Unlocking Energy

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

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:

Thread 0  Thread 1  Thread 2  Thread 3  Thread 4  Thread 5  Thread 6  Thread 7

Shared resource:

Approach to use shared resource:

Lock resource ➔ Use it ➔ Unlock it
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(), \hspace{1cm} (moved line to T1 cpu)
• T1: processing
• T2: TAS() failed \hspace{1cm} (moved line to T2 cpu)
• T3: TAS() failed \hspace{1cm} (moved line to T3 cpu)
• T2: TAS() failed \hspace{1cm} (moved line to T2 cpu)
• T3: TAS() failed \hspace{1cm} (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

• 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=\text{CAS(state},0 \rightarrow 1)
    • old= 0 → Return
    • old=1 → \text{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.

- $\text{Power(\text{local spinning})} = 1.03 \times \text{Power(\text{global spinning})}$
- Average CPI of global spinning = 530 cycles
- Why is CPI of sleeping not infinite?
- CPI(\text{global}) > CPI(\text{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

![Graph showing power consumption and CPI vs. number of threads.]

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

![Graph showing power consumption]

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
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

Observations

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

<table>
<thead>
<tr>
<th></th>
<th>MUTEX</th>
<th>MUTEXEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td>for up to (\sim 1000) cycles spin with <code>pause</code> if still busy, sleep with <code>futex</code></td>
<td>for up to (\sim 8000) cycles spin with <code>mfence</code></td>
</tr>
<tr>
<td>unlock</td>
<td>release in user space ((\text{lock} \rightarrow \text{locked} = 0)) [wait in user space] wake up a thread with <code>futex</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Differences between MUTEX and MUTEXEE.
Evaluation
Uncontested locking performance

<table>
<thead>
<tr>
<th></th>
<th>MUTEX</th>
<th>TAS</th>
<th>TTAS</th>
<th>TICKET</th>
<th>MCS</th>
<th>MUTEXEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>11.88</td>
<td>16.88</td>
<td>16.98</td>
<td>16.97</td>
<td>12.04</td>
<td>13.32</td>
</tr>
<tr>
<td>TPP</td>
<td>174.31</td>
<td>248.14</td>
<td>249.41</td>
<td>249.24</td>
<td>176.72</td>
<td>195.48</td>
</tr>
</tbody>
</table>

Table 2: Single-threaded lock throughput and TPP.

- Simple logic $\Rightarrow$ 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

Lock

Critical Section (1000 cycles)

Unlock

T1(C) T2 T3 T4

T5 T6 T7 T8

T1(C) T6 T7 T8
Issue with NonBlocking (Queue)

- Insert into Queue
- Wait for my turn
- Critical Section (1000 cycles)
- Unlock

T1(w/c) → T2 → T3 → T4 → T5 → T6 → T7 → T8

T1(w/c) → T2 → T3 → T4 → T5 → T6 → T7 → T8
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 $T$s.

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

T1: \textit{futex-sleep} \ldots \textbf{2100} \ldots \textit{deschedule} \ldots \textbf{X} \ldots \textit{schedule} \ldots \textbf{4000} \ldots \textit{sysret}

T2: \hspace{1cm} \textit{futex-wakeup} \ldots \textbf{2700} \ldots \textit{sysret}

\textbf{X} depends on the state of the core that is sleeping

Critical path delays: T1: \textit{schedule} \ldots \textit{sysret} and T2: \textit{futex-wakeup} \ldots \textit{sysret}

Experiment: Vary the time between \textit{futex-sleep}(T1) and \textit{futex-wakeup}(T2) and study its impact on the time between \textit{actual-wakeup}(T2) and \textit{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
  - ThermaId : user daemon
    - DTS temperature sensor
    - uses Intel P state driver, Power clamp driver, Running Average Power Limit control and cpufreq as cooling methods