IBM STM Interface and X10 Extensions

Maged Michael and Vijay Saraswat
Outline

- STM Runtime Interface
- X10 Extensions
- Obstacles to a Single TM Standard
STM Runtime Interface
Acknowledgements

- Colin Blundell
- Trey Cain
- Calin Cascaval
- Sid Chatterjee
- Stefanie Chiras
- Takuya Nakaike
- Ken Mizuno
- Raul Silvera
- Michael Spear
- Christoph von Praun
- Michael Wang
- Peng Wu
IBM Common STM Runtime

- ** Implemented as a C library
- ** A version released to open source (June 2008)
- ** STM runtime supports:
  - A Java JIT compiler STM implementation
  - C/C++ STM compilers
  - Binary of IBM STM XL compiler released in May 2008
- ** Runs on:
  - Commodity platforms: AIX/Linux PPC/X86
  - Hardware acceleration models
STM Interface – Per-Thread STM Descriptors

```c
void * stm_thr_init();
```
- Creates a new per-thread STM descriptor
- Returns a pointer to per-thread STM descriptor

```c
void * stm_desc();
```
- Returns a pointer to per-thread STM descriptor of the current thread

```c
void stm_thr_retire(void * mydesc);
```
- Retire the current thread’s descriptor
# STM Interface – Transaction Begin and End

```c
int stm_begin(void * buf, void * mydesc, char * fname, int line);
```

- **Arguments:**
  - `buf`: pointer to a buffer for use by longjmp on abort
  - `mydesc`: pointer to per-thread transactional descriptor
  - `fname`: string representing the filename where the code of the transaction occurs, e.g., `__FILE__`
    - Used for per-static-transaction statistics
  - `line`: integer representing the line number where the code of the transaction starts, e.g., `__LINE__`

- **Returns an integer representing encountered state:**
  - INACTIVE (started outermost transaction)
  - ACTIVE (nested)
  - ABORTED (nested)
  - NON_SPECULATIVE (nested)

```c
int stm_end(void * mydesc);
```

- **Returns a Boolean value representing outcome:**
  - SUCCESS
  - FAILURE
STM Interface – Transaction Status and Validation

```c
int in_transaction(void * mydesc);
```

- Returns a Boolean value indicating whether the current thread is running inside a transaction or not

```c
int stm_validate(void * mydesc);
```

- Returns a Boolean result indicating whether the current transaction’s read set is valid or not
STM Interface – Non-Speculative Mode

```c
int become_inevitable(void * mydesc);
```

- Try to get into inevitable (non-speculative) mode
- If successful, then this transaction is guaranteed not to be aborted
- The transaction may execute non-speculative actions with irrevocable effects
- Returns Boolean value indicating whether the transaction was able or not to enter the non-speculative mode
STM Interface – Abort if Speculative

```c
int stm_abort(void *mydesc);
```

- **Aborts the current transaction if running speculatively**
- **Returns integer value representing status before abort**
  - INACTIVE
  - ACTIVE
  - ABORTED
  - NON_SPECULATIVE

**Note:**
- The transaction is not aborted if it is running in non-speculative mode
STM Interface – Transactional Reads

```c
void * stm_read_ptr(void * volatile * addr, void * mydesc);
float stm_read_float(float volatile * addr, void * mydesc);
... <other basic data types>
unsigned long stm_read_ulong(unsigned long volatile * addr, void * mydesc);
unsigned long long stm_read_ull(unsigned long long volatile * addr, void * mydesc);
```

- **Arguments:**
  - `addr`: pointer to the variable being read

- **Returns the value of the variable being read from the point of view of the current transaction**
STM Interface – Transactional Writes

```c
void stm_write_ptr(void * volatile * addr, void * val, void * mydesc);
void stm_write_float(float volatile * addr, float val, void * mydesc);
... <other basic data types>
void stm_write_ulong(unsigned long volatile * addr, unsigned long val, void * mydesc);
void stm_write_ull(unsigned long long volatile * addr, unsigned long long val, void * mydesc);
```

- **Arguments:**
  - `addr`: pointer to the variable to be written
  - `val`: value to be written
STM Interface – Memory Allocation

- Only memory allocations inside transactions need to call special STM functions

```c
void * stm_malloc(size_t sz, void * mydesc);
void * stm_calloc(size_t ne, size_t sz, void * mydesc);
void * stm_free(void * ptr, void * mydesc);
```

- Arguments and return values:
  - Same as standard malloc/calloc/free
Local variables initialized outside the transactions need to be checkpointed for rollback on abort, before being written inside a transaction

Arguments:
- `addr`: pointer to local variable
- `size`: size of local variable

pre-transaction value needs to be restored on abort
STM Interface – Handling Address-Taken Stack Variables

- If addresses of local variables are passed as arguments to function calls, the STM may end up treating these variables as shared.
- STM needs to handle accesses to these variables consistently as local.

```c
void stm_stack_range(void * addr, int size, void *mydesc);
```

- Arguments:
  - `addr`: beginning of range
  - `size`: size of range

```c
stm_begin(...); ...
int var = a; ...
foo(&var); ...
stm_end();
```

Treated as local variable

```c
void foo(int * ptr) {
  *ptr = b;
}
```

Must be treated as local for consistency
STM Interface – Collecting Statistics

```c
void stm_stats_out();
```

- Saves a snapshot of STM stats
- Inherent transactional stats
- Implementation-specific stats
- Stats per static transaction

**Statistics:**

- **65536** READ_WRITE_COMMITS
- **69528** READ_SET_VALIDATIONS
- **0** READ_ENCOUNTER_RETRIES
- **0** SIGNAL_RETRIES
- **0** WRITE_ACQUIRE_RETRIES
- **0** READ_VALIDATION_RETRIES
- **525712** WRITE_BARRIERS
- **13419** NUM_SILENT_WRITES
- **0** SILENT_WRITES_BECAME_READS
- **0** WRITE_BARRIERS_OUTSIDE_TXNS
- **131010** WRITE_BARRIERS_FOR_STACK
- **9730** DUPLICATE_WRITES
- **0** DUPLICATE_WRITE_CONFLICT_SET
- **10726816** READ_BARRIERS
- **0** READ_BARRIERS_OUTSIDE_TXNS
- **0** READ_BARRIERS_FOR_STACK
- **4225604** DUPLICATE_READS
- **6874933** DUPLICATE_READ_CONFLICT_SET
- **0** USEFUL_DUP_READ_CHECKS
- **0** USELESS_DUP_READ_CHECKS
- **934007** BLOOM_FILTER_CHECKS
- **102826** BLOOM_FILTER_MATCHES
- **88121** READ_AFTER_WRITE_MATCHES
- **10638695** READ_LIST_SIZES
- **384972** WRITE_LIST_SIZES
- **6501212** READ_SET_SIZES
- **384972** WRITE_SET_SIZES
- **653** READ_LIST_MAX_SIZE
- **230** WRITE_LIST_MAX_SIZE
- **272** READ_SET_MAX_SIZE
- **230** WRITE_SET_MAX_SIZE
- **1** MAX_NESTING
- **99.20** AVG_READ_LIST_SIZE
- **5.87** AVG_WRITE_LIST_SIZE
- **162.33** AVG_READ_SET_SIZE
- **5.87** AVG_WRITE_SET_SIZE
- **0** TOTAL_RETRIES
- **0.00** AVG_RETRIES_PER_TXN
- **0.00** AVG_CHECKPOINTING_CALLS_PER_TXN
- **39.39** PCT_DUPLICATE_READS
- **64.09** PCT_DUPLICATE_WRITE_CONFLICT_SET
- **1.85** PCT_DUPLICATE_WRITES
- **2.55** PCT_SILENT_WRITES
STM Interface – Sub-Operations

- Interface for uncommon sub-operations, in order to enable inlining of common sub-operations

```c
void stm_read_bloom_match(void * addr, int size, void * mydesc);
void expand_reads(void * mydesc);
void stm_cleanup_aborted(void * mydesc);
... 
```

- Interface to fences and validation checks, in order to enable aggregation of fences and validation checks

```c
void stm_read_orec_check(void * addr, void * mydesc);
void stm_read_orec_mem_fence();
void stm_read_from_mem(void * addr, int size, void * mydesc);
... 
```
X10 Extensions
X10 Atomic Constructs

- **atomic**
  - Unconditional atomic block

  ```x10
  atomic {
    S;
  }
  ```

- **when**
  - Conditional atomic block
  - Atomically guarantees that the condition \( c \) holds and executes the atomic section \( S \).

  ```x10
  when( c ) {
    S;
  }
  ```
X10 Common Patterns

- Atomic blocks with static data sets
  
  ```java
  atomic {
  x = y + z;
  }
  ```

- Shared data accessed in atomic sections that is guaranteed to have no conflicts
  
  ```java
  atomic {
  x = y + z;
  }
  ```

  Guaranteed no conflicts  Need conflict detection
Extensions to Exploit Patterns

- **Capability to specify:**
  - Shared variables that may be read, written, or read and written inside an atomic block
  - Whether the identified data set is complete or not
  - Shared variables that are guaranteed to have no conflicts

```c
}@un(rd(y), wr(w), rd_wr(x), nc(z)) atomic {
  w = x + y + z;
  x = x + w;
}
```

or

```c
}@un(rd(y), wr(w), rd_wr(x), complete) atomic {...}
```
STM Extensions

- Add address range to the read set and write set of the current transaction

  ```c
  void stm_add_to_read_set(void * addr, int size, void * mydesc);
  void stm_add_to_write_set(void * addr, int size, void * mydesc);
  ```

- Ignore subsequent transactional reads and writes to locations in the address range

  ```c
  void stm_no_conflict(void * addr, int size, void * mydesc);
  ```

- Indicate that the specified transactional data set is complete

  ```c
  void stm_data_set_complete(void * mydesc);
  ```
Obstacles to a Single TM Standard
Variety of TM Features and Requirements

- **Allowing non-speculative actions**
  - e.g., to execute I/O, system calls

- **Non-blocking progress**
  - e.g., in real-time apps

- **Allowing user abort, abort on exception**
  - e.g., for convenience of recovery

- **Strong atomicity**
  - e.g., for simulation of complex atomic operations

- **Privatization-safety, publication-safety**

- **Open nesting, transactional boosting**

- **Allowing condition variables**
Limitations on TM Features

- **Some TM Features are contradictory**
  - Some features cannot be allowed concurrently without programming restrictions
    - E.g., Non-blocking transactions that may conflict with transactions with non-speculative actions
  - Some features have per-transaction restrictions:
    - E.g., User abort after executing non-speculative actions

- **Unused features are often costly**
  - Performance overheads
    - Strong atomicity
  - Complexity of combination with other features
    - Non-speculative actions and strong atomicity

*No One TM Standard Fits All*
Variety of Performance Priorities

- **Performance characteristics**
  - High/low Parallelism
  - Low/high Overheads
  - Graceful/fall-off-a-cliff degradation

- **Performance depends on TM implementation options**
  - Conflict detection policies
  - Contention management
  - Consistency granule, e.g., object/block-based, block size

- **Sharp trade-offs among performance characteristics**
  - e.g., graceful-degradation vs. low best-case overheads

- **Adaptivity is often costly**

- **Performance is a primary motivation for many TM uses**

*A single omni-featured TM is likely to deliver inadequate performance*
A Multi-TM Standard?

- Allow multi-TM co-existence

*Can this be done without compromising code modularity? and without an explosion in feature combinations?*

*A single TM standard will have to make careful choices that hopefully capture the most useful features of TM*

*Thank You*
BACKUP
Constructs

- Multiple TM instances

```c
__tm_attribute((non_blocking)) __tm_atomic { r = x; .... };
__tm_attribute((nonspeculative)) __tm_critical { r = y; .... }
```