On the Performance of Contention Managers for Complex Transactional Memory Benchmarks

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

***Abstract*—In Transactional Memory (TM), contention management is the process of selecting which transaction should be aborted when a data access conflict arises. In this paper, the performance of published contention managers (CMs) is re-investigated using *complex benchmarks* recently published in the literature.**

Our results redefine the CM performance hierarchy. Greedy and Priority are found to give the best performance overall. Polka is still competitive, but by no means best performing as previously published, and in some cases degrading performance by orders of magnitude. In the worst example, execution of a benchmark completes in 6.5 seconds with Priority, yet fails to complete even after 20 minutes with Polka. Analysis of the benchmark found it aborted only 22% of all transactions, spread consistently over the duration of its execution.

More generally, all *delay-based* CMs, which pause a transaction for some finite duration upon conflict, are found to be unsuitable for the evaluated benchmarks with even moderate amounts of contention. This has significant implications, given that TM is primarily aimed at easing concurrent programming for mainstream software development, where applications are unlikely to be highly optimised to reduce aborts.

I. Introduction

Transactional Memory (TM) [1], [2] promises to ease concurrent programming effort in comparison to fine-grain locking, yet still provide similar scalability and performance. TM has seen a rise in research activity as it became clear that scalable software would be essential to take advantage of future multi-core technology.

In TM, code blocks that access shared data are defined as *transactions*, similar to how they are guarded by locks in traditional explicit concurrent programming. However, in contrast to locks, a TM runtime manages conflicting data accesses between the code blocks, and the developer is freed from the responsibility of orchestrating lock acquisition and release. The TM runtime logs all read and write accesses for each transaction, and compares them to detect conflicts. A *contention manager* (CM) is invoked when two transactions have conflicting accesses, and *aborts* one of the transactions. A transaction *commits* if it completes executing its code block and is not aborted due to conflicts, making its writes to shared data globally visible.

Several CMs have been published [3], [4], [5], [6] that offer a variety of algorithms for selecting the victim transaction to abort. Since their publication in 2004-5, the CMs have not been re-evaluated in the light of new, complex, TM benchmarks. In this paper, we investigate the performance of eight well-known CMs using Lee's routing algorithm [7], [8], and a port of the STAMP benchmark suite [9]. Our investigation reveals several interesting results.

The most important result is that Polka, the published best-performing CM, suffers severe performance degradation when even a moderate (22%) proportion of executed transactions abort. This trend extends to other delay-based CMs investigated. Overall, Greedy and Priority provide the best performance, although Greedy offers stronger progress guarantees.

The paper is organised as follows: Section II describes the CMs, and Section III describes the complex benchmarks used in the evaluation. Section IV presents the evaluation, and Section V further investigates the effect of changing Polka's parameters on its performance. Section VI describes related work, and Section VII summarises the paper.

II. CMs

A CM is invoked by a transaction (the *calling* transaction) when it finds itself in conflict with another transaction (the *opponent* transaction). The CM decides which transaction should be aborted, although delay-based CMs wait for a finite amount of time to give the opponent transaction a chance to commit. The CMs investigated are: Aggressive, Polite (called Backoff in this paper), Eruption, Karma, and Kindergarten from [3], Polka from [4], Greedy from [5], and Priority, a new variant of Timestamp [3]. Of these, the following are delay-based CMs: Backoff, Eruption, Karma, and Polka. Brief descriptions of each CM follow.

Backoff gives the opponent transaction exponentially increasing amounts of time (delay) to commit, for a fixed number of iterations, before aborting it. Default parameters [3]: minimum delay of 2^4ns, maximum delay of 2^{26}ns, and 22 iterations.

Aggressive always aborts the opponent transaction immediately.

Karma assigns a transaction dynamic priority equal to the number of reads performed by it. Karma gives the opponent transaction, for a dynamic number of iterations, a fixed amount of time delay per iteration to commit. If the opponent transaction has not completed after all the iterations of delay, it is aborted. The delay given is 1000ns per iteration, and the number of iterations is equal to the opponent's priority minus the caller's priority.

Eruption, like Karma, assigns dynamic priorities to transactions based on the number of reads. Conflicting transactions with lower priorities add their priority to their opponent, increasing the opponent's priority, to allow the opponent to 'erupt' through any conflicts it has, or may have, to completion.

Kindergarten follows a toy sharing analogy, and makes transactions abort themselves when they conflict with a transaction the first time, but alternates to aborting the opponent if it is encountered in a conflict a second time, and so on.

Polka combines Karma and Backoff by extending Karma to give the opponent transaction exponentially increasing amounts of time delay to commit, before aborting the opponent transaction. The delay parameters used are identical to Backoff's. Additionally, if a conflicting object is being read by several transactions, Polka will immediately abort all of them if the calling transaction wishes to write to the conflicting object.

Greedy aborts an opponent transaction if it is younger or sleeping, otherwise it waits for the opponent indefinitely (i.e., if the opponent is older, and not sleeping). A waiting, or suspended (e.g. during I/O) transaction is marked as 'sleeping'.

Priority aborts the younger of the conflicting transactions immediately. Priority can lead to a transaction never completing if it conflicts with an older transaction that has a fault that prevents it from completing. Greedy provides stronger progress guarantees than Priority by not allowing such a situation if the faulty transaction is suspended.

III. Platform & Benchmarks

Results are obtained on a 4x dual-core (8 core) Opteron 2.4GHz system with 16GB RAM, openSUSE 10.1, and Sun Java 1.6 64-bit with the parameters `-Xms1024m -Xmx14000m`. DSTM2 [10], a software TM implementation, is used to evaluate the CMs. Past research in contention management has also used DSTM2, its variants, or predecessors. In this paper, DSTM2 is set to its default configuration of eager validation, visible reads, and visible writes.

The benchmarks used are Lee's routing algorithm [7], and KMeans and Vacation from the STAMP benchmark suite (version 0.9.5) [9]. All the benchmarks have been ported to DSTM2. STAMP's Genome benchmark has been investigated, but is not presented as it generates very few conflicts on the hardware used in the experiments, and its results give no greater insight than the results from Vacation. As shown in Table I, eight benchmark configurations are used, with a range of transactional conflict rates (contention) are used in this evaluation. The parameters used for each benchmark are those suggested by their respective providers, except KMeansHS and KMeansLS, which we created for quick experiments. Below, the benchmarks are briefly described, and in particular their concurrency characteristics are mentioned with respect to the inputs detailed in Table I.

a) Lee's routing algorithm: is a circuit routing algorithm that automatically connects pairs of points in parallel, without overlapping any existing connections. The application loads pairs of points from an input file (measured execution time excludes parsing of the input file). Threads attempt to find a route between a pair of points by performing a breadth-first search of the grid from the first point, avoiding any grid cells occupied by previous connections. If a route is found, 'backtracking' writes the route onto the grid. Transaction-based routing requires backtracking to be performed transactionally. An early release [11] variant (Lee-TM-ter) removes data from the transaction's read set during the breadth-first search, which reduces false-positive conflicts. Execution is completely parallel, and the amount of parallelism is controlled by the amount of overlap in the connections attempted. The input file used has 1506 connections, i.e. transactions to commit, many of which are quite long, which increases contention and transaction execution time.

b) KMeans: groups a large pool of objects into a specified number of clusters in two alternating phases. A parallel phase transactionally assigns objects to their nearest cluster, and a serial phase re-calculates cluster centres based on the mean of the objects in each cluster (initial cluster centres are random). Execution continues until two consecutive iterations generate similar cluster assignments within a specified threshold. The input files supply a large number of objects to cluster, and thus transactions to execute, but parallelism is controlled by the distribution of objects to the randomised cluster centres. Furthermore, randomised cluster centres result in considerable execution time variance, as observed in Section IV, Figure 1. Transactions are extremely short since they only read cluster centres and assign objects to the closest one.

c) Vacation: is a travel booking database simulation that has operations to book or cancel cars, hotels, and flights on behalf of customers transactionally, and must update the customer's linked list of reservations as necessary. Threads can also modify the availability of cars, hotels, and flights transactionally. The input parameters lead to low contention for the hardware used, and transactions are short since they update the simulated database and customers' linked lists.

IV. Initial Evaluation

Each benchmark configuration is executed using each CM, and using 1, 2, 4, and 8 threads. Each unique combination of benchmark configuration, CM, and threads is called an experiment. Each experiment is automatically terminated after 20 minutes, and when this occurs the associated CM is deemed 'too poor' for the given experiment, and we say the CM has *failed* the experiment. Results are averaged over eight runs of each experiment, except the failed experiments, which are run only three times to reduce doubt.

Figure 1 and Table II show the execution time results. Single thread results are presented to give an idea of execution time variance for the benchmarks, as obviously there is no contention or non-determinism when using a single thread, and thus execution times should be almost identical. This is true for all benchmarks except KMeans, where the randomised initial cluster centres have a significant impact on execution time. For performance comparisons, only the multi-threaded results are of interest, and less so with KMeans due to the large variances observed in it, although KMeans is still important due to the failures seen.

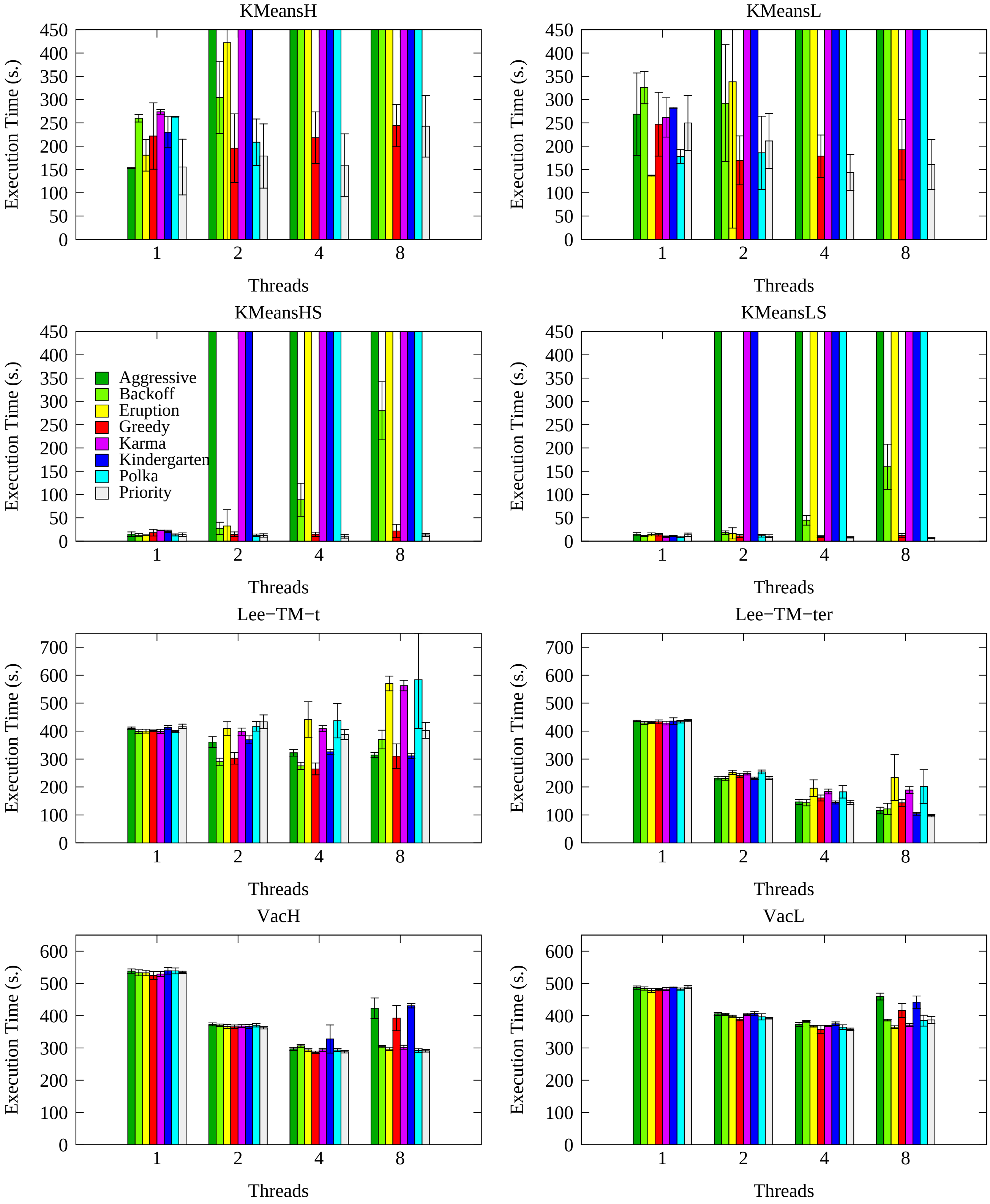

Fig. 1. Execution times with ±1 standard deviation. Times that go beyond the y-axis range are experiments in which the CM failed (to finish within 20 minutes). Less is better.

Configuration Name	Application	Configuration
KMeansL	KMeans low contention	clusters:40, threshold:0.00001, input_file:random50000_12
KMeansH	KMeans high contention	clusters:20, threshold:0.00001, input_file:random50000_12
KMeansLS	KMeans low contention with small data set	clusters:40, threshold:0.0001, input_file:random10000_12
KMeansHS	KMeans high contention with small data set	clusters:20, threshold:0.0001, input_file:random10000_12
VacL	Vacation low contention	relations:65536, percent_of_relations_queried:90, queries_per_transaction:4, number_of_transactions:4194304
VacH	Vacation high contention	relations:65536, percent_of_relations_queried:10, queries_per_transaction:8, number_of_transactions:4194304
Lee-TM-ter	Lee low contention	early_release:true, file:mainboard
Lee-TM-t	Lee high contention	early_release:false, file:mainboard

TABLE I
PARAMETERS USED FOR EACH BENCHMARK CONFIGURATION USED IN THE EVALUATION.

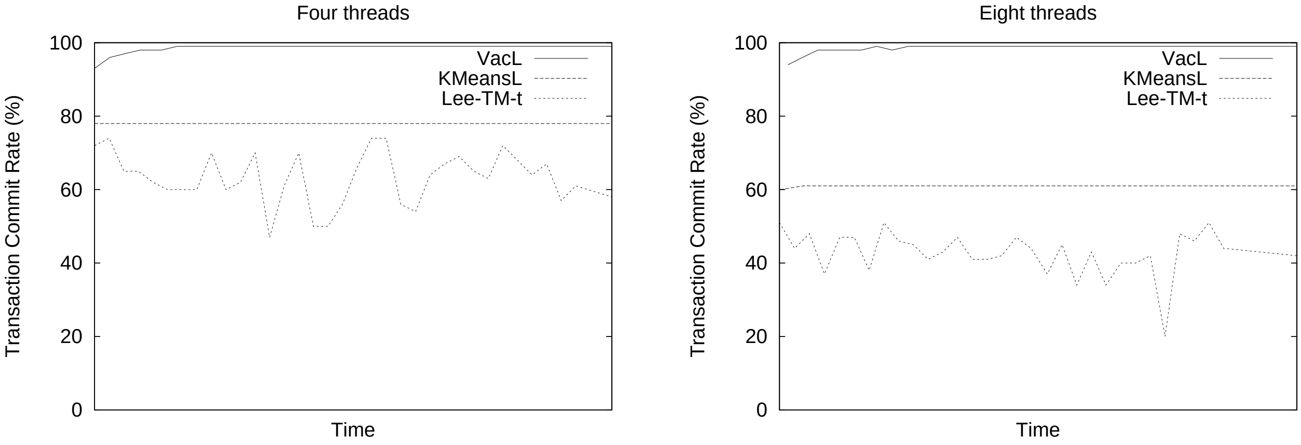

Fig. 2. Sample observed transaction commit percentages (using the Priority CM). VacL has far more commits (i.e., less contention) than Lee-TM-t or KMeansL

The results are mixed, with different CMs showing competitive performance with different benchmarks, reflecting the varying contention and execution characteristics of the benchmarks. For instance, Polka shows good performance in VacH and VacL, but Kindergarten does not, and the opposite is true in Lee-TM-t and Lee-TM-ter, especially as the number of threads increase. In general, a high consistency of good performance is only seen with Greedy and Priority. Averaged either over all threads, or only over 2-8 threads, the performance difference between them is less than 0.6%.

Polka is the published best CM [4], in the past producing best or near-best execution times for all benchmarks with which it has been executed. For VacH and VacL this is certainly the case, but this benchmark results in low contention on the hardware used. Strikingly, Polka is one of the worst in Lee-TM-t, and consistently joint worst in KMeansH and KMeansL. KMeans experiments, and Lee-TM-t exhibit large amounts of contention that increase with the number of threads, as shown in Figure 2. Worryingly, KMeansL at 4 threads has at least a 78% commit rate (using Priority CM, not theoretical best commit rate), but Polka fails to complete. Polka manages to complete execution with Lee-TM-t as there are only 1506 routes to connect transactionally, and thus the number of transactions executed is much fewer than KMeans, which typically executes millions of transactions. In KMeansHS and KMeansLS, which typically take less than 20 seconds to complete with the well-performing CMs, Polka fails to complete in 20 minutes.

Critically, wherever Polka fails, Karma also always fails, but Backoff does not also always fail. As mentioned earlier, Polka combines Karma and Backoff. The difference between the first two and the latter is the number of delay intervals: Backoff has a fixed number of 22, whereas the other two calculate it dynamically. By deduction, the average number of iterations in Polka and Karma must have been larger than 22. We also note that Eruption similarly fails in KMeans experiments, and performs poorly in Lee-TM-t. This suggests that delay-based CMs in general are not suitable for applications that exhibit

░ <= 110% of best execution time
▒ <= 125% of best execution time
▓ > 125% of best execution time, and < 20 minutes
■ > 20 minutes (DNF)

Benchmark	Thread	CM							
		Aggressive	Backoff	Eruption	Greedy	Karma	Kindergarten	Polka	Priority
KmeansH	2	■	304	203	196	■	■	208	**179**
	4	■	■	■	218	■	■	■	**159**
	8	■	■	■	244	■	■	■	**243**
KmeansL	2	■	292	187	**169**	■	■	186	211
	4	■	■	■	179	■	■	■	**144**
	8	■	■	■	192	■	■	■	**161**
KmeansHS	2	■	27.7	14.7	14.6	■	■	12.5	**12.1**
	4	■	88.9	■	14.8	■	■	■	**10.6**
	8	■	279.8	■	21.8	■	■	■	**13.4**
KmeansLS	2	■	18.3	10.7	11.3	■	■	11.6	**10.6**
	4	■	44.8	■	9.7	■	■	■	**8.0**
	8	■	159.7	■	12.0	■	■	■	**6.5**
Lee-TM-t	2	361	**290**	409	303	398	369	417	433
	4	322	276	448	**264**	409	326	437	388
	8	314	370	571	**310**	563	311	584	403
Lee-TM-ter	2	232	**230**	252	241	249	231	254	232
	4	147	**143**	198	161	184	145	182	145
	8	116	122	242	143	189	104	202	**98**
VacH	2	373	371	366	366	368	366	371	**363**
	4	297	306	294	**286**	294	328	293	288
	8	423	304	296	393	302	431	292	**291**
VacL	2	405	404	398	**389**	404	407	396	392
	4	373	382	367	**357**	368	375	364	358
	8	459	386	**364**	416	371	442	384	387

TABLE II
EXECUTION TIMES (IN SECONDS). BOLD INDICATES BEST TIME FOR AN EXPERIMENT (I.E., A ROW).

non-negligible amounts of contention.

Finally, KMeans experiments with 2 threads deserve further attention because in these Polka completes execution with a competitive execution time, and Karma does not. The difference between Polka and Karma are a) that Polka aborts a set of reading transactions immediately if the calling transaction wishes to write, and b) the amounts of time delay per iteration. Although the first point may explain Polka's higher performance, the second point calls into question the choice of parameters used for Polka, and, more generally, whether they were to blame for the poor performance observed in other experiments above. We investigate this further in the next section.

V. Investigation of Polka's Parameters

Polka has two tuning parameters: `LOG_MIN_BACKOFF` and `LOG_MAX_BACKOFF`, which bound the exponential delay. Polka calculates delay for an iteration as 2^n nanoseconds where n starts at `LOG_MIN_BACKOFF`, and increments by one every iteration up to `LOG_MAX_BACKOFF`. A spin loop that calls Java's `System.nanoTime()` is used to determine if the required delay time has passed. In this section, we investigate the performance effect of altering Polka's parameters. Scherer *et al.* [4] do not suggest a method by which the parameters should be calculated so we use the scheme explained below.

Through empirical evaluation we determined the minimum timing accuracy of our system to be 3600±100 nanoseconds. This is significantly higher than the published minimum value of 2^4ns (by using a `LOG_MIN_BACKOFF` of 4), but testing on other x86/Linux platforms similarly gave us minimum accuracies much higher than 2^4ns, and never less than 2500ns. Thus we set `LOG_MIN_BACKOFF` to 11, to give a calculated minimum delay of 2^{11} nanoseconds = 2048 nanoseconds $\approx$ 2 microseconds, but which of course rounds up to the minimum system accuracy. Since the original values were based on SPARC/Solaris, Polka potentially needs parameter re-tuning for every new hardware platform used.

For `LOG_MAX_BACKOFF` we select a range of values based on approximately *half* the average committed transaction execution time for each benchmark. The observed values are shown in Table III . We select `LOG_MAX_BACKOFF` values of 13 (≈8 microseconds), 16 (≈65 microseconds), 19 (≈528 microseconds), and 28 (≈134 milliseconds). The results of the executions are shown in Figure 3, again averaged over eight runs for all experiments, except failed experiments, which are again only run three times.

There is minimal effect of changing the parameters in VacL and VacH, as these have low contention, which leads to the CM being invoked rarely. For the remaining experiments, different parameters give the best performance improvement over the default parameters. Although the improvements are slight, this suggests per-application parameter tuning may be necessary. However, the important results have not changed, and Polka continues to give extremely poor performance in all KMeans experiments with 4 or more threads, irrespective

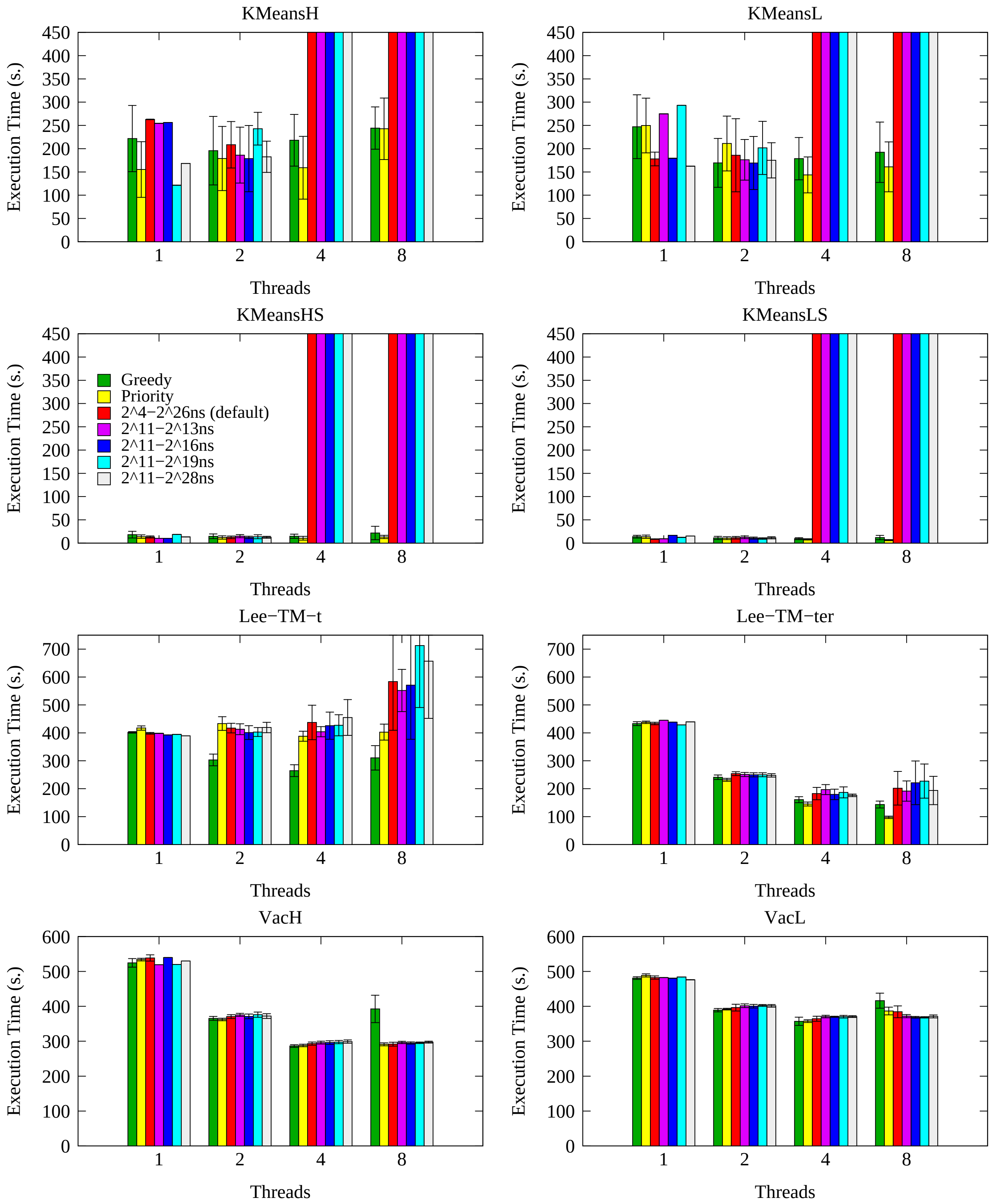

Fig. 3. Execution times with ±1 standard deviation for Greedy, Priority, and Polka with several minimum and maximum delay parameters. Lower is better.

	KMeans*	**Lee-TM***	**Vac***
1 thread	12	264288	126
2 threads	19	330592	167
4 threads	210	380275	265
8 threads	422	524702	537

TABLE III
AVERAGE COMMITTED TRANSACTION EXECUTION TIME FOR EACH BENCHMARK, IN MICROSECONDS. BOTH HIGH AND LOW CONTENTIONS NOT SHOWN AS EXECUTION TIMES IN THE SAME ORDER OF MAGNITUDE FOR EXPONENTIAL DELAY CALCULATION.

of the wide range of tuning parameters used. This strengthens our original hypothesis: delay-based contention management may be unsuitable for applications with appreciable amounts of aborting transactions.

VI. RELATED WORK

Guerraoui et al. [5] developed the Greedy CM, which has provable progress properties, and their evaluation showed Greedy performed on par with Polka. Our results confirm their findings. Scherer and Scott [4] evaluated Polka using six benchmarks in nine benchmark configurations. Three benchmarks added, removed, and queried elements in a set, the fourth implemented a concurrent stack, the fifth a ‘torture test’ that updated all values in an array per transaction, and the sixth an LFU cache simulator. Clearly their benchmarks exhibited contention to provide variation in execution time between CMs, and they found Polka to be a consistent top performer, often by large margins over other CMs. Their investigation differs from ours in one critical way. All the CMs they investigated are delay-based, except Kindergarten, and they did not include Greedy or Priority, as neither had been published. Our investigation found all delay-based CMs and Kindergarten performed poorly compared to Greedy and Priority in benchmarks with appreciable amounts of aborts.

Our previous work in adaptive concurrency control [12], which dynamically changes the number of transactions executing simultaneously with respect to the measured transaction commit rate, resulted in applications’ performance at any number of initial threads being similar to best-case statically-assigned number of threads, for a given CM. Our adaptive mechanism would have had a dramatic positive effect on the performance of Polka in its failed experiments. Additionally, our work in reducing repeat conflicts [13] showed Polka’s performance could be improved in applications that exhibit repeat conflicts.

VII. SUMMARY

This paper re-evaluates well-known CMs (CMs) in the light of newly published complex benchmarks. A number of important findings result from this investigation. In general, we found Priority and Greedy to be joint best-performing CMs, although Greedy provides stronger progress guarantees than Priority.

Although Polka still provides competitive performance in benchmarks with very low contention, the most important finding of our investigation suggests Polka, the established best-performing CM, and in general all delay-based CMs, are unsuitable for the evaluated benchmarks that exhibit even moderate amounts of aborting transactions. Although we do not quantify what is meant by ‘moderate’, in one benchmark Priority executed in 6.5 seconds with an average of 78% of transactions committing (i.e., 22% aborting), whilst Polka failed to complete executing the benchmark in 20 minutes (after which time the execution was terminated). This result has wider implications given that TM is strongly aimed at easing concurrent programming for mainstream software development, where execution is unlikely to be highly optimised to reduce aborts in the general case.

Polka has two tuning parameters, and investigating a range of values concluded there was no benefit in tuning them to improve the extremely poor results seen in KMeans experiments, although tuning led to a degree of performance improvement in the remaining results. However, different parameters provided better performance for different applications, suggesting the need for application-specific tuning. Furthermore, the need to re-evaluate the parameters for every hardware platform used was also highlighted. Conversely, Greedy and Priority have no parameters.

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] 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.

[4] 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.

[5] 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.

[6] Rachid Guerraoui, Maurice Herlihy, Michal Kapalka, and Bastian Pochon. Robust contention management in software transactional memory. In *SCOOL ’05: Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

[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] 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*, pages 196–207. LNCS, Springer, June 2008.

[9] 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.

[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. ACM Press, October 2006.

[11] 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.

[12] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *EUROPAR '08: Fourteenth European Conference on Parallel Processing*, pages 719–728, August 2008.

[13] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Steal-on-abort: Dynamic transaction reordering to reduce conflicts in transactional memory. In *HIPEAC '09: Fourth International Conference on High Performance and Embedded Architectures and Compilers*, pages 4–18, January 2009.