Unlocking Energy

Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis

EPFL

Main Contributions

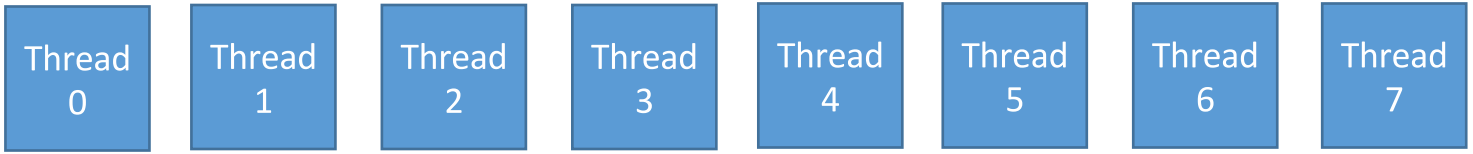
1. An extensive analysis of the energy efficiency of different types of locks. The results of this analysis can be used to optimize lock algorithms for energy efficiency.

2. The POLY conjecture: For locks, Energy efficiency $\propto$ Throughput.

3. MUTEXEE, an improved variant of pthread mutex lock. MUTEXEE delivers on average 28% higher energy efficiency than mutex on six modern systems

Need for Locking

Threads executing concurrently



Shared resource

Approach to use shared resource:

Logic & Implementation of Locks

Logic

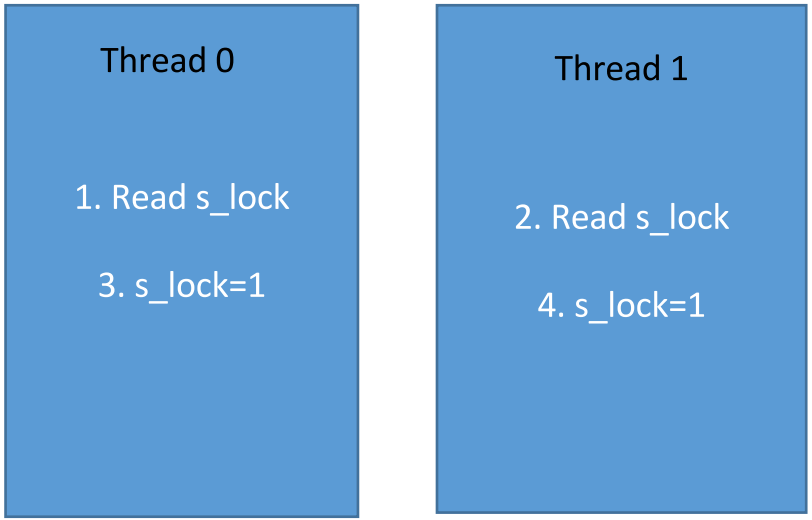
- Lock
  - Data structure
- Operations
  - Lock
  - Unlock

Implementation

- Data structure: Collection of memory locations
- Operations
  - Read from memory
  - Write to memory
  - Atomically update memory location
    - Test and set
    - Atomic addition

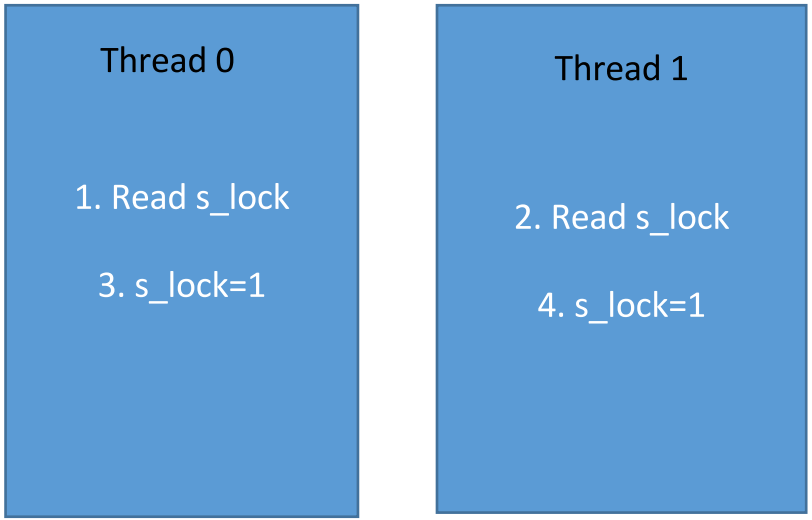
Trivial implementation of a Lock (but wrong)

- Data Structure
  - Bool s_lock=0 *[0=Unlocked, 1=Locked]*
- Operation
  - Lock:
    - While(s_lock==1){}
    - s_lock=1
  - Unlock:
    - s_lock=0
- Issue?



Trivial implementation of a Lock (but wrong)

- Data Structure
  - Bool s_lock=0 *[0=Unlocked, 1=Locked]*
- Operation
  - Lock:
    - While(s_lock==1){}
    - s_lock=1
  - Unlock:
    - s_lock=0
- Issue?
- **Conclusion: We need atomic read+update instructions**



Test And Set Algorithm

- TAS(&var, new_val):
  - Atomic{old_val = var; var=new_val; return old_val; }
- Data Structure
  - Bool s_lock=0
- Operation
  - Lock:
    - While(TAS(s_lock,1)){}
  - Unlock:
    - s_lock=0
- Issue?

Test And Set Algorithm(Contd)

- T1: TAS(), *(moved line to T1 cpu)*
- T1: processing
- T2: TAS() failed *(moved line to T2 cpu)*
- T3: TAS() failed *(moved line to T3 cpu)*
- T2: TAS() failed *(moved line to T2 cpu)*
- T3: TAS() failed *(moved line to T3 cpu)*
- ….
- T1: unlock()
- Issue: TAS keeps on *moving* cacheline from one core to another

Test & Test And Set Algorithm

- Data Structure
  - Bool S_lock=0
- Operation
  - Lock:
    - Do{
    - While(s_lock == 1)
    - }While(TAS(s_lock,1))
  - Unlock:
    - s_lock=0
- Only *copying*. No more *moving*
- Issue?

Test & Test And Set Algorithm

- Data Structure
  - Bool S_lock=0
- Operation
  - Lock:
    - Do{
    - While(s_lock == 1)
    - }While(TAS(s_lock,1))
  - Unlock:
    - s_lock=0
- Only *copying*. No more *moving*
- Issue?
  - Starvation – maybe.
  - Unfair: Based on luck. H/w Atomic instruction does not guarantee fairness.

How to guarantee fairness?

•

How to guarantee fairness?

- Queue (ordered by time of arrival)

Ticket

- SBI bank token system
- Lock:
  - Take a new token number
  - Wait for display counter to display my token number
- Unlock:
  - Display Counter increments the token number displayed

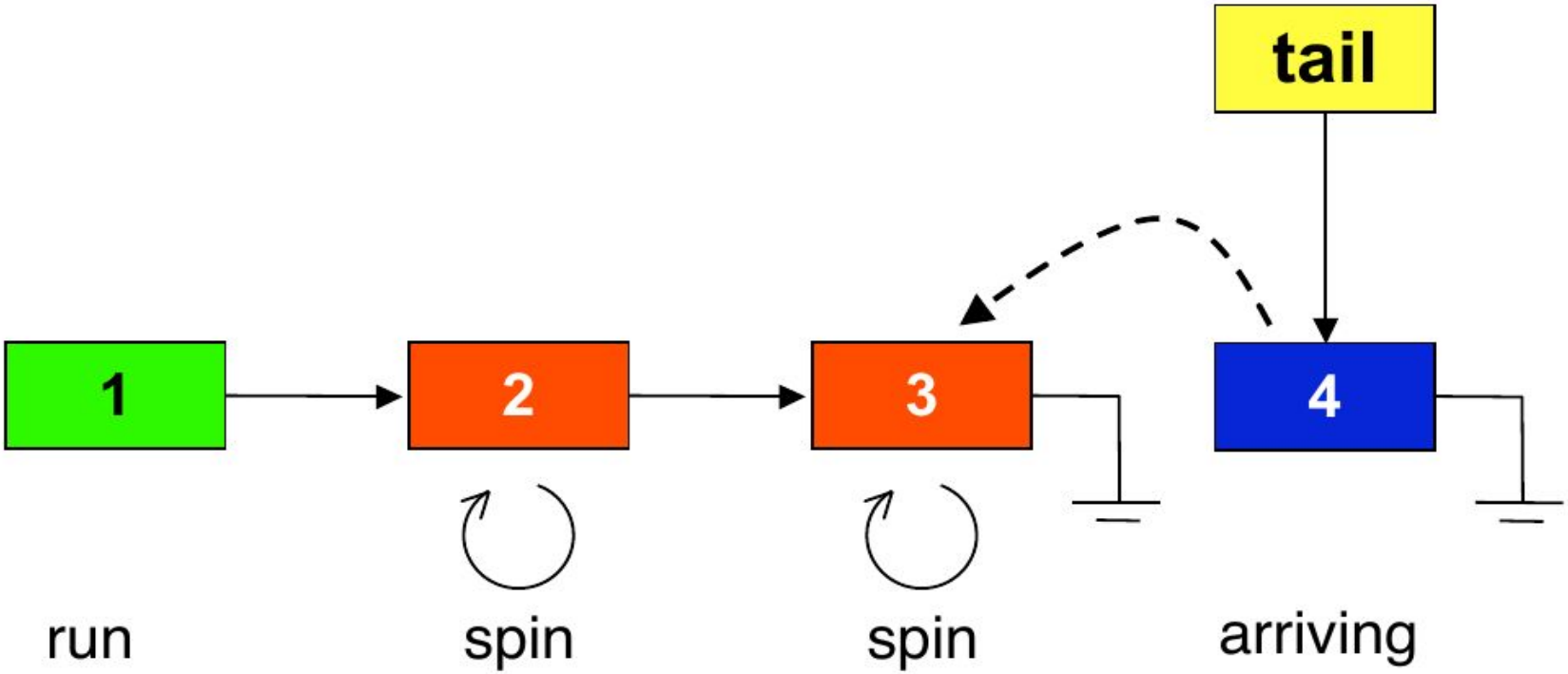
Ticket

- Data structure:
  - int display_counter
  - int next_token
- Lock:
  - int my_token = fetch_and_increment(next_token)
  - while(my_token != display_counter){}
- Unlock:
  - display_counter++
- Issue?

Ticket

- Data structure:
  - int display_counter
  - int next_token
- Lock:
  - int my_token = fetch_and_increment(next_token)
  - while(my_token != display_counter){}
- Unlock:
  - display_counter++
- Issue? All processors are spinning on the same variable. Implies that the time to retrieve the new value is linear in the number of waiting processors.

MCS



MCS

- Data structure:
  - Queue_node tail.
  - Where: Queue_node = struct { locked, next }
- Lock:
  - Atomically Insert my_node{locked=1, next=0}, at the end of the queue.
  - If other thread are running,
    - while (my_node.locked == 1)
- Unlock:
  - If queue not empty, next_node.locked=0;
- Issue: Non blocking

MCS

Locking

- my_node.next = NULL
- pred = fetch_and_store(queue, my_node)
- if (pred != NULL){
  my_node.is_locked = true
  pred.next = my_node
  while(my_node.is_locked){}
  }

Unlocking

- if (my_node.next == NULL){
  if (CAS(queue, my_node, NULL)){
  return;
  }else{
  while(my_node.next==NULL){}
  }
  my_node.next.is_locked=false

Discussion: How to optimize spinning? monitor/mwait

Mutex as Syscall : Blocking algorithm

- Data structure:
  - OS level: Lock, Blocked Queue of Threads waiting on lock
- Lock(Transfer control to OS)
  - If TAS(Lock,1)=1,
  - Suspend and Add to Block Queue
- Unlock(Transfer control to OS):
  - If Queue not empty,
  - Pop from Blocked Queue and unsuspend it.
- Issue?

Mutex as Syscall : Blocking algorithm

- Data structure:
  - OS level: Lock, Blocked Queue of Threads waiting on lock
- Lock(Transfer control to OS)
  - If TAS(Lock,1)=1,
  - Suspend and Add to Block Queue
- Unlock(Transfer control to OS):
  - If Queue not empty,
  - Pop from Blocked Queue and unsuspend it.
- Issue? Syscall Overhead even when there is no contention

Futex!= Mutex

- Data structure:
  - Shared between Userspace and Kernel
- WaitIf(addr,val)
  - Block if (*addr == val)
- Wake(addr,N):
  - Wakeup N threads waiting on this address
- CmpRequeue

Mutex based on Futex

- Lock
  - old=CAS(state,0 → 1)
    - old= 0 → Return
    - old=1 → CAS(state, 1→ 2)
      - Success? Call futex_wait(state,2)
    - old=2 → call futex_wait(state,2)
  - Repeat the above (but with 0→2)

- Data structure:
  - Int state; (0: Unlocked, 1: Locked and no one waiting, 2: Locked And waiting)
- Unlock:
  - old = State --
    - Old = 1 : Return
    - Old = 2 :
      - state = 0
      - Futex_wake(state,N=1)

Mutex : pthread_mutex

- Data structure:
  - User+Kernel level Lock
- Lock(Transfer control to OS)
  - For up to N cycles
    - Spin with pause instruction
  - If still busy, sleep with mutex.lock
- Unlock(Transfer control to OS):
  - Release lock in userspace
  - Wakeup thread with mutex.unlock
- Issue?

Mutex : pthread_mutex

- Data structure:
  - User+Kernel level Lock
- Lock(Transfer control to OS)
  - For up to N cycles
    - Spin with pause instruction
  - If still busy, sleep with mutex.lock
- Unlock(Transfer control to OS):
  - Release lock in userspace
  - Wakeup thread with mutex.unlock
- Issue? On Unlock: New thread took lock before we wakeup a thread

Mutexee unoptimized

- Data structure:
  - User+Kernel level Lock
- Lock(Transfer control to OS)
  - For up to N cycles
    - Spin with pause instruction
  - If still busy, sleep with mutex.lock
- Unlock(Transfer control to OS):
  - Release lock in userspace
  - **Wait in userspace for M cycles**
  - Wakeup thread with mutex.unlock
- Issue? Not fine tuned yet.

Characterization

Experiment setup

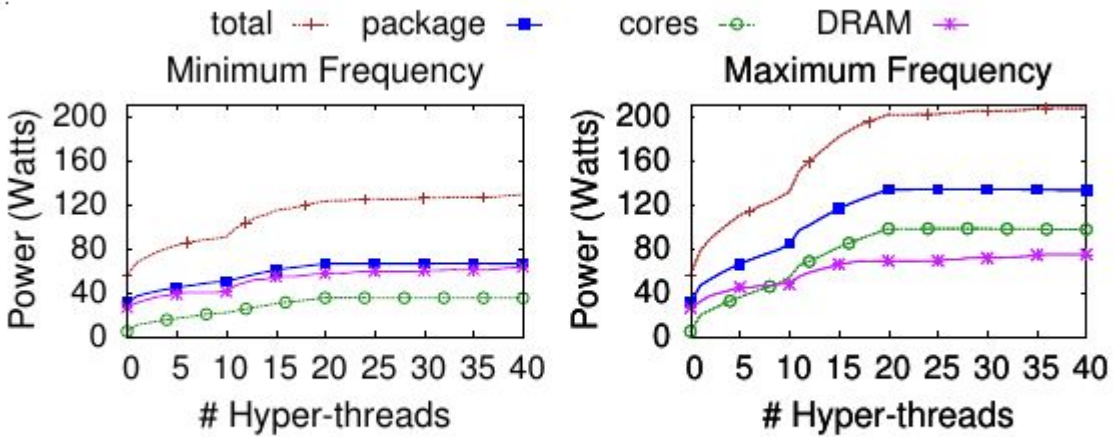
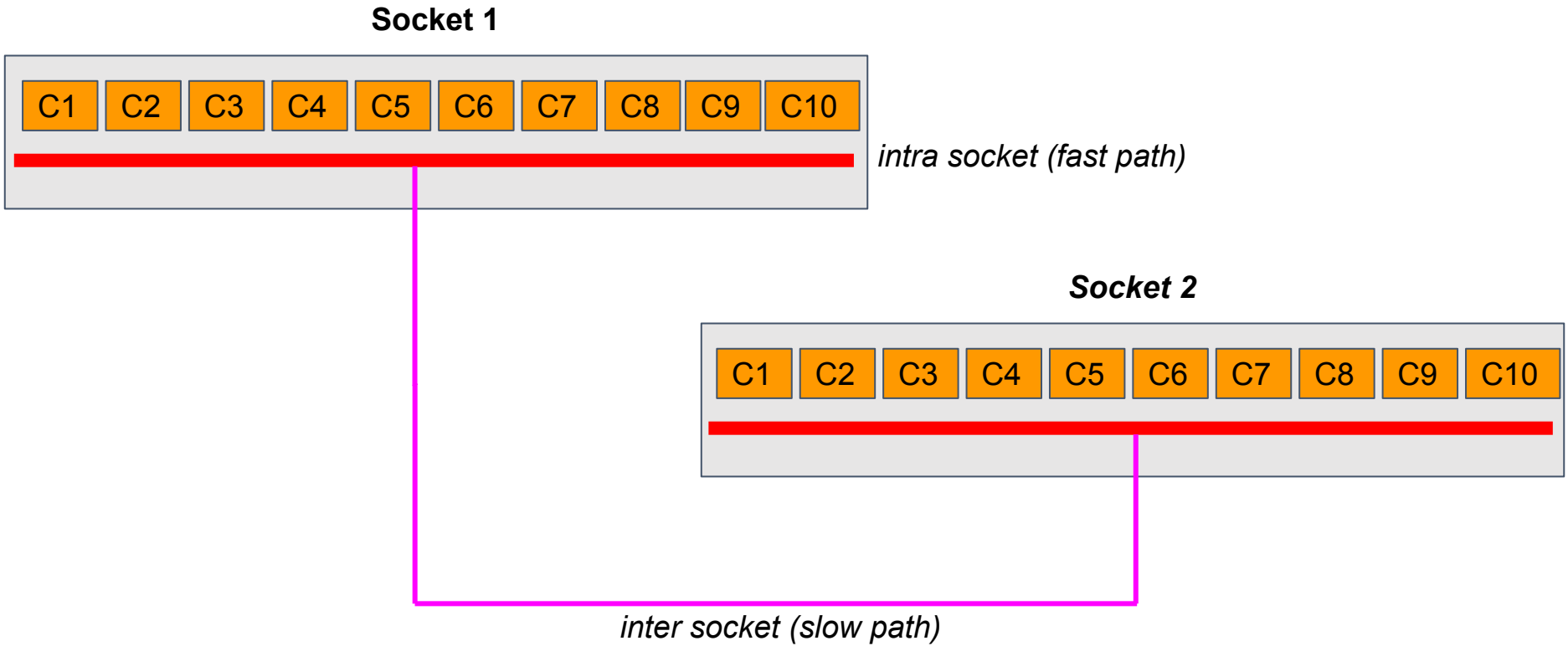


Figure 2: Power-consumption breakdown on Xeon.

- Measure power using Intel Performance counters
- SkyLake: 2 sockets, socket = 10 cores, core = 2 Hyperthreads



- 1-10 threads: socket 1 only (no Hyperthreading)
- 10-20 threads: both sockets (no Hyperthreading)
- 20-30 threads: Hyperthreading in socket 1
- 30-40 threads: Hyperthreading in both sockets

The Price of Busy Waiting

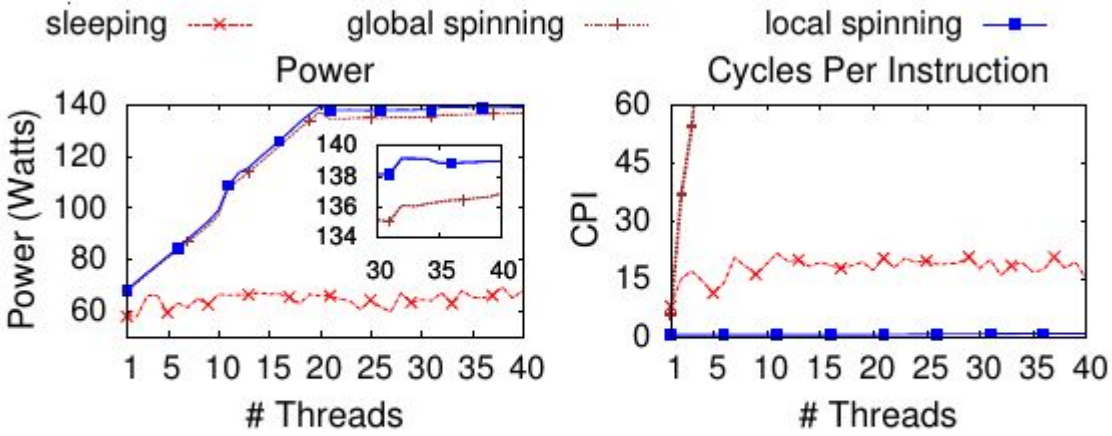


Figure 3: Power consumption and CPI while waiting.

- All threads are waiting for a lock that is never released
- *sleeping* consumes less power
- power consumption of global/local sleeping after 10 threads ?
- power consumption of global/local sleeping after 20 threads ?

The Price of Busy Waiting

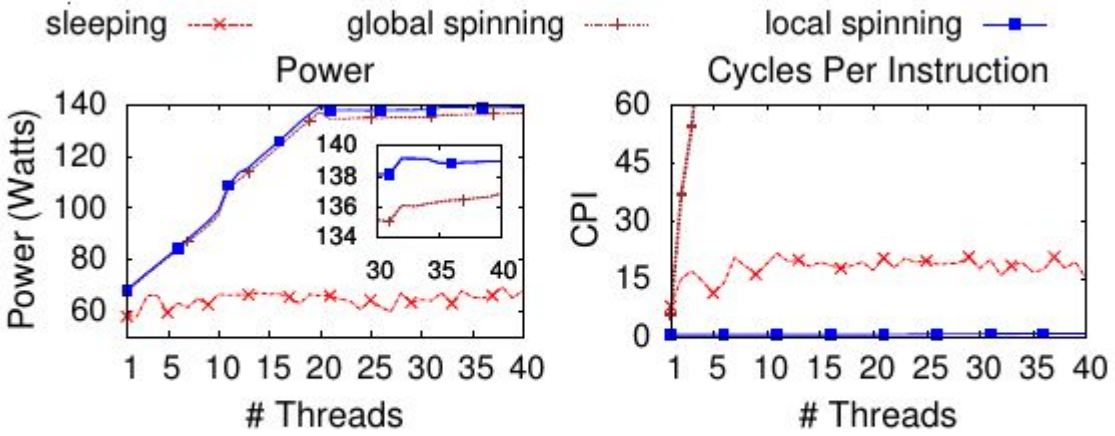


Figure 3: Power consumption and CPI while waiting.

- Power(local spinning) = 1.03 * Power(global spinning)
- Average CPI of global spinning = 530 cycles
- Why is CPI of sleeping not infinite ?
- CPI(global)>CPI(local). Still almost same power. Why?

Techniques to reduce power consumption of local spinning

1. pause or mfence instructions
2. Voltage frequency scaling
3. mwait+monitor instructions

Different types of local spinning

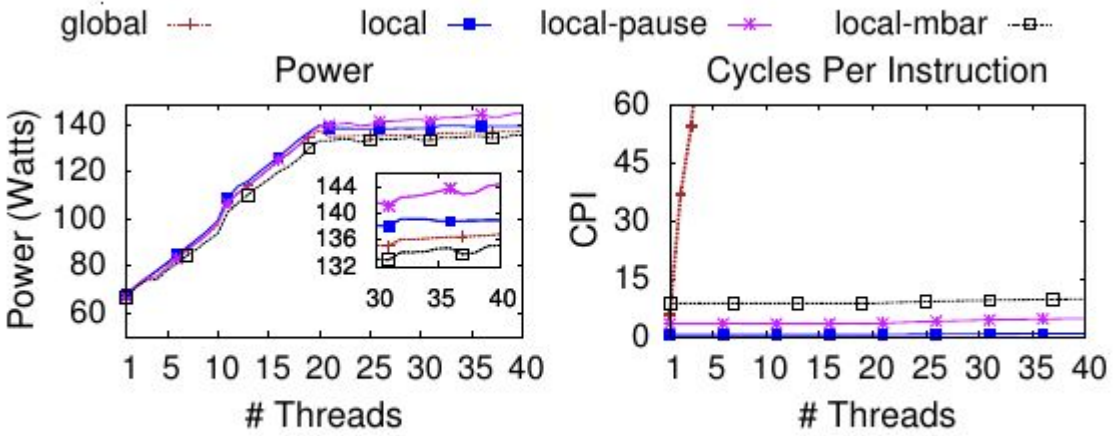


Figure 4: Power consumption and CPI while spinning.

- Upto 20 Threads: Power(local-pause) < Power(Local) *[not mentioned in the paper]*
- After 20 Threads: Power(local-pause) > Power(Local)

Different types of local spinning

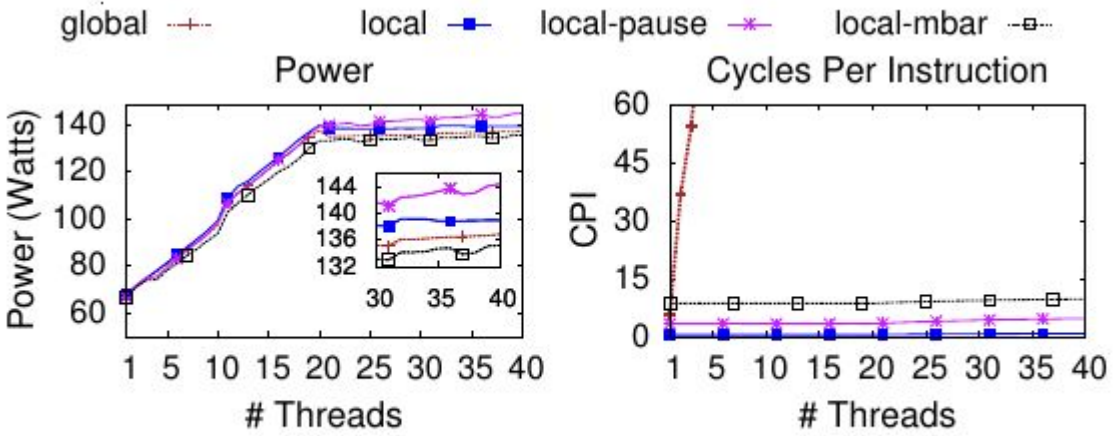


Figure 4: Power consumption and CPI while spinning.

- What if *pause* instruction inserts a delay of 100 cycles (Skylake)?

Impact of DVFS on Power

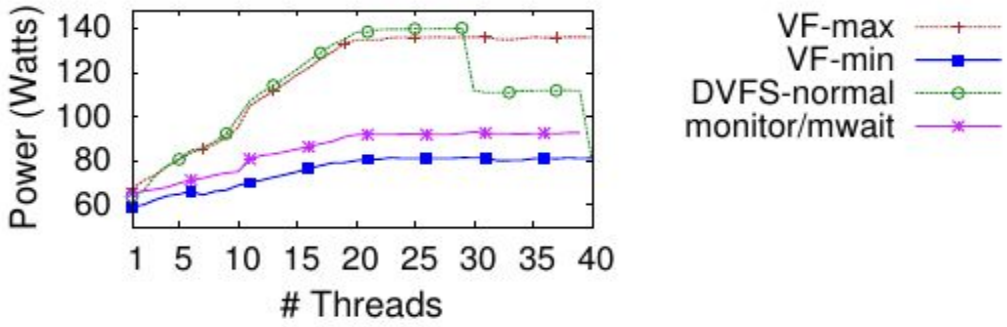


Figure 5: Power consumption of busy waiting using DVFS and `monitor/mwait`.

- VF-min: set the frequency to minimum
- VF-min: set the frequency to maximum
- DVFS-normal: hardware
- DVFS: Freq_core = Min(Freq_hyperthreads)

monitor/mwait

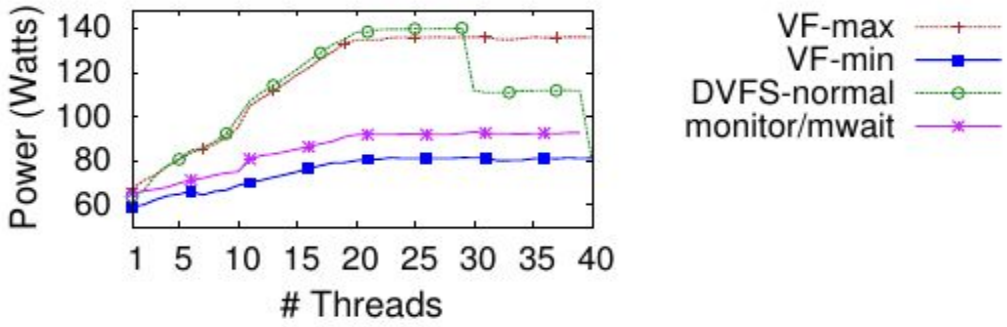


Figure 5: Power consumption of busy waiting using DVFS and `monitor/mwait`.

- Why does the power of DVFS-normal drop after 30 threads
  - Around 25 watts difference
- VF-switch operation takes around 5300 cycles
  - May work only if large critical sections (>11K cycles) and that too if both hyperthreads of the cores have reduced frequencies

monitor/mwait

```
MONITOR(lock)
LOOP
      tmpReg = load( lock )
      if( tmpReg == 0) then exit loop
      MWAIT( memLoc ) // wait until another processor may
                                 // have written the cache line
END LOOP
```

monitor/mwait

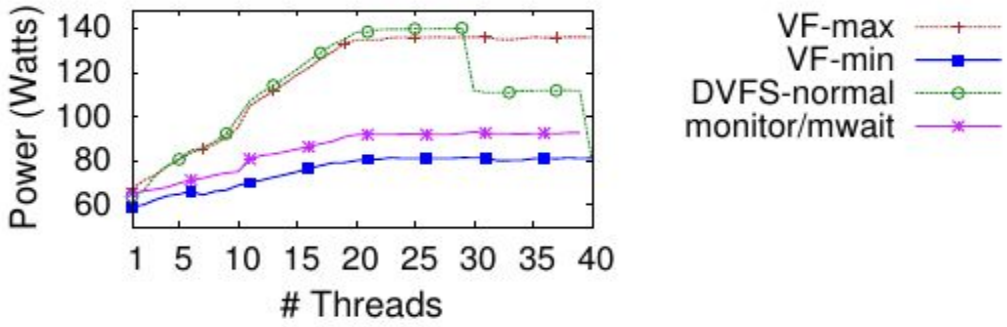


Figure 5: Power consumption of busy waiting using DVFS and `monitor/mwait`.

Consumes lesser power than local spinning.
wakeup latency(mwait) = 1600 cycles vs wakeup latency(local spinning) = 280 cycles

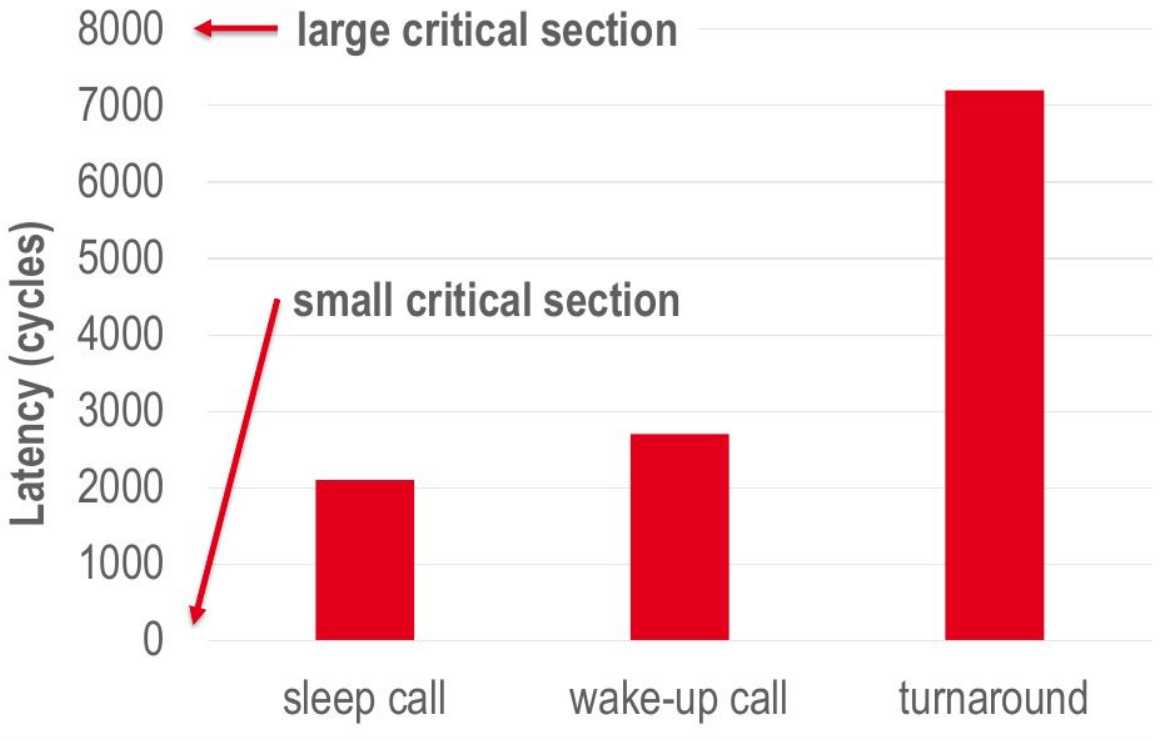
Reducing power consumption of busy waiting

1. pause instructions can increase power consumption
2. Techniques such as DVFS and monitor+mwait are more suited for OS code and not application code

Next: Understand the overheads of sleeping

Latency: The Price of Sleeping

2 threads invoke futex
1 sleeps,1 wakes up



Observations

1. Sleep call: release context
2. Wake-up call: to handover the lock
3. Turnaround latency ≈ lock handover latency

Frequent sleep/wake-up calls reduce throughput without saving energy

Futex

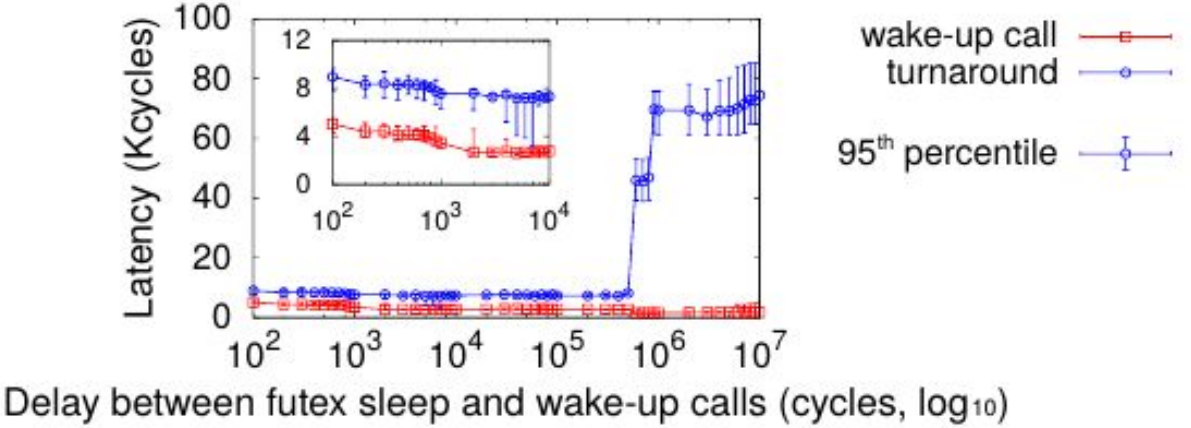


Figure 6: Latency of different `futex` operations.

- Wakeup call: 2700 cycles, Turnaround: 7000 cycles
- Beyond 600K cycles, most likely core goes to deeper idle state

Mutexee optimized

	MUTEX	MUTEXEE
lock	for up to ~ 1000 cycles spin with `pause`	for up to ~ 8000 cycles spin with `mfence`
	if still busy, sleep with `futex`	
unlock	release in user space (`lock->locked = 0`)	
	wake up a thread with `futex`	**wait in user space**

Table 1: Differences between MUTEX and MUTEXEE.

Turnaround time (7000)

Busy waiting

Coherence (384)

Evaluation

Uncontested locking performance

	MUTEX	TAS	TTAS	TICKET	MCS	MUTEXEE
Throughput	11.88	16.88	16.98	16.97	12.04	13.32
TPP	174.31	248.14	249.41	249.24	176.72	195.48

Table 2: Single-threaded lock throughput and TPP.

- Simple logic ⇒ higher performance

Performance comparison

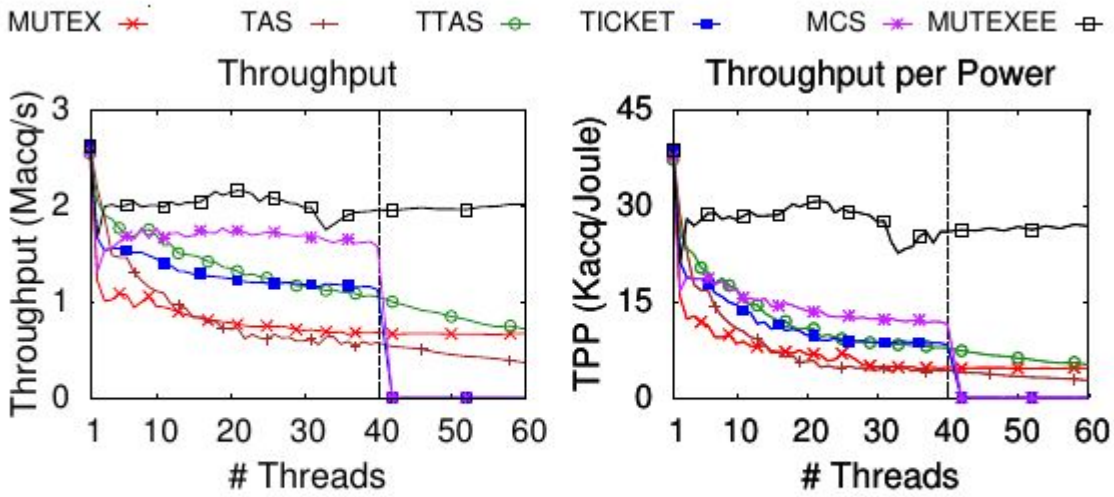
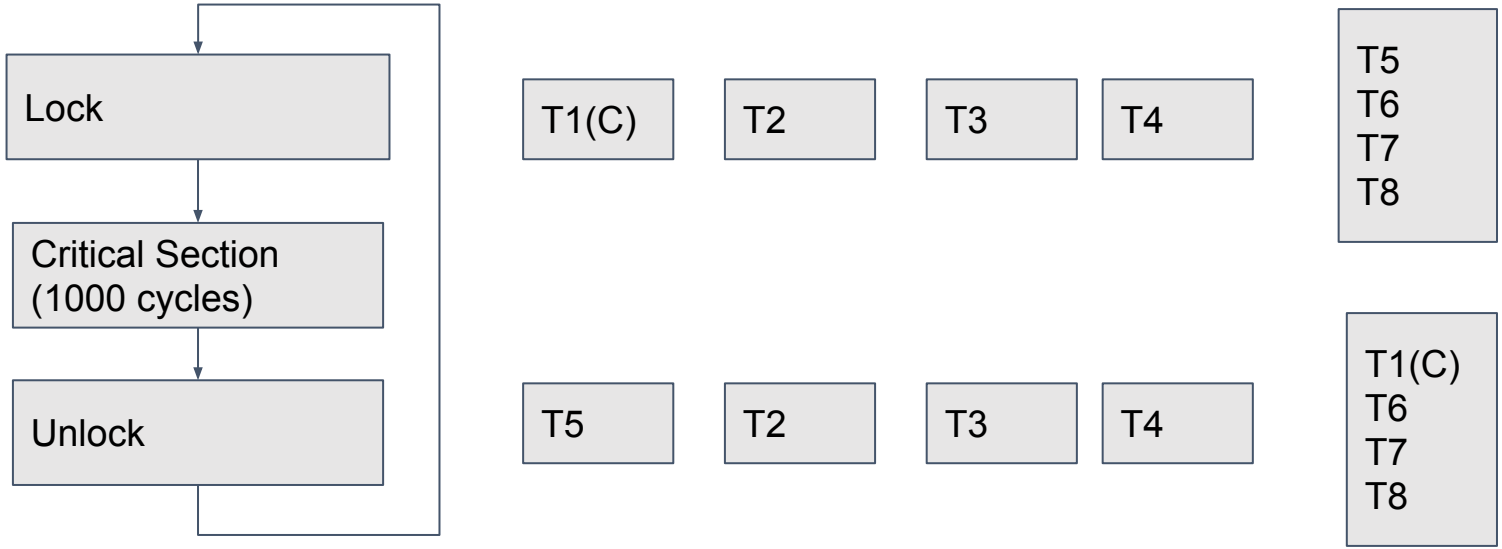


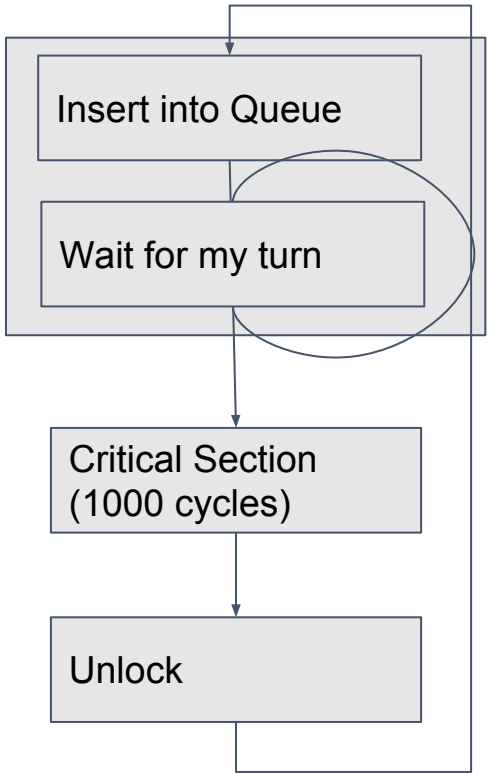
Figure 11: Using a single (global) lock.

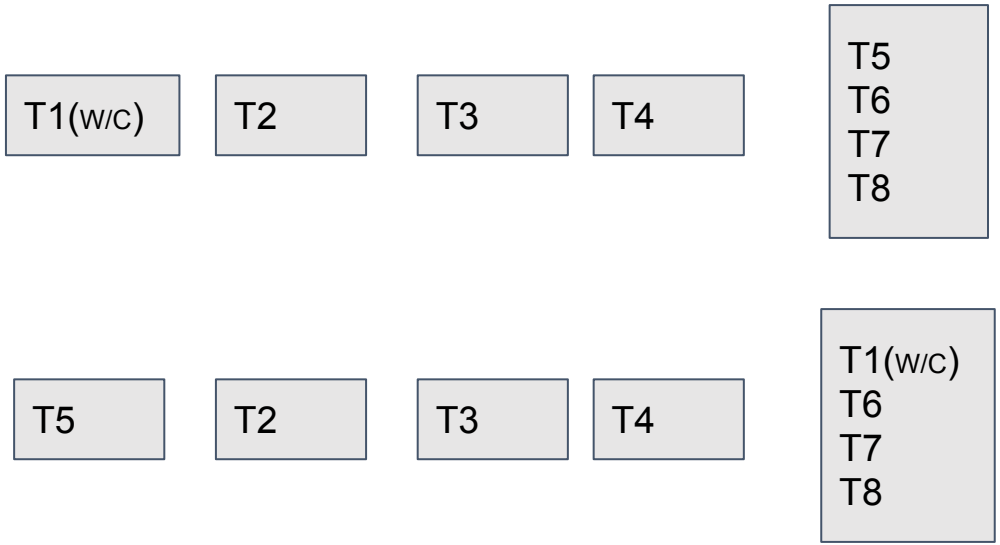
- After 40: Performance of TAS,TTAS,Ticket,MCS drops.
  - Ticket, MCS is dead after 40

Issue with NonBlocking



Issue with NonBlocking (Queue)





Ticket is better

Number of threads in MySQL, SQLite > Number of core

MacroBenchmarks

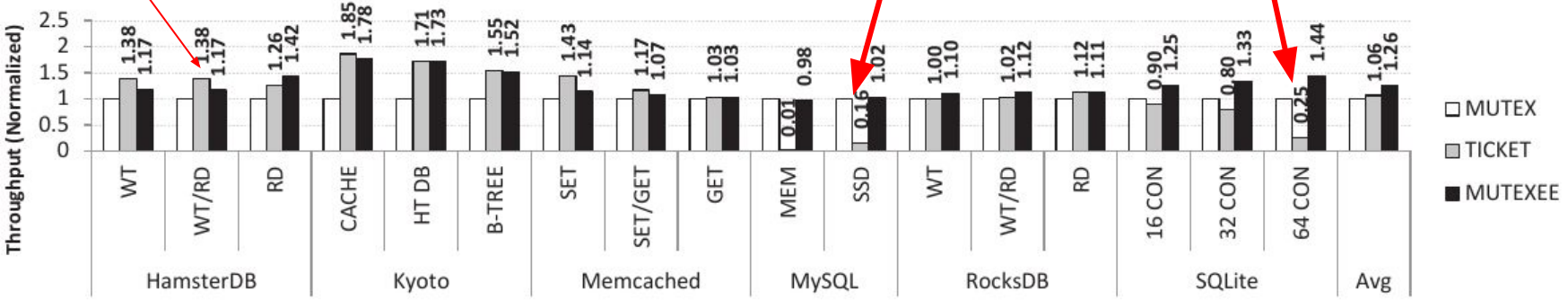


Figure 13: Normalized (to MUTEX) throughput of various systems with different locks. (Higher is better)

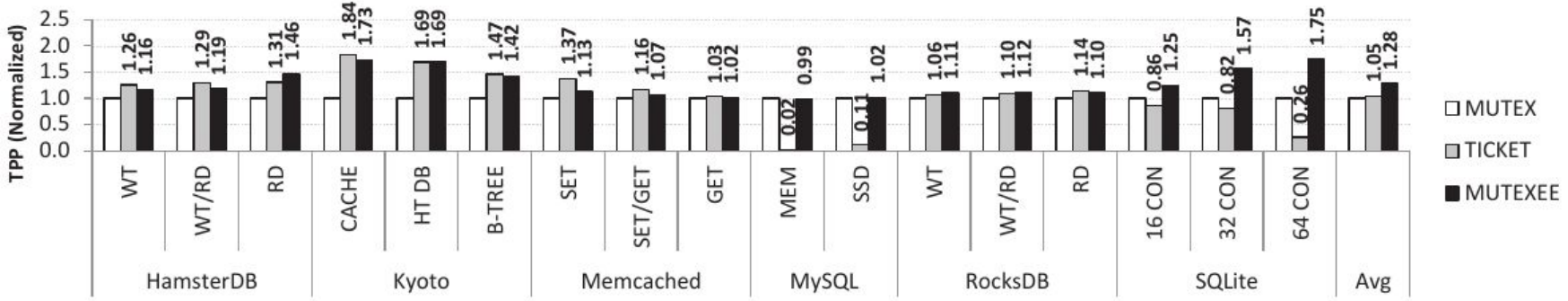


Figure 14: Normalized (to MUTEX) energy efficiency (TPP) of various systems with different locks. (Higher is better)

MacroBenchmarks

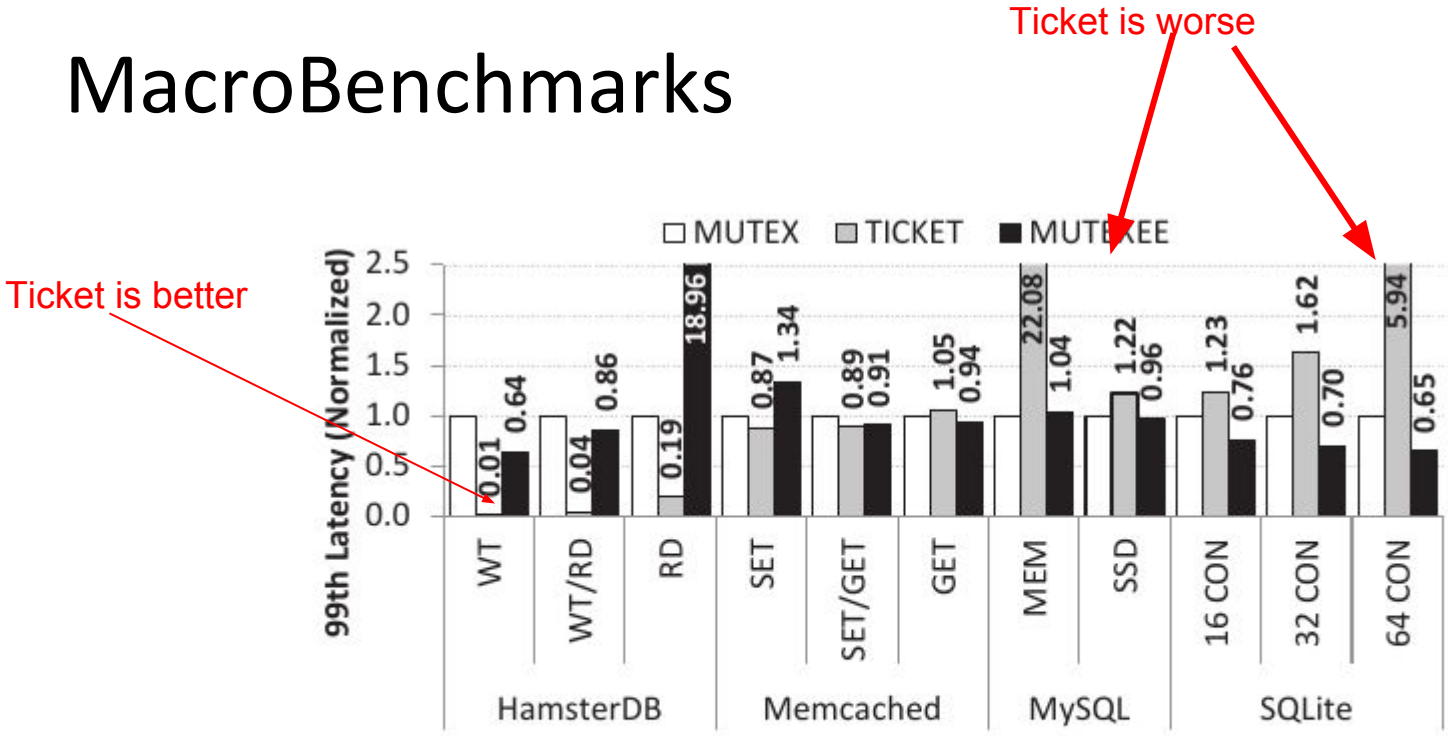


Figure 15: Normalized (to MUTEX) tail latency of various systems with different locks. (Lower is better)

Conclusion

Approach going forward

Issues with sleeping and waiting

- Sleep(Kernel level): Latency
- Busy waiting (User level): Power

Idea

- Combine both these techniques
- Lock: Try busy waiting X times and then call sleep

Time to wait at user level for mutex

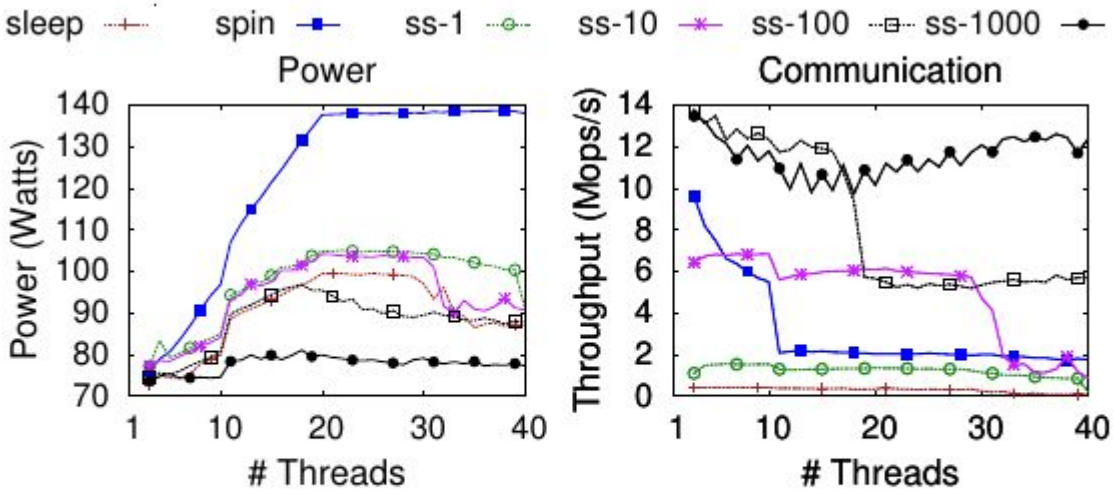


Figure 7: Power and communication throughput of sleeping, spinning, and spin-then-sleep for various Ts.[8]

- Spectrum: *sleep --- ss1 --- ss10 --- ss100 --- ss1000 --- spin*
- Power(spin) is the highest
- Throughput(spin) dropping after 10 threads?

Details of the Futex experiment

T1: *futex-sleep****....2100******.....deschedule****...X...****schedule****...4000....****sysret*

T2: *futex-wakeup****.....2700******.....sysret*

X depends on the state of the core that is sleeping

Critical path delays: T1: schedule**.....**sysret and T2: futex-wakeup**....**sysret

Experiment: Vary the time between *futex-sleep(T1)* and *futex-wakeup(T2)* and study its impact on the time between *actual-wakeup(T2)* and *sysret(T1)*

Power mode

- H/w power state
  - P : CPU is busy executing
    - P0: H/w managed, Turbo, opt for performance
    - P1-Pn
  - C0,C1,C6 (Core C states): when CPU is Idle/Hlt
  - C3 (Package C states) : turns L3 cache off (a part of)
- Tools:
  - Cpufreqd
  - Thermald : user daemon
    - DTS temperature sensor
    - uses Intel P state driver, Power clamp driver, Running Average Power Limit control and cpufreq as cooling methods