Software Transactional Memory

6.5060 Talk
Luka Govedič

What is a Transaction?

- A **transaction** is a sequence of operations that appears **atomic** (indivisible) to all outside observers, meaning it either completed in full or it didn't complete at all.
- Transactions are **serializable**, meaning that after they execute, the system remains in a state that is the same as if the transactions all executed one after the other in some order, and all future transactions always see the transactions executing in the same order.

Motivation for transactions

- Atomic operations in a concurrent environment
- Preventing deadlock
- Recovering from faults & adversarial scheduling
  - Avoiding priority inversion and convoying
- Composability of atomic actions

Talk Overview

- Synchronization recap
  - Mutual exclusion
  - CAS (Compare-and-Swap)
- Transactions
- Transactional memory in hardware
- Software Transactional Memory (STM) implementation

Synchronization recap

Q: Why do we care about atomicity?

A: Preserve invariants in our data structure.

Q: What's the invariant in a singly-linked list?

A: `head` always points to the first element.

```
// a simple example of insertion
// into a singly linked list

template <class T>
void LinkedList<T>::insert(T value) {
    auto *tmp = new Node<T>{value};

    tmp->next = this->head;
    this->head = tmp;

}
```

Mutual exclusion

- Achieved using locks
- Only allows one process at a time
- Can lead to:
    - Deadlock
    - Priority inversion
    - Convoying

```
// a simple example of insertion
// into a singly linked list

template <class T>
void LinkedList<T>::insert(T value) {
    auto *tmp = new Node<T>{value};
    this->mutex->lock();
    tmp->next = this->head;
    this->head = tmp;
    this->mutex->unlock();
}
```

Compare-and-Swap

- Uses hardware support for atomic instructions
- Not mutually exclusive
- Correctness can be hard to prove
- Can only provide atomicity on a single (or double) word(s)

```
// a simple example of insertion
// into a singly linked list

template <class T>
void LinkedList<T>::insert(T value) {
    auto *tmp = new Node<T>{value};
    do {
        tmp->next = this->head;
    } while (!CAS(&this->head,
                  tmp->next, tmp));
}
```

Transactions

- Not mutually exclusive
- Conflicts managed by the transaction manager instead of user
- Might abort and be restarted
- Can introduce higher overheads
- (language support from [4])

```
// a simple example of insertion
// into a singly linked list

template <class T>
void LinkedList<T>::insert(T value) {
    auto *tmp = new Node<T>{value};
    atomic {
        tmp->next = this->head;
        this->head = tmp;
    }
}
```

Managing transactions

- Transactions either COMMIT or ABORT
  - If they ABORT, they have to be restarted
- Goal: achieve serializability and atomicity -> avoid/prevent conflicts
  - Serial order = order of transactions COMMITing
  - Two methods: pessimistic (locking) and optimistic
- Static transactions: set of memory locations accessed is known ahead of time

Serializing transactions

- Pessimistic concurrency control
  - Each transaction locks the memory locations it needs to access
  - For static transactions, deadlock avoided by ordering and locking all at the start
  - For dynamic transactions, either abort after timeout or abort if deadlock cycle detected
- Optimistic concurrency control
  - Each transaction keeps track of the memory locations accessed
  - Before commit, the manager validates that the transaction doesn't meaningfully overlap with any transactions that have already committed but only did so after it started

Hardware Transactional Memory [2]

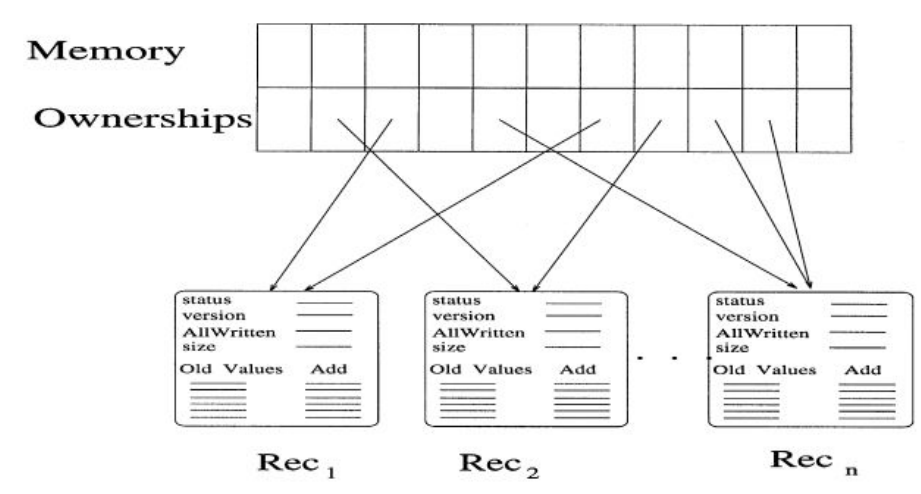
- Supports memory and transaction operations
  - Read (load), write (store), etc.
  - Commit, abort, validate
- Matches or outperforms other synchronization methods
- Implemented as an extension of the cache coherency protocol
  - Does not support more locations than fit in the L1 cache
  - [3] optimizes and extends the scheme to support transactions of arbitrary size

Software Transactional Memory

- “Supports multiple changes to its addresses [via] transactions”
- Explicit support for (static) transactions
  - A static transaction is defined by its dataset and deterministic transition function
- “Helping” methodology to help the owner of data one needs complete and release the data
- Inferior performance to other synchronization techniques

STM Implementation

- Each process holds a **record** to keep track of the transaction it's currently executing.
- For each location in transactional memory, we record the process that owns it
- Helping policy: if a transaction can't acquire a piece of memory, the process will instead execute the owner of the transaction of that address



Implementation

```
StartTransaction(DataSet)
  Initialize(Rec_i, DataSet)
  Rec_i↑.stable = True
  Transaction(Rec_i, Rec_i↑.version, True)
  Rec_i↑.stable = False
  Rec_i↑.version++
  If Rec_i↑.status = Success then
    return(Success, Rec_i↑.OldValues)
  else
    return Failure

Initialize (Rec_i, DataSet)
  Rec_i↑.status = Null
  Rec_i↑.AllWritten = Null
  Rec_i↑.size = |DataSet|
  for j = 1 to |DataSet| do
    Rec_i↑.Add[j] = DataSet[j]
    Rec_i↑.OldValues[j] = Null
```

```
Transaction(rec, version, IsInitiator)
  AcquireOwnerships(rec, version)
  (status, failadd) = LL(rec↑.status)
  if status = Null then
    if (version ≠ rec↑.version) then return
    SC(rec↑.status, (Success, 0))
  (status, failadd) = LL(rec↑.status)
  if status = Success then
    AgreeOldValues(rec, version)
    NewValues = CalcNewValues(rec↑.OldValues)
    UpdateMemory(rec, version, NewValues)
    ReleaseOwnerships(rec, version)
  else
    ReleaseOwnerships(rec, version)
    if IsInitiator then
      failtran = Ownerships[failadd]
      if failtran = Nobody then
        return
      else
        failversion = failtran↑.version
        if failtran↑.stable
          Transaction(failtran, failversion, False)
```

```
AgreeOldValues(rec, version)
  size = rec↑.size
  for j = 1 to size do
    location = rec↑.Add[j]
    if LL(rec↑.OldValues[j]) = Null then
      if rec↑.version ≠ version then return
      SC(rec↑.OldValues[j], Memory[location])

UpdateMemory(rec, version, newvalues)
  size = rec↑.size
  for j = 1 to size do
    location = rec↑.Add[j]
    oldvalue = LL(Memory[location])
    if rec↑.AllWritten then return
    if version ≠ rec↑.version then return
    if oldvalues ≠ newvalues[j] then
      SC(Memory[location], newvalues[j])
  if (not LL(rec↑.AllWritten)) then
    if version ≠ rec↑.version then return
    SC(rec↑.AllWritten, True)
```

```
AcquireOwnerships(rec, version)
  transize = rec↑.size
  for j = 1 to size do
    while true do
      location = rec↑.add[j]
      if LL(rec↑.status) ≠ Null then return
      owner = LL(Ownerships[rec↑.Add[j]])
      if rec↑.version ≠ version return
      if owner = rec then exit while loop
      if owner = Nobody then
        if SC(rec↑.status, (Null,0)) then
          if SC(Ownerships[location], rec) then exit while loop
      else
        if SC(rec↑.status, (Failure, j)) then return

ReleaseOwnerships(rec, version)
  size = rec↑.size
  for j = 1 to size do
    location = rec↑.Add[j]
    if LL(Ownerships[location]) = rec then
      if rec↑.version ≠ version then return
      SC(Ownerships[location], Nobody)
```

Conclusion

- Transactions allow arbitrary atomicity and composability
- They perform well in faulty environments
- They also bring a considerable overhead

References

1. Software transactional memory
2. Transactional Memory: Architectural Support for Lock-Free Data Structures
3. Hardware Transactional Memory
4. Language Support for Lightweight Transactions