# Locks

## To read more...

This day's papers:

Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors"
Guiroux and Lachaize, "Multicore Locks: The Case Is Not Closed Yet"

Supplementary readings:

Hennessy and Patterson, section 5.5
Mellor-Crummey and Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors"
Chabbi et al, "High performance Locks for Multi-level NUMA Systems"

## homework 2, part C

out-of-order processors really like overlapping operations with cache accesses

e.g. arithmetic while cache miss pending

question 3: about how much does this matter here?

or: how much overlap is there?

baseline 1: how much time does the cache take?

baseline 2: how much time does program take without cache misses?

## some confusion from paper reviews

Guirous and Lachaize benchmarked application throughput

macrobenchmark — application performance

e.g. runtime? database transactions per second?

not clear whether they varied this between applications

## weaknesses/discussion topics from paper reviews

Anderson:
- hardware solutions not evaluated — do they help?
- (how) does this effect real applications?
- what about relinguishing the CPU?
- what about thread priorities?

Guirous and Lachaize:
- so much data, so few useful conclusions
- how should we actually choose??
- what about the locks implementation matters??
- is the interposition actually cheap??

## Anderson hardware improvements

broadcast read — avoid double invalidations

lock bus to read for test-and-set, don't just invalidate

## lock-free stack (1)

```
class StackNode { StackNode *next; int value; };
StackNode *head;

void Push(int newValue)  {
    StackNode* newItem = new QueueNode;
    newItem->value = newValue;
    do {
        newItem->next = head;
        MEMORY_FENCE(); // ???
    } while (!compare-and-swap(&head, newItem->next, newItem));
}
```

## lock-free stack (2)

```
class StackNode { StackNode *next; int value; };
StackNode *head;

int Pop() {
    StackNode* removed;
    do {
        removed = head;
        MEMORY_FENCE(); // ???
    } while (!compare-and-swap(&head, removed, removed->next));
    /* missing: deallocating removed safely */
    return removed->value;
}
```

## wait-freedom

if you stop all other threads, one thread can always make progress

not true with locks — no progress if thread holding lock is stopped

good for latency?

## Linux lock-freedom

read-copy-update: Linux kernel pattern

no locking for reads

writing is slow — used for read-mostly data

complicated handling of deallocation

## spinlocks

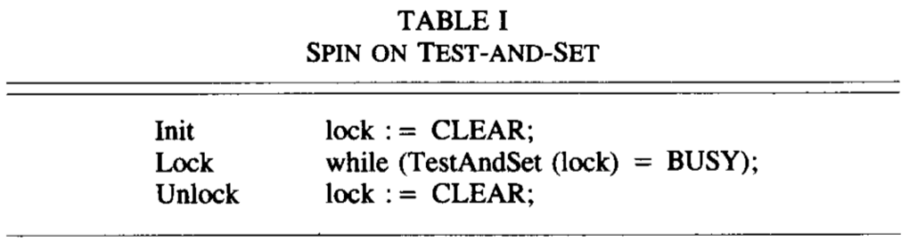
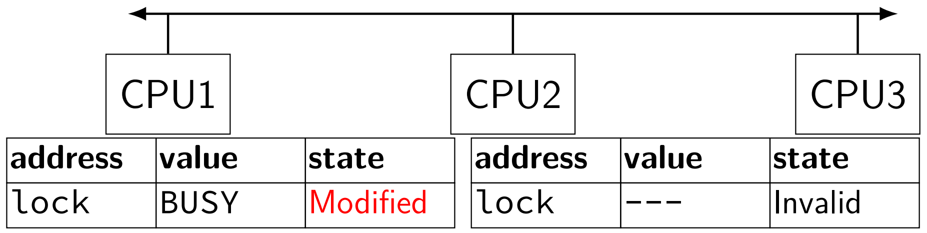


TABLE I
SPIN ON TEST-AND-SET

| | |
|---|---|
| Init | lock := CLEAR; |
| Lock | while (TestAndSet (lock) = BUSY); |
| Unlock | lock := CLEAR; |

## test and set: two CPUs

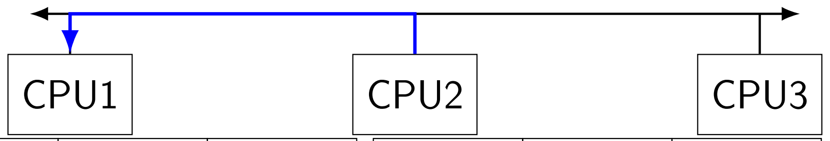


| address | value | state |
|---|---|---|
| lock | BUSY | Modified |

| address | value | state |
|---|---|---|
| lock | --- | Invalid |

## test and set: two CPUs

CPU2: read-to-own lock (TestAndSet)

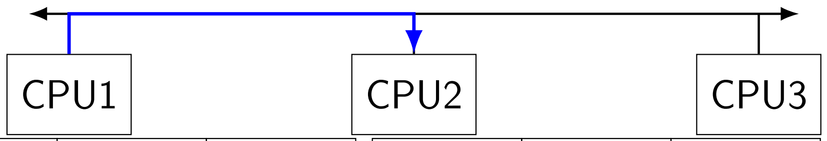


| address | value | state |
|---|---|---|
| lock | ~~BUSY~~ | Invalid |

| address | value | state |
|---|---|---|
| lock | BUSY | Exclusive |

## test and set: two CPUs

CPU1: read-to-own lock (lock = CLEAR)

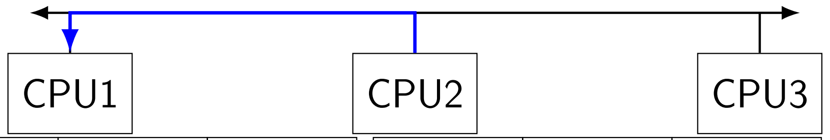


| address | value | state |
|---|---|---|
| lock | CLEAR | Modified |

| address | value | state |
|---|---|---|
| lock | ~~BUSY~~ | Invalid |

## test and set: two CPUs

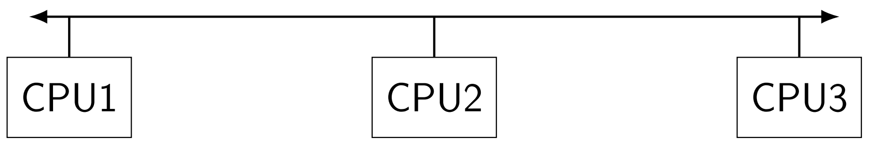
CPU2: read-to-own lock (TestAndSet)



| address | value | state |
|---|---|---|
| lock | ~~CLEAR~~ | Invalid |

| address | value | state |
|---|---|---|
| lock |  | Modified |

## test and set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Modified |

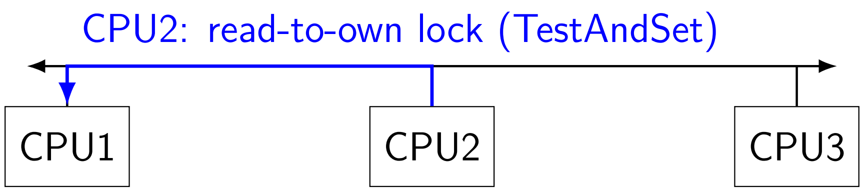
CPU3 cache

| address | value | state |
|---|---|---|
| lock | --- | Invalid |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | --- | Invalid |

## test and set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | ~~BUSY~~ | Invalid |

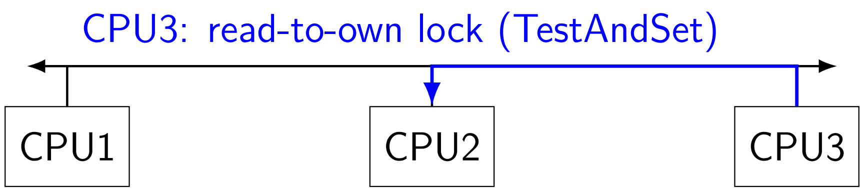
CPU3 cache

| address | value | state |
|---|---|---|
| lock | --- | Invalid |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Exclusive |

## test and set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock |  | Invalid |

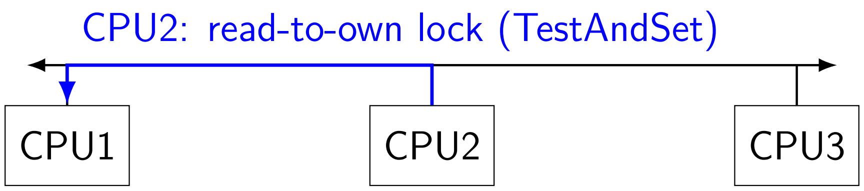
CPU3 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Exclusive |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | ~~BUSY~~ | Invalid |

## test and set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock |  | Invalid |

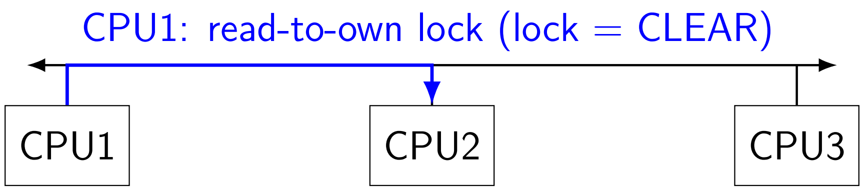
CPU3 cache

| address | value | state |
|---|---|---|
| lock | ~~BUSY~~ | Invalid |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Exclusive |

## test and set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | CLEAR | Modified |

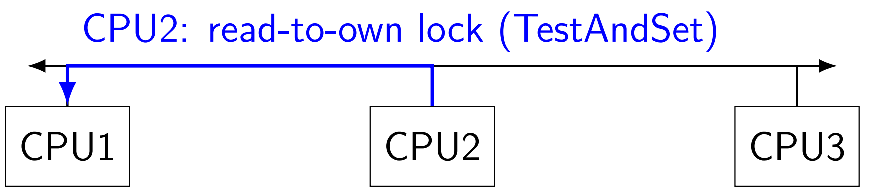
CPU3 cache

| address | value | state |
|---|---|---|
| lock | ~~BUSY~~ | Invalid |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | ~~BUSY~~ | Invalid |

## test and set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | ~~CLEAR~~ | |

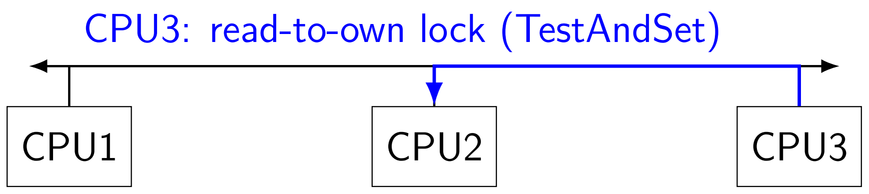
CPU3 cache

| address | value | state |
|---|---|---|
| lock | ~~BUSY~~ | Invalid |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Modified |

## test and set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | ~~CLEAR~~ | |

CPU3 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Exclusive |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | | Invalid |

## test and set: contention costs

1 processors waiting: 3 invalidations

2 processors waiting: 7 to $\infty$ invalidations

- 2 to $\infty$ invalidations while BUSY (first time)
- 1 invalidation to set to CLEAR (first time)
- 1 invalidation to acquire lock (first waiter)
- 1 invalidation while BUSY (second time)
- 1 invalidation to set to CLEAR (second time)
- 1 invalidation to acquire lock (second waiter)

## test and test-and-set

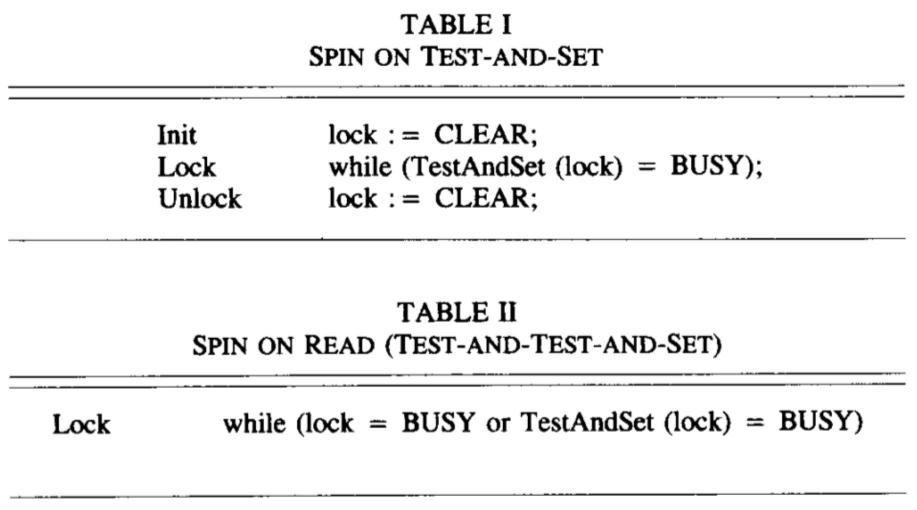


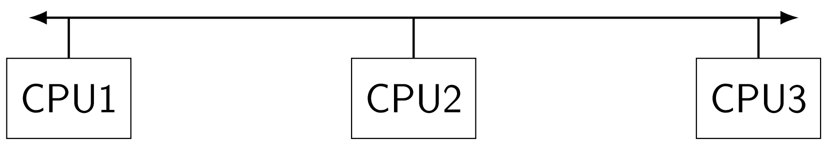
TABLE I
SPIN ON TEST-AND-SET

| | |
|---|---|
| Init | lock := CLEAR; |
| Lock | while (TestAndSet (lock) = BUSY); |
| Unlock | lock := CLEAR; |

TABLE II
SPIN ON READ (TEST-AND-TEST-AND-SET)

| | |
|---|---|
| Lock | while (lock = BUSY or TestAndSet (lock) = BUSY) |

## test and test-and-set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Modified |

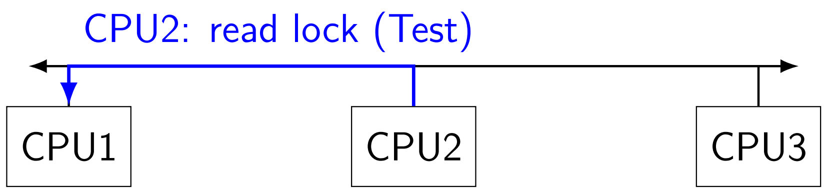
CPU3 cache

| address | value | state |
|---|---|---|
| lock | --- | Invalid |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | --- | Invalid |

## test and test-and-set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Owned |

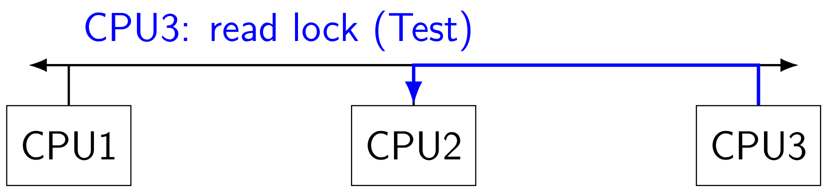
CPU3 cache

| address | value | state |
|---|---|---|
| lock | --- | Invalid |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Shared |

## test and test-and-set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Owned |

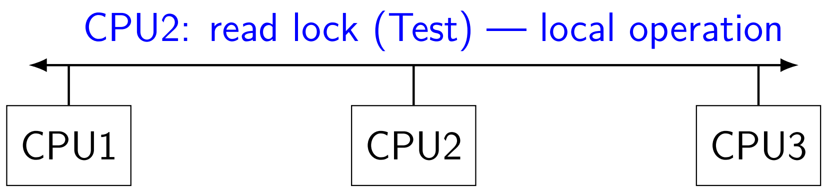
CPU3 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Shared |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Shared |

## test and test-and-set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Owned |

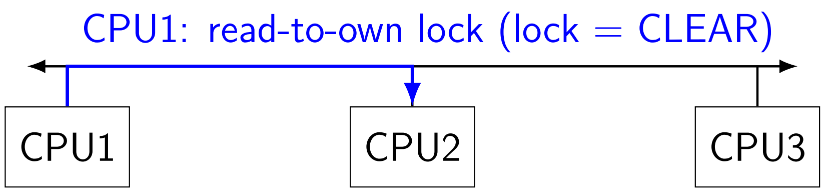
CPU3 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Shared |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Shared |

## test and test-and-set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | CLEAR | Modified |

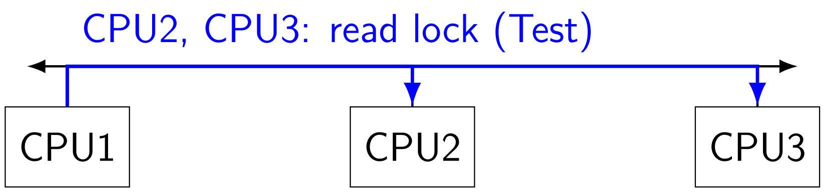
CPU3 cache

| address | value | state |
|---|---|---|
| lock | ~~BUSY~~ | Invalid |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Invalid |

## test and test-and-set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | CLEAR | Owned |

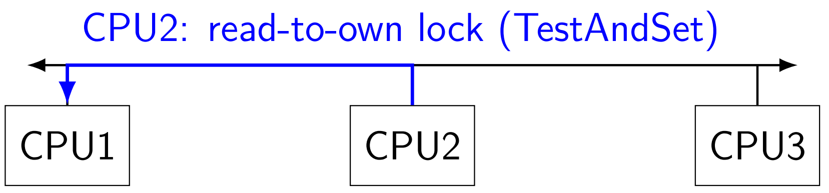
CPU3 cache

| address | value | state |
|---|---|---|
| lock | CLEAR | Shared |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Shared |

## test and test-and-set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | --- | Invalid |

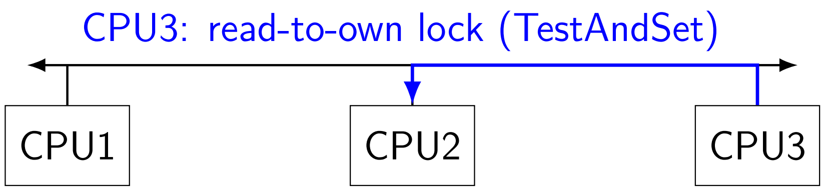
CPU3 cache

| address | value | state |
|---|---|---|
| lock | ~~CLEAR~~ | Invalid |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Modified |

## test and test-and-set: three CPUs



CPU1 cache

| address | value | state |
|---|---|---|
| lock | | |

CPU3 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Exclusive |

CPU2 cache

| address | value | state |
|---|---|---|
| lock | BUSY | Invalid |

## test-and-test-and-set: contention costs

1 processors waiting: 2 invalidations

- 1 invalidation to release lock (original holder)
- 1 invalidation to acquire lock

2 processors waiting: 5 invalidations

- 1 invalidation to release lock (original holder)
- 1 invalidation to acquire lock (first waiter)
- 1 invalidation to read BUSY (failed test-and-set)
- 1 invalidation to release lock (first waiter)
- 1 invalidation to acquire lock (second waiter)

## test-and-test-and-set: contention costs

1 processors waiting: 2 invalidations

- 1 invalidation to release lock (original holder)
- 1 invalidation to acquire lock

2 processors waiting: 5 invalidations

- 1 invalidation to release lock (original holder)
- 1 invalidation to acquire lock (first waiter)
- 1 invalidation to read BUSY (failed test-and-set)
- 1 invalidation to release lock (first waiter)
- 1 invalidation to acquire lock (second waiter)

## adding delay

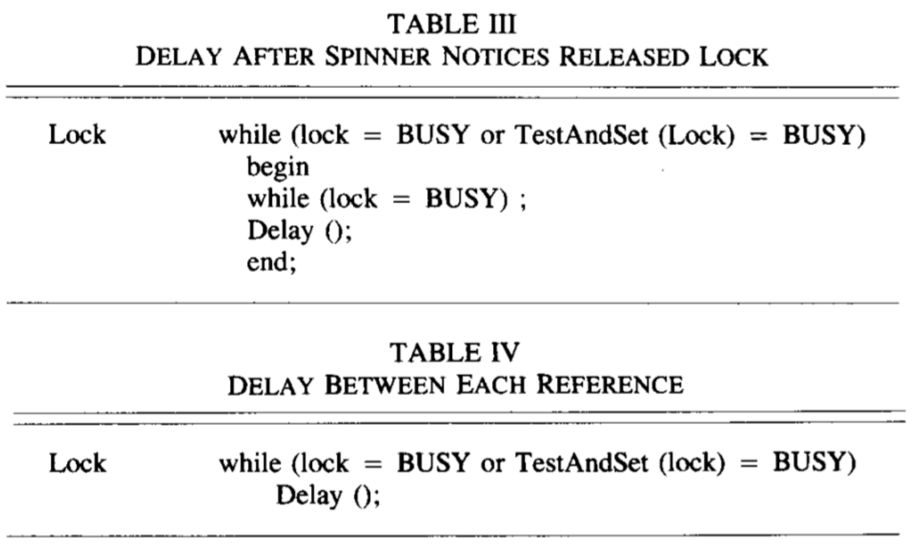


TABLE III
DELAY AFTER SPINNER NOTICES RELEASED LOCK

```
Lock        while (lock = BUSY or TestAndSet (Lock) = BUSY)
              begin
              while (lock = BUSY) ;
              Delay ();
              end;
```

TABLE IV
DELAY BETWEEN EACH REFERENCE

```
Lock        while (lock = BUSY or TestAndSet (lock) = BUSY)
                Delay ();
```

## test-and-test-and-set + delay

2 processors waiting: 4 invalidations

- 1 invalidation to release lock (original holder)
- 1 invalidation to acquire lock (whoever delayed least)
- 1 invalidation to release lock (whoever delayed least)
- 1 invalidation to acquire lock (whoever delayed most)

## choosing how much to delay

slot for each processor

dynamic — based on frequency of conflict

very related to networking work (shared bus networks — e.g. wireless)

## ticket-based lock

Init
```
nextTicket := 0
currentlyServing := 0
```
Lock
```
myTicket := ReadAndIncrement(nextTicket
while (myTicket != currentlyServing)
    ;
```
Unlock
```
currentlyServing := currentlyServing +
```

## ticket lock: invalidations

2 processors waiting: 5 invalidations + 5 transfers

- 2 invalidations to choose ticket number
- 2(?) transfers to read currentlyServing
- 1 invalidation to release lock (original holder)
- 2(?) transfers to read currentlyServing
- 1 invalidation to release lock (first waiter)
- 1 transfer to read currentlyServing
- 1 invalidation to release lock (second waiter)

## lock fairness

variation in time to wait?

## queue-based lock

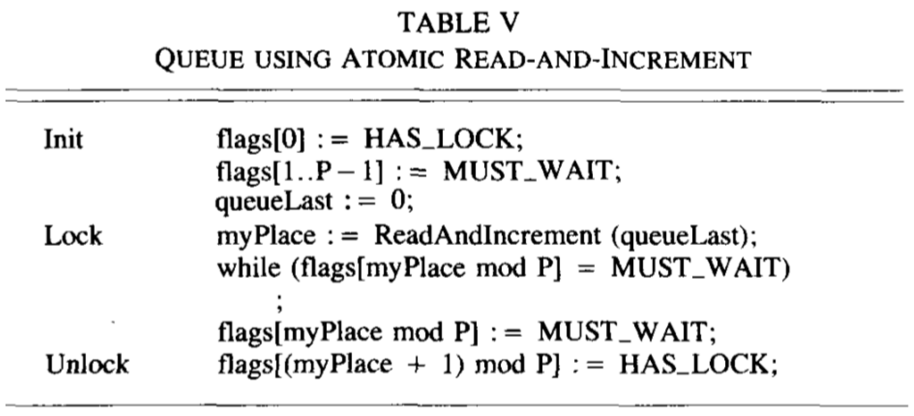


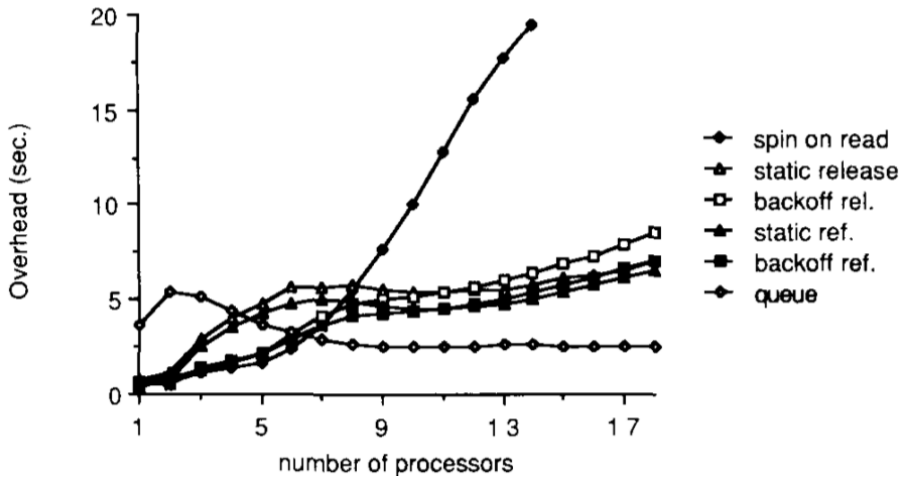
TABLE V
QUEUE USING ATOMIC READ-AND-INCREMENT

```
Init      flags[0] := HAS_LOCK;
          flags[1..P-1] := MUST_WAIT;
          queueLast := 0;
Lock      myPlace := ReadAndIncrement (queueLast);
          while (flags[myPlace mod P] = MUST_WAIT)
              ;
          flags[myPlace mod P] := MUST_WAIT;
Unlock    flags[(myPlace + 1) mod P] := HAS_LOCK;
```

## queue-based lock

2 processors waiting: 4 invalidations + 2 transfers
- 2 invalidation to choose queue location
- 1 invalidation to change flag (of first waiter)
- 1 transfer to read changed flag (first waiter)
- 1 invalidation to change flag (of second waiter)
- 1 transfer to read changed flag (second waiter)

## microbenchmark results



## locks and the OS

often want to run something else instead of waiting
- more than one thread per core

Linux mechanism: futex WAIT/WAKE:
- WAIT — go to sleep if value in memory unchanged
- WAKE — wake up anyone waiting on value

## lock fairness

test-and-set lock
three threads try to acquire, release in a loop
how many times does each thread actually get lock?

times acquired lock (100M trials)

| | |
|---|---|
| Thread 1 | 29964090 (30%) |
| Thread 2 | 28054597 (28%) |
| Thread 3 | 41981317 (42%) |

## locks and NUMA (1)

better to spin on local values

MCS lock: like queue lock, but linked list instead of array

processor's linked list node allocated from local memory

## locks and NUMA (2)

communicate locally first if multiple waiters

one lock per node (e.g. processors sharing a directory in DASH)

acquire local lock before contending for global lock

## delegation

run a server thread for critical operations

lock-free queue: sending functions to run to the server

get locality for critical section

## lock tradeoffs

uncontended performance
- worse for queue/ticket-based locks

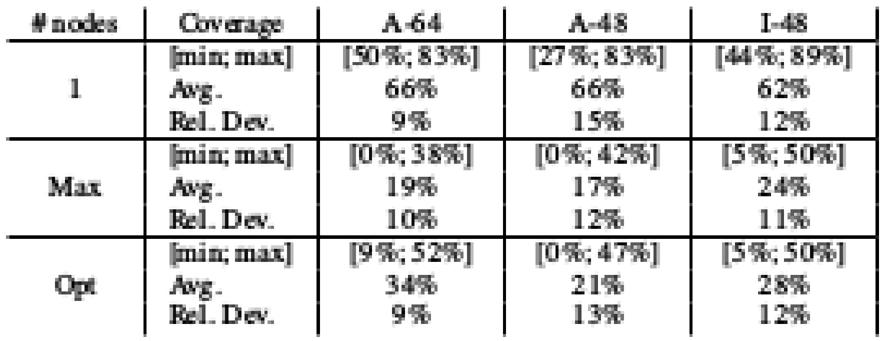
memory traffic in contention
- especially from bogus invalidations
- better for ticket/queue-based locks

excess delay (for backoff strategies)
- waiting for no one?

fairness
- better for ticket/queue-based locks

releases CPU — if more than one thread/core

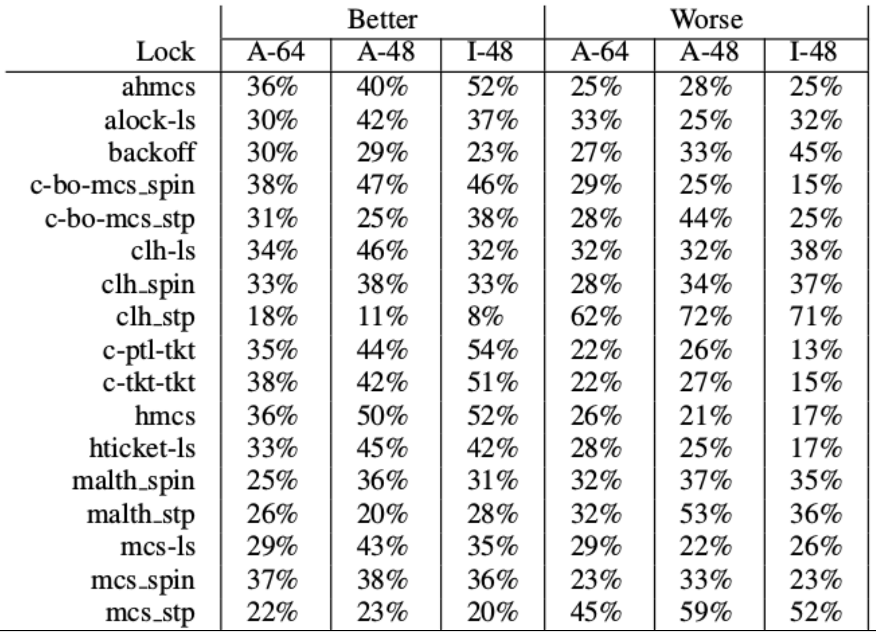
## locks 5% of best %



| # nodes | Coverage | A-64 | A-48 | I-48 |
|---|---|---|---|---|
| 1 | [min; max] | [50%; 83%] | [27%; 83%] | [44%; 89%] |
| | Avg. | 66% | 66% | 62% |
| | Rel. Dev. | 9% | 15% | 12% |
| Max | [min; max] | [0%; 38%] | [0%; 42%] | [5%; 50%] |
| | Avg. | 19% | 17% | 24% |
| | Rel. Dev. | 10% | 12% | 11% |
| Opt | [min; max] | [9%; 52%] | [0%; 47%] | [5%; 50%] |
| | Avg. | 34% | 21% | 28% |
| | Rel. Dev. | 9% | 13% | 12% |

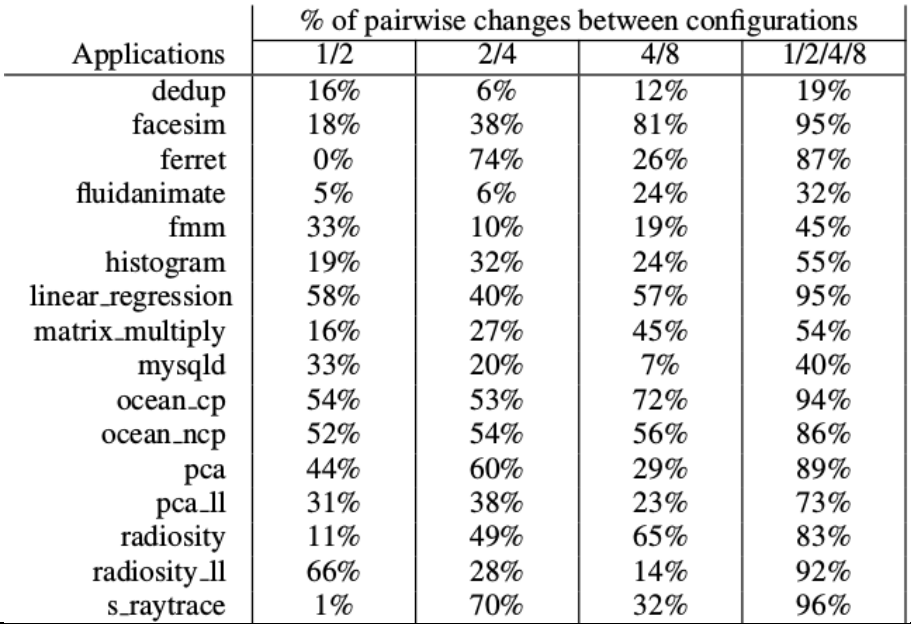
Table 8: Statistics on the coverage of locks for three configurations: 1 node, max nodes, and opt nodes (**all machines**).

## better/worse



| Lock | Better A-64 | Better A-48 | Better I-48 | Worse A-64 | Worse A-48 | Worse I-48 |
|---|---|---|---|---|---|---|
| ahmcs | 36% | 40% | 52% | 25% | 28% | 25% |
| alock-ls | 30% | 42% | 37% | 33% | 25% | 32% |
| backoff | 30% | 29% | 23% | 27% | 33% | 45% |
| c-bo-mcs_spin | 38% | 47% | 46% | 29% | 25% | 15% |
| c-bo-mcs_stp | 31% | 25% | 38% | 28% | 44% | 25% |
| clh-ls | 34% | 46% | 32% | 32% | 32% | 38% |
| clh_spin | 33% | 38% | 33% | 28% | 34% | 37% |
| clh_stp | 18% | 11% | 8% | 62% | 72% | 71% |
| c-ptl-tkt | 35% | 44% | 54% | 22% | 26% | 13% |
| c-tkt-tkt | 38% | 42% | 51% | 22% | 27% | 15% |
| hmcs | 36% | 50% | 52% | 26% | 21% | 17% |
| hticket-ls | 33% | 45% | 42% | 28% | 25% | 17% |
| malth_spin | 25% | 36% | 31% | 32% | 37% | 35% |
| malth_stp | 26% | 20% | 28% | 32% | 53% | 36% |
| mcs-ls | 29% | 43% | 35% | 29% | 22% | 26% |
| mcs_spin | 37% | 38% | 36% | 23% | 33% | 23% |
| mcs_stp | 22% | 23% | 20% | 45% | 59% | 52% |

## depends on scale



| Applications | % of pairwise changes between configurations 1/2 | 2/4 | 4/8 | 1/2/4/8 |
|---|---|---|---|---|
| dedup | 16% | 6% | 12% | 19% |
| facesim | 18% | 38% | 81% | 95% |
| ferret | 0% | 74% | 26% | 87% |
| fluidanimate | 5% | 6% | 24% | 32% |
| fmm | 33% | 10% | 19% | 45% |
| histogram | 19% | 32% | 24% | 55% |
| linear_regression | 58% | 40% | 57% | 95% |
| matrix_multiply | 16% | 27% | 45% | 54% |
| mysqld | 33% | 20% | 7% | 40% |
| ocean_cp | 54% | 53% | 72% | 94% |
| ocean_ncp | 52% | 54% | 56% | 86% |
| pca | 44% | 60% | 29% | 89% |
| pca_ll | 31% | 38% | 23% | 73% |
| radiosity | 11% | 49% | 65% | 83% |
| radiosity_ll | 66% | 28% | 14% | 92% |
| s_raytrace | 1% | 70% | 32% | 96% |

## next time: transactional memory

transaction user model

```
do_transaction {
    // manipulate shared values here
}
```

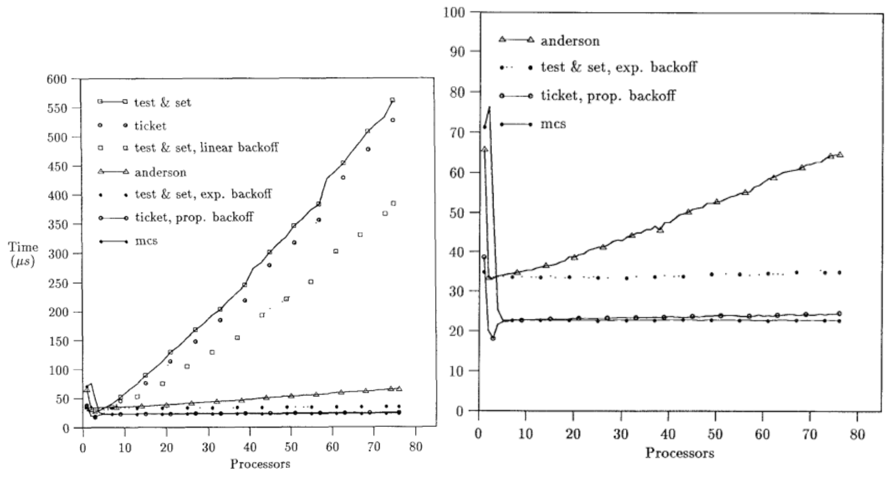
changes happen all at once or not at all

- commit or abort

implementation trick: try, detect conflicts, repeat

implemented in software or hardware

## microbenchmarks — clear hierarchy?



## next time: papers

Herlihy and Moss:

- primitives to implement user model
- code for retry needs to be generated by compiler
- takes advantage of cache states — like we did for consistency

McKenney et al:

- a critique of transactional memory as a programming model