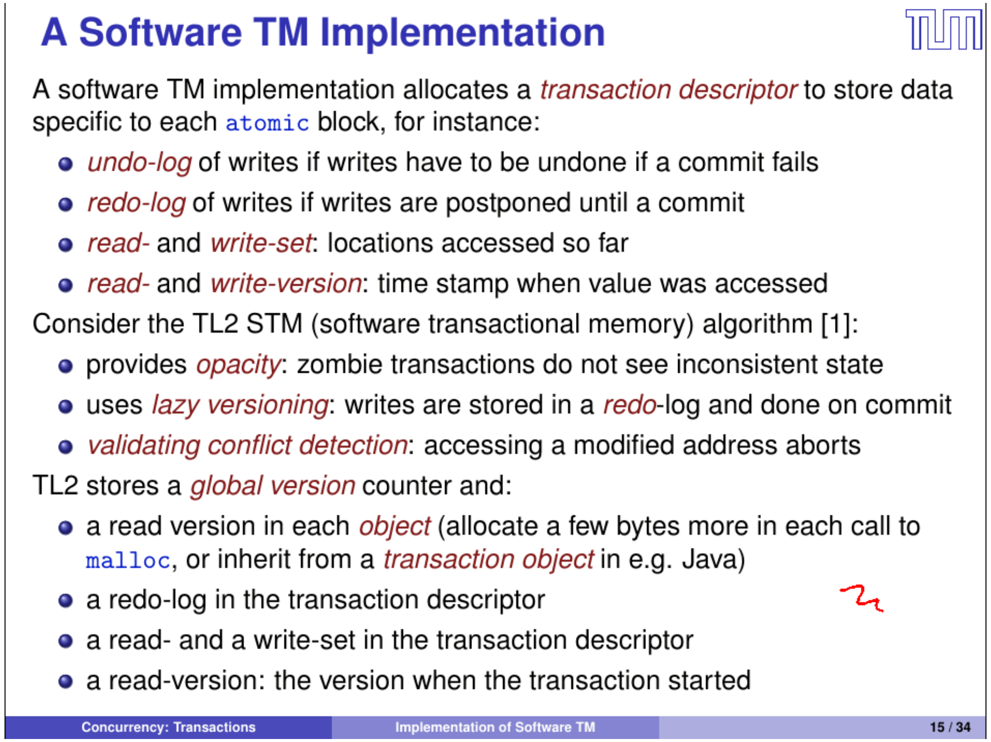
Script **generated by TTT**

Title: Petter: Programmiersprachen (12.11.2014)

Date: Wed Nov 12 14:14:31 CET 2014

Duration: 82:01 min

Pages: 86



A Software TM Implementation

A software TM implementation allocates a *transaction descriptor* to store data specific to each `atomic` block, for instance:

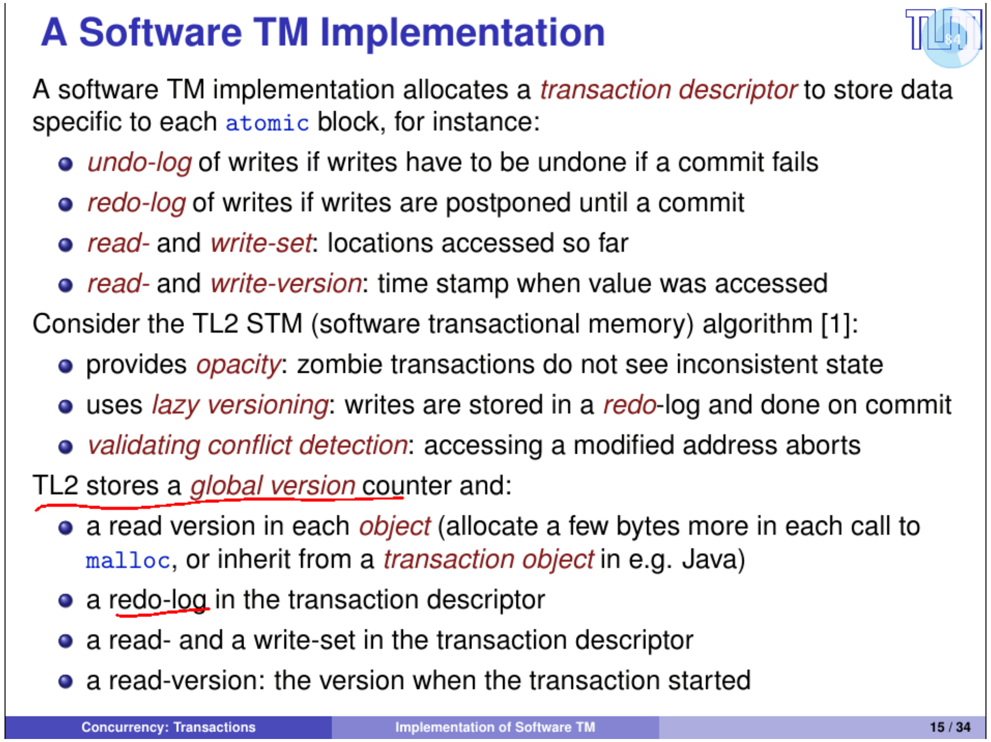
- *undo-log* of writes if writes have to be undone if a commit fails
- *redo-log* of writes if writes are postponed until a commit
- *read-* and *write-set*: locations accessed so far
- *read-* and *write-version*: time stamp when value was accessed

Consider the TL2 STM (software transactional memory) algorithm [1]:

- provides *opacity*: zombie transactions do not see inconsistent state
- uses *lazy versioning*: writes are stored in a *redo*-log and done on commit
- *validating conflict detection*: accessing a modified address aborts

TL2 stores a *global version* counter and:

- a read version in each *object* (allocate a few bytes more in each call to `malloc`, or inherit from a *transaction object* in e.g. Java)
- a redo-log in the transaction descriptor
- a read- and a write-set in the transaction descriptor
- a read-version: the version when the transaction started



A Software TM Implementation

A software TM implementation allocates a *transaction descriptor* to store data specific to each `atomic` block, for instance:

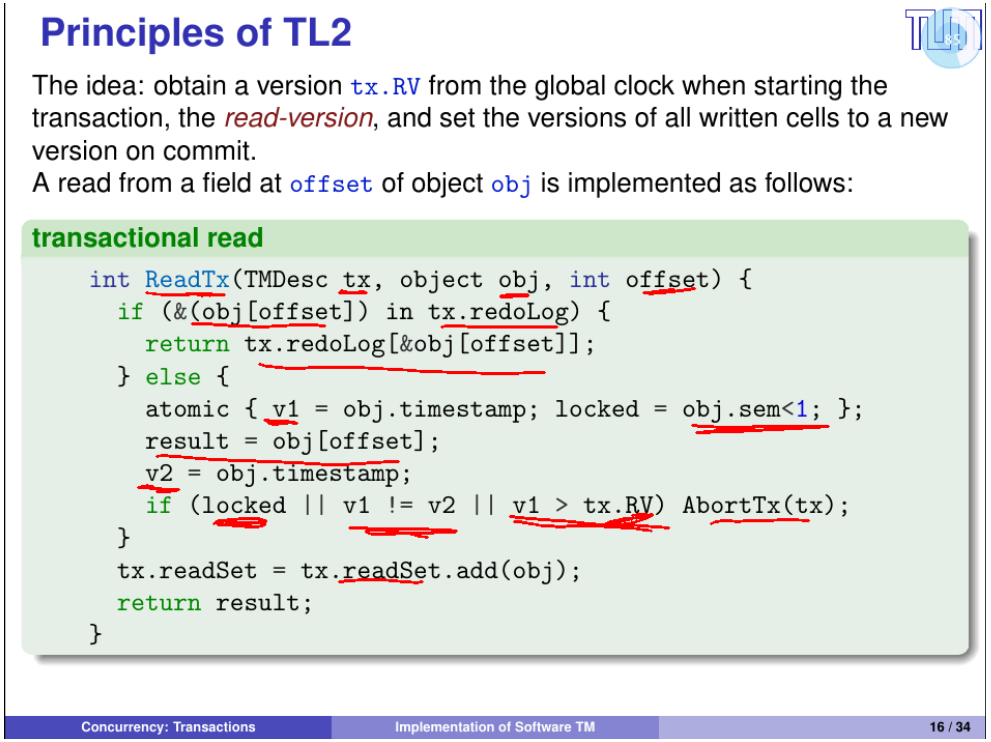
- *undo-log* of writes if writes have to be undone if a commit fails
- *redo-log* of writes if writes are postponed until a commit
- *read-* and *write-set*: locations accessed so far
- *read-* and *write-version*: time stamp when value was accessed

Consider the TL2 STM (software transactional memory) algorithm [1]:

- provides *opacity*: zombie transactions do not see inconsistent state
- uses *lazy versioning*: writes are stored in a *redo*-log and done on commit
- *validating conflict detection*: accessing a modified address aborts

TL2 stores a *global version* counter and:

- a read version in each *object* (allocate a few bytes more in each call to `malloc`, or inherit from a *transaction object* in e.g. Java)
- a redo-log in the transaction descriptor
- a read- and a write-set in the transaction descriptor
- a read-version: the version when the transaction started



Principles of TL2

The idea: obtain a version `tx.RV` from the global clock when starting the transaction, the *read-version*, and set the versions of all written cells to a new version on commit.
A read from a field at `offset` of object `obj` is implemented as follows:

transactional read

```
int ReadTx(TMDesc tx, object obj, int offset) {
  if (&(obj[offset]) in tx.redoLog) {
    return tx.redoLog[&obj[offset]];
  } else {
    atomic { v1 = obj.timestamp; locked = obj.sem<1; };
    result = obj[offset];
    v2 = obj.timestamp;
    if (locked || v1 != v2 || v1 > tx.RV) AbortTx(tx);
  }
  tx.readSet = tx.readSet.add(obj);
  return result;
}
```

Committing a Transaction

A transaction can succeed if none of the read locations has changed:

committing a transaction

```
bool CommitTx(TMDesc tx) {
  foreach (e in tx.writeSet)
    if (!try_wait(e.obj.sem)) goto Fail;
  WV = FetchAndAdd(&globalClock);
  foreach (e in tx.readSet)
    if (e.obj.version > tx.RV) goto Fail;
  foreach (e in tx.redoLog)
    e.obj[e.offset] = e.value;
  foreach (e in tx.writeSet) {
    e.obj = WV; signal(e.obj.sem);
  }
  return true;
Fail:
  // signal all acquired semaphores
  return false;
}
```

Principles of TL2

The idea: obtain a version `tx.RV` from the global clock when starting the transaction, the *read-version*, and set the versions of all written cells to a new version on commit.
A read from a field at `offset` of object `obj` is implemented as follows:

transactional read

lock+update+version+unlock

```
int ReadTx(TMDesc tx, object obj, int offset) {
  if (&(obj[offset]) in tx.redoLog) {
    return tx.redoLog[&obj[offset]];
  } else {
    atomic { v1 = obj.timestamp; locked = obj.sem<1; };
    result = obj[offset];
    v2 = obj.timestamp;
    if (locked || v1 != v2 || v1 > tx.RV) AbortTx(tx);
  }
  tx.readSet = tx.readSet.add(obj);
  return result;
}
```

`WriteTx` is simpler: add or update the location in the redo-log.

Committing a Transaction

A transaction can succeed if none of the read locations has changed:

committing a transaction

```
bool CommitTx(TMDesc tx) {
  foreach (e in tx.writeSet)
    if (!try_wait(e.obj.sem)) goto Fail;
  WV = FetchAndAdd(&globalClock);
  foreach (e in tx.readSet)
    if (e.obj.version > tx.RV) goto Fail;
  foreach (e in tx.redoLog)
    e.obj[e.offset] = e.value;
  foreach (e in tx.writeSet) {
    e.obj = WV; signal(e.obj.sem);
  }
  return true;
Fail:
  // signal all acquired semaphores
  return false;
}
```

General Challenges when using TM

Executing `atomic` blocks by repeatedly trying to executing them non-atomically creates new problems:

- a transaction might unnecessarily be aborted
  - the granularity of what is locked might be too large
  - a TM implementation might impose restrictions:

```
// Thread 1                      // Thread 2
atomic { // clock=12
  ...
                                 atomic {
                                   WriteTx(&x,0) = 42; // clock=13
                                 }

  int r = ReadTx(&x,0);
} // tx.RV=12/=clock (X)
```

General Challenges when using TM

Executing atomic blocks by repeatedly trying to executing them non-atomically creates new problems:

- a transaction might unnecessarily be aborted
  - the granularity of what is locked might be too large
  - a TM implementation might impose restrictions:

```
// Thread 1                    // Thread 2
atomic { // clock=12
  ...                          atomic {
                                 WriteTx(&x,0) = 42; // clock=13
                               }
  int r = ReadTx(&x,0);
} // tx.RV=12/=clock
```

- lock-based commits can cause contention

General Challenges when using TM

Executing atomic blocks by repeatedly trying to executing them non-atomically creates new problems:

- a transaction might unnecessarily be aborted
  - the granularity of what is locked might be too large
  - a TM implementation might impose restrictions:

```
// Thread 1                    // Thread 2
atomic { // clock=12
  ...                          atomic {
                                 WriteTx(&x,0) = 42; // clock=13
                               }
  int r = ReadTx(&x,0);
} // tx.RV=12/=clock
```

- lock-based commits can cause contention
  - organize cells that participate in a transaction in one object
  - compute a new object as result of a transaction

General Challenges when using TM

Executing atomic blocks by repeatedly trying to executing them non-atomically creates new problems:

- a transaction might unnecessarily be aborted
  - the granularity of what is locked might be too large
  - a TM implementation might impose restrictions:

```
// Thread 1                    // Thread 2
atomic { // clock=12
  ...                          atomic {
                                 WriteTx(&x,0) = 42; // clock=13
                               }
  int r = ReadTx(&x,0);
} // tx.RV=12/=clock
```

- lock-based commits can cause contention
  - organize cells that participate in a transaction in one object
  - compute a new object as result of a transaction
  - atomically replace a pointer to the old object with a pointer to the new object if the old object has not changed

General Challenges when using TM

Executing atomic blocks by repeatedly trying to executing them non-atomically creates new problems:

- a transaction might unnecessarily be aborted
  - the granularity of what is locked might be too large
  - a TM implementation might impose restrictions:

```
// Thread 1                    // Thread 2
atomic { // clock=12
  ...                          atomic {
                                 WriteTx(&x,0) = 42; // clock=13
                               }
  int r = ReadTx(&x,0);
} // tx.RV=12/=clock
```

- lock-based commits can cause contention
  - organize cells that participate in a transaction in one object
  - compute a new object as result of a transaction
  - atomically replace a pointer to the old object with a pointer to the new object if the old object has not changed
  - ⇝ idea of the original STM proposal
- TM system should figure out which memory locations must be logged

General Challenges when using TM

Executing atomic blocks by repeatedly trying to executing them non-atomically creates new problems:

- a transaction might unnecessarily be aborted
  - the granularity of what is locked might be too large
  - a TM implementation might impose restrictions:

```
// Thread 1                  // Thread 2
atomic { // clock=12
  ...                        atomic {
                               WriteTx(&x,0) = 42; // clock=13
                             }
  int r = ReadTx(&x,0);
} // tx.RV=12/=clock
```

- lock-based commits can cause contention
  - organize cells that participate in a transaction in one object
  - compute a new object as result of a transaction
  - atomically replace a pointer to the old object with a pointer to the new object if the old object has not changed
  - ⇝ idea of the original STM proposal
- TM system should figure out which memory locations must be logged
- danger of live-locks: transaction B might abort A which might abort B . . .

Integrating Non-TM Resources

Allowing access to other resources than memory inside an atomic block poses problems:

- storage management, condition variables, volatile variables, input/output
- semantics should be as if atomic implements SLA or TSC semantics

Integrating Non-TM Resources

Allowing access to other resources than memory inside an atomic block poses problems:

- storage management, condition variables, volatile variables, input/output
- semantics should be as if atomic implements SLA or TSC semantics

Usual choice is one of the following:

- *Prohibit It.* Certain constructs do not make sense. Use compiler to reject these programs.
- *Execute It.* I/O operations may only happen in some runs (e.g. file writes usually go to a buffer). Abort if I/O happens.
- *Irrevocably Execute It.* Universal way to deal with operations that cannot be undone: enforce that this transaction terminates (possibly before starting) by making all other transactions conflict.
- *Integrate It.* Re-write code to be transactional: error logging, writing data to a file, ...

Integrating Non-TM Resources

Allowing access to other resources than memory inside an atomic block poses problems:

- storage management, condition variables, volatile variables, input/output
- semantics should be as if atomic implements SLA or TSC semantics

Usual choice is one of the following:

- *Prohibit It.* Certain constructs do not make sense. Use compiler to reject these programs.
- *Execute It.* I/O operations may only happen in some runs (e.g. file writes usually go to a buffer). Abort if I/O happens.
- *Irrevocably Execute It.* Universal way to deal with operations that cannot be undone: enforce that this transaction terminates (possibly before starting) by making all other transactions conflict.
- *Integrate It.* Re-write code to be transactional: error logging, writing data to a file, ...

⇝ currently best to use TM only for memory; check if TM supports irrevocable transactions

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

⇝ limited by fixed hardware resources, a software backup must be provided

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

⇝ limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

⇝ limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - requires separate transaction instructions

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

⇝ limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - requires separate transaction instructions
   - ⇝ a transaction has to be translated differently
   - ⚠ mixing transactional and non-transactional accesses is problematic

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

⇝ limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - requires separate transaction instructions
   - ⇝ a transaction has to be translated differently
   - ⚠ mixing transactional and non-transactional accesses is problematic
2. Implicit Transactional HTM: only the beginning and end of a transaction are marked

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

⇝ limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - requires separate transaction instructions
   - ⇝ a transaction has to be translated differently
   - ⚠ mixing transactional and non-transactional accesses is problematic
2. Implicit Transactional HTM: only the beginning and end of a transaction are marked
   - same instructions can be used, hardware interprets them as transactional

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

⇝ limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - requires separate transaction instructions
   - ⇝ a transaction has to be translated differently
   - ⚠ mixing transactional and non-transactional accesses is problematic
2. Implicit Transactional HTM: only the beginning and end of a transaction are marked
   - same instructions can be used, hardware interprets them as transactional
   - only instructions affecting memory that can be cached can be executed transactionally

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

⇝ limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - requires separate transaction instructions
   - ⇝ a transaction has to be translated differently
   - ⚠ mixing transactional and non-transactional accesses is problematic
2. Implicit Transactional HTM: only the beginning and end of a transaction are marked
   - same instructions can be used, hardware interprets them as transactional
   - only instructions affecting memory that can be cached can be executed transactionally
   - hardware access, OS calls, page table changed, etc. all abort a transaction

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

⇝ limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - requires separate transaction instructions
   - ⇝ a transaction has to be translated differently
   - ⚠ mixing transactional and non-transactional accesses is problematic
2. Implicit Transactional HTM: only the beginning and end of a transaction are marked
   - same instructions can be used, hardware interprets them as transactional
   - only instructions affecting memory that can be cached can be executed transactionally
   - hardware access, OS calls, page table changed, etc. all abort a transaction
   - provides *strong isolation*

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's Haswell microarchitecture (from Sep 2013 to Aug 2014):

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's Haswell microarchitecture (from Sep 2013 to Aug 2014):

- *implicit transactional*, can use normal instructions within transactions

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's Haswell microarchitecture (from Sep 2013 to Aug 2014):

- *implicit transactional*, can use normal instructions within transactions
- tracks read/write set using a single *transaction* bit on cache lines

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's Haswell microarchitecture (from Sep 2013 to Aug 2014):

- *implicit transactional*, can use normal instructions within transactions
- tracks read/write set using a single *transaction* bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's Haswell microarchitecture (from Sep 2013 to Aug 2014):

- *implicit transactional*, can use normal instructions within transactions
- tracks read/write set using a single *transaction* bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)
- use a simple counter to support nested transactions

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's Haswell microarchitecture (from Sep 2013 to Aug 2014):

- *implicit transactional*, can use normal instructions within transactions
- tracks read/write set using a single *transaction* bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)
- use a simple counter to support nested transactions
- may abort at any time due to lack of resources

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's Haswell microarchitecture (from Sep 2013 to Aug 2014):

- *implicit transactional*, can use normal instructions within transactions
- tracks read/write set using a single *transaction* bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)
- use a simple counter to support nested transactions
- may abort at any time due to lack of resources
- aborting in an inner transaction means aborting all of them

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's Haswell microarchitecture (from Sep 2013 to Aug 2014):

- *implicit transactional*, can use normal instructions within transactions
- tracks read/write set using a single *transaction* bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)
- use a simple counter to support nested transactions
- may abort at any time due to lack of resources
- aborting in an inner transaction means aborting all of them

Intel provides two software interfaces to TM:

1. Restricted Transactional Memory (RTM)
2. Hardware Lock Elision (HLE)

Restricted Transactional Memory (Intel)

Provides new instructions XBEGIN, XEND, XABORT, and XTEST:

- XBEGIN takes an instruction address where execution continues if the transaction aborts

Restricted Transactional Memory (Intel)

Provides new instructions XBEGIN, XEND, XABORT, and XTEST:

- XBEGIN takes an instruction address where execution continues if the transaction aborts
- XEND commits the transaction started by the last XBEGIN

Restricted Transactional Memory (Intel)

Provides new instructions XBEGIN, XEND, XABORT, and XTEST:

- XBEGIN takes an instruction address where execution continues if the transaction aborts
- XEND commits the transaction started by the last XBEGIN
- XABORT aborts the current transaction with an error code

Restricted Transactional Memory (Intel)

Provides new instructions XBEGIN, XEND, XABORT, and XTEST:

- XBEGIN takes an instruction address where execution continues if the transaction aborts
- XEND commits the transaction started by the last XBEGIN
- XABORT aborts the current transaction with an error code
- XTEST checks if the processor is executing transactionally

The instruction XBEGIN can be implemented as a C function:

```
int data[100]; // shared
void update(int idx, int value) {
  if(_xbegin()==-1) {
      data[idx] += value;
      _xend();
    } else {
      // transaction failed
    }
}
```

Restricted Transactional Memory (Intel)

Provides new instructions XBEGIN, XEND, XABORT, and XTEST:

- XBEGIN takes an instruction address where execution continues if the transaction aborts
- XEND commits the transaction started by the last XBEGIN
- XABORT aborts the current transaction with an error code
- XTEST checks if the processor is executing transactionally

The instruction XBEGIN can be implemented as a C function:

```
int data[100]; // shared
void update(int idx, int value) {
  if(_xbegin()==-1) {
      data[idx] += value;
      _xend();
    } else {
      // transaction failed
    }
}
```

⇝ user must provide fall-back code

Considerations for the Fall-Back Path

Consider executing the following code in parallel with itself:

```
int data[100]; // shared
void update(int idx, int value) {
  if(_xbegin()==-1) {
      data[idx] += value;
      _xend();
    } else {
      data[idx] += value;
    }
}
```

Considerations for the Fall-Back Path

Consider executing the following code in parallel with itself:

```
int data[100]; // shared
void update(int idx, int value) {
  if(_xbegin()==-1) {
      data[idx] += value;
      _xend();
    } else {
      data[idx] += value;
    }
}
```

Problem:

- if the fall-back code is executed, it might be interrupted by the transaction
- the write in the fall-back path thereby overwrites the value of the transaction

Protecting the Fall-Back Path

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

```
int data[100]; // shared
int mutex;
void update(int idx, int value) {
  if(_xbegin()==-1) {
      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

Protecting the Fall-Back Path

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

```
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

Transactional operation:

- augment each cache line with an extra bit T

Implementing RTM using the Cache

Transactional operation:

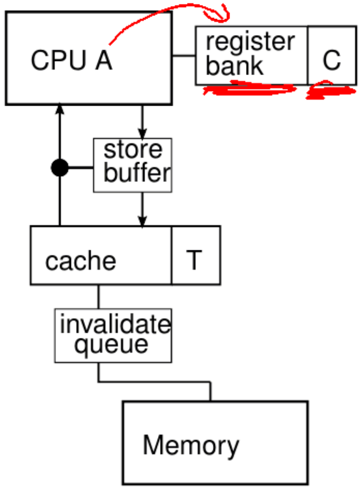
- augment each cache line with an extra bit T
- use a nesting counter C and a backup register set

Implementing RTM using the Cache

Transactional operation:

- augment each cache line with an extra bit T
- use a nesting counter C and a backup register set

⇝ additional transaction logic:

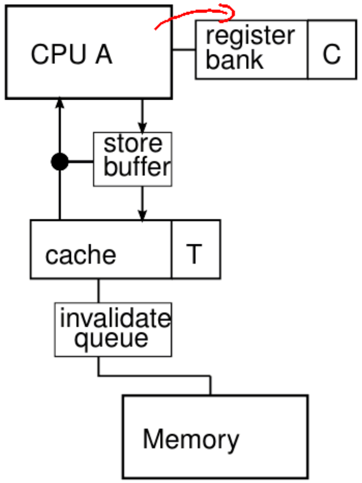


Implementing RTM using the Cache

Transactional operation:

- augment each cache line with an extra bit T
- use a nesting counter C and a backup register set

$\leadsto$ additional transaction logic:



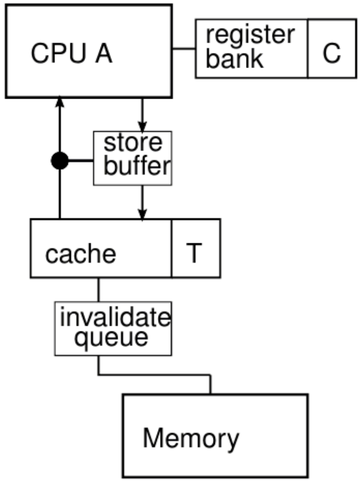
- `XBEGIN` increment C and, if $C = 0$, back up registers

Implementing RTM using the Cache

Transactional operation:

- augment each cache line with an extra bit T
- use a nesting counter C and a backup register set

$\leadsto$ additional transaction logic:



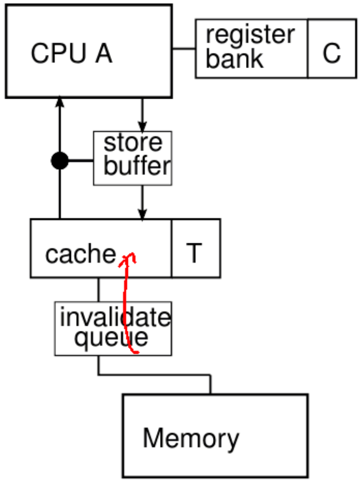
- `XBEGIN` increment C and, if $C = 0$, back up registers
- read or write access to a cache line sets T if $C > 0$

Implementing RTM using the Cache

Transactional operation:

- augment each cache line with an extra bit T
- use a nesting counter C and a backup register set

$\leadsto$ additional transaction logic:



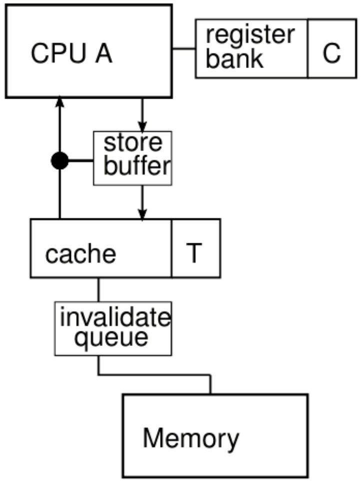
- `XBEGIN` increment C and, if $C = 0$, back up registers
- read or write access to a cache line sets T if $C > 0$
- applying an *invalidate* message from *invalidate queue* to a cache line with $T = 1$ issues `XABORT`

Implementing RTM using the Cache

Transactional operation:

- augment each cache line with an extra bit T
- use a nesting counter C and a backup register set

$\leadsto$ additional transaction logic:



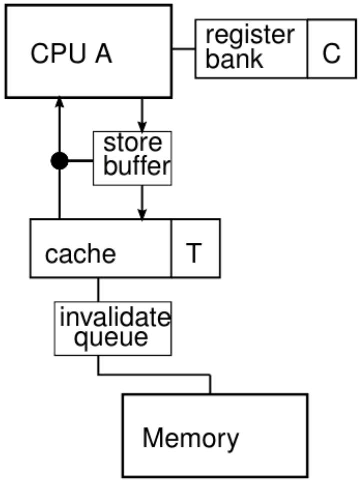
- `XBEGIN` increment C and, if $C = 0$, back up registers
- read or write access to a cache line sets T if $C > 0$
- applying an *invalidate* message from *invalidate queue* to a cache line with $T = 1$ issues `XABORT`
- observing a *read* message for a *modified* cache line with $T = 1$ issues `XABORT`

Implementing RTM using the Cache

Transactional operation:

- augment each cache line with an extra bit T
- use a nesting counter C and a backup register set

⇝ additional transaction logic:



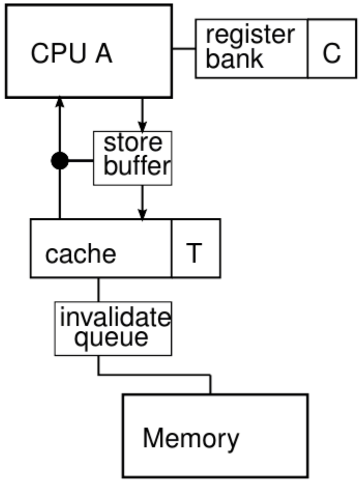
- XBEGIN increment C and, if $C = 0$, back up registers
- read or write access to a cache line sets T if $C > 0$
- applying an *invalidate* message from *invalidate queue* to a cache line with $T = 1$ issues XABORT
- observing a *read* message for a *modified* cache line with $T = 1$ issues XABORT
- XABORT clears all T flags, sets $C = 0$ and restores CPU registers

Implementing RTM using the Cache

Transactional operation:

- augment each cache line with an extra bit T
- use a nesting counter C and a backup register set

⇝ additional transaction logic:



- XBEGIN 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

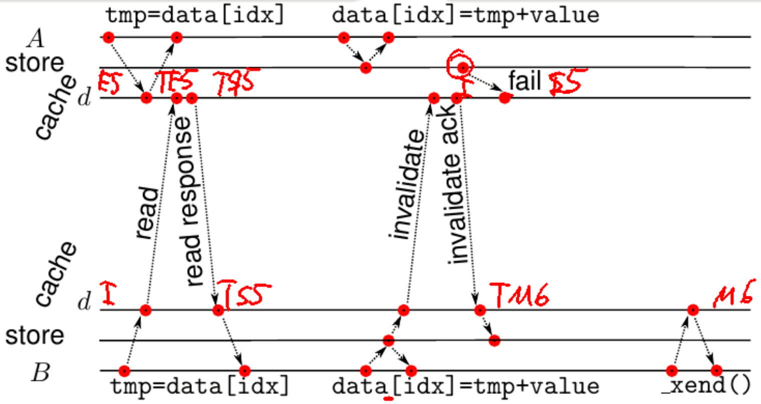
Augment MESI state with extra bit T per cache line. CPU A: E5, CPU B: I

Thread A

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

Thread B

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



Common Code Pattern for Mutexes

Using HTM in order to implement mutex:

```
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);
  }
}
```

```
void update(int idx, int val) {
  lock(mutex);
  data[idx] += val;
  unlock(mutex);
}
void lock(int mutex) {
  if(_xbegin()==-1)
    if (mutex>0) _xabort();
    else return;
  wait(mutex);
}
void unlock(int mutex) {
    if (mutex>0) 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
⇝ provide special handling in hardware: HLE

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
  - instruction that increments the semaphore must be prefixed with `XRELEASE`

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
  - instruction that increments the semaphore must be prefixed with `XRELEASE`
  - these prefixes are ignored on older platforms

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
  - instruction that increments the semaphore 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
⇝ provide special handling in hardware: HLE

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
  - instruction that increments the semaphore 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
- the memory location of the lock is locally visible as ~~0~~ 1 (“not taken”)

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
  - instruction that increments the semaphore 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
- the memory location of the lock is locally visible as 0 (“taken”)
- other processor see the lock as 1 (“not taken”)

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
  - instruction that increments the semaphore 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
- the memory location of the lock is locally visible as 0 (“taken”)
- other processor see the lock as 1 (“not taken”)
- only a finite number of locks can be elided

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
  - instruction that increments the semaphore 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
- the memory location of the lock is locally visible as 0 (“taken”)
- other processor see the lock as 1 (“not taken”)
- only a finite number of locks can be elided
- all but one elided lock may abort ⇝

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
  - instruction that increments the semaphore 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
- the memory location of the lock is locally visible as 0 (“taken”)
- other processor see the lock as 1 (“not taken”)
- only a finite number of locks can be elided
- all but one elided lock may abort ⇝
  - progress guarantee since lock is taken on abort

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
  - instruction that increments the semaphore 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
- the memory location of the lock is locally visible as 0 (“taken”)
- other processor see the lock as 1 (“not taken”)
- only a finite number of locks can be elided
- all but one elided lock may abort ⇝
  - progress guarantee since lock is taken on abort
  - no back up path is required

Hardware Lock Elision

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

- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
  - instruction that increments the semaphore 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
- the memory location of the lock is locally visible as 0 ("taken")
- other processor see the lock as 1 ("not taken")
- only a finite number of locks can be elided
- all but one elided lock may abort ⇝
  - progress guarantee since lock is taken on abort
  - no back up path is required
  - avalanche of blocked threads once elision fails

Implementing Lock Elision

Transactional operation:

- re-uses infrastructure for Restricted Transactional Memory

Implementing Lock Elision

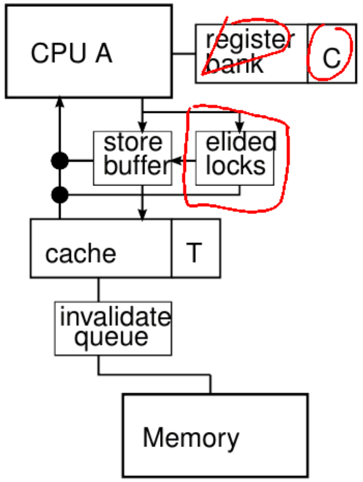
Transactional operation:

- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer

Implementing Lock Elision

Transactional operation:

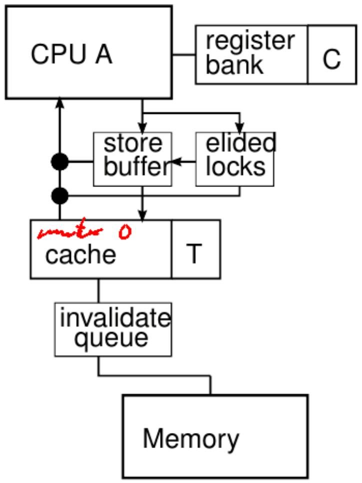
- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer



Implementing Lock Elision

Transactional operation:

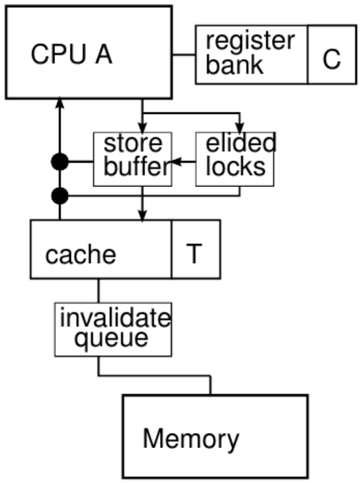
- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer
  - XACQUIRE of lock ensures *shared/exclusive* cache line state with $T = 1$, issues XBEGIN and stores written value in *elided lock* buffer



Implementing Lock Elision

Transactional operation:

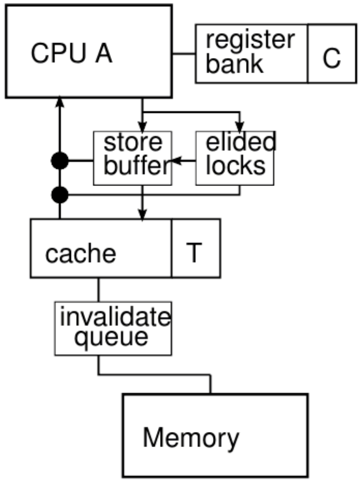
- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer
  - XACQUIRE of lock ensures *shared/exclusive* cache line state with $T = 1$, issues XBEGIN and stores written value in *elided lock* buffer
  - r/w access to a cache line sets T



Implementing Lock Elision

Transactional operation:

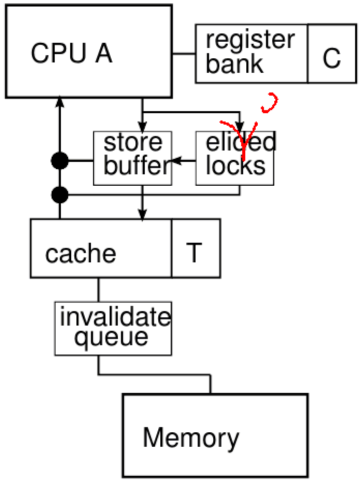
- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer
  - XACQUIRE of lock ensures *shared/exclusive* cache line state with $T = 1$, issues XBEGIN and stores written value in *elided lock* buffer
  - r/w access to a cache line sets T
  - like HLE, applying an *invalidate* message to a cache line with $T = 1$ issues XABORT, analogous for *read* message to a *modified* cache line



Implementing Lock Elision

Transactional operation:

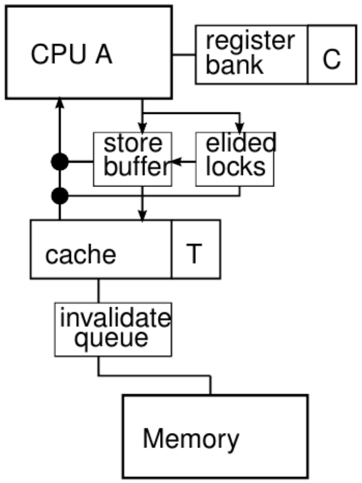
- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer
  - XACQUIRE of lock ensures *shared/exclusive* cache line state with $T = 1$, issues XBEGIN and stores written value in *elided lock* buffer
  - r/w access to a cache line sets T
  - like HLE, applying an *invalidate* message to a cache line with $T = 1$ issues XABORT, analogous for *read* message to a *modified* cache line
  - a CPU *read* to the address of the elided lock accesses the buffer (and reads 0 as if the lock was taken); cache contains 1



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 HLE, applying an *invalidate* message to a cache line with $T = 1$ issues XABORT, analogous for *read* message to a *modified* cache line
- a CPU *read* to the address of the elided lock accesses the buffer (and reads 0 as if the lock was taken); cache contains 1
- on XRELEASE on the same lock, decrement C and, if $C = 0$, clear T flags and elided locks buffer (thus, all locks contain the value 1 stored in the cache)

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

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*) Opacity
- *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 Software TM:

- converting each read/write access to shared variables is tedious
- GCC can translate accesses in __transaction_atomic regions into library calls
- the library libitm may provide different STM algorithms
- GCC implements proposal for STM in C++ using this library http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3341.pdf

TM in Practice

Availability of Software TM:

- converting each read/write access to shared variables is tedious
- GCC can translate accesses in `__transaction_atomic` regions into library calls
- the library `libitm` may provide different STM algorithms
- GCC implements proposal for STM in C++ using this library `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3341.pdf`

Use of hardware lock elision is limited:

- allows to easily convert existing locks
- `pthread` locks in `glibc` use RTM `https://lwn.net/Articles/534758/`:
  - allows implementation of back-off mechanisms
  - HLE only special case of general lock
- implementing monitors is challenging
  - lock count and thread id may lead to conflicting accesses
  - in `pthreads`: error conditions often not checked anymore

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

D. Dice, O. Shalev, and N. Shavit.
Transactional Locking II.
In *Distributed Coputing*, LNCS, pages 194–208. Springer, Sept. 2006.

T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.
Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.

Online blog entries on Intel HTM:

1. `http://software.intel.com/en-us/blogs/2013/07/25/fun-with-intel-transactional-synchronization-extensions`
2. `http://www.realworldtech.com/haswell-tm/4/`