Weaving Atomicity Through Dynamic Dependence Tracking

Suresh Jagannathan

joint work with Lukasz Ziarek, Philip Schatz, Jeremy Orlow
Jan Vitek

PURDUE UNIVERSITY
Software Transactions

• Classical concurrency control abstractions using locks requires a great deal of programmer care to ensure correctness and efficiency.
  ★ deadlocks, priority inversion, etc.

• Transactions can significantly relieve this burden:
  ★ Provide serializability properties
    ✦ Atomicity: effects of updates seen all-at-once or not-at-all
    ✦ Isolation: transactions appear to execute one-at-a-time
  ★ API
    ✦ Start a transaction
    ✦ Validate serializability
      › Commit
      › Abort
    ✦ Contention management to schedule transactions in case of conflicts
Composability

• Transactions encourage modular reasoning
  ★ Unlike lock-based mutual-exclusion, we can reason about transactional behavior without exposing details about:
    ✦ The order in which transactions execute
    ✦ The data transactions protect

• Transactions support composability
  ★ The manner in which transactions are combined does not affect correctness

• What happens when transactions need to communicate in ways that (presumably) break isolation and hinder composability?
  ★ Producer/consumer pipelines
  ★ Message-passing primitives (ICFP’06)
  ★ Exceptions or faults
  ★ Interaction with lock-based code (ECOOP’06)
Composability

- Not easy to prohibit these violations
  ★ Violations may be buried under many software layers.
- May be necessary for correctness and performance
  ★ Naive implementations would block communication until the transaction commits.
- Specific instance of more general question of legacy support
- Open-nested transactions are one alternative
  ★ Impose an abstract notion of serializability based on higher-level invariants of a concurrent data structure
  ★ Defining abstract serializability for complex communicating actions is non-trivial
  ★ It’s difficult to reason about composability
  ✦ removing the effect of a message send on a channel may require compensating actions on all behavior that witnessed the send
Pragmatics

• Real programs frequently violate isolation:
  ★ Apache: 68% (mostly acyclic communication patterns)
  ★ JavaGrande: 50% (mostly cyclic)
  ★ Splash: 13% (mixture of both)

• These programs exploit some form of producer/consumer communication pattern within their critical sections.

• Using existing techniques requires either:
  ★ transforming the code to break cycles, and explicitly order communication actions, or
  ★ allowing communication in an open-nested transaction:
    ✦ permits producers to notify consumers before the outer transaction completes
    ✦ use compensations to undo actions of sends in aborted outer transactions
Robustness

• Once isolation is relaxed, arbitrary actions internal to the transaction may impact global behavior.
  ★ Consider exceptions or errors within a transaction
  ★ Or, retry primitives that explicitly abort the transaction

• How do we rationalize visible effects within the transaction that are no longer valid?
  ★ Reminiscent of a cascading abort model
  ★ But, how do we keep track of induced dependencies?
  ★ Can we constrain the scope of influence?

• Hard for applications to encode these dependencies
  ★ Non-determinism
  ★ Understanding these dependencies involve reasoning about complex data and control-flow across different threads
• Consider a language with:
  ★ Closed-nested transactions
  ✦ Retry operations for explicit abort
  ✦ Allow threads to be spawned within transactions
★ Message-passing communication primitives:
  ✦ First-class typed channels
  ✦ Synchronous send and receives
    ▷ Any object (including procedures, references, or channels) can be transmitted along a channel
  ✦ Events and choice (CML)
★ First-class references

• Goal: Devise a precise semantics and implementation for transactional behavior in the presence of channel communication.
Model
Model

Transaction T
**Model**

Transaction T

\[\text{send}(ch, x)\]

Transaction T'

\[\text{recv}(ch)\]
Inter-Transaction Dependency
Model

Transaction T

send(ch,x)  
retry()

Transaction T'

recv(ch)

Inter-Transaction Dependency
Inter-Transaction Dependency Model

Transaction T
- send(ch,x)
- retry()

Transaction T'
- recv(ch)

Inter-Transaction Dependency
Inter-Transaction Dependency
Inter-Transaction Dependency
Inter-Transaction Dependency
Interaction with Non-Transactional Code

Transaction T

Thread t
Interaction with Non-Transactional Code

Transaction T

send(ch, x)

recv(ch)

Thread t
Interaction with Non-Transactional Code

Transaction T

send(ch, x)

retry()

recv(ch)

Thread t
Interaction with Non-Transactional Code

Transaction T

Thread t

send(ch, x)
retry()
recv(ch)

9
Interaction with Non-Transactional Code

Transaction T

Thread t

send(ch,x)

recv(ch)

retry()
Interaction with Non-Transactional Code

Thread $t$

Transaction $T$

send($ch, x$)

recv($ch$)

retry()

restore thread to state prior to channel communication initiated by the transaction
Interaction with Non-Transactional Code

Transaction T

Thread t

send(ch,x)
retry()
recv(ch)

restore thread to state prior to channel communication initiated by the transaction

strong atomicity
Semantics

- Transactions that communicate via non-isolated communication actions must commit together
- If any transaction in a group of transactions aborts, the entire group aborts
- Reads and writes to memory still implemented using basic transactional machinery:
  - Pessimistic writes
  - Optimistic reads
- A transactional group can commit only if each transaction in the group is conflict-free
  - Operations performed within a transaction group are serializable with respect to other transactions outside the group.
  - Allows progress in the presence of synchronous communication
Observations

• Transactions must still adhere to serializability constraints on memory accesses with respect to other transactions.

• Non-isolated actions augment constraints:
  ★ The success of a transaction commit within a transaction group depends on the successful commit of all other transactions within that group.
  ★ Non-local reasoning limited to communication actions.

• Congruence:
Observations

• Transactions must still adhere to serializability constraints on memory accesses with respect to other transactions

• Non-isolated actions augment constraints:
  ★ The success of a transaction commit within a transaction group depends on the successful commit of all other transactions within that group
  ★ Non-local reasoning limited to communication actions

• Congruence:
Observations

• Transactions must still adhere to serializability constraints on memory accesses with respect to other transactions

• Non-isolated actions augment constraints:
  ★ The success of a transaction commit within a transaction group depends on the successful commit of all other transactions within that group
  ★ Non-local reasoning limited to communication actions

• Congruence:
• Transactions must still adhere to serializability constraints on memory accesses with respect to other transactions

• Non-isolated actions augment constraints:
  ★ The success of a transaction commit within a transaction group depends on the successful commit of all other transactions within that group
  ★ Non-local reasoning limited to communication actions

• Congruence:
Observations

- Transactions must still adhere to serializability constraints on memory accesses with respect to other transactions.

- Non-isolated actions augment constraints:
  - ★ The success of a transaction commit within a transaction group depends on the successful commit of all other transactions within that group.
  - ★ Non-local reasoning limited to communication actions.

- Congruence:
Observations

• Transactions must still adhere to serializability constraints on memory accesses with respect to other transactions

• Non-isolated actions augment constraints:
  ★ The success of a transaction commit within a transaction group depends on the successful commit of all other transactions within that group
  ★ Non-local reasoning limited to communication actions

• Congruence:

\[ T_1 \xrightarrow{\text{send}(ch,v)} T_2 \xrightarrow{x = \text{recv}(ch)} T_1 \text{commit} \]
\[ T_1 \xrightarrow{v} T_2 \xrightarrow{x = v} T_2 \text{commit} \]
Issues

• Key issues:
  ★ How do we effectively track communication actions across transactions?
  ★ How do we deal with nesting?
  ★ How do we build transaction groups?
  ★ What happens when a communication event within a transaction is paired with an action that occurs outside?
  ★ Progress properties. Loss of obstruction-freedom?

• Build a runtime communication graph that records dependencies among communication actions
  ★ Structure of the graph determines how transactions coalesce into groups
Approach

spawn(atomic(g) (...))

atomic f
spawn(atomic(g) (...))
Approach

\[
\text{spawn(atomic(g) (...)})
\]

\[
\text{send(c, x)}
\]

\[
\text{recv(c)}
\]

atomic f

atomic g
Approach

spawn(atomic(g) (...))

send(c, x)

recv(c)

recv(c)

send(c, y)

atomic f

atomic g

atomic h
Approach

spawn(atomic(g) (...))

send(c, x)

recv(c)

recv(c)

retry

atomic f

atomic g

atomic h

send(c, y)
Approach

spawn(atomic(g) (...))

send(c, x)

recv(c)

retry

recv(c)

send(c, y)

atomic f

atomic g

atomic h
Behavior

- Atomic sections delimited by programmer
- Safety violations may be due to serializability violations or explicit retry
- Save continuations to allow thread execution to resume within a partially executed atomic section
- Abort semantics
  - Revert control to globally consistent state based on communication events observed within an atomic section.
• What are the best continuation-save points within an atomic section?
• Memoization opportunities in the presence of synchronous communication
  ★ Can lead to substantial savings: 40% execution improvement on STMBench7
• Only need to record a specific communication event once
  ★ Only a single edge between two atomic sections needs to be recorded
• Use weak references to collect unreachable portions of the graph
• Need to track read and write operations to shared data accessed outside atomic blocks
  ★ State of live variables at a communication point must be saved
  ★ Avoid saving variables that have been previously recorded and which have not changed
  ★ Use write barriers
Overheads

- Implemented in MLton
  - Insertion of write barriers
  - hooks in the CML library to update the dependency graph
- Overheads to maintain dependency graph small, roughly 6%
  - eXene: a windowing toolkit
  - Swerve: a web server

<table>
<thead>
<tr>
<th></th>
<th>Threads</th>
<th>Channels</th>
<th>Events</th>
<th>Shared Writes</th>
<th>Shared Reads</th>
<th>Graph Size (MB)</th>
<th>Runtime Overheads (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle</td>
<td>205</td>
<td>79</td>
<td>187</td>
<td>88</td>
<td>88</td>
<td>.19</td>
<td>.59</td>
</tr>
<tr>
<td>N-Body</td>
<td>240</td>
<td>99</td>
<td>224</td>
<td>224</td>
<td>273</td>
<td>.29</td>
<td>.81</td>
</tr>
<tr>
<td>Pretty</td>
<td>801</td>
<td>340</td>
<td>950</td>
<td>602</td>
<td>840</td>
<td>.74</td>
<td>6.23</td>
</tr>
<tr>
<td>Swerve</td>
<td>10532</td>
<td>231</td>
<td>902</td>
<td>9339</td>
<td>80293</td>
<td>5.43</td>
<td>6.60</td>
</tr>
</tbody>
</table>
Conclusions

• Can rationalize a semantics and implementation for atomic transactions that engage in non-isolated communication actions.

• Makes transactions more useful in distributed message-passing environments.

• Improve robustness and expressivity of concurrency and synchronization abstractions
  ★ Valuable for long-lived applications
  ★ Useful to help coordinate activities of dynamically-related threads

• Provides useful safety guarantees

• Can be implemented with relatively small overhead