Lee-TM: A Non-Trivial Benchmark Suite for Transactional Memory

Mohammad Ansari, Christos Kotselidis, Ian Watson,
Chris Kirkham, Mikel Luján, and Kim Jarvis

School of Computer Science, University of Manchester
{ansari, kotselidis, watson, kirkham, lujan, jarvis}@cs.manchester.ac.uk

Abstract. Transactional Memory (TM) is a concurrent programming paradigm that aims to make concurrent programming easier than fine-grain locking, whilst providing similar performance and scalability. Several TM systems have been made available for research purposes. However, there is a lack of a wide range of non-trivial benchmarks with which to thoroughly evaluate these TM systems.
This paper introduces Lee-TM, a non-trivial and realistic TM benchmark suite based on Lee's routing algorithm. The benchmark suite provides sequential, lock-based, and transactional implementations to enable direct performance comparison. Lee's routing algorithm has several of the desirable properties of a non-trivial TM benchmark, such as large amounts of parallelism, complex contention characteristics, and a wide range of transaction durations and lengths. A sample evaluation shows unfavourable transactional performance and scalability compared to lock-based execution, in contrast to much of the published TM evaluations, and highlights the need for non-trivial TM benchmarks.

1 Introduction

Concurrent programming is a complex discipline known for its difficulty even to obtain correct programs. Orchestrating lock acquisition and release between multiple threads to ensure functionally correct execution is challenging and time-consuming. Transactional Memory [1, 2] (TM) is an alternative concurrent programming paradigm that promises to abstract away the difficulties of managing access to shared resources, but still maintain good scalability and performance. With the growing need for widespread concurrent programming to take advantage of multi-core processors [3], TM research has surged.

Following from database theory, TM guarantees atomicity, consistency, and isolation among threads accessing shared structures, but abstracts away the details of how these guarantees are achieved. Programmers simply have to annotate those parts of their code that access shared structures as *transactions*, and the TM system automatically detects and manages access conflicts.

Recently, several Software TM (STM) systems have been proposed in the literature that provide sufficient performance for use as research platforms such as DSTM2 [4], McRT-STM [5], RSTM [6], tinySTM [7], and TL2 [8]. However,

there is a lack of non-trivial benchmarks with which to evaluate them, and with which to evaluate novel TM ideas.

Lee-TM is a new non-trivial benchmark suite for TM systems based on the well known Lee's routing algorithm [9] used in circuit routing. Lee's routing algorithm has many of the desirable properties of a non-trivial TM benchmark such as large amounts of parallelism, complex contention characteristics, and a wide range of transaction durations and lengths. Lee-TM provides the following implementations of Lee's routing algorithm: sequential, coarse-grain and medium-grain locking, and transactional and optimized transactional. Lock-based implementations are provided to enable direct performance comparison with transactional versions, and meaningfully measure the benefit of using TM.

The rest of this paper is organized as follows. Section 2 gives an overview of TM and the desirable properties of non-trivial benchmarks. Section 3 describes Lee's routing algorithm, and Section 4 describes the implementations provided by the Lee-TM benchmark suite. Section 5 presents a sample evaluation using a state-of-the-art TM system. Section 6 describes related work, and Section 7 concludes this paper.

2 TM and Non-trivial Benchmarks

TM is a concurrent programming paradigm that aims to make parallel programming as straightforward as programming with coarse-grain locks, but provide the performance and scalability of fine-grain locks. TM requires a programmer to annotate those parts of their code that access shared structures as *transactions*, and an underlying TM run-time automatically detects and manages *access conflicts*. A transaction performs writes on shadow memory as the run-time maintains a *read set* of accessed data, and *write set* of modified data. Access conflicts between concurrently executing transactions occur as read/write or write/write conflicts to shared data, and are detected by the TM run-time by comparing the read and write sets of all transactions. This *validation* of sets can be lazy (at the end of a transaction's execution its sets are compared against all others), or eager (each read or write request is compared as it happens). When a conflict is detected, it is necessary to *abort* (and restart) one of the conflicting transactions. *Contention management* is invoked to make this decision, and there are several contention management policies in the literature [10–12]. Only when a transaction completes execution (i.e. *commits*), are the values in its write set made visible to the rest of the program.

However, there is a lack of complex TM benchmarks with which to evaluate TM systems, and it has been argued [13] that non-trivial, or realistic, benchmarks are needed to further TM research (by studying their execution), and to present the 'real' benefits of TM. Informally, the desirable features of a non-trivial TM benchmark are:

- large amounts of potential parallelism
- difficult to fine-grain parallelize using locks (making TM attractive),

– based on a real-world application (giving confidence in TM),
– several types of transactions (several annotated code blocks),
– complex contention (amount of contention varies widely during execution),
– transactions with a wide range of durations (length), and
– transactions with a wide range of numbers of data accesses (size).

Recently, non-trivial TM benchmarks have become an active research area, and a few non-trivial benchmarks have appeared in the literature [13, 14] that meet many of the characteristics mentioned above, and they are compared with Lee's routing algorithm in the related work (Section 6). Lee's algorithm is presented in the next section.

3 Lee's Routing Algorithm

Circuit routing is the process of automatically producing an interconnection between electronic components. Lee's routing algorithm is attractive for parallelization as realistic circuits consist of thousands of routes, and each one can potentially be concurrently routed. Table 1 presents key terminology used in this paper. Lee's routing algorithm connects a source grid cell to a target grid cell in two phases: *expansion* and *backtracking* (Figure 1). Expansion performs a breadth-first search from the source grid cell until the target grid cell is located, or all cells have been visited. During the search each grid cell is checked that it is not occupied, and then numbered by its distance from the source grid cell. Occupied cells cannot be crossed directly, and routing must divert around them.

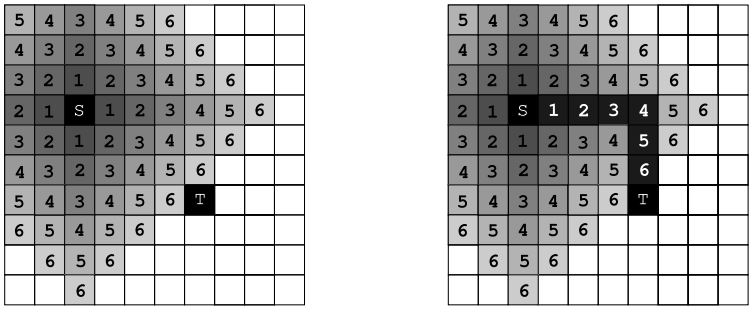
Grid — represents abstractly the final printed circuit board on which all components and routes will be placed. The grid can be multi-layered, permitting a 3D grid
Grid cell — a grid consists of indivisible grid cells.
Grid block — contiguous grid cells can be grouped into grid blocks.
Route — a list of grid cells that connects a source grid cell to a target grid cell.
Obstruction — a predefined grid block inaccessible for routing. Examples are electronic components, mounting holes, servicing areas, etc.

Table 1. Circuit routing terminology.

Backtracking executes if expansion locates the target grid cell. Backtracking starts at the target grid cell and iteratively finds a neighboring grid cell with a lower number than its own and occupies it, until it reaches the source grid cell.

It is usual to perform routing in ascending order of length, i.e. shortest routes first. This ensures that longer routes, which naturally have more alternatives, do not displace shorter ones from their natural positions. This also minimizes the number of unroutable routes; a desirable property for performance comparisons.

In addition, to achieve successful and realistic routing of the example circuits, a certain amount of refinement in both the expansion and backtracking phases



Expansion phase from source grid cell S to target grid cell T.

Backtrack phase connecting target grid cell T to source grid cell S.

Fig. 1. Illustration of expansion and backtracking in Lee's routing algorithm.

of the algorithm have been added. These are concerned with constraining the routes in certain ways so that the routing does not generate a 'spaghetti' layout, and their detail is omitted in this paper.

4 Lee-TM

Lee-TM is a benchmark suite that has five implementations of Lee's routing algorithm: sequential, coarse-grain and medium-grain lock-based, and transactional and optimized transactional. They are named Lee-TM-seq, Lee-TM-cg, Lee-TM-mg, Lee-TM-t, and Lee-TM-ter, respectively, and are described below.

4.1 Sequential (Lee-TM-seq)

First, the source and target grid cell coordinates of each route, and coordinates for each obstruction, are read from a file. The obstructions are marked on the grid immediately, whilst the source-target pairs are added to a work queue. The work queue is then sorted in ascending route length order, as motivated in Section 3.

The main program loop gets a route from the work queue by calling the function `getNextRoute()`, and then performs expansion and backtracking with `layNextRoute()`. Expansion is performed by reading from a *main grid* and writing the expansion values on a private *temporary grid*. If the expansion is successful, the values in the temporary grid are used in backtracking, which writes to the main grid. The program finishes when the work queue is empty.

4.2 Concurrent Implementations

Minimal changes are required to make Lee-TM-seq multi-threaded. Each thread needs its own temporary grid, and the work queue needs to be synchronized to ensure multiple threads do not get the same route. The single work queue could become a bottleneck, but the experiments have not yet shown contention in its

access. Nonetheless, a future version of the benchmark will decentralize the work queue. Finally, access to the main grid needs to be kept consistent, and this is explained separately for each concurrent implementation below.

Coarse-Grain Lock-based (Lee-TM-cg) Lee-TM-cg is simple: all threads serialize on access to `layNextRoute()`. This prevents the main grid from being read by a thread (expansion) while another thread is modifying it (backtracking), which could lead to a race condition.

Medium-Grain Lock-based (Lee-TM-mg) Lee-TM-mg splits the main grid into as many equal-sized grid blocks as there are threads and associates a lock with each grid block. For each route, if the source and target coordinates are located in the same grid block, then the associated lock is requested, and routing is performed. If the source and target coordinates are in different grid blocks, then multiple alternatives are available.

A complex alternative for routes that span multiple grid blocks is to acquire locks for all the necessary grid blocks. A priori, it is impossible to know which grid blocks may be needed, thus requiring progressive lock acquisition. Without a careful lock acquisition/release protocol in place, threads will deadlock. This approach is applicable to fine-grain locking, and quickly shows how challenging that would be to implement (consequently making TM attractive).

Instead, Lee-TM-mg adopts a simpler alternative where routes that do not fit in a single grid block are added to a *deferred work queue*. Once the main work queue is exhausted, the grid blocks are re-sized such that there are half as many as before, and the deferred work queue is swapped with the main queue. As grid blocks double in size at each swap of work queues, more routes can be laid. This reduction of grid blocks continues until there is only one grid block for the whole grid, at which point any existing route will definitely be routed or discarded (as unroutable), albeit serially.

Transactional (Lee-TM-t) There is something naturally transactional about circuit routing. Each route can be treated as an independent transaction. Each routing transaction can perform its own expansion, backtrack, and then try to commit the route it has found. If any of the grid cells used by the route have concurrently been occupied and committed by another route, then the transaction must be abandoned and restarted. However, it is important to realize that now the detection of interference, abandonment and restarting are fundamental functionality provided by TM. There is no need to program safe access to the main grid explicitly as is required with the previous lock-based implementations.

Lee-TM-t is implemented in DSTM2 [4], a state-of-the-art Java STM implementation. DSTM2 transactional semantics require concurrently accessed data to be annotated as transactional data. Since DSTM2 offers object-level conflict detection the main grid was changed from a three dimensional primitive array into a three dimensional transactional object array. This was the only change needed to provide the equivalent of fine-grain locking, but using transactions.

Optimized Transactional (Lee-TM-ter) Lee-TM-ter extends Lee-TM-t. Watson *et al.* [15] studied Lee's routing algorithm in an abstracted TM environment. Their key insight was understanding that the expansion phase adds unnecessary data to the read set, and that a transaction that generates a complete route between a source point and a destination point simply needs in its read set those grid cells that identify the complete route; no more, no less. They suggested that using early release [16], which removes data from the read set before any validation occurs, would optimize the read set and lead to dramatically more exploitable parallelism. The optimized transactional implementation provided by Lee-TM implements this approach.

4.3 Verifier

Lee-TM includes a verifier to check that all successful routes exist on the grid when routing is complete. A verification error suggests either an error in the code (if it has been changed) or in the TM system if executing transactional implementations. This feature is useful when evaluating novel TM ideas, as subtle errors in the TM system can be difficult to recognize from the, often large, execution output of a non-trivial benchmark that has no verifier.

5 Workload Characterization

A sample evaluation using a state-of-the-art TM system, called DSTM2 [4], is presented to highlight the value of Lee-TM. A discussion of the performance results is presented, followed by an investigation of the transactional characteristics of Lee-TM's transactional implementations to explain the observed performance.

5.1 Experimental Environment

The lock-based and transactional implementations provided by Lee-TM are compared using one synthetic and two real circuit boards (Figure 2) — each circuit is of size 600×600, and two layers. The workload characterization is performed on a shared memory 8-core (i.e. 4 dual core) Opteron 2.4GHz machine running openSUSE 10.1 64-bit and Sun JDK 1.6 64-bit with flags -Xms1024m -Xmx4096m, with 2, 4, or 8 threads. The transactional implementations are executed using all contention management policies [10–12] provided with DSTM2, but results are only shown for the Priority contention manager, as it had the best performance overall. The Priority manager aborts younger transactions, i.e. those with the most recent start time. Experiments are run three times and the best results reported.

5.2 Sample Performance Evaluation

The first experiment employs a trivial synthetic circuit layout, shown in Figure 2a. It contains 841 sparsely spaced short routes with no overlaps, i.e. 841 transactions to commit, with no possible contention. The aim of this experiment is to

present baseline performance and scalability that can be achieved by lock-based and transactional implementations.

Figure 2b shows the execution time to route the circuit *simple*. For Lee-TM-cg there is no speedup regardless of the number of threads used. This occurs since the the coarse-grain lock on the whole main grid effectively leads to serial execution. Lee-TM-mg shows poor results initially, but has a speedup of 2.5 from 2 to 8 threads, and surpasses coarse-grain performance at 8 threads. Lee-TM-mg shows poorer than expected results because the routes, all having identical length, are ordered top-left to bottom-right, thus all threads are usually attempting to acquire the same grid block lock. The transactional implementations perform best in all cases except at two threads, and scale well with a 3.4 fold speedup from 2 to 8 threads.

Figures 2c and 2e show two complex circuit called *main* and *mem*, respectively. Both are microcode microprocessor layouts consisting of 1506 and 3101 routes, i.e. transactions to commit, respectively, and were used in routing algorithm research. The layouts contain a rich variety of route lengths and overlaps. Their execution times are shown in Figures 2d and 2f, respectively.

Lee-TM-cg again shows no scalability regardless of the number of threads. Lee-TM-mg has better initial performance than with the circuit *simple* because it is less prone to the problem of route ordering, but shows worse scalability with a speedup of only 1.2 from 2 to 8 threads. However, the most significant outcome is that of the transactional implementations.

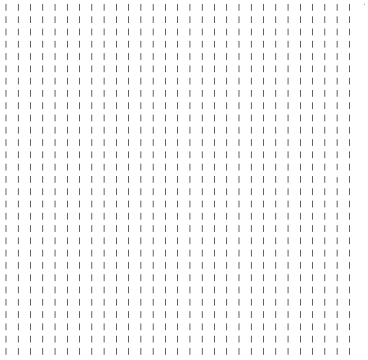
Lee-TM-t is consistently worse than both lock-based implementations by a large margin. Even in the best case (at 2 threads) it is 3.8 times slower than Lee-TM-mg, and the scalability is far worse than seen with the circuit *simple*, with a speedup of only 1.16 from 2 to 8 threads. Lee-TM-ter performs 2-3 times better than Lee-TM-t, and has a speedup of 2.15 from 2 to 8 threads, but is only on par with Lee-TM-cg performance at 8 threads.

It is obvious that transactional performance and scalability seen in the circuit *simple* has not been realized with the two more complex circuits. To better understand the losses in performance, the next section analyzes profiled data to characterize the transactional behavior of Lee's routing algorithm.

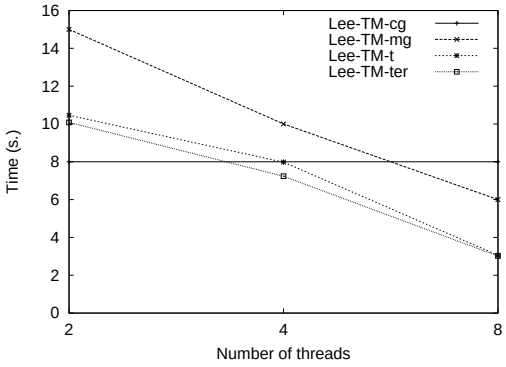
5.3 Analysis of the Transactional Profile

Figure 3a shows the ratio of aborts to commits for each experiment. The experiment with the circuit *simple* has no aborts by design, but the other two complex circuits have an increasing ratio of aborts as the number of threads rises. The benefit of early release is obvious as the ratio of aborts to commits falls dramatically for both *main* and *mem*, re-emphasizing the benefit of early release as concluded by Watson et al. [15].

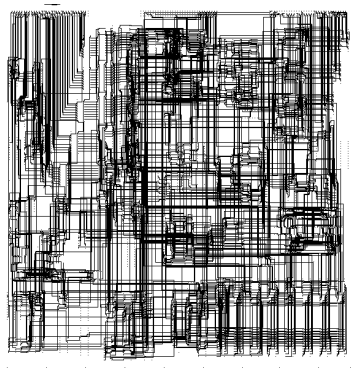
Figure 3b shows the percentage of time spent executing wasted work (aborted transactions). Both *main* and *mem* show increasing amounts of wasted work as the number of threads rises, but the wasted work for Lee-TM-t increases by greater amounts. This helps explain difference in execution time between the two implementations: Lee-TM-t spends more time executing aborted transactions.



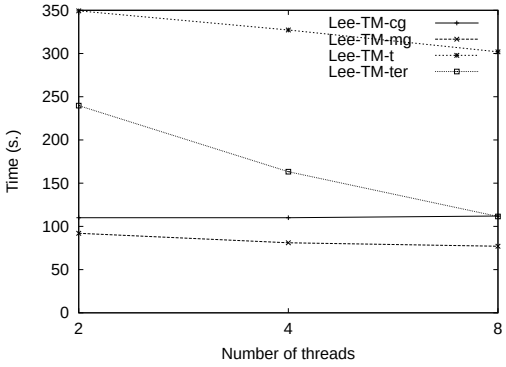
(a) Circuit *simple*.



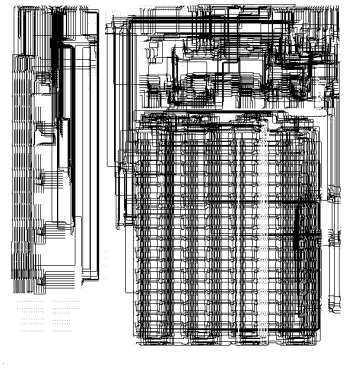
(b) Circuit *simple* execution times.



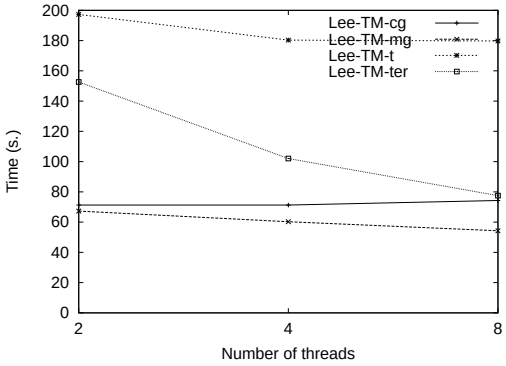
(c) Circuit *main*.



(d) Circuit *main* execution times.

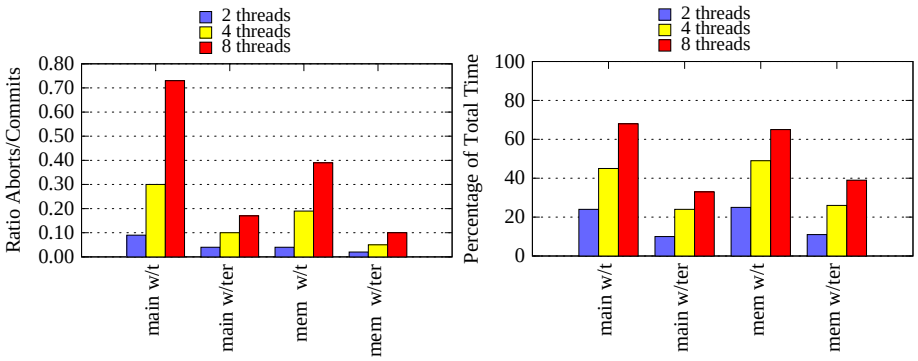


(e) Circuit *mem*.



(e) Circuit *mem* execution times.

Fig. 2. Circuits used in the sample evaluation, and their execution times using lock-based and transactional implementations.



a) aborts per commit. b) wasted work.

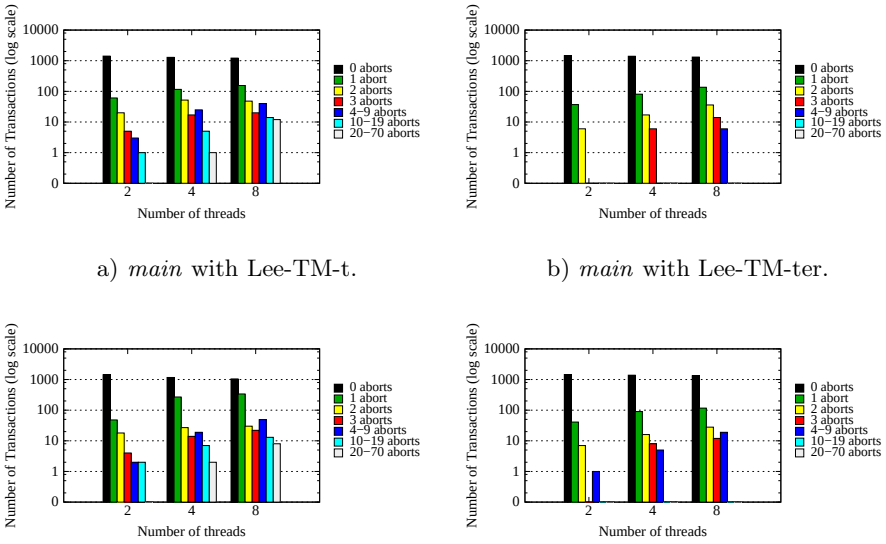
Fig. 3. Transactional profiling data: a) shows the ratio of aborted transactions to committed transactions, b) shows the percentage of total execution time spent executing aborted transactions (wasted work). Circuit *simple* is not shown as it has no aborts or wasted work. Note: Lee-TM-t results have 'w/t' suffix, and Lee-TM-ter results have 'w/ter' suffix

Since the ratios in Figure 3a correlate to those in Figure 3b, there may be benefit in attempting to detect doomed (to abort) transactions sooner, and abort them early to reduce the amount of wasted work, and improve execution time.

Figure 4 shows the abort histograms for the circuits *main* and *mem* (the histograms for the circuit *simple* have been omitted as it has no aborts). These graphs show the count of routes aborted by a given number before finally committing, and present perhaps the most interesting results because the histogram for Lee-TM-t indicates that a few routes take tens of aborted attempts before committing. For circuit *main* at 8 threads, Lee-TM-ter commits all routes within nine aborts each, while Lee-TM-t commits 26 routes with more than nine aborts each (the abort profile of the circuit *mem* shows similar statistics). Although this represents 1.7% of the routes (i.e. workload), it is almost solely responsible for a 35% difference in wasted work between Lee-TM-t and Lee-TM-ter for the circuit *main* at 8 threads. The large amount of aborts experienced by a small number of transactions is another sign that the contention manager could be enhanced to make better decisions, and thus may be an avenue to explore in future work to reduce the amount of wasted work.

	simple	*main*	*mem*
2 threads	41	453	288
4 threads	41	342	226
8 threads	41	267	190

Table 2. Ratio of avg committed transaction read set size in Lee-TM-t to Lee-TM-ter.



a) *main* with Lee-TM-t.

b) *main* with Lee-TM-ter.

c) *mem* with Lee-TM-t.

d) *mem* with Lee-TM-ter.

Fig. 4. Abort histograms of routes in complex circuits *main* and *mem*. Circuit *simple* is not shown as it has no aborts.

Finally, Table 2 shows the ratio of the average committed transaction read set size in Lee-TM-t to Lee-TM-ter. The benefit of early release is significant in the execution of all three circuits, resulting in Lee-TM-ter's read set shrinking by at least 41 times over Lee-TM-t's read set. Reducing the read set results in faster execution time as it reduces cache thrashing.

6 Related Work

The lack of complex benchmarks for studying TM is a known issue. The majority of benchmarks used in the evaluation of TM systems fit into the following categories:

- micro-benchmarks, such as linked lists, and concurrent hash tables;
- benchmarks whose parallelism is already explicit and optimized, such as JavaGrande, and SPECjbb; and
- benchmarks with limited concurrency, such as SPEC JVM98.

Apart from Lee-TM, few other benchmarks provide complex transactional behavior. Those seen in the literature are STMBench7 [13], and the STAMP benchmark suite [14]. STMBench7 is adapted from OO7, a database benchmark, and has over 5000 lines of code. It simulates real-world scenarios by performing dynamic and complex modifications and traversals on a non-trivial shared

data structure. STMBench7 provides a coarse-grain and medium-grain locking implementation for comparison with the transactional one. The STAMP benchmark suite consists of three benchmarks: genome (gene sequencing), kmeans (k-clustering), and vacation (travel booking system). Each of these is over 1000 lines of code, and is supplied with a transactional and a sequential implementation, but no lock-based solution. Lee-TM has a smaller code base (<800 lines), yet provides complex transactional behavior through the complex circuits employed for routing. Lee-TM has sequential, coarse-grain and medium-grain locking, transactional, and optimized transactional (using early release) implementations.

Both STMBench7 and STAMP benchmarks (except for genome), due to the nature of their computation, lack a verifier as there is no simple way to validate the final data structure. Lee-TM comes with a verifier since it is easy to use the original circuit layout data set and follow, for each route, from the source grid cell to the target grid cell.

7 Summary

Lee-TM is a new benchmark suite based on Lee's routing algorithm with sequential, lock-based, and transactional implementations. Lee's routing algorithm provides many of the desirable properties of a non-trivial TM benchmark through complex circuit layouts, such as large amounts of parallelism, complex contention behavior, and large variety of transaction durations and sizes.

A sample performance evaluation using complex circuits, which had potential parallelism in the thousands, showed optimized transactional performance only reaching par with coarse-grain locking at 8 threads, and never reaching the performance of medium-grain locking. Unoptimized transactional execution was, in the best case, four times slower than medium-grain locking. This result highlights the need for complex benchmarks to stress TM systems.

The analysis of the transactional characteristics of Lee-TM's transactional implementations showered there is much work wasted in executing doomed (to abort) transactions. At 8 threads, less than 2% of the transactions, due to being aborted tens of times, resulted in unoptimized transactional execution having 35% more wasted work than the optimized transactional execution, and consequently 2.7x slower execution time. The analysis identified contention management as a target of future research to make better decisions that result in less wasted work, and thus better performance.

References

1. Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, August 1995.
2. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. ACM Press, May 1993.

3. Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–29, September 2005.
4. Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 253–262. ACM Press, October 2006.
5. Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197. ACM Press, March 2006.
6. Virendra Marathe, Michael Spear, Christopher Herio, Athul Acharya, David Eisenstat, William Scherer III, and Michael Scott. Lowering the overhead of software transactional memory. In *TRANSACT '06: First ACM SIGPLAN Workshop on Transactional Computing*, June 2006.
7. Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*. LNCS, Springer, September 2006.
8. David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*. LNCS, Springer, September 2006.
9. F. Rubin. The Lee path connection algorithm. *IEEE Transactions on Computers*, C-23(9):907–914, 1974.
10. William Scherer III and Michael Scott. Contention management in dynamic software transactional memory. In *CSJP '04: Workshop on Concurrency and Synchronization in Java Programs*, July 2004.
11. William Scherer III and Michael Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 240–248. ACM Press, July 2005.
12. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 258–264. ACM Press, July 2005.
13. Rachid Guerraoui, Michał Kapałka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. In *EuroSys '07: Proceedings of the 2nd European Systems Conference*, pages 315–324. ACM Press, March 2007.
14. Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 69–80. ACM Press, June 2007.
15. Ian Watson, Chris Kirkham, and Mikel Luján. A study of a transactional parallel routing algorithm. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 388–400. IEEE Computer Society Press, September 2007.
16. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101. ACM Press, July 2003.