Performance Evaluation of Software Transactional Memory Implementations

Dániel Urbán
Bell Labs, Nokia, Network Systems and Security Research
Budapest, Hungary
email: daniel.urban@nokia-bell-labs.com

Péter Fazekas
Bell Labs, Nokia, Network Systems and Security Research
Budapest, Hungary
email: peter.fazekas@nokia-bell-labs.com

***Abstract*—Software Transactional Memory (STM) was introduced as a promising technology to handle memory conflicts in parallel computing. In this paper, a performance comparison of various STM engine implementations is presented. The well-known Lee's algorithm was used for benchmarking ten different Scala based STM API variants, and one written in Kotlin. Results compare how these implementations scale in terms of the number of processor cores available and how they perform in terms of running time, compared to each other and a single threaded baseline implementation.**

Keywords – Software Transactional Memory; parallel computing; concurrent programming; functional APIs; performance measurement.

I. INTRODUCTION

Parallel computing has a decades long history from emerging concepts to practical applications already in early mainframe systems. Nowadays, concurrent programming is applied in almost all domains from end user applications, enterprise software to exascale computing workloads.

Concurrent threads using shared resources (such as memory) have been identified early as a critical aspect. Straightforward solution is to prevent threads using the resource at the same time, therefore plethora of solutions and approaches have been designed and implemented in various architectures, such as using critical sections in the code, atomic operations, locks, semaphores, mutexes, etc.

Most aforementioned approaches are using some form of locking based solution (preventing threads to execute while some conditions apply), which brings well known shortcomings such as potential deadlocks, livelocks, convoying, priority inversion, starvation, etc.

To overcome these issues, several solutions were proposed and implemented, which are basically building on special representation of data or programming phenomena to avoid reading/writing shared information at the same time. The main directions are using lock-free or wait-free data structures, such as queues, ring buffers or stacks among others; or to basically prevent using shared context data and apply messaging among threads instead, such as actor model, or message passing channels. Furthermore, several approaches are targeting the complete avoidance of using shared mutable data, hence eliminating the root of the problem, such as data partitioning, thread-local storage or immutability in functional programming.

Transactional memory was introduced in the early 90's [1] to overcome shared memory challenges in concurrent programming. This approach is motivated by how transactions work in database systems. Basically, transactions are defined as serializable atomic instructions, that read and ultimately tentatively write shared memory spaces. Then, a *validate* operation is needed to ensure that there are no conflicts, that is, the memory content read for the computations and to be written as result is consistent. If validation is successful, the thread tries *committing* the changes. If the validation fails, the transaction aborts and retries. Commit is successful if no other transactions have modified the process's read set and no other transaction has read the write set, i.e., contention has not occurred since the last validation. When the commit is successful, the changes are made visible to other processes, otherwise the transaction *aborts* and tentative changes are reverted. This transactional model for memory operations was introduced as a low-level Application Programming Interface (API) in [2], so that the transactional memory is implemented in software (Software Transactional Memory, STM). Since then, numerous implementations have appeared, which provide these transactional functionalities over their APIs. These differ in various basic algorithms, data structures and optimizations provided; in Sections III and IV, we detail the ones relevant for our work.

The rest of this paper is organized as follows. Section II introduces the basic algorithm used for evaluating various STM implementations' performance and some related work. Section III summarizes various implementations evaluated with this work. Section IV addresses some important details of the implementations behind our analysis and the hardware and software environment used. Section V shows and analyses numerical results.

II. STM PERFORMANCE RELATED WORK

The main contribution of this paper is to provide a comparison on the performance of various STM implementations, focusing on the execution time of certain multi-threaded computation tasks.

To assess STM performance, one may select proper multi-threaded applications, for example, the authors of [3] list an excessive number of those. Their focus is on evaluating how the applications themselves behave with STM, in terms of size of read/write sets, transaction lengths statistics and depth of nested transactions, but the emphasis is not on comparing different STM engine implementations. Another suggestion is described in [4] as STMbench7, which is a synthetic benchmark defining a multitude of operations on a shared data structure.

However, we wanted to use a computing problem that has practical significance and enables a comparable

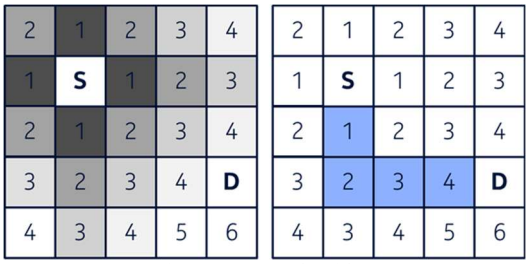

Figure 1. Lee's algorithm

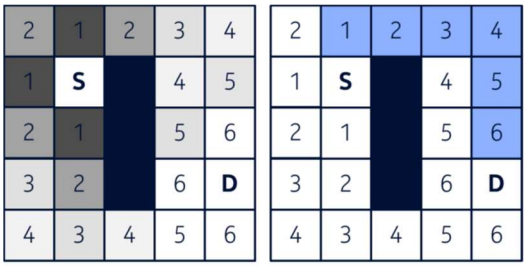

Figure 2. Lee's algorithm with occupied cells

benchmarking between various STM engine implementations; thus, the problem should be well parallelizable, the effect of concurrency should be significant, and the level of concurrency should be controllable via setting the inputs to the problem.

Therefore, as in [5]-[7], we selected the well-known circuit-board routing problem and used Lee's algorithm [8] to solve it. Circuit routing has practical significance when designing connections among electronic components on a surface, where crossing of connections (routes) is forbidden or has significant extra cost. In its simplest form, the surface is represented as a two-dimensional grid of square cells, representing potential insertion points of components and potential placeholders for connections.

Lee's algorithm has a number of source-destination pair cells (endpoints needing connections) as input. For a given source-destination pair, the algorithm starts with an expansion phase. This basically starts a "wave" from the source, searching all neighboring (along the edges of the square) cells and enumerating them with their distance from the source. This breadth-search continues from every neighboring cell to the neighbors of those (which are second neighbors to the source), until the search reaches the destination or the edge of the surface. In general, any cell might be occupied by an already existing route; these cells are not enumerated and not taken into account in the next phase of the algorithm.

The second phase is the backtracking, when from the destination to the source a list of cells is found, their enumeration should be in decreasing order. As there are multiple such routes, the particular implementation should rank those and select the optimal one. Typically, the shortest route, or the route with the least turns, or routes that are closer to blocked cells, etc. could be selected. This final phase of the algorithm is referred to as laying the route. As mentioned above, if a cell is already occupied by a route, it is not considered in the expansion phase, hence the backtracking will efficiently find routes avoiding occupied cells. Naturally, a laid route will occupy its cells for any later runs of the algorithm. Figure 1 shows a basic example of Lee's algorithm without occupied cell, the left Figure shows the expansion phase, while on the right the backtracking is represented with a laid route between source (S) and destination (D). Figure 2 visualizes the algorithm in the case where there are already occupied cells on the board (denoted by black).

As for parallel computing, it is apparent that this algorithm can be implemented in a way that multiple source-destination pairs are being calculated in parallel, using a shared data representing the grid of cells. It is easy to see how contention is occurring if a thread reserves a route in backtracking, while the other counts it in expansion. It is also evident that a grid with large number of cells but short routes (source-destination are close to each other) is well parallelizable with lower chance of contention, while in smaller grids with relatively long routes, contention will occur with higher probability.

As mentioned, [5] proposed Lee's algorithm as a benchmark for STM. The authors implemented the algorithm using Java and evaluated various optimizations in handling the transactions, assessing the number of routes the algorithms found. That work was expanded in [6], and evaluated STM performance in terms of abort ratio, wasted work and number of transactions in realistic large circuits. In [7], a Ruby based STM implementation was evaluated, in terms of finding the routes on modest difficulty grids.

III. IMPLEMENTATIONS

Due to the practical significance of STM, naturally there is rich support in various programming languages, in the forms of various libraries, or being implemented in the standard library. Without the need to be exhaustive, some examples are as follows. Haskell, as a purely functional language suitable for parallel programming, has native STM support through its standard library. Similarly, Clojure has this kind of built-in STM support. In C/C++, STM is not natively supported, but throughout the years several libraries were built, such as *stmmap*, or *cpp_stm_free*, etc., but none of the implementations were standardized yet. Similarly, Java offers several STM implementation libraries, examples are *JVSTM*, *Deuce* or *DSTM2*. Naturally, these extensions exist in all the other popular languages as well, such as in Ruby, Rust or Golang.

During our evaluations we focused on STM implementations in Scala and one written in Kotlin; in the following we briefly recap these. We selected Scala as our main focus, because of its popularity as a functional programming language supporting concurrent and parallel programming on the Java Virtual Machine (JVM). Scala offers a variety of STM APIs to test. We also looked at Kotlin, as a similar, but less functional programming language. Our goal was to compare both purely functional and imperative STM APIs.

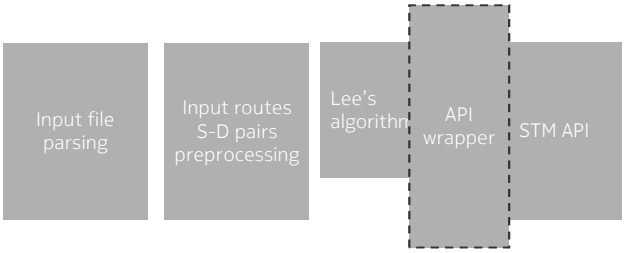

Figure 3. Functional blocks of the implementation

Cats STM [10] is a Scala library enabling composable in-memory transactions. It implements fine grained optimistic concurrency handling with no global locks; automatic retries and composing complex transactions out of elementary ones with its purely functional API. Cats STM supports using multiple runtimes. Also, Cats STM does not have a built-in transactional array, or similar type, so in the implementation we store grid matrices in array of transactional variables.

CHOAM's *Rxn* [11] is our own Scala based implementation. It does not use locks, instead it uses a lock-free multi-word compare-and-swap algorithm [17] to commit transactions. It has both a purely functional, and an imperative API; these use the same underlying engine, so we were able to compare their performance. *Rxn* is technically not a full-featured STM, but it is close enough: it does not have Haskell-style modular blocking (i.e., the *orElse* combinator), but that is not necessary for parallelizing Lee's algorithm. It has a built-in *Ref.Array* type (transactional array), which we use for the board matrices.

The next implementation we tested is based on Kyo [12], a library for algebraic effects in Scala. One of its built-in effects is STM. This STM implementation uses fine-grained locking and has a purely functional API. We run the transactions on Kyo's own runtime with its default configuration. For the board matrices we use an array of transactional variables (*Array[TRef[A]]*), because Kyo does not have a built-in transactional array type.

ScalaSTM is a lightweight STM implementation [13][14] inspired by the STM API in the Haskell standard library. It has a mostly imperative API and uses fine-grained locking. It also has a sophisticated contention manager for retrying conflicting transactions. We use ScalaSTM's built-in *TArray* (transactional array) for the board matrices.

ZSTM is an implementation in the ZIO concurrency framework [15]. It has a purely functional API, similar to the one in the Haskell standard library. We run the ZSTM transactions on their own *zio.Runtime* and we use ZSTM's *TArray* for the board matrices.

The Kotlin implementation we tested is within the Arrow concurrency framework [16]. The algorithm is written in Kotlin, with a thin Scala wrapper. The API of *arrow-fx-stm* is inspired by Haskell's STM package, but it is nevertheless mostly imperative. We run the STM transactions on the default coroutine dispatcher of Kotlin. We use *TArray* for the grid matrices.

During the evaluation of results in Section 0, we refer to two possible basic solutions for STM with regards to the implementations listed above, that is *opacity* and *early release*. *Opacity* [19] is a consistency property specifically for STM systems. The consistency of committed transactions is usually guaranteed by all STM systems (e.g., by performing a validation step during commit). However, an opaque STM also guarantees the consistency of all running transactions. That is, a transaction in an opaque STM is never able to observe an inconsistent view of memory. Conversely, a transaction in a non-opaque (i.e., transparent) STM might observe such an inconsistent view, and then later (e.g., when trying to commit) detect the inconsistency, roll back, and retry. Depending on the specific logic of a transaction, the lack of opacity could lead to observing violation of invariants, which in turn could lead to, e.g., out-of-bounds reads or infinite loops. On the other hand, if an STM guarantees opacity, it will typically need to roll back and retry transactions more often, which could lead to performance degradation.

The authors of [18] proposed early release as an optimization for STM transactions. This is a mechanism to remove items from the read set of a transaction, in effect releasing those memory locations earlier than the commit of the transaction (because the transaction does not need them anymore). On one hand, this has the potential to reduce the number of conflicts the transaction encounters, thus potentially increasing performance. On the other hand, the released memory locations will not be part of any later automatic validation (e.g., during commit), so early release must be used with care, to preserve the correctness of the transaction.

IV. IMPLEMENTATION ARCHITECTURE AND TEST ENVIRONMENT

To enable better understanding of the results, main design and implementation considerations are introduced in the following subsections.

A. Design and implementation

The bases of main building blocks of the software implemented to test performance of various STM implementations is shown in Figure 3. The first block is responsible for parsing the input file given to the algorithm; that contains the description of the board (grid) and the source-destination (S-D) pairs between which the routes are to be laid. Then there is an initial optimization, as for all the source-destination pairs a simple grid-distance is calculated, and S-D pairs are sorted in increasing order. For those pairs that have the same grid-distance, a pseudorandom shuffling is applied, to reduce the number of trivial conflicts (because S-D pairs with coordinates close to each other are often also specified close to each other in the input files). Lee's algorithm will be then executed on the S-D pairs in this order.

In this implementation, a small generalization of Lee's algorithm is introduced, compared to the basics shown in Section II. Namely, in this version, we still allow routes to cross in the grid. In terms of route laying on a circuit board, this mimics the case when there can be multiple layers. However, in this version of the algorithm we assign a cost to the routes. That is, we assign a unit cost to each cell allocated for a route and if another route crosses an already existing one,

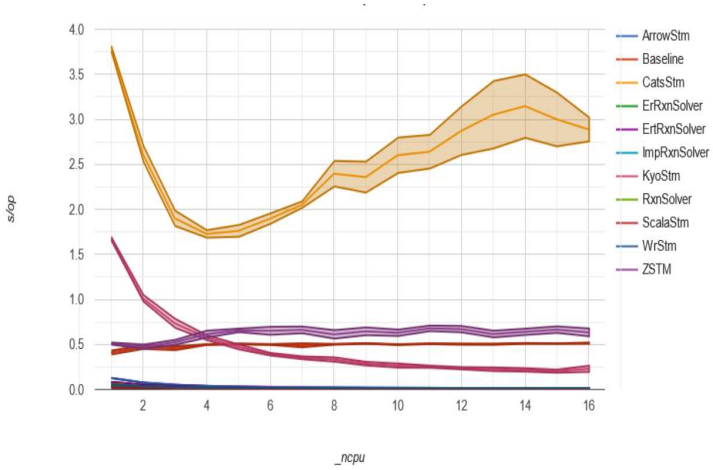

Figure 4. Completion time for simple input

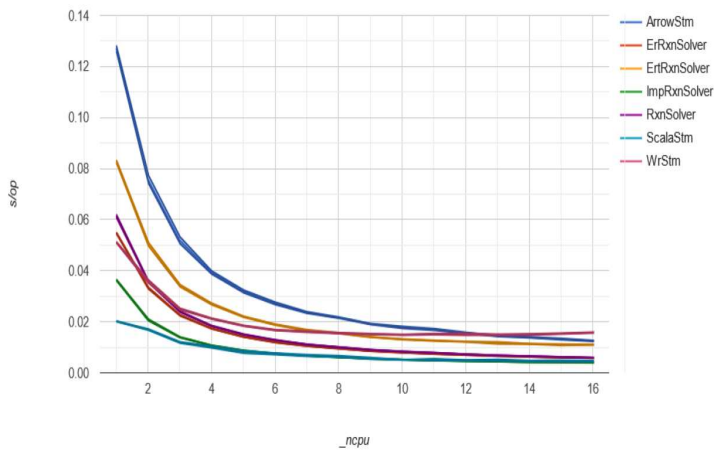

Figure 5. Completion time for simple input, zoomed

there is a double cost associated to that cell within this next route. Similarly, if a third route is to be laid using this same cell, that would again double this cell's cost (hence it would cost four units) and so on, each layer doubles the cost (exponentially rising cost). Finally, the algorithm selects the route with the lowest cost. Note that the original version of the algorithm that does not allow route crossing is a subset of this approach with allocating infinite cost to route crossing.

The parallelization is handled in the following manner: the S-D pairs are evaluated in parallel batches that have the size equivalent to available CPU threads. Whenever a thread finishes (a route for an S-D pair is laid), the next one from the ordered list starts. Note that the algorithm finishes when an S-D route is found; when the transaction should abort and restart for example due to validation error or commit error, that is handled by the STM engine itself.

In Figure 3 the functional blocks of the algorithm, the tested STM API (listed in Section II) and a block labelled as "API wrapper" are interwoven. This is because we have implemented the algorithm for each STM API in a way that the implementation natively uses the API and its data structures, therefore, the very implementation code is specific to the given API. For example, for a functional API a function itself can be passed, hence the STM engine itself can call "back" to the algorithm.

The API wrapper part in the Figure is specific to testing the ScalaSTM API. Namely, ScalaSTM was tested in an idiomatic way, using its default imperative API. However, as in general we would like to harness the strengths of functional programming, we have also implemented and tested a thin layer, that wraps the ScalaSTM API in a monadic (purely functional) API similar to that of Cats STM. This way we can also get some ideas about the overhead of a monadic ("programs as values") API in Scala. (We have considered creating a unified API for *all* the STM libraries, and implementing Lee's algorithm only *once*, using this API. However, as measurements on the wrapped ScalaSTM API showed significant performance degradation due to the wrapping, we have not done this.) We summarize all the variants we implemented, and the STM libraries we used in Table I.

CHOAM has both a purely functional and an imperative API; it also has various optimization options. To compare the performance effect of these variations, we have implemented four versions of Lee's algorithm with CHOAM:

- One using the default (purely functional and safe) API (*RxnSolver*).
- An optimized one, which uses "early release" [18] to make the transaction log smaller (*ErtRxnSolver*). This optimization would not be safe in arbitrary transactions, but as discussed in [5], it is safe for Lee's algorithm. This version also uses non-opaque (i.e., "transparent") reads [19], to further decrease the probability of conflicts.
- Another optimized version, which uses "tentative reads", as an alternative implementation of early release (*ErRxnSolver*).
- A version which (unlike the other three) uses the imperative API of CHOAM (*ImpRxnSolver*). It has no early release, or other extra optimization (thus, it can be seen as the direct imperative equivalent of *RxnSolver*).

We run the various implementations on asynchronous runtimes they are designed for. When they are not designed for a specific runtime, we run them on the thread-pool of Cats Effect. We configure these runtimes by turning off features which could have a negative performance impact.

The transactions in these implementations of Lee's routing algorithm are read heavy, but at the end they always write to some locations (to lay a route). This means that read-only transactions, and transactions which only access a very small number of memory locations are not measured.

We also have implemented a sequential (non-parallelized) version of the same algorithm, which serves as the *baseline* for comparison to the parallel ones. This sequential implementation is intentionally not very well optimized, because we wanted to compare it to similarly high-level and easy-to-use STMs.

All the implementations used for this benchmarking are available as open source [20].

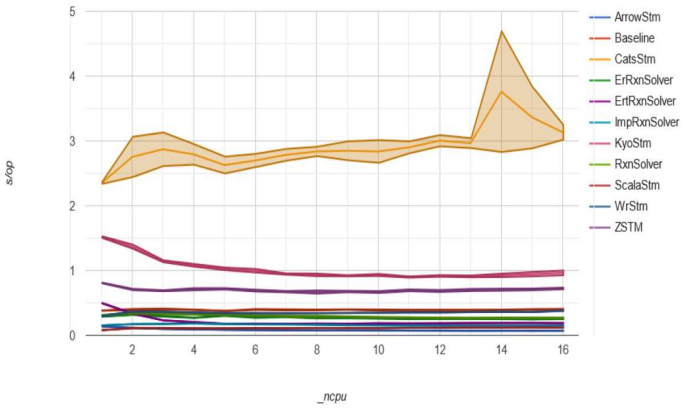

Figure 6. Moderate complexity input

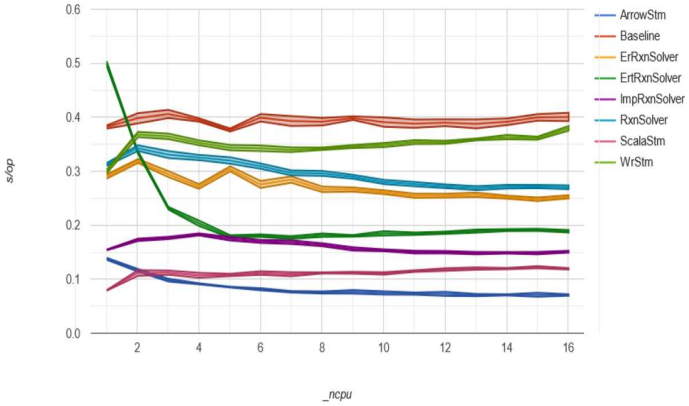

Figure 7. Moderate complexity input, zoomed

B. *Experimental setup*

We run the benchmarking software described above on the Java Virtual Machine (JVM). This is packaged into a Docker container, because we wanted the measurement software to be portable and easily automatable, and the measurement easily reproducible.

The server used has two Intel Xeon E5-2680 v3 processors running at 2.5 GHz, with 12 physical cores, that is 24 cores in total. During the measurements hyperthreading was disabled, therefore each thread is running on a physical core. Turbo boost was also disabled. During the measurements, one control parameter is the number of cores allocated to the JVM, and the software itself implements parallelization in a way that the number of available cores is queried from the JVM.

The server is equipped with 256 Gbytes of physical memory, but the JVM heap size was configured to be 16 Gbytes. All the implementation is based on Scala 3.7.0 and OpenJDK 21.0.7 (Corretto).

We used three inputs (circuit boards for laying routes) with different sizes in terms of the number of cells in the grid and number and length of routes to be laid, as will be discussed in the next section: a well parallelizable simple synthetic input, a modest one, and a complex one coming from real circuitry.

The algorithm for laying routes in the simple and moderate complexity boards was continuously run for 300 seconds for each input, and for each implementation, for a given number of available CPUs. Based on the completion times needed for solving an input (see next section), this results in several hundreds to several thousands of runs for each data point. For the complex input, due to its excessive complexity, 20 runs were performed for each data point.

We used the Java Microbenchmark Harness (JMH) [21] to perform the measurements, in its default time-based "average time" benchmark mode. In this mode JMH repeatedly calls a benchmark method until a timeout of 10 seconds is reached (JMH calls this 1 iteration). JMH performs the measurements in a forked JVM (i.e., it launches a separate process just for the measurement); we configured it to repeat this forking 6 times. We performed 5 warmup iterations and 5 measurement iterations (that is, 50+50 seconds total per fork); the measurement results of the warmup iterations are ignored, and the execution times of the benchmark method during the measurement iterations are averaged. (An exception to this is the last complex input, where we used the "single-shot" mode of JMH, resulting in the average of 20 benchmark method executions, as mentioned above.) The purpose of the warmup iterations is to avoid measuring in a "cold" JVM, i.e., in which the just-in-time compiler (JIT) did not yet optimize the running methods.

V. RESULTS AND EVALUATION

The charts in this Section show the results of our measurements. On the vertical axis, we show the completion time, i.e., the time required (in seconds) to solve one particular input board. The curves show the average time required to run on the input; the shaded area shows a 99.9% confidence interval (it is not visible on some of the curves). The horizontal axis shows the number of CPU cores available to the solvers. This way we can analyze the scalability of the various STM engines when used for parallelization.

Figures 4 and 5 show measurement results for a 200×200 circuit board with 90 routes (i.e., source-destination pairs), which is the simple input. The routes are all very short (10), the solutions are trivial (each is a straight line), and they never cross each other. (This board is a smaller version of the board called "simple" in [6].) Thus, solving this synthetic input is, in theory, perfectly parallelizable. While this is not a realistic circuit board, we use it to measure the ability of the various STM engines to exploit the potential parallelism (which is very high here). Figure 4 shows results for all the STM engines and variants we measured. The smaller results (i.e., results for the faster implementations) are not visible on that chart, so they are shown in Figure 5 (which is essentially the zoomed in version of the bottom of Figure 4).

In Figure 4, we can see that the slowest STM implementation on this particular input is Cats STM (labeled *CatsStm*). As we increase the number of cores, at first it scales well until around 4 cores; then performance starts to degrade. We suspect the reason for this is the behavior of the locks used under higher contention (Cats STM uses the built-in locks of Cats Effect, which use a single atomic reference). Even at the best point in the chart (at 4 cores), this engine is slower than the non-parallelized baseline implementation (*Baseline* in the chart). The reason for this is probably (at least in part) the high

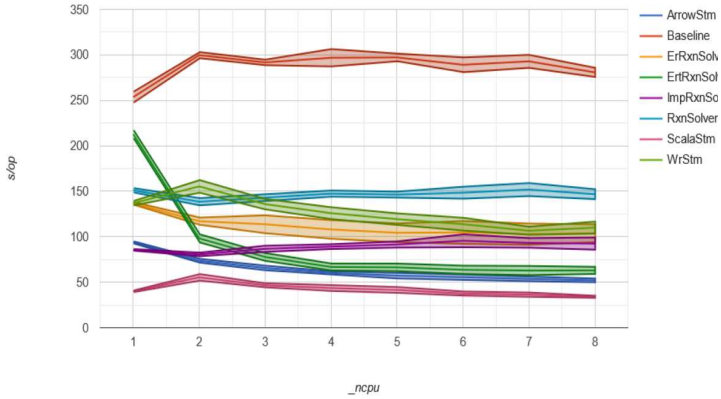

Figure 8. Completion time of complex realistic board

overhead of the immutable and purely functional data structures used by Cats STM.

In the same chart, we can see that the STM engine of Kyo (labeled *KyoStm*) seems to scale well with the number of processors, although there is less and less improvement the more cores are used (this is expected of any parallelization scheme that requires some coordination between cores). On the other hand, ZSTM seems unable to scale beyond 2 cores; we suspect the reason is that the locks it uses are blocking physical threads, and its runtime does not seem to start other threads, or compensate somehow for these threads that are not doing useful work.

On Figure 5 we can see the implementations which are able to solve this input much faster. All of them show a scaling curve similar to *KyoStm* (i.e., they scale well, but the improvements are smaller and smaller). If we compare the default ScalaSTM implementation (*ScalaStm*), and its variant wrapped in a purely functional API (*WrStm*), we can see that the purely functional API has a very significant overhead (around 2-3 times slower). We see similar, but smaller differences between the solvers using the functional and imperative APIs of CHOAM (*RxnSolver* and *ImpRxnSolver* respectively). The variants using the various forms of early release (*ErRxnSolver* and *ErtRxnSolver*) show little or no performance advantage over *RxnSolver*; this is expected, as early release is used to decrease the number of conflicting transactions, and due to the nature of the input, there are no (or very few) conflicting transactions here. (*ErtRxnSolver* is even slower here, due to the overhead associated with that particular implementation of early release.)

Figures 6 and 7 show results for another input with moderate complexity, a "small but realistic board" (*testBoard.txt* from [7]). This board is 75×75, and has 203 routes to solve, both short and long. This input has lots of potential conflicts, so we expect solving it to scale worse with the number of cores. (As before, some implementations are significantly faster than others, so Figure 7 shows the zoomed-in lower part of Figure 6.)

As expected, we see the implementations becoming only modestly faster as the number of cores increases, or not at all. An interesting exception to this is *ErtRxnSolver*, which seems to scale very well from 1 to 5 cores (and it is mostly flat after that). We suspect this is due to the relatively high overhead of

TABLE I. SUMMARY OF THE IMPLEMENTED VARIANTS

Name	STM library	API style
CatsStm	Cats STM	functional
RxnSolver	CHOAM	functional
ErRxnSolver		functional
ErtRxnSolver		functional
ImpRxnSolver		imperative
KyoStm	Kyo	functional
ScalaStm	ScalaSTM	imperative
WrStm		functional
ZSTM	ZIO	functional
ArrowStm	Arrow	imperative

implementing early release this way, which is then able to be overcome by more parallelism (allowed by using early release and non-opaque reads to decrease transaction conflicts).

As Figure 6 shows, the slowest implementation is Cats STM as previously. The fact that it is the slowest on both inputs suggests that it has very high single-threaded overhead (probably due to the immutable data structures used and the purely functional API).

The STM engine of Kyo shows some limited ability to scale, but despite this, it is slower than ZSTM, which (as before) does not scale well. This contrasts with the previously discussed results, where Kyo's superior scalability was able to overtake ZSTM at 4 cores.

Interestingly, none of these three implementations (*CatsStm*, *KyoStm*, *ZSTM*) is faster than the baseline non-parallelized implementation (on this input).

In Figure 7 we see the results of the faster implementations on the same input. All of them are faster than the *Baseline* (non-parallelized) version. The fact that they are faster even on a single core (i.e., no parallelism) is because we did not bother optimizing the baseline (we wanted to compare "conveniently coded", high level implementations). As mentioned before, all of them show no or limited scaling. Interestingly, *ScalaStm* (and its *purely* functional variant, *WrStm*) show only performance degradation with more cores (i.e., they are fastest with 1 core). This suggests that they are unable to exploit the very limited potential parallelism of this input.

RxnSolver and *ErRxnSolver* show modest scaling, (but still, they are slower than ScalaSTM). Of the two, *ErRxnSolver* is the faster: as expected, using early release helps to reduce transaction conflicts.

The fastest implementation (on this input) is *ArrowStm*, which scales reasonably well, and overtakes ScalaSTM at 3 cores.

Figure 8 shows our measurement results on a complex real circuit board of a memory module (board "mem" in [6]). This is a 600×600 board, with 3101 routes to solve. As this board is much bigger and more complicated than the previous two, solving it requires orders of magnitude more time (minutes

instead of fractions of seconds as before). For this input, we do not show results for Cats STM, ZSTM or Kyo as they showed limited performance even for the moderately complex board.

As for the previous input, we see all the (faster) implementations improving on the *B*aseline (even at 1 core). ScalaSTM is the fastest here, showing an interesting curve: when running on multiple cores, it is first slower than on 1 core, but slowly getting faster, and overtaking its single-core performance at 6 cores. We suspect ScalaSTM has some optimizations specifically for single-threaded execution. (Its purely functional version, *WrStm* shows the same scaling behavior, but with a significant overhead due to the API wrapping). As before, *ArrowStm* performs well, and scales well, but in this case, cannot overtake ScalaSTM.

Comparing the various versions implemented with CHOAM, we see the unoptimized, purely functional variant (*RxnSolver*) being generally the slowest (and much slower than Scala STM). The variant *ErRxnSolver* (using early release) shows a significant improvement, which grows as the number of cores increases (this is expected, as the potential for conflicts is bigger with more cores, and early release reduces these conflicts). *ErtRxnSolver* (which uses both early release and non-opaque reads) starts slower (as before, due to the bookkeeping overhead of this particular implementation), but scales much better, overtaking all the other CHOAM variants, but it is still unable to overtake *ArrowStm*. Again, this scaling behavior is expected, like for testBoard.txt.

VI. Conclusions and Future Work

Considering all the measurement results detailed in the previous section, we make the following observations.

Comparing purely functional APIs with their imperative counterparts (i.e., *WrStm* with *ScalaStm*, and *RxnSolver* with *ImpRxnSolver*), we see overheads from around 30% to around 300% for the purely functional APIs. This is probably due to the purely functional APIs allocating an enormous amount of very small objects, which stresses the garbage collector of the JVM.

If we compare all the functional APIs with all the imperative ones, we see a similar trend: imperative ones tend to be faster (as expected). However, there is a significant difference in performance between the functional ones themselves, so there is clearly a way to decrease their overhead.

The Kotlin implementation (*ArrowStm*) performs consistently well and scales well. This is probably in part due to its imperative nature, but we suspect it might also have something to do with it being executed on the Kotlin co-routine scheduler. All the other implementations run on runtimes of Scala effect systems, which tend to schedule tasks differently from the coroutine scheduler. We leave examining the precise effect of the scheduler behavior on STM performance for future work.

On inputs where we expect transaction conflicts, using early release (and non-opaque reads) shows a clear performance advantage. This is expected, as these optimizations aim to decrease the number of conflicts, and they succeed at that goal.

Preliminary profiling shows that both Cats STM and ZSTM spend a considerable portion of their execution time maintaining the transaction log. This is not surprising, as the transactions we measured here are relatively big (i.e., their logs contain a lot of entries), especially for the last input (the memory module). Thus, optimizing their log data structures is a potential future performance improvement for these STM engines.

References

[1] M. Herlihy, J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures” ACM SIGARCH Computer Architecture News, Volume 21, Issue 2, 1993, pp. 289 - 300.

[2] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, “Software transactional memory for dynamic-sized data structures”, PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing, 2003, pp. 92 – 101

[3] J. Chung et al., “The common case transactional behavior of multithreaded programs”, Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture, 2006, pp. 266–277

[4] R. Guerraoui, M. Kapalka, and J. Vitek, “STMBench7: a benchmark for software transactional memory”, ACM SIGOPS Operating Systems Review, Volume 41, Issue 3, pp. 315 - 324

[5] I. Watson, C. Kirkham, and M. Lujan, “A Study of a Transactional Parallel Routing Algorithm,” 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), Brasov, Romania, 2007, pp. 388-400

[6] M. Ansari et al., “Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory”. In: Bourgeois, A.G., Zheng, S.Q. (eds) Algorithms and Architectures for Parallel Processing. ICA3PP 2008. Lecture Notes in Computer Science, vol 5022.

[7] C. Seaton, “Context on STM in Ruby”, online https://chrisseaton.com/truffleruby/ruby-stm/ accessed: 17/6/2025

[8] C. Y. Lee, “An Algorithm for Path Connections and Its Applications,” in *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346-365, Sept. 1961, doi: 10.1109/TEC.1961.5219222

[9] R. Guerraoui, M. Kapalka, and J. Vitek, “STMBench7: A benchmark for software transactional memory”. EuroSys ’07: Proceedings of the 2nd European Systems Conference, pp. 315–324. ACM Press, March 2007.

[10] T. W. Spence, Cats STM, online https://github.com/TimWSpence/cats-stm/, accessed: 18/6/2025

[11] D. Urban, CHOAM, online: https://github.com/durban/choam , accessed: 18/6/2025.

[12] Online: https://getkyo.io/ , accessed: 18/6/2025

[13] N. Bronson, Scala-STM, online: https://github.com/nbronson/ accessed: 18/6/2025.

[14] N. Bronson, H. Chafi, and K. Olukotun, “CCSTM: A Library-Based STM for Scala”. Proceedings of the First Annual Scala Workshop. Lausanne, 2010

[15] Online: https://github.com/zio/zio/tree/series/2.x/core/shared/src/main/scala/zio/stm , accessed: 18/6/2025

[16] Online: https://arrow-kt.io/learn/coroutines/stm/ , accessed: 18/6/2025

[17] R. Guerraoui, A. Kogan, V. Marathe, and I. Zablotchi, “Efficient Multi-word Compare and Swap,” in Proceedings of

the 34th International Symposium on Distributed Computing, Virtual Event, Oct. 2020.

[18] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, “Software transactional memory for dynamic-sized data structures”. In Proceedings of the twenty-second annual Symposium on Principles of Distributed Computing, pages 92–101, 2003.

[19] R. Guerraoui, M. Kapalka, “On the Correctness of Transactional Memory”. In PPoPP ‘08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 175–184, New York, NY, USA, 2008. ACM.

[20] D. Urban, stm-benchmark, online: https://github.com/nokia/stm-benchmark, accessed: 19/6/2025 Online: https://openjdk.org/projects/code-tools/jmh/ accessed: 19/6/2025

[21] Online: https://openjdk.org/projects/code-tools/jmh/ , accessed: 19/6/2025