Concurrent Systems

*Nebenläufige Systeme*

XII. Transactional Memory

Wolfgang Schröder-Preikschat

January 8, 2015
Agenda

Preface

Principles
  General
  Characteristic
  Operation

Utilisation
  Abstraction
  Examplification
  Discussion

Summary
Preface

Principles
   General
   Characteristic
   Operation

Utilisation
   Abstraction
   Examplification
   Discussion

Summary
discussion on abstract concepts as to facilitation of programming of parallel processes by means of **transactional regions**

- explicitly versus implicitly transactional approaches
- hardware (HTM), software (STM), or hybrid (HyTM) solutions

minimal subset of system functions i.e. machine **instructions**

- load, store, and commit for the explicit case
- begin and end for the implicit case
- abort for both cases—the exception proves the rule...

last but not least, a **critical examination** of the paradigm/concepts

- strength of transparency: extent of transactional data set, retry loop
- failure of transactions: frequency, reason, alternative measures
- type of synchronisation: unilateral, multilateral

**Universal Remedy?**

The bigger the critical shared state is, the better TM seems to be. But what about support, overhead, control, and coordination?
Decrease of Suffering, Increase of Happiness...

abstraction paves the way for a “decrease of suffering” in the design of programs for non-blocking synchronisation of parallel processes

similar to blocking synchronisation such as mutual exclusion, the secured program sections describes a seemingly sequential process

but unlike that synchronisation pattern, the respective program sections may be run by non-sequential processes

within those sections, on a simple load/store basis, instructions are given what data need to be kept consistent

it is up to the TM (hardware/software) system functioning underneath to maintain consistency of that data set

as the case may be, “increase of happiness” can be reached due to a potential relaxation in identifying a solution

stressless labour, contentment, room for creativity, higher productivity

physiological (physical) and psychological (emotional, cognitive) aspects

in spite of everything, not run the risk of overstating sequential and understating parallel thinking...
Outline

Preface

Principles
  General
  Characteristic
  Operation

Utilisation
  Abstraction
  Examplification
  Discussion

Summary
to come to the point, TM has a silver lining but also a demerit:

**Pros**
- allows for the definition of customized atomic operations that apply to a group of possibly arbitrary computer words
- can be seen as a caching method for implementing data structures in a lock-free manner [2]
- technically feasible as “straightforward extensions to multiprocessor cache-coherence protocols” [10]
- offers a more convenient handling compared to, e.g., a multi-word CAS using (software-implemented CAS-based) LL/SC [14]

**Cons**
- may be a replacement for multilateral synchronisation (i.e., mutual exclusion using e.g. locks or binary semaphores), only
- neither facilitates nor supports, but rather hampers, unilateral (i.e., logical/conditional) synchronisation
- prone to overhead in case of mindless reuse of external functions or procedures from libraries, for instance
- tempt developers of non-sequential programs to see things through rose-coloured glasses

**TM is a means to an end**—and is far from being a cure-all...
Levels of Abstraction

depending on the rootedness of the implementation, divided into:

HTM
- **hardware** transactional memory [10], typically classified into [8]:
  - *explicitly transactional* (ASF proposal [1], RTM [12])
    - memory instructions indicate each single transactional load/store
    - may also provide instructions to start and commit transactions
  - *implicitly transactional* (SLE [16], Rock [18], PPC [7], HLE [12])
    - begin/end instructions, only, specify the boundaries of a transaction
    - if applicable, options to identify *non-transactional* memory locations
- buffering capabilities limited by (L1, L2) cache size

STM
- **software** transactional memory [17], destined for any hardware
- distinguishes between *static* [17] and *dynamic* [9] approaches
- buffering capabilities limited by (virtual) memory size
- scalability problems [15], significant *metadata* overhead [4]

HyTM
- **hybrid** transactional memory [5], STM as a fall-back solution
- heterogeneous transactions: (1) HTM-based and (2) STM-based
  - (1) gets aborted if in conflict with (2), may be restarted as (2)

howsoever, **linguistic support** is desirable—but, with or without it, TM is no panacea to solve all non-sequential programming issues
read-set tracking and write-set buffering takes direct advantage of existing hardware capabilities to capture memory accesses

original idea [10] was cache duplication to add a transactional cache
- augments hardware design with significant complexity
- introduces an additional structure from which data may be sourced

another approach is by means of cache extensions
- additional “sticky” read bit per cache line used as read-set indicator
- for the write-set, addresses involved are given a “speculative written” state

granularity of conflict detection is the cache line $\sim$ false sharing
- data-sets of different transactions should be mapped to different cache lines
- requires static program analysis to render that problem manageable, if at all

but, not yet really common in available processors architectures:

IBM Blue Gene/Q (PowerPC A2 [7])
- limited to multi-versioned L2 cache (20 MiB out of 32 MiB, [19, p. 129])
- “watch granule” is 64 B [11, p. 509], same as cache line size

Intel Haswell (TSX [12])¹
- transaction size limited to L1 cache (64 KiB), 64 B cache line

¹Mindless of the TSX bug [13, p. 47], which leaves TSX barred for normal use.
transactional memory is a shared region of problem-specific size
- for each memory cell therein, an ownership relationship is maintained
  - which identifies the owning transaction comprising the particular cell
    - described by a per-process transaction record also held in shared memory

Open array of memory locations read/written (address []).
STM: Course of a Transaction

assuming that the real/virtual machine provides LL/SC (cf. p. 31):

1. acquire **ownership** of each location of a data-set member
   - reserve (LL) the respective location and, if still unowned:
     (a) try to establish reservation (SC), if transaction is valid anymore
     (b) retry reservation (LL), otherwise
   - otherwise, return failure including reference of failed location

2. if successful:
   2.1 readout memory corresponding to the data-set locations:
      (a) in case of a free backup location (LL), (b) try to save former value (SC)
   2.2 compute new values based on the values read out to backup locations
   2.3 update memory corresponding to the data-set locations:
      (a) reserve memory location (LL) and (b) try to assign new value (SC)
   2.4 release ownership of each location of a data-set member:
      (a) in the case of a still acquired location (LL), (b) try to reset (SC)

3. if unsuccessful:
   3.1 release existing ownerships (cf. 2.4), if any
   3.2 as the case may be, help the transaction which owns failing locations

**a generation number** (preferable 64-bit) makes a transaction unique

- additional parameter of steps 1, 2.1, 2.3, and 2.4: needs to be checked
fundamental idea is to “attempt to kill two birds with one stone”:

i. provide STM independent from specific hardware support beyond what is currently available and, at the same time,

ii. support execution of transactions by using whatever HTM feature so that both concepts of TM will coexist correctly

assumption is that in most cases HTM transactions will succeed

- if the HTM path fails, a run-time system decides how to retry:
  
  i. on the HTM path, repeatedly, if contention is weak or can be contained
  
  ii. on the STM path, otherwise, possibly with more flexible contention control

- engage STM in case of hardware limitations or high/complex contention

therefore, a dedicated compiler ejects two different code paths

- HTM actions are augmented with code that allows coexistence with STM
  
  - logical and physical values of a particular TM location are monitored
  
  - they may differ if a STM transaction is in progress and overlaps HTM actions

- a HTM transaction will abort if STM actions caused data-set changes

an implicitly transactional model is assumed, but not the only way

- by concept, an explicitly transactional approach is feasible as well
operations for accessing memory (implicitly or explicitly):

- **load**
  - transfers a value from shared memory to a private placeholder
  - add the source location to the transaction **read set**
- **store**
  - provides a value for transfer to shared memory, but the value to be transferred becomes visible not before a successful commit
  - add the destination location to the transaction **write set**

- the union of the read and write sets is the **data set** of the transaction

operations for manipulating transaction state (initiated explicitly):

- **commit**
  - attempts to make the changes as to the write set visible
  - succeeds for the current process only if no other transaction:
    - updated any location in the current data set and
    - read any location in the current write set
  - otherwise, aborts the current transaction and fails
- **abort**
  - discards all changes to the write set of the current transaction

Further operations are customary, according to circumstances
- depending on the level of abstraction the TM system is associated with
it is assumed that contention of simultaneous processes is improbable

- a **successful commit** seems to be probable, other than abort and retry

```c
extern word_t foo, bar, foobar;
do {
    word_t foo' = load(&foo);
    word_t bar' = load(&bar);
    store(&foobar, foo' + bar');
} while (!commit());
```

- other **exceptional events**, besides conflicting simultaneous processes
  - originated in the operating-system machine level and below:
    - traps (e.g., page faults) and interrupts (e.g., quantum expiration)
    - context switches (e.g., system calls, process dispatching)
  - originated in the non-sequential program itself:
    - avoidance or resolution of serialisation conflicts

- thus, be aware of reasons and frequency of the failure of transactions
  - if applicable, take care of region-specific **counteractive measures**
  - reflect on alternative concepts/solutions in achieving data consistency
a more advanced abstraction is to merely declare an **atomic region**
- at the expense of a loss of control of the extent of the actual data set

```plaintext
extern word_t foo, bar, foobar;
begin(&& dropout);
    word_t foo' = foo;
    word_t bar' = bar;
    foobar = foo' + bar';
end();
```

- unless the compiler knows about **critical variables** that make up the data set, **all variables** read or written need to be tracked by the processor
- this results in unnecessarily larger data sets and increases overhead

**retry-loop concealment** is not always an advantageous measure
- aside from other exceptional events (p. 15), retries are due to contention
- contention control depends not only on dynamic but also static data
  - i.e., number of contending processes and duration of a single retry
  - whereby the latter is determined by the regions’s execution path length
- **begin/end are unaware of expectable execution times** of atomic regions
in case of STM, it is worth to consider the following refinements:

- upper-bound size of the read- and write-set in `btx`
- specification of the reason of abort in `atx`
- declaration of further modes of operation (flags) in `btx`
- additional (first) parameter indicating this transaction in each operation

however, as (most of) these depend on the program structure of the transactional region, determination should be up to the compiler
inline void push_dos(stack_t *this, chain_t *item) {
    item->link = this->head.link;
    this->head.link = item;
}

void push_tm_it(stack_t *this, chain_t *item) {
    btx(0);
    item->link = this->head.link;
    this->head.link = item;
    etx();
}

void push_tm_et(stack_t *this, chain_t *item) {
    do {
        item->link = (chain_t *)ltx(&this->head.link);
        stx(&this->head.link, (long)item);
    } while (!ctx());
}
FIFO-List Revisited I

```c
inline void chart_dos(queue_t *this, chain_t *item) {
    item->link = 0;        /* finalise chain */
    this->tail->link = item; /* append item */
    this->tail = item;     /* set insertion point */
}

void chart_tm_it(queue_t *this, chain_t *item) {
    item->link = 0;
    btx(0);
    this->tail->link = item;
    this->tail = item;
    etx();
}

void chart_tm_et(queue_t *this, chain_t *item) {
    item->link = 0;
    do {
        stx(&this->tail->link, (long)item);
        stx(&this->tail, (long)item);
    } while (!ctx());
}
```

Both TM variants appear to be equivalent.
LIFO-List Revisited II

Take a sledgehammer to crack the nut...

1 chain_t *wear_dos(stack_t *this) {
   2 chain_t *node = this->head.link;
   3 this->head.link = 0;
   4 return node;
   5 }

6 chain_t *wear_tm(stack_t *this) {
   7 chain_t *node;
   8 do {
       9         node = ltx(&this->head.link);
       10        stx(&this->head.link, 0);
       11    } while (!ctx());
   12    return node;
   13 }

14 chain_t *wear_wfs(stack_t *this) {
   15 return FAS(&this->head.link, 0);
   16 }

© wosch  
CS (WS 2015, LEC 12)  
Utilisation – Examplification  
21 – 34

Overshoot

Definitely, TM is no magic bullet…
The TM programming model itself, whether implemented in hardware or software, introduces complexities that limit the expected productivity gains, thus reducing the current incentive for migration to transactional programming and the justification at present for anything more than a small amount of hardware support. [4, p. 55]

- **logical/conditional synchronisation**, e.g. condition variables [6]:
  - waiting on a condition inside a transaction is difficult or impossible
    - difficult, e.g., in case of an I/O operation that cannot be rolled back
    - impossible, if the transactional process is implemented as kernel-level thread
  - as a signalling transaction may abort, the stated condition never occurred
    - furthermore, signaller and signallee transactions may happen simultaneously, which is prone to lost-wakeup as the latter may complete before the former
- Thus, TM is merely an abstraction to **multilateral synchronisation**
  - most attractive semantics is its “single global lock atomicity” [3]

---

2Assuming that TM applies to user-level processes, only—which is usual.
Outline

Preface

Principles
   General
   Characteristic
   Operation

Utilisation
   Abstraction
   Examplification
   Discussion

Summary
Résumé

- TM abstractions as to the different rootedness of the implementation
  - HTM  ■ hardware, explicitly/implicitly transactional, hardly available
  - STM  ■ software, lock-less/based solutions, metadata overhead
  - HyTM ■ hybrid, try HTM first, fall back on STM in critical situations

- principle concepts of TM and functions or instructions, respectively
  - read set, write set, and the union thereof: data set
  - load, store, commit, abort, begin, end—and more. . .

- examination and discussion of the pros and cons of TM
  - especially limited hardware support still hampers wide use
  - independent thereof, programming introduces other types of complexities
  - also because it merely is an abstraction to multilateral synchronisation

- TM is a means to an end, it has a silver lining but also a demerit. . .

Transactionable Memory Should Be an Implementation Technique, Not a Programming Interface. [3]
[1] ADVANCED MICRO DEVICES, INC. (Hrsg.):

[2] BARNES, G.:
A Method for Implementing Lock-Free Shared-Data Structures.

[3] BOEHM, H.-J.:
Transactional Memory Should be an Implementation Technique, Not a Programming Interface.
In: Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar ’09).
Berkeley, Ca, USA: Usenix Association, 2009, S. 1–6
Software Transactional Memory: Why is it Only a Research Toy?
In: Queue 6 (2008), Sept., Nr. 5, S. 46–58

Hybrid Transactional Memory.
In: Proceedings of the 12th International Conference on Architectural Support for
Programming Languages and Operating Systems (ASPLOS XII).
New York, NY, USA : ACM, 2006. –

Condition Variables and Transactional Memory: Problem or Opportunity?
SIGPLAN Workshop on Transactional Computing (TRANSACT 2009),


[14] Israeli, A.; Rappoport, L.:
Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives.

The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-Core Environment.

[16] Rajwar, R.; Goodman, J. R.:
Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution.
[17] **Shavit, N. ; Touitou, D. :**
Software Transactional Memory.
In: *Distributed Computing* 10 (1997), S. 99–116

[18] **Tremblay, M. ; Chaudhry, S. :**
A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC® Processor.
In: *Proceedings of the 2008 IEEE International Solid-State Circuits Conference (ISSCC).*

Evaluation of Blue Gene/Q Hardware Support for Transactional Memories.
In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12).*
typedef struct ref {
    int *label; /* actual location in (shared) memory */
    int owner; /* reservation number: initial anything but -1 */
} ref_t;

inline int ll(ref_t *ref, int key) {
    int owner, value;
    do {
        owner = ref->owner;
        value = *(ref->label);
    } while ((ref->owner == -1) || !CAS(&ref->owner, owner, key));
    return value;
}

inline bool sc(ref_t *ref, int key, int val) {
    bool done;
    if ((done = CAS(&ref->owner, key, -1))) {
        *(ref->label) = val;
        ref->owner = 0;
    }
    return done;
}

**key**
- unique transaction number
- advanced when transaction completes
inline chain_t *pull_dos(stack_t *this) {
    chain_t *node;
    if ((node = this->head.link))
        this->head.link = node->link;
    return node;
}

chain_t *pull_tm(stack_t *this) {
    chain_t *node;
    do {
        if ((node = (chain_t *)ltx(&this->head.link)))
            stx(&this->head.link, (long)node->link);
    } while (!ctx());
    return node;
}

the implicitly transactional variant would unnecessarily include node in the transaction data set...
inline chain_t * fetch_dos (queue_t * this) {
    chain_t * node;
    if ((node = this->head.link) /* filled? */
        && !(this->head.link = node->link)) /* last item? */
        this->tail = &this->head; /* reset */
    return node;
}

chain_t * fetch_tm(queue_t * this) {
    chain_t * node;
    do {
        if ((node = (chain_t *)ltx(&this->head.link))) {
            stx(&this->head.link, (long)node->link);
            if (!node->link)
                stx(&this->tail, (long)&this->head);
        }
    } while (!ctx());
    return node;
}

the implicitly transactional variant would unnecessarily include node in the transaction data set...
inline chain_t *drain_dos(queue_t *this) {
    chain_t *head = this->head.link;
    this->head.link = 0; /* null item */
    this->tail = &this->head; /* linkage item */
    return head;
}

chain_t *drain_tm(queue_t *this) {
    chain_t *head;
    do {
        head = (chain_t *)ltx(&this->head.link);
        stx(&this->head.link, 0);
        stx(&this->tail, (long)&this->head);
    } while (!ctx());
    return head;
}

the implicitly transactional variant would unnecessarily include \textit{head} in the transaction data set...