Agenda

• Current Design
• Issues in Current Design
• Ingredients for lock free algorithms
• Design for lock free RW concurrency in rte_hash
• Reclaiming Memory
• Q & A
Current Design

- **Single Writer**
- **Multiple Simultaneous Readers**

The diagram illustrates a design for handling readers and writers in a system, with a focus on locking and hashing. The key concepts are:

1. **rte_hash_key**
   - `void* pdata;`
   - `char key[0];`

2. **rte_hash_bucket**
   - `uint32_t signature`
   - `uint32_t key_index`
   - `uint32_t alt_signature`

3. **hash_fn (key)**
   - Takes a key and computes a signature and alternate signature.

4. **Primary Bucket**
   - Stores data based on the computed signature.

5. **Secondary Bucket**
   - Stores data based on the computed alternate signature.

The diagram also shows the interaction between the bucket and the key store, indicating how readers and writers can access or modify data under lock control.
Current Design

- Cuckoo Hash - Keys might move to alternate locations during hash add
Current Issues

• Preempted writer will block the readers
• Problem applies for platforms with HTM when it falls back to traditional locks

static inline void rte_rwlock_write_lock_tm(rte_rwlock_t *rwl)
{
    if (likely(rte_try_tm(&rwl->cnt)))
        return;
    rte_rwlock_write_lock(rwl); // fallback to traditional lock
}

• Application uses the key store index to reference its data
  • Index should not be freed till the application has stopped using it
Current Issues

• Performance does not scale

Lookup performance with and without hash add

<table>
<thead>
<tr>
<th># cores</th>
<th>Cycles per Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Lookup with add</td>
</tr>
<tr>
<td>4</td>
<td>Lookup</td>
</tr>
<tr>
<td>0</td>
<td>Lookup with add</td>
</tr>
<tr>
<td>8</td>
<td>Lookup</td>
</tr>
<tr>
<td>10</td>
<td>Lookup with add</td>
</tr>
<tr>
<td>12</td>
<td>Lookup</td>
</tr>
</tbody>
</table>

© 2018 Arm Limited
Ingredients for lock free algorithms

- Atomic operations – sizes matter, look for wider support
- Memory ordering operations – Memory barriers or C11 atomics
  - Orderings by themselves are not enough. Need to identify the ‘payload’ and ‘guard’
  - Payload – the data being propagated from writer to reader, accesses are not required to be atomic
  - Guard – Protects access to the payload, accesses need to be atomic
  - Synchronizes with relationship between writer and reader
Ingredients for lock free algorithms

• Data structure specific challenges

• Re-claiming memory
  • Readers continue to reference an entry in the data structure even after the delete
  • Memory cannot be ‘freed’ immediately after ‘delete’
  • Delete – Remove the reference to memory/entry
  • Free – Returning the memory/entries to free pool
  • Mechanisms are required to identify when to ‘free’ the entry/memory
Design

- Atomic Operations – relying on 32b operations
- Memory orderings – working with C11 atomic functions
  - Payload 1 – key (stored in key store)
  - Guard 1 – pdata (stored in key store)
  - Payload 2 - current signature and alternate signature (stored in bucket entry)
  - Guard 2 – Index to {key, pdata} entry in the key store (stored in bucket entry)
Design – Ordering Mem Operations – Hash Add/Lookup

Writer

Key, pData (K, D)

KS

Sig, Key-Index (S, I)

BE

Reader

Store - Rel

Store

Store - Rel

Load - Acq

Load

Load - Acq

Load

Non-Atomic

Atomic

B Synchronizes with C

A Synchronizes with D

Notice that A and D need not be atomic operations

K

a

b

B

C

D

S

I

D

K
For hash update ‘Key Index’ can’t be used as the guard. This forces us to have 2 sets of payloads and guards. If hash update is changed to allocate a new key entry from key store, memory orderings can be simplified.
Design – Data Structure Specific Challenge

- Hash add can move entries to their alternate positions
- Due to concurrent adds, reader might not find the entry even though it is present
Design – Data Structure Specific Challenge

• Solution uses a global counter

• Counter indicates to the reader that the table has changed

Writer

Global Counter

C

Increment

Move to primary

Entry is moved from secondary bucket

Primary Bucket

P

Secondary Bucket

S

Reader

Fails

Entry is in secondary bucket

Fails

Entry is NOT in secondary bucket
Counter changed, repeat the lookup

Read

Lookup

Entry present but not moving – entry is found immediately

Entry present but moving – reader has to chase it

Entry not present – Repeats till the move stops – Can be improved by using bucket counter
Q & A
Backup
Thread Quiescent State (TQS) - Any place in the code where the thread does not hold a reference to shared memory.
Free the memory after every thread has gone through at least 1 quiescent state.

Won’t reference the ‘deleted’ memory.

Delete an entry from data structure D2.

Can’t free memory during this period as threads are still referencing it.

Remove the reference to shared memory.

Delete the memory after every thread has gone through at least 1 quiescent state.
Re-claiming Memory – rte_tqs library

• rte_tqs library
  • Provides the ability to check if a set of reader threads have entered at least 1 quiescent state

• Goals
  • Provide flexibility to check the quiescent state
    ➢ Single data structure, a group of data structures or any application defined granularity
  • Ability to check the quiescent state of a given set/all of readers
  • Ability to check quiescent state synchronously
  • Ability to check quiescent state asynchronously
Current Issues

• Performance scaling with lock-free changes