Revisting Condition Variables and Transactions

Victor Luchangco and Virendra Marathe
Oracle Labs
Transactional Memory

- Provide atomicity and isolation
  - encapsulate abstract atomic operations
  - replace locking (monitors)
- But transaction memory is not a panacea
  - prohibits some common synchronization patterns
    - exchangers
    - producer-consumer queues
    - condition synchronization

Can we extend transactions to cover these patterns?
Condition variables

• Avoid busy-waiting
• Developed for monitors (Hoare 1974)
  • refined in Mesa (Lampson, Redell 1980)
• Built-in support in Java
• Operations
  • wait
  • notify
  • notifyAll
• Beware of lost notifications
Isolation can be limiting
Isolation can be limiting
Isolation can be limiting
Isolation can be limiting

Committed transaction is not undone!
Job processing with transactions

• Client threads separate from database threads
  • pre- and post-processing must be in separate transactions from transaction that accesses database
  • requires communication between concurrent transactions

• Entire transaction should abort if any part of it aborts
Transaction communicators (PPoPP 11)
Relaxing isolation to enable cooperation

- Special objects for inter-transactional communication
  - compromises transactional isolation
  - txcomm fields (read/write communicators)
- Updates to communicators visible to concurrent txns
  - even before updating txn commits (even if it may abort)
  - communicator accesses require synchronization
- Introduces dependencies between transactions
  - txn depends on txns whose effects it observes
  - txn must not commit before txns it depends on commit
  - cyclic dependencies allowed
    - transactions must commit together (or else all abort)
Communicator-isolating transactions

- Communicators enable races among transactions
  - txns must synchronize accesses to use effectively
- Special txns that isolate even communicator accesses
  - avoids need to use locks to synchronize
  - `txcommatomic` block
- Can be nested within ordinary transactions
  - effects on communicators within CIT not visible until CIT commits
  - CITs cannot be flattened into ordinary txns
Producer-consumer queue from read-write communicators

class ProducerConsumerQueue {
    txcomm Node head = null;
    txcomm Node tail = null;

    public void produce(Object data)...
    public Object consume()...
}

class Node {
    txcomm Object data;
    txcomm Node next;

    public Node(Object data) {
        this.data = data;
    }
}
Producer-consumer queue from read-write communicators

```java
public void produce(Object data) {
    Node myNode = new Node(data);
    txcommatomic {
        if (tail == null) {
            head = tail = myNode;
        } else {
            tail.next = myNode;
            tail = myNode;
        }
    }
}

public Object consume() {
    Node node;
    while (true) {
        txcommatomic {
            if (head != null) {
                node = head;
                if (head.next == null) {
                    head = tail = null;
                } else {
                    head = head.next;
                }
            } else {
                head = null;
            }
        }
        return node.data
    }
}
```
Job-processing application

class Job {
    Object getKey() ...
    void postprocess(Object resp) ...
}
ProducerConsumerQueue reqQueue;

class ReqResp {
    txcomm Object request;
    txcomm Object response;

    public ReqResp(Object req) {
        request = req;
        response = null;
    }
}


Job-processing application

// client thread method
public void handleJob(Job j)
    atomic {
        ReqResp rr = new ReqResp(j.getKey());
        reqQueue.produce(rr);
        while (rr.response == null);
        j.postprocess(rr.response);
    }

// database thread
public void processNextJob()
    atomic {
        ReqResp rr = reqQueue.consume();
        rr.response = datastore.get(rr.request);
    }
Problems with communicator-based solution

- Busy-waiting
- Unnecessary dependencies

```java
public Object consume() {
    Node node;
    while (true) {
        txcommatomic {
            if (head != null) {
                node = head;
                if (head.next == null) {
                    head = tail = null;
                } else {
                    head = head.next;
                }
            } else {
                head = head.next;
            }
        }
        return node.data
    }
}
```
Ordinary producer-consumer queue with condition synchronization

synchronized void produce(Object data) {
    Node myNode = new Node(data);
    if (tail == null) {
        head = tail = myNode;
    } else {
        tail.next = myNode;
        tail = myNode;
    }
    notify();
}

synchronized Object consume() {
    Node node;
    while (true) {
        if (head != null) {
            node = head;
            if (head.next == null) {
                head = tail = null;
            } else {
                head = head.next;
            }
            return node.data;
        } else {
            wait();
        }
    }
}
Transactional condition synchronization

- Conditional critical regions (Harris, Fraser 2003)
  - don’t execute transaction until condition is satisfied
- retry (Harris, et al., Haskell STM 2005)
  - abort transaction
  - retry from the beginning when something relevant changes
- “Punctuation” (Smaragdakis et al., 2007; Dudnik, Swift 2009)
  - commit partial transaction
  - upon notification, resume execution in new transaction

- Our proposal: xCondition
  - allow waiting inside transaction
  - enforce dependencies
**xCondition**

- Preserves spirit of condition variables
  - `txwait`, `txnotify`, `txnotifyAll`
  - waiter suspends execution, resumes upon notification
  - allows communication between waiter and notifier

- Compromises transaction isolation
  - waiter sees communication event (the notification)
  - dependency of waiter on notifier
    - waiter does not commit unless notifier commits (*)

- Compatible with transaction communicators
  - notifier typically establishes a condition desired by waiter

- Extra requirement to avoid lost notifications
  - if waiter aborts after being notified, it “forwards” the notification
Synchronized access to xCondition

- Traditional condition variables are always accessed within a critical section
  - lock released after adding thread to wait list, obtained before resuming thread execution
  - helps avoid lost notifications
- Use within communicator-isolating transaction (CIT)
  - this is the only use we support

```java
txcommatomic {
  canProceed = true;
  X.txnotify();
}
```

What does this mean in the context of a CIT?

```java
txcommatomic {
  if (!canProceed) {
    X.txwait();
  }
}
```
Semantics of txwait within communicator-isolating transactions

• Desiderata
  • preserve spirit of condition synchronization
  • preserve isolation of CIT
  • avoid establishing unnecessary dependencies
  • ensure notifications are not lost

• “retry-on-txwait”, with a twist
  • abort CIT, suspend execution until notification
  • if CIT again calls txwait, the notification is forwarded
    • at most one notification “consumed” per successful CIT
Producer-consumer queue from read-write write communicators

class ProducerConsumerQueue {
    txcomm Node head = null;
    txcomm Node tail = null;
    xCondition xc = new xCondition();

    public void produce(Object data)...
    public Object consume()...
}

class Node {
    txcomm Object data;
    txcomm Node next;

    public Node(Object data) 
        this.data = data;
}
Producer-consumer queue with xCondition

```java
public void produce(Object data) {
    Node myNode = new Node(data);
    txcommatomic {
        if (tail == null) {
            head = tail = myNode;
        } else {
            tail.next = myNode;
            tail = myNode;
        }
        xc.txnotify();
    }
}

Something missing?

```
**xCCondition vs regular condition variable**

```java
public void produce(Object data) {
    Node myNode = new Node(data);
    txcommatomic {
        if (tail == null) {
            head = tail = myNode;
        } else {
            tail.next = myNode;
            tail = myNode;
        }
    }
    xc.txnotify();
}
```

```java
synchronized void produce(Object data) {
    Node myNode = new Node(data);
    if (tail == null) {
        head = tail = myNode;
    } else {
        tail.next = myNode;
        tail = myNode;
    }
    notify();
}
```
public Object consume() {
    Node node;
    txcommatonic {
        if (head != null) {
            node = head;
            if (head.next == null) {
                head = tail = null;
            } else {
                head = head.next;
            }
            return node.data
        } else
            xc.txwait();
    }
}

synchronized Object consume() {
    Node node;
    while (true) {
        if (head != null) {
            node = head;
            if (head.next == null) {
                head = tail = null;
            } else {
                head = head.next;
            }
            return node.data;
        } else {
            wait();
        }
    }
}
xCondition vs communicators

- Fewer dependencies
- Communicators encapsulate data; xCondition captures communication event
  - notification must be forwarded if notified waiter aborts
- txwait is blocking
  - naïve semantics do not work within txcommatomic blocks
  - although invoked before txnotify, it is ordered after
Summary: xCondition

- Condition variables for use with transactions
  - compromises transaction isolation
  - compatible with transaction communicators
  - alternative to retry, conditional critical sections, punctuation
    - different approaches appropriate for different situations
    - special semantics within communicator-isolating transactions

- Part of larger effort to make common synchronization patterns compatible with transactional programming