LiTM: A Lightweight Deterministic Software Transactional Memory System

Yu Xia, Xiangyao Yu, William Moses, Julian Shun, Srinivas Devadas
MIT CSAIL
{yuxia,yxy,wmoses,jshun,devadas}@mit.edu

Abstract

Deterministic software transactional memory (STM) is a useful programming model for writing parallel codes, as it improves programmability (by supporting transactions) and debuggability (by supporting determinism). This paper presents *LiTM*, a new deterministic STM system that achieves both simplicity and efficiency at the same time. *LiTM* implements the *deterministic reservations* framework of Blelloch et al., but without requiring the programmer to understand the internals of the algorithm. Instead, the programmer writes the program in a transactional fashion and *LiTM* manages all data conflicts and automatically achieves deterministic parallelism. Our experiments on six benchmarks show that *LiTM* outperforms the state-of-the-art framework Galois by up to 5.8× on a 40-core machine.

Keywords Deterministic Parallelism, Software Transactional Memory

1 Introduction

A *transaction* is a sequence of operations that must succeed or fail as a whole. Transactions offer a powerful abstraction for parallel programming as they enable programmers to wrap sequences of operations in regions that execute atomically, and not have to worry about concurrency issues. Software and hardware transactional memory techniques are an active topic of research, where the goal is to provide programmability, while maintaining good performance and scalability. Transactions are also used in database management systems, where they bring useful and easy-to-understand ACID properties (Atomicity, Consistency, Isolation, and Durability).

One disadvantage of transactions, however, is the nature of non-determinism — even the strongest isolation level, serializability, allows transactions to be arbitrarily interleaved. This makes it challenging to reason about the correctness of the parallel program, as well as making debugging difficult.

Deterministic parallelism [2, 18, 19] is a promising technique that can largely resolve this issue. A parallel program is considered *deterministic* if each execution produces identical results, regardless of platform- or execution-specific features such as the number of processors being used or the scheduling of threads. Determinism makes programming significantly easier since a programmer only needs to consider a single execution order instead of all possible interleavings of parallel code. Debuggability is also improved since a bug can be reproduced by simply re-executing the program, effectively solving the notorious Heisenbug [7] issue where a bug manifests itself differently in multiple executions of the same program.

Supporting determinism in transaction processing has been an active research area. Previous work has proposed to use determinism to improve debuggability of STM systems [12, 13, 21]. These previous approaches, however, either suffer from scalability bottlenecks [13, 21] or incur performance overhead in conflict resolution [12]. For example in Galois, the state-of-the-art deterministic STM, performance is compromised by as much as 6× when determinism is turned on [12]. It is not surprising that a deterministic program performs worse than a non-deterministic one due to having a more constrained execution. However, the dramatic performance overhead can deter programmers from writing and using deterministic programs. Therefore, improving the performance of deterministic transaction processing is crucial for its widespread adoption.

In this paper we design *LiTM*, a deterministic STM system for multicores.[1] *LiTM* follows the *deterministic reservations* paradigm proposed by Blelloch et al. [1], but provides a more convenient and natural interface. In contrast to [1], *LiTM* hides the algorithm completely from the programmer, who only needs to provide the content of the transactions instead of analyzing the algorithms and supplying the corresponding functions to handle the data conflicts manually. *LiTM* takes as input a set of transactions, automatically groups them into batches, analyzes their access patterns, detects conflicts, applies changes, and re-executes aborted transactions in a way that is transparent to the programmer. We also explore variants of *LiTM* that can reduce memory usage.

PMAM'19 , February 17, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6290-0/19/02…$15.00
https://doi.org/10.1145/3303084.3309487

[1] We have open-sourced the framework at https://github.com/yuxiamit/LiTM.

Our evaluation on a 40-core machine demonstrates that *LiTM* can achieve up to 9.4× performance improvement over the sequential versions of the algorithms. Compared to the state-of-the-art deterministic STM framework Galois [12], *LiTM* achieves up to 5.8× speedup. Compared to hand-optimized parallel baselines [14], which require significantly more effort to program, *LiTM* incurs a modest overhead of at most 3×.

This paper makes the following contributions:

- We design the *LiTM* deterministic STM system, which provides a simple yet powerful abstraction for programmers to write highly efficient parallel programs.
- We implement six applications in *LiTM*: maximal independent set, maximal matching, spanning forest, PageRank, random permutation, and list contraction.
- We compare *LiTM*'s performance against hand-optimized serial and parallel baselines, including Galois, a state-of-the-art deterministic parallel programming framework. We show that *LiTM* can outperform Galois by up to 5.8×, and incurs modest overhead over the hand-optimized baselines.

The rest of the paper is organized as follows. Section 2 discusses the background and motivation for *LiTM*. Section 3 presents the protocol of *LiTM*. In Section 4, we evaluate *LiTM* on six applications and compare them against existing implementations. Finally, we present related work in Section 5, describe avenues for future work in Section 6, and conclude the paper in Section 7.

2 Background

In this section, we discuss deterministic reservations, the technique that motivates *LiTM*. We also discuss the limitations of the protocol, which *LiTM* seeks to resolve.

2.1 Deterministic Reservations

Deterministic reservations [1] is a framework to write internally deterministic parallel programs comprised of multiple iterates. It guarantees that the program outcome is identical even when the iterates are executed in different orders. The key technique behind the framework is a way to deterministically resolve conflicts based on the predefined priorities of iterates, while allowing them to be executed in parallel. If there are any conflicts between iterates, the higher priority iterate takes precedence. This is achieved by having each iterate *reserve* its accessed data elements by writing its own priority into a reservation array *only when its priority is higher* than the current iterate reserving on the same data. Regardless of the order in which a data element is reserved, the iterate with the highest priority will reserve it in the end. Random numbers used in the programs must be deterministic across multiple runs [10].

For efficiency, the deterministic reservations framework orders the iterates by their priorities and processes them in multiple batches. In each batch, the framework first runs a *reserve()* function for each iterate in the batch in parallel.

Algorithm 1: MIS using *LiTM*.

```
1  enum {InMIS, NotInMIS} States;
2  DVector Flags = {NotInMIS, NotInMIS, ..., NotInMIS} # size n
3  Function T.runTxn()
4      for ngh in T.vertex.neighbors do
5          if Flags[ngh] == InMIS then
6              Flags[T.vertex] = NotInMIS
7              return
8      Flags[T.vertex] = InMIS
```

Algorithm 2: MIS using deterministic reservations.

```
1  enum {Undecided, InMIS, NotInMIS} States
2  char Flags[] = {Undecided, Undecided, ..., Undecided} # size n
3  int reservation[] = {∞, ∞, ..., ∞} # size n
4  Function reserve(i)
5      for ngh in vertices[i].neighbors do
6          if Flags[ngh] == InMIS then
7              Flags[i] = NotInMIS
8              return false # bypass the commit phase
9      # writeMin(&x, i) atomically sets x to min(x, i)
10     writeMin(&reservation[i], i)
11     return true
12 Function commit(i) # returns whether the iterate commits
13     for ngh in vertices[i].neighbors do
14         if (Flags[ngh] == Undecided and reservation[ngh] < i)
             or Flags[ngh] == InMIS then
15             reservation[i] = ∞
16             return false
17     Flags[i] = InMIS
18     return true
```

It reserves all of the data elements that the iterate accesses. Then, in parallel the framework runs a *commit()* function for each iterate to check whether it has successfully reserved all elements, and if so, applies the iterate's changes to shared memory, and otherwise, moves the iterate to the next batch and re-processes it later. Inside these two functions, data conflicts are handled manually by the programmer. The algorithm continues processing iterates in batches until no more remain. It has been shown that deterministic reservations can achieve very competitive performance compared to other approaches [1].

2.2 Example on Maximal Independent Set

We use maximal independent set (MIS) as an example application to demonstrate how deterministic reservations works and also to point out its limitations.

In a graph, an *independent set* is a set of vertices with no edge connecting any pair of them. A *maximal independent set* is an independent set such that if any other vertex joins the set, independence is violated.

Algorithm 1 shows how MIS can be implemented using transactions (this is also the code to implement MIS in *LiTM*). Initially, all vertices are assigned to not be in the MIS using

the *NotInMIS* flag (line 2). For each vertex, the *runTxn()* function is executed. If any neighbor of the vertex is already in the MIS (lines 4-5), by definition, the current vertex cannot be selected in the set, and hence the transaction returns leaving the vertex out of the MIS (line 6–7). Otherwise, the current vertex is added to the MIS on line 8 with the *InMIS* flag.

To implement this application in the deterministic reservations framework, a programmer writes two functions, *reserve()* and *commit()*, as shown in Algorithm 2.[2] Both functions take the iterate number i as input. The iterate uses i as its priority, with a smaller value being higher priority. Initially, all vertices are given an undecided status (line 2), and the memory locations for performing reservations are initialized to ∞ (line 3). In the reserve function, if any neighbor of the vertex of interest has been selected, the current vertex is marked as not in the MIS (*NotInMIS*) and the function returns and tells the framework to bypass the *commit* phase (lines 5–8). Otherwise, the current vertex *might* be in the set and is therefore reserved using the *writeMin()* function, which atomically updates *reservation*[i] with i if its previous value is greater than i (line 10).

In the commit function, if any neighbor of the vertex is in the MIS (*InMIS*), or undecided (*Undecided*) but reserved by a higher-priority transaction, the reservation is cleared and the current transaction has to abort (i.e., return false) in this batch (lines 13–16). Otherwise, the vertex is marked as *InMIS* and the transaction commits (lines 17–18).

LiTM inherits the same spirit of processing iterates in two phases. However, *LiTM* does not require the programmer to write the two functions manually. Instead, the programmer can provide the sequential logic of the code as a transaction (i.e., like in Algorithm 1), and *LiTM* automatically coordinates the transactions. The programmer does not need to deal with concurrency issues in *LiTM*. The details of the *LiTM* protocol will be discussed in Section 3.

3 The *LiTM* Protocol

The goal of *LiTM* is to implement the deterministic reservations framework without requiring a programmer to provide the *reserve* and *commit* functions. Instead, the programmer writes a transaction, and *LiTM* automatically extracts the codes for reserve and commit functions based on the provided transaction. We discuss the API of *LiTM* in Section 3.1, the data structures used in *LiTM* in Section 3.2, the detailed protocol in Sections 3.3 to 3.6, design parameter selection in Section 3.7, and design variants in Section 3.8.

3.1 API of *LiTM*

In *LiTM*, the programmer writes the logic of transactions. The shared data structures are declared as special types, so that the system can capture read and write operations to them.

[2]We note that there is a faster MIS implementation using an algorithm-specific optimization in [1]. We do not present it here because the optimization is not general across applications. Later we will use it as the hand-optimized baseline in Section 4.

Algorithm 3: The overall logic of *LiTM* — The reserve, commit, and cleanup phases are highlighted in different colors. **foreach** loops are executed by all threads in parallel in any order.

```
1  # txns: list of transactions to be executed.
2  Function run(txns)
3      batch = txns.pop(batch_size)
4      while not batch.empty() do
5          next_batch = []
6          foreach T in batch do
7              T.runTxn()
8          foreach T in batch where T.WS ≠ ∅ do
9              can_commit = run_commit_phase(T)
10             if not can_commit then
11                 next_batch.append(T)
12         batch = next_batch +
               txns.pop(batch_size - next_batch.size())
```

Specifically, *LiTM* supports a special *DVector* type as a shared array. In Algorithm 1, for example, the *Flags* structure has the type of *DVector* with its base type as chars. Section 3.4 shows how *LiTM* handles reads and writes to a *DVector*.

3.2 Data Structures in *LiTM*

The following data structures are required in *LiTM* to perform the reserve and commit functions.

Lock Table (LT) is used to perform reservations. It is implemented as a hash table, where each entry in LT contains the priority (which is of integer type) of a transaction that reserves the data element that maps to the LT entry. The number of entries in LT does not have to equal the total number of data elements. If the LT is too small, hash collisions may occur, leading to more transaction aborts and performance degradation, but the execution is still functionally correct.

Read and Write Sets (RS and WS). RS (WS) of a transaction T keeps the pointers (pointers and values) of each data element that T reads (writes). These two sets are updated during the reserve phase and later used in the commit phase to check whether T has successfully reserved all elements it has touched so that it is able to commit. For a committed transaction, the values in the WS are copied to the *DVector*s.

3.3 Overall logic of *LiTM*

Algorithm 3 shows the overall logic of *LiTM* for executing a set of ordered transactions. The execution contains multiple batches, each broken down into three separate phases. The *reserve* and *commit* phases correspond to the two functions in the deterministic reservations algorithm. The *cleanup* phase tidies up relevant data structures between different batches.

LiTM begins by taking a prefix of *batch_size* transactions as a batch (line 3). If the number of remaining transactions is less than *batch_size*, the batch will contain all remaining transactions. The following three phases are performed as long as there are still transactions remaining (line 4).

Algorithm 4: The reserve phase — Read and write operation to the ith element in a *DVector*.

```
1  Function DVector.read(T, i)
2      addr = &DVector._values[i]
3      T.RS.insert(addr)
4      if addr in T.WS then
5          return T.WS[addr]
6      return DVector._values[i]
7  Function DVector.write(T, i, val)
8      addr = &DVector._values[i]
9      T.WS.insert(addr, val)
10     # begin atomic section
11     if T.priority < LT[ hash(addr)] then
12         # Smaller numbers mean higher priorities
13         LT[ hash(addr) ] = T.priority
14     # end atomic section
```

During the *reserve phase*, the *runTxn()* function is executed for each transaction in the batch. This function is the transaction logic defined by the programmer (see Algorithm 1 for an example). *runTxn()* performs reservations and updates the RS and WS. More details will be in Section 3.4.

The *commit phase* begins after all transactions in the batch have finished the reserve phase. If a transaction T from the batch is read-only, i.e., having an empty WS, T can simply commit as if it happens logically at the beginning of the batch. Skipping the commit phase for read-only transactions is an optimization that can reduce the RS storage as well as improve performance, while retaining determinism. For a read-write transaction T, the *run_commit_phase(T)* is executed, which checks whether T is able to commit or not. More detailed discussion will be presented in Section 3.5. If T is not able to commit in the current batch, it is appended to *next_batch* in a deterministic way and will be reprocessed in the next batch.

Finally, the *cleanup phase* assembles the next batch of transactions by taking the remainder of the current batch, and another prefix from the input transactions.

3.4 Reserve Phase

During the reserve phase, the *runTxn()* function of each transaction is executed, wherein the *DVectors* are accessed. For each read or write operation to a *DVector* element, the logic in Algorithm 4 is executed. For a read by transaction T, the address of the accessed element is inserted into T's RS (line 3). If it is in T's WS, the value in the WS is returned (line 5), otherwise, the value stored in the *DVector* is returned (line 6).

For a write by transaction T, both the address and the new value of the element are inserted the T's WS (line 9). Then the write is reserved in the LT by atomically replacing the corresponding entry with T's priority if the existing priority is lower (the value is larger) (line 10–14). Updating the value in the shared data structure is delayed until the commit phase.

Both the *read()* and *write()* functions in Algorithm 4 are deterministic. Each read always returns the value at the start of the batch (since there are no modifications to shared data structures yet). After the batch, the state of the LT is always the same, regardless of the order in which the transactions are scheduled — an element is always reserved by the transaction with the highest priority that writes to the element.

Algorithm 5: The commit phase – implementation of the *run_commit_phase(T)* function.

```
1  Function run_commit_phase(T)
2      # Check whether T successfully reserved all elements
3      for addr in T.RS ∪ T.WS do
4          if T.priority < LT[ hash(addr) ] then
5              return false
6      # Copy elements in the WS to DVectors
7      for (addr, val) in T.WS do
8          memcpy( addr, &val, sizeof(val) )
9      return true
```

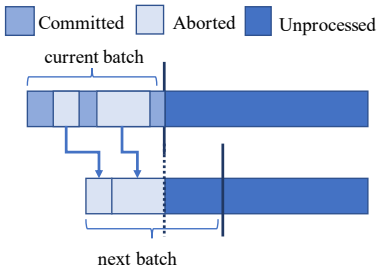


Figure 1. The deterministic execution of batches of transactions.

3.5 Commit Phase

The *run_commit_phase(T)* function is the main logic in the commit phase, which is defined in Algorithm 5. *LiTM* first checks whether a transaction T has successfully reserved all of the elements that it accesses during the reserve phase by comparing T's priority with the priority stored in the LT. If any element is reserved by another transaction with a higher priority, T is not able to commit and has to be moved to the next batch. If all elements are reserved by T, T's WS is copied back to the shared *DVector*s. The execution in the commit phase is deterministic because the states in the LT are deterministic and only the highest-priority transaction can reserve and hence possibly update a particular element.

3.6 Cleanup Phase

After all transactions within a batch finish the reserve and commit phases, transactions that cannot commit due to conflicts are moved to the next batch. The job of the *cleanup phase* is to pack these transactions and obtain more transactions from the input so that the next batch has the desired size. Figure 1 depicts the idea of the cleanup phase. The additional transactions are always obtained by taking a prefix of the unprocessed transactions. Given that the current batch is deterministically executed, the next batch is also deterministic.

3.7 Parameter Selection

The parameter space of *LiTM* mainly consists of the *batch size* and the *lock table size*. A small batch size corresponds

to less exploitable parallelism during the reserve and commit phases, and more overhead during the cleanup phase. A large batch size can better amortize the cost of cleaning up. If the batch size is too large, however, the maintained local RS and WS can consume significant space. Large batches may also increase the conflict rate of transactions within a batch, increasing the abort rate and hurting performance.

The lock table size has a similar tradeoff. A small lock table leads to more hash collisions and increased number of aborts. A large lock table that is too large may not fit in cache leading more expensive data accesses from main memory.

Fortunately, as we show in Section 4.4, *LiTM* can achieve very close to the best performance for a large range of batch sizes and lock table sizes. Therefore, the programmer does not need to devote significant effort to tuning these parameters.

3.8 Variants

In this subsection we discuss some of the design choices and potential variants of *LiTM*.

Handling of the WS. In Algorithm 4, we used an abstract set for the write set data structure. A straightforward implementation of a set is an array, which is what we use in our implementation. However, in transactions that frequently writes to a few places, an array that always appends values at the end may cause a hot element to be duplicated, hurting space efficiency. In this case a small hash table is better as it ensures that only the latest values are recorded in the WS.

Repeated Execution (RE). In Algorithm 4, *LiTM* tracks the entire read set for each transaction to check for conflicts during the commit phase. However, for transactions that have a large read set, this can lead to a large memory footprint. An alternative design is to re-execute the transactions during the commit phase to re-calculate the read set for conflict detection. The re-execution checks the reservations on-the-fly. However, since the system does not know whether it is going to commit or abort until the end of the transaction, all of the writes have to be recorded locally instead of to the memory immediately. So even with repeated execution, *LiTM* still keeps track of the write sets. If the transaction aborts, the write set can be dropped immediately.

Repeated execution also requires Algorithm 5 to be split into two separate phases. Specifically, the check of read and write sets (lines 3–5) has to be performed for all transactions before the write sets are copied to the *DVectors* (lines 7–8). This separation is required because a transaction needs to read the original data during its repeated execution. Updating the shared *DVectors* before all transactions finish repeated execution would pollute the data.

An advantage of the repeated execution variant is its lower memory footprint. In Section 4.5, we will show the memory usage of this variant as well as the performance tradeoff.

4 Evaluation

In this section we compare the performance of *LiTM* with several baselines using the *Problem Based Benchmark Suite* (PBBS) [14, 15]. The experiments are performed on a single machine running Ubuntu 14.04 LTS. The machine has four Intel(R) Xeon(R) E7-4850 CPUs, with a total of 40 cores, and 128 GB of main memory. We turned off hyper-threading to get reliable measurements. When reporting the runtime of experiments, we exclude the time for loading the input files.

Benchmark	Input Files	Input Size
Maximal Independent Set	random	$n = 10^8$, $m = 5 \times 10^8$
	rMat	$n = 10^8$, $m = 5 \times 10^8$
	3DGrid	$n = 10^8$, $m = 3 \times 10^8$
Maximal Matching	random	$n = 10^8$, $m = 5 \times 10^8$
	rMat	$n = 10^8$, $m = 5 \times 10^8$
	3DGrid	$n = 10^8$, $m = 3 \times 10^8$
Spanning Forest	random	$n = 10^8$, $m = 5 \times 10^8$
	rMat	$n = 10^8$, $m = 5 \times 10^8$
	3DGrid	$n = 10^8$, $m = 3 \times 10^8$
Pagerank	random	$n = 10^8$, $m = 5 \times 10^8$
Random Permutation	random	$n = 10^9$
List Contraction	random	$n = 10^9$

Table 1. Benchmarks and Inputs. For the graph inputs, n is the number of vertices and m is the number of edges.

4.1 Workloads

We used six benchmarks from PBBS. The benchmark names, input files, and input sizes are summarized in Table 1. The three input graphs are generated as follows:

Random Local Graphs (random): We generate random local graphs where edges are sampled independently with probability inversely proportional to the difference between vertex IDs. The average node degree of the graph is set to 5.

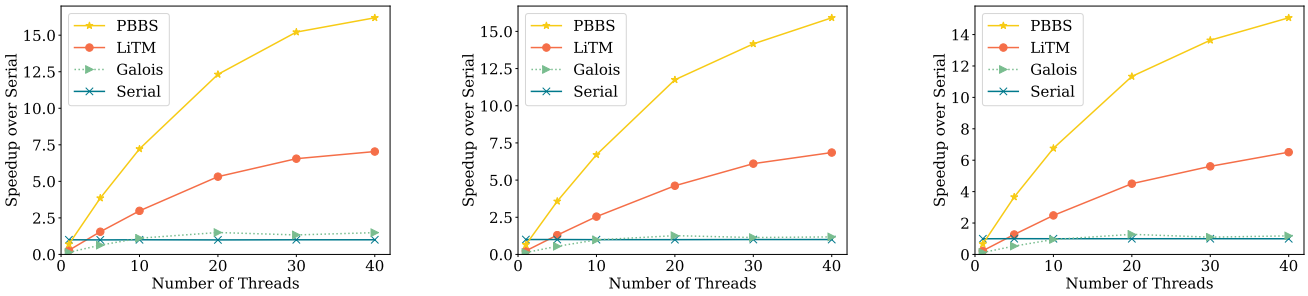
Recursive-Matrix Graphs (rMat): The Recursive-Matrix graphs [3] are synthetic graphs that model a power-law degree distribution. Again, we set the average degree to 5.

3D Grid Graphs (3DGrid): 3D Grid Graphs are graphs where n vertices are placed evenly on a 3-dimensional cube with side-length $n^{1/3}$, and each vertex has edges to its six closest neighbors (two in each dimension).

4.2 Baseline Systems

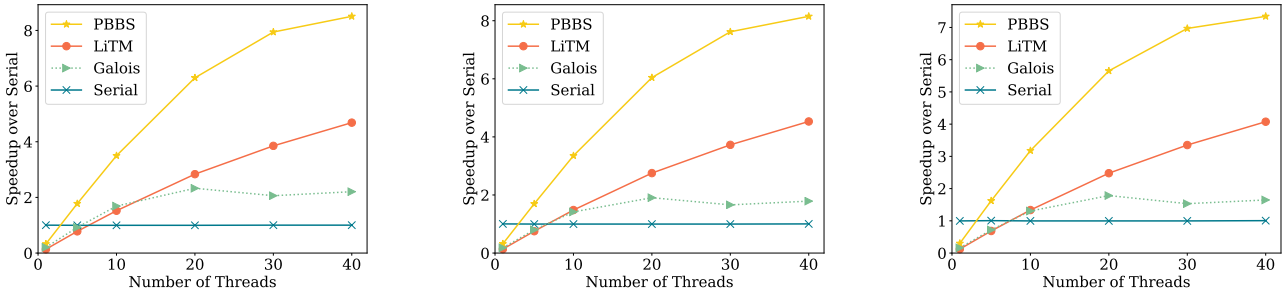
PBBS Implementation. This is a suite of hand-optimized codes released with the Problem Based Benchmark Suite by Shun et al. [14, 15]. The PBBS implementations that we compare with are all deterministic. In contrast to the algorithm example we discussed in Algorithm 2, the PBBS implementations that we used for evaluation include application-specific optimizations. They require the programmer to be fully aware of the problem details as well as be skilled in parallel programming. In contrast, *LiTM* is easier to use due to its transactional interface. By being a general framework, we do not expect *LiTM* to outperform the PBBS implementations. However, later we will show that *LiTM* exhibits only a modest slowdown compared to the hand-optimized codes.

***LiTM* Implementation.** Our implementation of *LiTM* is designed for multicore machines. The underlying parallel for-loops can be compiled with either Cilk Plus [9] or OpenMP [4]



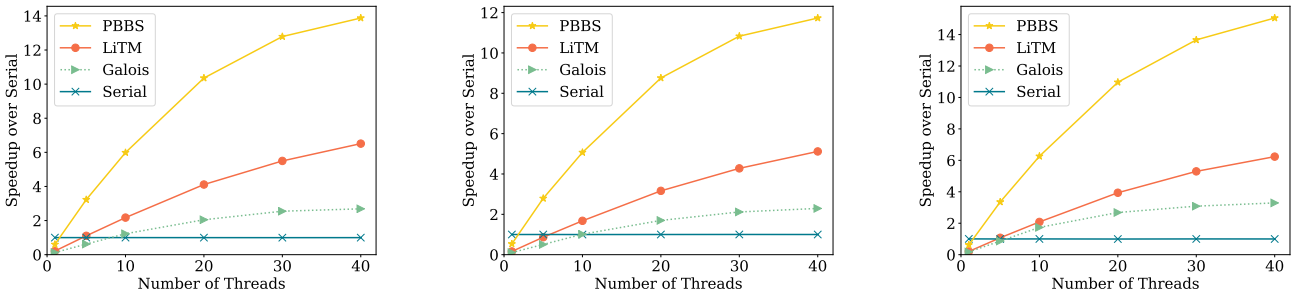
(a) Random **(b)** rMat **(c)** 3DGrid

Figure 2. Performance comparison on *maximal independent set* — Speedup normalized to the serial baseline with varying thread count.



(a) Random **(b)** rMat **(c)** 3DGrid

Figure 3. Performance comparison on *maximal matching* — Speedup normalized to the serial baseline with varying thread count.



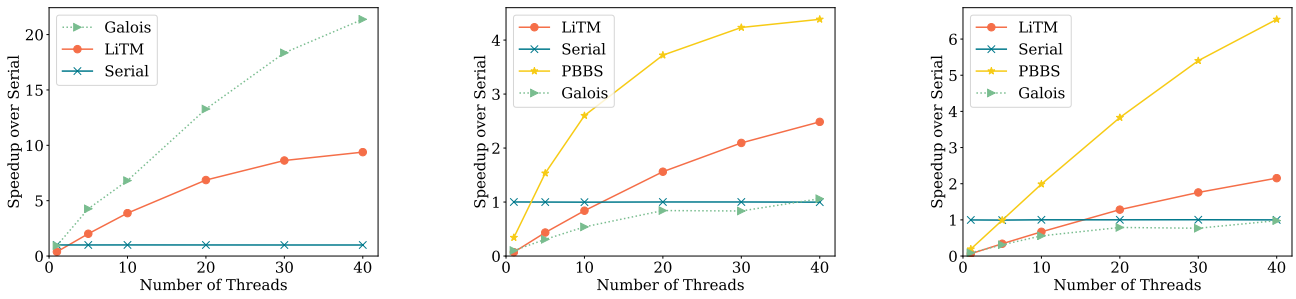
(a) Random **(b)** rMat **(c)** 3DGrid

Figure 4. Performance comparison on *spanning forest* — Speedup normalized to the serial baseline with varying thread count.

(we use Cilk Plus for our experiments). The code base is compiled with the GNU Compiler Collection version 5.4.0 with `-O2` flag. Unless otherwise stated, the lock table size is the number of vertices for the graph algorithms and number of elements for the others, and the batch size is 500,000. We will evaluate the sensitivity to these two parameters in Section 4.4.

Galois Implementation. The Galois implementation is adapted from the default applications shipped with Galois version 2.2.1, which is the latest version that has stable support for deterministic STM. The transactional interface in Galois is more limited than that of *LiTM*. Galois requires transactions to be *cautious*, namely, the transactions must read everything it needs before writes anything. This limitation makes Galois less expressive than *LiTM*, as we will show later in Section 4.3. Galois offers special interfaces which take advantage of the programmer's hints. For example, if the programmer is aware that the transactions do not have any conflicts, he or she can use a special interface to turn off the conflict detection in order to gain extra performance. Such optimizations could also be applied to *LiTM*. To perform a fair comparison, we use the versions with the general Galois deterministic interface unless otherwise specified.

Serial Execution. We compare against a serial implementation of each benchmark without any framework overheads.



(a) *PageRank* **(b)** *random permutation* **(c)** *list contraction*

Figure 5. Performance comparison on *PageRank*, *random permutation*, and *list contraction*. — Speedup normalized to the serial baseline with varying thread count.

Benchmark	*LiTM*	PBBS	Galois
Maximal Independent Set	15	23	48
Maximal Matching	14	26	54
Spanning Forest	34	56	106
Pagerank	73	–	92
Random Permutation	20	27	32
List Contraction	13	22	32

Table 2. Code Length. — Number of lines of code of each benchmark in each of the frameworks.

Table 2 shows a summary of the code lengths of the implementations listed above. We only count the lines in the transaction payload, excluding comments and blank lines. These line numbers show that the *LiTM* programming interface is relatively simple.

4.3 Main Results

Figures 2–5 show the performance of *LiTM*, PBBS, Galois, and the serial code on all of our benchmarks. We report speedup numbers normalized to the serial baseline. For each workload, we increase the number of threads from 1 to 40. From the figures, we can see that *LiTM* scales well for all the benchmarks studied. For *maximal independent set*, *LiTM* achieves 24.7–28.5× speedup on 40 threads compared to its single-threaded performance on the random graph (Figure 2(a)), recursive matrix graph (Figure 2(b)), and 3D grid graph (Figure 2(c)). Compared to the serial baseline, *LiTM* achieves a 6.5–7.0× speedup on 40 threads. *LiTM* is 2.3× slower than the hand-optimized PBBS baseline on all 3 graphs.

For *maximal independent set*, deterministic Galois does not scale well when the number of threads is larger than 20 due to *non-uniform memory access* (NUMA) issues. This scalability result is consistent with Galois' performance reported in [12].

The results are similar for *maximal matching* and *spanning forest*. On the three input graphs, *LiTM* outperforms the serial baseline by 4.1-4.7× and 5.1-6.5× for *maximal matching* and *spanning forest*, respectively. Note that the Galois performance reported in Figure 4 corresponds to the *blocked-asynchronous* version[3] shipped with Galois release 2.2.1. The algorithm is slightly different from the one used in PBBS, where the Union-Find Set (UNS) is explicitly implemented on the shared states. The Galois system provides a special node type *UnionFindNode* to support a hand-optimized lock-free UNS. From Figure 4 we can see that *LiTM* outperforms Galois by up to 1.9–2.4× at 40 threads. The fact that we did not implement the exact same algorithm of *spanning forest* in Galois as in PBBS is due to a limitation of Galois' programming model, where a transaction has to perform all the reads it needs before it writes to anything. The PBBS implementation uses a typical union-find set that performs path compression at the time of finding the root, resulting in reads and writes being interleaved.

We also tested *LiTM* on three additional workloads: PageRank, random permutation, and list contraction. Due to space constraints, we only show their performance on random graphs in Figure 5. We can see that *LiTM* scales well for all three workloads, achieving 24.4×, 34.7×, and 33.6× speedup compared to a single thread for PageRank, random permutation, and list contraction, respectively. Compared to the serial baseline, *LiTM* achieves a speedup of 9.4×, 2.5×, and 2.2×, respectively. *LiTM* achieves 2.2–2.3× faster performance over Galois'.[4] For PageRank, Galois did not provide a version using its general framework, but only a version with the application-specific optimization of turning off conflict detection. As shown in Figure 5(a), the optimized Galois version performs extremely well in our benchmark test for PageRank, achieving over 21.6× speedup against the serial version. *LiTM* is able to achieve about half of the performance using a more general framework (with conflict detection).

Finally, we observe that the performance of *LiTM*, PBBS, and Galois on a single thread is worse than the serial baseline due to overheads in the frameworks and in parallel scheduling.

[3]Galois 2.2.1 comes with 3 versions of *spanning forest*. (1) 'Demo' is a demonstrative baseline that only works on strongly connected graphs. (2) 'Asynchronous' and (3) 'Blocked-Asynchronous' are two Kruskal-like algorithms and the latter is claimed to be an improved version over the former.

[4]For Galois, we excluded the cost in computing the swapping destinations in the random permutation workload because this process is not easily decoupled from the input processing, and so the performance of Galois shown in Figure 5(b) is an upper bound.

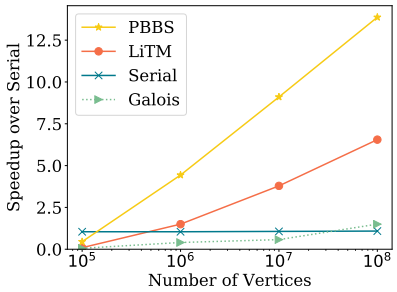


Figure 6. Input Size Sweep — The speedup over the serial baseline improves as the input size increases.

4.4 Sensitivity Study

In this subsection, we perform a sensitivity study on *LiTM* with respect to input size, batch size, and lock table size, to better understand its performance. We use *maximal independent set* as the workload and random local graphs as the inputs for these studies. Three metrics are reported for these results: (1) speedup over the serial baseline; (2) abort rate, which is defined as the ratio of the total number of aborts over the total number of transactions (note that if a transaction is aborted multiple times, it will be counted multiple times in calculating the abort rate); and (3) execution time breakdown of the different phases in *LiTM*.

Input Size. Figure 6 shows the results of *maximal independent set* on varying sizes of the input graph. We vary the number of vertices from 10^5 to 10^8 on the x axis (in log-scale), and set the number of edges to 5 times the number of vertices. We choose the batch size to be 200,000 and the lock table size to be the number of vertices. Unless otherwise specified, they are the default parameters we use in the following experiments. The larger the input is, the better the speedup we get from *LiTM* over the serial baseline. This is due to two factors: (1) larger inputs leads to more parallelism in *LiTM*, and thus higher speedup, and (2) larger graphs make the benchmarks more memory bound, hiding the overhead of the extra computation in *LiTM*.

Batch Size. The batch size is critical to the performance in *LiTM*. The optimal batch size is determined by the intrinsic properties of the problem and the number of threads available. If the batch size is too small, there is not much parallelism that we can exploit within a batch, and hence adding more threads will not help. If the batch size is too large, the system will suffer from (1) higher abort rates due to conflicts among transactions within the same batch, and (2) higher overhead from managing large auxiliary data for the transactions (e.g., local read and write sets of each transaction).

Figure 7(a) shows how the performance change as we increase the batch size from 10^4 to 10^7. We see that the speedup increases consistently when the batch size goes up to 2×10^5. When the batch size increases beyond that, the speedup starts to drop slowly. Such decrease is caused by higher abort rate and system overhead due to larger auxiliary data structures. Figure 7(b) shows that the abort rate increases when the batch size increases. When the batch size is larger, a transaction is more likely to conflict with another. Figure 7(c) shows the normalized time breakdown of *LiTM*. With a larger batch size, the time spent in the cleanup phase is amortized and hence decreases, giving better overall performance. Although the performance varies with batch size, we see that we can achieve close to the best performance for a wide range of batch sizes (i.e., from 10^5–10^7), so this parameter does not require significant tuning.

Lock Table Size. The lock table size is important to the performance as we frequently access the lock table during execution. A smaller lock table will result in more false conflicts caused by hash table collisions. A larger lock table leads to more cache misses which may also hurt performance.

Figure 8 reports how sensitive the performance of *LiTM* is to the size of the lock table. The benchmark is executed with a batch size of 200,000. The x-axis is the ratio of the input graph size (number of vertices) to the lock table size (in number of entries). A larger x-value corresponds to a smaller lock table. The false conflicts caused by a small lock table lead to more unnecessary aborts, which hurts performance. From Figure 8, we observe that the abort rate increases rapidly when the lock table size becomes small. Figure 8(c) shows that the time of the cleanup phase increases as the lock table size decreases because of more aborted transactions.

4.5 Evaluation of Repeated Execution

In this subsection, we evaluate the performance and memory usage of the repeated execution (RE) variant from Section 3.8 compared to *LiTM* without repeated execution.

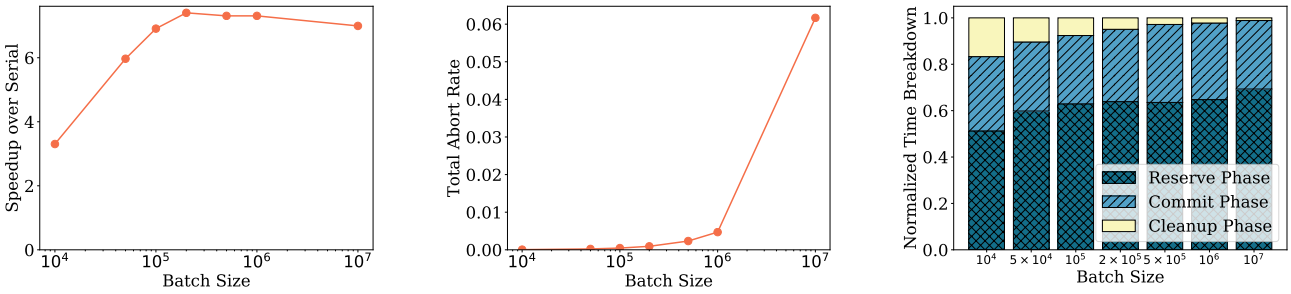
Figure 9 shows that the performance of the repeated execution variant is very close to that of the version without RE for *maximal matching*, where the transactions are small and relatively cheap to re-execute. The memory savings is also negligible due to the small read sets of *maximal matching*, specifically, 2 reads on each edge.

The performance difference is larger for some applications like *maximal independent set*. Figure 10 shows that in the same parameter setting as that in Section 4.3, the RE variant of *LiTM* takes only 4.3 megabytes for the metadata, 81% less compared to *LiTM* without RE, but at the cost of being 11% slower. Note that in this example, every node has 5 neighbors on average, and therefore each time a transaction is executed, it will perform 5 reads on average and one write. For denser graphs, we expect the relative memory savings to increase.

5 Related Work

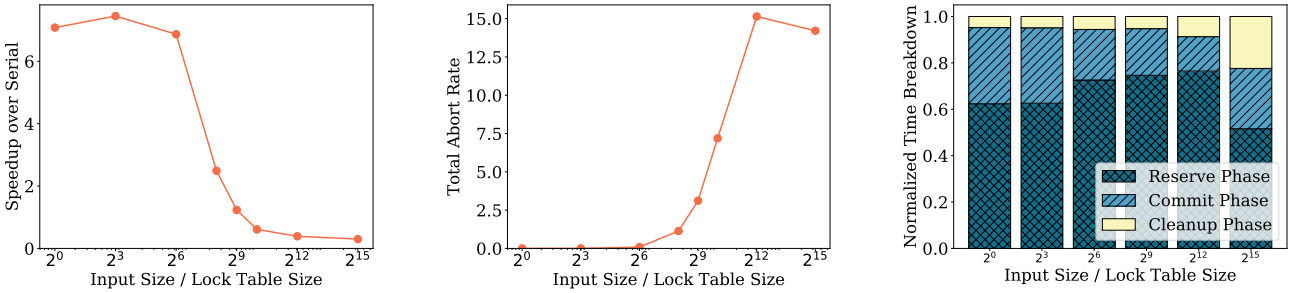
5.1 Deterministic STM

Software Transactional Memory (STM) is a programming model that allows concurrent transactions to access shared states with certain consistency guarantees. Deterministic STM is a special kind of STM that ensures that the interleaving order is deterministic.



(a) Speedup over Serial **(b)** Total Abort Rate **(c)** Time Breakdown

Figure 7. Batch Size Sweep — The speedup over the serial baseline, total abort rate, and time breakdown change as the batch size increases. The x-axis is in log scale.



(a) Speedup over Serial **(b)** Total Abort Rate **(c)** Time Breakdown

Figure 8. Lock Table Size Sweep — The speedup over the serial baseline, total abort rate, and time breakdown change as the lock table size increases. The x-axis is in log scale.

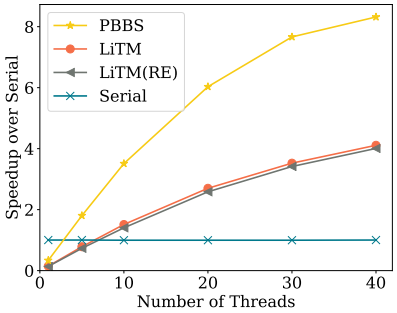


Figure 9. Performance of repeated execution on *maximal matching*.

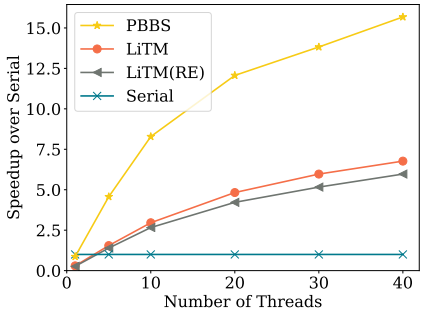


Figure 10. Performance of repeated execution on *maximal independent set*.

DeSTM [13] is the first attempt to non-trivially introduce determinism into an STM system. The system not only provides techniques to reach determinism in general multi-threaded program (the double-barrier technique), but also exploits many properties of STM to potentially remove barriers and improve performance. However, DeSTM shows insufficient performance and scalability [21]. **DeTrans** [16] used similar double-barrier techniques. **DeTrans-lib** [17] leverages the system to a standard library level.

Pot [21] is a system built upon TL2 [5] that pre-orders the transactions and enforces the order with its scheduling algorithm. It executes transactions in two modes, namely the fast mode and the speculative mode, to reduce the system overhead caused by the order enforcement. Pot also uses hardware transactional memory (HTM) to further improve performance. Pot is shown to scale better than prior work when tested on STMBench7 [8], but it still relies on global states in its design to enforce the transaction order. Specifically, it has a global counter, gv. When a transaction t tries to commit, it will wait for the global counter gv to be exactly $wv_t - 1$ and then apply the changes, where wv_t is the pre-ordered sequence number of t. Finally, the transaction updates the global counter gvp to be wv_t so that the next transaction can start to commit. Therefore, the global counter is a contended resource that every thread is accessing frequently.

Galois [11, 12] is a state-of-the-art framework that supports deterministic parallelism. It provides a nested-loop interface, as well as useful tools like reducers and other containers.

5.2 Deterministic Databases

Determinism has also been used in database management systems (DBMS). We highlight the Calvin and Bohm systems.

Calvin [20] is a distributed and replicated DBMS using deterministic execution to enforce consistency across data replicas. In Calvin, commands of transactions are ordered by a sequencer and sent to all replicas, where they are deterministically executed. All replicas are in identical states after the execution. The deterministic model outperforms traditional log shipping due to the elimination of the two-phase commit protocol. The purpose of determinism in Calvin is very different from *LiTM*. Furthermore, Calvin requires the read and write sets of a transaction to be known before execution, which is a strong requirement that *LiTM* avoids.

Bohm [6] is a single-node multi-version DBMS that uses determinism to increase concurrency. Bohm also requires the read and write sets to be known before executing a transaction. Furthermore, conflict resolution is performed using a single thread which is a natural scalability bottleneck. *LiTM*'s use of deterministic reservations avoids this scalability bottleneck.

6 Future Work

Application-specific optimizations are a natural extension of a transactional processing system. Galois's performance in Figure 5(a) shows the power of these optimizations. It would be great if we can automatically exploit such hidden properties in the problem. Static analysis is a potential approach.

Another interesting extension would be supporting dynamically batching online inputs to bypass the requirement of the system that all of the inputs need to be ready before the execution begins. One trivial approach to do this would be to divide the time into rounds with constant length. At the end of each round, the system would process the inputs that arrived in the last round. However, such an approach would result in variable sizes of batches, and small batches may hurt parallelism. Furthermore, this approach would increase the latency of the transactions.

7 Conclusion

We have introduced *LiTM*, a deterministic STM system that achieves both simplicity and efficiency at the same time. *LiTM* implements the *deterministic reservation* paradigm in a novel way, such that the programmer no longer has to manually write functions to handle data conflicts. We show that *LiTM* achieves a better performance than the current state-of-the-art deterministic STM system for six applications, providing a practical choice for deterministic transactional processing.

Acknowledgments

This work is supported by the National Science Foundation Grant No. CCR-1822920, DOE Computational Sciences Graduate Fellowship DE-SC0019323, DOE Early Career Award DE-SC0018947, and DARPA SDH Award HR0011-18-3-0007. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the grant sponsors.

References

[1] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally Deterministic Parallel Algorithms Can Be Fast. In *PPoPP*. 181–192.

[2] Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel Programming Must Be Deterministic by Default. In *Usenix HotPar*.

[3] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *SDM*. 442–446.

[4] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.

[5] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *DISC*. 194–208.

[6] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. *VLDB* (2015).

[7] Jim Gray. 1986. Why do computers stop and what can be done about it?. In *Symposium on Reliability in Distributed Software and Database Systems*. 3–12.

[8] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2007. STMBench7: A Benchmark for Software Transactional Memory. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 315–324.

[9] Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *Journal of Supercomputing* 51, 3 (2010), 244–257.

[10] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. 2012. Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms. In *PPoPP*.

[11] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.

[12] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic Galois: On-demand, portable and parameterless. In *ASPLOS*.

[13] Kaushik Ravichandran, Ada Gavrilovska, and Santosh Pande. 2014. DeSTM: harnessing determinism in STMs for application development. In *PACT*. 213–224.

[14] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. In *SPAA*. 68–70.

[15] Julian Shun, Yan Gu, Guy Blelloch, Jeremy Fineman, and Phillip Gibbons. 2015. Sequential Random Permutation, List Contraction and Tree Contraction are Highly Parallel. In *SODA*. 431–448.

[16] Vesna Smiljkovic, Srdan Stipic, Christof Fetzer, Osman Ünsal, Adrián Cristal, and Mateo Valero. 2014. DeTrans: Deterministic and parallel execution of transactions. In *SBAC-PAD*.

[17] Vesna Smiljković, Osman Ünsal, Adrián Cristal, and Mateo Valero. 2017. Determinism at Standard-Library Level in TM-Based Applications. *International Journal of Parallel Programming* (2017).

[18] Alexander Thomson and Daniel J Abadi. 2010. The case for determinism in database systems. *VLDB* (2010).

[19] Alexander Thomson and Daniel J Abadi. 2011. Building Deterministic Transaction Processing Systems without Deterministic Thread Scheduling. In *WoDet*.

[20] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*. 1–12.

[21] Tiago M Vale, João A Silva, Ricardo J Dias, and João M Lourenço. 2016. Pot: Deterministic transactional execution. *ACM TACO* (2016).