IBM alphaWorks Software Transactional Memory Compiler
OpenSource Amino STM runtime

Michael Wong, Maged Michael, Peng Wu
michaelw@ca.ibm.com,

IBM Rational Disclaimer

The IBM Rational C/C++ Café

ibm.com/rational/cafe/community/ccpp

Agenda

- **Explorations in Software Transactional Memory**
- **IBM's freely downloadable** XL C/C++ Enterprise Edition for AIX **STM compiler release**
- **Amino: OpenSource STM runtime**
- **STM challenges**

Transactional Memory

- Transactions appear to execute atomically
- A transactional memory implementation may allow transactions to run concurrently but the results must be equivalent to some sequential execution

Example

```
Initially, a == 1, b == 2, c == 3
```

T1

```
atomic {
   a = 2;
   b = a + 1;
   c = b + 1;
}
```

T2

```
atomic {
   r1 = a;
   r2 = b;
   r3 = c;
}
```

```
T2 then T1 r1==1, r2==2, r3==3
```

```
T1 then T2 r1==2, r2==3, r3==4
```

T1	T2
	`r1 = 1`
`a = 2;`	
`b = 3;`	
	`r2 = 3;`
	`r3 = 3`
`c = 4;`	

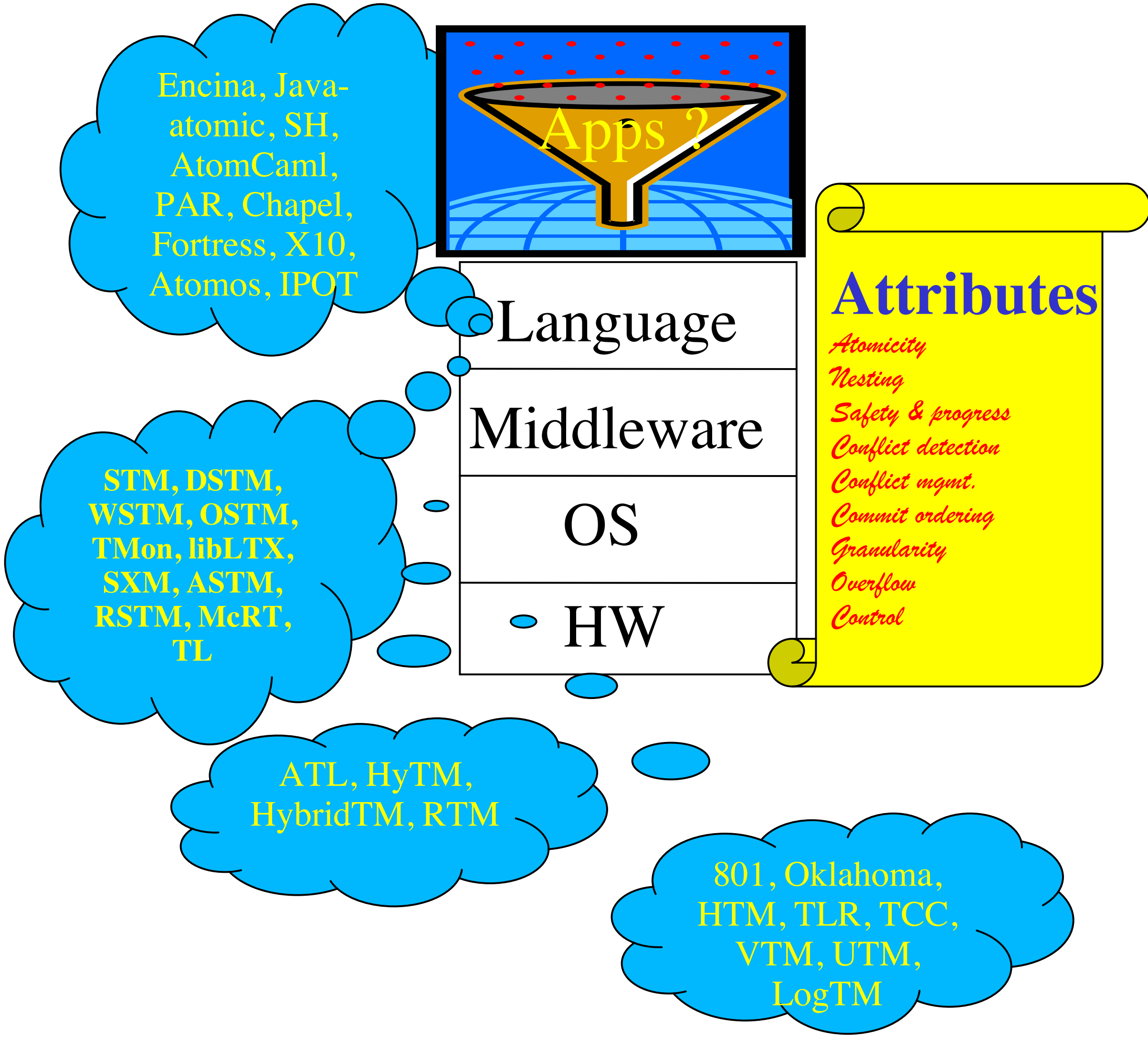
```
Incorrect r1==1, r2==3, r3==3
```

What is Transactional Memory (Again) ?

- **ACI(D) properties of a transaction make it easier to ensure that shared memory programs are correct.**
  - *Atomic*: each transaction either commits (it takes effect) or aborts (its effects are discarded).
  - *Consistent* (or *serializable*): they appear to take effect in a one-at-a-time order.
  - *Isolated* from other operations: the effects are not seen until the transaction has committed.
  - *(Durable*: their effects are persistent.)

Explorations in Transactional Memory

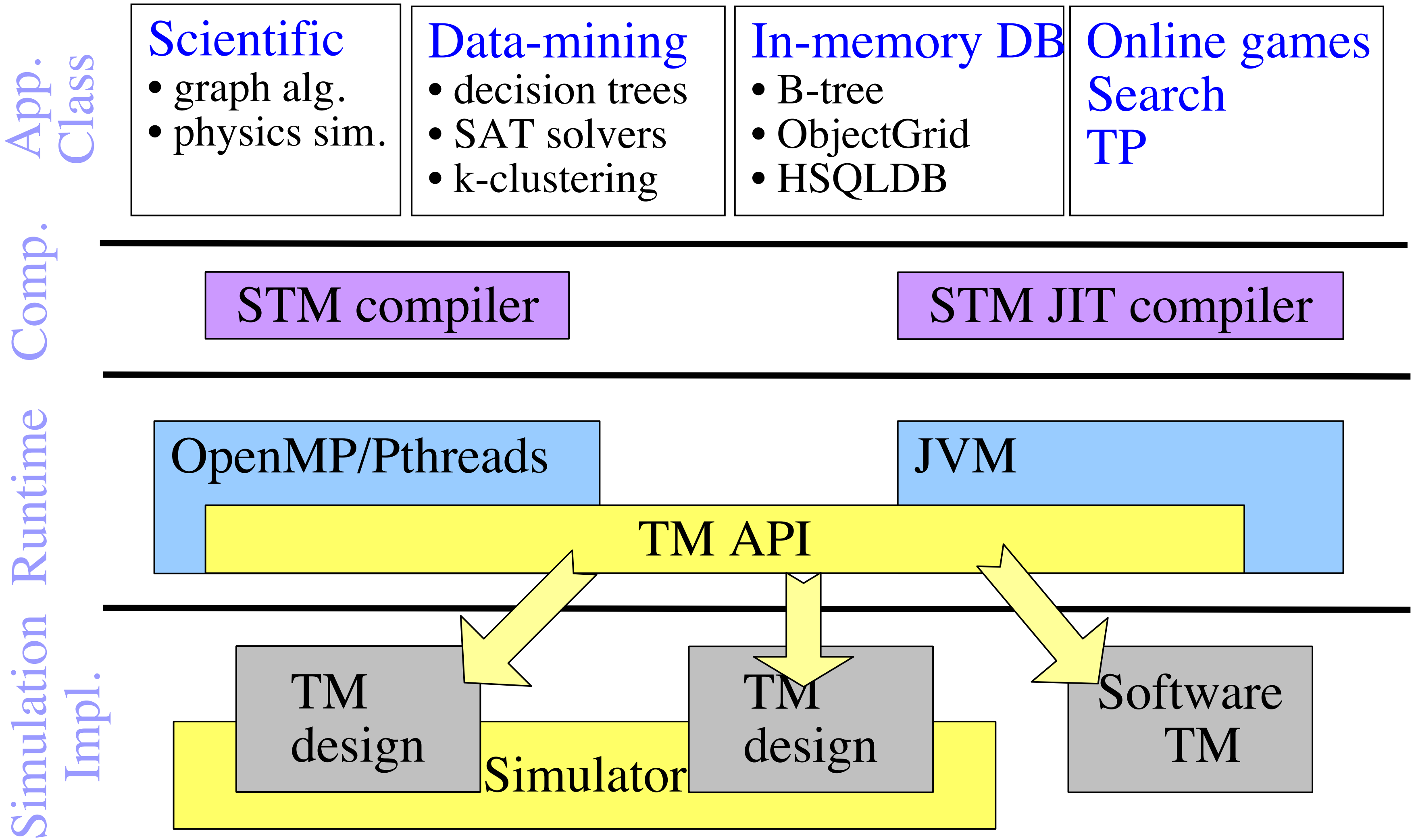
- Workload Characterization
  - –Are there workloads (and with what characteristics) that can benefit from TM?
- TM System Design
  - – trade-offs between HW and SW complexity, productivity and performance?
- Other Uses
  - –If TM support mechanisms were added in HW & SW, are there other interesting uses for them?
- Theory of TM: Semantics and programming models
  - –What is the right TM semantics and its interaction with memory consistency semantics?
  - –What are the right programming models to exploit transactions?



IBM alphaWorks STM Releases

- http://www-949.ibm.com/software/rational/cafe/blogs/ccpp-parallel-multicore/2009/08/11/ibmsalphaworks-software-transactional-memory-compiler
- Releases:
  - **alphaWorks :**
    - **IBM XL C/C++ Enterprise Edition for AIX, Version 9.0**
    - **http://www.alphaworks.ibm.com/tech/xlcstm**
    - **Standard and debugging version of the runtime**
  - **Open source STM runtime:**
    - **IBM Amino Concurrency Building Blocks projects**
    - http://sourceforge.net/projects/amino-cbbs/

Project Components



Software Transactional Memory (STM)

- **General approach**
  - All transactional state is managed by a software library
  - Read/write must be annotated for the system for version control and conflict detection
  - If conflicting with other transactions, the transaction may wait, abort itself and retry, or abort other transactions.
  - At the end of the transaction, reads are validated. If no conflicts are detected, writes are committed.
- **Primary advantages**
  - No changes to hardware
  - Enable scalability for workloads with inherent concurrency
- **Primary challenge**
  - High contention-free overhead

Overview of our STM Implementation

- **Block-based mapping of shared memory location**
  - More general than object-based mapping
- **Write-barrier follows buffered-write policy**
  - Writes are buffered and written to global space only when transaction is guaranteed to commit
- **Read-barrier follows invisible-read policy**
  - Read-barrier does not write to shared meta-data
- **Checkpoint-barrier records original value for writes to contention-free locations**
- **Retry is done through setjmp and longjmp**
- **Report runtime statistics to help identify STM bottleneck**

STM characteristics	IBM STM compiler
Isolation	Weak
Granularity	8 byte block
Direct/Deferred Update	Deferred or Lazy
Concurrency Control	Optimistic
Synchronization	Blocking
Conflict Detection	Early(read after write) Late (write after write)
Inconsistent Reads	Tolerated(signals and infinite loop checks)
Conflict Resolution	Polite
Nested Transaction	Flat
Exception	terminate

Compiler Instrumentation for STM

- **Manual instrumentation of STM barriers are time-consuming and error-prone**

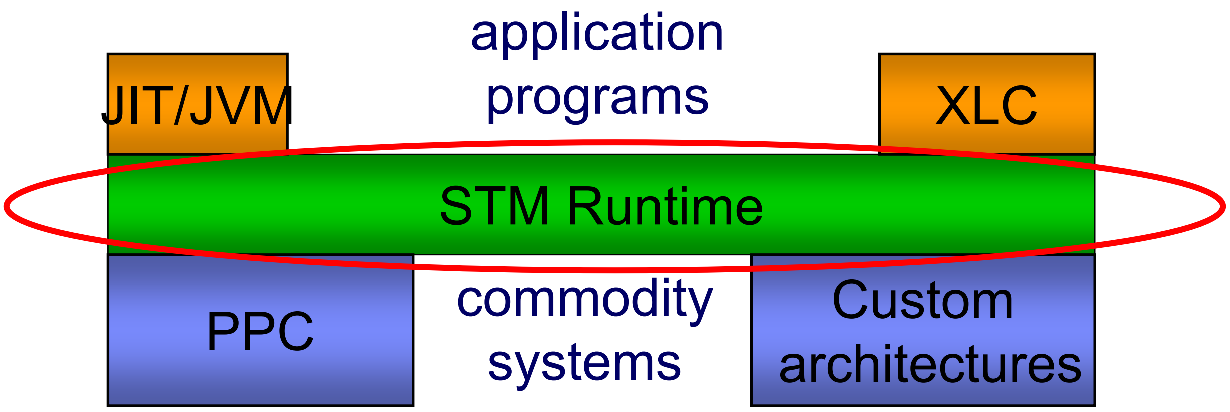
- **Transformation implemented I**
  - make all STATS=on
  - Instrument memory references in transactional scope to stm read/write-barriers
  - Instrument procs that are called by transaction
  - Checkpoint of writes to conflict-free locations
  - Reduce barrier overhead for conflict-free locations
  - Replacement of malloc routines to stm equivalent ones
  - Warning messages and statistics report

STM runtime statistics

Statistic	Description
READ_ONLY_COMMITS	Number of committed transactions with no writes
READ_WRITE_COMMITS	Number of committed transactions with writes
TOTAL_COMMITS	Number of successfully committed transactions
TOTAL_RETRIES	Number of retried transactions
AVG_RETRIES_PER_TXN	Average number of retries per committed transaction
MAX_NESTING	Maximum level of transaction nesting
READ_SET_SIZES	Unique locations in read sets of committed transactions
WRITE_SET_SIZES	Unique locations in write sets of committed transactions
AVG_READ_SET_SIZE	Average number of unique locations in read set per transaction
AVG_WRITE_SET_SIZE:	Average number of unique locations in write set per transaction
READ_SET_MAX_SIZE	Maximum number of unique locations in a read set
WRITE_SET_MAX_SIZE	Maximum number of unique locations in a write set
READ_LIST_MAX_SIZE	Maximum number of locations in a read list
WRITE_LIST_MAX_SIZE	Maximum number of locations in a write list
DUPLICATE_READS	Number of transactional reads of locations previously read in the same transaction
PCT_DUPLICATE_READS	Percentage of transactional reads of locations previously read in the same transaction
DUPLICATE_WRITES	Number of transactional writes to locations previously written in the same transaction
PCT_DUPLICATE_WRITES	Percentage of transactional writes to locations previously written in the same transaction
NUM_SILENT_WRITES	Number of transactional writes of already stored value
PCT_SILENT_WRITES	Percentage of transactional writes of already stored value
READ_AFTER_WRITE_MATCHES	Number of transactional reads that follow a transactional write of the same location in the same transaction
PCT_READ_AFTER_WRITE	Percentage of transactional reads that follow a transactional write to the same location in the same transaction
NUM_MALLOCS	Number of calls to `malloc` inside transactions
NUM_FREES	Number of calls to `free` inside transactions
NUM_FREE_PRIVATE	Number of calls to `free` blocks allocated in that transaction

STM Runtime

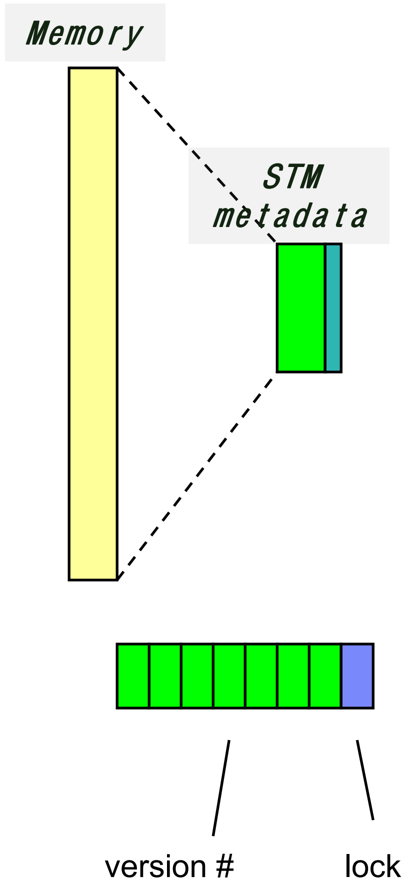
TM Project STM Runtime



- STM runtime written in C
- Supports:
  - C/C++ programs directly
  - JIT/JVM
  - XLC compiler
- Runs on:
  - Commodity platforms: AIX/Linux, PPC/X86, 32/64 bit
  - Probably Solaris/Sparc, but not tested
  - hardware acceleration of concurrency (models)

Software TM

- Use metadata to synchronize transactional access to memory
- Each shared memory location that may be accessed by transactions is associated with a metadata entry
- A metadata serves as:
  - Lock on associated data
  - Version number of updates to associated data
- A thread writes to memory only while it is holding the associated metadata lock
- When a transaction releases a metadata lock, it also increments its version number



- If a transaction
  - reads a metadata version number,
  - then reads an associated data,
  - and then checks the metadata and finds it unchanged
- then the transaction
  - is guaranteed that the data was not updated between the time it first read the metadata and the time it reread the metadata

Per-Thread Private STM Information

- Read set information:
  - Metadata location
  - Metadata value
- Write set
  - Address
  - Value
  - Size
  - Metadata location
- Statistics
- Status
- Level of nesting
- Lists of mallocs, frees, modified local variables
- …

STM Operations

```
stm_begin(...)
stm_end(...)
stm_read(addr,...)
stm_write(addr,value,...)
stm_malloc(size,...)
stm_free(ptr,...)
stm_checkpoint(addr,...)
stm_stack_range(min,max,..)
...
```

Example

```
...
atomic {
  r1 = a;
  r2 = b;
  a = r2;
  b = r1;
}
...
```

```
...
{
    jmpbuf buf;
    setjmp(buf);
    stm_begin(buf,...); {
      r1 = stm_read(&a,...);
      r2 = stm_read(&b,...);
      stm_wtite(&a,r2,...);
      stm_write(&b,r1,...);
   } stm_end(...);
}
...
```

Transactional Reads

- **Several policies**
- **Invisible reads**
  - Readers do not write to shared metadata
  - Avoids unnecessary cache coherence traffic on metadata
  - Allows complete read concurrency
  - Difficult to guarantee consistency of read set
  - **Invisible reads with post-validation**
    - Guarantees consistency
    - Validation may be slow
  - **Invisible reads without post-validation**
    - Avoids slow validation
    - Relies on catching faults: segmentation faults, division by zero, …
    - Not robust
- **Visible reads**
  - Reads acquire metadata either exclusively or in reader-only mode
  - Easy to guarantee consistency of read set
  - May cause unnecessary cache coherence traffic on metadata
  - Prevents or reduces read concurrency

stm_read(addr)

```
if write_set_contains(addr)
   return value_in_write_set(addr)
meta = locate_meta(addr)
metaval = meta.orec
if metaval.locked
   conflict path ...
add_to_read_set(meta,metaval)
val = *addr
if !stm_validate()
   conflict path ...
return val
```


Transactional Writes

- **Several policies**
- **Encounter-time writes:**
  - Acquire metadata
  - Remember old value
  - Write to memory
  - At commit time:
    - If transaction succeeds, release metadata
    - If transaction fails, undo write, release metadata
- **Commit-time acquire:**
  - Just remember address and value
  - At commit time:
    - Acquire all metadata
    - Do all writes
    - Release all metadata
- **Encounter-time acquire commit-time write:**
  - Acquire metadata at encounter time
  - At commit time
    - Do all writes
    - Release all metadata

stm_write(addr,val)

```
meta = locate_meta(addr)
add_to_write_set(addr,val,meta)
add_to_ws_filter(addr)
```

Design configurations

- **Default:**
  - Consistent read sets
  - Global version number
  - Write set lock acquisition at the end of the transaction
  - 1M metadata data entries
  - 8 byte conflict unit
  - 512-bit write set Bloom filters
- **To use different policies and config, add:**
  - make all EXTRA_FLAGS="-DENCOUNTER_ACQUIRE“
- **Some config flags:**
  - NOINC_VALIDATION: To allow inconsistent reads and use signals to catch failures.
  - NOGLOBAL_VERSION: To avoid the use of a global version number.
  - ENCOUNTER_ACQUIRE: To acquire metadata locks on encountering transactional writes instead of at the end of the transaction
  - LOG_2_NUM_ORECS=<integer>: Log 2 of the number of metadata entries. Default 20.
  - LOG_2_BLOOM_FILTER_BITS=<integer>: Log 2 of the write set Bloom filter size. Default 9.
  - LOG_2_BLOCKS_SIZE=<integer>: Log 2 of the conflict unit block size. Default 3.
  - MAX_TXNS=<integer>: Maximum number of static transactions in the programs. Used only for statistics. Default 64.

STM syntax (prior to Draft C++ STM specification)

- **Transactions**
- 1 **#pragma** tm atomic [**default** (trans / notrans)]
- 2 {
- 3
- 4 }

- **Annotating function attribute**
- 1 **i n t** foo (**i n t** sum) a t t r i b u t e ((tm func)) ;
- 2 **i n t** foo (**i n t** sum)
- 3 {
- 4 **return** ++sum ;
- 5 }

- 1 **i n t** b [2 5] , j ;
- 2 **i n t** index [5] = {4 ,5 ,675 ,22 ,34};
- 3
- 4 **for**(j = 0 ;j<25; j ++)
- 5 b[j] = 0;
- 6
- 7 **#pragma omp parallel for**
- 8 {
- 9 **for** (j =0; j <25; j ++)
- 10 {
- 11 . . .
- 12
- 13 **#pragma** tm atomic
- 14 {
- 15 b [index [j]] = . . . ;
- 16 } 17 } 18 } 19 }