A Universal Construction for Transaction-based Programs

Tyler CRAIN†  Damien IMBS†  Michel RAYNAL*,:†
{tyler.crain, damien.imbs, raynal}@irisa.fr

*Institut Universitaire de France
†IRISA, Université de Rennes 1, France
More developments in

- Tyler Crain, Damien Imbs, Michel Raynal,
  *Towards a universal construction for transaction-based multiprocess programs*
  Tech Report 1971, 16 pages, IRISA, Université de Rennes 1 (France), 2011

This research is part of the Marie Curie ITN project TRANSFORM funded by the European Union FP7 Program (grant 238639)
Summary

- Motivation, issues and problem statement
- A (proved) Universal construction
- Discussion and conclusion
Concurrent programming

- A set of $n$ processes $p_1, p_2, \ldots, p_n$
  - Sequential
  - Reliable
- A set a concurrent objects
  - Each object is a linearizable (atomic)
  - All operations on an object appear as if they have been executed sequentially and this order respects real-time order
    - $op_1$ returns before $op_2$ starts $\Rightarrow op_1$ appears before $op_2$
  - Linearizability (atomicity) is composable
Accessing several objects simultaneously

- Provide the programmer with **locks**
- Aim of a lock:
  
  ```
  prevent data from being accessed concurrently
  ```

  Data point of view: its environment is then sequential

- According to the “syntactic sugar” and the language locks are semaphores, signals, monitors, serializers, etc.

- Are locks good???
  - Creating and forking threads is easy, but
  - Synchronizing them is hard!
  - Managing locks is difficult!
  - Locks cannot be composed!
The concept of a transaction

- The answer: provide the application programmer with a synchronization abstraction called transaction which is an atomic procedure

  ✭ Simple to use (with the hope to be efficient)
  ✭ No more locks, no need to handle conflicts, etc.

- Instead of
  lock $x$; read $x$; lock $y$; read $y$; write $x$; unlock $y$; unlock $x$

the programmer has only to write something like

  transaction { read $x$; read $y$; write $x$ }

(s)he no longer has to manage the underlying synchronization needed to face the concurrent accesses to the shared objects: this is now the job of the STM system!
The notion of abstraction level

- Aim: allow the programmer to concentrate on the problem(s) he has to solve and not on the base machinery needed to solve it
- Example 1: high level languages
- Example 2: Automatic garbage collection
- Example 3: “double click”
- Now STM: hide synchro “implementation details” to programmers
How to implement transaction atomicity

- Conservative approach
  - a lock (few locks) that serializes the transactions
  - works but is inefficient

- Optimistic approach: allows for speculative execution
  - If no conflict: commit the corresponding transaction
  - If conflict on a concurrent object: abort and restart
  - Remark: this is similar to deadlock resolution in classical parallel systems (with conflict $\equiv$ deadlock)
Managing transaction aborts

- Left to the application (similar to exception handling)
- Best effort approach
- Modify the pair “compiler + scheduler” in order each transaction is “practically always” committed
- Design schedulers that behave particularly well in appropriate workloads (e.g., read dominated workloads)
- Notion of irrevocable transactions: executed exactly once and cannot be aborted
- Notion of deadline-aware transactions
- Etc.
This talk

- At the programming level:
  - no notion of abort/commit
  - each transaction is executed exactly once

- No modification of the compiler or the scheduler

- The proposed STM system is a universal construction
  - Its input: any transaction-based concurrent program $P$
  - Its output: an execution of $P$ on a multiprocessor
  - It is proved (no transaction typing, not best effort!)
A universal construction: user programming level

- **Transaction (atomic procedure)**
  - Can access any nb of concurrent objects: \( t \)-objects
  - Assumption: when executed alone, a transaction always terminates

- **Non-transactional code**
  - Can access any nb of concurrent objects: \( nt \)-objects
  - Includes inputs/outputs (irrevocable code)

- \( t \)-objects \( \cap \) \( nt \)-objects = \( \emptyset \) (this adds no limitation)

- **Process** = (transaction ; non-trans code)*
Example (1)

- Two processes
- Three $t$-objects: Red, blue, yellow
- Two $nt$-objects: green, cyan
From an external observer’s point of view
A universal construction: Underlying system

- $m$ ($1 \leq m \leq n$) processors (simulators) $P_1, P_2, ..., P_m$
- They communicate through
  - atomic read/write registers
  - Compare&Swap registers:
    \[
    X.\text{Compare&Swap}(old, new) = \\
    \begin{cases} 
    \text{if } (X = old) & \text{then } X \leftarrow new; \text{return(true)} \\
    \text{else} & \text{return(false)} 
    \end{cases}
    \]
  - Fetch&Increment register:
    \[
    X.\text{Fetch&Increment}() = [X \leftarrow X + 1; \text{return}(X)]
    \]
Representing the $nt$-objects

- Directly in the memory shared by the $m$ processors
- Reminder: by definition they are linearizable
- It exists very nice packages with efficient linearizable implementations for base objects such as stacks, lists, sets, queues, etc.
Representing the $t$-objects

- Can be retrieved from a list shared by the processors
- This list contains all “committed” transactions
- Compare & Swap is used by a processor when it wants to commit a transaction on behalf of a process $p_i$
Each processor $P_x$ (1 ≤ $x$ ≤ $m$) is responsible for the progress of a set of processes

array $OWNED\_BY[1..m]$ such that

$OWNED\_BY[x] = \text{the set of processes “owned” by } P_x$
Helping mechanism

- A processor $P_x$ speculatively executes the next transaction of a process $p_i$ it owns and then tries to “commit” it.
- If it does not succeed it requires the other processors to help it.
- To that end, the processors share two data structures:
  - A logical clock $CLOCK$ (accessed by Fetch&Add)
  - An array $STATE[1..n]$
The size $n$ array $STATE$

$STATE[i] = $ current state of the simulation of process $p_i$

- $STATE[i].tr_n$: seq nb of $p_i$'s next transaction
- $STATE[i].local_state$: local state of $p_i$ just before executing its next transaction
- $STATE[i].help_date$:
  - $STATE[i].help_date = +\infty$: no help is required
  - $STATE[i].help_date \neq +\infty$: help is required
- $STATE[i].last_ptr$: initially points to $FIRST$, then updated to shortcut a prefix of the list
Three (non-trivial) technical issues

- Prevent a processor from always helping the others without making any progress on the processes it owns.
- As the list of “committed” transactions pointed to by \textit{FIRST} can increase forever, ensure that an (asynchronous) processor will eventually attain the end of this list when it wants to “commit” a transaction.
- Ensure that each transaction is “committed” no more than once.
- These issues are solved by appropriate enrichments of the base helping mechanism.
Structure of a simulator $P_x$

- Select from $STATE[1..n]$ a process $p_i$ in order to execute its next transaction $T$
- Speculative execution of $T$ (in $P_x$ local memory)
- If $T$ has not yet been committed and its “commit predicate” is true, tries to commit it
- If $p_i$ is owned by $P_x$ and has been committed (by $P_x$ or another processor $P_y$)
  - Execute non-transactional code that follows $T$ in $p_i$
  - Modify $STATE[i]$ accordingly
  - If $p_i$ has terminated, suppress it from $OWNED[x]$
- If $OWNED[x] \neq \emptyset$: restart from the first item
Proof

• It is 4.5 (11 point, A4) pages long (11 lemmas)
• Let $P$ be a transaction-based multiprocess program
• Theorem: Any simulation of $P$ by the universal construction (on $m$ processors) is an execution of $P$ that could have been produced by executing $P$ on $n$ processors (i.e., it is a correct execution of $P$)
Summarizing the construction

- It is for time-free transaction-based programs
- It is lock-free (no lock is ever used)
- It uses Compare&Swap on pointer registers
- It imposes no restriction on the concurrency pattern
- It works for finite and infinite programs
- The nb of tries to commit a transaction is bounded
- Helping mechanism can be improved to be more efficient
- Crash of a processor entails only the crash of the processes it owns
Conclusion

- A universal construction for transaction-based programs
- Proved!
- Better understanding of the nature of the atomicity provided to programmers in order to cope with concurrency issues
- Next step: replace the list of “committed” transactions by something more efficient
- Next step: make it tolerant to processor crashes