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) {
    if (_xbegin() == -1) {
        if (!mutex) _xabort();
        data[idx] += val;
        _xend();
    } else {
        wait(mutex);
        data[idx] += val;
        signal(mutex);
    }
}

void lock(int mutex) {
    if (_xbegin() == -1) {
        if (!mutex) _xabort();
        else return;
    } wait(mutex);
}

void unlock(int mutex) {
    if (!mutex) signal(mutex);
    else _xend();
}
```

- 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
- provides special handling in hardware
- 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

Transactionally 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
    - message to a modified cache line
  - `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

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.
- ISO Standard to come: `C++ Extensions for Transactional Memory`, introducing `synchronized {}` (preview in GCC 6.1).
- 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:
- **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.
- **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.
- **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**

D. Dice, O. Shalev, and N. Shavit.
Transaction Locking II.

T. Harris, J. Larus, and R. Rajwar.

Online resources on Intel HTM and GCC’s STM: