep until the value is positive. Any attempt to increase the value when there are sleeping processes associated with the semaphore wakes one of them.

A semaphore can thus be used as a counter (of number of items in a buffer for example), or if its values are restricted to zero or one, it can be used as a mutual-exclusion lock (mutex).

Many systems provide *adaptive spinlocks* — these spin for a short while, then if the lock has not been acquired the process is put to sleep.

Doing without locks: (See Fraser & Harris (2007) and McKenney (2003))

- Atomic operations
 - Differ from architecture to architecture
 - compare-exchange common
 - Usually have implicit memory barrier
 - Fetch-*op* useful in large multiprocessors: *op* carried out in interconnect.

Simple operations can be carried out lock free. Almost every architecture provides some number of atomic operations. Architectures that do not, can emulate them using spinlocks. Some architectures may provide fetch-and-add or fetch-and-mul etc., instructions. These are implemented in the interconnect or in the L2 cache, and provide a way of atomically updating a variable and returning its previous or subsequent value.

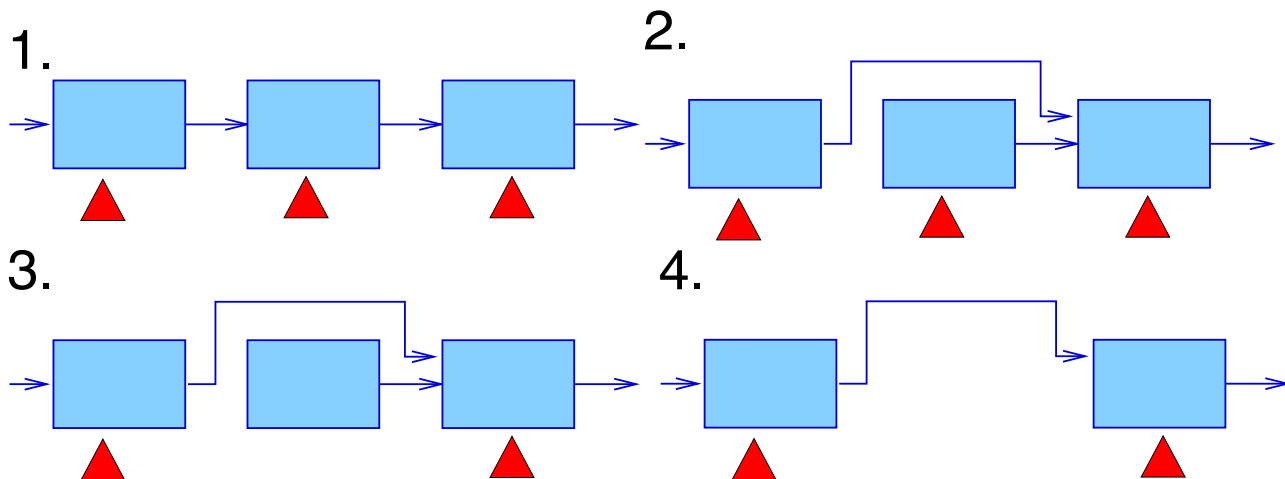
Optimism:

- Generation counter updated on write
- Check before and after read/calc: if changed retry
- Also called *Sequence Locks*

If you can afford to waste a little time, then an optimistic lock can be used. The idea here is that a reading process grabs a generation counter before starting, and checks it just after finishing. If the generation numbers are the same, the data are consistent; otherwise the operation can be retried.

Concurrent writers still need to be serialised. Writers as their last operation update the generation count.

RCU: (McKenney 2004, McKenney, Sarma, Arcangelli, Kleen, Krieger & Russell 2002)



Another way is so called *read-copy-update*. The idea here is that if you have a data structure (such as a linked list), that is very very busy with concurrent readers, and you want to remove an item in the middle, you can do it by updating the previous item's *next* pointer, but you cannot then free the item just unlinked until you're sure that there is no thread accessing it.

If you prevent preemption while walking the list, then a sufficient condition is that every processor is either in user-space or has done a context switch. At this point, there will be no threads accessing the unlinked item(s), and they can be freed.

Inserting an item without locking is left as an exercise for the reader.

Updating an item then becomes an unlink, copy, update, and insert the copy; leaving the old unlinked item to be freed at the next quiescent point.

The Multiprocessor Effect:

- Some fraction of the system's cycles are not available for application work:
 - Operating System Code Paths
 - Inter-Cache Coherency traffic
 - Memory Bus contention
 - Lock synchronisation
 - I/O serialisation

We've seen that because of locking and other issues, some portion of the multiprocessor's cycles are not available for useful work. In addition, some part of any workload is usually unavoidably serial.

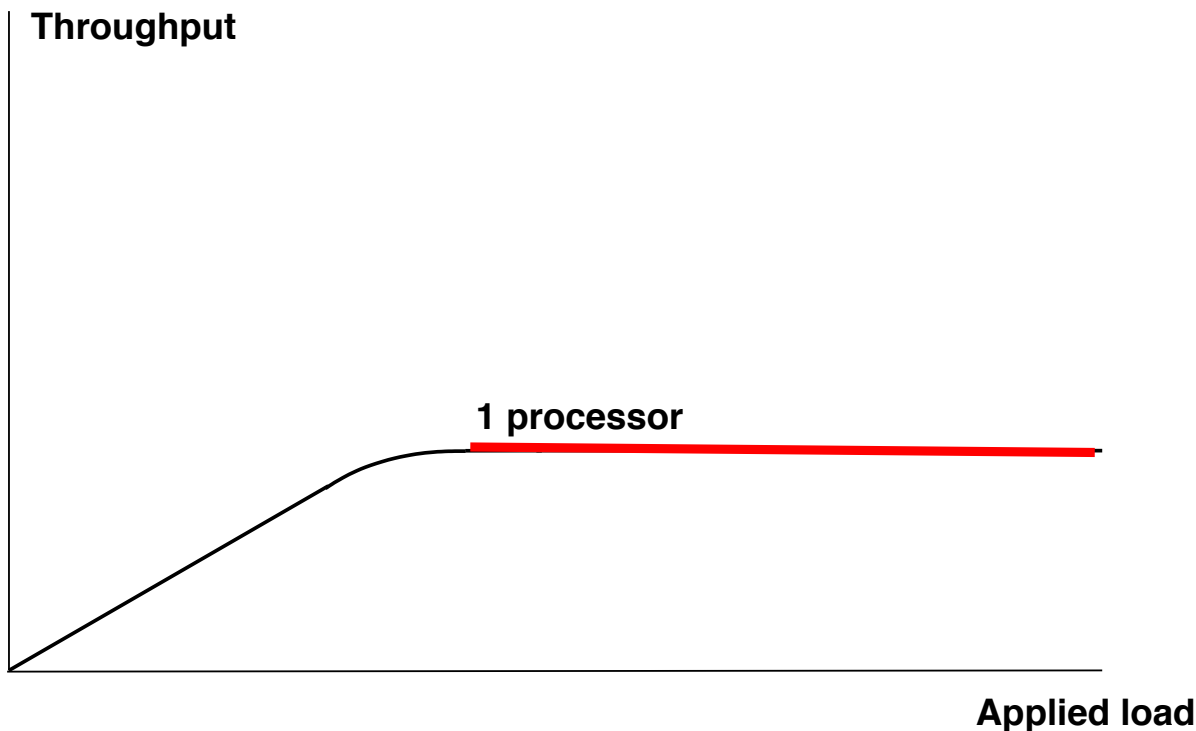
Amdahl's law:

If a process can be split into two parts, and σ of the running time cannot be sped up, but the rest is sped up by running on p processors, then overall speedup is

$$\frac{p}{1 + \sigma(p - 1)}$$

It's fairly easy to derive Amdahl's law: perfect speedup for p processors would be p (running on two processors is twice as fast, takes half the time, that running on one processor).

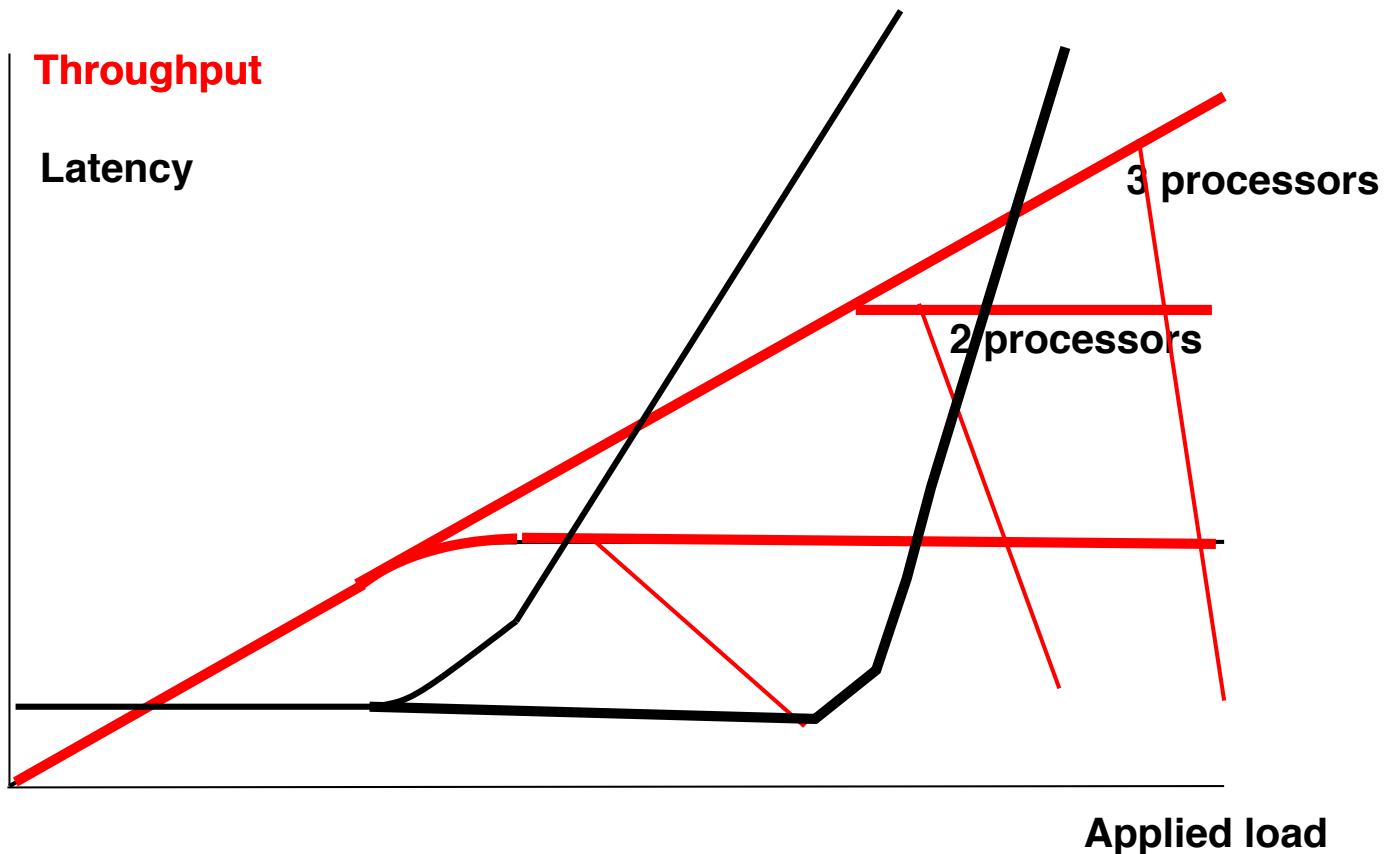
The time taken for the workload to run on p processors if it took 1 unit of time on 1 processor is $\sigma + (1 - \sigma)/p$. Speedup is then $1/(\sigma + (1 - \sigma)/p)$ which, multiplying by p/p gives $p/(p\sigma + 1 - \sigma)$, or $p/(1 + \sigma(p - 1))$



The general scalability curve looks something like the one in this slide. The Y-axis is throughput, the X-axis, applied load. Under low loads, where there is no bottleneck, throughput is determined solely by the load—each job is processed as it arrives, and the server is idle for some of the time. Latency for each job is the time to do the job.

As the load increases, the line starts to curve. At this point, some jobs are arriving before the previous one is finished: there is queueing in the system. Latency for each job is the time spent queued, plus the time to do the job.

When the system becomes overloaded, the curve flattens out. At this point, throughput is determined by the capacity of the system; average latency becomes infinite (because jobs cannot be processed as fast as they arrive, so the queue grows longer and longer), and the bottleneck resource is 100% utilised.



This graph shows the latency ‘hockey-stick’ curve. Latency is determined by service time in the left-hand flat part of the curve, and by service+queueing time in the upward sloping right-hand side.

When the system is totally overloaded, the average latency is infinite.

Gunther's law:

$$C(N) = \frac{N}{1 + \alpha(N - 1) + \beta N(N - 1)}$$

where:

N is demand

α is the amount of serialisation: represents Amdahl's law

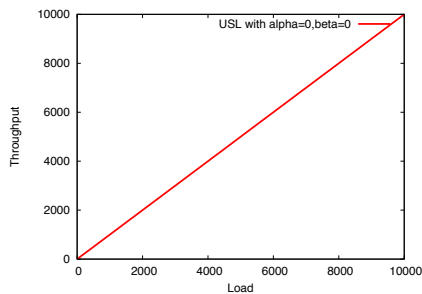
β is the coherency delay in the system.

C is Capacity or Throughput

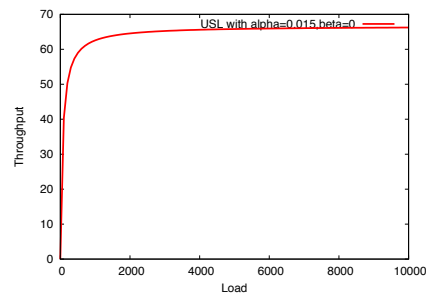
Neil Gunther (2002) captured this in his 'Universal Scalability Law', which is a closed-form solution to the machine-shop-repairman queueing problem.

It has two parameters, α which is the amount of non-scalable work, and β which is to account for the degradation often seen in system-performance graphs, because of cross-system communication ('coherency' or 'contention', depending on the system).

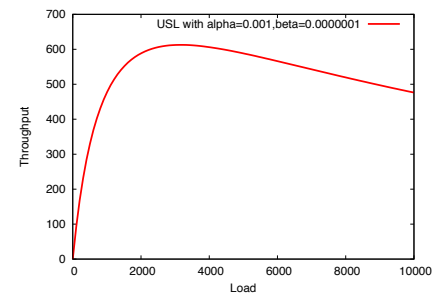
The independent variable N can represent applied load, or number of logic-units (if the work per logic-unit is kept constant).



$$\alpha = 0, \beta = 0$$



$$\alpha > 0, \beta = 0$$



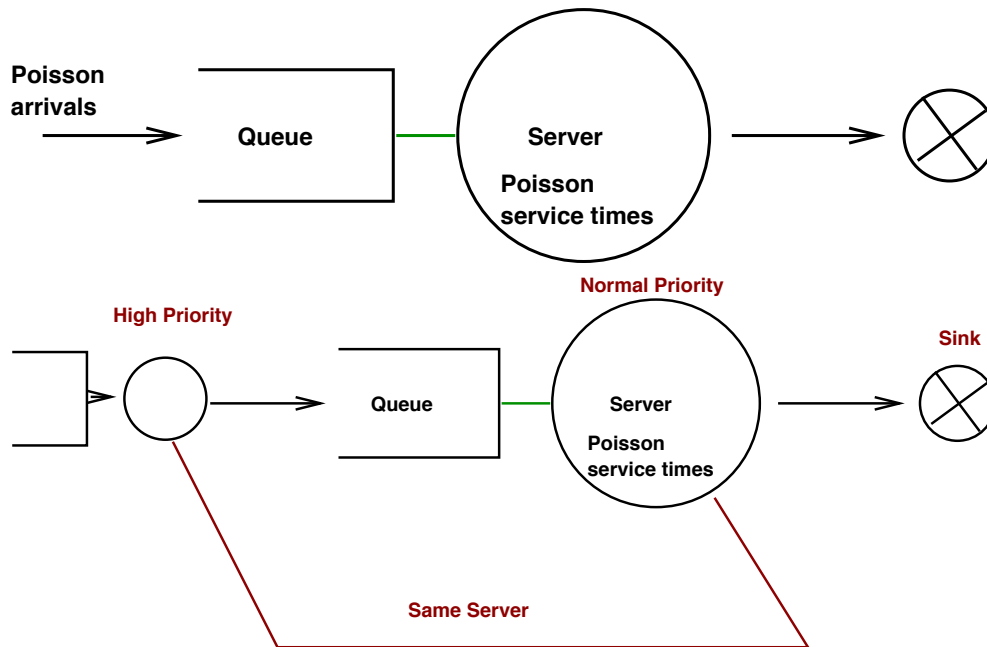
$$\alpha > 0, \beta > 0$$

Here are some examples. If α and β are both zero, the system scales perfectly—throughput is proportional to load (or to processors in the system).

If α is slightly positive it indicates that part of the workload is not scalable. Hence the curve plateaus to the right. Another way of thinking about this is that some (shared) resource is approaching 100% utilisation.

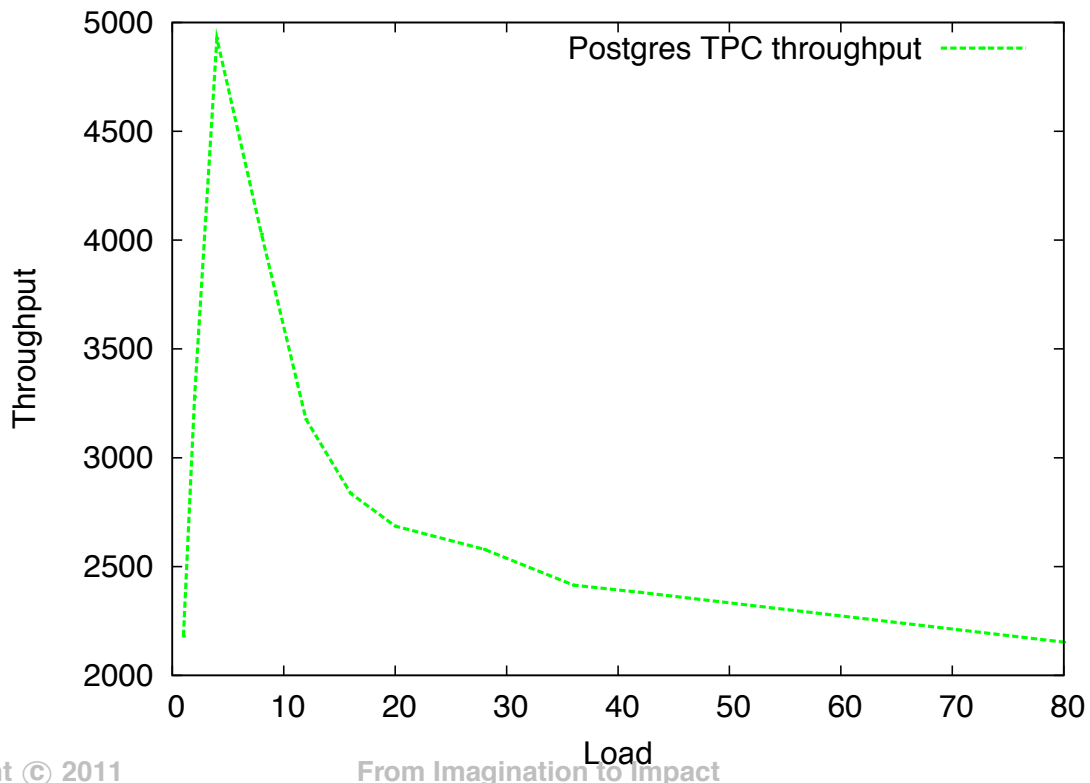
If in addition β is slightly positive, it implies that some resource is contended: for example, preliminary processing of new jobs steals time from the main task that finishes the jobs.

Queueing Models:

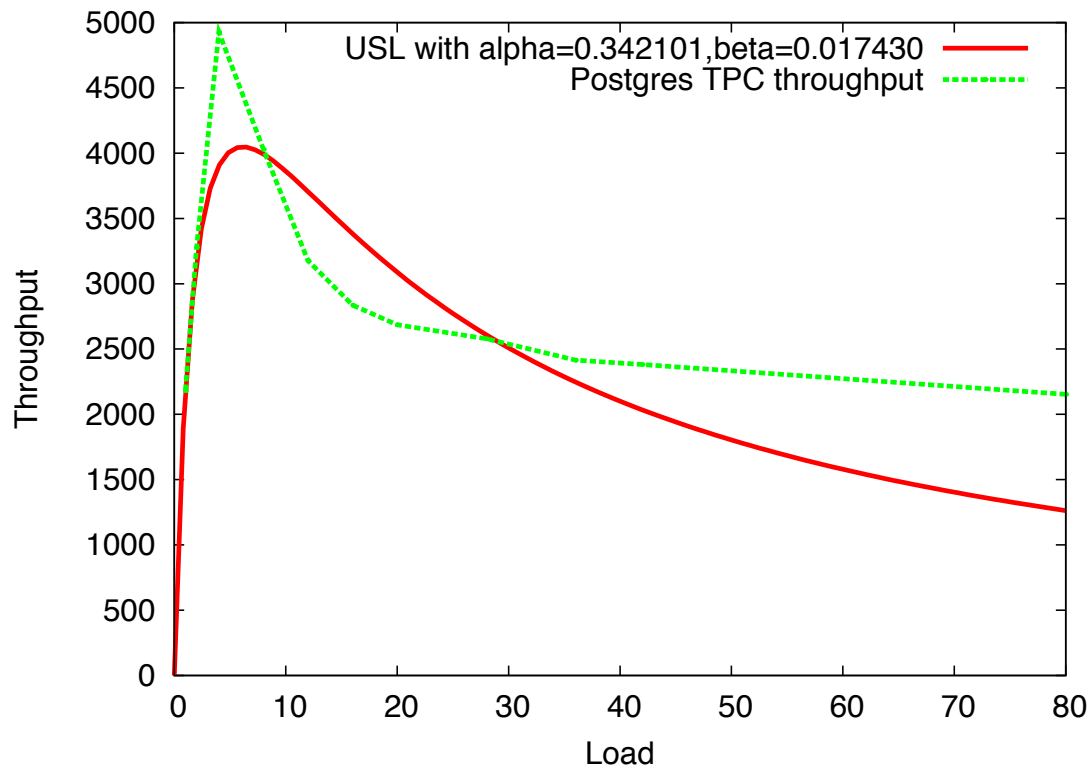


You can think of the system as in these diagrams. The second diagram has an additional input queue; the same servers service both queues, so time spent serving the input queue is stolen from time servicing the main queue.

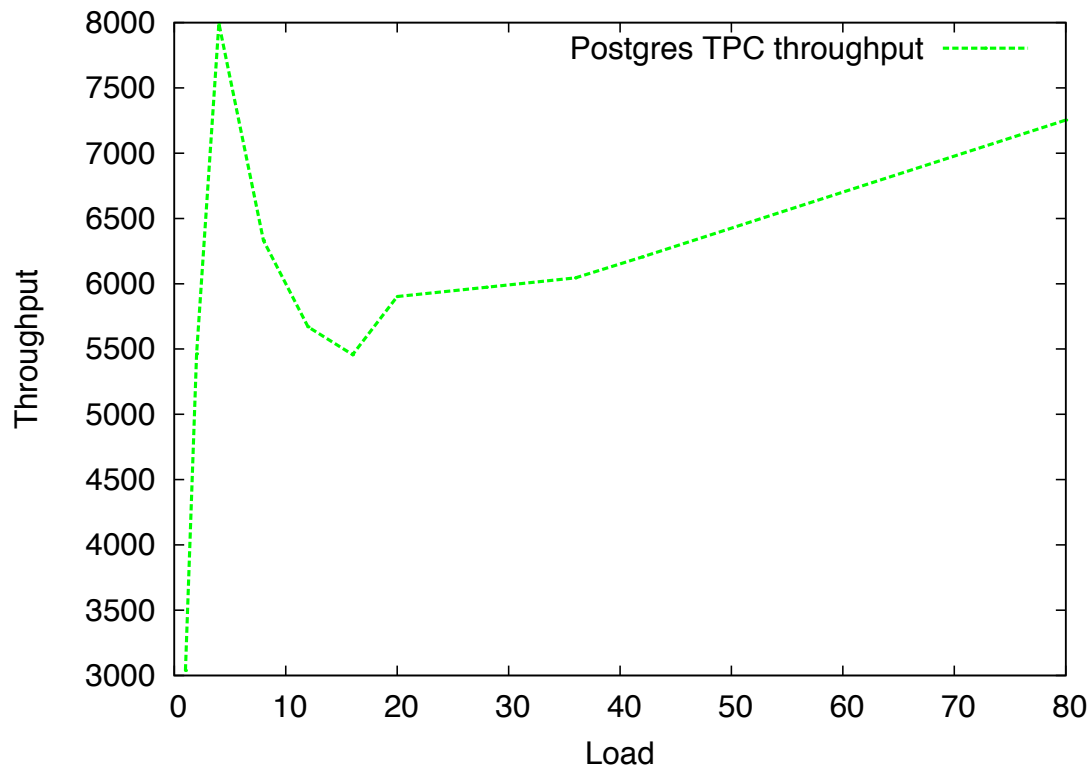
Real examples:



These graphs are courtesy of Etienne and the Rapilog team. This is a throughput graph for TPC-C on an 8-way multiprocessor using the ext3 filesystem with a single disk spindle. As you can see, $\beta > 0$, indicating coherency delay as a major performance issue.

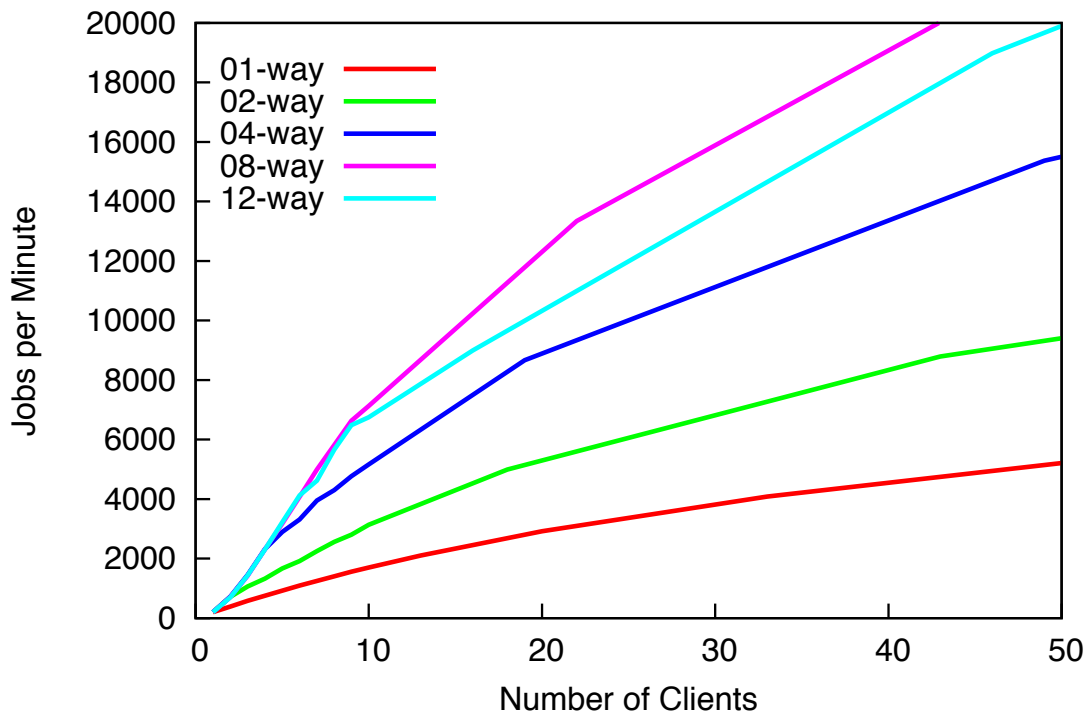


Using R to fit the scalability curve, we get $\beta = 0.017$, $\alpha = 0.342$ — you can see the fit isn't perfect, so fixing the obvious coherency issue isn't going to fix the scalability entirely.



Moving the database log to a separate filesystem shows a much higher peak, but still shows a $\beta > 0$. There is still coherency delay in the system, probably the file-system log. From other work I've done, I know that ext3's log becomes a serialisation bottleneck on a busy filesystem with more than a few cores — switching to XFS (which scales better) or ext2 (which has no log) would be the next step to try.

Another example:



This shows the reaim-7 benchmark running on various numbers of cores on an HP 12-way Itanium system. As you can see, the 12-way line falls below the 8-way line — α must be greater than zero. So we need to look for contention in the system somewhere.

SPINLOCKS		HOLD		WAIT					
UTIL	CON	MEAN(MAX)	MEAN(MAX) (% CPU)	TOTAL	NOWAIT	SPIN	RJECT	NAME	
72.3%	13.1%	0.5us(9.5us)	29us(20ms)(42.5%)	50542055	86.9%	13.1%	0%	<code>find_lock_page+0x30</code>	
0.01%	85.3%	1.7us(6.2us)	46us(4016us)(0.01%)	1113	14.7%	85.3%	0%	<code>find_lock_page+0x130</code>	

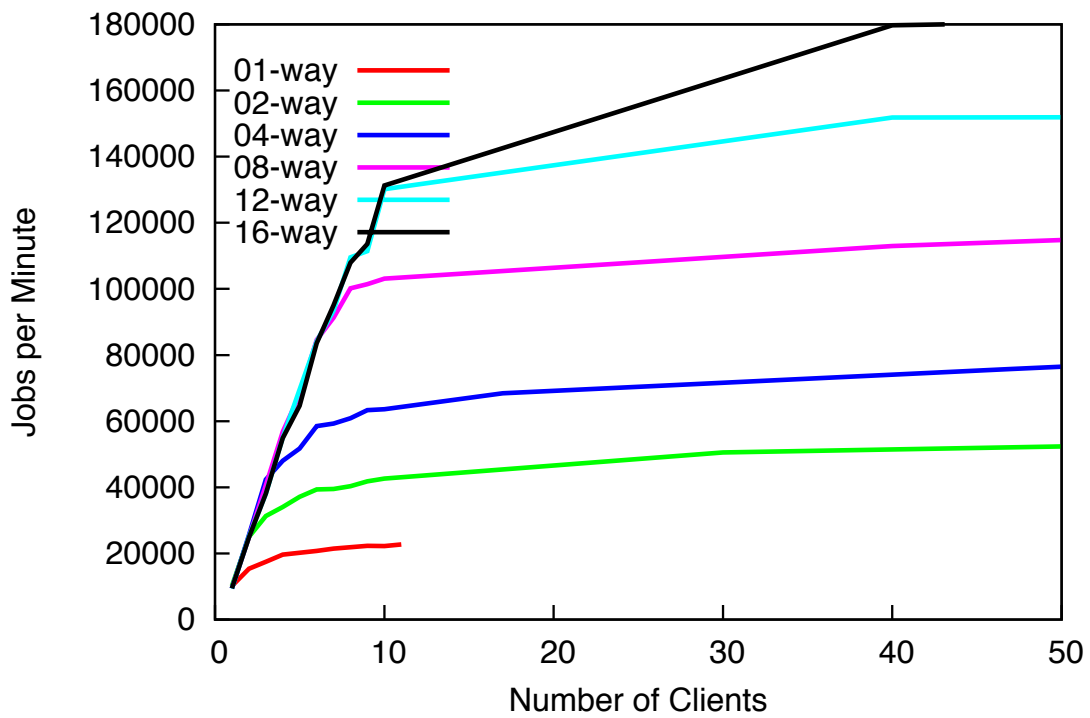
Lockmetering shows that a single spinlock in `find_lock_page()` is the problem:

```
struct page *find_lock_page(struct address_space *mapping,
                            unsigned long offset)
{
    struct page *page;

    spin_lock_irq(&mapping->tree_lock);

repeat:
    page = radix_tree_lookup(&mapping>page_tree, offset);
    if (page) {
        page_cache_get(page);
        if (TestSetPageLocked(page)) {
            spin_unlock_irq(&mapping->tree_lock);
            lock_page(page);
            spin_lock_irq(&mapping->tree_lock);
        }
    }
}
```

So replace the spinlock with a rwlock, and bingo:



The scalability is much much better.

- Find the bottleneck
- fix or work around it
- check performance doesn't suffer too much on the low end.
- Experiment with different algorithms, parameters

REFERENCES

References

- Anderson, T. (1990), 'The performance of spin-lock alternatives for shared memory multiprocessors', *IEEE Transactions on Parallel and Distributed Systems* **1**(1), 6–16.
<http://www.cs.washington.edu/homes/tom/pul>
- Corbet, J. (2008), 'Ticket spinlocks', *Linux Weekly News* .
<http://lwn.net/Articles/267968/>.
- Dijkstra, E. W. (1965), 'Cooperating sequential processes (ewd-123)',

REFERENCES



NICTA

<http://www.cs.utexas.edu/users/EWD/ewd01xx>

Fraser, K. & Harris, T. (2007), 'Concurrent programming without locks', *ACM Trans. Comput. Syst.* **25**.

<http://doi.acm.org/10.1145/1233307.1233309>

Hoare, C. (1974), 'Monitors: An operating system structuring concept', *CACM* **17**, 549–57.

McKenney, P. E. (2003), 'Using RCU in the Linux 2.5 kernel', *Linux Journal* **1**(114), 18–26.

<http://www.linuxjournal.com/article/6993>.

McKenney, P. E. (2004), Exploiting Deferred Destruction:

An Analysis of Read-Copy-Update Techniques in

NICTA Copyright © 2011

From Imagination to Impact

60

REFERENCES



NICTA

Operating System Kernels, PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University.

<http://www.rdrop.com/users/paulmck/RCU/RCU>

McKenney, P. E. (2010), 'Memory barriers: A hardware view for software hackers',

<http://www.rdrop.com/users/paulmck/scalab:>

McKenney, P. E., Sarma, D., Arcangelli, A., Kleen, A., Krieger, O. & Russell, R. (2002), Read copy update, *in* 'Ottawa Linux Symp.'

<http://www.rdrop.com/users/paulmck/rclock>,

NICTA Copyright © 2011

From Imagination to Impact

61

Schimmel, C. (1994), *UNIX Systems for Modern Architectures*, Addison-Wesley.

Segall, Z. & Rudolph, L. (1984), Dynamic decentralized cache schemes for an MIMD parallel processor, *in* 'Proc. 11th Annual International Symposium on Computer Architecture', pp. 340–347.