Locks are critical for application performance

![Graph showing performance of Lock A and Lock B](image-url)

- **Lock A**
- **Lock B**

- X-axis: Number of threads
- Y-axis: Operations per microsecond (Ops/µsec)
One lock cannot rule all scenarios

Read-intensive workload

Write-intensive workload

# of threads

Ops/µsec

Lock A

Lock B
Depending on scenarios, different lock perform best

Hardware

Software
Locks considering hardware

**NUMA (non-uniform memory access)**

Accessing local socket data is faster than remote socket data.

**AMP (Asymmetric multicore processors)**

Faster performance cores and slower efficiency cores in one processor.
Locks considering software requirements

• Read / write ratio?

• Length of critical section?

• Any specific threads need to be prioritized?
Kernel locks also affect application performance

Kernel lock

Application

SYSCALL

User space

Kernel space
Kernel locks also affect application performance

- Application-agnostic
- Invisible to application developers
- Generic design to support common cases
...But difficult to change

- Application-agnostic
- Invisible to application developers
- Generic design to support common cases

TO-DO LIST
1. Modify kernel
2. Compile kernel
3. Install new kernel
4. Reboot
Issue with current kernel locks

- Lock implementations are application agnostic
- Only a few locks contend for given application
- Difficult to implement a new lock design
The solution – SynCord

Lock implementations are application agnostic

→ Let application developers safely change locks in the kernel on the fly

Only a few locks contend for given application

→ Modify set of locks at various granularities

Difficult to implement a new lock design

→ Expose set of APIs to easily write various lock policies
Key behavior of queue-based lock

To access shared resource, thread needs to acquire lock
Key behavior of queue-based lock

If lock is free, thread directly **acquires** lock
Key behavior of queue-based lock

Since lock is already held, other threads **join waiting queue**
Key behavior of queue-based lock

Reorder waiters in the queue to group waiters from same socket (ShflLock$^1$, CNA$^2$)

1. Scalable and Practical Locking With Shuffling. SOSP ’19
2. Compact NUMA-aware Locks. EuroSys ’19
Key behavior of queue-based lock

- Lock acquired
- Reorder waiting queue

Lock released

**Release** lock when thread finishes using resource
Key behavior of queue-based lock

- Lock acquired
- Reorder waiting queue
- Lock released
- Next waiter acquire lock
SynCord exposes kernel locks’ key behaviors as APIs

- lock_acquired ([lock])
- lock_released ([lock])
- bool should_reorder ([lock], anchor_node, curr_node)

And 7 more APIs!
SynCord overview with NUMA-aware example

- NUMA (non-uniform memory access)

Accessing local socket memory is faster than remote socket memory
SynCord overview with NUMA-aware example

- NUMA (non-uniform memory access)

Accessing local socket memory is faster than remote socket memory

Minimize cache line bouncing
User writes custom lock policy and specify target point

```c
bool should_reorder(lock *lock, node *anchor, node *curr) {
    return (anchor->socket_id == curr->socket_id);
}
```

Target point: rename_lock
User writes custom lock policy and specify target point

```c
bool should_reorder(lock *lock, node *anchor, node *curr) {
    return (anchor->socket_id == curr->socket_id);
}
```

1. User writes custom lock policy and specify target point

2. Compile program

3. Load and verify

SynCord overview with NUMA-aware example
User writes custom lock policy and specify target point

```c
bool should_reorder(lock *lock, node *anchor, node *curr)
{
    return (anchor->socket_id == curr->socket_id);
}
```

1. **SynCord overview with NUMA-aware example**

   - **1. User writes custom lock policy and specify target point**
   - **2. Compile program**
   - **3. Load and verify**

   - **Verifier**
     - **memory access**
       - No arbitrary memory update
     - **helper functions**
       - Only allowlisted functions can be called
     - **code termination**
       - Lock policy must not hang
SynCord overview with NUMA-aware example

1. User writes custom lock policy and specify target point

   ```c
   bool should_reorder(lock *lock, node *anchor, node *curr) {
     return (anchor->socket_id == curr->socket_id);
   }
   ```

2. Compile program

3. Load and verify

4. If failed, notify users

   If passed, SynCord proceeds.
SynCord overview with NUMA-aware example

1. User writes custom lock policy and specify target point

```c
bool should_reorder(lock *lock, node *anchor, node *curr) {
    return (anchor->socket_id == curr->socket_id);
}
```

2. Compile program

3. Load and verify

4. Passed

5. Notify user on patch complete

- If failed, notify users

- Convert Lock Patcher
SynCord overview with NUMA-aware example

Rename files in a directory

Performance similar to its static implementation

2.5x Throughput

stock
Syncord-NUMA
static-NUMA

1 socket >1 socket
What if a user provide wrong code?

- Verifier + API design $\rightarrow$ sandboxed impact
- Mechanism remains intact

```plaintext
lock_acquired(lock)

lock_released(lock)

bool should_reorder(lock, anchor_node, curr_node)

Lock holder

Waiter

Waiter

Resource

Read-only

Never break mutual exclusion

• Only provide hint for reordering
• Runtime check to prevent starvation
```
What user can do & can’t do with SynCord

**Can do**

- Prioritize/penalize specific threads
- Run additional code blocks in hooking points
- Affect performance
- Affect fairness

**Can’t do**

- Break mutual exclusion
- Change underlying mechanism
- Change lock type
Usecases

1. NUMA-aware lock
2. Asymmetric multicore lock
3. Scheduler-cooperative lock
4. Biased per-CPU readers-writer lock
5. Dynamic lock profiling

Customized for

- HW: NUMA
- HW: AMP+NUMA
- SW: Length of CS
- SW: Read-intensive
- HW: NUMA
**Dynamic lock profiling**

**Lockstat** vs **Dynamic lock profiling**

- In-kernel lock statistic tool
- System-wide tracing
- Enabled in compile time
- More memory usage from booting

- Implemented with SynCord APIs
- Can trace single lock instance
- Dynamically enabled
- No memory overhead once disabled

![Graph showing slowdown % vs # threads]
Dynamic lock profiling: avg critical section length
Dynamic lock profiling: avg critical section length

```c
void lock_acquired(lock *lock)
{
    lock->acquisitions++;
    lock->hold_start = get_time();
}

void lock_released(lock *lock)
{
    lock->holdtime += (lock->hold_start - get_time());
}
```
Dynamic lock profiling: avg critical section length

```c
void lock_acquired(lock *lock) {
    lock->acquisitions++;
    lock->hold_start = get_time();
}

void lock_released(lock *lock) {
    lock->holdtime += (get_time() - lock->hold_start);
}
```

Auxiliary data structures
Conclusion

- Kernel locks are basic building of concurrent OSes
  - Affect performance and scalability of applications
  - Out of reach of application developers

- SYNCOND Framework
  - Allow users to fine-tune locking primitives dynamically
  - Exposes a set of user implementable APIs
  - No need to reinstall the kernel or reboot the system

- Application can now address pathological locking cases

Thank you!