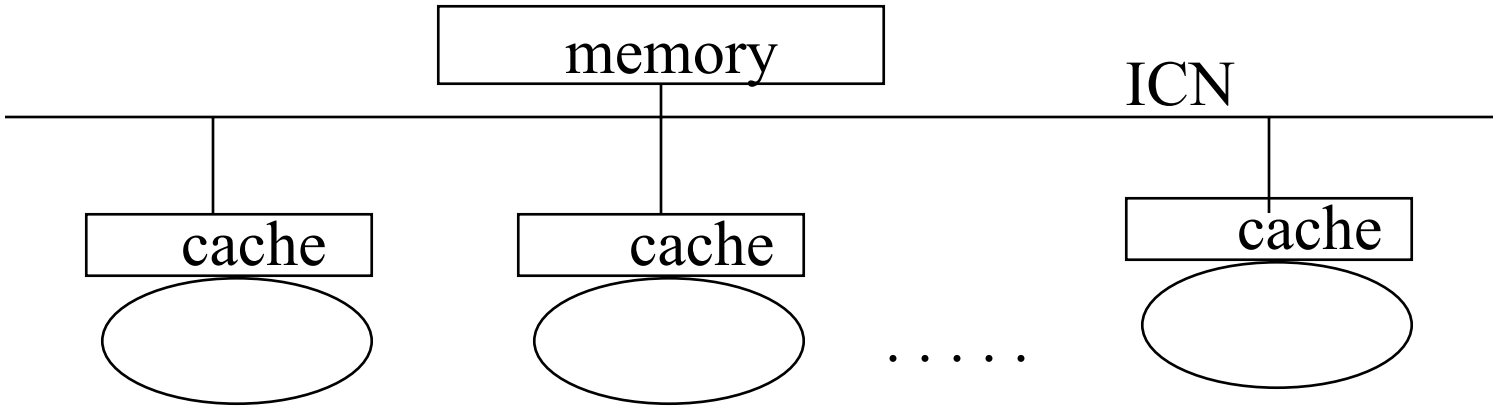
Symmetric Multiprocessors and Performance of Spin Lock Techniques

- Based on Anderson's paper "Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors"
  - interesting paper, good for further thinking
  - a lecture on multiprocessor architectures with spin locks as an application

- Multiprocessor: a machine with many processors where all memory is accessed through the same mechanism
  - there is no distinction between local and non-local memory at the machine level
  - processors, caches, memory and interconnect network or shared bus
  - Shared Memory Symmetric Multiprocessors: all processors are equal and have full access to all resources, main memory is equidistant from all processors

- need to communicate and synchronize:
- message passing MPs
  - communication is explicit, synchronization implicit
- shared address space MPs
  - synchronization explicit, communication implicit
  - cache coherent (CC)
    - KSR-1, Origin 2000
  - non-cache coherent (NCC)
    - BBN Butterfly
    - Cray T3D/T3E

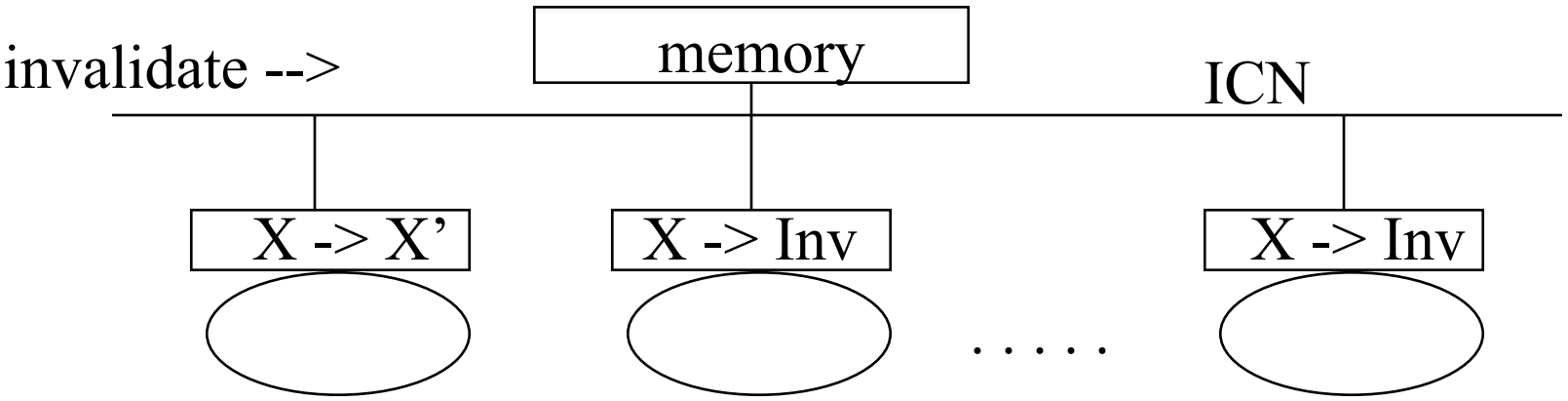
Caches and Consistency



- cache with each processor to hide latency for memory accesses
- miss in cache => go to memory to fetch data
- multiple copies of same address in caches
- caches need to be kept consistent

Cache Coherence

- write-invalidate

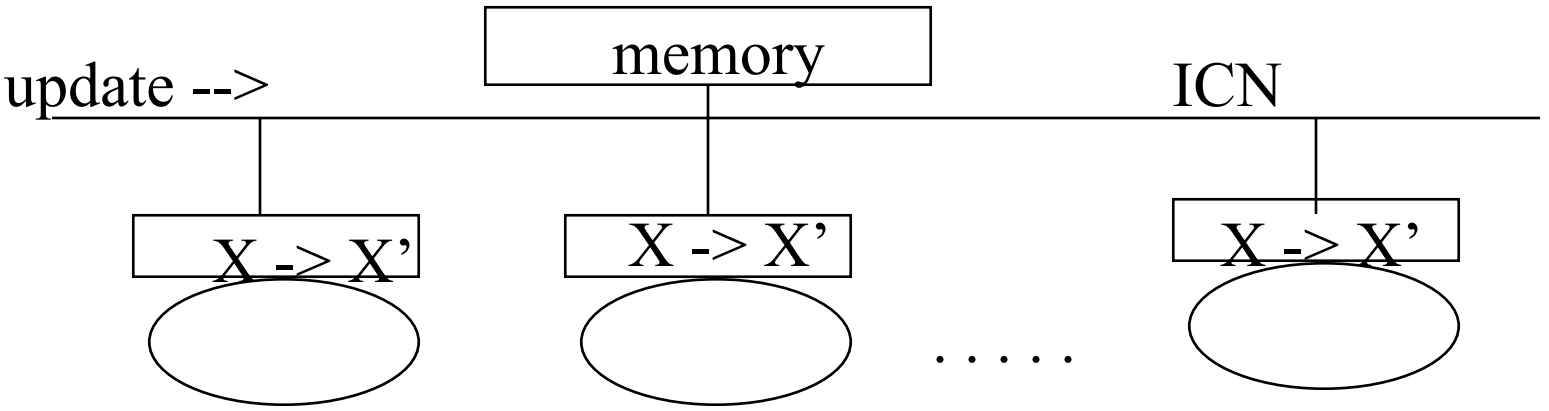


Before a write, an "invalidate" command goes out on the bus. All other processors are "snooping" and they invalidate their caches. Invalid entries cause read faults and new values will be fetched from main memory.

Cache Coherence

As soon as the cache line is written, it is broadcast on the bus, so all processors update their caches.

- write-update (also called distributed write)



write-invalidate vs. write-update

- write-update
  - simple
  - wins if most values are written and read elsewhere
  - eats bandwidth
- write-invalidate
  - complex
  - wins if values are often overwritten before being read (loop index)
  - conserves bandwidth

Paper objectives

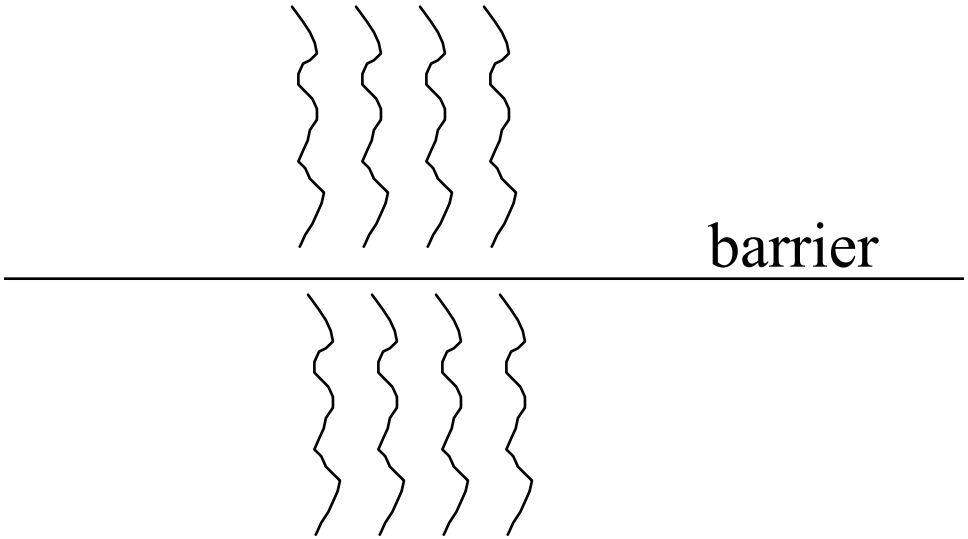
- Anderson's paper discusses some spin lock alternatives
- proposes a new technique based on sequence numbers, i.e. queued lock requests
- analogies are drawn to CSMA networks (carrier-sense multiple access – e.g. Ethernet)

Shared Memory Programming

- synchronization
  - for coordination of the threads
  - atomic op to read-modify-write a mem location
    - test-and-set
    - fetch-and-store(R, <mem>)
    - fetch-and-add(<mem>, <value>)
    - compare-and-swap(<mem>, <cmp-val>, <stor-val>)
- communication
  - for inter-thread sharing of data

Why spin locks?

- efficient for brief critical sections, (e.g. barriers)
  - context switching and blocking is more expensive
- can be done solely in software but expensive
- based on hardware support for atomic operations
  - during an atomic operation, other atomic operations are disabled, and caches are invalidated, even if value unchanged
  - why?



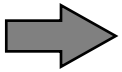
Sync. Alg. Objectives

- reduce latency
  - how quickly can I get the lock in the absence of competition
- reduce waiting time – delaying the application
  - time between lock is freed and another processor acquires it
- reduce contention / bandwidth consumption
  - other processors need to do useful work, process holding lock needs to complete asap…
  - how to design the waiting scheme to reduce interconnection network contention

Any conflicts here??

Locks in Shared Memory MP

- L is a shared memory location



```
lock:
  while (test-and-set(L) == locked);
unlock:
  L = unlocked;
```

- spin or block if lock not available?
  - focus of Anderson's paper is efficient spin-waiting
- spin locks are a polling mechanism. if they poll often, they'll hurt bandwidth, if rarely, they'll hurt delay
  - (note: with snooping, not exactly polling)

Spin locks

- two opposing concerns
  - retry to acquire lock as soon as it is released
  - minimize disruptions to processors doing work

- spin on test-and-set

```
init    lock = clear;
lock    while(t&s(lock) == busy); /* spin */
unlock  lock = clear;
```

spin on test&set performance

- should be terrible:
  - t&s always goes to memory (bypasses caches)
    - does not exploit caches
  - unlocks are queued behind other test&sets
  - in bus-based systems, delay lock holder and other processes
  - in network-based systems create hot spots

- spin on read (test-t&s)

```
lock    while (lock == busy) spin;
        if (t&s(lock) == busy) goto lock;
```

  - why is this better?
    - exploits caching, goes to bus only when lock "looks" free
  - problem?
    - parallel contention on lock release
    - increased network traffic on lock release
      - invalidation -- O(N**2)
        - » some processors will get new L and see it's busy, others will think it's clear and go to t&s => invalidate L again
      - update -- O(N)
        - » all N processors will see L is free and issues t&s
    - normal memory accesses in C.S. affected

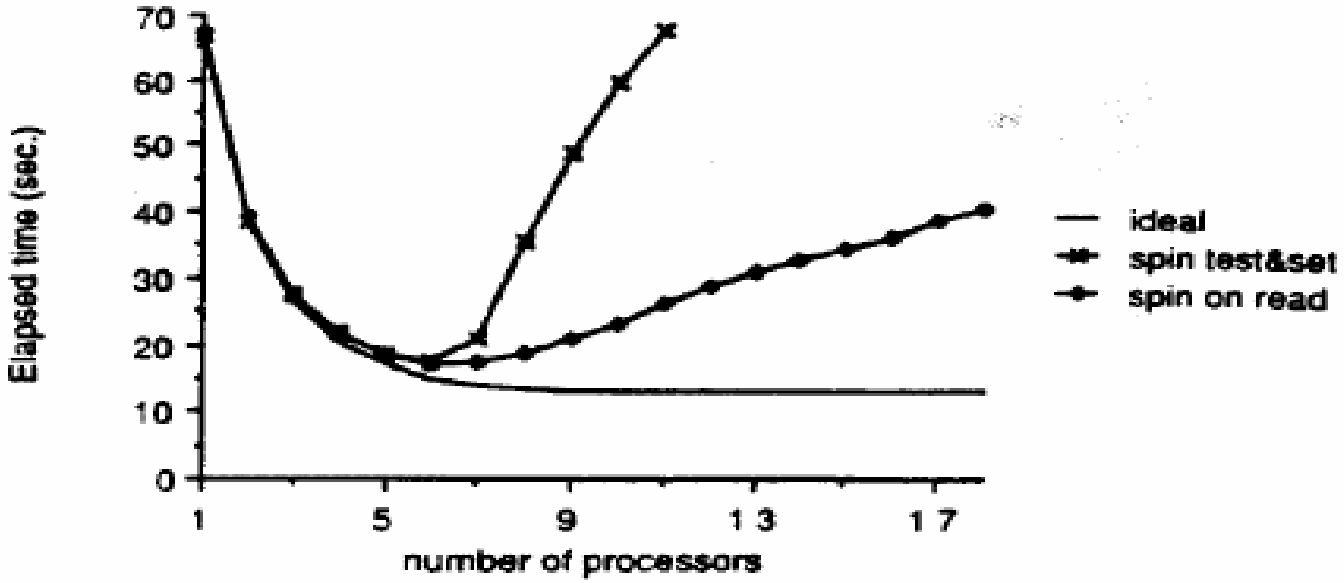


Fig. 1. Principal performance comparison: elapsed time (second) to execute benchmark (measured). Each processor loops one million/P times: acquire lock, do critical section, release lock, and compute.

20 processor Sequent Symmetry CC bus-based WI MP

other issue

- after lock is released, it creates “bandwidth storm”
- Quiescence: time for “b/w storm” to subside
- to measure:
  - acquire lock, idle for time t, generate bus accesses, release lock
  - should take same total time if 1 or many processors waiting on lock, but doesn’t

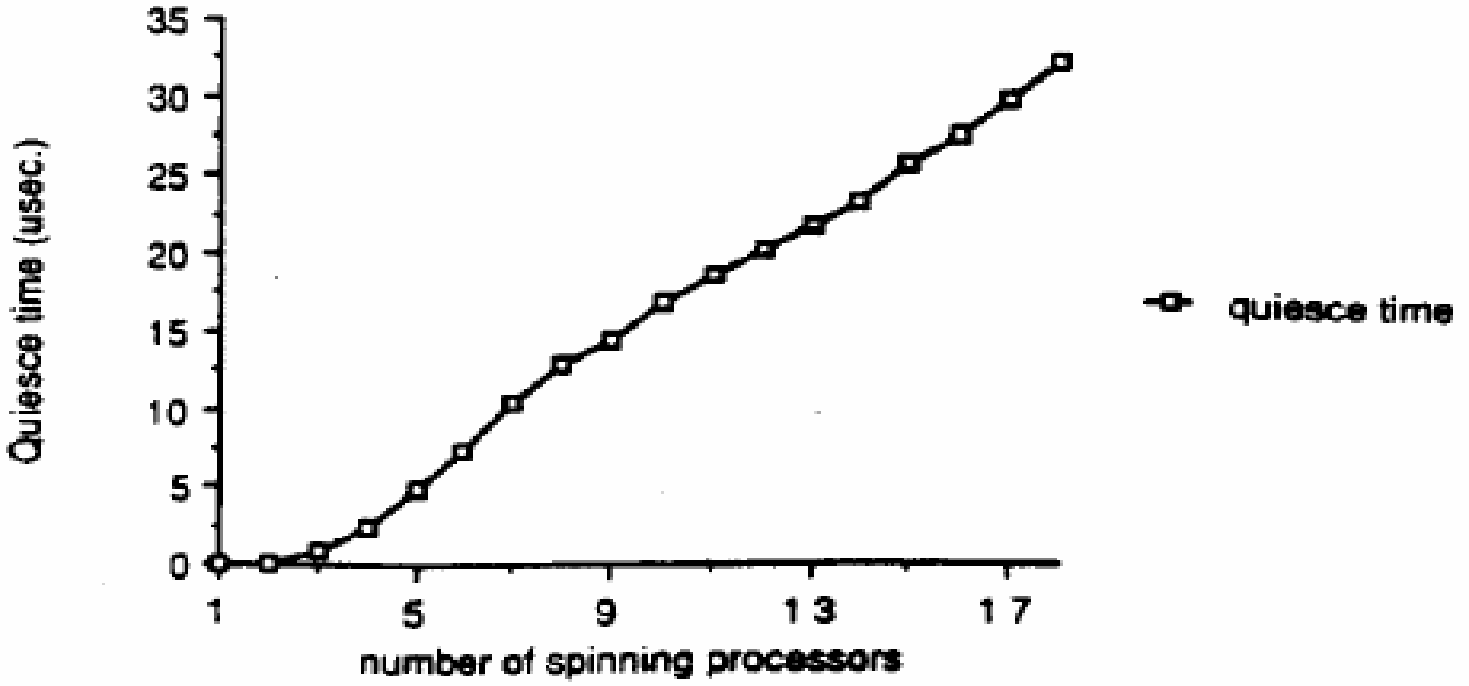


Fig. 2. Time to quiesce, spin on read (microseconds).

20 processor Sequent Symmetry CC bus-based WI MP

in summary so far

- problem:
  - more than one processor sees lock is free
  - more than one processor tries to acquire it
- can be avoided
  - one processor sees the release
  - does t&s before others decide that they should do so

Delay Alternatives

- delay after lock release

```
while ((lock == busy) or (t&s(lock) == busy))
  {
  /* failed to get lock */
  while (lock == busy) spin;
  delay(); /* don't check immediately */
}
```

  - Pi's statically assigned different delay periods
  - at most one Pi will try t&s at a time reducing contention
  - when is this good/bad?
  - problem?
    - lock release noticed immediately but acquire delay

Dynamic Delay

- idea – vary delay based on contention level
  - how do we detect contention level online
  - like in CSMA: the more collisions you have the more contention there is

- delay on lock release with exponential backoff
  - dynamic delay proportional to contention for lock
  - try t&s rightaway
  - double delay on failure
  - problem?
    - “collisions” in the spin lock case cost more if there are more processors waiting – it will take longer to start spinning on the cache again
      - bursty arrival of lock requests will take a long quiesce time making static slots better
      - each t&s is bad news in combination with invalidation-based CC protocol
  - exponential backoff with write-update good for a scalable spinlocks

- good heuristic for backoff
  - maximum bound on mean delay equal to the number of processors
  - initial delay should be a function of the delay last time before the lock was acquired (in Ethernet things start over, here t&s collision more costly)
  - Anderson suggest half of last delay
  - note that if lock is free it will be acquired immediately
- both static and dynamic delay perform poorly with bursty t&s's

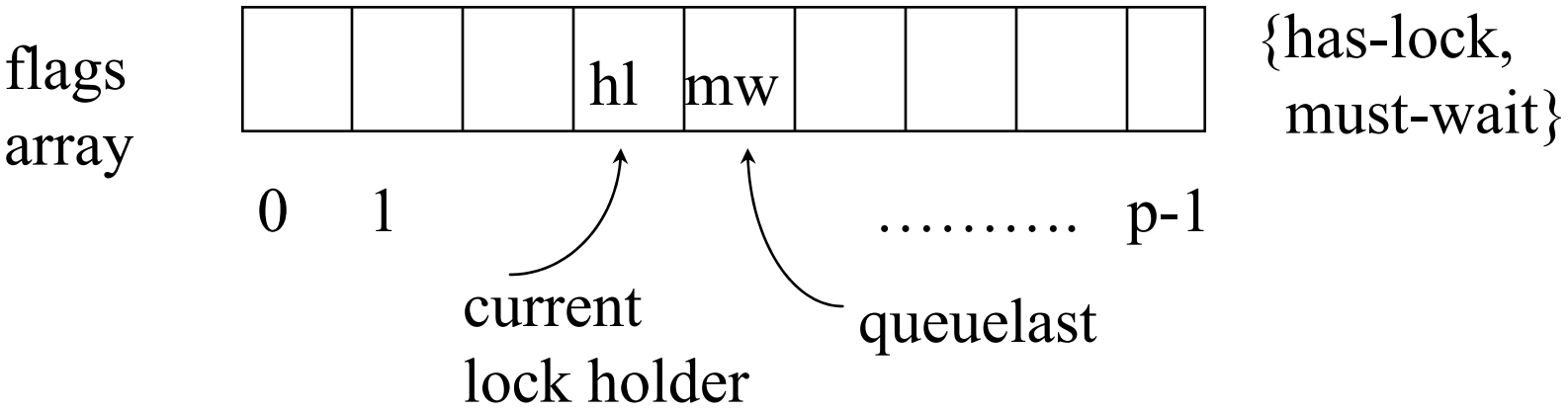
- delay between each memory reference

  while ((lock == busy) or (t&s(lock) == busy))

  delay();

  – works for NCC and CC MPs
  – static delay similar as delay-after-noticing-released-lock
  – dynamic delay assignment with backoff may be very bad (delay will increase while waiting, not contending)

Queuing locks



- a unique ticket for each arriving processor
- get ticket in waiting queue
- go into CS when you have lock
- signal next waiter when you are done
- assume a primitive op r&inc(mem)

```
init      flags[0] = has-lock;
          flags[1..p-1] = must-wait;
          queuelast = 0; /* global variable */
lock      myplace = r&inc(queuelast); /* get ticket */
          while (flags[myplace mod p] == must-wait)
                /* now in C.S */
          flags[myplace mod p] = must-wait;
unlock    flags[myplace+1 mod p] = has-lock;
```

- latency not good (test, read, and write to acquire lock, a write to release it). bandwidth/delay?
- only one atomic op per CS
- processors are sequenced
- spin variable is distinct for each processor (invalidations reduced)

- with write-update
  - All P's can spin on a single counter
  - Pj on lock release writes next sequence number into this counter for notification
  - single bus transaction notifies everyone; winner determined locally
- with write-invalidate
  - each Pj has a flag in a separate cache block
  - two bus transactions to notify the next winner
- bus-based MP without caches
  - algorithm no good without delay
- MIN-based MP without caches
  - no hot-spots due to distinct spin locations

problem

- not only is it bad for the lock holder to be preemted, but also for each spin-waiting thread that reaches the front of the queue before being scheduled
  - why?
- can be addressed by telling thread it is about to be preemted (remember Psyche?)
- what if architecture doesn't support r&inc?

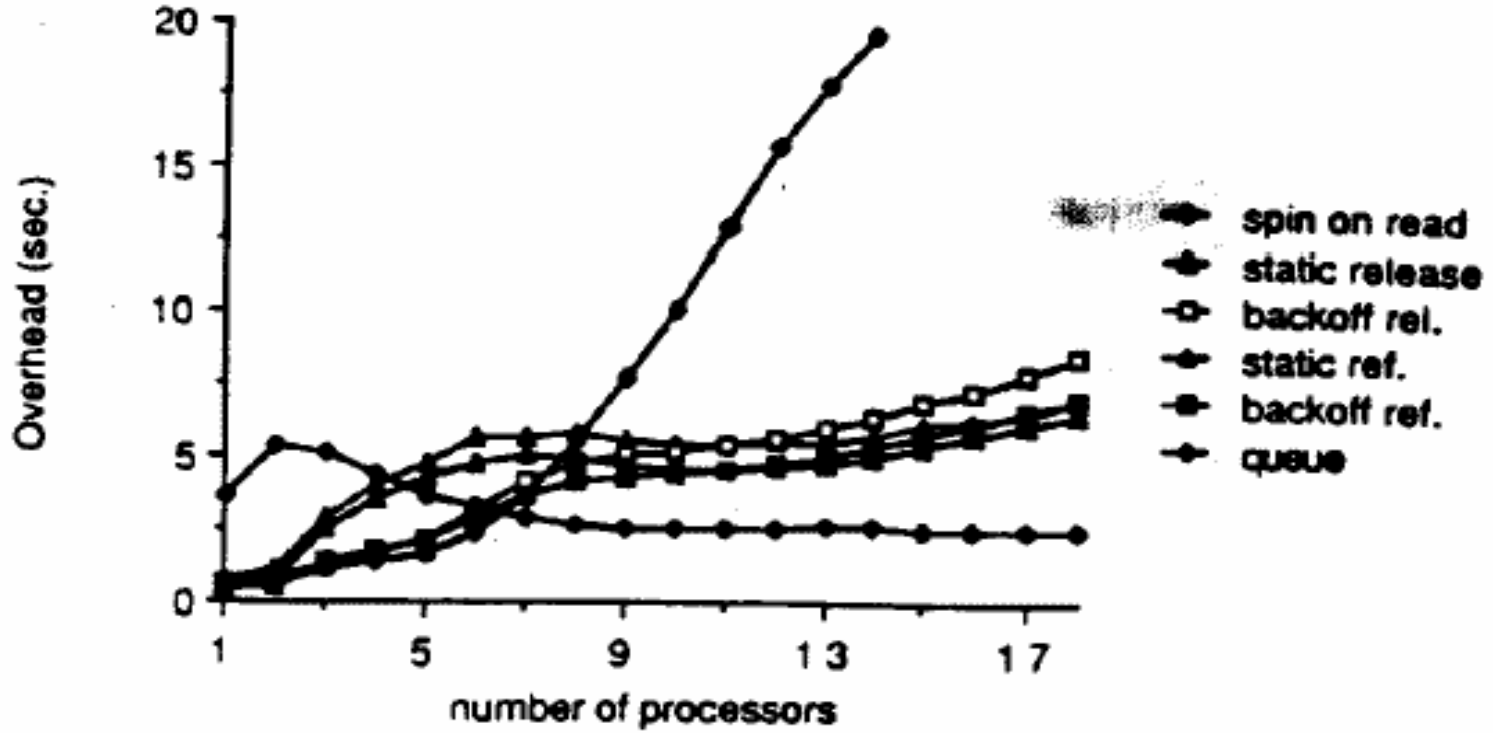


Fig. 3. Principal performance comparison: spin-waiting overhead (seconds) in executing the benchmark (measured). Each processor loops one million/ P times: acquire lock, do critical section, release lock, and compute.

20 processor Sequent Symmetry CC bus-based WI MP

- queueing high latency (startup cost)
  - need to simulate r&inc
- static release better than backoff at high loads
  - why?
    - some collisions needed for the backoff to kick-in
- delay after ref. better than after rel. since Sequent uses a WI protocol
- queuing best at high load

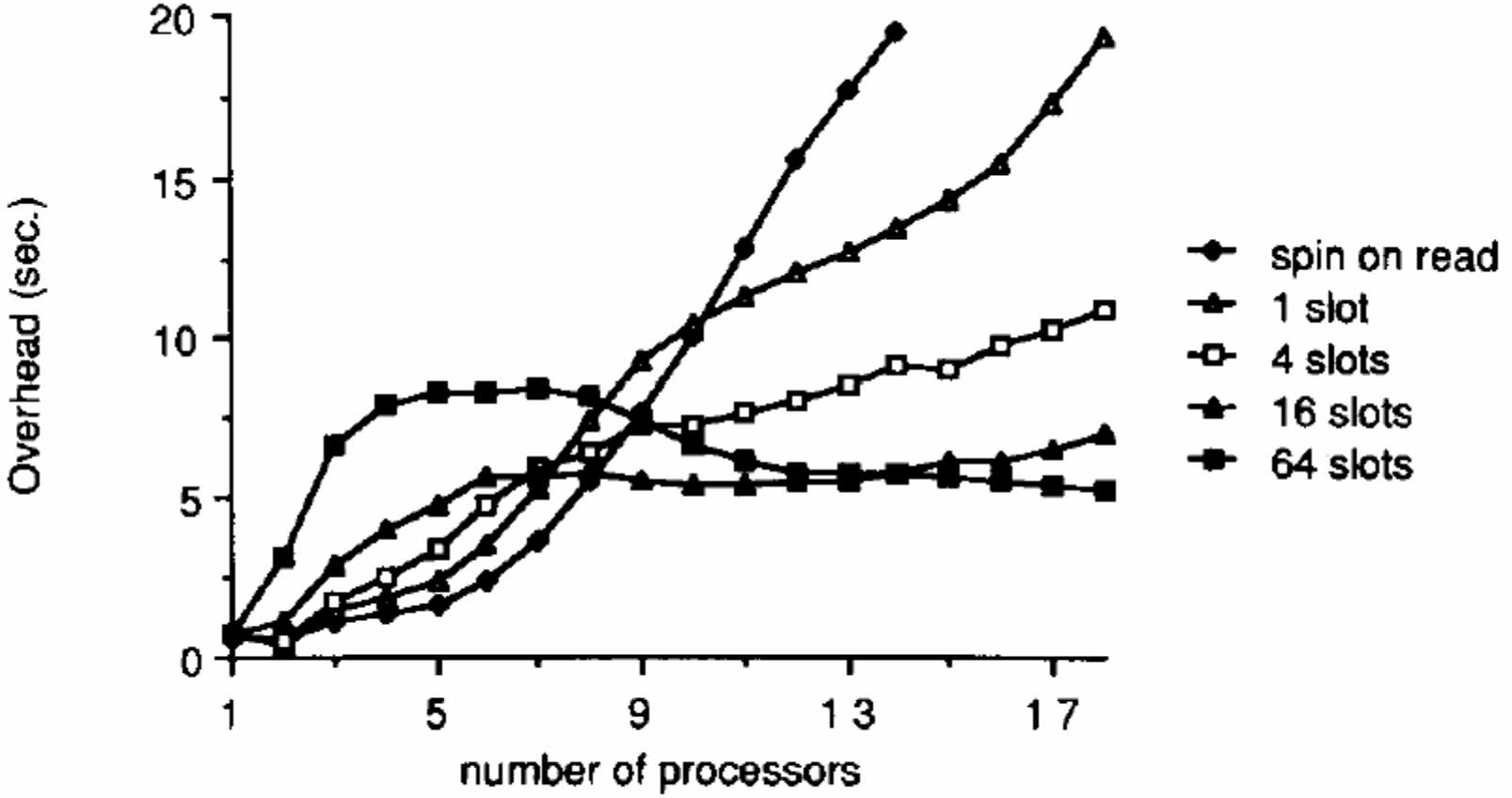


Fig. 4. Spin-waiting overhead (seconds) versus number of slots.

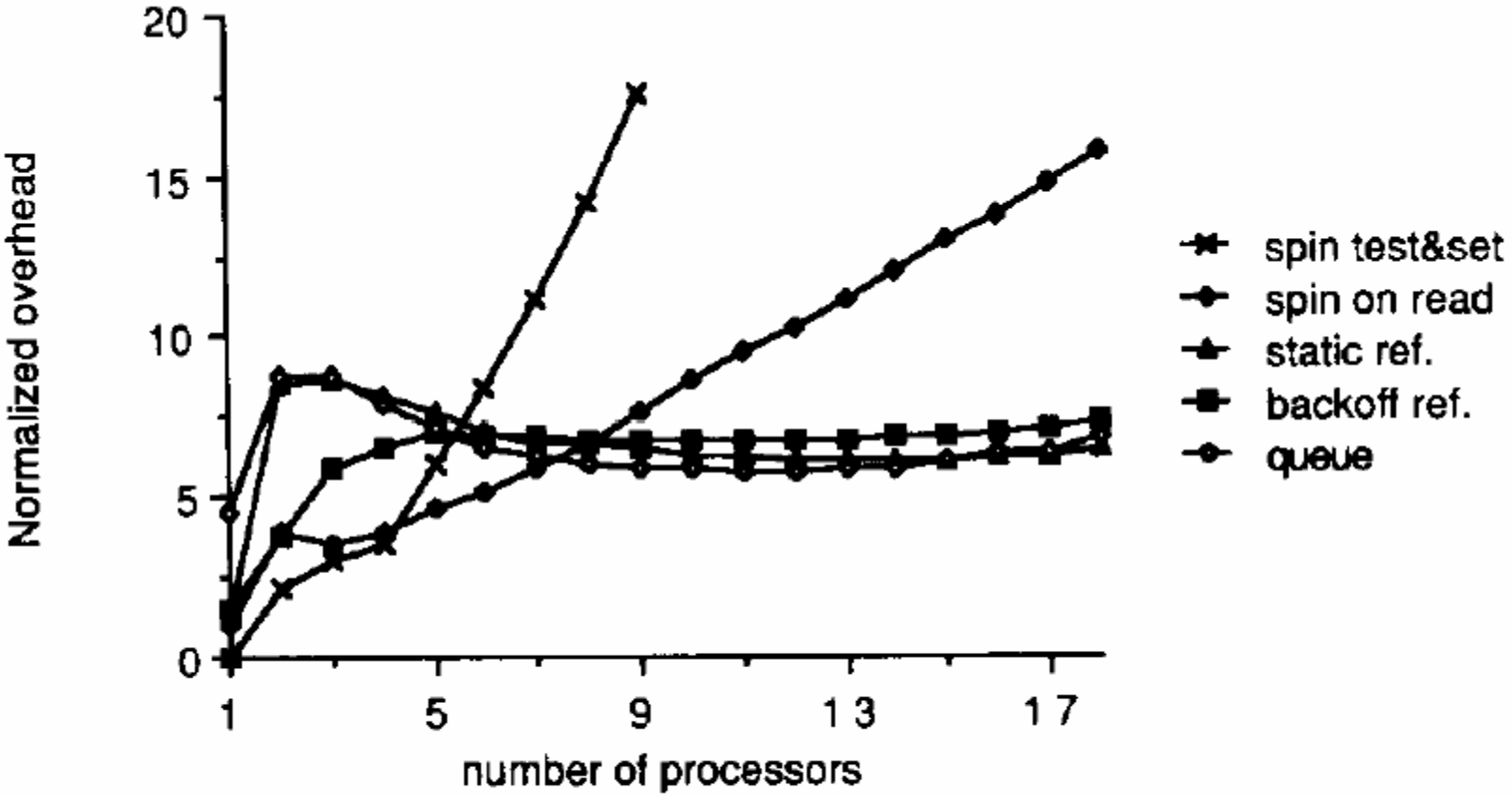


Fig. 5. Spin-waiting overhead in achieving barrier, normalized by the number of processors (microseconds per processor).