### Implementing RTM using the Cache

**Transactional operation:**
- **augment each cache line with an extra bit $T$**
- use a nested counter $C$ and a backup register set
- additional transaction logic:
  - **XBEGIN** increment $C$ and, if $C = 0$, backup registers
  - read or write access to a cache line with $T$ if $C > 0$
  - applying an *invalidate* message from an *invalidate queue* to a cache line with $T = 1$ issues *XABORT*
  - observing a *read* message for a *modified* cache line with $T = 1$ issues *XABORT*
  - *XABORT* clears all $T$ flags, sets $C = 0$ and restores CPU registers
  - *XCOMMIT* decrement $C$ and, if $C = 0$, clear all $T$ flags

---

### Protecting the Fall-Back Path

Use a lock to prevent the transaction from interrupting the fall-back path:

```c
int data[100]; // shared
int mutex;
void update(int idx, int value) {
    if (__begin__(==1)) {
        if (__mutex>() __xabort();
        data[idx] ++ value;
        __xend();
    } else {
        wait(mutex);
        data[idx] ++ value;
        signal(mutex);
    }
}
```

- fall-back path may not run in parallel with others
- transactional region may not run in parallel with fall-back path
Illustrating Transactions
Augment MESI state with extra bit $T$ per cache line. CPU A: E5, CPU B: I

Thread A
```c
int tmp = data[idx];
data[idx] = tmp+value;
_xend();
```

Thread B
```c
int tmp = data[idx];
data[idx] = tmp+value;
_xend();
```

Protecting the Fall-Back Path
Use a lock to prevent the transaction from interrupting the fall-back path:
```c
int data[100]; // shared
int mutex;
void update(int idx, int value) {
  if (xbegin()==-1) {
    if (mutex>0) _xabort();
data[idx] += value;
    _xend();
  } else {
    wait(mutex);
data[idx] += value;
    signal(mutex);
  }
}
```

- fall-back path may not run in parallel with others ✓
- transactional region may not run in parallel with fall-back path

Implementing RTM using the Cache
Transaction operation:
- augment each cache line with an extra bit $T$
- use a nesting counter $C$ and a backup register set
- additional transaction logic:
  - _XBEG_ increment $C$ and, if $C = 0$, back up registers
  - read or write access to a cache line sets $T$ if $C > 0$
  - applying an _invalidate_ message from _invalidate queue_ to a cache line with $T = 1$ issues _XABORT_
  - observing a _read_ message for a _modified_ cache line with $T = 1$ issues _XABORT_
  - _XABORT_ clears all $T$ flags, sets $C = 0$ and restores CPU registers
  - _XCOMMIT_ decrement $C$ and, if $C = 0$, clear all $T$ flags

Illustrating Transactions
Augment MESI state with extra bit $T$ per cache line. CPU A: E5, CPU B: I

Thread A
```c
int tmp = data[idx];
data[idx] = tmp+value;
_xend();
```

Thread B
```c
int tmp = data[idx];
data[idx] = tmp+value;
_xend();
```

```c
A
 tmp=data[idx]  d   invalidate
 store          cache
 invalidate queue
 store B
 tmp=data[idx]  d   invalidate
```
Common Code Pattern for Mutexes

Using HTM in order to implement mutex:

```c
int data[100]; // shared
int mutex;
void update(int idx, int val) {
    lock(mutex);
    data[idx] += val;
    unlock(mutex);
}
```

- the critical section may be executed without taking the lock (the lock is elided)
- as soon as one thread conflicts, it aborts, takes the lock in the fallback path and thereby aborts all other transactions that have read mutex

---

Hardware Lock Elision

Observation: Using HTM to implement lock elision is a common pattern to provide special handling in hardware: HLE

- provides a way to execute a critical section without the need to immediately modify the cacheline in order to acquire and release the lock
- requires annotations:
  - instruction that increments the semaphore must be prefixed with XACQUIRE
  - instruction setting the semaphore to 0 must be prefixed with XRELEASE
  - these prefixes are ignored on older platforms
- for a successful elision, all signal/wait operations of a lock must be annotated
Hardware Lock Elision

*Observation:* Using HTM to implement lock elision is a common pattern
- provides special handling in hardware: HLE
- provides a way to execute a critical section without the need to immediately modify the cacheline in order to acquire and release the lock
- requires annotations:
  - instruction that increments the semaphore must be prefixed with **XACQUIRE**
  - instruction setting the semaphore to 0 must be prefixed with **XRELEASE**
  - these prefixes are ignored on older platforms
- for a successful elision, all signal/wait operations of a lock must be annotated

Implementing Lock Elision

Transactional operation:
- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer
- **XACQUIRE** of lock ensures shared/exclusive cache line state with $T = 1$, issues **XBEGIN** and stores written value in **elided lock buffer**
- r/w access to a cache line sets $T$
- like RTM, applying an **invalidate** message to a cache line with $T = 1$ issues **XABORT**, analogous for **read** message to a modified cache line
- a local CPU **read** from the address of the elided lock accesses the buffer
- on **XRELEASE** on the same lock, decrement $C$ and, if $C = 0$, clear $T$ flags and elided locks buffer and commit to memory

Transactional Memory: Summary

Transactional memory aims to provide **atomic** blocks for general code:
- frees the user from deciding how to lock data structures
- compositional way of communicating concurrently
- can be implemented using software (locks, atomic updates) or hardware

The devil lies in the details:
- semantics of explicit HTM and STM transactions quite subtle when mixing with non-TM (weak vs. strong isolation)
- single-lock atomicity and transactional sequential consistency semantics
- STM not the right tool to synchronize threads without shared variables
- TM providing opacity (serializability) requires eager conflict detection or lazy version management

Devils in implicit HTM:
- RTM requires a fall-back path
- no progress guarantee
- HLE can be implemented in software using RTM
TM in Practice

Availability of TM Implementations:

- GCC can translate accesses in `.transaction.atomic` regions into `libitm` library calls
- The library `libitm` provides different TM implementations:
  - On systems with TSX, it maps atomic blocks to HTM instructions
  - On systems without TSX and for the fallback path, it resorts to STM
- RTM support slowly introduced to OpenJDK Hotspot monitors

Outlook

Several other principles exist for concurrent programming:

1. **non-blocking message passing (the actor model)**
   - A program consists of actors that send messages
   - Each actor has a queue of incoming messages
   - Messages can be processed and new messages can be sent
   - Special filtering of incoming messages
   - *example*: Erlang, many add-ons to existing languages

2. **blocking message passing (CSP, π-calculus, join-calculus)**
   - A process sends a message over a channel and blocks until the recipient accepts it
   - Channels can be send over channels (π-calculus)
   - *examples*: Occam, Occam-π, Go

3. **(immediate) priority ceiling**
   - Declare *processes* with priority and *resources* that each process may acquire
   - Each resource has the maximum (ceiling) priority of all processes that may acquire it
   - A process' priority at run-time increases to the maximum of the priorities of held resources
   - The process with the maximum (run-time) priority executes

References


Online resources on Intel HTM and GCC's STM: