7th European Lisp Symposium

Massimiliano Ghilardi

May 5-6, 2014

IRCAM, Paris, France

High performance concurrency in Common Lisp

—

hybrid transactional memory with STMX

Beautiful and fast concurrency

in Common Lisp

—

hybrid transactional memory

with STMX

STMX: hybrid transactional memory

- Motivations: why now
- STMX is…
- Examples and API
- Main features
- Strengths & weaknesses
- Performance
- Q&A

Motivations: why now (1/3)

Parallel programming **CANNOT** be avoided

Recent tablets and smartphones are usually dual-core or quad-core

Consumer CPUs are increasingly multi-core

▪ Dual-core	Intel Pentium D (2005) AMD Athlon 64 X2 (2005)
▪ Quad-core	Intel Xeon X 32xx (2007) AMD Opteron 8xxx (2007)
▪ Octa-core	Intel Xeon E7xxx (2008) AMD Opteron Magny-Cours (2010)
▪ 12-core	Intel Xeon E5-269x v2 (2013) AMD Opteron Magny-Cours (2010)
▪ 16-core	AMD Opteron Interlagos (2011)

Commercial & high-end systems are even more parallel

Motivations: why now (2/3)

Parallel programming is **NOT** a solved problem:
many different programming paradigms exist,
each with its (strengths and) weaknesses

- Multi-threading with locks and mutable shared state
- Message passing
- Futures and promises
- π-calculus
- Coroutines, continuations, channels…
- Transactional memory (TM)

Many paradigms choose to avoid mutable shared state;
transactional memory promises to tame it.

Motivations: why now (3/3)

Transactional memory – a quick history

1986	Initial idea, requires unavailable HW support
1995	New idea: SW-only transactions
2005	First public implementation in Haskell
2006	Improvement: guaranteed read consistency
2006	CL-STM born and immediately abandoned
2007-2012	Further improvements, libraries for many languages: C/C++, Java, C#, OCaml, Python…
2012	IBM and Intel announce HW implementation in one year
2013, March	Hybrid transactional memory designed for Intel HW
2013, May	STMX released, SW-only transactions
2013, August	STMX adds hybrid transactions for Intel HW

STMX is… (1/2)

Transactional memory is an alternative synchronization mechanism for mutable shared state.

Gives strong correctness & thread-safety guarantees.

Elegant and intuitive to use.

Immune from:

- Deadlocks
- Starvation
- Priority inversion
- Non-composability
- Non-determinism
- Race conditions

Disadvantages:

- Prone to near-livelocks under high contention
- Historically poor performance – solved by hybrid implementations

STMX is… (2/2)

An actively maintained, highly optimized implementation of hybrid transactional memory

Developed in approximately 3 months of spare time (probably less)

One of the first published implementations of hybrid transactional memory (August 2013)

Freely available under LLGPL - http://www.stmx.org/

Portable – runs on ABCL, CCL, CMUCL, SBCL (~ECL)
tested on x86, x86-64, arm, powerpc

Examples and API (1/2)

```
(quicklisp:quickload :stmx)
(use-package :stmx)

(quicklisp:quickload :stmx.test)
(fiveam:run! 'stmx.test:suite)

(defvar *v* (tvar 42))
(print ($ *v*))                  ;; prints 42

(atomic
  (if (oddp ($ *v*))
      (incf ($ *v*))
      (decf ($ *v*))))           ;; *v* now contains 41
```

TVAR is the smallest unit of transactional memory: it holds a single value (of any type)

The functions **`$`** and **`(setf $)`** read and write a TVAR value.

The macro **`(atomic &body body)`** executes Lisp forms inside a transaction.

TVARs are versioned using a global clock “GV1” – needed to guarantee read consistency

Examples and API (2/2)

It is usually more convenient to take advantage of STMX integration with closer-mop

```
(transactional
  (defclass bank-account ()
    ((balance :type rational :initform 0
              :accessor account-balance))))

(defun bank-transfer (from-acct to-acct amount)
  (atomic
    (when (< (account-balance from-acct) amount)
      (error "not enough funds for transfer"))

    (decf (account-balance from-acct) amount)
    (incf (account-balance to-acct) amount)))
```

The macro **`(transactional (defclass ...))`** defines a transactional class: its instance slots are transparently wrapped by TVARs. **`(slot-value)`** and accessors work as expected: they read or write the value inside the TVAR

A macro **`(transactional-struct (defstruct ...))`** is currently under development

Main features (1/5)

STMX guarantees full A.C.I.~~D.~~ semantics inside (atomic …) forms:

- Atomicity: (atomic …) forms are committed if they complete normally,
  they are rolled back in case of non-local exit: signal a condition, (throw), (go), (return) …
  Effects of an (atomic …) form are invisible to other threads until it commits.
- Consistency: an (atomic …) form sees a consistent snapshot of transactional memory.
  If consistency cannot be guaranteed, STMX aborts and restarts the (atomic …) form.
- Isolation: inside an (atomic …) form, effects of transactions committed by other threads are not visible. They become visible only after the current (atomic …) form commits or rolls back.
- STMX transactions are NOT durable – but we are working on it[1]
- Composability: multiple transactions can be composed into a single, larger transaction:

```
(atomic
   (atomic ...)
   (atomic ...)
   ...)
```

[1] https://github.com/cosmos72/hyperluminal-db

Main features (2/5)

- Waiting for changes: the function **`(retry)`** aborts the current transaction, waits until another thread changes some of the TVARs read since the beginning of the transaction, then re-executes the transaction from scratch. Examples:

```
(defmethod put ((v tvar) value)
  (atomic
    (if ($ v)
        (retry)
        (setf ($ v) value))))

(defmethod take ((v tvar))
  (atomic
    (if ($ v)
        ($ v)
        (retry))))
```

- Nested, alternative transactions: `(atomic (orelse form1 form2 ...))`
  If `form1` calls `(retry)` or aborts spontaneously, `form2` is invoked and so on.
- Delayed execution: `(before-commit ...)` and `(after-commit ...)`

Main features (3/5)

Transactional version of popular data structures:

- TCONS and TLIST
- TVECTOR
- THASH-TABLE
- TMAP – sorted map, backed by red-black tree
- TSTACK and TFIFO
- TCHANNEL and TPORT – reliable multicast channel

Ready to use, they show how to write transactional structures and algorithms

Changes are usually small and mechanic:

- replace Lisp built-in structures with transactional counterparts
- replace (defclass …) with (transactional (defclass …))
- insert (atomic …) where appropriate

Main features (4/5)

Hardware transactional memory

- IBM Power ISA v.2.0.7 – currently **NOT** supported by STMX
- Intel TSX – supported by STMX on 64-bit SBCL, requires latest Intel Core i5/i7[2]
  - XBEGIN start a HW memory transaction; needs address of fallback routine
  - XEND commit
  - XABORT abort and jump to fallback routine
  - XTEST check whether a HW transaction is running

  All CPU memory accesses (MOV, PUSH, POP…) become transactional.

  L1 cache currently used as transactional buffer.

  Memory conflicts, context switches, syscalls … “may” abort the HW transaction.

  Never guaranteed to succeed, requires fallback routine.

  Very fast: ~20 nanoseconds initial overhead,
  memory accesses maintain native, non-transactional speed

[2]”Haswell” generation (June 2013) – except some models

Main features (5/5)

Hybrid transactional memory

(2013, March) A. Matveev and N. Shavit describe how to efficiently mix Intel TSX and SW transactional memory

STMX implements a three-level strategy (requires 64-bit SBCL)

1. HW transactions using Intel TSX
2. SW transactions, with commit implemented by a HW transaction
3. Fully SW transactions, disabling HW ones

Some details:

- Adaptive global clock (GV1 + GV5 = GV6)
- HW transactions use un-instrumented reads. Writes also set TVAR version.
- Fallback 2 allows to run HW and SW transactions concurrently.

Strengths & weaknesses (1/2)

- Correct
- Intuitive
- Powerful
- Elegant – can I say beautiful?
- Heavily optimized – not slow anymore

- Vulnerable to near-livelocks
- Requires legacy code changes
- I/O and other irreversible operations should be avoided

Misquote:
Every sufficiently complex lock-based algorithm contains a bug-ridden implementation of half transactional memory

Strengths & weaknesses (2/2)

Optimizations

- Transparent HW acceleration (requires 64-bit SBCL + Intel TSX)
- Specialized hash table with thread-local pools and sortless TVAR locking
- No consing in most cases
- Iteratively inserted type declarations and optimizations based on profiling and disassembly
- Fast compare-and-swap locks + memory barriers (requires SBCL)
- Optimizes away redundant TVAR writes during commit

Transactional I/O

- Intel TSX limitations can be worked around – result is HW accelerated transactional output on memory-mapped files and/or shared memory. Extremely useful for database-like workloads requiring persistence.

Performance (1/2)

Micro-benchmarks – Intel Core i7 4770, Linux, SBCL 1.1.5 (64-bit)

nanoseconds per operation

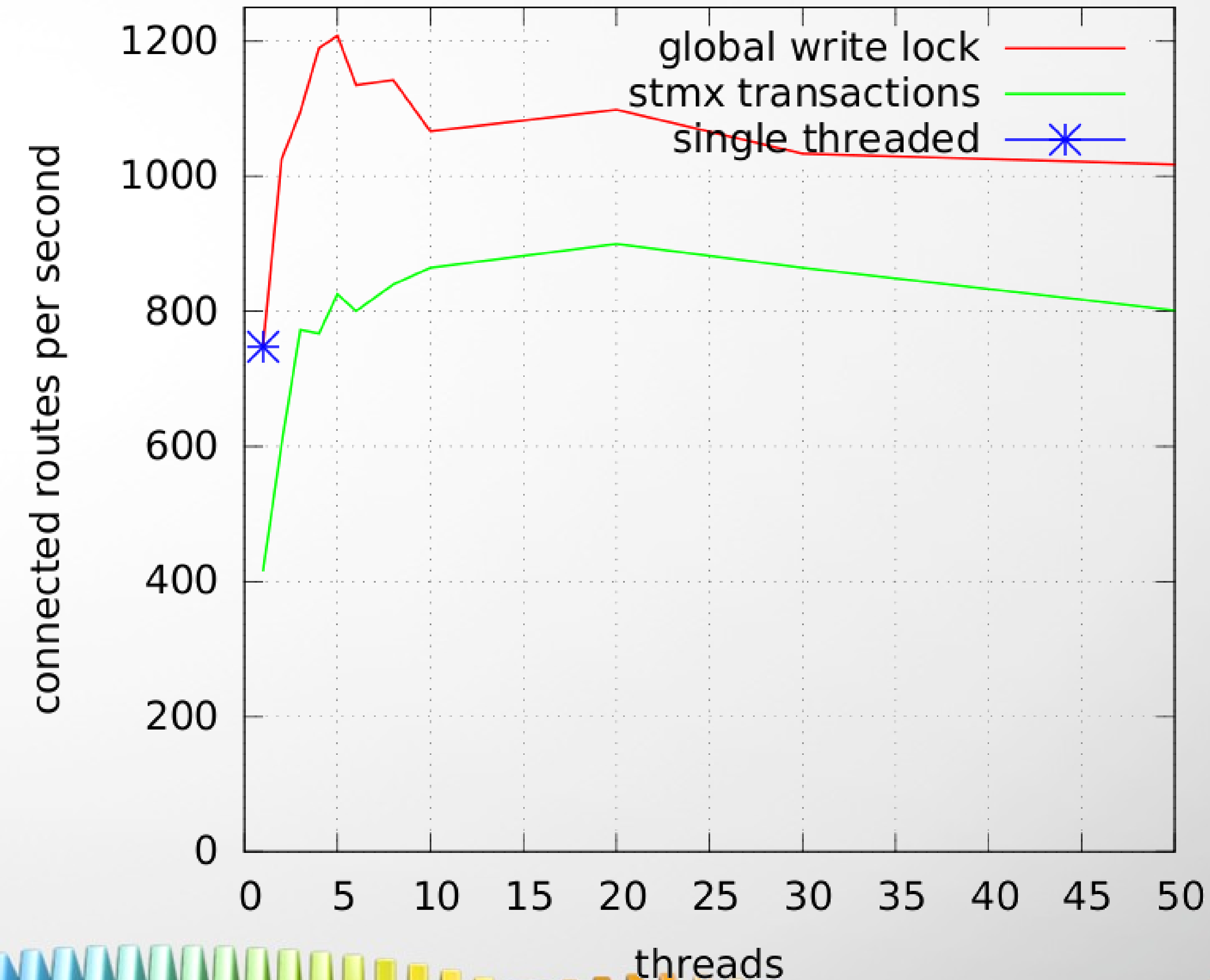
Name	Code	SW tx	Hybrid tx	No tx
read	`($ v)`	87	22	<1
write	`(setf ($ v) 1)`	113	27	<1
incf	`(incf ($ v))`	148	27	3
10 incf	`(dotimes (i 10) (incf ($ v)))`	272	59	19
100 incf	`(dotimes (i 100) (incf ($ v)))`	1399	409	193
1000 incf	`(dotimes (i 1000) (incf ($ v)))`	12676	3852	1939
map read	`(get-gmap tm 1)`	274	175	51
map update	`(incf (get-gmap tm 1))`	556	419	117
hash-table read	`(get-ghash th 1)`	303	215	74
hash-table update	`(incf (get-ghash th 1))`	674	525	168

Performance (2/2)

Lee-TM benchmark
Intel Core i7 4770
Debian GNU/Linux
SBCL 1.1.5 (64-bit)

Input: discrete grid, pairs of points to connect
(ex. a mainboard)

Output:
non-intersecting routes



Questions & answers

http://www.stmx.org/

massimiliano.ghilardi@gmail.com