Transactional Memories
An Overview

Krishna M. Kavi
Professor and Chairman
Department of Computer Science and Engineering
University of North Texas, Denton, Texas

kavi@cse.unt.edu
www.csrl.cse.unt.edu/~kavi

1. Motivation
2. Traditional models of parallelism
3. Speculation
4. Hardware supported TM
5. Software supported TM
6. Issues and Challenges
Motivation

Consider the following examples

a). for (j=0; j<n; j++)
   hist[j]++;

b). for (j=0; j<n; j++)
   if (local_min(j) < global_min)
      global_min = local_min(j);

Can we consider concurrent implementations using
(i) Multithreading (shared memory)
(ii) MPI (distributed memory)

Shared Memory Model

a). for (j=0; j<n; j++)
   thread_id[j] = spawn_thread(accumulate, j);
   for (j=0; j<n; j++)
      join (thread_id[j]);

void accumulate (j)
{
   ...
   lock(lock_variable);
   hist(j)++;
   unlock(lock_variable);
   exit();
}
**Shared Memory Model**

b). for (j=0; j<n; j++)
    thread_id[j] = spawn_thread(compute_min, j);
    for (j=0; j<n; j++)
        join (thread_id[j]);

```c
void compute_min (j)
{
    ...local_min = min();
    lock(lock_variable);
    if ( local_min < global_min )
        atomic global_min = local_min;
    unlock(lock_variable);
    exit();
}
```

**Optimistic/Speculative Model**

Consider a system that does not use locks to acquire shared data

But, all updates to shared data are still atomic

```c
void compute_min (j)
{
    ...local_min = min();
    if ( local_min < global_min )
        atomic global_min = local_min;
    exit();
}
```

Is there a problem with this?
But, consider the other example.

```c
void accumulate (j)  
{...
atomic hist(j)++;  
exit();
}
```

Are we gaining anything?  
Are we losing anything?

If working on local copies, avoid contention on memory bus.

Consider a different application that uses Speculative execution

```c
while (continue_cond)  
{     ......  
x= hash[index1];  
hash[index2] = y;  
......}
```

Can we execute the loop in parallel?  
Is there a dependency between index1 and index2?  
And can we determine this dependency at compile time?
Optimistic/Speculative Model

Speculative execution

<table>
<thead>
<tr>
<th>Processor1</th>
<th>Processor2</th>
<th>Processor3</th>
<th>Processor4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Epoch 1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = hash[index1];</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash[index2] = y;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Pseudo-Code

(b) TLDS Execution

Parallelism and issues

In most parallel programming languages we use:
- mutual exclusion
- monitors
- or message communication

To communicate and synchronize among computations:

The key is: shared data cannot be accessed outside critical sections

And: must use the same order on locks to avoid deadlocks
Atomic Transactions

Atomicity of actions
When two concurrent **computations** interact
we need to assure atomicity of the interactions
Note that we are not talking about data directly but implicitly

Concurrency in database is based on atomic transactions

Consider booking airline tickets
Each "agent" works on a copy of data
and either commits or retries the results

---

Atomic Transactions

Atomic Transactions can cover both
mutual exclusion
speculative execution

However, still need programming discipline

Thread 1
atomic{
    while (!flagA);
    flagB = true:
}

Thread 2
atomic{
    flagA = true;
    while(!flagB);
}
Atomic Transactions

The code is not correct since the loops do not terminate unless the respective flags are true before starting the blocks.

So, one cannot assume that you can mechanically replace mutual exclusion with atomic transactions.

Still need to make sure that shared objects are accessed only by atomic transactions.

Implementing Atomicity and Speculation

1. Need to be able to rollback -- changes must be buffered
2. Need to recognize failures
   - Maintain read and write sets with each transaction
   - Cache coherency extensions
   - Version numbers
3. Need to differentiate between values that are speculated and regular values
   - Additional instructions such as speculative-load/store
   - Start transaction, end transaction
4. Determine the order of committing/write-back
   - Program order
   - One thread at a time (no specified order)
Some Architectural Background

How is atomicity achieved in current systems?

Implementation of locks in hardware
- Test&Set instructions (and Fetch&Add type instructions)
- Load Linked and Store Conditional Instructions
- Memory Barriers (or fences)

Performance Issues
- Where to locate locks (local cache?)
- Tree and queue locks
- Shadow locks

Load Linked and Store Conditional

LL remembers the memory address of the load.
If some other access (either on the same processor or another processor) to the same address is made, the remembered address is lost
On a SC, if the remembered address is the same as that of SC, store will be successful

Let us take a simple example of implementing the test and set using these instructions.

Try:
- Move R3, R4  ; Move value to be exchanged
- LL R2, 0(R1)  ; load linked to memory
- SC R3, 0(R1)  ; Store conditional to the same memory

location
- BEQZ R3, Try  ; if unsuccessful, try again
- BNEQZ R2, Try  ; if a nonzero value was read on LL try again
  ; lock is held by someone else
Cache Coherency

Why do caches cause a problem in multicore systems?

Snooping Protocols -- MESI (and MOESI)

Each cache line will be associated with one of 4 states.
Invalid (I); Modified (M); Exclusive (E) or Shared (S)

Before modifying data, if it is shared, need to either send an Invalidation message, or send updates to other caches

---

Read Miss: That is, the data is not available in local cache or the local data is Invalid.

Cases:
- No other cache has the data. Missing data is fetched from main memory. The local copy is set to Exclusive state.
- One or more caches have copies of the data (in Shared) One of the cache supply the data (or main memory supplies the data), the local copy is set to Shared.
- Only one other cache has a copy (either in Exclusive or Modified state). If in M, data is written back to memory, data is supplied to requesting cache; all copies are set to Shared If in E, data is supplied to requesting cache and all copies are set to Shared
**Cache Coherency**

Write Miss: That is, the address to which you are writing to is not in local cache, or the line is Invalid.

Cases:

- No other cache has a copy of the address. Main memory supplies the data, local cache copy is set to Modified and write can proceed.

- One or more caches have copies in Shared state. One of the caches or main memory supplies the data, all other copies are set to Invalid; local copy is set to Modified and write can proceed.

- Only one other cache has the data in Exclusive or Modified state. If in Modified state, the data is written back to main memory, the remote copy is set to Invalid. Data is supplied to local cache (either by main memory or remote cache) and the local cache is set to Modified. If in Exclusive state, the remote copy is set to Invalid, data is supplied to local cache (either by main memory or remote cache) and the local cache is set to Modified.
Cache Coherency

Write Hit: The address to which you are planning to write is in local cache (and not in Invalid state).

Cases:
- The local copy is in Shared State. An invalidation message is broadcast on the bus; all remote copies are set to Invalid; local copy is set to Modified
- Local copy is in Exclusive state. Only copy hence write can proceed; local copy is set to Modified
- Local copy is in Modified state; only copy and write can proceed; local copy remain in Modified state

When the shared memory is physically distributed, use Directory based protocols

In addition to states with Cache line, need a directory
Sequential Consistency

**Coherency vs Consistency**

A consistent memory means that the entire address space as seen by a program (that may be running as multiple processes on a single processor or on multiple processors) will be consistent.

What does consistent mean?

All processes/processor will see the entire memory containing the same values at all times.

Coherency is related to reads/writes of a single data item. If every single data item is coherent, then the memory is consistent.

---

What is a **correct parallel program**? --- memory consistency is required

Consider the following example. Assume $a = b = c = 0$ initially.

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1</td>
<td>b = 1</td>
<td>c = 1</td>
<td></td>
</tr>
<tr>
<td>print(b,c)</td>
<td>print(a,c)</td>
<td>print(a,b)</td>
<td></td>
</tr>
</tbody>
</table>

There are $2^6 = 64$ different combinations of outputs. Which of them are legal?

Sequential Consistency:

“The result of any execution is the same as if the operations of all processors were executed in some sequential order; and the operations of each individual processor appears (to all processors) in this sequence as specified by the program order.”
Sequential Consistency

Two conditions are required by the definition for sequential consistency

A). **Maintaining program order.** That is, the changes to memory made by any single program running on a single processor must maintain the order of changes as they appear in the program code.

B). **Maintain sequential ordering among all operations.** That changes to all memory locations should be sequentialized and all processors must see the same sequential ordering. **Note all processors must see the same order.**

Assuring these two properties in modern (out of order and multi-core) processors is very difficult and can cause loss of performance.

This led to relaxed or weak consistency models

But weak models require programmer to take more responsibility and use locks, barriers or other types of atomic actions and ordering among computations.

Hardware will only assure consistency at the lock acquires and/or lock releases only.

This brings us back to the problem with parallel programming And need for newer models
Transactional Memory Issues

Transactional Memory Semantics

Operational Semantics
Transactions should behave as if the execution uses a single lock

Note that if every computation is a transaction, this semantics is sufficient

But, if program include both (atomic) transactional and non-transactional computations, then we can still have race conditions
Transaction Memory Semantics

Weak and Strong atomicity (isolation) properties

Transactions on objects are “linearized”

Strong semantics requires (or automatically convert) all operations to behave as atomic transactions

Weak semantics only applies to atomic transaction (and race conditions may exist with non-transactional operations)

Nested Transactions

What happens if an inner transaction aborts?

Flattened transactions.

int x = 1;
atomic {
x = 2;
atomic flatten {
x = 3;
abort
}
}

x = 1 always
Aborting any nested transaction causes abortion of outer transactions
Transactional Memory Semantics

**Closed transactions.**

```c
int x = 1;
atomic {
    x = 2;
    atomic closed {
        x = 3;
        abort
    }
}
```

- `x=2` if outer transaction commits
- Aborting inner transactions will not cause outer transactions to abort
- And inner transactions can only commit if outer transaction commits.

**Open transactions.**

```c
int x = 1;
atomic {
    x = 2;
    atomic open {
        x = 3;
        abort
    }
}
```

- `x=3` even if outer transaction aborts
- Inner transactions can commit even if outer transactions fail.

---

- Flattened transactions are easy to implement (but not good since aborted library transactions will abort your programs)
- Closed transactions lead to higher retries and overheads
- Open transactions do not guarantee Sequential consistency
Transactional Memory Design Issues

Granularity
Object granularity -- typically in Software TM

Word or cache block granularity
-- typically in hardware TMs

Performance impacts
need to maintain meta data with each data item
to track all transactions that access them
read sets and write sets

Direct or Indirect updates
Direct -- modify an object directly.
must be able undo modifications
Deferred -- modify a copy
must ensure local reads see the updates
discard local copies if aborted

If conflicts are rare direct (or in-place) updates are efficient
What if private copies cause buffer overflows
or need to write caches copies
Transactional Memory Design Issues

Conflict Detection
- Early detection -- detection when opened
  - assuming instructions exist to open/close
  - OpenForReading (..)
  - OpenForWriting
- Late detection -- detect on commit

Performance issues
- Wasted computation
- Meta data needed to track read sets and write sets

In this example, in part (a) we will detect the conflict using either eager or lazy detection.

In part (b), only eager detection detects the conflicts. Lazy detection allows both transactions commit (that is the ReadX of T1 happened before the WriteX of T2).
Transactional Memory Design Issues

In early or eager conflict detection, T1 may be aborted twice; ReadX may conflict with WriteX of T2 (if T2 commits before T1) and ReadY may conflict with WriteY of T3 (if T3 commits first).

In lazy or delayed conflict detection, T1 will only be aborted once (when it tries to commit at the end).

Another issue is which transaction to abort when a conflict is detected. In this example, when the conflict between T1 and T2 is detected, we can either abort T1 or T2. If T2 is aborted, then we need to again decide between T1 and T3 when the second conflict between T1 and T3 is detected (2 transactions are aborted).

If we abort T1 instead of T2, then both T2 and T3 can commit (only one transaction aborted).
Transactional Memory Design Issues

Read and Write sets
Each Transaction maintains its own read sets
or makes the read set public

Maintain read/write sets with objects
notify readers of conflicts
read sets may be very large

Software TM Implementations

General Outline
Depending on the granularity need to maintain
indirection to permit atomicity of commit
need to maintain buffers for updates
need to track read and write lists for validation
Object TM Implementations

Figure 3: Transactional object structure

Figure 4: Opening transactional object after recent commit

Figure 5: Opening transactional object after recent abort

Object TM Implementations
Hardware TM Implementations

TCC (Stanford)

LogTM
Implicit Transactions in Kilo

Views computations as regions.

Check-pointing and rollback mechanisms are used to either commit the computations of a region, or redo.
SDF and Transactions

Scheduled Dataflow Architecture divides a computation into three phases:
- Preload: When a thread receives all its inputs, the input data is loaded into the thread’s registers.
- Execute: The computation uses data in registers to complete its computations.
- Post-store: The results of the computation are transferred to memory.

This permits us to view a thread as a transaction and complete post-store to commit, or ignore post-store to abort.

A Programming Example

Pre-Load
- LOAD RFP\(\#2\), R2 / load A into R2
- LOAD RFP\(\#3\), R3 / load B into R3
- LOAD RFP\(\#4\), R4 / load X into R4
- LOAD RFP\(\#5\), R5 / load Y into R5
- LOAD RFP\(\#6\), R6 / frame pointer for returning first result
- LOAD RFP\(\#7\), R7 / frame offset for returning first result
- LOAD RFP\(\#8\), R8 / frame pointer for returning second result
- LOAD RFP\(\#9\), R9 / frame offset for returning second result

Execute
- ADD R8, R2, R13 / compute A+B, Result in R11 and R13
- ADD R6, R10 / compute X+Y, Result in R10
- SUB R8, R12 / compute X – Y, Result in R12
- MUL R14, R11 / compute (X+Y)*(A+B), Result in R14
- DIV R15, R14 / compute (X-Y)/(A+B), Result in R15

Post-Store
- STORE R14, R6R7 / store first result
- STORE R15, R8R9 / store second result
SDF Architecture

**Execute Processor (EP)**

**Memory Access Pipeline**

**Synchronization Processor (SP)**

TLS Schema for SDF Architecture

**Architecture Model**
Cache States in TLS

<table>
<thead>
<tr>
<th></th>
<th>SpRead</th>
<th>Valid</th>
<th>Dirty (Exclusive)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>E/M</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Sp.R Ex</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sp.R Sh</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Implementation

Note the Commit order can either be
program order (epoch numbers to define which thread
commits first)
or no order to implement mutual exclusion

Although currently we only implement lazy conflict detection
(when a thread is ready to commit)
SDF implementation can be extended to implement
eager detection
in place update
(at present we use thread registers as buffers)

Conclusion

✓ TM presents a new approach to writing concurrent programs
✓ TM model can be used for both “atomicity” and “speculation”
✓ Unless all computations are viewed as (atomic) transactions, race
  conditions can still exist
✓ Implementations must carefully trade-off different design choices
to achieve desired levels of performance
✓ Need to tools to translate programs into transaction models
✓ Or design new languages with TM as the concurrency model
Key References


Memory Barriers and Fences

Consider the following code segment

For (i = 1, i <N+1, i ++){
    WaitFlag (Flag[j][i]);
    AcquireMemBar;
    DoWork(X[i]);
    ReleaseMemBar;
    SetFlag(Flag[j+1][i]);
}

Notice AcquireMemBar ensures that all previous loads (including lock/flag acquires) complete before any subsequent memory operations

Notice that here we are using Array of locks. Here we have multiple locks, one for processor -- each processor is setting a different flag.

If we use Software Pipelining ideas, we can rewrite the code to acquire locks ahead

For (i =1; i <N, i ++) {
    AcquireMemBar;
    WaitFlag(Flag[j][i +1]);
    DoWork(X[i]);
    ReleaseMemBar;
    SetFlag(Flag[j+1][i]);
}