Transactional Memory

(design considerations primer)

Vasileios Trigonakis

TM Correctness – Opacity

- **Serializability**
  - equivalent to some serial execution
- **Consistent memory view**
  - even aborted transactions have to observe a consistent view of memory

T: 1 / (y - x)

Transactional reads

- **Visible reads**
  - Tx is reading object O
    - → other txs can observe that Tx read O
- **Invisible reads**
  - Tx is reading object O
    - → other txs cannot observe that Tx read O
- **Multiversioning**
  - Tx is reading object O
    - → Tx finds the „correct“ version of O

Visible reads – Implementation

- `tx_read(m)`: Inform the other txs that you read `m`

```
tx_read(m)
   lockm = stm_find_lock(m);
   if(!stm_read_lock(lockm))
      tx_abort();
   stm_read_set_add(m);
   return *m; //finally read m
```

Invisible reads – Implementation

- `tx_read(m)` : detect whether your reads are still valid

```
tx_read(m)
    for each l in stm_read_set():
        if (l.version != stm_get_version(l.addr))
            tx_abort();
    versionm = stm_get_version(m);
    stm_read_set_add(m, versionm);
    return *m;
```

Multiversioning – Implementation

not very suitable for procedural languages (e.g., c)

- `tx_read(m)` : Your transaction has a version assigned (in the beginning of the tx), find the value for that version

`tx_read(m)`

```
my_v = stm_curr_tx_version();
(val, version) = stm_read_version(m, my_v);
if (version != my_v)
   tx_abort(); // could not find correct v
stm_read_set_add(m, versionm);
return *m;
```

If we keep the full history of objects, read-only transctions can never be aborted!

Transactional writes

- **Eager writes**
  - grab the locks for writes directly
- **Deferred (Lazy) writes**
  - grab all the locks together on commit time

- **Undo log**
  - write directly to memory, keep the old values
- **Buffered writes**
  - keep the new values in log, do not write to mem

Eager writes – Implementation

- `tx_write(m)`: grab the lock on time

```
tx_write(m, val)
    lockm = stm_find_lock(m);
    stm_write_lock(lockm);
    stm_write_set_add(m);
    // write or just log the write
```

Lazy writes – Implementation

- `tx_write(m)`: just log the write
  synchronize on commit
  (cannot write the value directly to memory)

`tx_write(m, val)`

```
stm_write_log_add(m, val);
```

Undo log – Implementation

- `tx_write(m)`: write the value to mem

  (does not work with lazy writes)

```
tx_write(m, val)
  // eager write synchronization
  val_cur = stm_get_val(m);
  stm_write_log_add(m, val_cur);
  *m = val;
```

Buffered writes – Implementation

- `tx_write(m)`: log the write, write the val to memory on commit

```
tx_write(m, val)
  // eager synchornization or not
  stm_write_log_add(m, val);
```

On commit

- **You *might* need to**
  - do synchronization for writes
  - validate reads
  - persist writes
  - persist memory frees
  - cleanup metadata
    (locks, read/write sets, logs, allocations, etc.)
  - …

On abort

- **You *might* need to**
  - revert memory values (writes)
  - cleanup metadata
    (release locks, empty read/write sets, etc.)
  - revert memory allocations
  - …

Contention management

Who is going to be aborted on a conflict?

- **Polite**: abort self (the one that detected conflict)
- **Aggressive**: abort other(s)
- **Greedy**: abort newer
- …