High performance concurrency in Common Lisp - hybrid transactional memory with STMX

Massimiliano Ghilardi
TBS Group
AREA Science Park 99, Padriciano
Trieste, Italy
massimiliano.ghilardi@gmail.com

ABSTRACT

In this paper we present STMX, a high-performance Common Lisp implementation of transactional memory.

Transactional memory (TM) is a concurrency control mechanism aimed at making concurrent programming easier to write and understand. Instead of traditional lock-based code, a programmer can use atomic memory transactions, which can be composed together to make larger atomic memory transactions. A memory transaction gets committed if it returns normally, while it gets rolled back if it signals an error (and the error is propagated to the caller).

Additionally, memory transactions can safely run in parallel in different threads, are re-executed from the beginning in case of conflicts or if consistent reads cannot be guaranteed, and their effects are not visible from other threads until they commit.

Transactional memory gives freedom from deadlocks and race conditions, automatic roll-back on failure, and aims at resolving the tension between granularity and concurrency.

STMX is notable for the three aspects:

- It brings an actively maintained, highly optimized transactional memory library to Common Lisp, closing a gap open since 2006.
- It was developed, tested and optimized in very limited time - approximately 3 person months - confirming Lisp productivity for research and advanced programming.
- It is one of the first published implementations of hybrid transactional memory, supporting it since August 2013 - only two months after the first consumer CPU with hardware transactions hit the market.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures; F.1.2 [**Modes of Computation**]: Parallelism and concurency; D.2.13 [**Reusable Software**]: Reusable Libraries; D.2.11 [**Software Architectures**]: Patterns

General Terms

Algorithms, Theory

Keywords

Common Lisp, parallelism, concurrency, high-performance, transactions, memory

1. INTRODUCTION

There are two main reasons behind transactional memory.

The first is that in recent years all processors, from high-end servers, through consumer desktops and laptops, to tablets and smartphones, are increasingly becoming multi-core. After the Pentium D (2005), one of the first dual-core consumer CPU, only six years passed to see the 16-core AMD Opteron Interlagos (2011). Supercomputers and high-end servers are much more parallel than that, and even tablets and smartphones are often dual-core or quad-core. Concurrent programming has become mandatory to exploit the full power of multi-core CPUs.

The second reason is that concurrent programming, in its most general form, is a notoriously difficult problem [5, 6, 7, 11, 12]. Over the years, different paradigms have been proposed to simplify it, with various degrees of success: functional programming, message passing, futures, π-calculus, just to name a few.

Nowadays, the most commonly used is multi-threading with shared memory and locks (mutexes, semaphores, conditions ...). It is very efficient when used correctly and with fine-grained locks, as it is extremely low level and maps quite accurately the architecture and primitives found in modern multi-core processors. On the other hand, it is inherently fraught with perils: deadlocks, livelocks, starvation, priority inversion, non-composability, nondeterminism, and race conditions. The last two can be very difficult to diagnose, to reproduce, and to solve as they introduce non-deterministic behavior. To show a lock-based algorithm's correctness, for

example, one has to consider all the possible execution interleavings of different threads, which increases exponentially with the algorithm's length.

Transactional memory is an alternative synchronisation mechanism that solves all these issues (with one exception, as we will see). Advocates say it has clean, intuitive semantics and strong correctness guarantees, freeing programmers from worrying about low-level synchronization details. Skeptics highlight its disadvantages, most notably an historically poor performance - although greatly improved by recent hardware support (`Intel TSX` and `IBM Power ISA v.2.0.7`) - and that it does not solve livelocks, as it is prone to almost-livelocks in case of high contention.

STMX is a high-performance Common Lisp implementation of transactional memory. It is one of the first implementations supporting hybrid transactions, taking advantage of hardware transactions (`Intel TSX`) if available and using software-only transactions as a fallback.

2. HISTORY

Transactional memory is not a new idea: proposed as early as 1986 for Lisp [8], it borrows the concurrency approach successfully employed by databases and tries to bring it to general purpose programming. For almost ten years, it was hypothesized as a hardware-assisted mechanism. Since at that time no CPU supported the required instructions, it was mainly confined as a research topic.

The idea of software-only transactional memory, introduced by Nir Shavit and Dan Touitou in 1995 [11], fostered more research and opened the possibility of an actual implementation. Many researchers explored the idea further, and the first public implementation in Haskell dates back to 2005 [6].

Implementations in other languages followed soon: C/C++ (LibLTX, LibCMT, SwissTM, TinySTM), Java (JVSTM, Deuce), C# (NSTM, MikroKosmos), OCaml (coThreads), Python (Durus) and many others. Transactional memory is even finding its way in C/C++ compilers as GNU gcc and Intel icc.

Common Lisp had CL-STM, written in 2006 Google Summer of Code[1]. Unfortunately it immediately went unmaintained as its author moved to other topics. The same year Dave Dice, Ori Shalev and Nir Shavit [4] solved a fundamental problem: guaranteeing memory read consistency.

Despite its many advantages, software transactional memory still had a major disadvantage: poor performance. In 2012, both Intel[2] and IBM[3] announced support for hardware transactional memory in their upcoming lines of products. The IBM products are enterprise commercial servers implementing "Power ISA v.2.0.7": Blue Gene/Q[4] and zEnterprise EC12, both dated 2012, and Power8[5] released in May 2013. Intel products are the "Haswell" generation of Core i5 and Core i7, released in June 2013 - the first consumer CPUs offering hardware transactional memory under the name "Intel TSX".

Hardware support greatly improves transactional memory performance, but it is never guaranteed to succeed and needs a fallback path in case of failure.

Hybrid transactional memory is the most recent reinvention. Hypothesized and researched several times in the past, it was until now speculative due to lack of hardware support. In March 2013, Alexander Matveev and Nir Shavit [10] showed how to actually implement a hybrid solution that successfully combined the performance of Intel TSX hardware transactions with the guarantees of a software transaction fallback, removing the last technical barrier to adoption.

STMX started in March 2013 as a rewrite of CL-STM, and a first software-only version was released in May 2013. It was extended to support hardware transactions in July 2013, then hybrid transactions in August 2013, making it one of the first published implementations of hybrid transactional memory.

3. MAIN FEATURES

STMX offers the following functionalities, common to most software transactional memory implementations:

- atomic blocks: each `(atomic ...)` block runs code in a memory transaction. It gets committed if returns normally, while it gets rolled back if it signals an error (and the error is propagated to the caller). For people familiar with ContextL[6], transactions could be defined as layers, an atomic block could be a scoped layer activation, and transactional memory is analogous to a layered class: its behavior differs inside and outside atomic blocks.
- atomicity: the effects of a transaction are either fully visible or fully invisible to other threads. Partial effects are never visible, and rollback removes any trace of the executed operations.
- consistency: inside a transaction data being read is guaranteed to be in consistent state, i.e. all the invariants that an application guarantees at commit time are preserved, and they can be temporarily invalidated only by a thread's own writes. Other simultaneous transactions cannot alter them.
- isolation: inside a transaction, effects of transactions committed by other threads are not visible. They become visible only after the current transaction commits or rolls back. In database terms this is the highest possible isolation level, named "serializable".
- automatic re-execution upon conflict: if STMX detects a conflict between two transactions, it aborts and restarts at least one of them.

[1] http://common-lisp.net/project/cl-stm/
[2] http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell
[3] https://www.power.org/documentation/power-isa-transactional-memory/
[4] http://www.kurzweilai.net/ibm-announces-20-petaflops-supercomputer
[5] https://www.power.org/documentation/power-isa-version-2-07/
[6] http://common-lisp.net/project/closer/contextl.html

- read consistency: if STMX cannot guarantee that a transaction sees a consistent view of the transactional data, the whole atomic block is aborted and restarted from scratch **before** it can see the inconsistency.
- composability: multiple atomic blocks can be composed in a single, larger transaction simply by executing them from inside another atomic block.

STMX also implements the following advanced features:

- waiting for changes: if the code inside an atomic block wants to wait for changes on transactional data, it just needs to invoke `(retry)`. This will abort the transaction, sleep until another thread changes some of the transactional data read since the beginning of the atomic block, and finally re-execute it from scratch.
- nested, alternative transactions: an atomic block can execute two or more Lisp forms as alternatives in separate, nested transactions with `(atomic (orelse form1 form2 ...))`. If the first one calls `(retry)` or aborts due to a conflict or an inconsistent read, the second one will be executed and so on, until one nested transaction either commits (returns normally) or rollbacks (signals an error or condition).
- deferred execution: an atomic block can register arbitrary forms to be executed later, either immediately before or immediately after it commits.
- hybrid transactional memory: when running on 64-bit Steel Bank Common Lisp (SBCL) on a CPU with Intel TSX instructions, STMX automatically takes advantage of hardware memory transactions, while falling back on software ones in case of excessive failures. The implementation is carefully tuned and allows software and hardware transactions to run simultaneously in different threads with a constant (and very low) overhead on both transaction types. STMX currently does **not** support IBM Power ISA hardware transactions.

4. DESIGN AND IMPLEMENTATION

STMX brings efficient transactional memory to Common Lisp thanks to several design choices and extensive optimization. Design and implementation follows three research papers [6] [4] [10]. All of them contain pseudo-code for the proposed algorithms, and also include several correctness demonstrations.

Keeping the dynamically-typed spirit of Lisp, STMX is value-based: the smallest unit of transactional memory is a single cell, named `TVAR`. It behaves similarly to a variable, as it can hold a single value of any type supported by the hosting Lisp: numbers, characters, symbols, arrays, lists, functions, closures, structures, objects... A quick example:

```
(quicklisp:quickload :stmx)
(use-package :stmx)
(defvar *v* (tvar 42))
(print ($ *v*))      ;; prints 42
(atomic
  (if (oddp ($ *v*))
    (incf ($ *v*))
    (decf ($ *v*)))) ;; *v* now contains 41
```

While `TVAR`s can be used directly, it is usually more convenient to take advantage of STMX integration with `closer-mop`, a Metaobject Protocol library. This lets programmers use CLOS objects normally, while internally wrapping each slot value inside a `TVAR` to make it transactional. Thus it can also be stated that STMX is slot-based, i.e. it implements transactional memory at the granularity of a single slot inside a CLOS object. This approach introduces some space overhead, as each `TVAR` contains several other informations in addition to the value. On the other hand, it has the advantage that conflicts are detected at the granularity of a single slot: two transactions accessing different slots of the same object do not interfere with each other and can proceed in parallel. A quick CLOS-based example:

```
(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)))
```

Object-based and stripe-based implementations exist too. In the former, the smallest unit of transactional memory is a single object. In the latter, the smallest unit is instead a "stripe": a (possibly non-contiguous) region of the memory address space - suitable for languages as C and C++ where pointers are first-class constructs. Both have lower overhead than slot-based transactional memory, at the price of spurious conflicts if two transactions access different slots in the same object or different addresses in the same stripe.

4.1 Read and write implementation

The fundamental operations on a `TVAR` are reading and writing its value. During a transaction, `TVAR` contents are never modified: that's performed at the end of the transaction by the commit phase. This provides the base for the atomicity and isolation guarantees. So writing into a `TVAR` must store the value somewhere else. The classic solution is to have a transaction write log: a thread-local hash table recording all writes. The hash table keys are the `TVAR`s, and the hash table values are the values to write into them.

Reading a `TVAR` is slightly more complex. Dave Dice, Ori Shalev and Nir Shavit showed in [4] how to guarantee that a transaction always sees a consistent snapshot of the `TVAR`s contents. Their solution requires versioning each `TVAR`, and also adding a "read version" to each transaction. Such version numbers are produced from a global clock. One bit of the `TVAR` version is reserved as a lock.

To actually read a `TVAR`, it is first searched in the transaction write log and, if found, the corresponding value is returned. This provides read-after-write consistency. Otherwise, the `TVAR` contents is read without acquiring any lock - first retrieving its full version (including the lock bit), then issuing a memory read barrier, retrieving its value, issuing another memory read barrier, and finally retrieving again its full version. The order is intentional, and the memory read barriers are fundamental to ensure read consistency, as they couple

with the memory write barriers used by the commit phase when actually writing TVAR contents. Then, the two TVAR versions read, including the lock bits, are compared with each other: if they differ, or if one or both lock bits are set, the transaction aborts and restarts from scratch in order to guarantee read consistency and isolation. Then, the TVAR version just read is compared with the transaction read version: if the former is larger, it means the TVAR was modified after the transaction started. In such case, the transaction aborts and restarts too. Finally, if the TVAR version is smaller than or equal to the transaction read version, the TVAR and the retrieved value are stored in the transaction read log: a thread-local hash table recording all the reads, needed by the commit phase.

4.2 Commit and abort implementation

Aborting a transaction is trivial: just discard some thread-local data - the write log, the read log and the read version.

Committing a STMX transaction works as described in [4]:

First, it acquires locks for all TVARs in the write log. Using non-blocking locks is essential to avoid deadlocks, and if some locks cannot be acquired, the whole transaction aborts and restarts from scratch. STMX uses compare-and-swap CPU instructions on the TVAR version to implement this operation (the version includes the lock bit).

Second, it checks that all TVARs in the read log are not locked by some other transaction trying to commit simultaneously, and that their version is still less or equal to the transaction read version. This guarantees the complete isolation between transactions - in database terms, transactions are "serializable". If this check fails, the whole transaction aborts and restarts from scratch.

Now the commit is guaranteed to succeed. It increases the global clock by one with an atomic-add CPU instruction, and uses the new value as the transaction write version. It then loops on all TVARs in the write log, setting their value to match what is stored in the write log, then issuing a memory write barrier, finally setting their version to the transaction write version. This last write also sets the lock bit to zero, and is used to release the previously-acquired lock.

Finally, the commit phase loops one last time on the TVARs that have been just updated. The semaphore and condition inside each TVAR will be used to notify any transaction that invoked `(retry)` and is waiting for TVARs contents to change.

4.3 Novel optimizations

In addition to the algorithm described above, STMX uses two novel optimizations to increase concurrency, and a third to reduce the overhead:

If a transaction tries to write back in a TVAR the same value read from it, the commit phase will recognize it before locking the TVAR by observing that the TVAR is associated to the same value both in the write log and in the read log. In such case, the TVAR write is degraded to a TVAR read and no lock is acquired, improving concurrency.

When actually writing value and version to a locked TVAR, the commit phase checks if it's trying to write the same value already present in the TVAR. In such case, the value and version are not updated. Keeping the old TVAR version means other transaction will not abort due to a too-large version number, improving concurrency again.

To minimize the probability of near-livelock situations, where one or more transactions repeatedly abort due to conflicts with other ones, the commit phase should acquire TVAR locks in a stable order, i.e. different transactions trying to lock the same TVARs A and B should agree whether to first lock A or B. The most general solution is to sort the TVARs before locking them, for example ordering by their address or by some serial number stored inside them. Unluckily, sorting is relatively expensive - its complexity is $O(N \log N)$ - while all other operations performed by STMX during commit are at most linear, i.e. $O(N)$ in the number of TVARs. To avoid this overhead, STMX omits the sort and replaces it with a faster alternative, at the price of increasing vulnerability to near-livelocks (crude tests performed by the author seem to show that near-livelocks remain a problem only under extreme contention). The employed solution is to store a serial number inside each TVAR and use it for the hashing algorithm used by the read log and write log hash tables. In this way, iterating on different write logs produces relatively stable answers to the question "which TVAR should be locked first, A or B ?" - especially if the hash tables have the same capacity - maintaining a low probability for near-livelock situations, without any overhead.

4.4 Automatic feature detection

ANSI Common Lisp does not offer direct access to low-level CPU instructions used by STMX, as memory barriers, compare-and-swap, and atomic-add. Among the free Lisp compilers, only Steel Bank Common Lisp (SBCL) exposes them to user programs. STMX detects the available CPU instructions at compile time, while falling back on slower, more standard features to replace any relevant CPU instruction not exposed by the host Lisp.

If memory barriers or compare-and-swap are not available, STMX inserts a `bordeaux-threads:lock` in each TVAR and uses it to lock the TVAR. The operation "check that all TVARs in the read log are not locked by some other transaction" in the commit phase requires getting the owner of a lock, or at least retrieving whether a lock is locked or not and, in case, whether the owner is the current thread. `Bordeaux-threads` does not expose such operation, but the underlying implementation often does: Clozure Common Lisp has `(ccl::%%lock-owner)`, CMUCL has `(mp::lock-process)` and Armed Bear Common Lisp allows to directly call the Java methods `ReentrantLock.isLocked()` and `ReentrantLock.isHeldByCurrentThread()` to obtain the same information. STMX detects and uses the appropriate mechanism automatically.

Similarly, the global counter uses atomic-add CPU instructions if available, otherwise it falls back on a normal add protected by a `bordeaux-threads:lock`.

4.5 Hybrid transactions

In June and July 2013 we extended STMX to support the Intel TSX CPU instructions[7], that provide hardware memory transactions.

Intel TSX actually comprise two sets of CPU instructions: HLE and RTM. Hardware Lock Elision (HLE) is designed as a compatible extension for existing code that already uses atomic compare-and-swap as locking primitive. Restricted Transactional Memory (RTM) is a new set of CPU instructions that implement hardware memory transactions directly at the CPU level:

- `XBEGIN` starts a hardware memory transaction. After this instruction and until the transaction either commits or aborts, all memory accesses are guaranteed to be transactional. The programmer must supply to `XBEGIN` the address of a fallback routine, that will be executed if the transaction aborts for any reason.
- `XEND` commits a transaction.
- `XABORT` immediately aborts a transaction and jumps to the fallback routine passed to `XBEGIN`. Note that hardware transactions can also abort spontaneosly for many different reasons: they are executed with a "best effort" policy, and while following Intel guidelines and recommendations usually results in very high success rates ($> 99.99\%$), they are **never** guaranteed to succeed and they have limits on the amount of memory that can be read and written within a transaction. Also, many operations usually cause them to abort immediately, including: conflicting memory accesses from other CPU cores, system calls, context switches, `CPUID` and `HLT` CPU instructions, etc.
- `XTEST` checks whether a transaction is in progress.

Exposing the `XBEGIN`, `XEND`, `XABORT`, and `XTEST` CPU instructions as Lisp functions and macros is non-portable but usually fairly straightforward, and we added them on 64-bit SBCL.

The real difficulty is making them compatible with software transactions: the software-based commit uses locks to prevent other threads from accessing the `TVAR`s it wants to modify, so if a hardware transaction reads those `TVAR`s at the wrong time, it would see a half-performed commit: isolation and consistency would be violated. A naive solution is to instrument hardware transactions to check whether `TVAR`s are locked or not when reading or writing them. It imposes such a large overhead that cancels the performance advantage. Another attempt is to use hardware transactions **only** to implement the commit phase of software transactions. Tests on STMX show that the performance gain is limited - about 5%.

The key was discovered by Alexander Matveev and Nir Shavit [10] in 2013: use a hardware transaction to implement the commit phase of software transactions, not to improve performance, but to make them really atomic at the CPU level. Then the software commit phase does not need anymore to lock the `TVAR`s: atomicity is now guaranteed by the hardware transaction. With such guarantees, hardware transactions can directly read and write `TVAR`s without any instrumentation - no risk of seeing a partial commit - and their overhead is now almost zero. The only remaining overhead is the need to write both `TVAR`s value and version, not just the value.

There were two problems left.

The first is: as stated above, hardware transaction are never guaranteed to succeed. They may abort if hardware limits are exceeded or if the thread attempts to execute a CPU instruction not supported inside a hardware transaction. For example, memory allocation in SBCL almost always causes hardware transactions to abort - this is an area that could be significantly improved by creating thread-local memory pools in the host Lisp.

Alexander Matveev and Nir Shavit [10] provided a sophisticated solution to this problem, with multiple levels of fallbacks: software transactions using a smaller hardware transaction during commit, software-only transactions, and intrumented hardware transactions.

We added hybrid transactions to STMX using a simplified mechanism: if the commit phase of software transactions fails (remember, it is now implemented by a hardware transaction), it increments a global counter that prevents **all** hardware transactions from running, then performs an old-style software-only commit, finally decrements the global counter to re-enable hardware transactions.

The second problem is: the commit phase of a transaction - either hardware or software - must atomically increment the global clock. For hardware transactions, this means modifying a highly contended location, causing a conflict (and an abort) as soon as two or more threads modify it from overlapping transactions.

A partial solution is described in [4, 10]: use a different global clock algorithm, named GV5, that increases the global clock **only** after an abort. It works by writing the global clock +1 into `TVAR`s during commit without increasing it, and has the side effect of causing approximately 50% of software transactions to abort.

The full solution, as described in [10, 2] is to use an adaptive global clock, named GV6, that can switch between the normal and the GV5 algorithm depending on the success and abort rates of software and hardware transactions. STMX stores these rates in thread-local variables and combines them only sporadically (every some hundred transactions) to avoid creating other highly contended global data.

We released STMX version 1.9.0 in August 2013 - the first implementation to support hybrid transactional memory in Common Lisp, and one of the first implementations to do so in any language.

4.6 Data structures

STMX includes transactional versions of basic data structures: `TCONS` and `TLIST` for cons cells and lists, `TVECTOR` for vectors, `THASH-TABLE` for hash tables, and `TMAP` for sorted maps (it is backed by a red-black tree).

[7] http://www.intel.com/software/tsx

`THASH-TABLE` and `TMAP` also have non-transactional counterparts: `GHASH-TABLE` and `GMAP`. They are provided both for completeness and as base classes for the corresponding transactional version. This makes them practical examples showing how to convert a normal data structure into a transactional one.

In many cases the conversion is trivial: change `(defclass foo ...)` definition to `(transactional (defclass foo ...))`[8]. When needed, it is also possible to decide on a slot-by-slot basis whether they should become transactional or not. This can significantly reduce the overhead in certain cases, as shown in [3]. For slots that contain non-immutable values (i.e. objects, arrays, etc.), such inner objects must also be replaced by their transactional counterparts if their contents can be modified concurrently. STMX also includes some transactional-only data structures: a first-in last-out buffer `TSTACK`, a first-in first-out buffer `TFIFO`, a reliable multicast channel `TCHANNEL`, and its reader side `TPORT`.

5. BENEFITS

The conceptual simplicity, intuitivity and correctness guarantees of transactional memory are not its only advantages.

A more subtle, important advantage is the fact that converting a data structure into its transactional version is almost completely mechanical: with STMX, it is sufficient to replace a CLOS `(defclass foo ...)` with `(transactional (defclass foo ...))`, with object-valued slots needing the same replacement.

This means that arbitrarily complex algorithms and data structures can be easily converted, without the need to analyze them in deep detail, as it's usually the case for the conversion to fine-grained lock-based concurrency. Such ability makes transactional memory best suited for exactly those algorithms and data structures that are difficult to parallelize with other paradigms: large, complex, heterogeneous data structures that can be modified concurrently by complex algorithms and do not offer easy divisions in subsets.

Clearly, analyzing the algorithms and data structures can provide benefits, in the form of insights about the subset of the data that really needs to become transactional, and which parts of the algorithms should be executed inside transactions.

A practical example is Lee's circuit routing algorithm, also used as transactional memory benchmark [1]: the algorithm takes as input a large, discrete grid and pairs of points to connect (e.g. an integrated circuit) and produces non-intersecting routes between them. Designing a lock-based concurrent version of Lee's algorithm requires decisions and trade-offs, as one has to choose at least the locking approach and the locks granularity. The transactional version is straightforward: the circuit grid becomes transactional. A deeper analysis also reveals that only a small part of the algorithm, namely backtracking, needs to be executed inside a transaction.

[8] an analogous macro for structure-objects defined with `(defstruct foo ...)` is currently under development.

6. DISADVANTAGES

Transactional memory in general has some drawbacks, and STMX inherits them.

One is easy to guess: since transactions can abort and restart at any time, they can be executed more times than expected, or they can be executed when not expected, so performing any irreversible operation inside a transaction is problematic. A typical example is input/output: a transaction should not perform it, rather it should queue the I/O operations in a transactional buffer and execute them later, from outside any transaction. Hardware transactions - at least Intel TSX - do not support any irreversible operation and will abort immediately if you try to perform input/output from them.

Another drawback is support for legacy code: to take advantage of transactions, code must use transactional cells, i.e. `TVAR`s. This requires modifications to the source code, which can be performed automatically only by transaction-aware compilers or by instrumentation libraries as Java Deuce [9]. STMX is implemented as a normal library, not as a compiler plugin, so it requires programmers to adapt their code. The modifications are quite simple and mechanic, and STMX includes transactional versions of some popular data structures, both as ready-to-use solutions and as examples and tutorials showing how to modify a data structure to make it transactional.

The last disadvantage is proneness to almost-livelocks under high contention. This is common to all implementations that use non-blocking mutexes (STMX uses compare-and-swap ones) as synchronization primitives, as they either succeed or fail immediately, and they are not able nor supposed to sleep until the mutex can be acquired: doing so would cause deadlocks.

7. TRANSACTIONAL I/O

We present a novel result, showing that in a very specific case it is possible to perform I/O from a hardware transaction implemented by Intel TSX, working around the current Intel hardware limitations. The result is transactional output, i.e. the output is performed if and only if the hardware transaction commits.

Intel reference documentation[9] states that attempting to execute I/O from an Intel TSX transactions **may** cause it to abort immediately, and that the exact behavior is implementation-dependent. On the hardware tested by the author (Intel Core i7 4770) this is indeed the case: syscalls, context switches, I/O to hardware ports, and the other operations that "**may** abort transactions", actually abort them. The technique described below works around this limitation.

Hardware transactions are guaranteed to support only manipulation of CPU registers and memory. Anyway, the content and meaning of the memory is irrelevant for Intel TSX. It is thus possible to write to memory-mapped files or shared memory, as long as doing so does not immediately trigger a context switch or a page fault.

[9] http://download-software.intel.com/sites/default/files/319433-014.pdf - section 8.3.8.1, pages 391-392

Thus, if some pages of memory mapped file are already dirty - for example because we write into them from outside any transaction - it is possible to continue writing into them from hardware transactions. After some time, the kernel will spontaneously perform a context switch and write back the pages to disk. Since hardware transactions are atomic at the CPU level and they currently abort upon a context switch, the kernel will observe that some of them have committed and altered the pages, while some others have aborted and their effects are completely rolled back. The memory pages, altered only by the committed transactions, will be written to disk by the kernel, thus implementing transactional I/O.

Author's initial tests show that it is possible to reach very high percentages of successful hardware transactions - more than 99% - writing to memory mapped files, provided the transactions are short and there is code to dirty again the pages if the hardware transactions fail.

This is a workaround - maybe even a hack - yet it is extremely useful to implement database-like workloads, where transactions must also be persistent, and shared memory inter-process communication. The author is currently using this technique to implement Hyperluminal-DB[10], a transactional and persistent object store, on top of STMX.

8. PERFORMANCE

This paragraph contains benchmark results obtained on an Intel Core i7 4770, running 64-bit versions of Linux/Debian jessie, SBCL 1.1.15 and the latest STMX. **Disclaimer**: results on different systems **will** vary. Speed differences up to **100 times and more** have been observed, depending on the Lisp compiler and the support for features used by STMX. System setup: execute the forms

```
(declaim (optimize (compilation-speed 0) (space 0)
                   (debug 0) (safety 0) (speed 3)))
(ql:quickload "stmx")
(ql:quickload "stmx.test")
(fiveam:run! 'stmx.test:suite)
```

before loading any other Lisp library, to set optimization strategy, load STMX and its dependencies, and run the test suite once to warm up the system.

8.1 Micro-benchmarks

We then created some transactional objects: a `TVAR` v, a `TMAP` tm, a `THASH-TABLE` th and fill them - full details are described in STMX source code[11]. Note that `TMAP` and `THASH-TABLE` are CLOS objects, making the implementation short and (usually) clear but not heavily optimized for speed. Rewriting them as structure-objects would definitely improve their performance. Finally, `$` is the function to read and write `TVAR` contents.

To record the execution time, we repeated each benchmark one million times in a loop and divided the resulting time by the number of iterations.

In Table 1, we report three times for each micro-benchmark: the first for software-only transactions, the second for hybrid transactions, the third for non-transactional execution with non-transactional data structures.

Table 1: micro-benchmarks time, in nanoseconds

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

Some remarks and deductions on the micro-benchmarks results: STMX software-only transactions have an initial overhead of ~ 130 nanoseconds, and hybrid transactions reduce the overhead to ~ 25 nanoseconds.

In software-only transactions, reading and writing `TVAR`s, i.e. transactional memory, is 6–7 times slower than reading and writing normal memory. Hardware transactions improve the situation: inside them, transactional memory is twice as slow as normal memory. In this respect, it is worth noting that STMX can be further optimized, since in pure hardware transactions (which do not use `TVAR`s nor the function `$`) reading and writing memory has practically the same speed as normal memory access outside transactions.

The results on CLOS sorted maps and hash tables show that they are relatively slow, and the transactional version even more so. To have a more detailed picture, non-CLOS implementations of sorted maps and hash tables would be needed for comparison.

8.2 Lee-TM

Finding or designing a good synthetic benchmark for transactional memory is not easy. Lee's circuit routing algorithm, in the proposers' opinion [1], is a more realistic benchmark than classic ones (red-black trees and other micro-benchmarks, STMBench7 ...). It takes as input a large, discrete grid and pairs of points to connect (e.g. an integrated circuit) and produces non-intersecting routes between them. Proposed and used as benchmark for many transactional memory implementations (TL2, TinySTM, RSTM, SwissTM ...), it features longer transactions and non-trivial data contention.

After porting Lee-TM to STMX[12], we realized that it spends about 99.5% of the CPU time `outside` transactions due

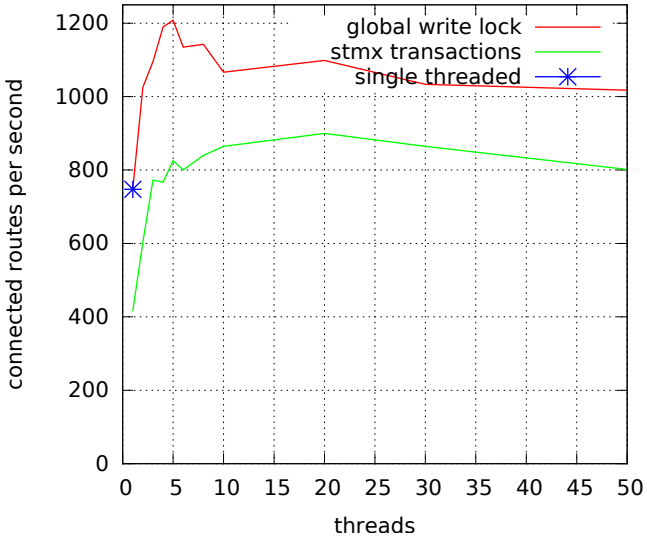
[10] https://github.com/cosmos72/hyperluminal-db
[11] http://github.com/cosmos72/stmx
[12] https://github.com/cosmos72/lee-stmx

to the (intentionally naive) grid exploration algorithm, and 0.5% in the backtracking algorithm (executed inside a transaction). It is thus not really representative of the strength and weaknesses of transactional memory. Lacking a better alternative we present it nevertheless, after some optimizations (we replaced Lee's algorithm with faster Hadlock's one) that reduce the CPU time spent outside transactions to 92–94%. The effect is that Lee-TM accurately shows the overhead of reading transactional memory from **outside** transactions, but is not very sensitive to transactional behavior.

In Table 2, we compare the transactional implementation of Lee-TM with a single-thread version and with a simple lock-based version that uses one global write lock. The results show that transactional memory slows down Lee's algorithm (actually, Hadlock's algorithm) by approximately 20% without altering its scalability.

The global write lock is a particularly good choice for this benchmark due to the very low time spent holding it (6–8%), and because the algorithm can tolerate lock-free reads from the shared grid. Yet, the overhead of the much more general transactional memory approach is contained. More balanced or more complex algorithms would highlight the poor scalability of trying to parallelize using a simple global write lock.

Table 2: Lee-TM, mainboard circuit



9. CONCLUSIONS

Transactional memory has a long history. Mostly confined to a research topic for the first two decades, it is now finding its way into high-quality implementations at an accelerating pace. The long-sought arrival of hardware support may well be the last piece needed for wide diffusion as a concurrent programming paradigm.

STMX brings state of the art, high-performance transactional memory to Common Lisp. It is one of the first publicly available implementations to support hybrid transactions, integrating "Intel TSX" CPU instructions and software transactions with minimal overhead on both.

STMX is freely available: licensed under the "Lisp Lesser General Public Licence" (LLGPL), it can be installed with Quicklisp `(ql:quickload "stmx")` or downloaded from `http://stmx.org/`

10. REFERENCES

[1] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-TM: A non-trivial benchmark suite for transactional memory. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '08, pages 196–207, 2008.

[2] H. Avni. A transactional consistency clock defined and optimized. Master's thesis, Tel-Aviv University, 2009. http://mcg.cs.tau.ac.il/papers/hillel-avni-msc.pdf.

[3] F. M. Carvalho and J. Cachopo. STM with transparent API considered harmful. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'11, pages 326–337, Berlin, Heidelberg, 2011.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC'06 Proceedings of the 20th international conference on Distributed Computing*, pages 194–208. Sun Microsystems Laboratories, Burlington, MA, 2006.

[5] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Publishing Company, Boston, Massachusetts, 2006.

[6] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05 Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. Microsoft Research, Cambridge, UK, 2005.

[7] G. Karam and R. Buhr. Starvation and critical race analyzers for Ada. *IEEE Transactions on Software Engineering*, 16(8):829–843, August 1990.

[8] T. Knight. An architecture for mostly functional languages. In *LFP '86 Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112. Symbolics, Inc., and The M.I.T. Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1986.

[9] G. Korland, N. Shavit, and P. Felber. Noninvasive Java concurrency with Deuce STM. In *Proceedings of the Israeli Experimental Systems Conference*, SYSTOR'09. Tel-Aviv University, Israel and University of Neuchâtel, Switzerland, 2009.

[10] A. Matveev and N. Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *SPAA '13 Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 11–22. MIT, Boston, MA and Tel-Aviv University, Israel, 2013.

[11] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95 Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. MIT and Tel-Aviv University, 1995.

[12] D. A. Wheeler. Secure programming for Linux and Unix HOWTO. http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/avoid-race.html, 1991.