FastLane is Opaque – A Case Study in Mechanized Proofs of Opacity

Gerhard Schellhorn
Universität Augsburg, Germany

Monika Wedel Oleg Travkin
Jürgen König Heike Wehrheim
Universität Paderborn, Germany

Overview

- Software Transactional Memory (STM)
- Correctness of STM Implementations: Opacity.
  Proof by refinement of `TMS2`-Automaton
- The `FastLane` implementation: Algorithm + Switching
- Mechanized proofs using the KIV theorem prover
  - "`FastLane` refines `TMS2`" $\Rightarrow$ `FastLane` is Opaque
  - Switching between correct implementations
  - Instantiation with `FastLane`+ Switching

Software Transactional Memory

- Synchronizing Threads on shared data using locks is often difficult and error prone.
- Instead adapt the concept of transactions from data bases to programs.
- Extend programming language with
  `success := tryatomic { <some code> }`
- All threads may execute such atomic blocks using arbitrary shared data.
  - success = true: the transaction committed: It "looks like" the code is executed atomically without interference.
  - success = false: The execution aborted due to conflicting accesses and there is no effect.
    Retry with a while loop possible until success = true:
    atomic { <some code> } automatically retries until success
- Very simple, inefficient implementation:
  Use one global lock, always commit (or abort randomly)

Implementation of STM

- Implementation of an STM provides four programs: `BEGIN`, `READ`, `WRITE`, `END`
- Compiler supports implementation by instrumenting code for atomic blocks.

For code block `success := tryatomic { x := x + y }` (with shared variables x,y) the compiler generates

```
BEGIN()
regx := READ(x)
regy := READ(y)
regx := regx + regy
WRITE(x, regx)
success := END()
```

Control structure is left as is. Load from/Store to main memory is replaced with calls to `READ`/`WRITE`

Strategies for implementing STM

- There is a large number of different algorithms that implement STM (NoRec, TL2, TML, ..., `FastLane`)
- They can be classified according to two dimensions:
  - The eager strategy updates main memory in `WRITE`, lazy strategy collects all updates locally in a writeset and applies them all in `END`.
  - Pessimistic detection of conflicting reads/writes aborts transaction already in `READ`/`WRITE`. Optimistic strategy detects conflicts only in `END`.

Overview

- Software Transactional Memory (STM)
- Correctness of STM Implementations: Opacity.
  Proof by refinement of `TMS2`-Automaton
- The `FastLane` implementation: Algorithm + Switching
- Mechanized proofs using the KIV theorem prover
  - "`FastLane` refines `TMS2`" $\Rightarrow$ `FastLane` is Opaque
  - Switching between correct implementations
  - Instantiation with `FastLane` + Switching

Correctness of STMs

- The simplest criterion is serializability: Aborted transactions have no effect. The effect of committed transactions must be as if they were executed sequentially. The memories in between transactions are called snapshots.
- strict serializability additionally requires: if transaction T1 finishes before T2 starts then T1 must be before T2 in the sequential order
- Opacity [Guerraoui, Kapalka, PPOPP, 2008] additionally requires that aborted transactions read from a *single* snapshot of memory.

Example for opacity

initially $x = y = 0$

snapshot invariant $x = y$

tryatomic $\{$ $x := x + 1$ $y := y + 1$ $\}$	$\parallel$	**tryatomic** $\{$ $localx := x$ $localy := y$ **while** $localx \neq localy$ **do skip** $\}$

- Assume eager writing with checks for conflicts at the end
- Right transaction may loop forever if it reads x,y in between the two updates of the left.
- Without opacity replacing the loop with $localz := 1/(localy - localx + 1)$ results in divide by zero.
- With opacity atomic code can be verified sequentially assuming the snapshot invariant.

Verification of Opacity: IO Automata refinement

- An established strategy for verification of opacity is: Encode the steps of the algorithms for `BEGIN`, `READ`, `WRITE` and `END` as steps of a transition system `IMPL`
  (formally: an IO Automaton).
- Show that `IMPL` refines the automaton `TMS2` [Doherty, Groves, Luchangco, Moir, FAC 2013] ($\texttt{IMPL} \leq \texttt{TMS2}$). This implies opacity.

An *IO automaton* A consists of

- state set $\mathit{states}(A)$ and initial ones $\mathit{start}(A) \subseteq \mathit{states}(A)$,
- internal and external actions $\mathit{act}(A) = \mathit{int}(A) \dot{\cup} \mathit{ext}(A)$
- transition relation $\mathit{steps}(A) \subseteq \mathit{states}(A) \times \mathit{act}(A) \times \mathit{states}(A)$.

Refinement of $C \leq A$ require that for each concrete run there is an abstract run with the same *external actions*

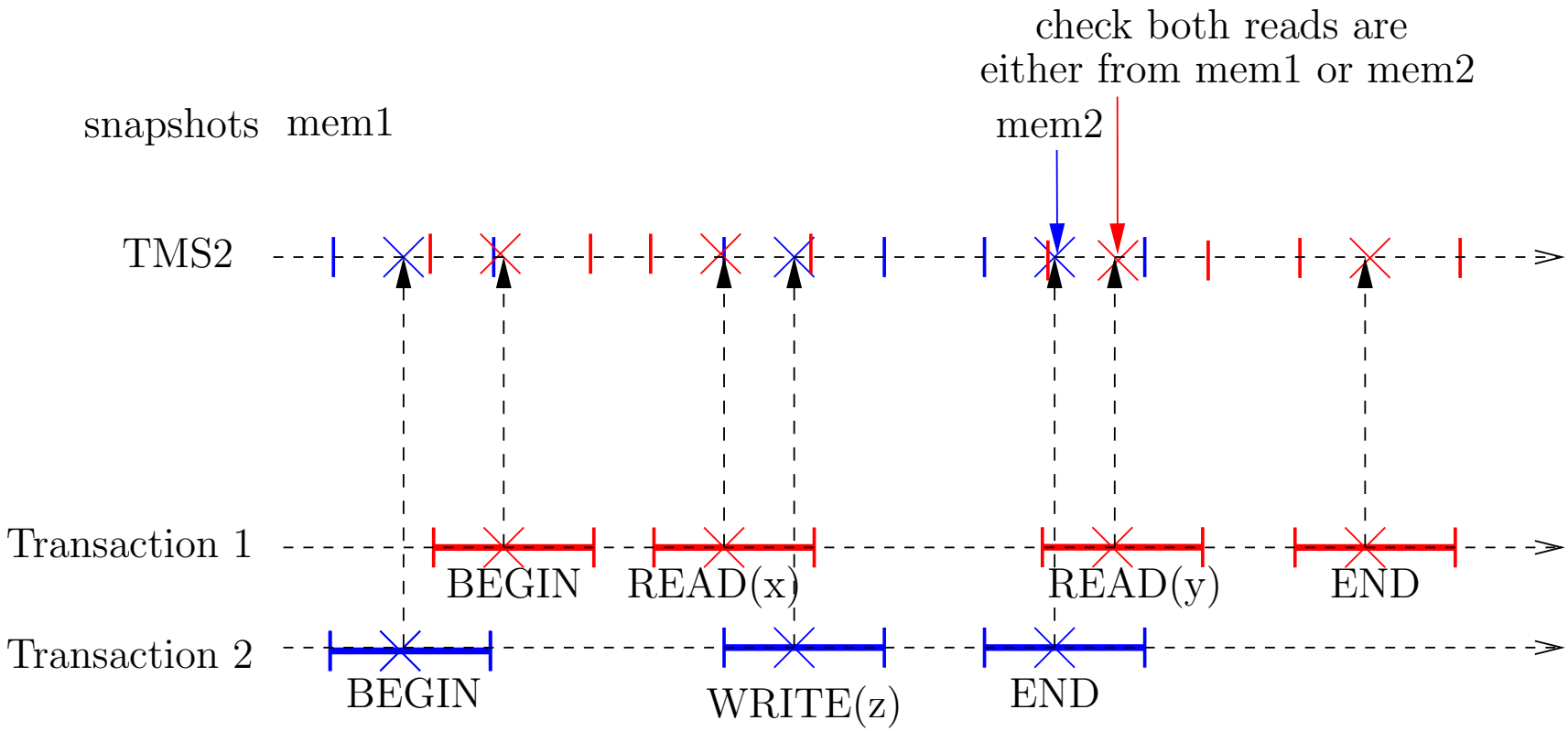
Opacity and the TMS2 automaton

External actions for Opacity are:

- $inv(OP, tid, input)$: transaction tid invokes $OP \in \{\texttt{BEGIN}, \texttt{READ}, \texttt{WRITE}, \texttt{END}\}$ with input in
- $ret(OP, tid, out)$: return from OP with output out

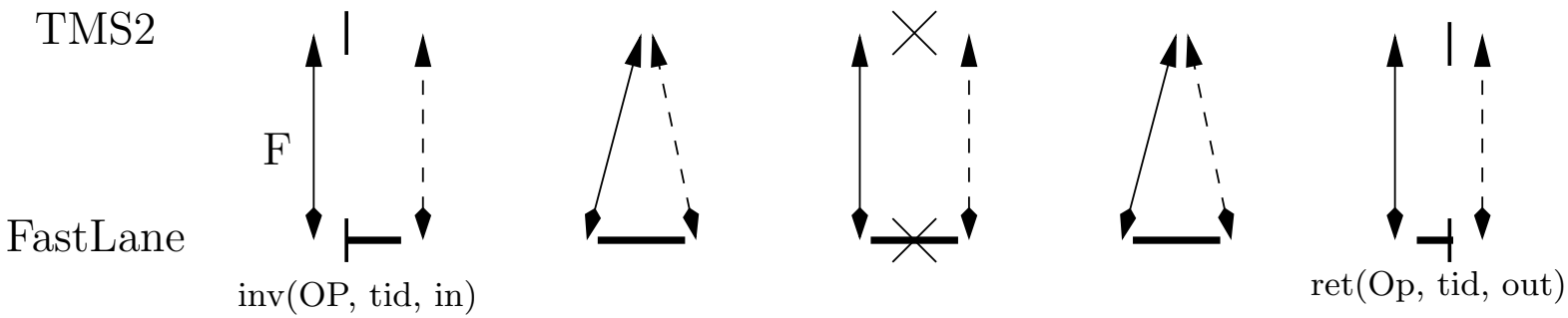
- TMS2 is a specification of opacity:
  It stores all snapshots created by committed transactions.
- TMS2 has a single internal step for each of the four programs (the effect point of the algorithm; similar to a lin. point).
- READ, WRITE are lazy: they use a readset/writeset.
- READ checks that all read values are from some single snapshot (it aborts if this not the case)
- END checks that reads (and writes) are compatible with some (the last) snapshot. Aborts, if not.
- BEGIN remembers the earliest snapshot it can read from.
- If END is successful, it creates a new snapshot.

Verification problem



Forward Simulation

Define an forward simulation F, such that diagrams commute:
F may be assumed (continuous line) before the step,
must be shown (dashed line) after the step



Main problems of defining F:
Where to place the effect points (marked with $\times$)
How does main memory + local data correspond to snapshots?

Overview

- Software Transactional Memory (STM)
- Correctness of STM Implementations: Opacity.
  Proof by refinement of `TMS2`-Automaton
- The `FastLane` implementation: Algorithm + Switching
- Mechanized proofs using the KIV theorem prover
  - "`FastLane` refines `TMS2`" $\Rightarrow$ `FastLane` is Opaque
  - Switching between correct implementations
  - Instantiation with `FastLane`+ Switching

The idea of FastLane

- Taken from [Wamhoff et al, 2013]
- Typical STM implementations IMPL(e.g. NoRec, TL2) are tuned for optimal concurrency under high loads (many threads)
- When there are few threads (e.g. below number of multicores) then the overhead is quite noticeable.
  $\Rightarrow$ Use the FastLane implementation.
- When there is a single thread, no instrumentation of the code is necessary at all (READ is just "load from memory").
- Idea: Generate three versions of the four programs: The uninstrumented version SEQ, the FastLane code, and a version for high loads: IMPL.
- Switch heuristically (in idle states) between the versions depending on the number of threads active.

Details on the FastLane Algorithm

- The four programs of FastLane are ca. 80 lines of code.
- One thread is master (is on the "fast lane").
- All other threads are helpers.
- Master writes directly (is eager), never aborts.
- Master uses a counter: value is odd iff a master is running.
- Variables have an additional dirty field overwritten by master with counter
- Helpers collect reads and writes in a read and write set.
- Helpers remember initial value of counter, and use it in READ, WRITE and END to ensure no interference.
- masterLock is used to switch master, helperLock protects helpers from each other when committing

Overview

- Software Transactional Memory (STM)
- Correctness of STM Implementations: Opacity.
  Proof by refinement of `TMS2`-Automaton
- The `FastLane` implementation: Algorithm + Switching
- Mechanized proofs using the KIV theorem prover
  - "FastLane refines `TMS2`" $\Rightarrow$ `FastLane` is Opaque
  - Switching between correct implementations
  - Instantiation with `FastLane` + Switching

FastLane $\leq$ TMS2

Crucial properties of simulation:

- If there is no master, then memory = latest snapshot.
- If there is a master, and it holds master Lock, then current memory is last snapshot of TMS2 plus the writes done by master (as stored in masters writeset of TMS2).
- If a helper holds master Lock while committing, then for every variable x:
  - $dirty(x) \neq counter$: memory has the value of latest snapshot.
  - $dirty(x) = counter$: x stores the value of the helper
- Lots of properties specific to locations in the code (ca. 80 lines of specification)

Switching between implementations

- Given correct implementations $C1$ and $C2$ of an interface A ($C1 \leq A$, $C2 \leq A$)
- When is an implementation $switch(C1, C2)$, that switches between C1 and C2 correct: $switch(C1, C2) \leq A$?
- In our case: $\texttt{SEQ} \leq \texttt{TMS2}$, $\texttt{FastLane} \leq \texttt{TMS2}$, and $\texttt{IMPL} \leq \texttt{TMS2}$
- Is $switch(switch(\texttt{SEQ}, \texttt{FastLane}), \texttt{IMPL}) \leq \texttt{TMS2}$?
- Challenges
  - Must allow for shared state between $C1$ and $C2$: main memory
  - Must not fix a specific switching scheme
    $\Rightarrow$ define a class of possible $switch(C1, C2)$
  - Must talk about running processes rather than running transactions (as TMS2 does)
  - Steps have additional restrictions.

A class of switching automata

Given two automata $C1$ and $C2$ an automaton C is in $\mathit{switch}(C1, C2)$, if it satisfies the following criteria:

- All states $s_C \in \mathit{states}(C)$ have a boolean mode-component. $s_C.\mathit{mode}$ to determine which algorithm is active.
- All states $s_C \in \mathit{states}(C)$ with $s_C.\mathit{mode} = \mathit{true}$ allow to extract via $s_C.s1 \in \mathit{states}(C1)$. Similarly, there is a selector $s_C.s2 \in \mathit{states}(C2)$, when $s_C.\mathit{mode} = \mathit{false}$.
- The transition relation of $\mathit{step}(C)$ can be split into $\mathit{step}1, \mathit{step}2, \mathit{step}3$: $\mathit{step}1$ and $\mathit{step}2$ must correspond to steps of $C1$ and $C2$, step are new internal steps, that enable switching.

Correctness of Switching

Theorem

Let $C1 \leq_{(F1)} A$, $C2 \leq_{(F2)} A$ *and* $C \in switch(C1, C2)$. *Then* $C \leq_{(F)} A$ *with* F *defined as*

$$
\begin{array}{rcl}
F0(s_C, s_A) & \Longleftrightarrow & \textbf{if } s_C.mode \textbf{ then } F1(s_A, s_C.s1) \wedge Inv1(s_C) \\
 & & \textbf{else } F2(s_A, s_C.s2) \wedge Inv2(s_C) \\
F(s_C, s_A) & \Longleftrightarrow & Inv(s_C) \wedge F0(s_C, s_A)
\end{array}
$$

holds under the proof obligations given in the [SEFM 18] paper.

Intuition: When $C1$ is running (mode = true), then $F1$ holds, and $Inv1$ guarantees that the state of $C2$ is not modified irrecoverably. INV is a new invariant that glues together the implementations and additional new state.

The combined implementation

The combined implementation has

- steps for (un)registering threads adding to a set (to count number of threads)
- a map: registered threads $\mapsto$ running transactions
- switches only when no transaction is running
- switches to `SEQ` only when there is at most one thread
- Otherwise switches from `FastLane` to `IMPL` when number of threads is bigger than constant c.

Correctness of combined implementation

Theorem

switch(*switch*(*SEQ*, *FastLane*), *IMPL*) $\leq$ *TMS2*, *holds under certain sanity conditions for* *IMPL*:

- *IMPL does not modify the registered threads*
- *IMPL correctly adds/removes a running transactions at the start of* *BEGIN* *resp. the end of* *END*.
- *switching from and to* *IMPL* *establishes the required invariants.*

Summary and Outlook

- We have verified opacity of the `FastLane` algorithm.
- A theory was developed for switching between implementations, and the instance for `FastLane` was verified
- All proofs with the KIV tool are online at
  `http://www.informatik.uni-augsburg.de/swt/projects/FastLane.html`
- Related Work on Opacity:
  - Several papers verify opacity of other algorithms using `TMS2`
  - Alternative: Model checking using reduction theorems has been used
- Related for switching [Armstrong, Dongol, FORTE 17]: Hybrid STMs, where individual threads switch between hardware and software transactions.
- Open question: Are there other instances of switching?