Hardware and software implementation for transactional memory

Alain Leblanc

Transactions

(From Sandhya Dwarkadas lost lecture)

- Modify multiple data items potentially at multiple locations/by multiple processes as a single atomic operation
- Transaction properties (ACID) –
  - Atomic – happens indivisibly to the outside world
  - Consistent – does not violate system invariants – must hold before and after but not necessarily during
  - Isolated (or serializable) – refers to multiple simultaneous transactions – the final result must appear as if each transaction occurred in some sequential order
  - Durable – once committed, the results become permanent – no failure can undo the results

Summary

- Why use transactional memory
- The 10000 foot view of transactional memory
  - Conflict detection, validation, contention management
- The early days – architecture support
- The recent days – software implementation

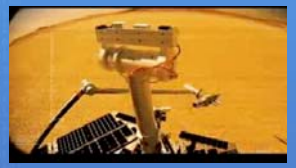
Why use transactional memory

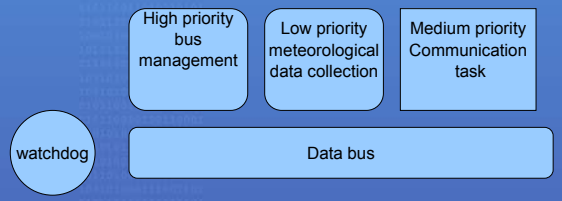
- Implement parallel programs without using locks.
- Locks have some issues:
  - Priority inversion
  - Convoying
  - Deadlock

Priority inversion

- Occurs when a lower priority process if preempted while holding a lock needed by a high priority process
- In the “best case” it makes an important job wait.
- In the worst case we lose data for which we paid lots of money.

Bad priority inversion: Mars Pathfinder

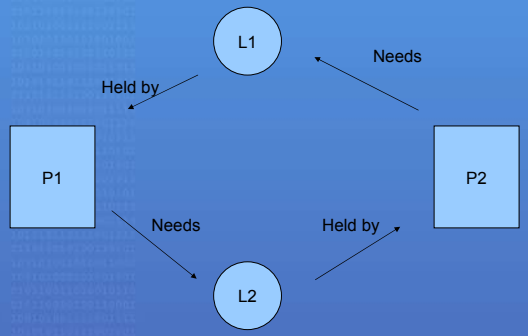




Convoying

- Situation where the processes wait in line for the process ahead in the line to finish some task

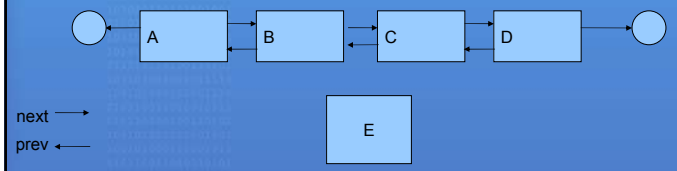
deadlock



Transactional memory

- Want multiple processes to access and possibly (try to) write to the same memory location concurrently.
- Lock free
- Illusion of atomicity on multiple memory locations
- Guarantee that memory as seen by each processor always in a consistent state

Example of use: doubly linked list



P1 wants to insert E between B and C:

B.next=E
C.prev = E
E.prev = B
E.next=C

P2 want to delete A

B.prev = null

P3 want to delete C

B.next=D
D.prev=B

Definition of transaction (again)

- Finite sequence of machine instructions executed by a single processor that include read and write to memory where
  - Transactions appear to execute serially:
    - Steps of one transaction never appear to be interleaved with the steps of another
    - Committed transactions never appear for different processors to execute in different order
  - Transactions execute atomically: at the end all the changes are written in memory (COMMIT), or they are all discarded (ABORT)

In practice

- Programmer marks where transactions start and end
- Transaction are short in time and involve relatively few memory locations.

Operations in TM

- Validation (not always necessary)
- Conflict detection
- Contention management

Conflict detection

- Recognize when committing two transactions would break the assumptions, which is when
  - Two transactions want to write to the same memory locations
  - One transaction wants to write to a memory location that has been read by another
- When a conflict occurs, one of the transaction may need to be aborted

Contention management

- How to decide which thread is aborted when a conflict occurs
- Usually not implemented as a separate process. More of a policy that everyone must follows
- Badly designed contention management can lead to bad performance, starvation, livelocks.

Some contention management algorithms

- Among the many contention management policies (from Scherer & al, 2004)
  - Aggressive
  - Polite
  - Randomized
  - Karma
  - Eruption
  - Killblocked
  - Kindergarten

Validation

- Aim to prevent a transaction that would be aborted to execute with inconsistent data. For instance:
  - Transactions T1 and T2 are in conflict.
  - T1 commits, T2 does not abort until it tries to commit.
  - T2 may be using some data read before T1 commits, some read after.
- May not be necessary as a separate step with aggressive contention management.

Transactional memory Implementations

- Can be implemented at the hardware or software level
- At the software level can be implemented at various granularity: word level, object level, etc.

Example hardware implementation (from Herlihy and Moss)

- Implemented by extension to multi-processor coherence cache
- Can change status of multiple cache lines in one bus cycle.
- Separate caches for transactional and non-transactional operations.
- Use “aggressive” snooping: a line is not written back to memory unless a cache is full.

Memory Instructions

- Three types of memory instructions
  - Load Transactional (LT): read into private register
  - Load Transactional exclusive (LTX): read into private register with option to modify
  - Store Transactional (ST): tentatively write to memory.

Sets

- Read set: set of locations read by LT
- Write set: set of locations accessed by LTX or ST
- Data set: Union of read set and write set.

Cache line states and tags

Name	Access	Shared?	Modified?
INVALID	none	—	—
VALID	R	Yes	No
DIRTY	R, W	No	Yes
RESERVED	R, W	No	No

Table 1: Cache line states

Name	Meaning
EMPTY	contains no data
NORMAL	contains committed data
XCOMMIT	discard on commit
XABORT	discard on abort

Table 2: Transactional tags

- Cache line states are used by both caches
- Transactional tags used by the transactional cache

Operations on transactional cache

- During a transaction the cache keeps two copies for each entry involved:
  - XCOMMIT tag: contains last valid value
  - XABORT tag: can be modified during transaction
- On COMMIT:
  - XABORT → NORMAL
  - XCOMMIT → EMPTY
- On ABORT:
  - XABORT → EMPTY
  - XCOMMIT → NORMAL

About XCOMMIT

- Not required for correct operation
- Just prevent line to be written to memory until absolutely necessary.

Bus cycles

Name	Kind	Meaning	New access
READ	regular	read value	shared
RFO	regular	read value	exclusive
WRITE	both	write back	exclusive
T_READ	trans	read value	shared
T_RFO	trans	read value	exclusive
BUSY	trans	refuse access	unchanged

Table 3: Bus cycles

- BUSY sent by process holding a line that is requested by another process
- Will cause process requesting the line to abort.

Processor actions

- A transaction aborts itself when it receives a BUSY signal
- If a transaction has not been aborted after its last instruction, then it can be committed.
- COMMIT/ABORT operations performed by modifying the tags of the relevant cache entries in parallel.

Software implementation

- Software implementation can be ported on many current architectures
- Only requires CAS or some similar atomic operation
- Can use different granularity: word, language object, ...

Example implementation: RTSM

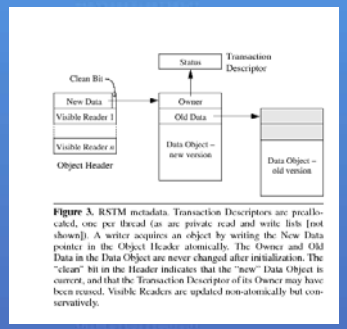
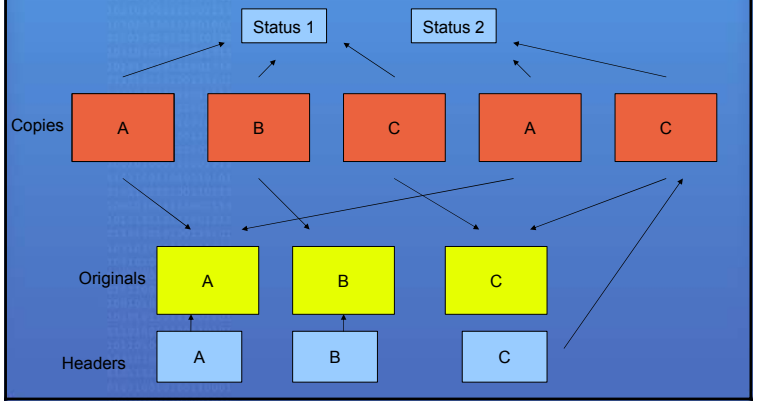


Figure 3. RSTM metadata. Transaction Descriptors are preallocated, one per thread (as are private read and write lists [not shown]). A writer acquires an object by writing the New Data pointer in the Object Header atomically. The Owner and Old Data in the Data Object are never changed after initialization. The "clean" bit in the Header indicates that the "new" Data Object is current, and that the Transaction Descriptor of its Owner may have been reused. Visible Readers are updated non-atomically but conservatively.

- Important concept: visible vs. invisible reader
- All object in a transaction have a pointer to the same status object.
- Each thread using an object creates a local copy
- Status is one of active/commited/aborted

Example implementation (contd)



Visible and invisible readers

- Method for determining rapidly which thread(s) has a transaction currently using a given object.
- Attribute of a thread: each transaction executing on this thread will be visible
- There can be both visible and invisible reader. In RTSM limit of 32 visible readers.
- Implemented as a bit map for each object.
- Separate VALIDATE method not necessary for a visible reader.

Acquiring an object

- A transaction P1 must acquire all its objects some time before committing
- At most one transaction can own an object at any one time
- Acquiring done by reassigning the “New Data” pointer to the local version of the object.
- All the transaction with a visible reader using this object will be aborted at this time by performing a CAS on their status object.
  - If any one of the CAS fails, then P1 is aborted

Committing transaction

- Process must first have been successful in acquiring all its objects
- Perform a CAS on the status of the transaction
- If it has lost any of its acquired objects, status will have been switched to aborted and CAS will fail.

STM issues: optimizing validation and conflict detection.

- Because there can be any transaction performed on inconsistent data, each transaction must be sure that all its data is valid at all time
- Not an issue with visible readers since a transaction is aborted as soon as one of its objects is committed by another transaction.
- Invisible reader must validate all the previously read object before reading any new object.
  - Expensive (?) as number of objects grows.

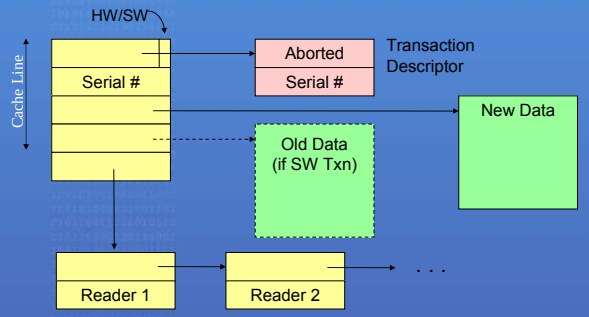
STM issues: optimizing validation and conflict detection (contd).

- Many heuristics and strategies have been proposed, usually involving a trade-off between detecting conflicts and aborting transactions early, and spending time validating just prior to commit.
- Some heuristics involve global counter that can determine the validity of an object without actually having to test.

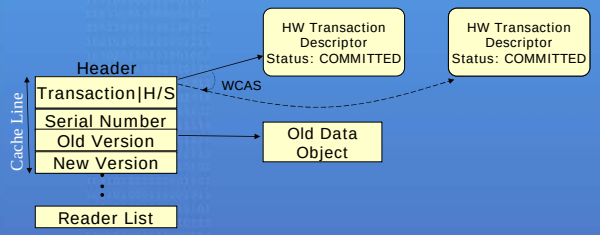
The end

RTM API and Object Metadata

Threads define a set of objects as shared with associated metadata headers
Transactions involve
(1) Indicating start of transaction and registering abort-handler PC
(2) Opening object metadata before reading/writing object data
(3) Acquiring ownership of all objects that are written
(4) Switching status atomically to committed, if still active.



HW Ownership Acquisition



SW Ownership Acquisition