March 1, 2008

On the Characterisation of Complex Transactional Memory Applications

Mohammad Ansari
Christos Kotselidis
Kimberly Jarvis
Mikel Luján
Chris Kirkham
Ian Watson

School of Computer Science
The University of Manchester
Preprint Series
CSPP-44

On the Characterisation of Complex Transactional Memory Applications

Mohammad Ansari
Christos Kotselidis
Kimberly Jarvis
Mikel Luján
Chris Kirkham
Ian Watson [*]

March 1, 2008

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. However, complex TM applications to test TM implementations have only recently begun to emerge, and their execution characteristics have not been fully investigated. Complicating matters further, the complex TM applications have been written in different programming languages, using different TM implementations, making comparisons difficult.

We have ported several complex TM applications to a single TM implementation, and built into it a framework to profile their execution. This paper presents performance figures and execution characteristics of major complex TM applications up to 8 processors, and for the first time, due to executing under a single TM implementation, presents directly comparable performance figures and execution characteristics. Also the priority contention manager is found to provide the best overall results for these applications, in contrast to previously published results that suggest the polka contention manageer gives the best overall results.

[*]School of Computer Science, The University of Manchester

1 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 (CMPs) [3].

TM implementations usually consist of a runtime system and language-based constructs (or library calls) to mark code blocks as *transactions*. Whenever a transaction executes, the 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. If conflicting accesses (e.g. write/write or read/write) are detected, then one of the conflicting transactions is *aborted* and restarted. A *contention manager*, part of the TM runtime system, decides which transaction to abort based upon a policy [4]. A transaction that completes execution of its code block, has no conflicts, and has not been aborted, can *commit* its writeset. TM implementations exist in a variety of flavours, including software-based (STM), hardware-based (HTM), and hardware/software hybrids (HyTM), and readers can refer to Larus and Rajwar [5] for details.

Several nontrivial programs specifically designed for parallelisation with TM have appeared in the last year [6, 7, 8, 9]. However, their execution characteristics have not been fully investigated. Complicating matters further, the complex TM applications have been written in different programming languages, using different TM implementations, making comparisons difficult.

This paper presents performance figures and execution characteristics of Genome, Vacation and KMeans (from the STAMP suite version 0.9.5 [6]) and Lee's routing algorithm [7]. These applications have been ported to a single state-of-the-art TM implementation, the Java-based Software TM (STM) called DSTM2 [10], which has been extended with a TM execution profiling framework. Furthermore, executing the complex TM applications in a single TM implementation allows, for the first time, comparisons between their performance and execution characteristics. Execution characteristics are derived from the analysis of metrics, and this paper also reports two new metrics that have not been used in previous related work [11, 12]; running percentage commit rates, and transaction execution time histograms (see Section 3 for their definitions).

The execution behaviour reveals observations such as a phase during the execution of the Genome application (see description in Section 2.1) where observed parallelism is significantly lower than the rest of the execution. Another example observation is that this collection of applications generate transactions with a wide range of execution times, and a wide degree of homogeneity/heterogeneity among the transactions that form part of an application. The metrics are illustrated and their relationship with the observed application scalability is pursued.

This paper is organised as follows: Section 2 describes briefly the parallelisation of the complex TM applications under investigation and the execution parameters used. Section 3 explains and motivates the metrics generated to characterise the TM behaviour of the applications. Section 4 walks through the different performance figures and execution characteristics trying to understand the scalability results, while Section 5 introduces related work. Section 6 completes the paper with a summary of the observations of the recorded TM behaviour.

2 Complex TM Applications

Recently, several research groups have worked towards building complex TM applications for thorough TM implementation analysis [6, 7, 8, 9]. Informally, by complex TM applications we mean those that have all or most of the following features:

- have the potential for large amounts of parallelism, but are difficult to parallelise using fine-grain locking,
- execute several different types of transactions (i.e. several code blocks marked as transactions),
- have dynamic amounts of parallelism over a single execution,
- execute transactions of varying length (varying amounts of computation),
- execute transactions of varying size (varying amounts of data accessed), and
- are based on real-world applications.

Complex TM applications are important for TM research as they allow performance analysis of TM implementations in realistic scenarios. For example, workloads with dynamic amounts of parallelism can be used to improve emerging adaptive concurrency mechanisms [13].

This paper analyses Lee's routing algorithm [7], and all the STAMP suite version 0.9.5; i.e. Genome, Kmeans, and Vacation [6]. Lee's routing algorithm, originally in Java, has been implemented using transactions with DSTM2 keywords. STAMP applications have been ported from C to Java, and converted from using TL2 [14], a STM, to DSTM2 [10]. STAMP applications also required the implementation of additional utility classes in DSTM2: a transactional linked list and a transactional hash map. The remainder of this section briefly describes each application and its parallel execution characteristics.

2.1 Analysed Applications

Lee's routing algorithm is used in circuit routing to make connections automatically between points. Routing is performed on a 3D grid, 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 into a global pool, and sorts them into ascending length order (to reduce 'spaghetti' routing). It 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, and synchronised access to the global pool of connections. Lee's routing algorithm is fully parallel, with conflicts at points where connections attempt to write to the same grid cell. Since routes are sorted in ascending length order, and the probability of a conflict rises with route length, the available parallelism should fall as execution progresses.

Genome is a gene sequencing program 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, and the available parallelism should fall as execution progresses.

Kmeans is an application that 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 centres are assigned randomly). The other phase re-calculates cluster centres 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. Available parallelism should remain the steady throughout execution as the object-to-cluster assignment conflicts are unlikely to change significantly.

Vacation simulates a travel booking database in which multiple threads book or cancel cars, hotels, and flights on behalf of customers transactionally. 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.

At this time, and given the page limit, we have not included Delaunay triangulation in the study given that the transactional part of the application is approximately 8% of the execution time on a SunFire machine or 1.2% on a Niagara-based machine [9]. Similarly, we have not included STMBench7, which is derived from the OO7 benchmark for object-oriented databases, but STMBench7 with its default execution parameters “performs very poorly” [8]. In future work, we will present the results of these two benchmarks; the porting effort is well underway.

3 Analysed Metrics

DSTM2 has been instrumented to capture execution data that is used to generate metrics for characterising the execution of the applications mentioned above. This section defines the metrics used for execution characterisation. Although we only motivate their use in characterising complex TM applications, these metrics can additionally be used for characterising any transactional application or benchmark.

Scalability is the classic metric presented to show how well applications speed up with more processing resources. Scalability is calculated as total time divided by the number of threads used. It is a key measure of the application’s parallelism, and the TM runtime system’s efficiency. Poor scalability motivates the study of the metrics described below.

In transactions (InTX) is the percentage of total time the applications spent executing aborted and committed transactions, i.e. the percentage of total time spent in parallel execution.

The remaining percentage of time is spent executing serial code. InTX is important when judging scalability, as less time spent in transactions leads to reduced maximum achievable scalability.

Wasted work shows the percentage of transaction execution time spent executing aborted transactions. 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 average number of 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 favouring 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 an average 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. A wide range of transaction execution times is one of the characteristics of a complex TM application, and the wider the spread of transaction execution times, the higher the confidence in a TM implementation's performance. This metric describes how homogeneous or heterogenous is the amount of work contained in transactions for a given application.

Running Percentage of Commit Rate (RPCR) shows the percentage of committed transactions at sample points during the execution of the application. These traces help identify phases where wasted work occurs during the execution of an application. This information can be useful, for example, in improving emerging adaptive concurrency techniques for TM [13].

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 complex TM applications gives higher confidence that the hardware will not overflow for a large proportion of transactions. Note that, 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 average number of reads that lead to a write in a committed transaction. Although this varies widely from application to application, a very high ratio combined with short running transactions will make the overhead of recording the readset and writeset in STMs more significant and make it difficult to scale. HTMs eliminate, or reduce by orders of magnitude, this recording overhead and the combination of RStoWS and transaction execution time histograms may indicate that such applications only scale on HTMs.

Writes-to-writeset ratio captures the average 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 average number of reads to a transactional data

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	Lee w/o early release	early_release:false, input_file:mainboard.txt
LeeER	Lee with early release	early_release:true, input_file:mainboard.txt

Table 1: Application parameters used to gather performance figures and execution characteristics.

element in a committed transaction. Reading transactionally shared data incurs extra costs, and a high RtoRS, 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 optimised away by not recording them again and again. For brevity this data is ommitted as only Lee's algorithm was found to have a high ratio, and this information is also visible in the RStoWS graphs because none of the applications performed multiple writes to a transactional data element.

4 Characterisation

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 factory and eager conflict detection [10]. Table 1 shows the configurations executed for each application. STAMP applications are executed using the parameters suggested in the guidance notes supplied with the suite. Lee's routing algorithm is executed using the default dataset (a real circuit) with and without early release [4]. Each experiment is repeated twenty times and the best result presented. All popular contention managers [15, 16, 17] have been used, but only results with the *priority manager* are presented as it gives the best overall performance results. 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 performance results, and then study the applications' execution characteristics to understand the observed performance. Hereonwards, experiments are referred to by their configuration names introduced in Table 1.

Each experiment is a combination of:

- number of threads (1,2,4 or 8),
- application configuration (e.g. Vacation with high contention), and

- contention manager.

Figure 1 presents the graphs for the scalability metric (speedups). LeeER (early release) shows a clear positive effect on scalability compared with Lee. Only LeeER is able to reach speedups above 2 on 4 and 8 threads. KmeansL, KmeansH, VacL and VacH generally obtain less than 1.5 speedup, with only Gen reaching above a speedup of 2 on 4 and 8 threads. An unexpected result is that VacH (high contention) outperforms VacL (low contention). This could be attributed to errors in porting the code, and to eliminate this possibility, the unmodified C/TL2 code is executed twenty times and the best run is shown in Figure 2. Except for Gen, that is able to scale close to linearly, the other results have a similar trend but with smaller speedups than their Java/DSTM2 counterparts.

Figure 3 shows the results for the metric InTX. LeeER, VacL, and VacH do not show significant serial execution (less than 15%), or increasing serial execution with the number of threads. The single thread result of Gen shows that the serialised final phase of constructing the gene sequence represents a small component of the execution time (less than 10%). The upper bound of the scalability for KMeansH and KMeansL is limited by the significant sequential phase (approximately 55% to 75%). Lee has no serial component by default, as shown in the single thread result. Gen, KmeansH, KmeansL and Lee, have increasing percentages of serial execution time as the number of threads rises given that the total execution time is reduced (see Figure 1).

Figure 4 presents the graph for the metric wasted work. Each application configuration executes a large number of transactions (between thousands to millions) as will be shown in Figure 7. Gen, VacL, and VacH have little wasted work (less than 10%). KmeansH, KmeansL, Lee, and LeeER have large amounts of wasted work (e.g. on 8 threads the wasted work is in the range 35% to 70%), suggesting there is possible room for improvement in contention management for these applications. Combined with the results in Figure 3, it is now clearer the reasons behind KmeansH and KmeansL only reaching a speedup below 1.5. Both have a significant sequential component, and they are further hampered by costly aborts depicted as wasted work. Interestingly, LeeER, VacH, and VacL all have little serial execution, yet LeeER has more wasted work and scales much better than VacH or VacL. In other words, a large wasted work may not prevent scalability but poor scalability can have its root in wasted work (see Lee).

Figure 5 shows the graph with the ApC metric for each application. It is difficult to determine what threshold represents a small ApC with respect to scalability or even wasted work. Note, for example, that although KmeansH has approximately 4 times higher ApC than Lee, Lee is the one with larger wasted work. LeeER shows low ApC, yet significant wasted work, suggesting that large/long transactions are being aborted. Also note VacH and VacL have the smallest ApC, but hide a wasted work higher than Gen. For these applications we find ApC the least indicative metric to understand the TM behaviour.

Figure 6 illustrates the results with the metric CMT. Consistently the priority contention manager provides a negligible overhead (cannot be seen for Gen, Lee, LeeER, VacH and VacL), thus giving the overall best performance results. Interestingly, we observed that at 8 threads KmeansH has 20% CMT, KmeansL has 10% CMT, but Lee has almost none, despite having more wasted work. This is due to the average transaction execution time being far greater in Lee than in KMeansH or KMeansL, as is shown later in Figure 8, and their high ApC shown in Figure 5.

Abort histograms are presented in Figure 7. The single thread results (no possible abort) show that Gen, VacH, and VacL execute around 1 million transactions, KmeansH and KmeansL approximately 250,000 transactions, and Lee and LeeER above 1500 transactions. This indicates that the number of transactions executed is not the reason for the poor scalability observed in most of the configurations. Each bar represents the number of transactions that aborted a given number of times before actually committing. The first bar from the left represents, for example, zero aborts. Moving from zero aborts to one abort, a drop in the number of transactions between three orders of magnitude (Gen) to one order of magnitude (e.g. KmeansH and Lee) is observed. Note the y-axis is log scale. Beyond 3 aborts, each bar represents a range rather than only one value. Thus, for example, the fifth bar from the left counts how many transactions aborted between 4 and 9 times before being able to commit. Gen shows a unique trend amongst the TM applications; a few transactions takes 100+ or 1000+ aborts, even with 2 threads, before committing. On 2 and 4 threads the histogram has a u-shape, that is levelled on 8 threads. On the other histograms we do not find such a lack of aborts in the range 4-100, but rather a natural growth from left to right as the number of threads increases. KmeansH and KmeansL, both with high ApC, show a fairly even spread of abort distribution in the range 1-49. Compared to each other, VacH and VacH show similar abort distributions suggesting that the difference between the low and high contention configurations (i.e. the recommended parameters) is not significant enough to have an impact. LeeER is considerably better than Lee, reducing the abort distribution by orders of magnitude, again highlighting the benefit of early release. The minimum number of aborts appears in LeeER with 3 or fewer aborts on 2 threads and 9 or fewer aborts on 8 threads.

Figure 8 shows transaction execution time histograms. Each bar represents the number of committed transactions that have executed in the given time range, up to, but not including the upperbound. KmeansH and KmeansL have the sharpest apex in their second bar (0.01-0.99ms) with the next largest bar being normally two orders of magnitude smaller. Note the log scale in the y-axis. The average transaction of KmeansH and KmeansL is the shortest average transaction. KmeansH and KmeansL also have the most homogenous profile; the majority of the transactions fall into the same range or have similar execution time. Gen, VacH and VacL have similar profiles with the majority of the transactions falling into three ranges (covering from 0.0 to 1.0 ms). Given these results, Gen, VacH and VacL can be classified as intermediate homogeneity. Lee and LeeER have fewer transactions (around 1500 connections to be laid on a real circuit) than the other applications. However the minimum execution time of any transactions is at least 5 times longer than the other applications. Also note that for both Lee and LeeTM the number of transactions is distributed evenly among four bars (covering from 10 to 999ms) and the neighbour bars normally only falling by one order of magnitude. This profile corresponds to a heterogenous mix of transaction execution times. The lengths of the connections in the circuit map into this heterogeneous profile.

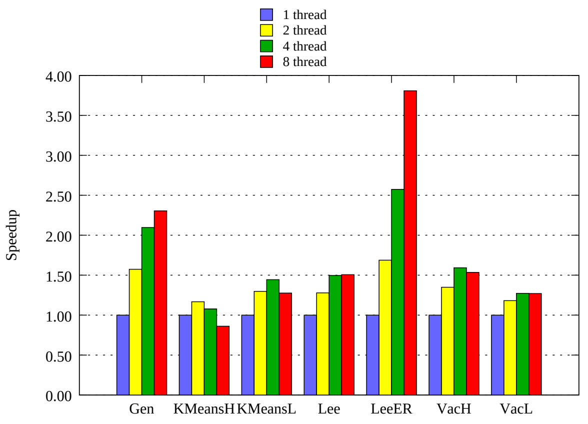


Figure 1: Speedup of the applications with DSTM2.

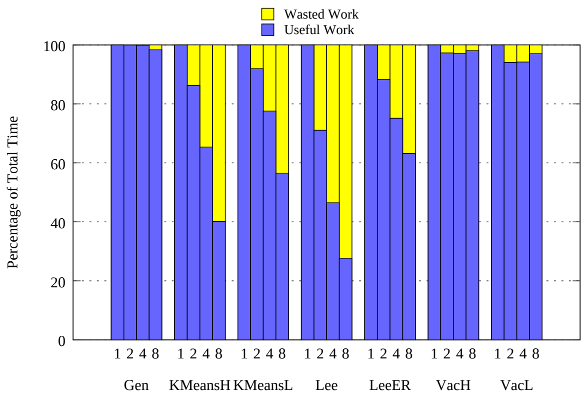


Figure 4: Wasted work (time in aborted transactions).

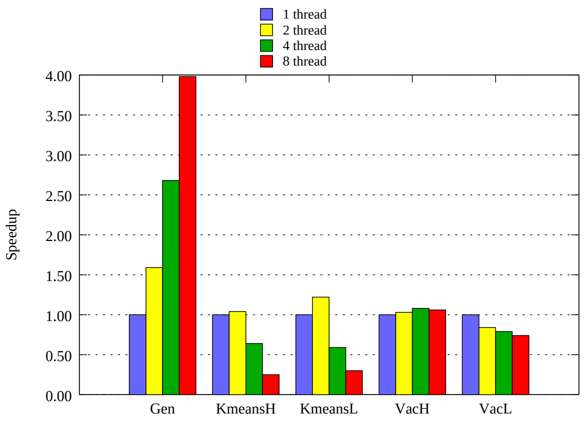


Figure 2: Speedup of unmodified C code with TL2.

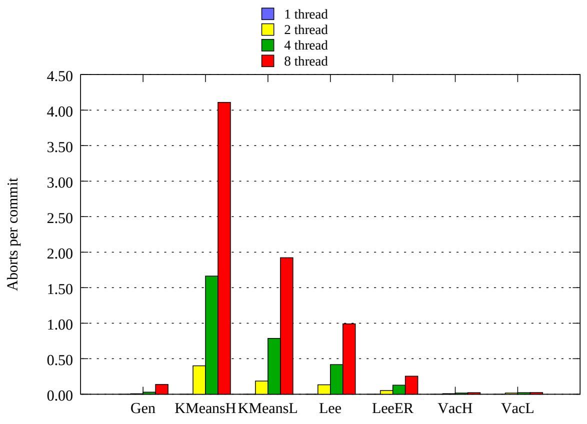


Figure 5: Average Aborts per Commit (ApC).

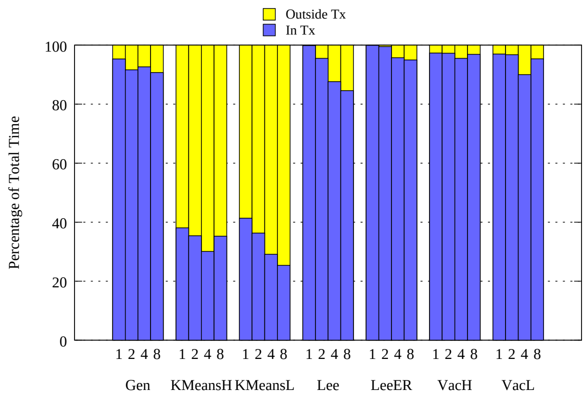


Figure 3: Execution time spent in transactions (InTX).

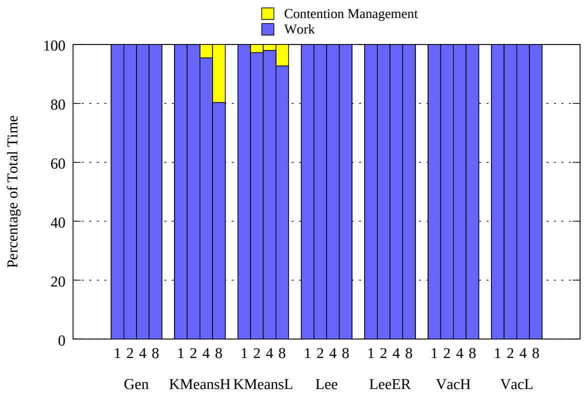


Figure 6: Proportion of time spent in contention management (CMT).

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	243231	196590	162015	130081	423	421	422	421
LeeER	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 2: Readset and writeset sizes per average committed transaction (in bytes).

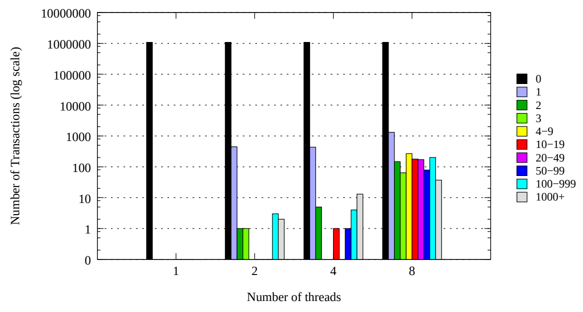
Figure 9 illustrates the graphs with the metric RPCR. They are obtained by sampling the percentage of committed transactions every 5 seconds. The x-axis represents the execution time of the application. Some of the lines representing the RPCR with a given number of threads are shorter in the x-direction than others as the execution of the application executes faster. KMeansH and KMeansL show a steady RPCR, thus wasted work is distributed evenly throughout execution. VacH and VacL exhibit a RPCR that rises slightly as execution progresses, and hardly varies with the number of threads. Gen shows a large dip in the middle, revealing an execution phase where a large proportion of the wasted work occurs. Lee shows a continual reduction in RPCR, and LeeER fluctuates significantly although it is possible to see that increasing threads reduces the overall RPCR, which is also reflected in the wasted work. A possible pitfall of using RPCR is illustrated with Lee and LeeTM. The sampling rate is too frequent compared with the execution time of the transactions in Lee and LeeER. Thus, poor samples record instances of the execution where several transactions are active, but few or none have had enough time to finish and commit, leading to the exacerbated peaks observed (sometimes dropping down to zero). The RPCR graphs should only be looked at in conjunction with the transaction execution time histograms.

In trying to understand the scalability results the missing piece in the puzzle is the difference between Gen, and VacH and VacL. Until now these three applications have had similar metrics, but Gen has been able to scale better than VacH and VacL (see Figure 1). Table 2 shows the readset and writeset sizes of the complex TM applications. The readset and writeset of Gen is 3-21 times smaller than VacH and VacL. Given the overhead of STM for recording these sets, this can explain the difference in the scalability behaviour of these applications. In section 3 it was mentioned that readset and writeset information is useful for selecting hardware buffer sizes. These applications would comfortably fit in a 512-byte buffer, except for Lee which would need a 256-KByte buffer. Also note how early release, LeeER, considerably reduces the set sizes.

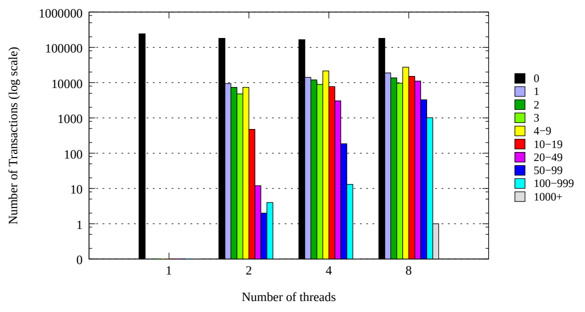
Finally, Figure 10 presents RStoWS. The benefit of early release in Lee's routing algorithm is highlighted; Lee has a very high ratio, and LeeER improves it significantly, with a good impact on all characteristics, as has been shown in this section. VacH has a slightly higher ratio than VacL, as expected from the configuration parameters.

5 Related Work

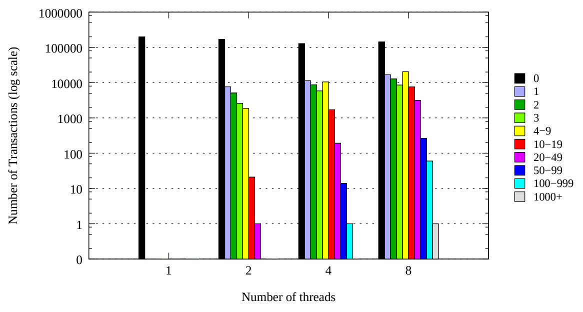
Chung *et al.* [12] presented the most comprehensive study looking at 35 different TM benchmarks covering from mainly scientific computing (JavaGrande, SPLASH-2, NAS, and SPEComp),



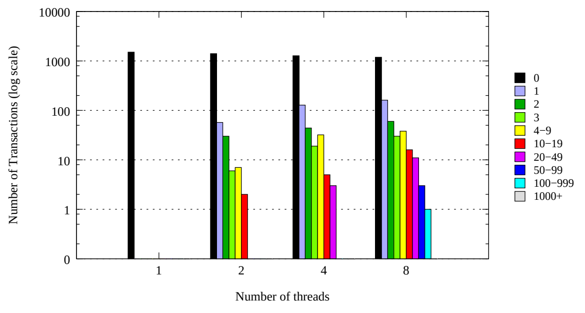
Gen



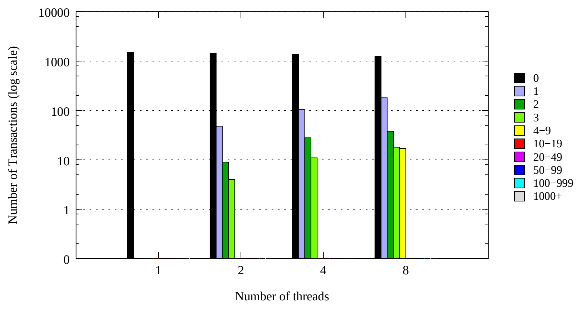
KMeansH



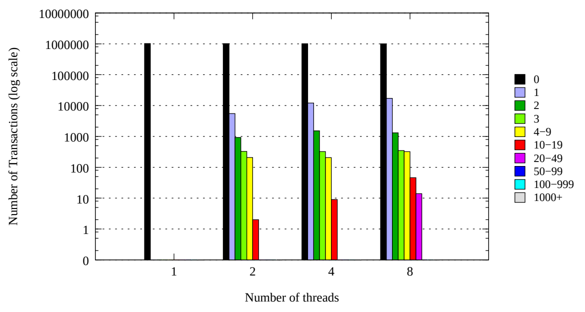
KMeansL



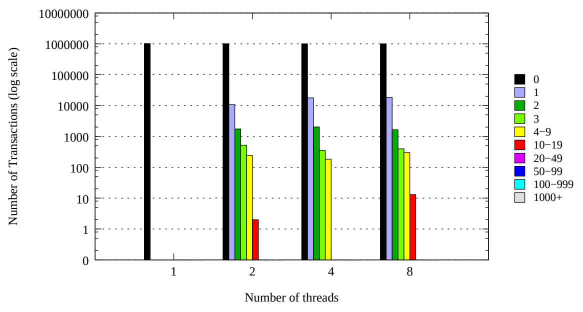
Lee



LeeER



VacH



VacL

Figure 7: Abort histograms for each complex TM application. Legend shows number of aborts per transaction.

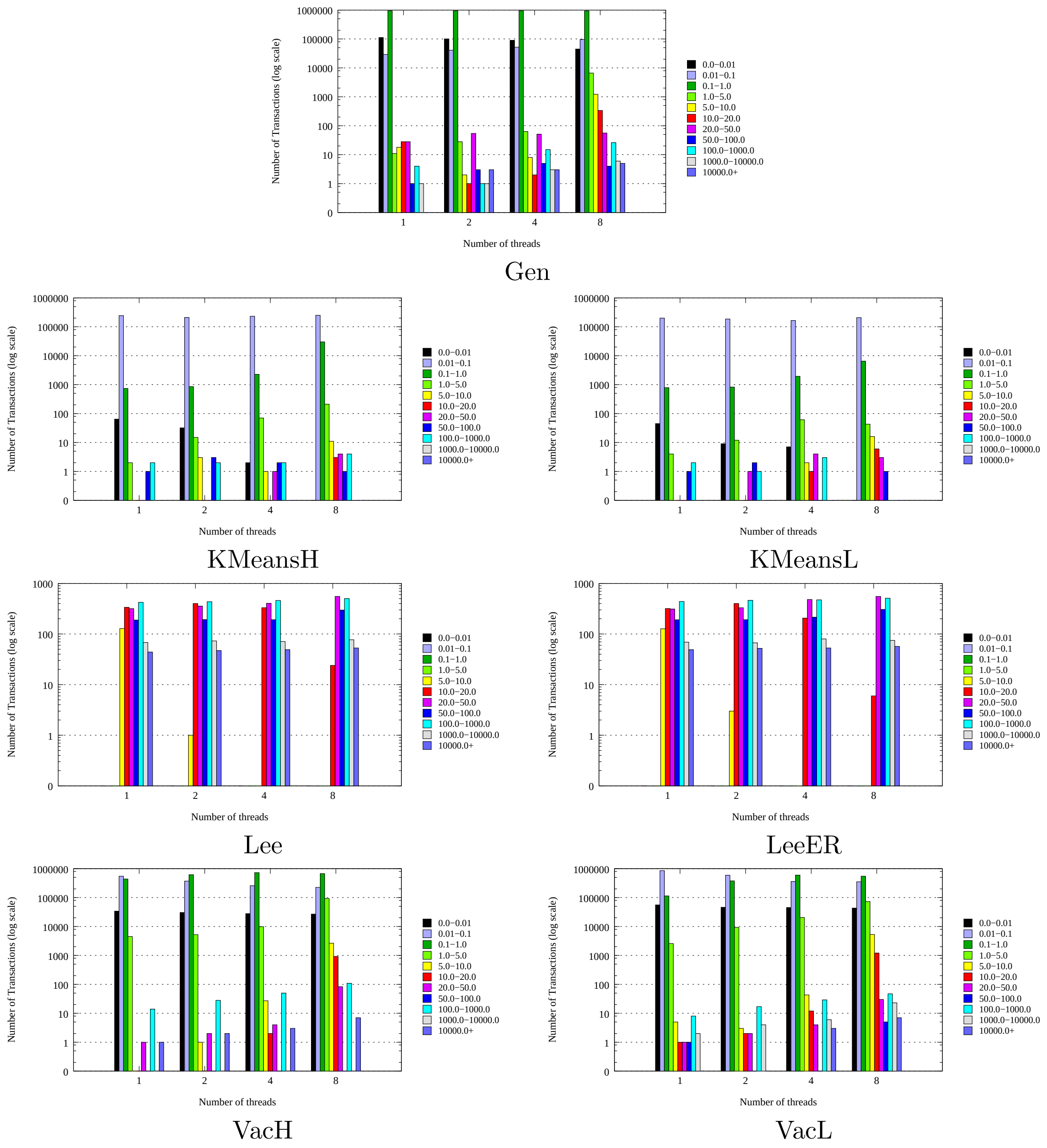
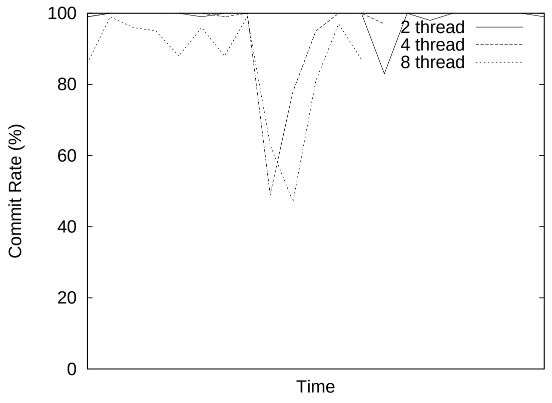
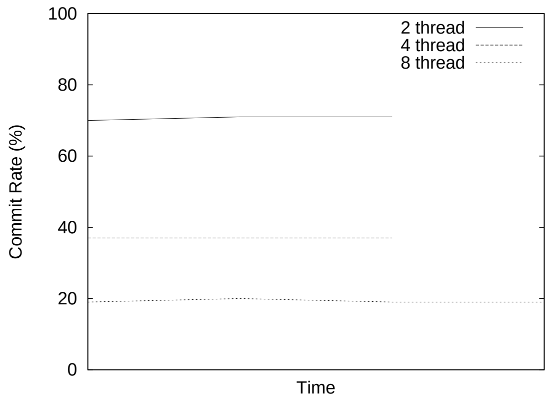


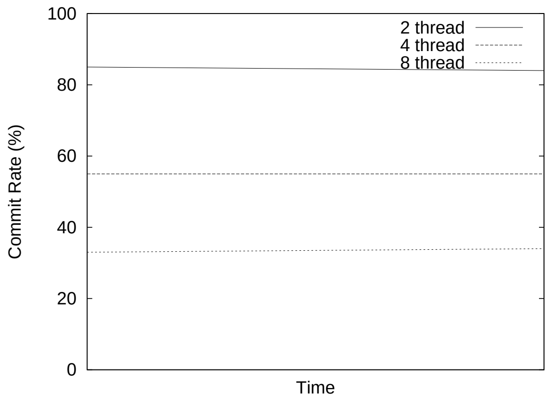
Figure 8: Transaction execution time histograms for each complex TM application. Legend shows execution time range in milliseconds. Each range is up to, but not including its respective upperbound.



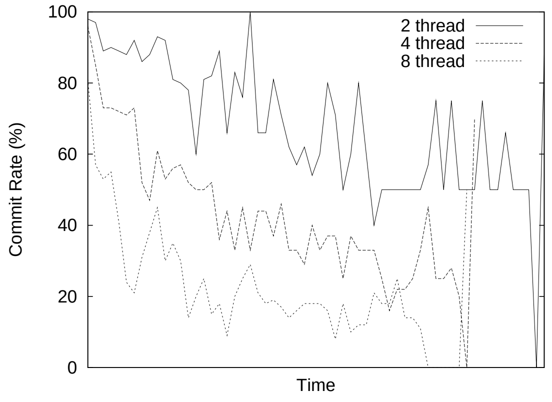
Gen



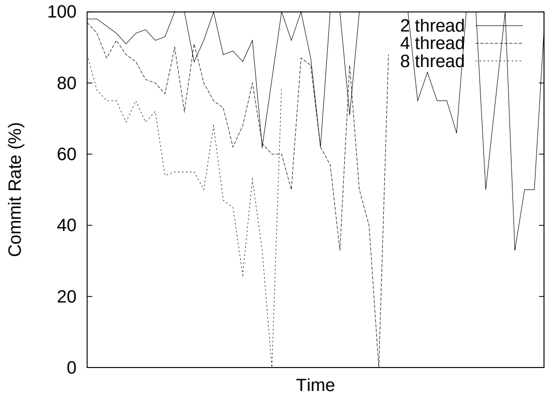
KMeansH



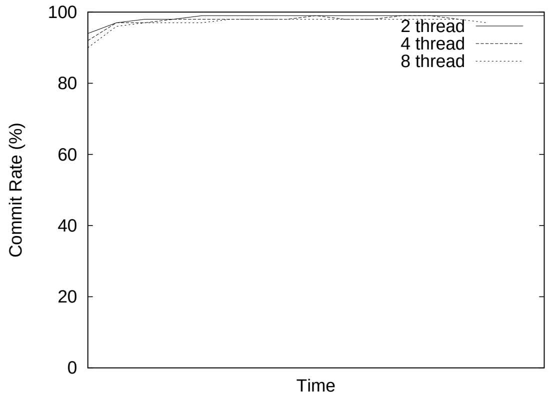
KMeansL



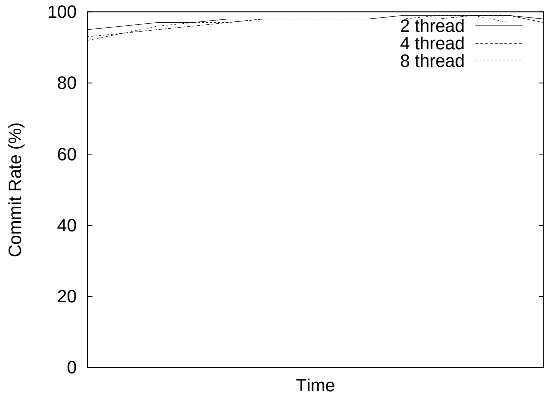
Lee



LeeER



VacH



VacL

Figure 9: Running commit rate percentage (RPCR) for each complex TM application.

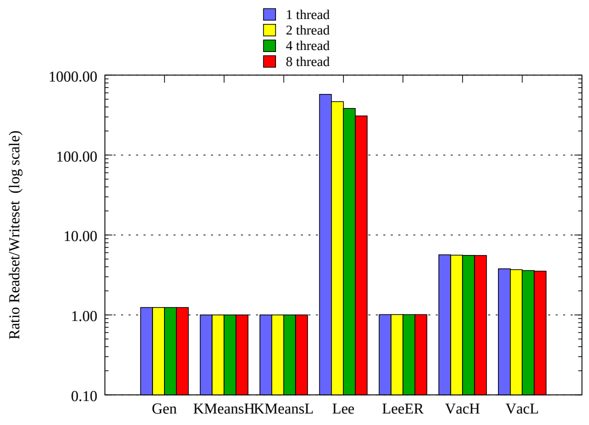


Figure 10: Ratio of readset to writeset (RStoWS).

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. Chung *et al.* [12] acknowledge a limitation of their methodology which is not able to generate the frequency of transactional aborts.

Perfumo *et al.* [11] perform a similar execution characterisations of Haskell TM benchmarks, but does not present the same range of metrics shown in this work, nor study any of the complex TM applications considered in this paper. Specifically, we have not found the same relationship between the size of the readset and the ApC that they present; larger readset leads to larger probability of aborts.

The complex 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 average number of instructions, read and write barriers per transaction, and the percentage of time spent executing transactions. Our work has extended the characteristics of these TM applications.

Lee's routing algorithm [7] was described as a suitable complex 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 and LeeER, as well as performance figures from executing on a state-of-the-art STM implementation.

Finally STMBench7 and Delaunay triangulation are complex TM applications that have been omitted in this paper. An effort is well underway to have them in our framework. STMBench7 [8], based on the database application OO7, was used as a comparison between TM-based execution (using ASTM [18]) and using lock-based execution. The results showed the TM-based execution scaled poorly with respect to locks. Delaunay triangulation, using RSTM [19], was found that the transactional part of the application is approximately 8% of the execution time on a SunFire machine or 1.2% on a Niagara machine [9]. Both works highlighted the need for complex TM applications, but did not study execution characteristics extensively.

6 Summary

This paper has presented metrics to describe TM behaviour and has reported performance figures and execution characteristics for a set of complex TM applications that have been recently published: the STAMP suite (Genome, KMeans, and Vacation), and Lee's routing algorithm. The study has been performed by developing a profiling framework into a state-of-the-art STM implementation (DSTM2) to gather execution data, and generating metrics to observe the execution characteristics of the complex TM applications up to 8 threads. The paper has navigated through the metrics to understand the observed scalability. The study has provided the following observations:

- Vacation has little wasted work, low ApC, negligible CMT, and a high average RPCR, yet the scalability on 8 threads is at best around 1.5 because of the small execution times of the transactions compared to the cost in a STM to record the relatively large readset and writeset.
- KMeans executes hundreds of thousands of transactions, but the scalability is limited due to the low amount of time spent executing transactions, and the large amount of wasted work
- Lee's routing algorithm shows the potential all-round benefits of early release, by increasing scalability, and reducing wasted work, contention time, and aborted transactions.
- Genome has a phase of execution in which significant amounts of the total wasted work occurs, which could be targeted for improving emerging adaptive concurrency techniques.
- The priority contention manager was found to give the best performance for these complex TM applications. This is in contrast to previously published results that suggested the polka contention manager gives the best overall performance. Although we have not presented details of the performance difference between these two contention managers, we note that the previous evaluation did not the same TM applications.
- The ApC graph was found to offer the least information.
- According to the distribution of transaction execution times, Lee's routing algorithm is the most heterogenous, Kmeans is the most homogeneous, while Genome and Vacation sit in the middle.
- Overall, the InTX, wasted work, RPCR and sizes of readset and writeset, and transaction execution time histograms metrics were the most informative to understand the observed scalability.

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 iIternational Symposium on Computer Architecture*, pages 289–300, New York, NY, USA, May 1993. ACM Press.

[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, New York, NY, USA, August 1995. ACM Press.

[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, New York, NY, USA, 2003. ACM Press.

[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, New York, NY, USA, Jun 2007. ACM Press.

[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, 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 *Proceedings of the 2007 IEEE International Symposium on Workload Characterization*, pages 107–113. 2007.

[10] 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, New York, NY, USA, 2006. ACM Press.

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

[12] 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. Feb 2006.

[13] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Adaptive concurrency control for transactional memory. In *MULTIPROG '08: First Workshop on Programmability Issues for Multi-Core Computers*, January 2008.

[14] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, Sept 2006.

[15] William Scherer III and Michael Scott. Contention management in dynamic software transactional memory. In *CSJP '04: Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.

[16] 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, New York, NY, USA, 2005. ACM Press.

[17] 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, New York, NY, USA, 2005. ACM Press.

[18] Virendra Marathe, William Scherer III, and Michael Scott. Adaptive software transactional memory. In *DISC '05: Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005.

[19] 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*, Jun 2006.