Implementing Atomic Operations in Hardware

• Intel’s `xchg` instruction (review):

```c
int32 xchg(int32 *lock, int32 val) {
    register int old;
    old = *lock;
    *lock = val;
    return (old);
}
```

• At the \texttt{\muop} level, “`xchg (\%esi), \%edi`” becomes 2 memory operations:
  1. load (\%esi) into a hardware register
  2. store \%edi into (\%esi)

• Challenges:
  1. Modern pipelines: only perform 1 memory operation per instruction
  2. What if we want slightly fancier functionality?
     • e.g., atomic increment/decrement, compare-and-swap, etc.
Implementing Atomic Operations in Hardware

• **Atomic increment:**

```c
int32 lock add(int32 *lock, int32 val) {
    register int old;
    old = *lock;
    val += old;
    *lock = val;
    return (old);
}
```

• **Compare-and-exchange (cmpxchg):**

```c
int32 lock add(int32 *lock, int32 old, int32 val) {
    register int t;
    t = *lock;
    if (t == old)  *lock = val;
    return (t);
}
```
Load-Linked / Store Conditional (LL/SC)

- **Key Idea:**
  - *speculate* that the read-modify-write can occur **without getting caught**
    - i.e. no other processor could have read/written the block during R-M-W sequence
      - e.g., because the cache block was held in an Exclusive/Dirty state throughout
  - **check** whether speculation succeeded by monitoring coherence traffic
    - also fails upon context switch, cache line eviction, etc.
  - if **speculation fails**, then retry
    - Store Conditional (**SC**) returns zero (in source register) if it fails

```c
void atomic_add(int *ctr, int delta) {
    do {
        old = LL(ctr);
        new = old + delta;
    } while (!SC(ctr,new));
}
```

- Start tracking `ctr` address
- Speculation failed if SC returns zero
Basic Hardware Trick for Implementing Atomicity

1. Bring some data into the cache
2. Perform calculations using that data
3. Store new result to memory
4. Did we get through Steps 1-3 without conflicting remote accesses to the data?
   - If so, then success!
   - If not, then try again.
     - (to avoid livelock, we may eventually retry non-speculatively)

• Observations:
  - Intel’s \texttt{xchg} does this \texttt{non-speculatively} (for a single memory address)
    - by refusing to give up access to the cache block until it is finished
  - \texttt{LL/SC} does this \texttt{speculatively}, for a \texttt{single memory address}
  - What if we did this \texttt{speculatively}, for \texttt{multiple memory addresses}?

→ \texttt{Transactional Memory}

\textit{Monitor coherence traffic!}
Wouldn’t it be nice if...

- Programmer simply specifies desired outcome:
  - “This code sequence should appear to execute *atomically.*”

```c
void remove_node(Node_type *node) {
    atomic {
        if (node->prev != NULL)
            node->prev->next = node->next;
        if (node->next != NULL)
            node->next->prev = node->prev;
    }
}
```

- The system (e.g., language, run-time software, OS, hardware) makes this happen
  - hopefully *optimistic* (rather than pessimistic) to achieve high performance
  - while enabling *composability* of implementations within abstract objects, etc.
Pessimistic vs. Optimistic Approaches to Atomic Sequences

- **Pessimistic** approach (e.g., locks, monitors):
  - allow **only one thread at a time** to execute a potentially-conflicting atomic sequence

- **Optimistic** approach (e.g., lock-free programming, transactional memory):
  - allow **multiple threads to speculatively execute** potentially-conflicting atomic sequences; roll-back and retry if speculation fails
Hardware transactional memory (HTM)

- Data versioning is implemented in caches
  - Cache the write buffer or the undo log
  - Add new cache line metadata to track transaction read set and write set

- Conflict detection through cache coherence protocol
  - Coherence lookups detect conflicts between transactions
  - Works with snooping and directory coherence

- Note:
  - Register checkpoint must also be taken at transaction begin (to restore execution context state on abort)
HTM design

- Cache lines annotated to track read set and write set
  - R bit: indicates data read by transaction (set on loads)
  - W bit: indicates data written by transaction (set on stores)
    - R/W bits can be at word or cache-line granularity
  - R/W bits gang-cleared on transaction commit or abort
  - For eager versioning, need a 2nd cache write for undo log

- Coherence requests check R/W bits to detect conflicts
  - Observing shared request to W-word is a read-write conflict
  - Observing exclusive (intent to write) request to R-word is a write-read conflict
  - Observing exclusive (intent to write) request to W-word is a write-write conflict
Example HTM implementation: eager-pessimistic

CPU changes
- Ability to checkpoint register state (available in many CPUs)
- TM state registers (status, pointers to abort handlers, …)
Example HTM implementation: eager-pessimistic

- **Cache changes**
  - R bit indicates membership to read set
  - W bit indicates membership to write set
HTM transaction execution

- Transaction begin
  - Initialize CPU and cache state
  - Take register checkpoint

Xbegin
  Load A
  Load B
  Store C ← 5
Xcommit
HTM transaction execution

- Load operation
  - Serve cache miss if needed
  - Mark data as part of read set

Xbegin

Load A

Load B

Store C ← 5

Xcommit
HTM transaction execution

- Load operation
  - Serve cache miss if needed
  - Mark data as part of read set

Xbegin

Load A

Load B

Store C ← 5

Xcommit
HTM transaction execution

- **Store operation**
  - Service cache miss if needed
  - Mark data as part of write set and request write permissions
HTM transaction execution: commit

- Fast two-phase commit
  - Validate: none as all conflicts already detected
  - Commit: Clear read- / write- set flags

Xbegin
  Load A
  Load B
  Store C ← 5

Xcommit ←
HTM transaction execution: detect/abort

Assume remote processor writes to A or D

- Fast conflict detection and abort
  - Check: lookup exclusive requests in the read set and write set
  - Abort: invalidate write set, gang-reset R and W bits, restore to register checkpoint

Coherence requests from another core’s actions
(remote core’s write of A conflicts with local read of A: triggers abort of pending local transaction)
Safely Undoing Writes

- If the cache has to abort writes, where is the old copy of data?
  - Memory was “stale”, but is now current

- If the cache will transactionally access a line that is already dirty, how do we setup the undo?
  - Writeback the data
Intel’s Transactional Synchronization Extensions (TSX)

Restricted Transactional Memory (RTM):
• **XBEGIN / XEND:** specify beginning and end of transaction

```c
void remove_node(Node_type *node) {
    atomic { /* XBEGIN */
        if (node->prev != NULL)
            node->prev->next = node->next;
        if (node->next != NULL)
            node->next->prev = node->prev;
    } /* XEND */
}
```

• Transactions may abort due to *conflict* or *explicit abort* instruction (**XABORT**)

• **If transaction does abort:**
  • jump to target specified by the XBEGIN operand
  • abort information is returned in %eax

Simple RTM Example: Implementing Locks

\texttt{acquire\_lock(mutex)}:

\begin{verbatim}
Retry: \texttt{xbegin Ab}ort \hspace{1em} // Enter RTM execution, Abort is fallback path
    \texttt{cmp mutex, 0} \hspace{1em} // Check to see if mutex is free
    \texttt{jz Success}
\texttt{xabort $0xff} \hspace{1em} // Abort transactional memory if mutex busy
\texttt{Abort:}
    // check EAX and do retry policy
    // (actually acquire lock or wait to retry)
\texttt{Success: ...}
\end{verbatim}

\texttt{release\_lock(mutex)}:

\begin{verbatim}
\texttt{cmp mutex, 0} \hspace{1em} // If mutex not free, then was not RTM execution
    \texttt{jz Commit}
    \texttt{mov mutex, 0} \hspace{1em} // non-RTM unlock (for compatibility)
\texttt{Commit: xend} \hspace{1em} // commit RTM execution
\end{verbatim}

• Can be used for other transactional operations, of course (beyond locks)
  Other uses may still need locks as a fallback mechanism

\textit{Source: Ravi Rajwar, Martin Dixon, “Intel Transactional Synchronization Extensions”, IDF 2012.}
TSX Performance

• TSX can only track a limited number of locations
  • Minimize memory touched

• For example, treap better than AVL tree
  • Self-balancing increases tracked set

• Transactions have a cost
  • Approximately equal to the cost of six atomic primitives to the same cache line

Results collected by Mario Dehesa-Azuara and Nick Stanley as 15-618 Spring 2016 project
Major Roles of the Hardware in Transactional Memory

1. Detects Conflicts between Transactions
   - typically done at a cache line granularity within L1 caches
     - leveraging cache coherence messages (in a MESI-like scheme)
   - conflict if at least one transaction writes to a location accessed by another
   - if a conflict is detected, then abort transaction
   - what if an accessed cache block is evicted?
     - in many TM designs: transaction aborts (can no longer track conflicts)

2. Buffers Side-Effects until Transaction either Commits or Aborts
   - held within cache in a special state (not visible to other processors)
   - if transaction commits: these blocks all become visible
   - if transaction aborts: these blocks are all invalidated

The size of a transaction is usually limited by cache capacity and associativity!
Summary

- Implementing locks on parallel machines
  - parallel applications often prefer spin-waiting (carefully!) to yielding
  - BUT naïve spin-waiting can result in devastating coherence traffic

- Improvements over “test-and-set” locks:
  - “test and test-and-set”: spin in caches with read hits
    - but still a burst of traffic when lock is released
    - backoff: may avoid burst, but what about starvation?
  - queuing locks: $O(1)$ traffic (array or list based)

- Transactional Memory:
  - e.g., Intel’s TSX instructions
  - enables atomic sequences involving multiple memory locations
    - (think “handful” of locations, not a huge number)