Profiling Transactional Memory Applications

Mohammad Ansari, Kim Jarvis, Christos Kotselidis, Mikel Luján, Chris Kirkham and Ian Watson
School of Computer Science, University of Manchester
Email:{ansari, kotselidis, jarvis, mikel, chris, watson}@cs.manchester.ac.uk

***Abstract*—Transactional Memory (TM) has become an active research area as it promises to simplify the development of highly scalable parallel programs. Scalability is quickly becoming an essential software requirement as successive commodity processors integrate ever larger numbers of cores. Non-trivial TM applications to test TM implementations have only recently begun to emerge, but have been written in different programming languages, using different TM implementations, making analysis difficult.**

We ported the popular non-trivial TM applications from the STAMP suite (Genome, KMeans, and Vacation), and Lee-TM to DSTM2, a software TM implementation, and built into it a framework to profile their execution. This paper investigates which profiling information is most relevant to understanding the performance of these non-trivial TM applications using up to 8 processors. We report commonly used transactional execution metrics and introduce two new metrics that can be used to profile TM applications.

I. Introduction

Transactional Memory (TM) [1], [2] is a promising concurrent programming abstraction that makes it easier to write scalable parallel programs. It aims to provide the scalability of fine-grain locking, but with the programming ease of coarse-grain locking. TM has seen a rise in research activity as the demand for scalable software increases in order to take advantage of future chip multiprocessors [3].

TM requires a programmer to mark code blocks that access shared data as *transactions*. Whenever a transaction executes, a runtime system records the transaction's data accesses into a *readset* and a *writeset*. These sets are compared with the sets of other concurrently executing transactions for access conflicts (write/write or read/write). If conflicting accesses are detected then one of the conflicting transactions is *aborted* and restarted. A *contention manager* [4], decides which transaction to abort. A transaction that completes execution of its code block without being aborted can *commit* its writeset. TM implementations exist in a variety of flavors, including software-based (STM), hardware-based (HTM), and hardware/software hybrids (HyTM), and readers can refer to Larus and Rajwar [5] for details.

Several non-trivial programs specifically designed for parallelization with TM have appeared recently [6], [7], [8], [9], but have been written in different programming languages, using different TM implementations, which has made it difficult to profile them.

This paper puts under the same umbrella, our TM profiling framework, the STAMP [6] applications Genome, Vacation and KMeans, and Lee-TM [7], [10]. These applications have become popular amongst researchers, as evidenced in recent publications [11], [12], [13], [14], [15], [16]. The applications have been ported to a single TM implementation, the Java-based Software TM (STM) called DSTM2 [17], which has been extended with a TM execution profiling framework. Executing the non-trivial TM applications in a single TM implementation allows an investigation of the relation between the profiling information gathered, and the performance of the applications.

This paper also reports two new transactional metrics that have not been used in previous related work [18], [19]: *running percentage commit rate*, and *transaction execution time histogram* (defined in Section III).

This paper is organized as follows: Section II introduces the applications profiled in this paper. Section III explains and motivates the metrics used to investigate the transactional behavior of the applications. Section IV walks through the different performance figures and execution characteristics. Section V introduces related work, and Section VI concludes the paper with observations of the recorded TM behavior.

II. Non-trivial TM Applications

Recently, several research groups have been working towards building non-trivial TM applications for thorough TM implementation analysis [6], [7], [8], [9]. Non-trivial TM applications are important for TM research as they allow performance analysis of TM implementations in realistic scenarios.

A. Analyzed Applications

This paper analyzes STAMP version 0.9.5 applications Genome, KMeans, and Vacation [6], and Lee-TM [7], [10]. STAMP applications have been ported from C to Java, and converted from using TL2 [20], another STM, to DSTM2 [17]. STAMP applications also required the implementation of additional utility classes in DSTM2: transactional implementations of a linked list, hash table, and hash map. Lee's routing algorithm, originally in Java and single-threaded, has been implemented using transactions in DSTM2, with the transactional version named Lee-TM. The remainder of this section briefly describes each application.

Genome is a gene sequencer that rebuilds a gene sequence from a large number of equal-length overlapping gene segments. Each gene segment is an object consisting of a character string, a link to the start segment, next segment, and end segment, and overlap length. The application executes in three phases. The first phase removes duplicate segments by transactionally inserting them into a hash set. The second

phase attempts to link segments by matching overlapping string subsegments. If two segments are found to overlap then linking the two segments (by modifying the links in each gene segment object, and setting the overlap length) and removing them from the hash set is done transactionally, as multiple gene segments may match and result in conflict. The matching is done in a for-loop that starts by searching for the largest overlap (length-1 characters, since duplicates were removed in the first phase), down to the smallest overlap (1 character). Thus, conflict is likely to rise as execution progresses since smaller overlaps will lead to more matches. In the third phase, a single thread passes over the linked chain of segments to output the rebuilt gene sequence. The execution of Genome is completely parallel except for the third phase.

KMeans clusters objects into a specified number of clusters. The application loads objects from an input file, and then works in two alternating phases. One phase allocates objects to their nearest cluster (initially cluster centers are assigned randomly). The other phase re-calculates cluster centers based on the mean of the objects in each cluster. Execution repeatedly alternates between the two phases until two consecutive iterations generate, within a specified threshold, similar cluster assignments. Assignment of an object to a cluster is done transactionally, thus parallelism is controlled by the number of clusters. Execution consists of the parallel phase assigning objects to clusters, and the serial phase checking the variation between the current assignment and the previous.

Vacation simulates a travel booking database in which multiple threads transactionally book or cancel cars, hotels, and flights on behalf of customers. Threads can also execute changes in the availability of cars, hotels, and flights transactionally. Each customer has a linked list holding his reservations. The execution of Vacation is completely parallel, but available parallelism is limited by the number of relations in the database and the number of customers.

Lee-TM is a circuit router that makes connections automatically between points. Routing is performed on a 3D grid that is implemented as a multidimensional array, and each array element is called a grid cell. The application loads connections (as pairs of spatial coordinates) from an input file, sorts them into ascending length order (to reduce 'spaghetti' routing), and then loads them into thread-local queues in a round-robin manner. Each thread then attempts to find a route from the first point to the second point of each connection by performing a breadth-first search, avoiding any grid cells occupied by previous routings. If a route is found, backtracking lays the route by occupying grid cells. Concurrent routing requires writes to the grid to be performed transactionally. Lee-TM is fully parallel, with conflicts at concurrent read/write or write/write accesses to a grid cell. A second version of Lee-TM has been implemented that uses early release [4]. This version removes grid cells from the readset during the breadth-first search. Two transactions may be routable in parallel, i.e. the set of grid cells occupied by their routes does not overlap, but because of their spatial locality, the breadth-first search of one transaction reads grid cells to which the second transaction writes its route, thus causing a read/write conflict. Removing grid cells from the readset during the breadth-first search eliminates such false-positive conflicts.

III. Analyzed Metrics

We instrumented the DSTM2 STM to collect execution data from the execution of the applications. We present those metrics commonly used to characterize applications in the TM literature, and introduce two new metrics not seen in the TM literature; the transaction execution time histograms and the Instantaneous Commit Rate (ICR).

Speedup is presented to show how well the applications scale with increasing number of threads, and is a measure of the effectiveness of the transactional execution of the applications. The speedup depends on characteristics of both the application and the TM implementation. In this paper we keep the TM implementation constant, and the metrics presented in this paper are intended to characterize the application and help us understand why linear speedup is not achieved. Note that single thread execution times include transactional overheads in the results reported.

In transactions (InTX) is the percentage of total time the applications spent executing transactions. For the applications studied, the remaining percentage of time is spent executing serial code. A high InTX means an application spent most of its time executing transactions, thus possibly stressing the TM implementation more than an application with low InTX.

Wasted work shows the percentage of transaction execution time spent executing transactions that subsequently aborted. It is calculated by dividing the total time spent in aborted transactions by the time spent in all (committed and aborted) transactions. High amounts of wasted work can be an indicator for poor contention management decision-making, low amounts of parallelism in the application.

Aborts per Commit (ApC) shows the mean aborted transactions per committed transaction. ApC is not directly related to wasted work, but is an indicator for the same issues mentioned for wasted work. For example, high wasted work in combination with a low ApC (aborting a few long/large transactions, and favoring many short/small transactions) may indicate poor contention management decision-making, and studying the application may lead to better contention management policies.

Abort histograms detail how the ApC is spread amongst the transactions; e.g. is the ApC due to a minority of transactions aborting many times before committing, or vice versa?

Contention Management Time (CMT) measures the percentage of time the mean committed transaction spends in performing contention management when conflicts are detected. In combination with wasted work and abort histogram data, it is possible to understand which contention manager may be most effective for the profiled application.

Transaction execution time histograms show the spread of execution times of committed transactions. This metric describes how homogeneous or heterogeneous is the amount of work contained in transactions for a given application.

Configuration Name	Application	Configuration
Gen	Genome	gene length:16384, segment length:64, number of segments:4194304
KMeansL	KMeans low contention	min_clusters:40, max_clusters:40, threshold:0.00001, input_file:random10000_12
KMeansH	KMeans high contention	min_clusters:20, max_clusters:20, threshold:0.00001, input_file:random10000_12
VacL	Vacation low contention	relations:65536, %_of_relations_queried:90, queries_per_transaction:4, number_of_transactions:1024768
VacH	Vacation high contention	as above, but %_of_relations_queried:10, queries_per_transactions:8
Lee-TM-t	Lee w/o early release	early_release:false, input_file:mainboard.txt
Lee-TM-ter	Lee with early release	early_release:true, input_file:mainboard.txt

TABLE I
APPLICATION PARAMETERS USED TO GATHER EXECUTION CHARACTERISTICS.

Instantaneous Commit Rate (ICR) graphs show the proportion of committed transactions at sample points during the execution of the application. ICR includes only completed, i.e. committed or aborted, transactions, and does not include active transactions. Low ICR is indicative of wasted work.

Readset & writeset sizes are a measure of the memory-boundedness of committed transactions in an application. They can be used for selecting buffer or cache sizes for Hardware TM (HTM) implementations [5]. Data from non-trivial TM applications gives higher confidence that the hardware will not overflow for a large proportion of transactions. In this work a writeset is always a subset of its corresponding readset because all applications first read data before writing. For other applications, these sets may only overlap, or be distinct.

Readset-to-writeset ratio (RStoWS) shows the mean number of reads that lead to a write in a committed transaction. Execution usually involves reading a number of data elements, performing computation, and writing a result to a data element.

Writes-to-writeset ratio captures the mean number of writes to a transactional data element in a committed transaction. Multiple writes indicate further refinement in the application, when using a STM, is possible since only the last write is valid, and all other writes add runtime system monitoring overhead. The data for this metric is omitted for the applications studied as they do not exhibit multiple writes to a transactional data element.

Reads-to-readset ratio captures the mean number of reads to a transactional data element in a committed transaction. Reading transactionally shared data incurs extra costs, and a high RtoRS ratio, when using a STM, indicates the need to study the implementation and remove multiple reads to the same transactional data element. For compilers, it describes an upper limit of how many read operations can be optimized away by not recording them again and again. For brevity this data is omitted as only Lee-TM was found to have a high ratio, and this information is also visible in the RStoWS ratio data because none of the applications performed multiple writes to a transactional data element.

IV. PROFILING RESULTS

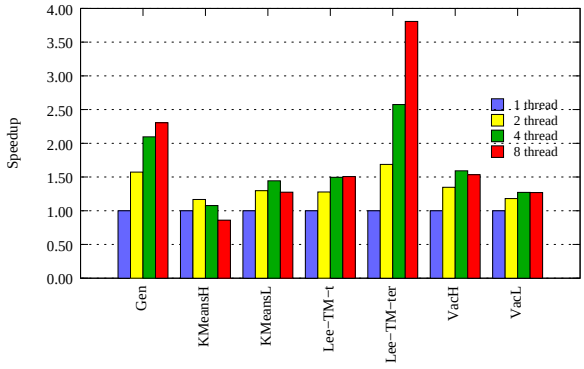
All experiments are performed on a 4 x dual-core 2.2GHz Opteron-based (i.e. an 8-core NUMA shared memory) machine with openSUSE 10.1, 16GB RAM, and using Sun Java 1.6.0 64-bit. The default configuration of DSTM2 is used: shadow atomic factory, visible readers, and eager conflict detection [17].

Table I shows the configurations executed for each application. STAMP applications are executed using the parameters suggested in the guidance notes supplied with the suite. Lee-TM is executed using the default dataset (a real circuit) with and without early release [4]. Each experiment is repeated twenty times and the mean results presented.

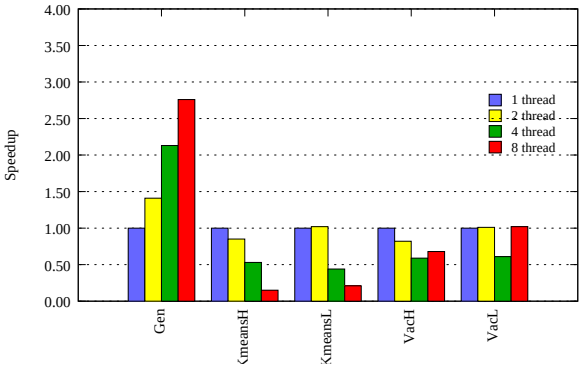
All popular contention managers [21], [22], [23] have been used, but only results with the *priority manager* are presented as it generally gives some of the best execution times in these benchmarks. The priority manager aborts younger transactions (those with a later start time). Execution time is measured from the point where multiple threads start executing transactions to the point where they stop executing transactions, thus excluding any setup time (e.g. of application data structures) and any shutdown time (e.g. of validating results, reporting metrics). We begin by presenting the speedup results, and to aid readability the discussion of each metric has been demarcated.

Fig. 1a illustrates the speedup of each of the target applications in the DSTM2 STM. Each application is executed with 1, 2, 4, and 8 threads. Speedup for STAMP applications Gen and Vac are similar to published results [6], but speedup for KMeans is significantly lower. Note that we are using ported versions of these applications (to Java/DSTM2), and using a different hardware platform than used in the published results. To allow direct comparison on our hardware platform, Fig. 1b illustrates the speedup of the original (i.e. C/TL2) Gen, Vac, and KMeans applications, and also shows that KMeans speedup is significantly lower than published results.

Fig. 2a shows InTX results. Readers are reminded transac-

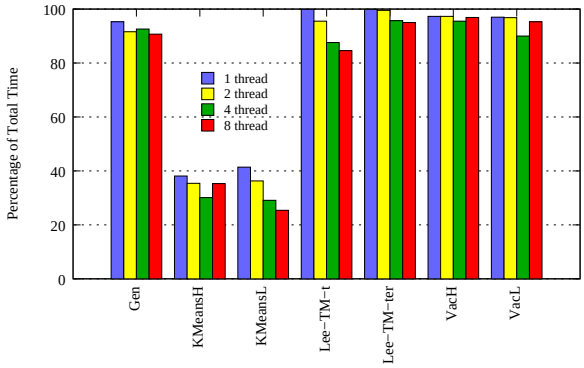


(a) Speedup of the applications with DSTM2.

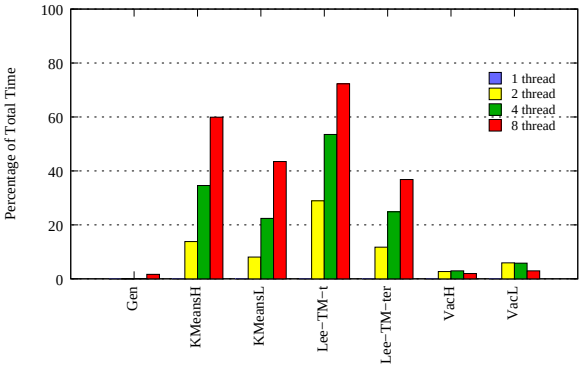


(b) Speedup of unmodified C code with TL2.

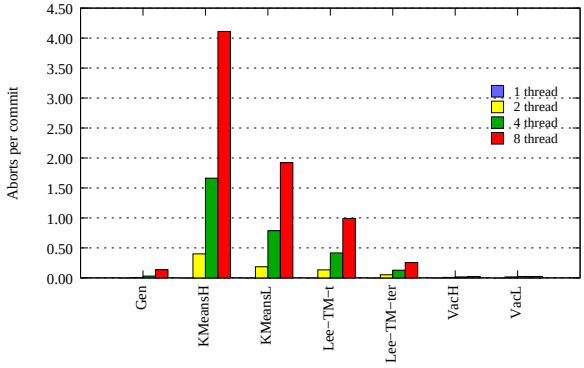
Fig. 1. Speedup of ported code in DSTM2, and original code in TL2.



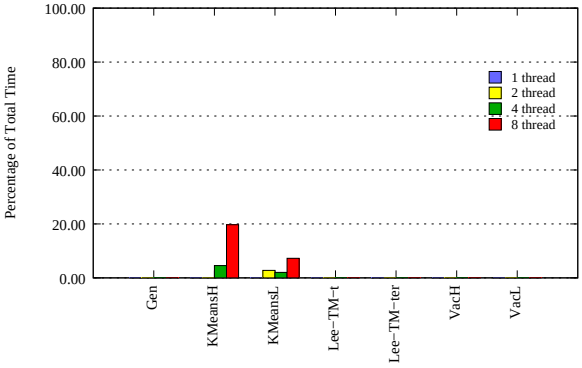
(a) Execution time spent in transactions (InTX).



(b) Wasted work (time in aborted transactions).



(c) Mean Aborts per Commit (ApC).



(d) Proportion of time spent in contention management (CMT).

Fig. 2. InTX, wasted work, ApC, and CMT results.

tional execution in these applications is parallel execution, and non-transactional execution is serial execution. Gen, Lee-TM-ter, and Vac have an InTX of over 95%, and Lee-TM-t has an InTx of 85% at 8 threads. KMeans spends much less time in transactions as its serial phase occupies 55% to 75% of total execution time. Gen, KMeans, and Lee-TM-t show decreasing amounts of InTX as the number of threads increase.

KMeans' alternating serial phase accounts for its serial execution time, which limits its speedup. The increasing proportion of serial execution as the number of threads rises, seen in some applications, is due to the parallel execution time falling, and thus representing a smaller proportion of the total execution time.

Fig. 2b shows wasted work results. Gen and Vac have little wasted work (less than 10%). KMeans and Lee-TM-t have large amounts of wasted work, e.g. on 8 threads the wasted work is between 35% to 70%. Applications with large amounts of wasted work may be suitable candidates for studying contention management.

KMeans speedup is limited by the significant sequential

phase seen in Fig. 2a, and large amounts of wasted work. Lee-TM-t has more wasted work than Vac, but scales similarly. Lee-TM-ter has more wasted work than Vac and Gen, but scales higher than both. This shows that poor scalability can have its root in wasted work (KMeans), but significant wasted work may not prevent scalability (Lee-TM).

Fig. 2c shows ApC results. KMeans has the highest ApC, followed by Lee-TM, Gen, and finally Vac. KMeansH has an ApC four times higher than Lee-TM-t, but 30% less wasted work. This suggests Lee-TM-t aborts large/long transactions as it has fewer aborts, yet large amounts of time spent in the aborted transactions. Gen and Vac exhibit a similar trend to KMeans and Lee-TM, respectively.

Fig. 2d shows CMT results. CMT is negligible for Gen, Lee-TM, and Vac. At 8 threads KMeansH has 20% CMT, KMeansL has 10% CMT, but Lee-TM-t has almost none. This is surprising since Lee-TM-t has half as much ApC as KMeansL, and thus should have half as much CMT. From Fig. 4, Lee-TM-t's mean transaction execution time is far greater than KMeans'. Since the priority contention manager executes deterministically, i.e. always in a very similar amount of time, the CMT represents a smaller proportion of the transaction time in Lee-TM-t.

Fig. 3 presents abort histograms. Single thread execution results (i.e., no possible aborts) show that Gen and Vac execute approximately 1 million transactions, KMeans approximately 250,000 transactions, and Lee-TM more than 1500 transactions. In all cases the abort distribution rises with the number of threads, although the least impact of this is seen in Vac. Gen shows a unique trend amongst the TM applications; a few transactions take 100+ or 1000+ aborts, even with 2 threads (i.e. when the probability of conflict is naturally low), before committing. This leads to Gen's abort histograms forming a u-shape when using 2 or 4 threads, which is levelled at 8 threads. KMeans shows a more even spread of aborts in the range 1-49 compared to the other applications. Lee-TM-ter significantly reduces the number of aborts compared to Lee-TM-t. VacH and VacL show little difference in abort distributions.

The large number of transactions executed suggests there is ample parallelism available, and the poor scalability observed in some experiments is not due to a lack of parallel work. The abort distribution rising with the number of threads is a characteristic of the applications: conflicts are increasing as more transactions are executed in parallel. We speculate Gen's u-shape is due to conflict in inserting elements into hash maps because Gen's result in Fig. 5 has a short phase where aborts become significant, which we believe is related to the u-shape, and that phase occurs during the insertion of gene segments in to hash maps in preparation for iterative substring matching. Although the same scenario occurs at 8 threads, the u-shape is masked by a natural rise in other conflicts. KMeans' even spread of aborts in the range 1-49 suggests that the large amounts of wasted work is not due to a few transactions aborting a large number of times, but rather a large number of transactions aborting a similar number of times. Lee-TM-ter reduces the number of aborts compared to Lee-TM-t, showing that early release is effective in reducing false conflicts. VacH and VacL's similar abort distributions suggesting that the low and high contention configurations (i.e. the recommended parameters) are not different enough to have an impact up to 8 threads.

Fig. 4 illustrates the transaction execution time histogram metric for each of the target applications. KMeans transactions are predominantly of short duration with 97% completing within 0.1ms. The number of KMeans transactions completing within a given interval decreases exponentially. Gen and Vac transactions have similar profiles with the majority of transactions completing within 1ms. Lee-TM transactions are of longer duration with a minimum execution time of 5ms. The number of Lee-TM transactions completing within a given interval decreases logarithmically.

We classify KMeans as having the least variance in execution time and Lee-TM as having the most variance in execution time. The profile of each application appears independent of the number of processors, and independent of low contention and high contention configuration parameters. The source of variance in transaction execution times can be deduced from the applications' descriptions in Section II-A. KMeans' lack of variance is due to all transactions executing the same code block with different input data. Gen and Vac have more variance since they execute a selection of code blocks as transactions, The logarithmic distribution of execution times for Lee-TM corresponds to the distribution of circuit lengths within the input file. Thus, we can conclude that the execution time metric represents a characteristic of the application rather than the execution environment.

Fig. 5 illustrates the ICR for each of the target applications. The graph for Gen shows a pronounced dip in the commit rate representing a point in the execution of the application when a large proportion of the wasted work occurs. The commit rate of Gen is insensitive to the number of threads. KMeans exhibits a constant commit rate, i.e. a fixed proportion of work is wasted throughout the execution of the application. The commit rate of KMeans is lower when more threads are used, and up to 60% of work is wasted when 8 threads are used. Lee-TM exhibits a continual reduction in the commit rate over time, and the commit rate is lower when more threads are used. Vac exhibits a high commit rate that continues to rise during the execution of the application. The commit rate of Vac is insensitive to the number of threads.

The graph for Lee-TM is less smooth than the others as the sample rate is low relative to the execution time of each transaction. The commit rate of Lee-TM decreases as

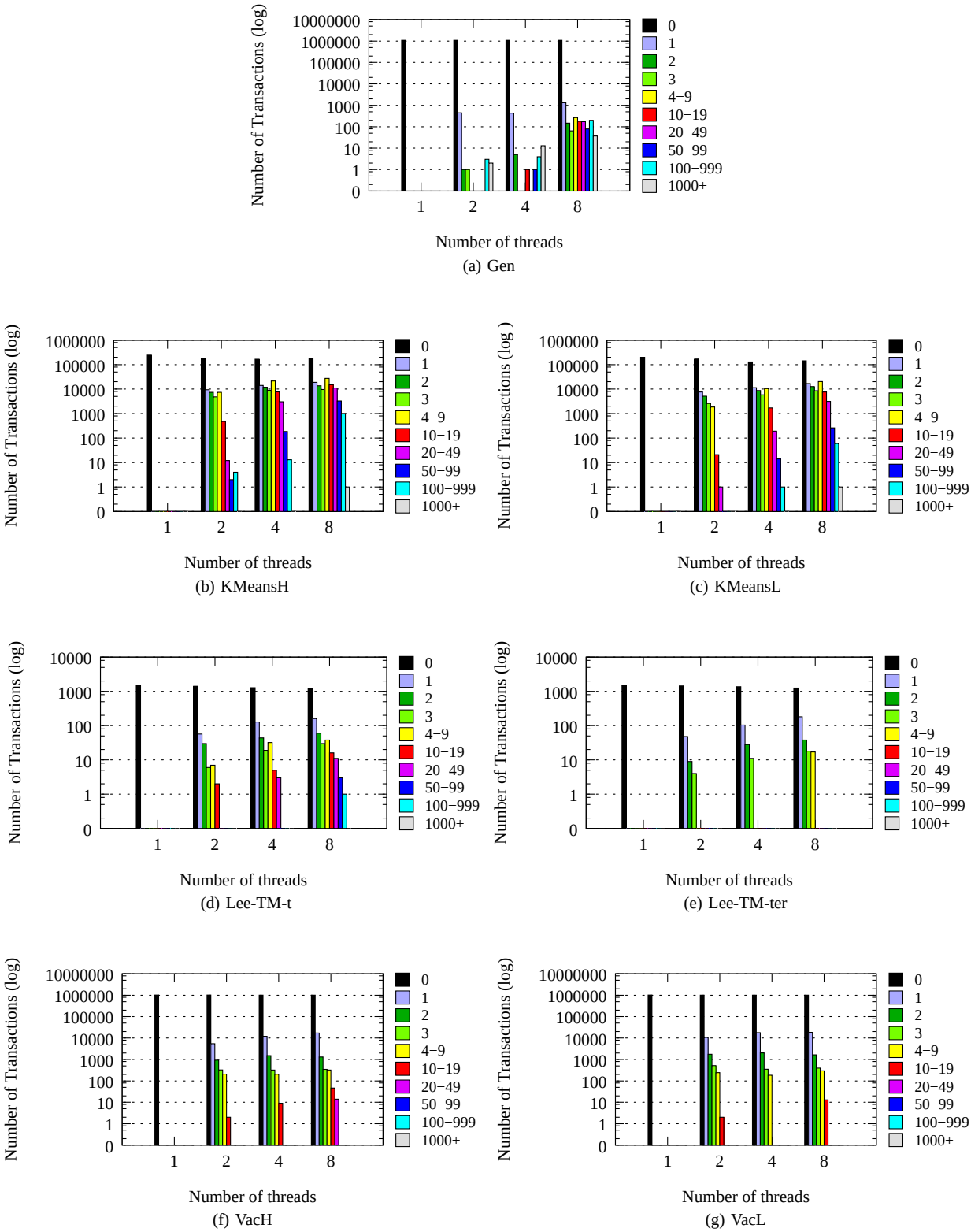
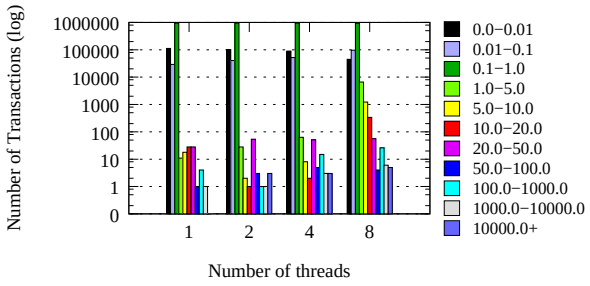
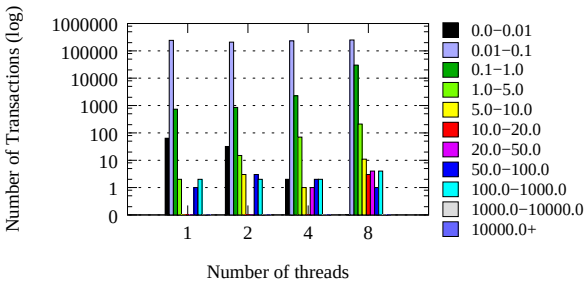


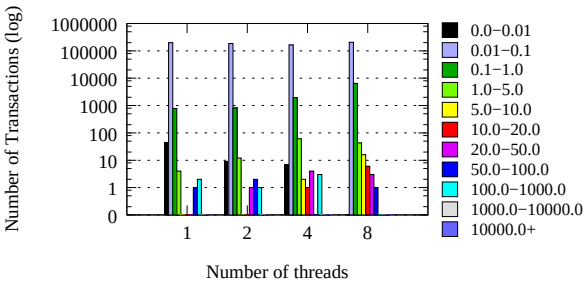
Fig. 3. Abort histograms for each non-trivial TM application. Each bar represents the number of transactions that aborted a given number of times before actually committing. Note y-axis is log scale.



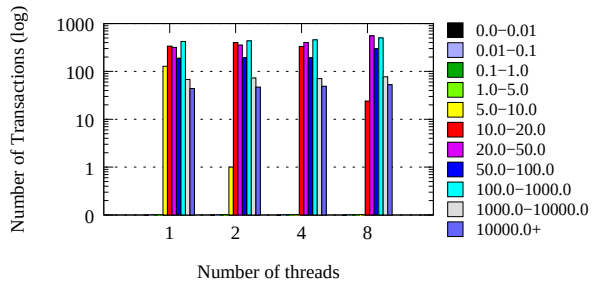
(a) Gen



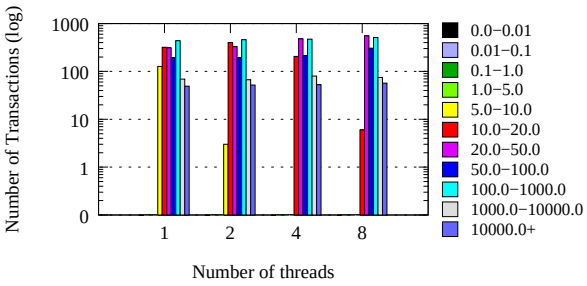
(b) KMeansH



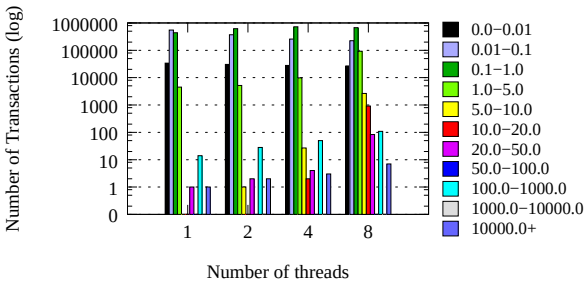
(c) KMeansL



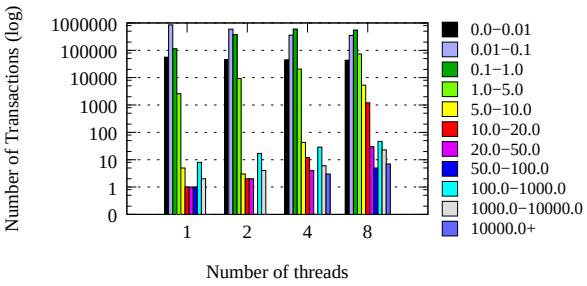
(d) Lee-TM-t



(e) Lee-TM-ter

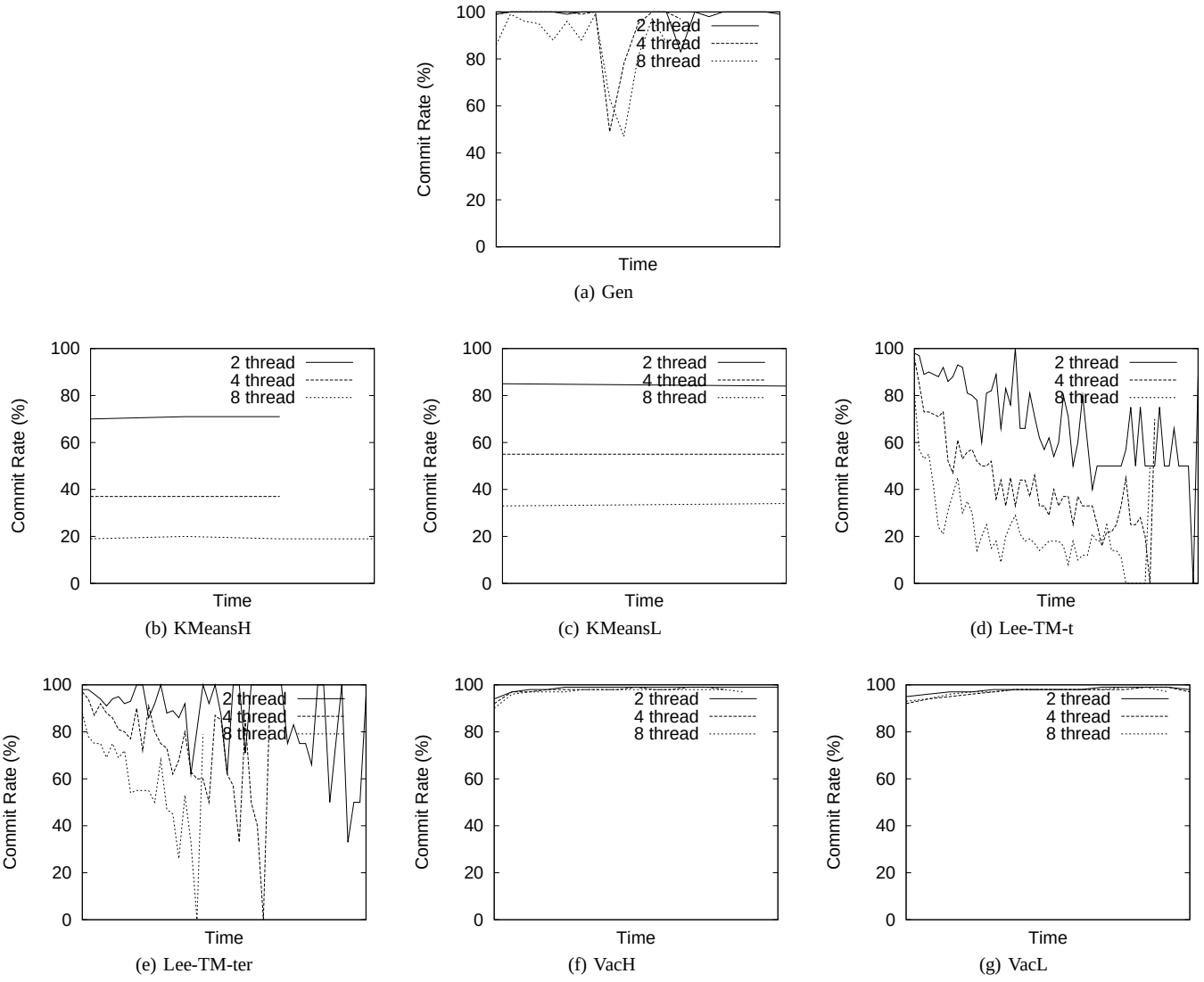


(f) VacH



(g) VacL

Fig. 4. Transaction execution time histograms for each non-trivial TM application. The color of each bar represents a range of elapsed execution times in milliseconds. The vertical axis represents the number of transactions completing within the time range.



(a) Gen

(b) KMeansH

(c) KMeansL

(d) Lee-TM-t

(e) Lee-TM-ter

(f) VacH

(g) VacL

Fig. 5. Instantaneous commit rate (ICR) graphs. In our experiments we sampled at 5 second intervals. The commit rate for 2,4 and 8 threads are plotted. When one thread is used the commit rate is always 100% as there are no conflicts and thus no aborted transactions.

execution progresses since the circuit becomes more crowded, leaving less space for routing new connections, and thus the number of conflicts increases. Both Vac and Gen have a high commit rate which is insensitive to the number of concurrent threads.

We observe that there is great variation in the ICR metric for each target application. The profile of some applications changes in time and altering the number of threads impacts each application differently. The ICR metric is an important characteristic of the application and its sensitivity to the concurrent execution environment.

Table II illustrates the mean readset and writeset size of committing transactions for each of the target applications. There is great variation in the readset and writeset sizes of the target applications. The readset and writeset size is independent of the number of threads, except in the case of the readset size for Lee-TM-t. The readset and writeset size is an indicator of the cost of validating concurrent transactions as the sets must be compared. The execution costs associated with validation are a characteristic of the execution environment.

We can see in Fig. 1a that Gen exhibits a larger speedup with additional threads than Vac, despite Fig. 2b showing that the amount of wasted work in both Gen and Vac is low and independent of the number of threads, and Fig. 4 showing that their transaction execution times are similar. The execution cost associated with the validation of transactions in Vac is greater than that of validating a transaction in Gen as the readset and writesets are larger. The differing scalability of Gen and Vac can be accounted for by their differing readset and writeset sizes. The effect of early release on Lee-TM is to make the readset size of Lee-TM-ter independent of the number of threads. The smaller readset and writeset size of Lee-TM-ter has a significant effect on its scalability when

Application	Readset				Writeset			
	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
Gen	8	8	8	8	7	7	7	7
KMeansH	152	152	152	152	152	152	152	152
KMeansL	157	156	156	156	157	156	156	156
Lee-TM-t	243231	196590	162015	130081	423	421	422	421
Lee-TM-ter	427	425	426	427	423	421	422	423
VacH	168	166	165	165	30	30	30	30
VacL	77	75	74	72	20	20	21	20

TABLE II
MEAN READSET AND WRITESET SIZE OF COMMITTED TRANSACTIONS, IN BYTES.

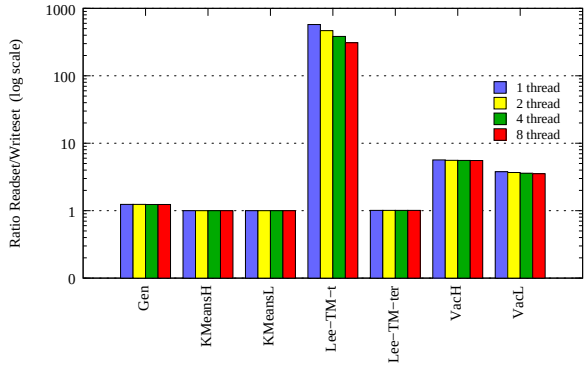


Fig. 6. Ratio of readset to writeset (RStoWS).

compared to Lee-TM-t.

Finally, Fig. 6 presents RStoWS ratios. Gen and KMeans have a 1-to-1 ratio. Lee-TM-t has a very high ratio, but Lee-TM-ter has a 1-to-1 ratio. Lee-TM-ter's lower ratio compared to Lee-TM-t highlights the benefit of early-release, and shows the expansion phase was increasing the ratio by a ratio of hundreds. VacH has a slightly higher ratio than VacL as expected from the configuration parameters.

V. RELATED WORK

Chung *et al.* [19] presented the most comprehensive study looking at 35 different TM benchmarks ranging from mainly scientific computing (JavaGrande, SPLASH-2, NAS, and SPEComp), to commercial workloads (DaCAPO, and SPECjbb). These TM benchmarks were generated following a direct translation from the original parallel benchmarks. The performance evaluation provided a wealth of data with respect to size of transactions, readset and writeset sizes, nested transaction depth, and so on. However, they have not evaluated the non-trivial TM applications studied in this paper, nor have they generated the execution characteristics reported here.

Perfumo *et al.* [18] perform execution characterizations of Haskell TM benchmarks, but does not present the same range of metrics shown in this work, nor study any of the non-trivial TM applications considered in this paper.

The non-trivial TM applications used in this work have been investigated in their respective publications. Gen, KMeansH, KMeansL, VacH and VacL [6] were used to show the scalability of a new hybrid (hardware/software) TM implementation, and the metrics presented included the mean number of instructions, read and write barriers per transaction, and the percentage of time spent executing transactions. Our work has analyzed further characteristics of these TM applications.

Lee's routing algorithm [7] was described as a suitable non-trivial TM application, and its study of aborts led to the use of early release. Early release showed dramatically improved scalability. However the evaluation was performed in an abstracted TM environment. This paper has presented a range of execution characteristics for Lee-TM-t and Lee-TM-ter, as well as performance figures from executing on DSTM2.

Scott *et al.* [9] developed another non-trivial TM application based on Delaunay triangulation. We were not able to build a fair port of the application for DSTM2 as their implementation uses features specific to the Solaris operating system.

Finally, Guerraoui *et al.* [8] developed another non-trivial TM application called STMBench7. Dragojevic *et al.* [24] performed an investigation and found DSTM2 (and other STM implementations) unable to execute STMBench7: due to significant memory overheads in the case of DSTM2.

VI. CONCLUSIONS

This paper has investigated profiling, and its relation to the performance, of several popular non-trivial TM applications: STAMP applications Genome, KMeans, and Vacation, and Lee-TM. To achieve this, we have developed a new TM profiling framework and ported the applications to a common STM system. Two new metrics that have not been presented in the TM literature were introduced: transaction execution time histograms and instantaneous commit rate (ICR). A summary of the most relevant findings follows.

- Poor scalability can have its root in wasted work (as shown by KMeans), but significant wasted work may not prevent scalability (as shown by Lee-TM).
- Readset and writeset size can be a good indicator of application scalability in the absence of wasted work.
- Transaction execution time histogram analysis led to the discovery that KMeans has the most homogeneous transaction execution times, Genome and Vacation have

medium variance in transaction execution times, and Lee-TM has the most heterogeneous transaction execution times.

- KMeans executes hundreds of thousands of transactions, but its scalability is limited due to a large amount of time spent executing serial code, and a large amount of wasted work.
- Lee-TM shows the potential all-round benefits of early release, by increasing scalability, and reducing wasted work, contention time, ApC, and readset size.
- Lee-TM has the highest probability of overflowing HTM resources, and accordingly stress HyTM systems due to its large readset and writeset.
- Vacation has little wasted work, low ApC, negligible CMT, and a high ICR, but its scalability up to 8 threads is at best around 1.5, and data on readset and writeset sizes revealed this may be due to the application's cost of transaction validation.
- Instantaneous commit rate (ICR) graphs revealed that Genome has a phase of execution where available parallelism drops dramatically, Lee-TM has a decaying ICR, and KMeans distributes its wasted work evenly over its total execution.
- ICR graphs also showed that Genome and Vacation's commit rates are independent of the number of threads up to 8 threads, whereas KMeans and Lee-TM's commit rate drops as the number of threads increases.
- From a contention management point of view the ICR graphs revealed KMeans and Lee-TM exhibit the most complex behavior.

REFERENCES

[1] 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, May 1993.

[2] 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.

[3] Richard McDougall. Extreme software scaling. *ACM Queue*, 3(7):36–46, 2005.

[4] 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.

[5] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[6] 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.

[7] 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.

[8] 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.

[9] Michael L. Scott, Michael F. Spear, Luke Dalessandro, and Virendra J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC '07: Proceedings of the 2007 IEEE International Symposium on Workload Characterization*, pages 107–113. IEEE Computer Society Press, September 2007.

[10] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Lee-TM: A non-trivial benchmark for transactional memory. In *ICA3PP '08: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing*. LNCS, Springer, June 2008.

[11] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Investigating software transactional memory on clusters. In *IWJPDC '08: 10th International Workshop on Java and Components for Parallelism, Distribution and Concurrency*. IEEE Computer Society Press, April 2008.

[12] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246. ACM Press, February 2008.

[13] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216. ACM Press, February 2008.

[14] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–196. ACM Press, February 2008.

[15] Torvald Riegel and Diogo Becker de Brum. Making object-based STM practical in unmanaged environments. In *TRANSACT '08: Third ACM SIGPLAN Workshop on Transactional Computing*, February 2008.

[16] Maurice Herlihy and Eric Koskinen. Checkpoints and continuations instead of nested transactions. In *TRANSACT '08: Third ACM SIGPLAN Workshop on Transactional Computing*, February 2008.

[17] 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.

[18] Cristian Perfumo, Nehir Sonmez, Adrian Cristal, Osman Unsal, Mateo Valero, and Tim Harris. Dissecting transactional executions in Haskell. In *TRANSACT '07: Second ACM SIGPLAN Workshop on Transactional Computing*, August 2007.

[19] JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. The common case transactional behavior of multithreaded programs. In *HPCA '06: Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pages 266–277, February 2006.

[20] Dave 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.

[21] William Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *CSJP '04: Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[22] 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.

[23] William Scherer III and Michael L. 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.

[24] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Dividing transactional memories by zero. In *TRANSACT '08: Third ACM SIGPLAN Workshop on Transactional Computing*, February 2008.