SMP & Locking

THE UNIVERSITY OF
NEW SOUTH WALES

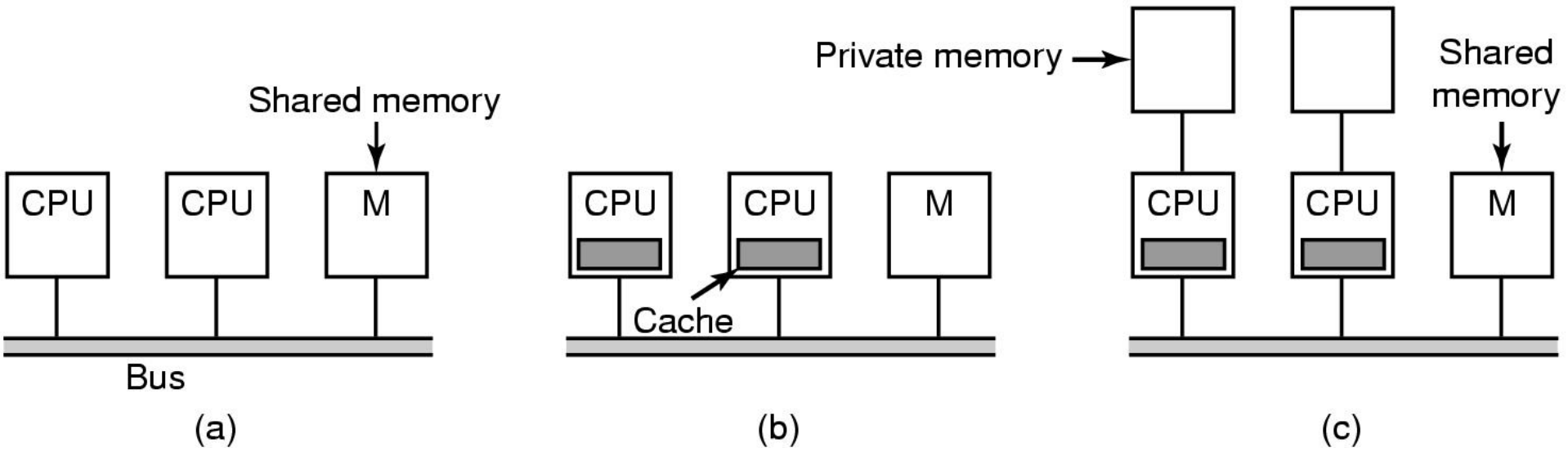
Types of Multiprocessors (MPs)

- UMA MP
  - Uniform Memory Access
    - Access to all memory occurs at the same speed for all processors.
- NUMA MP
  - Non-uniform memory access
    - Access to some parts of memory is faster for some processors than other parts of memory
- We will focus on UMA

Bus Based UMA

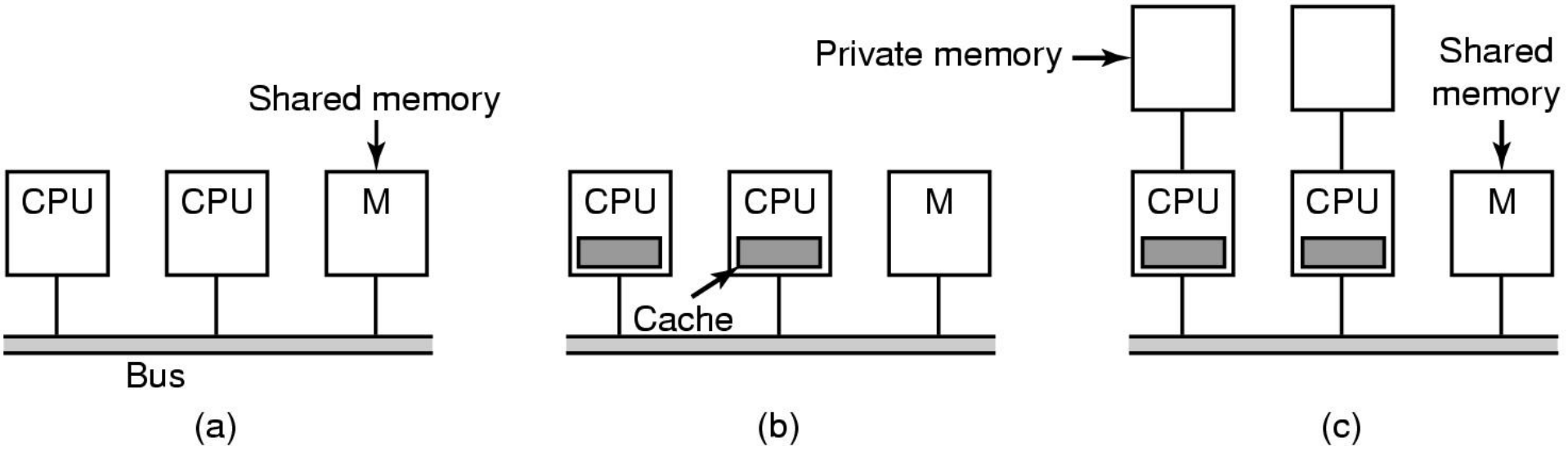
Simplest MP is more than one processor on a single bus connect to memory (a)

- Bus bandwidth becomes a bottleneck with more than just a few CPUs



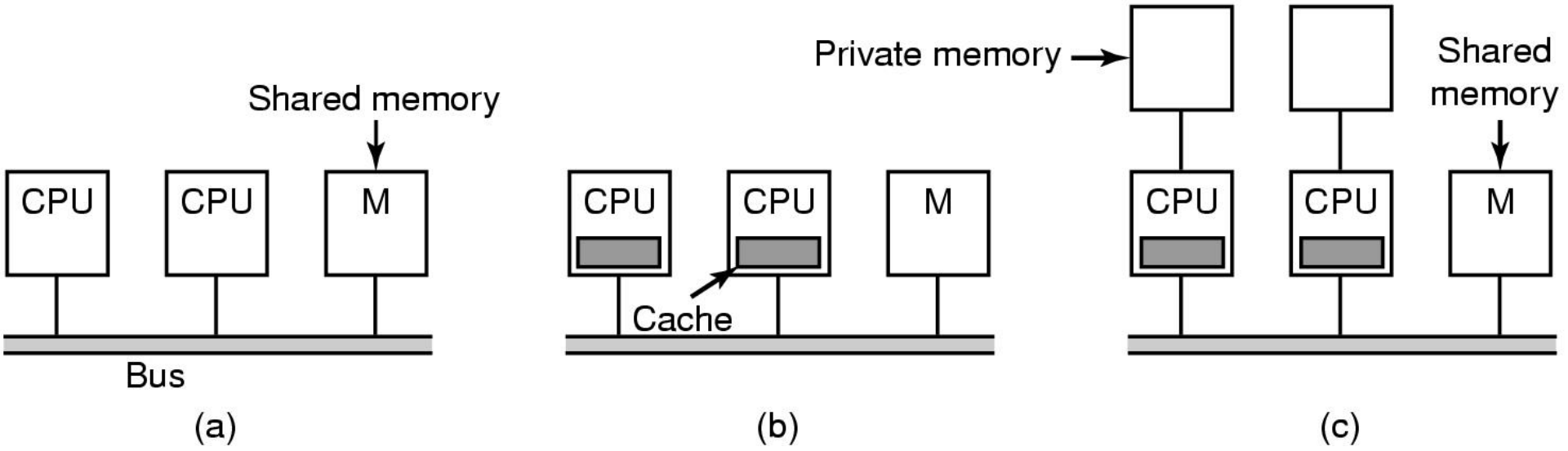
Bus Based UMA

- Each processor has a cache to reduce its need for access to memory (b)
  - Hope is most accesses are to the local cache
  - Bus bandwidth still becomes a bottleneck with many CPUs



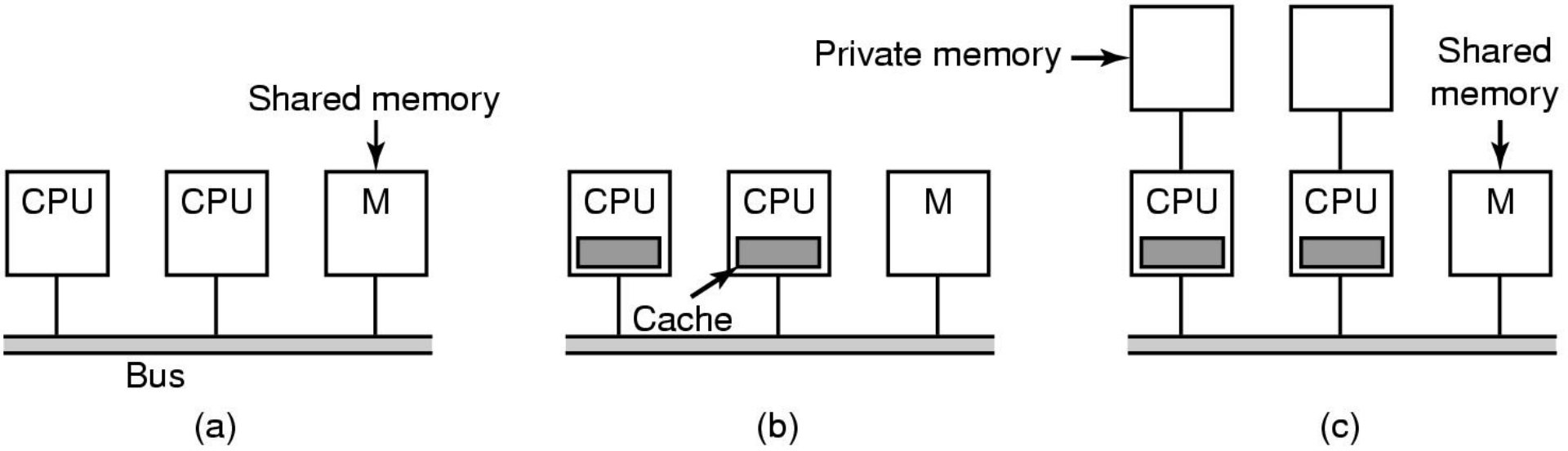
Cache Consistency

- 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)?



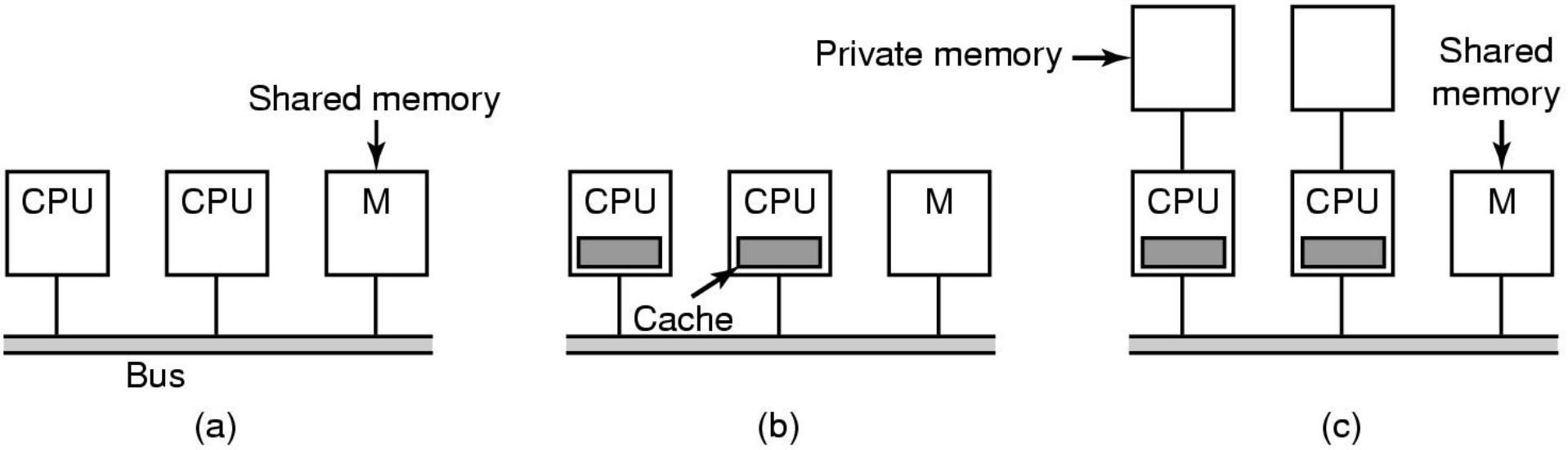
Cache Consistency

- Cache consistency is usually handled by the hardware.
  - Writes to one cache propagate to, or invalidate appropriate entries on other caches
  - Cache transactions also consume bus bandwidth



Bus Based UMA

- To further scale the number processors, we give each processor private local memory
  - Keep private data local on off the shared memory bus
  - Bus bandwidth still becomes a bottleneck with many CPUs with shared data
  - Complicate application development
    - We have to partition between private and shared variables



Focus on locking in the Common Case

- Bus-based UMA, per-CPU caches, no private memory, SMP.

Kernel Locking

- Several CPUs can be executing kernel code concurrently.
- Need mutual exclusion on shared kernel data.
- Issues:
  - Lock implementation
  - Granularity of locking

Mutual Exclusion Techniques

- Disabling interrupts (CLI — STI).
  - Unsuitable for multiprocessor systems.
- Spin locks.
  - Busy-waiting wastes cycles.
- Lock objects.
  - Flag (or a particular state) indicates object is locked.
  - Manipulating lock requires mutual exclusion.

Hardware Provided Locking Primitives

- `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 locks

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

- Busy waits. Good idea?

Spin Lock Busy-waits Until Lock Is Released

- Stupid on uniprocessors, as nothing will change while spinning.
  - Should release (yield) 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 be contended by the same CPU.
  - local contention can be prevented
    - by design
    - by turning off interrupts

Spinning versus Switching

- Blocking and switching
  - to another process takes time
    - Save context and restore another
    - Cache contains current process not new
      - » Adjusting the cache working set also takes time
    - TLB is similar to cache
  - Switching back when the lock is free encounters the same again
- Spinning wastes CPU time directly

- Trade off
  - If lock is held for less time than the overhead of switching to and back

⇒ It's more efficient to spin

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

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: Conditional Lock

```
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 aquire lock.
- **But** may not have much else to do.
- Starvation: May never get lock!

More Appropriate Mutex Primitive:

```
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.

Multiprocessor Spin Lock

```
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();
}
```

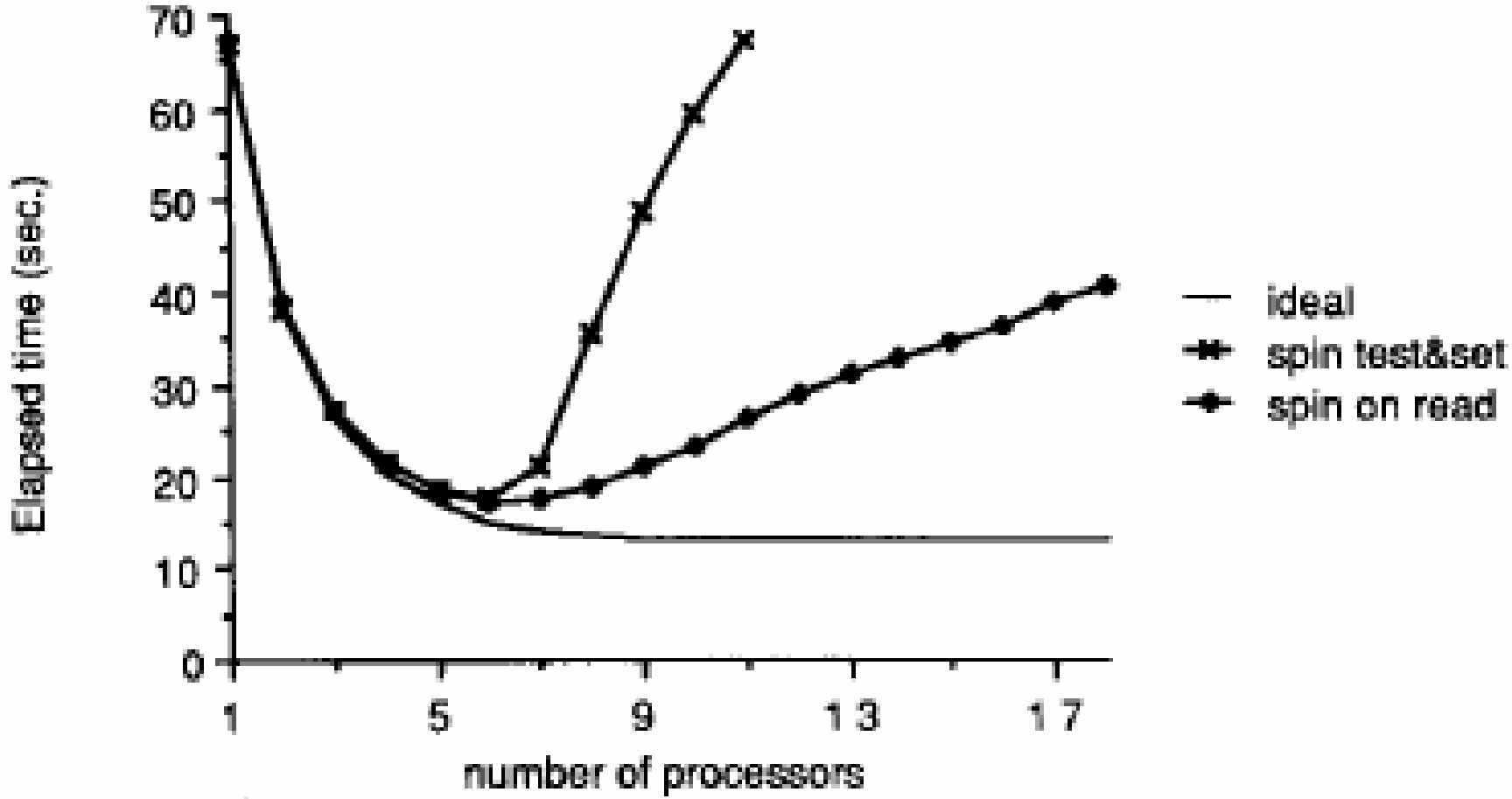
- Only good for short critical sections
- Does not scale for large number of processors
- Relies on bus-arbitrator for fairness
- Not appropriate for user-level
- Used in practice in small SMP systems

Thomas Anderson, “The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors”, *IEEE Transactions on Parallel and Distributed Systems*, Vol 1, No. 1, 1990

test_and_test_and_set LOCK

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

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


Elapsed time (sec.)
70
60
50
40
30
20
10
0
1
5
9
13
17
number of processors
ideal
spin test&set
spin on read

Examining Inserting Delays

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 ();
```

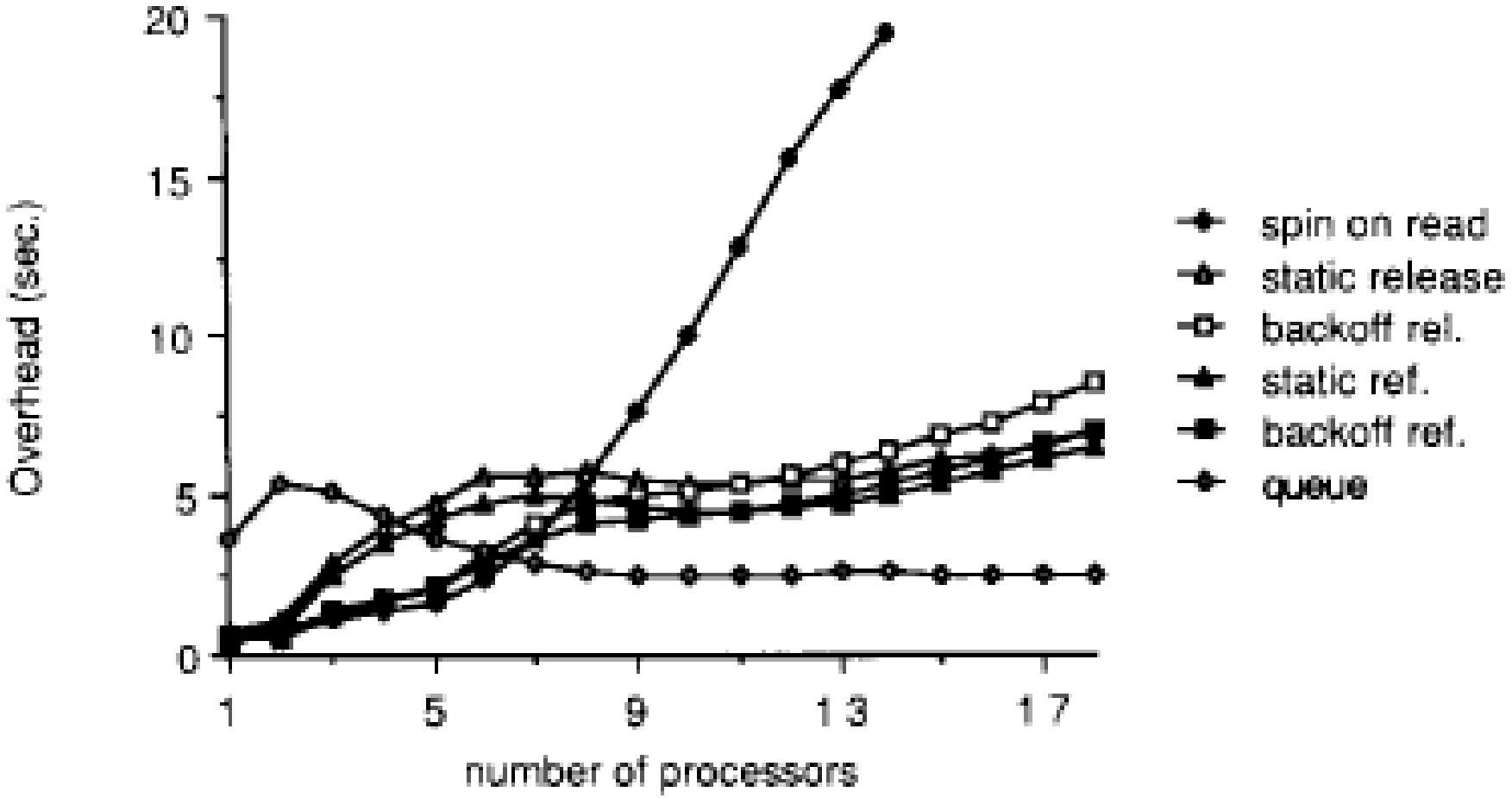
Delays

- Static
  - Tuned to expected processor contention level
- Dynamic
  - Based on exponential backoff scheme similar to ethernet

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


20
15
10
5
0
Overhead (sec.)
1
5
9
13
17
number of processors
spin on read
static release
backoff rel.
static ref.
backoff ref.
queue

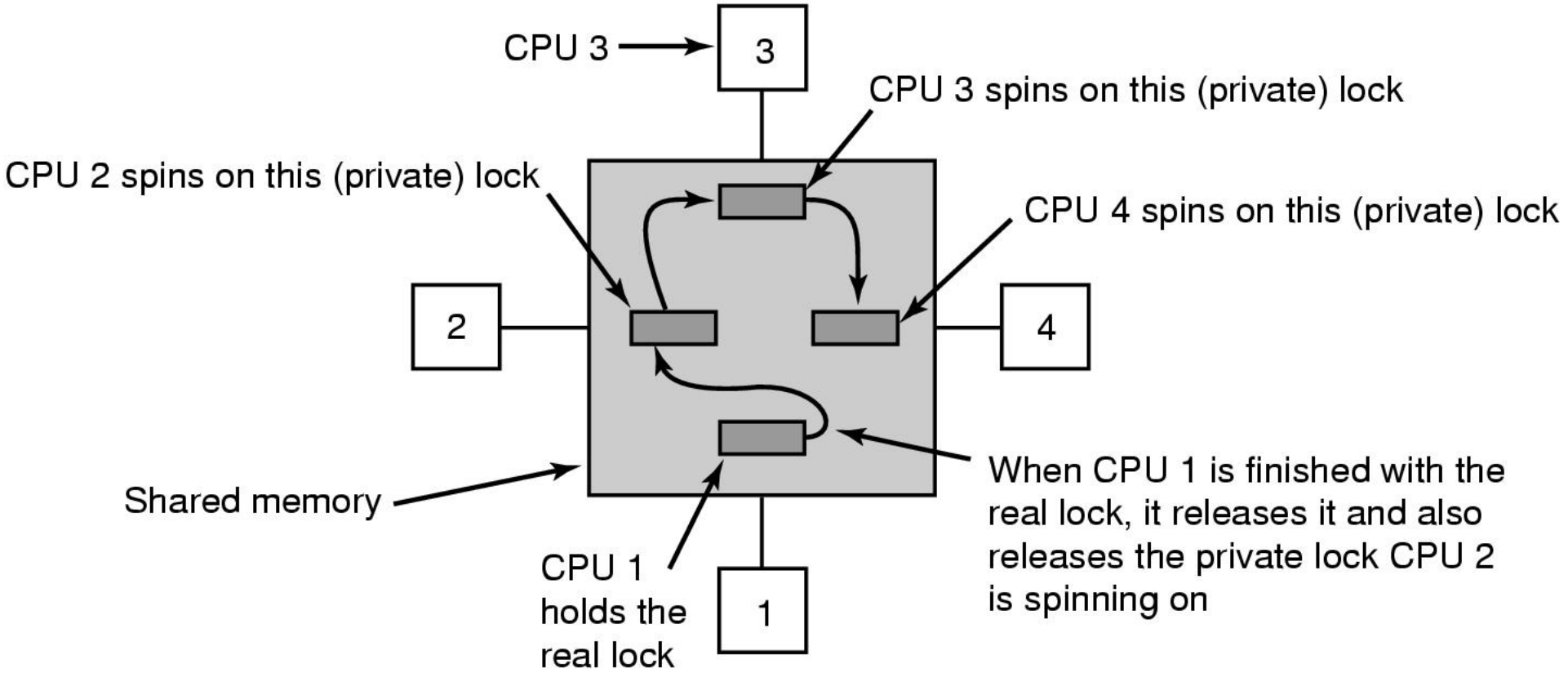
Results

- 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.

- John Mellor-Crummey and Michael Scott, “Algorithms for Scalable Synchronisation on Shared-Memory Multiprocessors”, *ACM Transactions on Computer Systems,* Vol. 9, No. 1, 1991

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 starvation (order of lock acquisitions defined by the list)



MCS Lock

- Requires
  - compare_and_swap()
  - exchange()
    - Also called fetch_and_store()

```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode

// parameter I, below, points to a qnode record allocated
// (in an enclosing scope) in shared memory locally-accessible
// to the invoking processor

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor : ^qnode := fetch_and_store (L, I)
    if predecessor != nil      // queue was non-empty
        I->locked := true
        predecessor->next := I
        repeat while I->locked              // spin

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil          // no known successor
        if compare_and_swap (L, I, nil)
            return
            // compare_and_swap returns true iff it swapped
        repeat while I->next = nil          // spin
    I->next->locked := false
```

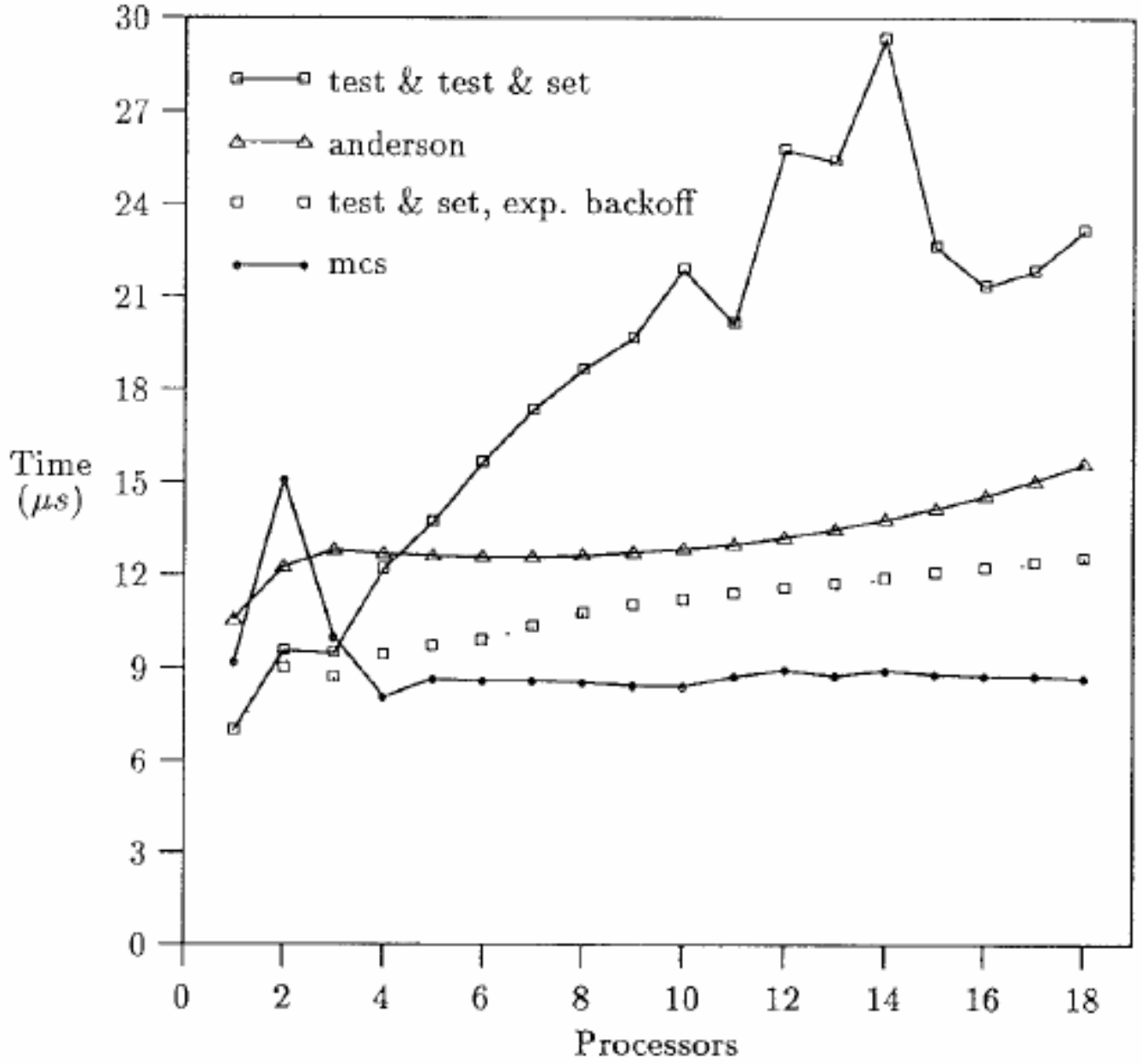
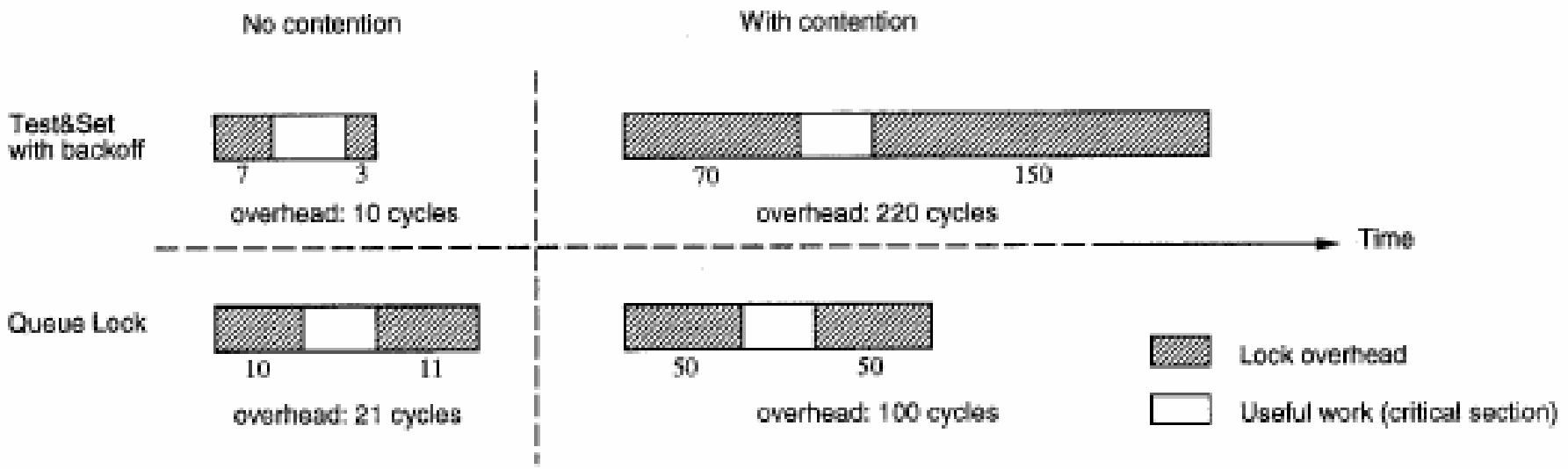


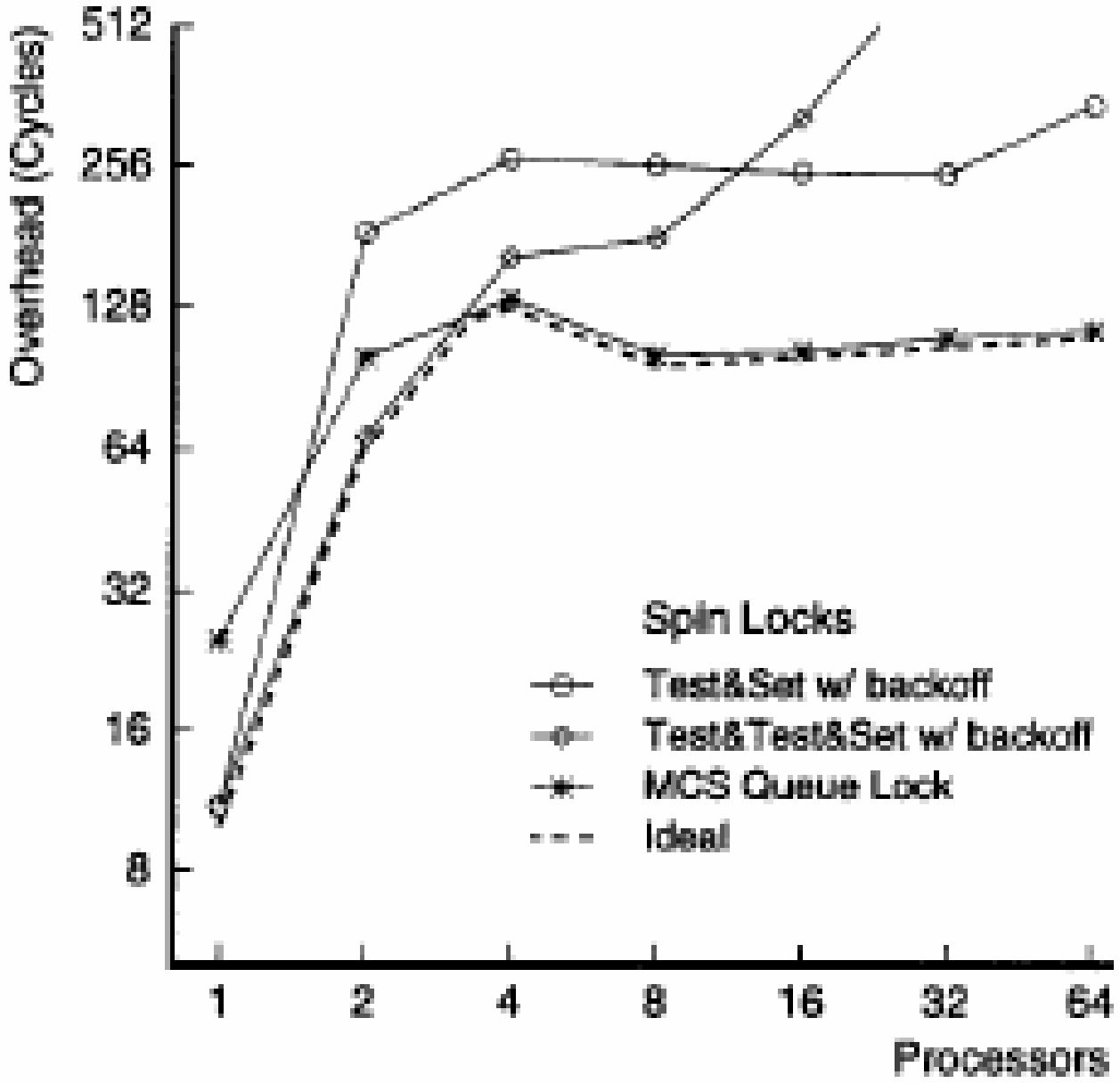
Fig. 17. Performance of spin locks on the Symmetry (empty critical section).

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


No contention
With contention
Test&Set
with backoff
7
3
overhead: 10 cycles
70
150
overhead: 220 cycles
Time
Queue Lock
10
11
overhead: 21 cycles
50
50
overhead: 100 cycles
Lock overhead
Useful work (critical section)


Overhead (Cycles)
512
256
128
64
32
16
8
1
2
4
8
16
32
64
Processors
Spin Locks
Test&Set w/ backoff
Test&Test&Set w/ backoff
MCS Queue Lock
Ideal

Issues

- How do we determine which algorithm to use?
  - Note the race condition
- How do we 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 with result in retry which can get it right next time
  - Assumption: Lock mode changes infrequently

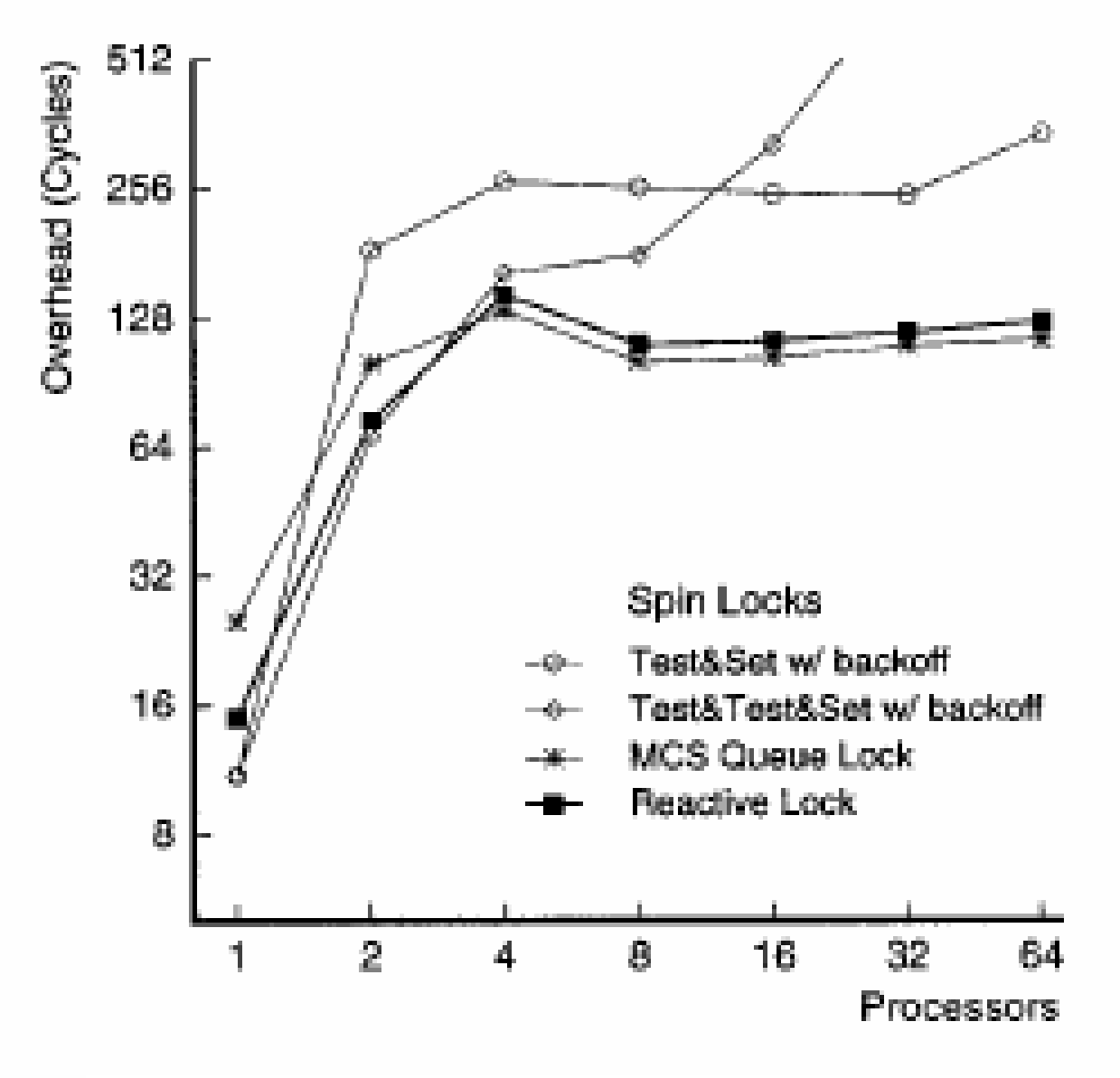
Changing Protocol

- Only lock holder can switch to avoid race conditions

When to change protocol

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

Result


Overhead (Cycles)
512
256
128
64
32
16
8
1
2
4
8
16
32
64
Processors
Spin Locks
Test&Set w/ backoff
Test&Test&Set w/ backoff
MCS Queue Lock
Reactive Lock

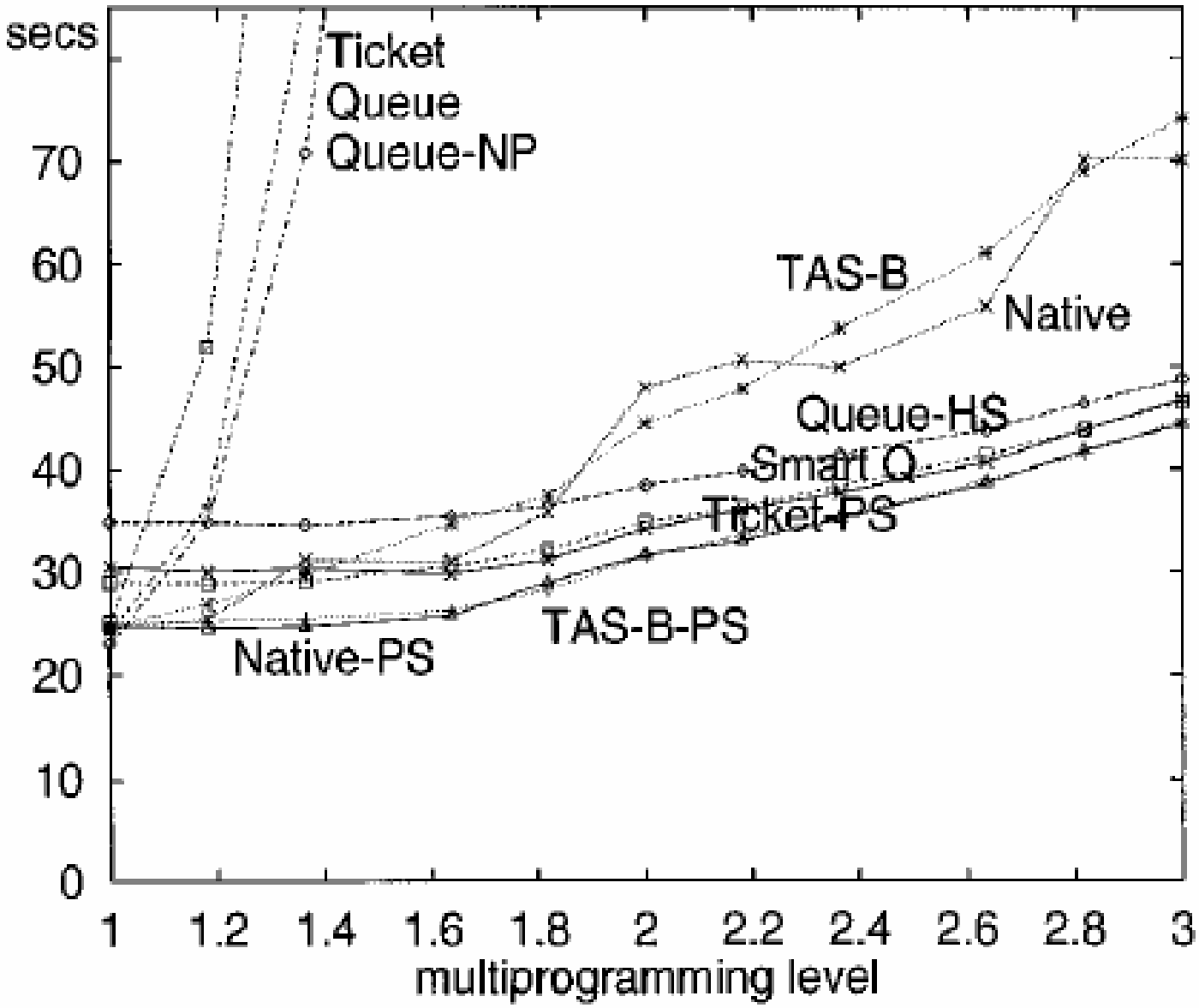
Have we found the perfect locking scheme?

- No!!
- What about preemption of the lock holder?
- For queue-based locking scheme, we switch to the next in queue:
  - What happens if the next in queue is preempted?

Kontothanassis, Wisniewski, and Scott, "Scheduler-Conscious Synchronisation", *ACM Transactions on Computer Systems*, Vol. 15, No. 1, 1997

Idea

- Share state/interface between kernel and lock primitive such that
  - Kernel does not preempt lock holders
  - Lock holder does not switch to preempted thread



NEW SOUTH WALES