CPEG 852 — Advanced Topics in Computing Systems
Introduction to Transactional Memory

Stéphane Zuckerman

Computer Architecture & Parallel Systems Laboratory
Electrical & Computer Engineering Dept.
University of Delaware
140 Evans Hall Newark,DE 19716, United States
szuckerm@udel.edu

December 1, 2015

Outline

Outline

A Simple Motivating Example

INSERT PSEUDO-CODE HERE

A Simple Example

A C Implementation

```
/*
 * Description: atomically adds an unsigned long
 * value to an unsigned long word in memory
 * Inputs:
 *  addr:  address where to add the new value
 *  mutex: lock to use to guarantee access to the memory word
 *         to modify
 *  value: a read-only value to add to the memory word
 * Output:
 *  The previous value contained in the unsigned long memory word
 */
unsigned long
atomic_add_ul(unsigned long* addr, pthread_mutex_t* mutex,
              const unsigned long value)
{
    unsigned long old_val = *addr;
    pthread_mutex_lock ( mutex );    // Enter critical section
    *addr += value;                  // In critical section
    pthread_mutex_unlock ( mutex ); // Leave critical section
    return old_val;
}
```

A Simple Example

A C Implementation – Busy Waiting

```
unsigned long
atomic_add_ul_busywait(unsigned long* addr, volatile uint64_t* lock,
                       const unsigned long value)
{
    // Enter critical section
    while ( __sync_bool_compare_and_swap(addr, 0UL, 1UL) == false )
        ; // do nothing

    // In the critical section
    unsigned long old_val = *addr;
    *addr += value;

    *lock = 0; // Leave the critical section

    return old_val;
}
```

A Simple Example

Busy Waiting + Exponential Backoff

```
#include <unistd.h> // for usleep(3)
#define EXP_BACKOFF_INIT_VAL 2

unsigned long
atomic_add_ul_exp_backoff(unsigned long* addr,
    volatile uint64_t* lock, const unsigned long value)
{
    // Enter the critical section
    useconds_t backoff = EXP_BACKOFF_INIT_VAL;

    while (__sync_bool_compare_and_swap(addr, 0UL, 1UL) == false) {
        usleep ( backoff ); // do nothing for "backoff" microseconds
        backoff *= 2; // double sleeping time for next attempt
    }

    // In the critical section
    unsigned long old_val = *addr;
    *addr += value;

    *lock = 0; // Leave critical section

    return old_val;
```

A Simple Example

How It Should Be Really Done in this Case

```
unsigned long
atomic_add_ul_instr
(
    unsigned long*       addr,
    const unsigned long  value
)
{
    return __sync_fetch_and_add(addr,value);
}
```

Abstracting Locks Into Critical Sections

- Critical sections using locks follow a rather similar pattern:
  1. Lock region (*i.e.*, prevent other threads from entering the region)
  2. Modify shared memory locations
  3. Unlock region (*i.e.*, allow other threads to enter the region)
- It should be possible to create a *semantic* construct that enables the programmer to express that only a single thread is allowed in a specific code region
- However, there are still *many* challenges to overcome:
  - Should the *implementation* of the critical section use a single global lock?
  - If not, how many locks should be used?
  - Do we need locks at all?

Critical Sections in Java

`synchronized` in a block of code

```
public class AtomicCounter
{
    public static void
    atomicAdd(SomeClassWithAnIntegerInside scwai, int value)
    {
        synchronized(scwaii)
        {
            scwai.setInteger(scwai.getInteger() + value);
        }
    }
}
```

Critical Sections in Java

`synchronized` to Qualify a Method

```
public class AtomicCounter
{
    public static synchronized void
    atomicAdd(SomeClassWithAnIntegerInside scwai, int value)
    {
        scwai.setInteger(scwai.getInteger() + value);
    }
}
```

Critical Sections in OpenMP

Anonymous Regions

```
unsigned long
atomic_add_ul_omp(unsigned long* addr, const unsigned long value)
{
#   pragma omp critical
    {                       // Enter critical region
        *addr += value;     // In critical region
    }                       // Leave critical region
}
```

Critical Sections in OpenMP

Named Regions

```
unsigned long
atomic_add_ul_omp_named(unsigned long* addr, const unsigned long valu
{
#   pragma omp critical XADD
    {                       // Enter critical region named "XADD"
        *addr += value;     // In critical region named "XADD"
    }                       // Leave critical region named "XADD"
}
```

Atomic Operations in OpenMP

```
unsigned long
atomic_add_ul_omp_atomic(unsigned long* addr, const unsigned long valu
{
#   pragma omp atomic
    *addr += value;
}
```

Locking Complex Data Structures

Let's assume we are accessing an array of integer values. The array can be modified as follows.

```
void
access_array(int* array, size_t idx)
{
    array[idx] = ( array[idx-1] + array[idx] + array[idx+1] ) / 3;
}
```

Locking Complex Data Structures

In a concurrent environment, there is a need to protect the cells while they are being used (*i.e.*, we want to avoid data races).

```
void
access_array(int* array, size_t idx, pthread_mutex_t* m)
{
    pthread_mutex_lock ( m );
    array[idx] = ( array[idx-1] + array[idx] + array[idx+1] ) / 3;
    pthread_mutex_unlock ( m );
}
```

Locking Complex Data Structures

In a concurrent environment, there is a need to protect the cells while they are being used (*i.e.*, we want to avoid data races).

```
void
access_array(int* array, size_t idx, pthread_mutex_t* m)
{
    pthread_mutex_lock ( m );
    array[idx] = ( array[idx-1] + array[idx] + array[idx+1] ) / 3;
    pthread_mutex_unlock ( m );
}
```

. . . But now we must lock the *whole* array to access only three elements!

Locking Complex Data Structures I

Using Fine-Grain Locking

What if we associated each element of the array with a lock? This way, it would be a simple matter of acquiring only those elements necessary to update the value of one element.

```
typedef struct {
    pthread_t* lock;
    int        array;
} LockedInteger;
#define LOCK(m)   pthread_mutex_lock( (m)->lock )
#define UNLOCK(m) pthread_mutex_unlock( (m)->lock )

void access_array(LockedInteger* array, size_t idx) {
    LOCK(array[idx-1]); LOCK(array[idx]);  LOCK(array[idx+1]);
    array[idx]->value = ( array[idx-1]->value
                        +   array[idx]->value
                        +   array[idx+1]->value ) / 3;
    UNLOCK(array[idx-1]); UNLOCK(array[idx]); UNLOCK(array[idx+1]);
}
```

- Much faster (reduces contention on a single lock, increases parallelism, . . .)
- . . . But now we must allocate *a lot of space* just to access small 4-byte words!
  - But with more complex/bigger data structures, this becomes less of a problem
  - Still, we *do* need to check for false-sharing, data alignment, and other tedious things to speed-up accesses.
- Reduces contention, but now requires to perform *three* locking operations! This can become very expensive!

Locking Complex Data Structures

A Better Fine-Grain Locking Method?

```
typedef struct {
    pthread_t* lock;
    int        array;
} LockedInteger;
#define LOCK(m)   pthread_mutex_lock( (m)->lock )
#define UNLOCK(m) pthread_mutex_unlock( (m)->lock )

void access_array(LockedInteger* array, size_t idx) {
    LOCK(array[idx-1]);  LOCK(array[idx+1]);
    array[idx]->value = ( array[idx-1]->value
                        +   array[idx]->value
                        +   array[idx+1]->value ) / 3;
    UNLOCK(array[idx-1]);  UNLOCK(array[idx+1]);
}
```

Locking Complex Data Structures

A Better Fine-Grain Locking Method?

```
typedef struct {
    pthread_t* lock;
    int        array;
} LockedInteger;
#define LOCK(m)   pthread_mutex_lock( (m)->lock )
#define UNLOCK(m) pthread_mutex_unlock( (m)->lock )

void access_array(LockedInteger* array, size_t idx) {
    LOCK(array[idx-1]);  LOCK(array[idx+1]);
    array[idx]->value = ( array[idx-1]->value
                        +   array[idx]->value
                        +   array[idx+1]->value ) / 3;
    UNLOCK(array[idx-1]);  UNLOCK(array[idx+1]);
}
```

This is a false good idea! (Why?)

Another Example: Linked Lists I

Sequential

```
typedef struct list_elt_s {
    struct list_elt_s* next;
    int                value;
} ListElt;

typedef struct linked_list_s {
    ListElt*           head;
    size_t             n_elts;
} LinkedList;

ListElt g_FIRST_ELT;

LinkedList g_LINKED_LIST = {
    .start = &g_FIRST_ELT;
    .n_elts = 0;
    .lock = g_LIST_LOCK;
};

bool contains(LinkedList* list, const int what) {
    bool found = false;
```

Another Example: Linked Lists II

Sequential

```
    for (ListElt* cur = list->head; !found && cur; cur = cur->next)
        if (cur->value == what)
            found = true;

    return found;
}
```

Another Example: Linked Lists I

Coarse-Grain Locking

```
typedef struct list_elt_s {
    struct list_elt_s* next;
    int                value;
} ListElt;

typedef struct linked_list_s {
    ListElt*           head;
    size_t             n_elts;
    pthread_mutex_t*   lock;
} LinkedList;

#define LOCK(m)    pthread_mutex_lock((m)->lock)
#define TRYLOCK(m) pthread_mutex_trylock((m)->lock)
#define UNLOCK(m)  pthread_mutex_unlock((m)->lock)

LinkedElt g_FIRST_ELT;
pthread_mutex_t g_LIST_LOCK = PTHREAD_MUTEX_INITIALIZER;
LinkedList g_LINKED_LIST = {
    .start = &g_FIRST_ELT;
    .n_elts = 0;
    .lock = g_LIST_LOCK;
```

Another Example: Linked Lists II

Coarse-Grain Locking

```
};

bool contains(LinkedList* list, const int what) {
    bool found = false;

    LOCK(list);
    for (ListElt* cur = list->head; !found && cur; cur = cur->next)
        if (cur->value == what)
            found = true;
    UNLOCK(list);

    return found;
}
```

Another Example: Linked Lists I

Fine-Grain Locking

```
typedef struct list_elt_s {
    struct list_elt_s* next;
    int                value;
    pthread_mutex_t    lock;
} ListElt;

typedef struct linked_list_s {
    ListElt*           head;
    size_t             n_elts;
    // No need for a global lock anymore...
} LinkedList;

#define LOCK(m)     pthread_mutex_lock((m)->lock)
#define TRYLOCK(m)  pthread_mutex_trylock((m)->lock)
#define UNLOCK(m)   pthread_mutex_unlock((m)->lock)

LinkedElt g_FIRST_ELT;
pthread_mutex_t g_LIST_LOCK = PTHREAD_MUTEX_INITIALIZER;
LinkedList g_LINKED_LIST = {
    .start = &g_FIRST_ELT;
    .n_elts = 0;
```

Another Example: Linked Lists II

Fine-Grain Locking

```
    // No need for a global lock anymore...
};

bool contains(LinkedList* list, const int what) {
    bool found = false;
    ListElt *cur = NULL,
            *tmp = NULL;

    LOCK(list->head);
    for (cur = list->head; !found && cur; cur = tmp) {
        found = cur->value == what;
        if (!found) {
            LOCK(cur->next);
            tmp = cur->next;
            UNLOCK(cur);
        }
    }
    UNLOCK(cur);

    return found;
}
```

Another Example: Linked Lists III

Fine-Grain Locking

Implementing Critical Sections With Locks I

A Summary

Locks are a natural solution to deal with thread synchronization when there is a need to ensure only one thread at a time can affect a set of shared memory locations.

Coarse-Grain Locking

- Pros:
  - Simple to implement
  - Does not need language constructs
  - Easy to get a correct solution
- Cons:
  - Simplistic – each time a complex data structure must be accessed/modified, there is a need to lock its *entirety*
  - Does not scale very well – if multiple threads try to access the locked data structure, they are all queued

Fine-Grain Locking

- Pros:
  - Good scalability: by extending the "lockable surface" threads tend to compete less often to access complex data structures
  - Does not need language constructs
- Cons:
  - Complex access patterns require complex locking solutions
    - Several locks must be acquired, and a specific order must be respected at all times to avoid deadlocks
    - If too fine-grained, locking can become detrimental in terms of memory management (locality, total required space, *etc.*)

A Possible Alternative to Locking: Lock-Free Data Structures

A shared data structure is *lock-free* if it does not require mutual exclusion.

- Lock-free data structures avoid *priority inversion* problems, *convoying*, and *deadlocks*.
- They rely on the use of atomic operations, *e.g.*, read-modify-write types of operations which are performed as an uninterruptible sequence.
  - Examples: `fetch-and-add`, `fetch-and-sub`, compare-and-swap; *etc.*
- Lock-free data structures and the algorithms that exploit them provide strong guarantees, such as forward progress
  - At least *one* thread will make some progress during its execution

Outline

All that follows is taken from the original paper M. Herlihy and Moss 1993. In the following, it is assumed threads execute only one transaction at a time.

Definition: Transaction

A *transaction* is a finite sequence of machine instructions, executed by a single thread, satisfying *serializability* and *atomicity*.

Definition: Serializability

Transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another. Committed transactions are never observed by different processors to execute in different orders.

Definition: Atomicity

Each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it either *commits*, making its changes visible to other threads (effectively) instantaneously, or it *aborts*, causing its changes to be discarded.

Memory Instructions

- Load-transactional (`LT`) – reads the value of a shared memory location into a private register.
- Load-transactional (`LTX`) – reads the value of a shared memory location into a private register, "hinting" that the location is likely to be updated.
- Store-transactional (`ST`) – tentatively writes a value from a private register to a shared memory location. This new value does not become visible to other processors until the transaction successfully commits.

Transaction Management Instructions

Commit (`COMMIT`) – attempts to make the transaction's tentative changes permanent.

- If it *succeeds*, all memory locations contained in the transaction's write set are made visible to all other threads. A `COMMIT` succeeds only if:
  - No other transaction has updated any location in its data set, and
  - No other transaction has read any location in its write set.
- If it *fails*, all changes to the write set are discarded.

Abort (`ABORT`) discards all updates to the write set.

Validate (`VALIDATE`) tests the current transaction status.

- Upon success: the transaction has not aborted. It may still fail later.
- Upon failure: the transaction has aborted. All tentative updates are discarded.

Transactional Memory Instruction Set III

Goal: to enable the (low-level) programmer to create read-modify-write constructs which are not limited to single-word or single-instruction types of operations.

Question: How should the system react when ordinary and transactional operations are mixed together?

How to Use Transactional Memory Constructs

To help implement lock-free data structures, the following sequence would typically be, using transactional memory operations:

1. Use `LT` or `LTX` to read from a set of locations
2. Use `VALIDATE` to check that the values read are consistent
3. Use `ST` to modify a set of locations
4. Use `COMMIT` to make the changes permanent. If either `VALIDATE` or `COMMIT` fails, return to step (1).

Implementation Overview

Transactional memory "piggy backs" on regular caches, and extends them a bit.

- Non-transactional (ordinary) operations use the same traditional caches
  - Same cache controllers
  - Same coherence protocol
- Need some custom hardware support for transactional operations
  - Only "primary" (first-level, or L1) caches need to be modified
- All `COMMIT` or `ABORT` operations, and in general transactions-related operations are kept local to the L1 cache.
  - No communication required with other caches

Taking Advantage of Existing Cache Mechanisms

- Exploitation of existing cache protocols, in particular *access rights* (*e.g.*, non-exclusive (shared) reads, or exclusive writes). At any time, a memory location is either
  1. Not immediately accessible by any processor – in memory only
  2. Accessible non-exclusively by one or more processors
  3. Accessible exclusively by a single processor
- If a protocol can detect the previous types of accesses, they can also detect transactional vs. non-transactional accesses, and if a transaction should be aborted.
- If a transaction conflict is detected, all transactions that try to revoke access of a transactional entry from another active transaction
  - This assumes that there are timer (or other) interrupts which will abort a stalled transaction after a fixed duration (Question: Why do we need this?)

Cache Architecture Overview

- Use two L1 caches:
  - Regular cache: same as found in other processors. Stores memory locations accessed through non-transactional operations.
  - Transactional cache: Stores memory locations accessed through transactional operations.
  - These caches are exclusive: if a memory location is stored in a regular cache, it cannot be found in the transactional cache, and vice-versa.
- Other (shared) caches can exist beyond L1s.
- The (small) transactional cache holds all tentative writes, *without* propagating them to other processors or to main memory unless the transaction commits.
  - If the transaction aborts, the lines holding the tentative writes are dropped (*invalidated*)
  - If it does commit, the lines may be snooped by other processors, written back to memory, *etc.*

Example Implementation

A Snoopy Cache Solution

Cache Line States

Name	Access	Shared?	Modified?
INVALID	none	—	—
VALID	R	Yes	No
DIRTY	R, W	No	Yes
RESERVED	R, W	No	No

Name	Meaning
EMPTY	Contains no data
NORMAL	Contains committed data
XCOMMIT	Discard on commit
XABORT	Discard on abort

- A modified cache line sets it to XABORT
- When a COMMIT operation succeeds:
  - All the relevant cache lines transition from XCOMMIT to EMPTY.
  - All the relevant cache lines transition from XABORT to NORMAL.
- When a COMMIT fails, or an ABORT operation is issued:
  - Entries marked as XABORT transition to EMPTY
  - Entries marked as XCOMMIT transition to NORMAL
- EMPTY lines are first to be evicted when new lines must be cached. If there is none to be found, NORMAL lines are then chosen.

Bus Cycles

Uses traditional cache coherence cycles, and adds new cycles.

- `T_READ`, `T_RFO` and `T_WRITE` are the transactional equivalent of `READ`, `READ-FOR-OWNERSHIP` (RFO), and `WRITE` in cache coherence protocols.
- `BUSY` is a way to *refuse* transactional requests. It is useful in case there is a very contended region.
  - When a transaction receives a `BUSY` signal, it aborts and retries. This prevents deadlocks or continual mutual aborts.
  - Question: What potential problems may arise from this architectural choice?

Processor Actions

- Each processor maintains two flags: `TACTIVE` and `TSTATUS`
  - Transaction Active (`TACTIVE`) indicates if a transaction is in progress
  - Transaction Status (`TSTATUS`) indicates if that transaction is active (*True*) or aborted (*False*)

Examples Using Transactional Memory

Shared Counter

```
#include <unistd.h> // for usleep(3)

unsigned long
atomic_add_tm(unsigned long* addr, unsigned long value)
{
    useconds_t backoff = BACKOFF_MIN;
    unsigned long wait = 0, old_val;
    bool        success = false;
    while (!success) {
        old_val = LTX(addr);
        ST(addr,old_val+value);
        if (COMMIT()) {
            success = true;
            backoff = BACKOFF_MIN;
        } else {
            usleep ( backoff );
            backoff *= 2;
        }
    }

    return old_val;
}
```

Examples Using Transactional Memory I

Linked List

```
typedef struct list_elt_s {
    struct list_elt_s* next;
    int                value;
} ListElt;
typedef struct linked_list_s {
    ListElt*           head;
    size_t             n_elts;
} LinkedList;
ListElt    g_FIRST_ELT;
LinkedList g_LINKED_LIST = {
    .start = &g_FIRST_ELT;
    .n_elts = 0;
};

bool contains(LinkedList* list, const int what) {
    bool        found   = false,
                success = false;
    useconds_t  backoff = BACKOFF_INIT_VAL;
    ListElt*    cur     = NULL,
                head    = list->head;
```

Examples Using Transactional Memory II

Linked List

```
    for (cur = TLX(head); !found && cur != NULL; cur = TLX(cur->next)) {
        while (!success) {
            if (VALIDATE()) {
                if (cur->value == what)
                    found = true;
                success = true;
                backoff = BACKOFF_INIT_VAL;
            } else {
                usleep(backoff);
                backoff *= 2;
            }
        }
    }

    return found;
}
```

Outline

Why Implement Transactional Memory in Software?

Shavit and Touitou 1997

- At the end of the 1990's, it became obvious that long and unknown latencies were here to stay, both for single and multiple processor systems.
- As a result, the use of traditional critical sections by means of locks was deemed unsuitable by Shavit and Touitou.
  - Limited parallelism
  - Added contention on memory and interconnect
  - Increased vulnerability to timing anomalies and processor failures
- To them, "the key to highly concurrent programming is "... by constructing classes of implementations that are *non-blocking*."
  - Non-blocking usually means *lock-free* or *wait-free*:
    - Lock-free (or non-blocking): *at least one thread* makes forward progress within a determined number of steps
    - Wait-free: *all threads* make forward progress within a fixed number of steps
  - They are obtained by decreasing the number and size of critical sections a multiprocessor program uses, sometimes down to no critical section altogether

Software Transactional Memory

- At the time STMs were proposed, there was no hardware implementation available in any multi-processor system
  - Most *atomic* operations that were available were of the `LL/SC` type.
- While STMs still have to rely on fine-grain locking or at least atomic operations, they still allow a more flexible programming style
  - Writing concurrent programs is supposedly easier with STMs

Load-Link / Store-Conditional (LL/SC)

The LL/SC combination was proposed in many high-end processors, such as the MIPS, DEC's Alpha, and IBM's PowerPC processors. They allow a processor to update a memory location in two steps:

1. `Load-Link` (LL) loads the content of a memory location in a private register. It tracks changes made to that location.
2. `Store-Conditional` attempts to write the content of some private register at the chosen memory location loaded thanks to LL. It succeeds only if nobody modified the location between the LL and SC steps.
3. This allows arbitrary code to be executed between the LL and SC steps.
4. If the LL/SC sequence is short enough, there is a good chance conflicts are going to be minimal
5. If SC fails, it is up to the programmer to repeat the LL/SC sequence.

STM in a Nutshell

- The traditional way to ensure atomicity is by using locks
  - Attempt to acquire a certain number of (memory) resources
  - If the attempt fails, release all previously acquired resources (and most often, retry)
  - If the attempt succeeds, apply the requested operation *Op*, *then* release the resources.
  - Need to ensure there are no deadlocks: need to acquire resources in the *same* order all the time
    - This helps guarantee *liveness*
    - Question: what if two threads try to acquire two different sets of resources with overlapping resources?
- Additional liveness guarantee: In a faulty environment, every transaction completes even if the thread which executes it has been delayed, swapped out, or crashed.
  - Use of *helpers* to achieve this
  - Other transactions trying to acquire the same locations will attempt to help the "faulty" transaction to complete its job.

The Problem with Herlihy's Method

- Original TM (Herlihy's method):
  - Use TM to implement a collection of changes to a series of shared objects
    - Effectively implements a multi-word compare-and-swap operation
  - "Algorithm:"
    1. Copy shared data in a new memory block
    2. Apply changes
    3. Attempt to switch the old and new data structures using `LL/SC`.
- It works for small data structures; not so much for bigger ones where data may be tempered with much more frequently.
- Other methods were proposed to help TM deal with larger data structures: Alemany and Felten 1992; Barnes 1993; LaMarca 1994; Turek, Shasha, and Prakash 1992

UNIVERSITY OF DELAWARE

Definition: Cooperation

Whenever a process needs (depends on) a location already locked by another process it helps the locking process to complete its dependency chain.

Cooperation in Practice

Cooperation *can* work, but it has two major short-comings:

- Cooperation has a recursive structure—it leads to helping disjoint sets of shared values
- A high percentage of of cooperative k-word compare-and-swap operations fail but generate contention

STMs propose to use a *transactional* approach which still relies at times on *helping*, but much less often than the cooperation method.

Transactional Memory: Recap (M. Herlihy and Moss 1993)

- Transaction: finite sequence of local and shared memory operations
- Operations: `READ_TRANSACTIONAL`, `WRITE_TRANSACTIONAL`
- *Data set* of a transaction: set of shared locations accessed by transactional operations.
- Any transaction may fail
- If a transaction is successful, the modified shared locations are made visible *atomically*

Definition: Software Transactional Memory (Shavit and Touitou 1997)

A software transactional memory (STM) is a shared object which behaves like a memory that supports multiple changes to its addresses by means of transactions.

Definition: Transaction

A transaction is a thread of control that applies a finite sequence of primitive operations to memory.

A static transaction is a special form of transaction in which the data set is known in advance.

Most synchronization procedures tend to belong to static transactions.

Wait-Free & Non-Blocking STMs

An STM is wait-free if any process which repeatedly attempts to execute a given transaction terminates successfully after a finite number of machine steps.

An STM is non-blocking if the repeated attempts to execute some transaction by a process implies that some process (not necessarily the same pone and with a possibly different transaction) will terminate successfully after a finite number of steps in the whole system.

An STM implementation is swap-tolerant if it is non-blocking and it is assumed it cannot be infinitely swapped out many times.

Concurrent System

- Concurrent *system*: collection of *processes*.
  - Processes communicate through shared data structures: *objects*.
  - Objects have sets of primitive *operations* which provide the only means to manipulate that object
  - Processes are a sequential thread of control.
    - They apply a sequence of operations to objects by issuing an invocation and receiving an associated response.
- *history*: Sequence of invocations & responses of some system execution.
  - $A \longrightarrow B$ means that an operation A precedes an operation B if A's response occurs before B's invocation *as if it happened in real-time*.
    - Sequential history: all operations are immediately followed by their associated response
- Two operations are *concurrent* $\Longrightarrow$ they are unrelated by the "real-time" order.
- Legal concurrent orderings are defined according to the *linearizability* property

Linearizability

M. P. Herlihy and Wing 1990

Definition: Linearizability

Every concurrent history is "equivalent" to some legal sequential history which is consistent with the partial real-time order induced by the concurrent history.

Linearizable implementations have operations appear to take effect atomically at some point between their invocation and corresponding response.

Note: Linearizability is a *local* property.

Question: How is linearizability different from sequential consistency?

How Well Does STM Perform? I

Shavit's Original Implementation (Shavit and Touitou 1997)

- Outperforms Herlihy's TM in simulations, BUT
- "[General] STM and other non-blocking techniques are inferior to standard *non-resilient* lock-based methods such as queue-locks Mellor-Crummey and Scott 1991
- Note that when *resiliency* is involved, STMs offer good/reasonable performance

Is STM Just a Research Toy?

The "Yes" Side (Cascaval et al. 2008)

- It has its uses in designing lock-free data structures: binary trees, hash tables
- BUT: no large scale application makes use of them
- Conclusions based on the authors' own optimized STM implementation
- Major problems with STM systems:
  - **Overheads**: too high overheads—even the most optimized ones.
    - In particular, sequential overhead is *much* higher than their lock-based systems counterparts
  - **Semantics**: transaction semantics are weakened & complicated $\Rightarrow$ programmer must be more careful. Consequences:
    - **Weak Atomicity**: semantics of atomicity are weakened to allow undetected conflicts with non-transactional accesses
    - **Privatization**: not easy or even impossible to perform once data is accessed transactionally
    - **Memory Reclamation**: regular memory allocation must be replaced with specific operations
    - **Legacy Binaries**: STM needs to observe *all* memory operations—but legacy code is already compiled

Is STM Just a Research Toy?

The Answer from the "No" Side (Dragojević et al. 2011)

- Does not deny some of the shortcomings underlined by Cascaval *et al.*
- Disputes their methodology:
  - They used only 8 threads vs. pure sequential executions
  - They only evaluated a subset of the STAMP benchmarks (created to evaluate STM frameworks)
- Produce more results, using a different configuration:
  - Used a state-of-the-art STM (SwissTM, different from IBM STM)
  - Runs all 10 STAMP benchmarks; also some STMBench benchmarks
  - Testbed: SPARC T1 processor (64 HW threads); AMD x86 processor (16 HW threads).
  - Results: as the number of threads increases, STM benefits become significant
    - Manual instrumentation $\Rightarrow$ best speedups; tedious for programmers
- Dragojević *et al.* agree that parallel programmers can only truly benefit from STM if they are handled by an STM-enabled compiler

It is still not clear that STM are truly useful—it is a trade-off between productivity and effective performance.

Outline

IBM Power 8 I

Le et al. 2015; Cain et al. 2013

Design constraints

- Single transaction per core at a time (not nested transactions)
- No guarantee of success: all transactions must provide a failure handler
- No requirement of support survival across context switches and/or paging of transactional data
- Use as much as possible the primitives already available in the POWER ISA. Deviations were allowed only if truly necessary
- Resulting implementation should be decoupled from already available atomic operations
- No architectural limits on transaction size, but encourage large transaction support by eliminating causes of transaction failures

Motivation for Implementing HTM

- Pure hardware implementation of lock elision:
  1. Optimistically acquire a lock, and
  2. Start executing the critical section without truly locking it
  3. Check if the critical section was correctly traversed
- More generally, TM is seen as an *enabler* for thread speculation
- Also: HW support for TM means easier ways to provide debugging mechanisms

Transactional Registers

Register Name	Role
Transaction Failure Handler Address Register (`TFHAR`)	Records address following `tbegin`. Will redirect flow to that address if transaction fails.
Transaction Failure Instruction Address Register (`TFIAR`)	Records the address of the instruction responsible for the transaction failure
Transaction Exception And Status Register (`TEXASR`)	Records misc. info w.r.t. status of current or most recently executed transaction

Transactional ISA

Instruction Name	Role
`tbegin R`	*Transaction begin*. *R* indicates it is a rollback-only transaction
`tend A`	*Transaction end*. Commits a transaction. When set, *A* forces commit regardless of nesting level
`tabort RA`	*Transaction abort*. Unconditionally aborts a transaction. Lower byte of *RA* is copied into `TEXASR`.
`tabortwc TO, RA, RB` `tabortdc TO, RA, RB`	*Transaction abort conditional*. *RA* and *RB* are compared using operator *TO*. Depending on the result, the transaction is aborted.
`tabortwci TO, RA, SI` `tabortwdi TO, RA, SI`	*Transaction abort conditional immediate*. Same as previous, but compare *RA* and *RB* with signed immediate operand
`tsr R`	*Transaction suspend or resume*. *L* controls whether to suspend or resume.
`tcheck BF`	*Transaction check*. Sets condition register if transaction has failed.

Transaction Failure I

Transaction Failures

- Thread is being suspended (*e.g.*, context-switched)
- Conflict: other transactions are being executed with an overlapping data set
- Conflict: Transactional vs. non-transactional access

Transaction Failure II

Failure Recording

- Records info w.r.t. cause and circumstances of failure in speculatir registers (SPRs)
- Sets `TFIAR`
- Sets failure cause bits, current privilege, suspend mode, *etc.*, in `TEXASR`
  - Also: may set `TFIAR` valid bit (to know if the register can be trusted)
- Goal: help the programmer determine which transaction caused the failure

Transaction Failure III

Failure Handling

- All updates to memory and SPRs are rolled back
- Control is transferred to failure handler address
- Condition register `CR0` is set to indicate a failure occurred
- If thread is in Transactional Mode, failure is recorded and handled immediately
- IF thread is in Suspended Transactional Mode, recording happens immediately, but handling happens when resuming Transactional Mode.

Outline

Critical Sections and Locking

- Critical sections are usually implemented with locks
- Coarse-grain locking is easy to implement but usually scale poorly
- Fine-grain locking provides much better scalability, but can become very complex to manage (very bug-prone)

Transactional Memory

- Transactional memory is a novel mechanism that attempts to provide a programmable set of operations
- Transactional operations are meant to help programmers build complex concurrent regions without the need for (fine-grain) locking
- Software TMs implement the concept purely in software
  - Help with productivity and reliability
  - But usually not performant enough
  - *May* become more interesting as the core/thread count increases on chips
- Hardware TMs are finally arriving (IBM POWER 8, Intel Haswell)
  - Provide HW instructions to handle transactional operations
  - Restrict themselves to a single transaction per core
  - Not fully HW for POWER 8: need the user to provide failure handler

Hybrid Transactional Memory

- Probably the future of TMs
- Combines (restricted-but-fast) HW-enabled TMs with (flexible-but-slow) STMs
- Leverage HTMs as much as possible
- If number of HTMs is exhausted, switch to STM.

Bibliography I

Alemany, Juan and Edward W. Felten (1992). "Performance Issues in Non-blocking Synchronization on Shared-memory Multiprocessors". In: *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*. PODC '92. Vancouver, British Columbia, Canada: ACM, pp. 125–134. ISBN: 0-89791-495-3. DOI: 10.1145/135419.135446. URL: http://doi.acm.org/10.1145/135419.135446.

Barnes, Greg (1993). "A Method for Implementing Lock-free Shared-data Structures". In: *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '93. Velen, Germany: ACM, pp. 261–270. ISBN: 0-89791-599-2. DOI: 10.1145/165231.165265. URL: http://doi.acm.org/10.1145/165231.165265.

Bibliography II

Cain, Harold W. et al. (2013). "Robust Architectural Support for Transactional Memory in the Power Architecture". In: *SIGARCH Comput. Archit. News* 41.3, pp. 225–236. ISSN: 0163-5964. DOI: 10.1145/2508148.2485942. URL: http://doi.acm.org/10.1145/2508148.2485942.

Cascaval, Calin et al. (2008). "Software Transactional Memory: Why Is It Only a Research Toy?" In: *Queue* 6.5, 40:46–40:58. ISSN: 1542-7730. DOI: 10.1145/1454456.1454466. URL: http://doi.acm.org/10.1145/1454456.1454466.

Dragojević, Aleksandar et al. (2011). "Why STM Can Be More Than a Research Toy". In: *Commun. ACM* 54.4, pp. 70–77. ISSN: 0001-0782. DOI: 10.1145/1924421.1924440. URL: http://doi.acm.org/10.1145/1924421.1924440.

Bibliography III

Herlihy, Maurice P. and Jeannette M. Wing (1990). "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3, pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: http://doi.acm.org/10.1145/78969.78972.

Herlihy, Maurice and J. Eliot B. Moss (1993). "Transactional Memory: Architectural Support for Lock-free Data Structures". In: *SIGARCH Comput. Archit. News* 21.2, pp. 289–300. ISSN: 0163-5964. DOI: 10.1145/173682.165164. URL: http://doi.acm.org/10.1145/173682.165164.

Bibliography IV

LaMarca, Anthony (1994). "A Performance Evaluation of Lock-free Synchronization Protocols". In: *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '94. Los Angeles, California, USA: ACM, pp. 130–140. ISBN: 0-89791-654-9. DOI: 10.1145/197917.197975. URL: http://doi.acm.org/10.1145/197917.197975.

Le, H.Q. et al. (2015). "Transactional memory support in the IBM POWER8 processor". In: *IBM Journal of Research and Development* 59.1, 8:1–8:14. ISSN: 0018-8646. DOI: 10.1147/JRD.2014.2380199.

Mellor-Crummey, John M. and Michael L. Scott (1991). "Synchronization Without Contention". In: *SIGPLAN Not.* 26.4, pp. 269–278. ISSN: 0362-1340. DOI: 10.1145/106973.106999. URL: http://doi.acm.org/10.1145/106973.106999.

Bibliography V

Shavit, Nir and Dan Touitou (1997). "Software transactional memory". English. In: *Distributed Computing* 10.2, pp. 99–116. ISSN: 0178-2770. DOI: 10.1007/s004460050028. URL: http://dx.doi.org/10.1007/s004460050028.

Turek, John, Dennis Shasha, and Sundeep Prakash (1992). "Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking". In: *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '92. San Diego, California, USA: ACM, pp. 212–222. ISBN: 0-89791-519-4. DOI: 10.1145/137097.137873. URL: http://doi.acm.org/10.1145/137097.137873.