Extending Hardware Transactional Memory to Support Non-busy Waiting and Non-transactional Actions

Craig Zilles and Lee Baugh
University of Illinois at Urbana-Champaign

paper available at:
http://www-faculty.cs.uiuc.edu/~zilles/papers/non_transact.transact2006.pdf

Two main TM thrusts

- HW-centric
  - ☑ common-case performance, strong atomicity
  - ☑ implicit (avoid re-compile of libraries)
  - ✱ simple semantics
  - ✱ handling overflow

Two main TM thrusts

- HW-centric
  - ☑ common-case performance, strong atomicity
  - ☑ implicit (avoid re-compile of libraries)
  - ✱ simple semantics
  - ✱ handling overflow
- SW-centric
  - ☑ flexibility/extensibility, richer semantics
  - ☑ tighter integration with language/run-time
  - ✱ lower performance, weak atomicity
  - ✱ explicit (code includes transaction info)

This Paper

- HW-centric
  - ☑ common-case performance, strong atomicity
  - ☑ implicit (avoid re-compile of libraries)
  - ✱ simple semantics
  - ✱ handling overflow
- SW-centric
  - ☑ flexibility/extensibility, richer semantics
  - ☑ tighter integration with language/run-time
  - ✱ lower performance, weak atomicity
  - ✱ explicit (code includes transaction info)

Outline

- Background: Virtual Transactional Memory (VTM)
- Waiting w/o spinning:
  - "retry"
  - due to conflict (much like semaphores)
- Pausing as a transactional loop-hole
  - accesses to contended data
  - performing non-transactional actions
  - retaining state across an abort

Virtual Transactional Memory

- Goals:
  - Small XACT: entirely in cache, no overhead
  - Large XACT: ownership/undo state stored in-memory, can persist across time-slice
  - Allow both kinds to co-exist
- Eager conflict detection, versioning
- Transactional status word (XSW)
  - Holds transaction state (active, commit, abort)
  - Pointed to by ownership records
  - Monitored by running transaction

Retry

- Avoid "lost wake-up" bugs
- Composable means of "wait for multiple objects"

```
element *get_element_to_process() {
   TRANSACTION_BEGIN;
   for (int i = 0 ; i < NUM_LISTS ; ++ i) {
      if (list[i].has_element()) {
         element *e = list[i].get_element();
         TRANSACTION_END;
         return e;
      }
   }
   retry;
}
```

Implementation

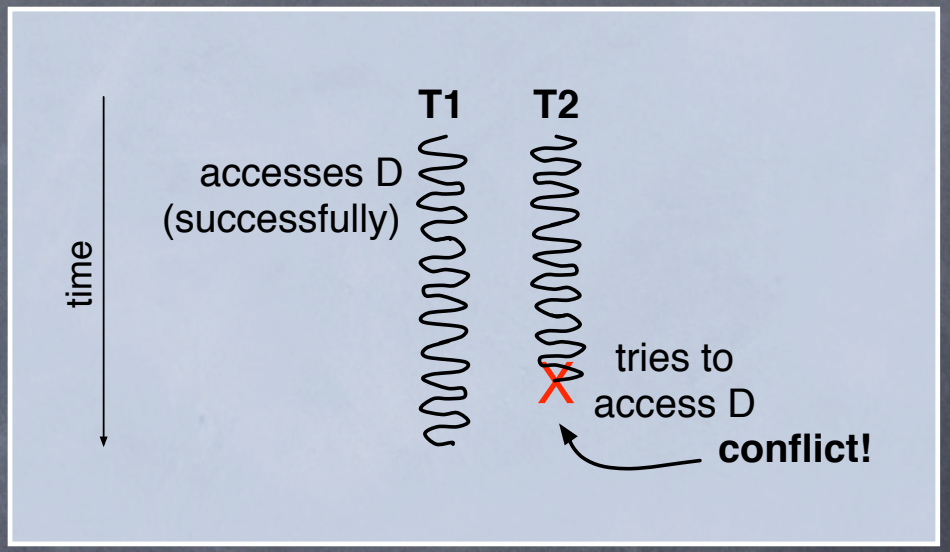
1. Ensure retry'ed transaction loses conflicts
2. Want to de-schedule thread until conflict
   - VTM already supports persistent transactions
   - Main challenge is making sure wake-up occurs

Ensuring Wake-up

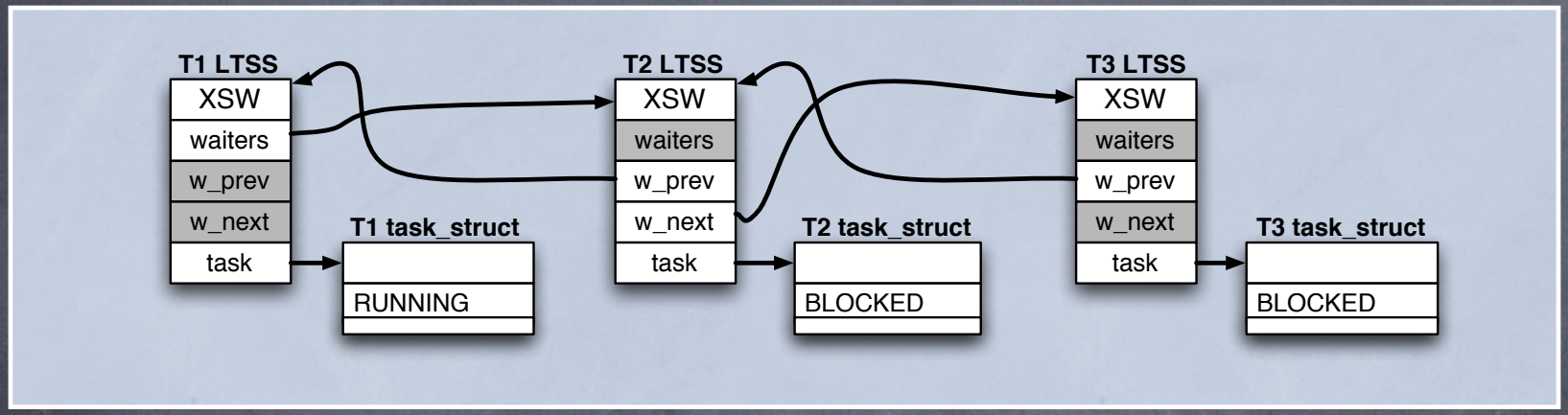
- Race condition between de-scheduling and being aborted
- Atomically transfer responsibility of waking thread
  - After marking thread as blocked,
  - Add marker to XSW with compare-and-swap
  - If fails, re-schedule thread (already aborted)

Wait on contention

- Three outcomes:
  - Abort
  - Spin
  - De-schedule
    - For long transactions with low contention
    - Mitigates worst case behavior
    - Corresponds to O/S semaphores



Implementation



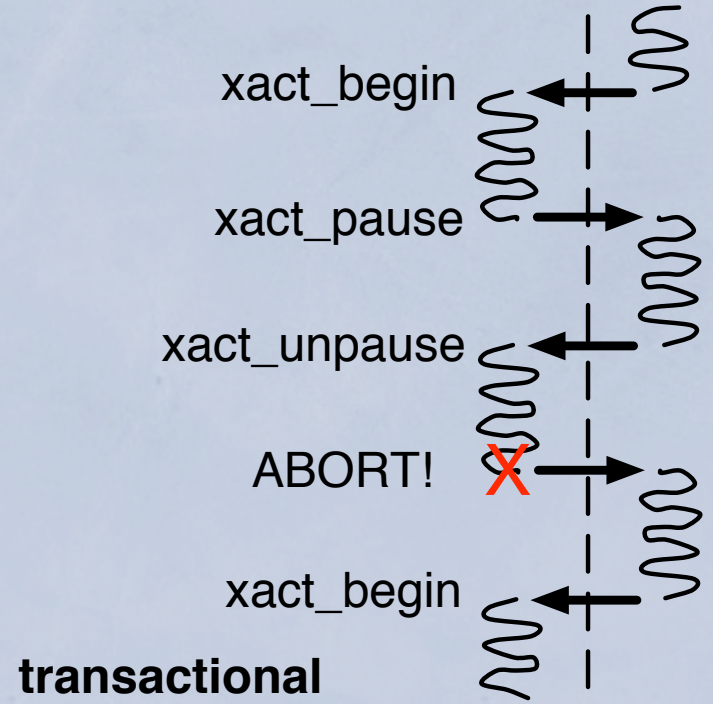
- Build a list of who waits on who
- Deterministic contention manager -> no cycles
- Annotated XSW indicates there are waiters
- Same trick to transfer wake-up responsibility

Pausing Transactions

- Providing a transactional loop-hole
  - HTM default is that everything is transactional
- Enable violating transaction's isolation
  - To avoid conflicts on highly-contended data
  - For performing non-transactional actions
  - Logging abort conditions, exceptions, tools

Simple Example:

```
...
transaction {
  ...
  ...
  ++ statistic;
  ...
}
...
```



(try transaction)

increment statistic atomically (using CAS)
register compensation action

(perform compensation)
decrement statistic atomically (using CAS)
deallocate compensation data
(retry transaction)

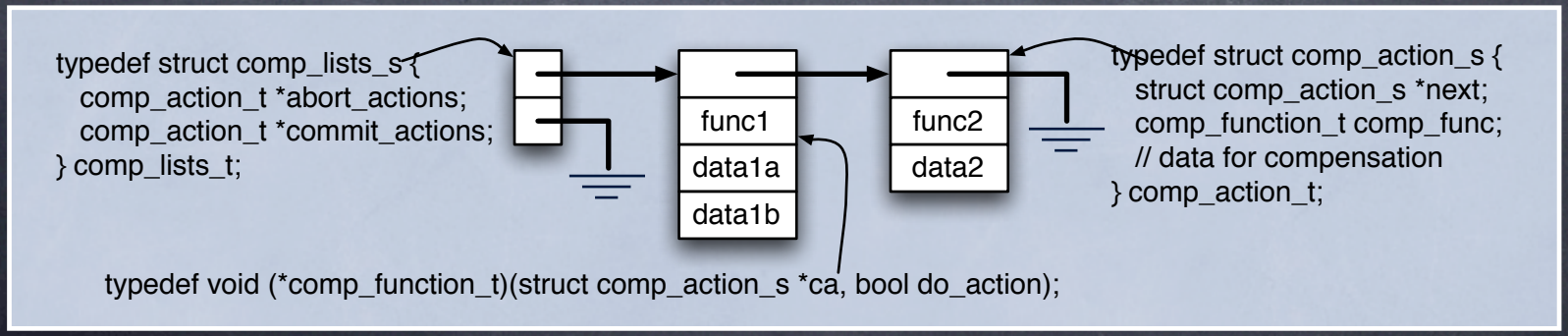
non-transactional

Implementation

- Paused modifier to transaction state
  - Distinct from "swapped"
- Load/stores not added to read/write set
  - Strong atomicity, but...
  - Allow reads to footprint (passing arguments)
- Handling writes to footprint?
  - Clean semantics demand write through
  - Common occurrences (e.g., stack) don't

Implementation, cont.

- No atomicity/isolation guarantees
  - Must conventionally synchronize
- Support registering compensation in software
  - Register function and arguments
  - Performed after commit/abort (+/- atomically)



Implementation, cont.

- Non-isomorphic to "non-xact load/store"
  - No (asynchronous) aborts in paused region
    - Must release locks, insert compensation

Support Malloc/Free

- dlmalloc uses mmap/munmap for large allocations
  - even HTM shouldn't absorb kernel activity
- aborted mmap leaks virtual address space
- munmap shouldn't be performed until commit
- free implementation: pause, query xact state
  - if no-xact: do operation
  - if xact: register commit action, unpause

Pause vs. Open Nesting

- Can be used for some of the same tasks
- Open Nesting
  - More overhead (nesting in hardware?)
  - Stronger guarantees (transaction)
    - Not always necessary
      - Isolated data items (use CAS)
      - Thread-local data

Conclusion

- Shown two extensions to HTM system
  - Support non-busy waiting by transactions
  - Support non-transactional work in transaction
- Minimal impact on hardware
  - extension of existing XSW
  - calling of software handlers through exceptions