Subject Matter

- discussion on abstract concepts as to facilitation of programming of parallel processes by means of \textit{transactional regions}
  - explicitly versus implicitly transactional approaches
  - hardware (HTM), software (STM), or hybrid (HyTM) solutions
- minimal subset of system functions i.e. machine \textit{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 \textbf{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?

\textit{The bigger the critical shared state is, the better TM seems to be. But what about support, overhead, control, and coordination?}
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...
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

STM: Organisation of the Data Structures acc. [17, p. 103]

- 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

HTM acc. [8, p. 149/150]

- 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 ~ 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
  - 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

STM: Course of a Transaction acc. [17, p. 104]

- 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:
    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. compute new values based on the values read out to backup locations
    3. update memory corresponding to the data-set locations:
      - (a) reserve memory location (LL) and (b) try to assign new value (SC)
  3. if unsuccessful:
    1. 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)
    2. as the case may be, help the transaction which owns failing locations
  3. if unapplicable, options to identify non-transactional memory locations
  4. additional parameter of steps 1, 2.1, 2.3, and 2.4: needs to be checked

1Mindless of the TSX bug [13, p. 47], which leaves TSX barred for normal use.
fundamental idea is to “attempt to kill two birds with one stone”: 
- provide STM independent from specific hardware support beyond what is currently available and, at the same time, 
- 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: 
  - on the HTM path, repeatedly, if contention is weak or can be contained
  - 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 injects 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 overlays 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

Positive Thinking I

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

Minimal Subset of System Functions

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
- 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
Operational Interface

```c
extern void btx(void *);  /* begin */
extern void atx();  /* abort */
extern bool ctx();  /* commit */
extern void etx();  /* end */
extern long ltx(void *);  /* load */
extern void stx(void *, long);  /* store */
```

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.

---

LIFO-List Revisited I

```c
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());
}
```

Both TM variants appear to be equivalent.

---

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

chain_t *wear_dos(stack_t *this) {
    chain_t *node = this->head.link;
    this->head.link = 0;
    return node;
}

chain_t *wear_tm(stack_t *this) {
    chain_t *node;
    do {
        node = ltx(&this->head.link);
        stx(&this->head.link, 0);
    } while (!ctx());
    return node;
}

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

Overshoot

Definitely, TM is no magic bullet...

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, signaler 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]

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

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...

Transactional Memory Should Be an Implementation Technique, Not a Programming Interface. [3]
Reference List I


Reference List II


Reference List III


Reference List IV


Emulation of LL/SC

```c
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));
  } while (true);
  return value;
}

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

LIFO-List Revisited III

```c
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 (!ctx())
      stx(&this->head.link, (long)node->link);
  } while (!ctx());
  return node;
}
```

- **key** — unique transaction number
- advanced when transaction completes

© wosch  CS (WS 2016, LEC 12)  Addendum – Characteristic
© wosch  CS (WS 2016, LEC 12)  Addendum – Examplification
FIFO-List Revisited II

```c
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...

FIFO-List Revisited III

```c
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 `head` in the transaction data set...