How fair is Fair?

A Software Transactional Memory Perspective

Anvesh Komuravelli*, Samir Sapra* and Jennifer D. Tam*
{akomurav, ssapra, jdtam}@cs.cmu.edu

1 Introduction

Software Transactional Memory (STM) is becoming more popular (in some ways surpassing Hardware Transactional Memory) and currently, a wide variety of STM implementations have become available. As the field of STMs has grown, an important subproblem has begun to gain in importance, namely *Contention Management* [4, 10]. One notable set of contention managers are described by Scherer and Scott [10]. In that relatively early work, they mainly concentrate on a narrower problem called *conflict resolution*: when multiple transactions are in conflict because they are competing for a common memory resource, which should be allowed commit and which should be forced to abort?

On the other hand, as recent work has pointed out, contention management is about a much broader issue. In a way, it relevant for the entire duration of the transaction – not just at commit time. More precisely, there are many aspects to be considered, such as

- conflict resolution (as was being done previously)
- data versioning (*eager* commits, as and when objects are updated, versus *lazy* commits, of all updated objects at the end of the transaction 'block' of code)
- livelock and/or starvation freedom

Very recently, a new contention manager (in the above broader sense) has been proposed by Spear et al. [12]. It uses *(i)* lazy version management, *(ii)* avoids livelock and starvation in practice, and *(iii)* enhances its conflict resolution by incorporating a priority mechanism among the transactions. Although the work compares the new manager with existing ones on *micro*benchmarks, it is not precise and does not emphasize real applications.

One of the *objectives* of this work is to put all the existing contention management techniques in a line and pick the one which works the best for *real* applications. This is motivated by several reasons. Firstly, we did not find any uniform framework for the different contention managers proposed in the literature and the evaluations were on widely different platforms, ranging from varying main memory sizes, varying processor speeds to simulations vs. running on actual multicore machines. We are not sure how useful such a comparison is. Secondly, contention managers were evaluated using different underlying STM implementations, for e.g., DSTM [6] has been used by Ansari et al. [1] for comparing the various contention managers proposed by Scherer and Scott [10] while RSTM [7] has been used for comparing the contention manager proposed by Spear et al. [12] with the same managers proposed by Scherer and Scott. As such, a uniform comparison can not be made.

Our contribution towards satisfying this objective is to compare the existing contention managers using an *RSTM implementation* (the latest version 5) when used by some *real applications*. Now, there are several different contention managers included in RSTM depending on whether the memory locations considered are at the word-based granularity or object-based. As the latest contention manager *Fair* by Spear et al. [12] has not been evaluated properly on real applications. And for the real applications we chose to work with the *STAMP [2] benchmark suite*. Importantly, STAMP is designed for a word-based STM, and the Fair contention manager also assumes word-based granularity.

[1] Author list is purely alphabetical in the last names

At this point we would like to mention that we were quite interested in comparing *Fair* with the object-based contention managers in RSTM. In fact, some recent literature [1] compared the object-based contention managers among themselves using the STAMP benchmarks with an underlying DSTM. They claim to have ported the word-based STAMP to work with the object-based contention managers but they do not describe how. We were also futile in obtaining such information from other sources. There was also a point where we tried to use a different set of benchmarks (STMBench7 [13]) to work with both Fair and the object-based contention managers but we figured out there is a compatibility issue with the older and newer versions of RSTM to work with these benchmarks. We tried a lot to make these benchmarks work with RSTM version 5 (which is the only one having the *Fair* manager) and although we figured out a solution for that, we did not have the time to do the modification. We leave this issue for a possible future work.

Our next objective is to design and implement a novel algorithm for assigning priorities to transactions which will assist in conflict resolution. There exist some priority mechanisms for conflict resolution described by Scherer and Scott [10]. But our focus is on a mechanism in the context of the new contention manager described by Spear et al [12]. The priority mechanism currently included in this manager is concerned only about the maintenance of the priorities and the dynamic changes of the priority assignments to various transactions and is not concerned with how the priorities are assigned to transactions to begin with. It is currently left to the user.

Thus, the key contributions of this work can be summarized as follows.

1. Evaluate the existing *word-based contention manager*s on the RSTM framework for real applications and not just microbenchmarks.
2. Try to build a better *word-based contention manager*.

2 Comparing Contention Managers

To achieve our objectives, we chose to work with STAMP [2] benchmarks which is a collection of real applications specifically written for evaluating STMs (as opposed to applications simply retrofitted to use STM). Our initial goal for this was to concentrate on *conflict resolution* techniques described by Scherer and Scott [10] (which were then called *contention management* techniques). But we later found that a similar comparison among these techniques has already been made by Ansari et al. [1]. Moreover we found a recent advancement in the *contention management* techniques (the modern view of *contention management* which has a broader perspective than just *conflict resolution*) by Spear et al. [12]. But we did not find a detailed comparison of this technique with the already existing ones. So, we chose to focus on comparing the new technique, the *Fair* contention manager, which is now a part of the RSTM suite [7] to the existing word-based contention managers. We give a detailed description of our comparison in Section 4.

3 Making hands dirty - building the new contention manager

As described by Spear et al. [12], the *Fair* contention manager has many good features like *lazy version management*, *extendable timestamps* which makes it practically a better contention manager than the other existing ones in terms of

1. number of aborts
2. throughput (#transactions committed in a given time)
3. livelock-free in practice
4. starvation-free in practice

The last one is achieved by a special mechanism of changing priorities of transactions dynamically (but at some predefined logical points, for e.g., when a transaction aborts).

We believe that the features of *lazy version management* and *extendable timestamps* are key to any good software transactional memory (as also argued in [12]). So, our focus in the development of a possibly better contention manager is on the priority mechanism used among the transactions.

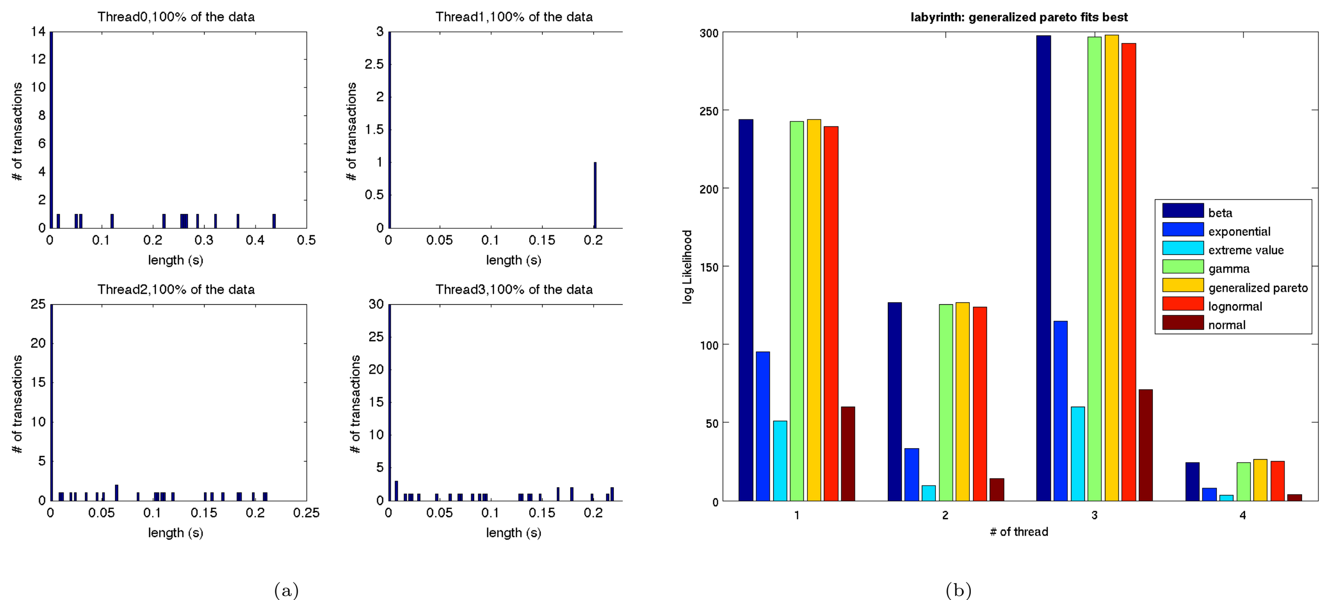
A natural way to come up with a new priority mechanism for transactions is to first understand the behavior of the transactions. So, we explored how the lengths of transactions in the various benchmarks vary when used with *Fair* contention manager setting the priority mode off (i.e. all transactions are of equal priority and no dynamic priority changes). We describe our analysis of the lengths in the following.

3.1 Analysis of transaction lengths

As discussed in Section 4, we first obtained the distributions of transaction lengths in each thread for various benchmarks. We then tried to fit a known distribution with the observed distributions. We made use of MATLAB [8] to plot a log-likelihood of a good number of known distributions and we found that the following distributions compete.

1. Generalized Pareto
2. Beta
3. Gamma
4. Lognormal
5. Exponential (in some cases)

The plots for some of the STAMP benchmarks when run with 4 threads are shown in Figures 1, 2, 3.



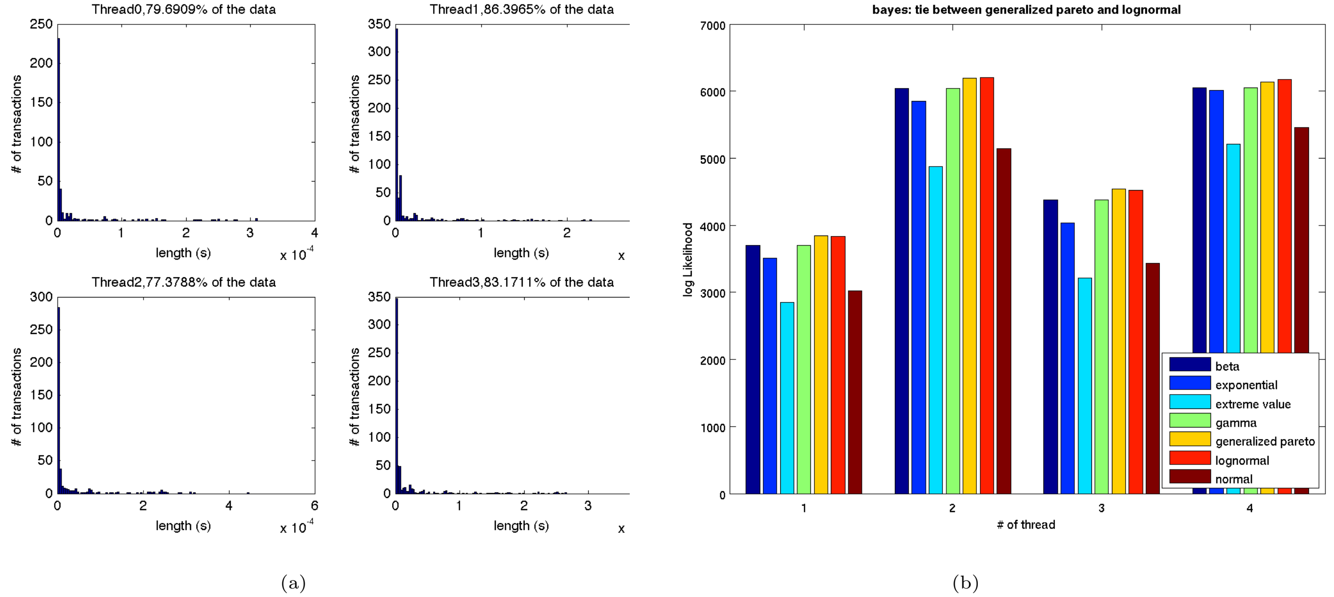
(a) (b)

Figure 1: Labyrinth benchmark - (a)Histogram showing majority lengths of transactions for 4 threads (exact percentages in title of graph). (b)Bar graph of log-likelihood valules for 4 threads.

3.1.1 Details

We examined all STAMP benchmarks using the Fair runtime from RSTM (unprioritized).

In what follows, we show the distributions of transaction lengths for the various STAMP benchmarks, with respect to different parameters. The significant parameter that we vary is the number N of processes that are running concurrently: $N = 1, 2, 4$ and 8. To study the distribution of the transaction lengths, we omitted outliers



(a)

(b)

Figure 2: Bayes benchmark - (a)Histogram showing majority lengths of transactions for 4 threads (exact percentages in title of graph). (b)Bar graph of log-likelihood valules for 4 threads.

(defined as values outside 1.5 $\times$ inter-quartile range. We plotted plotted the histogram plot for the remaining data. We are interested in (i) the visual shape of the plot, and (ii) whether the distributions are similar for each thread both within and across different runs.

Note that as long as there is a large number of transactions, distributions for the different threads are similar, both within and across runs.

3.1.2 Insights

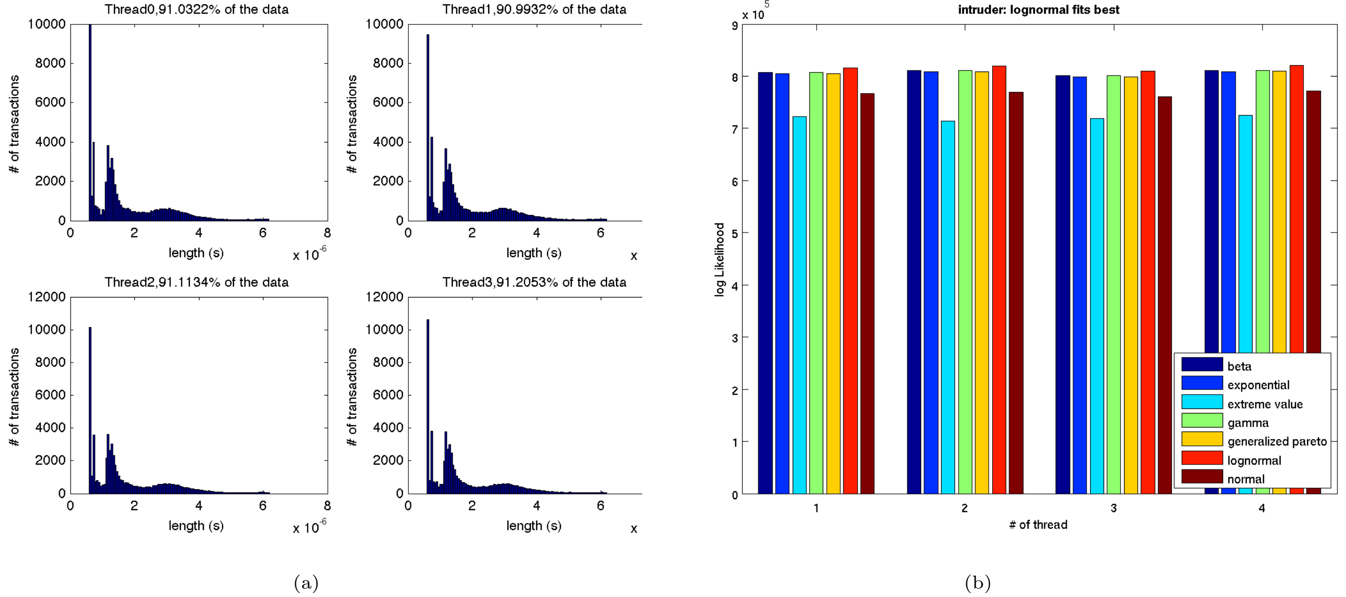
While we do not know of an efficient way to utilize the knowledge that the distribution is Beta, Gamma or Lognormal, we know some important properties of a Pareto Distribution. It is known that Pareto Distribution has a *Decreasing Hazard Rate*(DHR) [5] and it is also known from queuing theory that for Pareto distributions, there is a known optimal scheduling algorithm, *Least Attained Service* (LAS), which favors a process (according to the queuing theory terminology) which has attained the least service so far [14, 9]. Put simply, DHR and LAS in our context of transactional memory imply that whenever a transaction has to be preferred to others, the priority should be based on the service attained so far, or the duration for which the transaction has already run. This gives some insights into how a good priority mechanism should be developed. If we look back into the other distributions which we kept away (Beta, Gamma and Lognormal), the hazard rate functions do not have interesting behaviors (it is only that hazard rate varies but not how).

3.2 Our new priority mechanism

Based on the previous discussion, we first describe what an ideal implementation of the priority mechanism based on transaction lengths should be and we discuss some of its practical limitations and then we discuss our implementation of the mechanism.

3.2.1 The ideal way

The only situations where priority among transactions is used are the ones where a conflict has to be resolved. Some examples of conflicts are



(a)

(b)

Figure 3: Intruder benchmark - (a)Histogram showing majority lengths of transactions for 4 threads (exact percentages in title of graph). (b)Bar graph of log-likelihood values for 4 threads.

1. A committing transaction which wants to modify a memory location realises another transaction has already read the old value.
2. A transaction has read some datum which is later modified by another transaction which has committed.

As described already, we assume a Pareto distribution of lengths of transactions and our priorities are based on the LAS scheme. So, an ideal priority mechanism can be stated as follows.

A transaction t_1 is given more priority than a transaction t_2 if and only if the duration d_1 for which t_1 has been executed so far is strictly less than the duration d_2 for which t_2 has been executed so far.

For simplicity, we call such a duration *running length* of a transaction. A fundamental requirement for this is the ability for each thread to communicate its running length to the other threads (at least, whenever required). We believe this is a non-trivial task (keeping efficiency and throughput in mind) and could be one of the main bottlenecks for a priority mechanism based on transaction lengths, for as far as we know the existing literature does not discuss such mechanisms nor the reasons for not considering such mechanisms. Below, we first discuss some possible solutions and why they are not wise.

Lazy Priority Evaluation : This is probably the easiest solution one can think of. Whenever a conflict needs to be resolved, the thread t which detects the conflict sends out messages to other threads asking for their running lengths (which can be directly communicated or indirectly calculated via a communication of timestamps). Once the running lengths of all the threads are available, t compares them and decides to abort (continue) if there exists (does not exist) another thread whose running length is less than that of t.

Although this mechanism works, this is not a wise choice as we can easily see that it incurs a lot of overhead. Firstly, each message interrupts the execution of the current transaction of the other thread. Secondly, the current thread (t) has to wait for replies from all other threads.

Eager Priority Evaluation : In this solution, we require each thread to update its running length in some global shared memory which every thread has access to. This way whenever a thread t wants to resolve a conflict, it can just look up the running lengths of other threads from the shared memory (which may not be precise depending on the time of the last update) and decide to continue or abort.

This is not a wise option either. As it requires each thread to update the shared memory with its running length and as other threads may detect and want to resolve conflicts at any arbitrary points of time, each thread must do an update periodically. There are two problems with periodic updates. The first one is similar

to the problem with *lazy priority evaluation*. Each thread needs to maintain a timer so that it can do updates periodically which introduces unnecessary interrupts. The second problem is the granularity of the updates. We have seen examples of transactions whose lengths are as small as 100 nanoseconds (on a particular architecture). That hints us that any reasonable use of the mechanism should have a period of the order of a (and hence, an interrupt every) nanosecond which incurs a lot of overhead.

One wise option is to use the time quanta information of each thread which the scheduler of the operating system maintains. This needs to hack the operating system and modify the interface between the threads and the scheduler which is not our current focus. In the following Section 3.2.2 we discuss our workaround this obstacle in our implementation.

3.2.2 Our implementation

As discussed already, the existing mechanism in *Fair* uses a special technique called *automatic priority elevation* which, for practical purposes, avoids starvation. And there is already an efficient data structure in place to assist in the priority mechanism. We chose not to disturb this implementation. We implemented our priority mechanism on top of this existing mechanism to get a hybrid mechanism, as described below.

As calculating running lengths of transactions precisely is difficult (as outlined in the previous Section 3.2.1), we use a slightly different approach where we use *total length*s instead of *running length*s of transactions. The new mechanism can then be stated as follows.

A transaction t_1 is given more priority than a transaction t_2 if and only if the total length of t_1 is strictly less than the total length of t_2.

The only way to calculate the total length of a transaction is to first run it to completion. We note that when a transaction commits in its first run, its length is of no use in future. So, whenever a transaction aborts, we take note of its length just before aborting (which is an approximate of its actual length, the difference being the time taken for the actual commits of the modified memory locations which is very insignificant, or can be significant in the case where the abort is in the middle of the transaction execution in which case the length is updated upon the next abort). The transaction is also required to update a shared memory with its calculated length so that other transactions know of it when they need to resolve a conflict. This shared memory is intelligently maintained so that we only use some bounded memory at any given point of time. There are two variations of the priority mechanism implementation currently in place in RSTM [7] - one which uses *priority read bits* which is an array and the other which uses *Bloom filters* which maintains a list of transactions with priority. We incorporated our new mechanism in both these existing implementations.

When a transaction needs to resolve a conflict, it does the following. It first goes with the existing priority mechanism in *Fair* (using the above mentioned *priority read bits* or *Bloom filters*) and aborts if that mechanism suggests to do so. Otherwise (i.e., the existing mechanism suggests the transaction to continue), the new mechanism comes into play and the transaction first checks if its own length is available (from a previous abort). If so, it compares it with known lengths of other transactions and checks if any of them is smaller than its own length. If it finds at least one such transaction, it aborts and continues, otherwise.

There are assumptions behind such a hybrid mechanism. We assume that transactions are not too long for otherwise, calculation of the length seems to be of little use, intuitively as it takes place at the very end of the transaction which itself takes long time. In other words, the new priorities are not available for use for long time. We also assume high contention among the transactions for otherwise, priorities among transactions are of little help in improving the contention manager.

4 Results

Table 1: Characteristics of various benchmarks considered

	STAMP	**STMBench7**	**RSTM Microbenchmarks**
Realistic application that is not retrofitted	Yes	Yes	No
Long transactions?	No	Yes	No
Word-based?	Yes	unclear	Yes
Object-based?	No	Yes	Yes

Table 2: Architecture dependence of the benchmarks and simulators. *x86 isn't a primary focus area for GEMS. Support is provided for multi-core architectures, but: the memory consistency model is SC, and the cores do not support out-of-order execution., #not been tested for these platforms

	x86_64	x86 (i386)	IA-64	SPARC
STAMP	Yes	Yes	No#	No#
RSTM	No	Yes	Yes	Yes
SIMICS	Yes	Yes	No	No
GEMS	*	*	Yes	Yes

In this section, we discuss the results obtained for our objectives of this work. We first discuss the comparison of the different word-based *contention manager*s of the RSTM suite when used by STAMP benchmarks and then see how our new manager proposed in Section 3 fares when compared with the other word-based managers of the RSTM suite.

4.1 Comparison among contention managers

We will start off with what we intended to do at the beginning of the project and discuss through the hurdles we faced on the way and ultimately, discuss how we worked around those issues and the results we have.

4.1.1 The hurdles on the way

As described in Section 1 our initial objective was to compare the new contention manager *Fair* which is word-based with the object-based contention managers in the RSTM suite. But as we see from Table 1 the STAMP benchmarks [2], as available from their website, can only be used with the word-based managers in the suite.

As our main focus throughout the work is the *Fair* manager, we started off working on the transaction length behavior described in Section 3.1 for the STAMP benchmark suite using *Fair* and we later figured out that STAMP can not be run with the object-based contention managers. We found another benchmark suite STMBench7 [13] which provides us with long transactions (see Table 2) but again, as we discussed in Section 1 we faced problems with portability with the RSTM version we use for *Fair*.

Then, we also had compatibility issues on the way with the different benchmarks and simulators we were planning to use with the underlying hardware support.

We began our project with trying to get the simulators GEMS [3] and SIMICS [11] run correctly. We thought that they would help us do a broader analysis of the comparisons we were about to make. As we can see from Table 2 STAMP and RSTM intersect only with the x86 (i386) architecture. We also got SIMICS simulate this architecture but it did not have a great variety of processors available to simulate. The one which did have x86 architecture was a Pentium 4 20 MHz which was not a great option for our purpose. We also parallely spent a considerable amount of time installing GEMS (the various components of GEMS had so many dependency issues) and found that it is not an interesting option for us (see Table 2).

4.1.2 Our results

We finally chose to work with a real parallel machine with 8 Intel® Xeon® 2.33 GHz cores and 16 GB memory running the Linux kernel version 2.6.19.

We examined all STAMP benchmarks. We report two values of transactions per second

1. 'Per thread' transactions per second: For a given thread, this is defined as the total number of transactions, divided by the total CPU time that thread spent executing transactions. To compute a single aggregate value given multiple threads, a weighted average is used.
2. 'Per process' transactions per second: A lower value that reflects Amdahl's law. It is defined as the total number of transactions executed by the process, divided by the total wall clock running time of the process.

The plots for per thread values are shown in Figures 4, 5 and per process values are shown in Figures 6, 7 for different benchmarks, for $N = 1, 2, 4$ and 8 where N is the number of processors used. The contention managers we evaluated are *Fair* with *visible read bits* for priority, *Fair* with *Bloom filters* for priority, ET, LLT and SGLA, all from the RSTM suite.

We observe the following from the above figures.

1. Fair (using visible read bits) has the highest value of Transactions per second (Tps) in almost all cases with the version of Fair using Bloom filters winning occasionally.
2. The only other competing managers are LLT (Lazy-lazy timestamp) and ET (Extendable Timestamp). For example, LLT wins in the per thread Tps in the case of Bayes and Genome (Figure 4(a) and (b)) while ET wins in the per thread Tps in the case of K-means (Figure 5(a) and (b)). It is interesting to note that Fair employs the feature of Extendable Timestamps!
3. SGLA is the worst of all managers!
4. Similar behavior can be observed for per process Tps.

From the above observations we can conclude that, as suggested by their authors who did only compare microbenchmarks, *Fair* is fair enough!

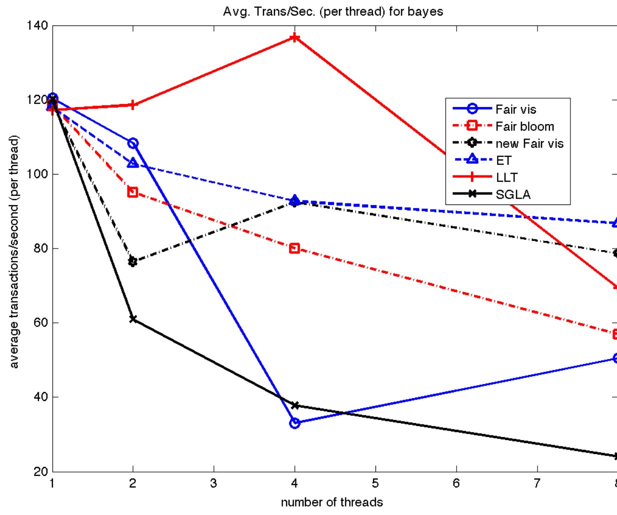
4.2 Evaluation of our new contention manager

Our own implementation is built on top of *Fair* (priority via visibility read bits) as discussed in Section 3.

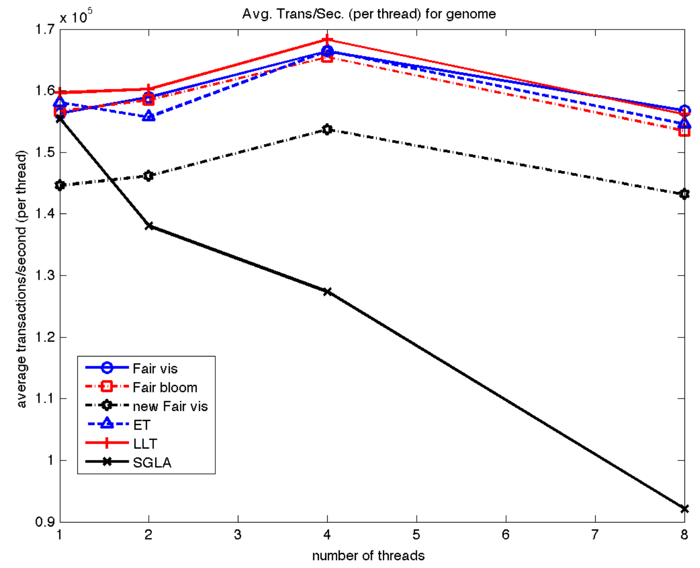
We find that

1. We do remarkably better than the old Fair with visibility read bits in the case of per thread Tps in Bayes (Figure 4(a)).
2. In the case of Yada (Figure 4(f)), we are competing with every other manager!
3. Occasionally, we fare worse, as in the cases of Genome and SSCA2 (Figure 4(b) and (e)).
4. In per process Tps, we fare better than old Fair with visibility read bits in the case of Bayes (Figure 6(a)).
5. In every other per process Tps, we find that we compete with the old Fair for all possibilities of N, the number of threads. This can be attributed to Amdahl's law as gains are diluted when we look at per-process numbers

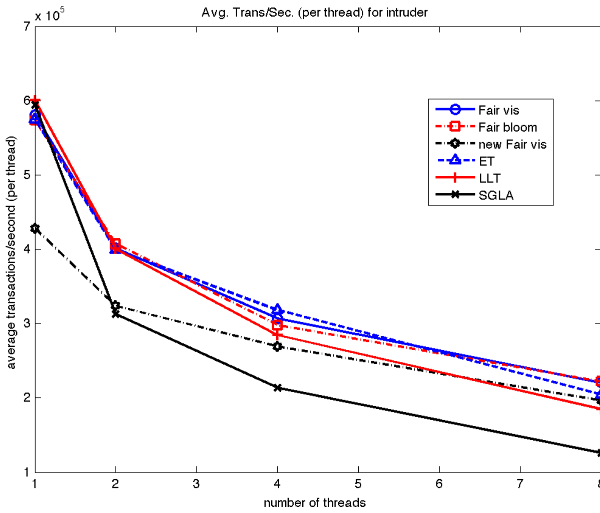
The above strongly says that the coarse-grain running length calculation described in Section 3 does not give us a great deal of improvement, but nevertheless, does compete with the old priority mechanism. Therefore, it is very important to consider the possibility of maintaining fine-grained running lengths, something for which novel algorithms have to be devised.



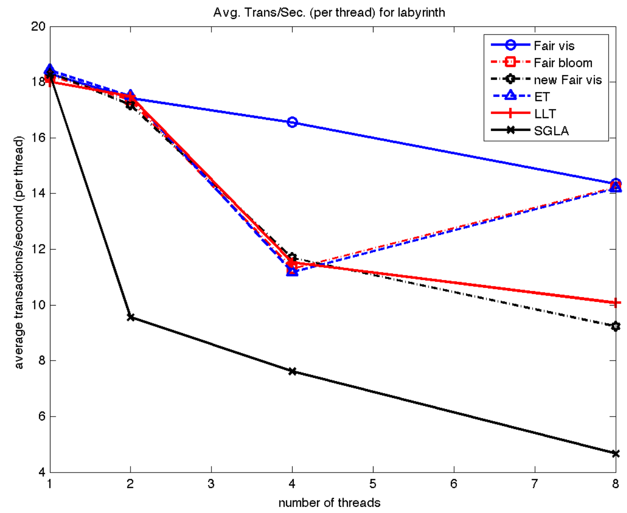
(a)



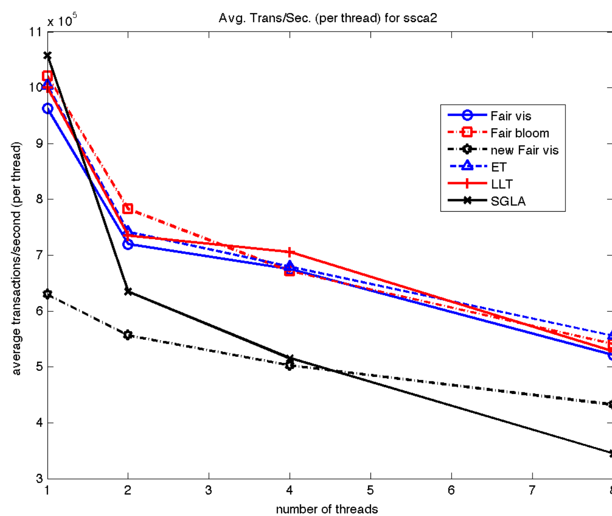
(b)



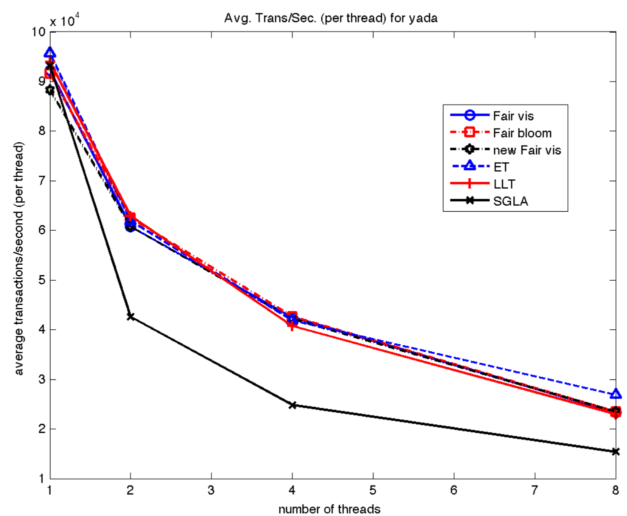
(c)



(d)

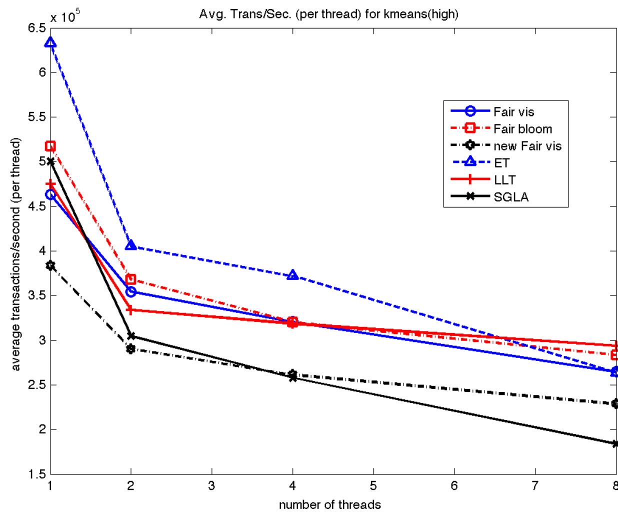


(e)

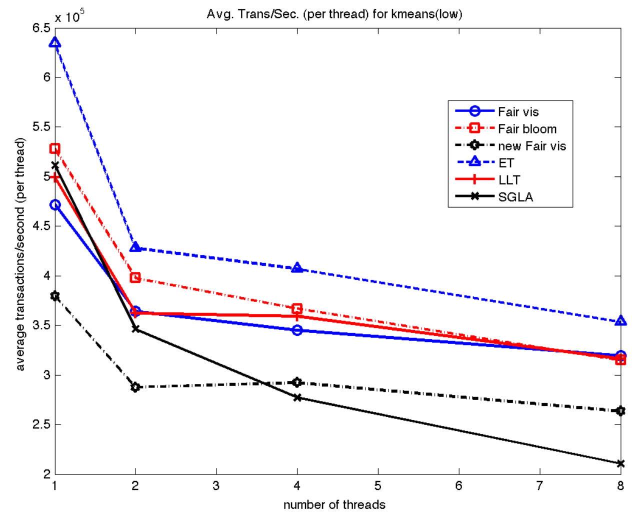


(f)

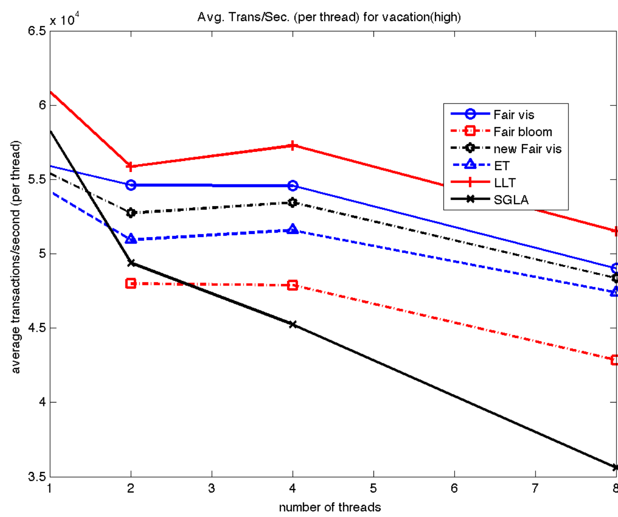
Figure 4: *Per Thread* Transactions per second vs Number of Threads (varied as 1, 2, 4 and 8) for (a) Bayes (b) Genome (c) Intruder (d) Labyrinth (e) Ssca2 (f) Yada benchmarks of STAMP.



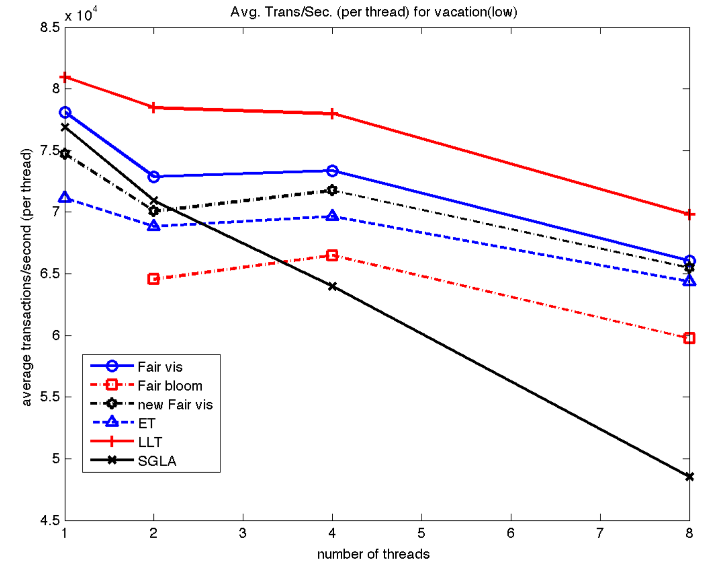
(a)



(b)

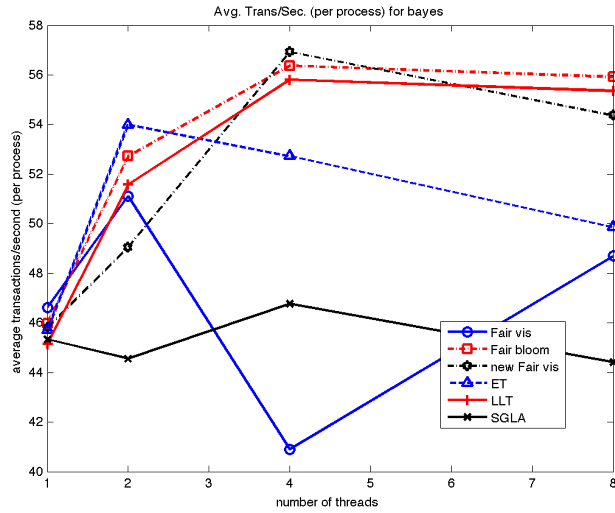


(c)

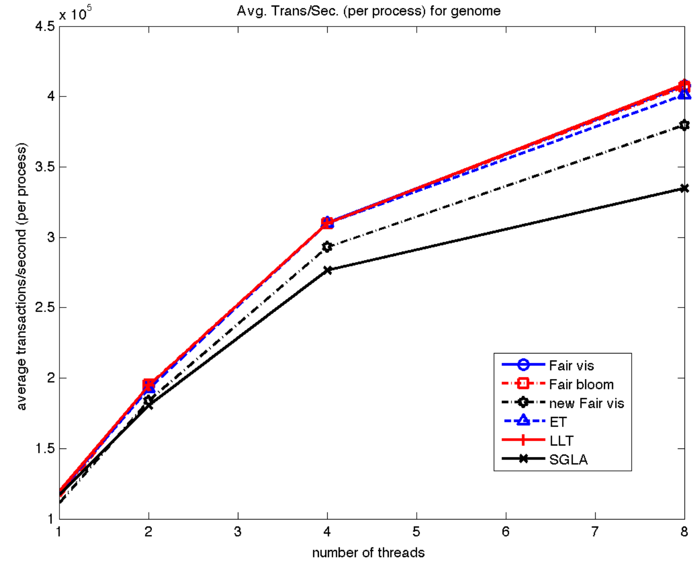


(d)

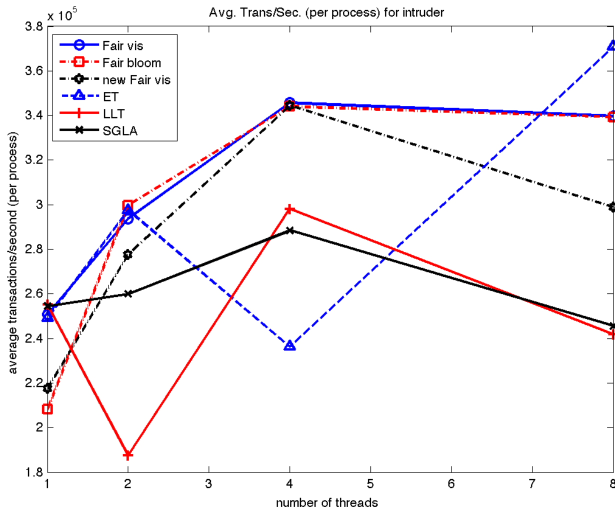
Figure 5: *Per Thread* Transactions per second vs Number of Threads (varied as 1, 2, 4 and 8) for (a) K-means (with high contention) (b) K-means (with low contention) (c) Vacation (with high contention) (d) Vacation (with low contention) benchmarks of STAMP.



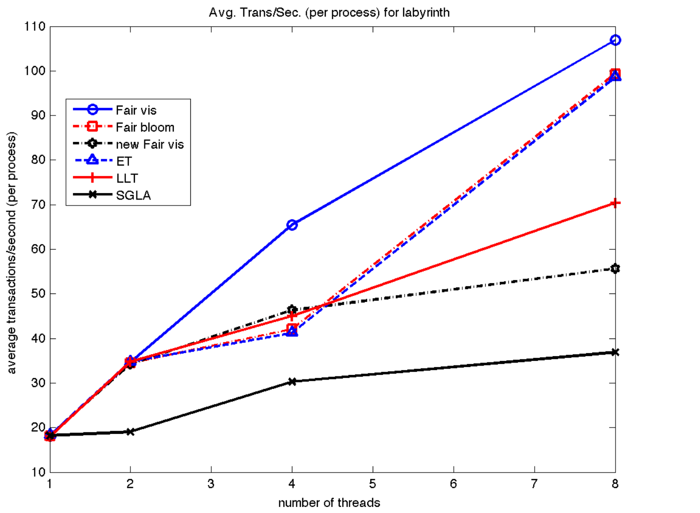
(a)



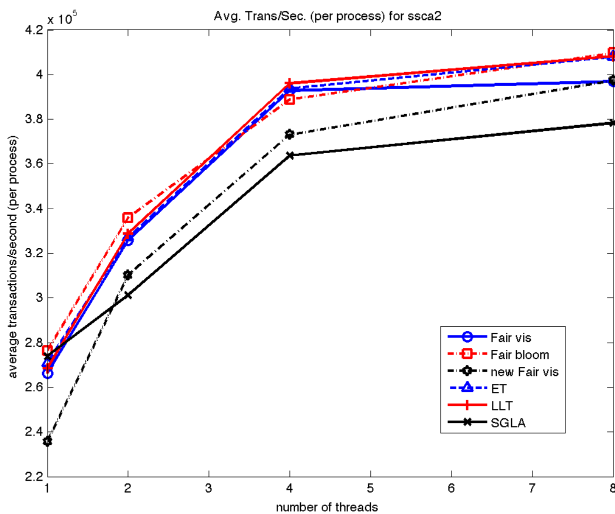
(b)



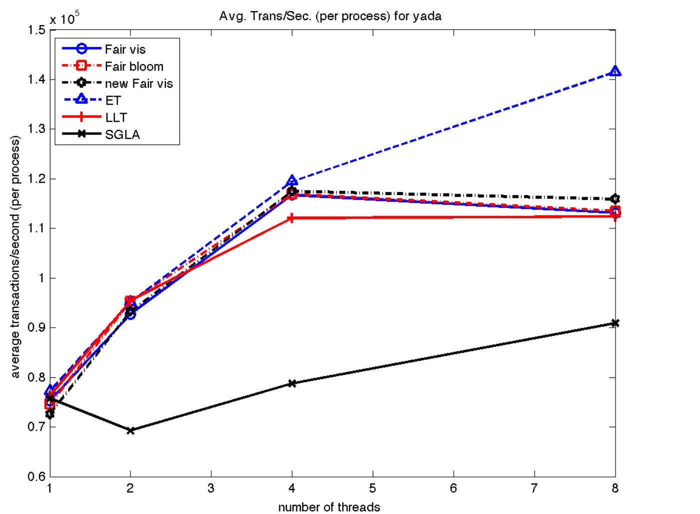
(c)



(d)

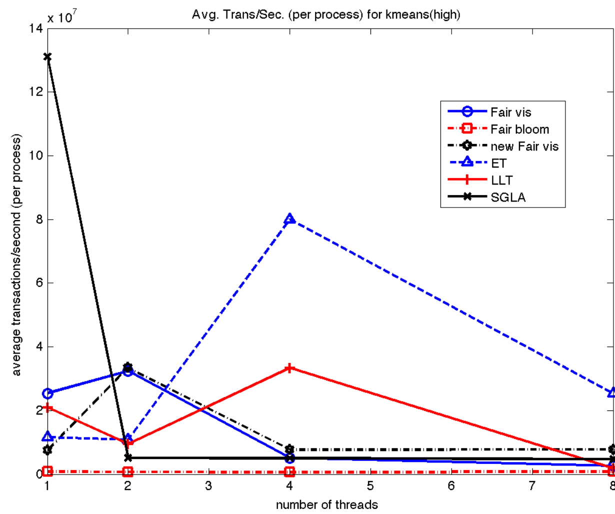


(e)

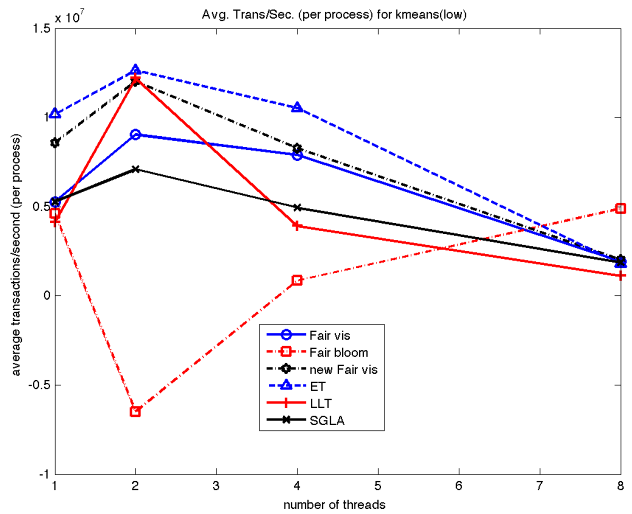


(f)

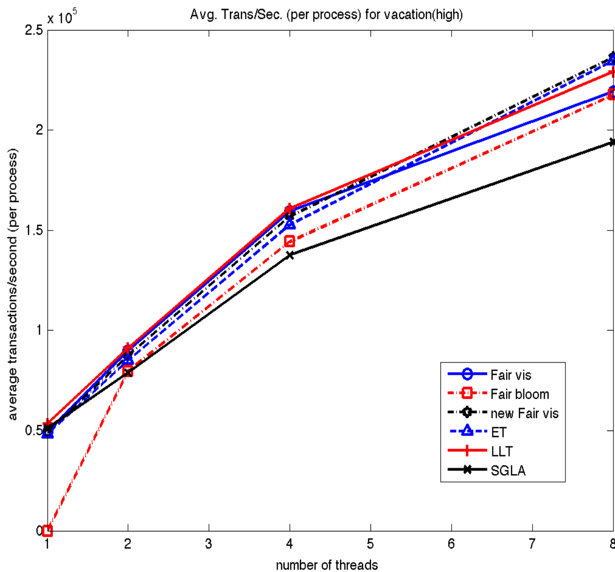
Figure 6: *Per Process* Transactions per second vs Number of Threads (varied as 1, 2, 4 and 8) for (a) Bayes (b) Genome (c) Intruder (d) Labyrinth (e) Ssca2 (f) Yada benchmarks of STAMP.



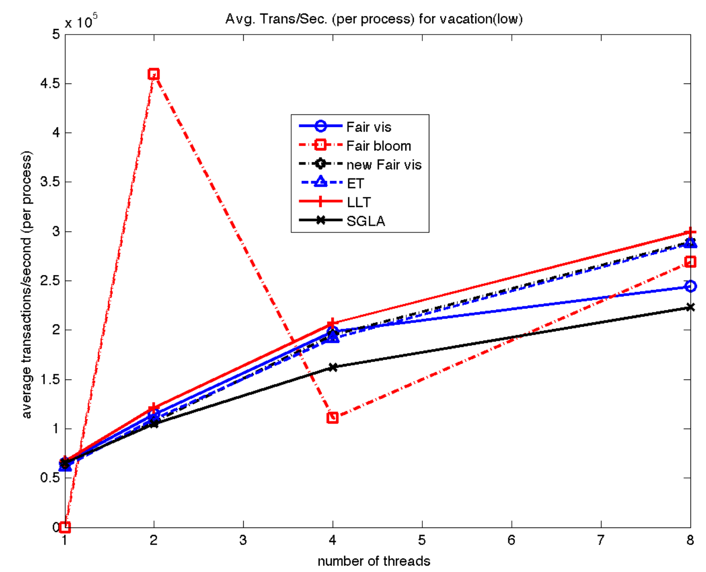
(a)



(b)



(c)



(d)

Figure 7: *Per Process* Transactions per second vs Number of Threads (varied as 1, 2, 4 and 8) for (a) K-means (with high contention) (b) K-means (with low contention) (c) Vacation (with high contention) (d) Vacation (with low contention) benchmarks of STAMP.

5 Conclusions and Future Work

To date, there has been a wide variety of STM implementations presented in the research literature, and there have also been some realistic TM benchmarks presented in the literature. However, cutting-edge STM implementations have not been tested comprehensively on the more widely accepted benchmark suites. With this in mind, we have tried to use the STAMP benchmark suite to evaluate the various contention management techniques implemented in the RSTM STM implementation. Given API and architecture restrictions, we were able to do this for word-based contention managers on the Intel Xeon architecture.

We found that in general, the Fair contention manager performs well. This is consistent with findings presented from evaluating RSTM on microbenchmarks. Occasionally on the STAMP benchmarks, the contention managers LLT (Lazy-Lazy-Timestamp) and ET (Extendable Timestamp) outperform Fair.

Further, we wanted to study the characteristics of the STAMP benchmarks, particularly with respect to transaction duration length. Based on this, we wanted to evaluate the performance of a different contention management strategy, i.e. to commit the only shortest transaction when there is a conflict. This strategy works well, both theoretically and in practice, when the workload follows a Pareto.

The distributions we observed did not visually match any traditional elementary probability distributions. For example, the distributions we saw appeared to be mixtures of a elementary probability distributions. While Pareto was not always the best fit overall, it came close to being the distribution with highest likelihood of matching the data.

We found that that it is difficult to implement the contention management scheme we set out to study, but we were able to implement an approximation. This scheme was comparable to Fair in performance, but only rarely did it significantly outperform Fair. Without investigating the possibility of fine-grained timestamps, there seems little possibility for optimizing this scheme significantly.

There are several possible directions for future work, including: (i) fitting observed distributions to *mixtures* of elementary probability distributions; (ii) considering criteria besides length while designing new contention managers; (iii) using simulators to evaluate RSTM on STAMP, in order to get very precise measurements (iv) comprehensively evaluating the RSTM implementation on the STMBench7 benchmark.

6 Division of work

We all believe that we did an equal share in this project and request the evaluators to give equal credit to each of us, i.e., (100/3)%.

References

[1] M. Ansari, C. Kotselidis, M. Luján, C. Kirkham, and I. Watson. Investigating contention management for complex transactional memory benchmarks. In *MULTIPROG '09: Proceedings of the second workshop on Programmability Issues for Multi-Core Computers.* http://multiprog.ac.upc.edu/multiprog09/resources/proceedings2009.pdf, 2009.

[2] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[3] GEMS. *http://www.cs.wisc.edu/gems/*.

[4] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 258–264, New York, NY, USA, 2005. ACM.

[5] M. Harchol-Balter. Task assignment with unknown duration. *J. ACM*, 49(2):260–288, 2002.

[6] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262, New York, NY, USA, 2006. ACM.

[7] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT '06: Proceedings of the first ACM SIGPLAN Workshop on Languages, Compilers and Hardware Support for Transactional Computing*, New York, NY, USA, 2006. ACM.

[8] MATLAB. *http://www.mathworks.com/.*

[9] R. Righter and J. G. Shanthikumar. Scheduling multiclass single server queueing systems to stochastically maximize the number of successful departures. *Probability in the Engineering and the Informational Sciences*, 3:323–333, 1989.

[10] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[11] SIMICS. *https://www.simics.net/.*

[12] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. *SIGPLAN Not.*, 44(4):141–150, 2009.

[13] STMBench7. *http://lpd.epfl.ch/site/research/tmeval.*

[14] S. F. Yashkov. Processor-sharing queues: some progress in analysis. *Queueing Syst. Theory Appl.*, 2(1):1–17, 1987.