ORACLE®

Specifying and Verifying Transactional Memory

Victor Luchangco
Oracle Labs

Transactional Memory

- Designate sections of code to be executed as **transactions**
  - committed transactions appear to take effect atomically
  - aborted transactions are not observed by other transactions
- Very active area of research
  - TM implementations: hardware, software, hybrid
  - specification and verification
  - applications and user studies

ORACLE

Why Specify and Verify?

- Show that a TM implementation is correct.
- Show that an application using TM is correct.

Transactional Memory Specifications

- Necessary for reasoning rigorously about TM
  - especially important as TM is a foundation for concurrent programming
- Variety of specifications
  - different contexts: hardware/software, managed/unmanaged, etc.
  - different uses: define allowed behavior, exposition, formal verification
- Interaction with other features
  - nontransactional operations, exceptions
  - condition synchronization

Desiderata for Specifications

- Precise, unambiguous
- Complete
- Easy to understand
- Flexible for implementors
- Composable
- Theory for reasoning about systems and their behavior
- Tools for formal verification

ORACLE

TM Specification: A First Attempt

- Committed transactions appear to execute atomically
- Aborted transactions not observed by other transactions

ORACLE

TM Specification: A First Attempt

- Committed transactions appear to execute atomically.
- Aborted transactions not observed by other transactions.

- Guarantees for active and aborted transactions?
- When do transactions commit or abort?
- “Execute atomically”? Ordering and consistency guarantees?
- TM interface and well-formedness?
- Nontransactional operations?

 |

Outline

- Formal model for concurrent programs
- Basic TM correctness properties (opacity, TMS1)
- Verifying real TM algorithms (TMS2, NOrec)
- Nontransactional operations (NTMS1)
- Adding support for transactions in C++

Formal Model for Concurrent Programs

Modeling Concurrent Programs

- State-transition system
  - label transitions with **actions**
  - actions may be **external** (i.e., observable) or **internal**
  - an **execution** is a sequence of steps/transitions
  - a **trace** (aka **history**) is the sequence of external actions in an execution
  - traces generated by system are observable behavior
- Specification specifies properties that the traces must satisfy.
  - traces that satisfy these properties are called **legal histories**

Background: I/O Automata

- states (including one or more start states)
- actions, either external (input/output) or internal
- transition relation: (state, action, state)
- fairness partition (elided)

- executions: s_0, a_1, s_1, a_2, s_2,… (s_0 is start state)
- traces: projection of executions onto external actions
  – visible behavior of automaton
  – trace inclusion = implementation (not bisimulation)

ORACLE

Background: Invariants

- A state is **reachable** if it is in some execution.
- An **invariant** is a property that is true of all reachable states.
  - the most important tool in reasoning about concurrent programs
  - often proved by induction on the length of executions

Background: Trace Properties

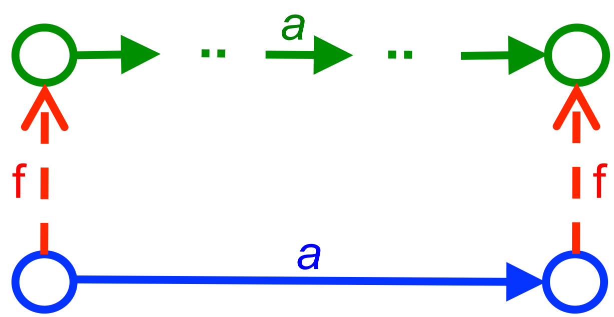
- A **trace property** is a set of sequences of events.
- Automaton **A** **satisfies** trace property **P** if every trace of **A** is in **P**.
  - typically proved by induction on the length of an *execution* (of which the trace is a projection)
  - proofs mostly ad hoc, with theorems specific to certain trace properties
- May include traces that are “infeasible”

Background: Automaton as Specification

- An automaton generates a set of traces.
  – can use this as a specification
  – includes only feasible traces (they are generated by automaton)
  – more detailed, more “boilerplate”
  – intuitive properties may be obscured
- Can embed in IOA: every step must preserve legal-history predicate

ORACLE

Background: Simulation Proofs



- Forward simulation f from **C** to **A**
  - relation on states(C) × states(A)
  - for every start state of C, there is a corresponding start state of A
  - for every step (s,a,s') of C and every state u of A corresponding to s, there is a state u' of A corresponding to s' such that there is a (possibly empty) sequence of steps from u to u' that appears identical to the step of C.

 |

Background: Simulation Proofs

- Many variants
  - forward simulation, backward simulation, refinement, history mapping,…
- Existence of simulation implies trace inclusion
  - forward and backward simulations are complete

Hierarchical, Reusable Proofs

- High-level specification captures abstract requirements
- Intermediate specification for implementation approach
- Model algorithms at multiple levels

- Automata all the way down
  - abstraction all the way up

Basic TM Correctness Properties

TM Interface

- invocations
  - begin_t
  - $\text{inv}_t(op)$
  - commit_t
  - cancel_t
- responses
  - beginOk_t
  - $\text{resp}_t(r)$
  - commitOk_t
  - abort_t

- Assumes sequential specification of “object type”
  - typically read/write memory (i.e., ops are read(x) or write(x,v))
- Only transactional operations

ORACLE

TM Correctness Properties

- Committed transactions appear to execute atomically.
- Aborted transactions not observed by other transactions.
- Traces are *well-formed*.

TM Correctness Properties

- Committed transactions appear to execute atomically
- Aborted transactions not observed by other transactions

- When do transactions commit or abort?
- Guarantees for aborted transactions? active transactions?
- Ordering and consistency guarantees?
- Nontransactional operations?
- …

ORACLE

Well-formedness

- Each transaction starts with begin invocation
- Alternating invocation and matching response
  - abort can match any invocation
- No invocation after commit or abort response

- These restrict clients of TM as well as the TM system.

ORACLE

Serializability

- “Equivalent” to some serial execution of committed transactions
  - ordering and consistency guarantees for committed transactions
- No guarantees for active and aborted transactions
- No nontransactional operations

- Define correct serial execution (only committed transactions)
- Define equivalence

Opacity

- Active/aborted transactions “consistent” with committed transactions
- Appropriate when transactions cannot be sandboxed
  – otherwise transactions may cause unrecoverable run-time errors

Opacity

- Active/aborted transactions “consistent” with committed transactions
- Appropriate when transactions cannot be sandboxed
  - otherwise transactions may cause unrecoverable run-time errors
- Specified as predicate on histories
  - originally not prefix-closed
  - all prefixes must satisfy “final-state opacity”
- Stronger than necessary to avoid run-time errors
  - virtual world consistency (VWC), TMS1

ORACLE

Opacity as an Automaton

- State variables:
  - `extOrder`
  - for each transaction t: $\mathtt{status}_t$, $\mathtt{ops}_t$, $\mathtt{pendingOp}_t$
  - updated in obvious way
- Well-formedness
- Responses have (final-state) opacity as postcondition
- Equivalent version with validation preconditions
  - validCommit, validFail, validResp

TMS1

- Active/aborted transactions only need to be consistent with some possible serial execution of transactions
  - must include all prior committed transactions
  - must not include any prior aborted transactions
- Specified as I/O automaton
  - validation conditions (validCommit, validFail, validResp)
- Proved that opacity automaton implements TMS1
  - verified in formal framework using PVS

Formal Framework for Specifying and Verifying Transactional Memory Algorithms

ORACLE

A framework for verifying TM

- I/O automata and simulation techniques
- PVS verification system
- Framework comprises:
  - formalize automata/simulation theory
  - specifications of TMS1, Opacity, TMS2 (several variants)
  - proof that Opacity implements TMS1
  - proof that TMS2 implements Opacity (for read-write memory)
  - proofs of equivalence of various TMS2 variants
  - formalization of NOrec algorithm
  - proof that NOrec implements TMS2

PVS verification system

- Typed higher-order logic
- Rewriting-based theorem prover
  - proof obligations: lemmas, type-correctness conditions (TCCs)

Automata in PVS

```
Automata[State, Action: TYPE+,
                start: nonempty_pred[State],
                trans: pred[[State,Action,State]]]: THEORY BEGIN

FiniteStepSeq: TYPE =
    [# actions: finseq[Action],
        states: { ss: nonempty_finseq[State] | length(ss) = length(actions) + 1 } #]

s, s0, s1: VAR State
a: VAR Action
stepseq: VAR FiniteStepSeq

length(stepseq): nat = stepseq`actions`length

steps(stepseq): finseq[Step] =
  (# length := length(stepseq`actions),
      seq := LAMBDA (n: below[length(stepseq`actions)]):
                  (stepseq`states(n), stepseq`actions(n), stepseq`states(n+1)) #)
```


Automata in PVS

```
finiteExecFrag(stepseq): bool =
  FORALL (n: below[length(stepseq)]): trans(steps(stepseq)(n))

finiteExecution(stepseq): bool =
  finiteExecFrag(stepseq) AND start(first(stepseq))

reachable(s: State): INDUCTIVE bool =
  start(s) OR (EXISTS s0: State, a:Action): reachable(s0) AND trans(s0,a,s))

invariant(p: pred[State]): bool =
  FORALL (s State): reachable(s) IMPLIES p(s)

invariantInduction: LEMMA
  FORALL (p: pred[State]):
     (FORALL s: start(s) IMPLIES p(s)) AND
     (FORALL s0: State, a: Action, s1: State:
        reachable(s0) AND reachable(s1) AND p(s0) AND trans(s0,a,s1) IMPLIES p(s1))
   IMPLIES invariant(p)

END Automata
```

TMS2: “Write-latest”

- $beginIdx_t$: “timestamp” of state at beginning of txn t
- mem: sequence of memory states
- $wrSet_t$: write set of t
- $rdSet_t$: read set of t
- pc_t: bookkeeping

```
TMS2[Txn, Loc, Val: TYPE+, validInit: nonempty_pred[[Loc -> Val]]]: THEORY BEGIN

ActionType: DATATYPE ...
Action: TYPE+ = [# txn: Txn, acttype: ActionType #]
State: TYPE =
  [# pc: [Txn -> PCValue],
     beginIdx: [Txn -> nat],
     mem: nonempty_finseq[RWState],
     wrSet: [Txn -> PartialFunction[Loc,Val]],
     rdSet: [Txn -> PartialFunction[Loc,Val]] #]

start(s): bool =
     s`mem`length = 1 AND
     validInit(last(s`mem)) AND
     (FORALL t:  s`pc(t) = notStarted AND
                 s`rdSet(t) = emptyMap AND
                 s`wrSet(t) = emptyMap)

precondition(a)(s): bool = …
effect(a,s): State = ...
trans(s0,a,s1): bool = precondition(a)(s0) AND s1 = effect(a,s0)
IMPORTING Automata[State, Action, start, trans]
```

```
ActionType: DATATYPE WITH SUBTYPES external, internal
BEGIN
  beginTxn: beginTxn?                                  : external
  beginOk: beginOk?                                    : external
  inv(i: Invocation): inv?                             : external
  resp(r: Response): resp?                             : external
  commit: commit?                                      : external
  commitOk: commitOk?                                  : external
  cancel: cancel?                                      : external
  abort: abort?                                        : external
  doReadWritten(l: Loc): doReadWritten?                          : internal
  doReadUnwritten(l: Loc, n: nat): doReadUnwritten?  : internal
  doWrite(l:Loc, v: Val): doWrite?                               : internal
  doCommitReadOnly: doCommitReadOnly?                            : internal
  doCommitWriter: doCommitWriter?                                : internal
END ActionType
```

```
precondition(a)(s): bool = LET t = a`txn IN
 CASES a`acttype OF
  beginTxn: s`pc(t) = notStarted,
  beginOk: s`pc(t) = beginPending,
  inv(i): s`pc(t) = ready,
  resp(r):     (readResp?(s`pc(t)) AND r = readOk(v(s`pc(t))))
           OR (writeRespOk?(s`pc(t)) AND r = writeOk),
  commit: s`pc(t) = ready,
  commitOk: s`pc(t) = commitRespOk,
  cancel: s`pc(t) = ready,
  abort: s`pc(t) = beginPending OR
         reading?(s`pc(t)) OR
         writing?(s`pc(t)) OR
         s`pc(t) = doCommit OR
         s`pc(t) = cancelPending,
  doReadWritten(l): s`pc(t) = reading(l) AND dom(s`wrSet(t))(l),
  doReadUnwritten(l,n): s`pc(t) = reading(l) AND
                        NOT dom(s`wrSet(t))(l) AND
                        validIndex(s,t,n),
  doWrite(l,v): s`pc(t) = writing(l,v),
  doCommitReadOnly:   s`pc(t) = doCommit AND dom(s`wrSet(t)) = emptyset,
  doCommitWriter: s`pc(t) = doCommit AND
                  dom(s`wrSet(t)) /= emptyset AND
                  readCons(last(s`mem),s`rdSet(t))
 ENDCASES
```

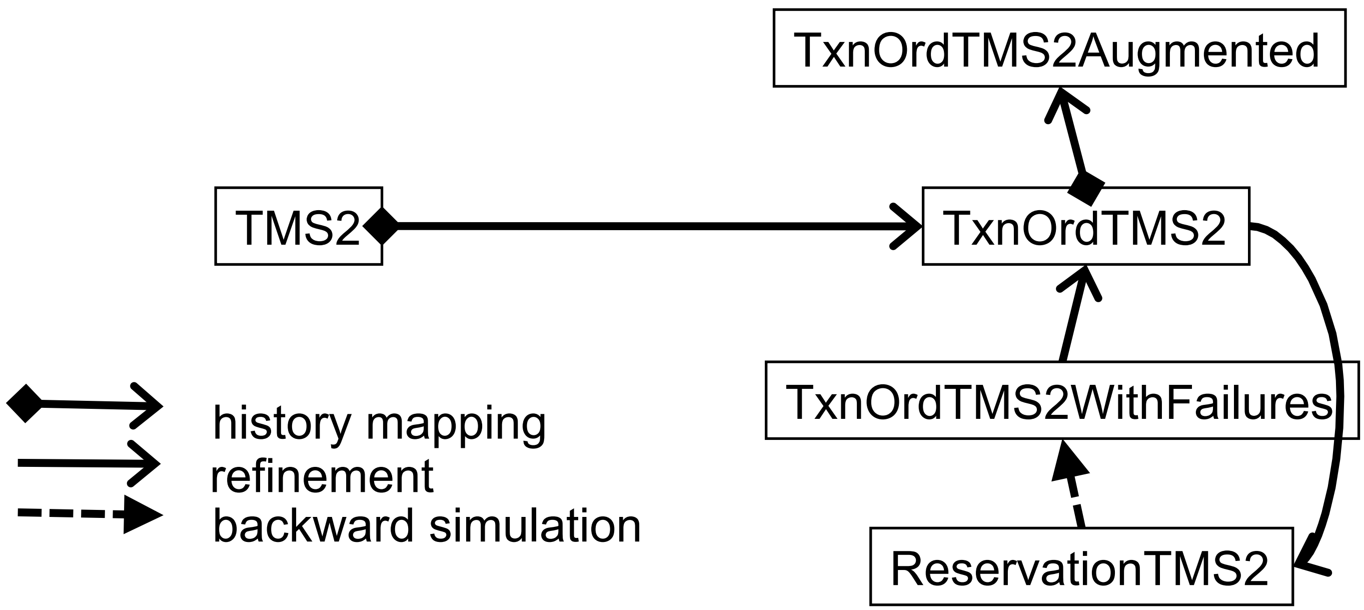


```
effect(a,s): State =
 IF precondition(a)(s) THEN LET t = a`txn IN
  CASES a`acttype OF
   beginTxn: s WITH [`pc(t) := beginPending,`beginIdx(t) := s`mem`length-1],
   beginOk: s WITH [`pc(t) := ready],
   inv(i): s WITH [`pc(t) := IF read?(i) THEN reading(l(i)) ELSE writing(l(i),v(i)) ENDIF],
   resp(r): s WITH [`pc(t) := ready],
   commit: s WITH [`pc(t) := doCommit],
   commitOk: s WITH [`pc(t) := committed],
   cancel: s WITH [`pc(t) := cancelPending],
   abort: s WITH [`pc(t) := aborted],
   doReadWritten(l): s WITH [`pc(t) := readResp(down(s`wrSet(t)(l)))],
   doReadUnwritten(l,n): (s WITH [`pc(t) := readResp(v), `rdSet(t)(l) := up(v)]
                                WHERE v = s`mem(n)(l)),
   doWrite(l,v): s WITH [`pc(t) := writeRespOk, `wrSet(t)(l) := up(v)],
   doCommitReadOnly: s WITH [`pc(t) := commitRespOk],
   doCommitWriter: s WITH [`pc(t) := commitRespOk,
                                 `mem := s`mem o oride(last(s`mem), s`wrSet(t))]
  ENDCASES
 ELSE
  arbitraryState
 ENDIF
```

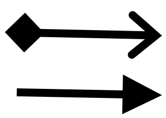

TMS2 variants

- TxnOrdTMS2
  - keeps track of order of committing writing transactions
  - history mapping from TMS2
- TxnOrdTMS2WithFailures
  - allows aborted transactions in order above
- ReservationTMS2
  - writers “reserve place” in order, but they may abort
  - requires backward simulation to TxnOrdTMS2WithFailures
- TxnOrdTMS2Augmented
  - maintains history variables useful to prove opacity

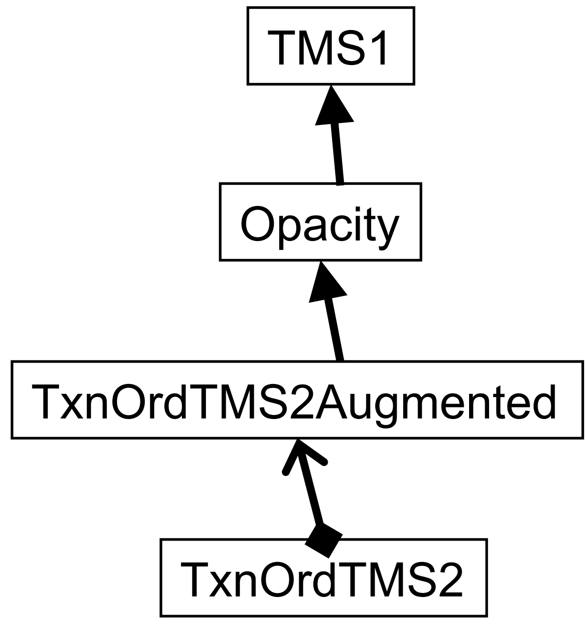
Proofs in framework



Proofs in framework



history mapping
forward simulation



NOrec algorithm [Dalessandro et al.]

- Simple deferred-update alg: “no ownership records”
  – write shared memory on commit
  – maintain private read and write sets
  – reads are invisible
- Sequence lock to protect writeback
  – serializes commit of writing transactions
  – readers check that lock is not held
- Value (re)validation when sequence lock changes
- Low overhead
  – good when conflicts are rare

NOrec automata

Automaton	Action types	Possible pc values
NOrecAtomicCommitValidate	15	13
NOrecDerived	19	13
NOrec	21	15
NOrecPaperPseudocode	45	35

Verifying Transactional Memory

- Formal framework in PVS
  - typed higher order logic
  - rewriting-based theorem prover
- Includes libraries for I/O automata, TM specs, etc.
  - also library for sequences
- Formally verified proofs of TM algorithm and specifications

NOrec

- Simple deferred-update algorithm: “no ownership records”
  – reads are invisible
- Sequence lock to protect writeback
  – serializes commit of writing transactions
  – readers check that lock is not held
- Value (re)validation when sequence lock changes
- Low overhead
  – good when conflicts are rare

Hierarchy of NOrec Automata

Automaton	Action types	Possible pc values
NOrecAtomicCommitValidate	15	13
NOrecDerived	19	13
NOrec	21	15
NOrecPaperPseudocode	45	35

- Verify that each implements the preceding one

TMS2: A Common Implementation Approach

- State variables
  - `mem`: sequence of memory states
  - $\texttt{beginIdx}_t$: “timestamp” of state at beginning of transaction t
  - $\texttt{wrSet}_t$: write set of t
  - $\texttt{rdSet}_t$: read set of t
  - $\texttt{pc}_t$: bookkeeping

- Implements TMS1

TMS2: A Common Implementation Approach

- Assumes read/write memory
- Deferred update
  - write shared memory on commit
  - maintain private read and write sets
- Can read in the past, but always write the current value
  - new reads extend and validate read set
  - writing transactions validate read set during commit
  - no validation needed to commit read-only transactions

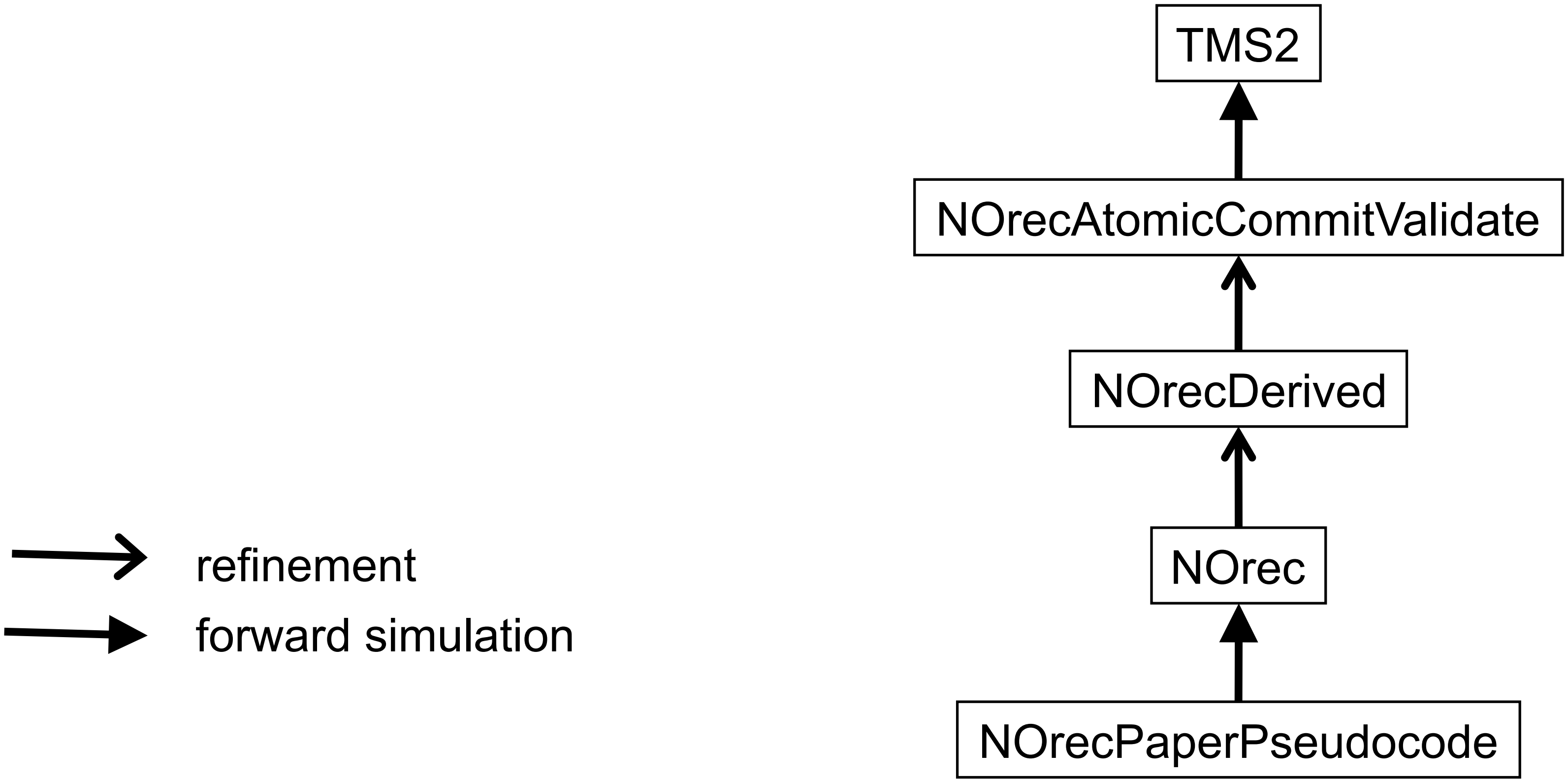
TMS2: A Common Implementation Approach

- State variables
  - `mem`: sequence of memory states
  - $\texttt{beginIdx}_t$: "timestamp" of state at beginning of transaction t
  - $\texttt{wrSet}_t$: write set of t
  - $\texttt{rdSet}_t$: read set of t
  - $\texttt{pc}_t$: bookkeeping

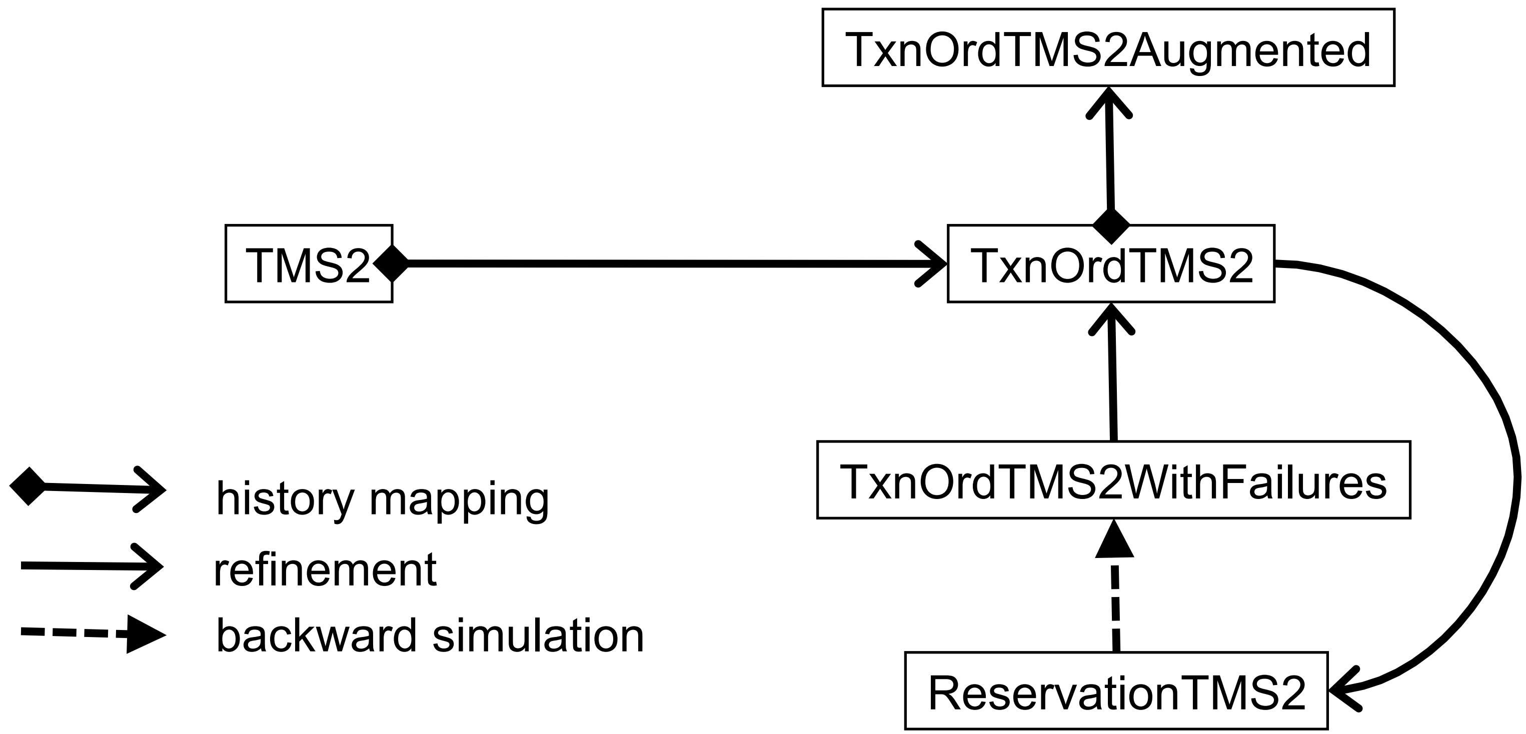
- Implements TMS1 (for read/write memory)
- Several variants: ReservationTMS2, …

ORACLE

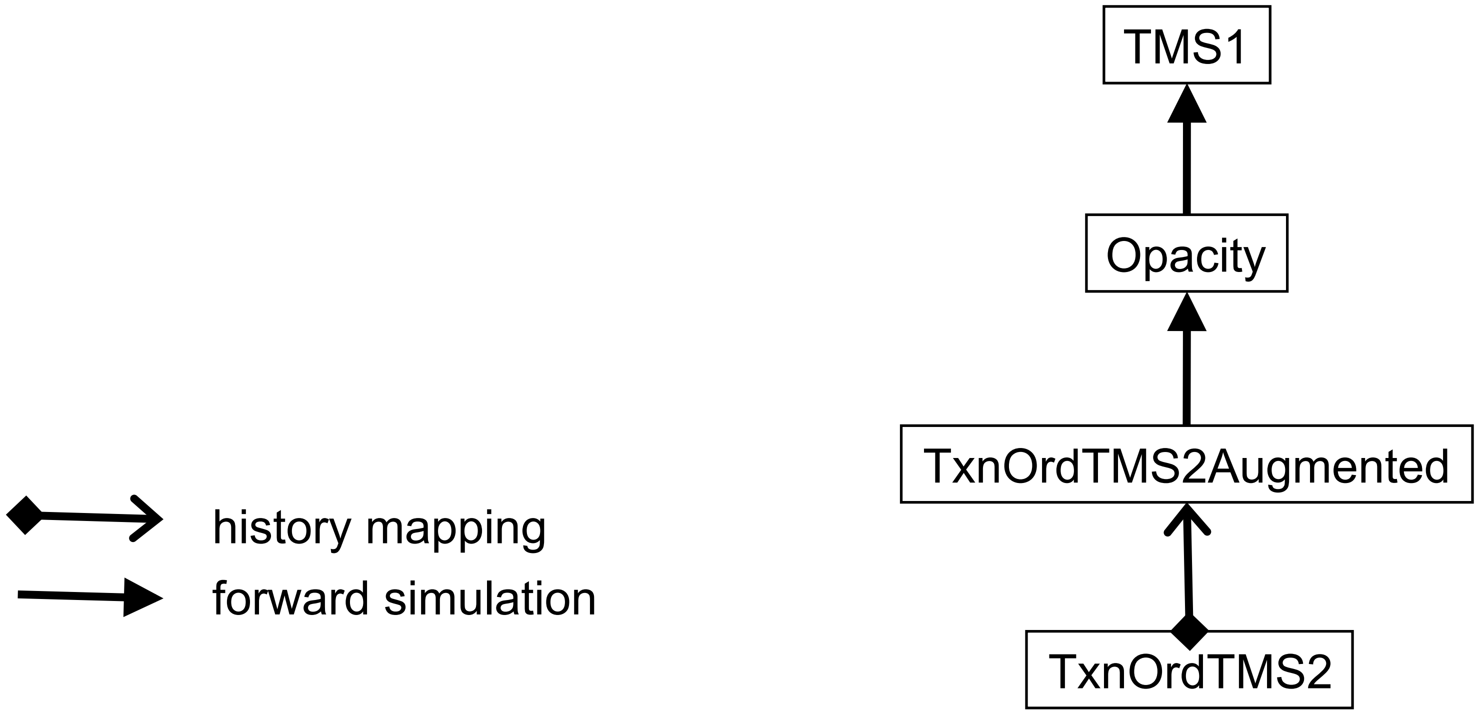
Proofs Verified in Framework



Proofs Verified in Framework



Proofs Verified in Framework



Nontransactional Operations

Why Nontransactional Operations?

- Real systems provide a variety of synchronization mechanisms.
- Different mechanisms are better for some tasks.
- Transactional access must be mediated, incurring overhead.
- Programs that use TM may need legacy libraries.
  - technical, legal, business issues

ORACLE

TM Interface with Nontransactional Operations

- input (invocations)
  - $begin_t$
  - $tInv_t(op)$
  - $commit_t$
  - $cancel_t$
  - $nInv_n(op)$
- output (responses)
  - $beginOk_t$
  - $tResp_t(r)$
  - $commitOk_t$
  - $abort_t$
  - $nResp_n(r)$
- well-formedness for data-race free programs and correct TMs

Extending TMS1 with Nontransactional Operations

- Validity conditions
  - adjust transaction validity conditions to handle nontransactional operations
  - new validity condition for nontransactional operations
- Handle data races
  - correct TM may exhibit arbitrary behavior if program is racy
  - non-racy programs may cause races if TM gives incorrect results

Defining data races

- Conflict relation: symmetric binary relation specified by object type
- Transactions never race with each other.
- Nontransactional operations race iff they conflict and overlap.
- Nontransactional operation races with a transaction iff
  – they overlap and
  – any operation invoked by the transaction conflicts with the nontransactional operation.

NTMS1 Internals

- State variables
  - `status[x]`
  - `ops[x]`
  - `opInv[x]`
  - `invokedCommit[x]`
  - `extOrder`
  - `tmHavoc`
    - set if a race is detected (new internal action: observeRace)
    - every output action is enabled when `tmHavoc` is set
- Validity preconditions
  - validCommit
  - validFail
  - tValidResp
  - nValidResp

ORACLE

 | |

Data-race-free Clients

- Add `cHavoc` flag
  - set when violation of TM correctness is detected
  - internal action: observeIncorrectTM
  - every output action is enabled when `cHavoc` is set

Data-race-free Clients

- Add `cHavoc` flag
  - set when violation of TM correctness is detected
  - internal action: observeIncorrectTM
  - every output action is enabled when `cHavoc` is set

TM clients specify required correctness condition (may be weaker than actual TM guarantee)

ORACLE

NTMS1 with Data-race-free Clients

- Proved that this is equivalent to same clients with strongly atomic TM
  - nontransactional operations equivalent to committed “mini-transaction”

NTMS1 with Data-race-free Clients

- Proved that this is equivalent to same clients with strongly atomic TM
  - nontransactional operations equivalent to committed "mini-transaction"
- No dependency on conflict relation!
  - change in conflict relation shifts burden between clients and TM
  - empty conflict relation = strongly atomic TM
  - total conflict relation = completely synchronized shared memory access

ORACLE

Privatization-safety

- A shared memory location that is made private by a transaction can be accessed without instrumentation after transaction commits.
- NTMS1 does not guarantee privatization-safety.
- No precise definition of privatization-safety exists.
- Privatization-safety can’t be specified without changing interface!
  – it restricts internal TM details

ORACLE

 |

Support for Transactions in C++

ORACLE

Transactional Memory for C++

- Developed by SG5
  - evolved from *Draft Specification for Transactional Constructs in C++* (written by industry group)
- Intended to provide pragmatic basic set of features
  - omits/simplifies several controversial/complicated features of *Draft Spec*

Disclaimer: Opinions/interpretations are my own.
They do not represent the position of my employer,
and may differ from others in SG5.

Atomicity and its discontents

- Transaction is indivisible (appears to occur at a single point)
  - within transaction: no outside interference
  - outside transaction: no partial effects/intermediate states observed
  - transaction either completes or has no effect
- Races
- Transaction-unsafe code
- Exceptions

Races

- Accesses within transactions do not race with each other.
- Transactional accesses may race with nontransactional accesses.
  - require additional synchronization to avoid data races
- Racy programs have undefined behavior.

Races

- Accesses within transactions do not race with each other.
- Transactional accesses may race with nontransactional accesses.
  - require additional synchronization to avoid data races
- Racy programs have undefined behavior.

Why is there a data race if transactions are atomic?

ORACLE

Transaction-unsafe code

- Some operations are difficult, expensive or impossible to execute atomically.
  - I/O
  - access to volatiles, atomic variables
  - asm

Transaction-unsafe code

- Some operations are difficult, expensive or impossible to execute atomically.
  - I/O
  - access to volatiles, atomic variables
  - asm

Implementation approaches:
- implicit global lock
- speculative execution

Transaction-unsafe code

- Some operations are difficult, expensive or impossible to execute atomically.
  - I/O
  - access to volatiles, atomic variables
  - asm
- Two approaches
  - forbid transaction-unsafe code within transaction
  - allow transaction-unsafe code, relax atomicity guarantee

Two kinds of transactions

- Atomic transactions
  - will appear atomic (guaranteed at translation time)
  - must not contain transaction-unsafe code
- Relaxed transactions
  - as if taking global mutex + no atomic transaction takes effect concurrently
  - any code permitted
  - not guaranteed to appear atomic (hence “relaxed”)

ORACLE

Two kinds of transactions

- Atomic transactions
  - will appear atomic (guaranteed at translation time)
  - must not contain transaction-unsafe code
- Relaxed transactions
  - as if taking global mutex + no atomic transaction takes effect concurrently
  - any code permitted
  - not guaranteed to appear atomic (hence “relaxed”)

No data races between transactional accesses

ORACLE

 |

Two kinds of transactions

- Atomic tran[illegible]ns
  - will app[illegible]ranteed at [illegible])
  - must not co[illegible]
- Relaxed transactio[illegible]
  - as if takin[illegible]on takes effect concurrently
  - any coo[illegible]
  - not guara[illegible]d to appear atomic (hence "re[illegible]d")

No data races between transactional accesses

ORACLE

Two kinds of transactions

- Atomic blocks
  - will appear atomic (guaranteed at translation time)
  - must not contain transaction-unsafe code
- Synchronized blocks
  - as if taking global mutex + no atomic transaction takes effect concurrently
  - any code permitted
  - not guaranteed to appear atomic

No data races between accesses in atomic and synchronized blocks.

ORACLE

 |

Synchronized blocks

- Allows transaction-unsafe code
- Some uses:
  - logging, error reporting
  - accessing mutex-protected resources
  - use of shared_ptr (which uses atomics)
  - “pure” functions that use helper threads
- Provides alternative to mutexes in many cases

Synchronized block example

```
int i = 0;

void f() {
  synchronized {
    if (unlikely_condition)
      std::cerr << “oops” << std::endl;
    ++i;
  }
}
```

ORACLE

 | |

Challenges for atomic blocks

- Checking for transaction-unsafe code
  – how to check function calls
- Handling escaping exceptions
  – commit or cancel?

Guaranteeing atomicity: transaction-safe code

- Some code is difficult, expensive, or impossible to execute atomically.
  - I/O, atomics, volatile, asm
- Such transaction-unsafe code is forbidden within atomic blocks.
  - guarantees atomicity, checked at translation time
  - easy for lexically enclosed code
  - what about function calls?

Transaction-safety for function calls

- Named functions
  - easy if definition is available
  - annotate declaration
  - otherwise, assume safe: check at link time (name mangling)
- Virtual functions
  - annotate declaration
- Function pointers
  - annotate declaration + extend type system

ORACLE

Transaction-safety for named functions

```
void f1() transaction_safe;
void f2();

void g() {
  atomic {
    f1();  // ok
    f2();  // ok iff defn of “f2” has no unsafe code
  }
}
```

NB: not final form

ORACLE

 | |

Transaction-safety for named functions

```
void f1() transaction_safe;  // header file

void f1() {
  volatile v = 0;  // error: unsafe code
}

void f2() {
  volatile v = 0;  // mangled name of “f2” prevents
}                  // use inside transactions
```


Transaction-safety for virtual functions

```
struct S {
  virtual void f() transaction_safe;
};

struct D : S {
  void f() {        // implicitly declared transaction-safe
    volatile v = 0; // error
  }
};
```


Transaction-safety for function pointers

```
void f() transaction_safe;
void g();
void (*pf1)() = &f;                      // ok
void (*pf2)() transaction_safe = &f; // ok
void (*pg)() transaction_safe = &g;  // ok iff defn of g is safe

void h() {
  atomic {
    (*pf1)();   // error
    (*pf2)();   // ok
  }
}
```

NB: not final form

Explicitly transaction-unsafe functions

- May explicitly declare functions transaction_unsafe
  - documents intention
  - reduces code bloat (i.e., generating superfluous “safe” variant)

```
void f() transaction_unsafe;
```

Transaction-safety of standard library

- `memcpy`, `memset`, etc.
- `malloc` and `free`
- `new` and `delete`
- `abort`

- containers (e.g., `vector`, `string`)

Transaction-safety for function calls: Summary

- Calls to named functions are considered safe unless
  - definition is available and contains transaction-unsafe code, or
  - declaration is explicitly annotated as `transaction_unsafe`.
- Assumption of transaction-safety checked at link time.
- Calls to virtual functions or through function pointers
  - safe only if declared `transaction_safe`.
- Some standard library functions are transaction-safe.

Exceptions

- What happens if an exception is thrown out of an atomic transaction?

Transaction example

```
void Account::deposit(double amount) {
  atomic {
    this->balance += amount;
    this->deposit_log.push_back(amount);
  }
}

void transfer(Account &from, Account &to, double amount) {
  atomic {
    from.deposit(-amount);
    to.deposit(amount);
  }
}
```

NB: not final form

Exceptions

- What happens if an exception is thrown out of an atomic transaction?
  - commit: transaction’s effects made visible
    - simple to specify
    - programmer must provide exception-safety
  - cancel: transaction’s effects discarded (but throws exception)
    - provides strong exception-safety
    - exception “leaks” information
  - terminate

Exceptions

- Specify how to handle exceptions with additional keyword:
  - `noexcept`
  - `commit_except`
  - `cancel_except`

Exceptions

- Specify how ... ndle exceptions with ad ... al keyword:
  - noexc...
  - commit_ex...
  - cancel_except...

ORACLE

Exceptions

- Augment `atomic` keyword:
  - `atomic_noexcept`
  - `atomic_commit`
  - `atomic_cancel`

Canceling a transaction on exception

- Exception: “cannot complete operation”
- Transaction: “complete operation, or do nothing”
  - exception indicates if and why operation is not done (e.g., bad_alloc)
- Exception “leaks” information about transaction
  - no problem for scalar types
  - what about pointers to objects constructed/modified by transaction?

ORACLE

Transaction example revisited

```
void Account::deposit(double amount) {
  atomic_cancel {
    this->balance += amount;
    this->deposit_log.push_back(amount);
  }
}

void transfer(Account &from, Account &to, double amount) {
  atomic_cancel {
    from.deposit(-amount);
    to.deposit(amount);
  }
}
```

ORACLE

Exceptions: Summary

- Atomic blocks must specify how to handle exceptions
  - atomic_noexcept
  - atomic_commit
  - atomic_cancel (works for only "transaction-safe" exceptions)
- Synchronized blocks always commit on exception

Conclusion

Summary

- Precise specifications for transactional memory
  - formal framework for reasoning about TM
- Different specifications appropriate for different contexts
- TM must be integrated with other parts of the system

ORACLE

The Future of Transactional Memory

- Improving transactional memory implementations
  - integrate with other parts of the system
- Using transactional memory effectively
  - education
  - linguistic support
- Reasoning about transactional memory
  - precise specifications
  - formal framework

ORACLE

Hardware and Software

ORACLE

Engineered to Work Together

ORACLE®