Providing QoS in Contention Management for Software Transactional Memory

1st Sherif F. Fahmy  
Computer Engineering  
AASTMT  
Cairo, Egypt  
fahmy@vt.edu

2nd Zakaria Senousy  
Computer Engineering  
AASTMT  
Cairo, Egypt  
zakaria.senousy@gmail.com

3rd Ahmed F. Amin  
Computer Engineering  
AASTMT  
Cairo, Egypt  
afahmy1610@gmail.com

Abstract—We study the issue of providing quality of service guarantees to software transactional memory. Specifically, we attempt to quantify the type of quality of service (or QoS) guarantees that we can provide using only contention management in software transactional memory. We then present three different algorithms for providing these QoS guarantees and show analytically and empirically that they do in fact operate as expected.

Index Terms—software transactional memory, quality of service, contention management

I. INTRODUCTION

As Moore’s Law begins to reach its physical limitations, computer designers have increasingly begun to look at ways other than increasing clock rates to improve performance. One such alternative is the use of multi-core architectures, where instead of increasing the speed of a single core, several cores are included on-chip to make a more powerful processor. Of course, sequential programs, programs written for the single core era, do not gain any performance benefits from the presence of this increasing number of cores unless they are specifically re-written to take advantage of this new trend in CPU architecture. However, by converting sequential programs into parallel programs to take advantage of the increasingly multi-core CPU architecture landscape, one is faced with the new problem of concurrency control in this massively parallel environment. The traditional method of using locks does not scale well as the number of cores increases since it is very difficult to reason about the correctness – particularly deadlock freedom – of massively parallel programs. Toward that end, many alternatives to lock-based concurrency control have been developed.

In this paper, we concentrate on one of these alternatives, software transactional memory (or STM) [8]–[10]. STM is a software version of transactional memory, an idea that also has hardware solutions [2]–[4], [13]. STM treats accesses to shared memory like database transactions, allowing us to optimistically allow two or more threads to make concurrent modifications to memory locations and only rolling back the operations of those threads that conflict with each other. Since most memory operations do not conflict with each other, this allows programs to take advantage of increased concurrency and only suffer the overhead of rollbacks and retries when absolutely necessary. It also reduces the intellectual complexity of the program as it abstracts away the need of thinking about concurrency and allows the programmer to concentrate on the correctness of their programs instead. In that sense, STM is to locks what automatic garbage collection is to programmer-defined memory management, an abstraction that frees the programmer from low-level details and instead allows him/her to concentrate on the logic of the program being written [5].

When conflicts between two memory transactions occur, it is the responsibility of a part of the STM known as the contention manager (CM) to handle the conflict and determine which of the transactions would be allowed to continue and which have to retry. In this paper we concentrate on the contention manager and attempt to design new contention management algorithms that would allow STM to provide some QoS guarantees to the programs that use it. In particular, we attempt to determine whether transactions with different priorities can expect the CM to respect their respective priorities and deliver results consistent with this. Toward that end, we develop three different algorithms that attempt to provide QoS guarantees in STM. Our results show that the QoS aware contention managers designed outperform their non-QoS aware counterpart, and that postponing abort decisions for a while also contributes to better results. The rest of this paper is organized as follows, Section II contains a brief review of the literature, Section III formalizes the system model used in the paper and the QoS guarantees that we can provide, Section IV contains a description of the proposed algorithms, Section V provides some theoretical results pertaining to the algorithms, Section VI contains the experimental results obtained and Section VII concludes the work.

II. LITERATURE REVIEW

Many papers have been published that discuss the issue of contention management in software transactional memory [1], [3], [6], [10]–[12], [14], but, to the best of our knowledge, this is the first work that attempts to quantify the QoS guarantees that can be provided system-wide when STM is used as the underlying concurrency paradigm while developing concurrent programs.

Of the many contention managers discussed in the literature, we compare our result with the two most relevant to the
work done in this paper, namely the Aggressive Contention Manager and the Polite Contention Manager. The next couple of subsections provide a brief overview of these two contention managers.

A. Aggressive Contention Manager

In this contention manager, a transaction that finds a conflict with other transactions immediately aborts all conflicting transactions and commits. Naturally, this approach has a number of shortcomings. Chief among which is the fact that no parameters are used to intelligently choose which transaction to abort. This means that it is impossible to reason about the behavior of such a contention manager since there is no way to determine a priori which transactions will abort and which will commit in the presence of conflict. In addition, since the contention manager does not provide a “grace period”, but immediately aborts all other transactions, it significantly reduces throughput and may result in livelock as a transaction repeatedly attempts to do work and gets aborted. However, it is a very simple strategy, and thus has very low computational complexity. In this paper, we attempt to augment the Aggressive Contention Manager with a very simple heuristic that will allow it to intelligently choose which transaction to abort and which to commit. While adding this simple heuristic, one of our goals was to keep the modified algorithm as simple as possible so as not to detract from the main benefit of this contention manager: namely, extremely low computational overhead.

B. Back-off (polite) Contention Manager

The Exponential back-off contention manager attempts to solve the livelock problem of the Aggressive manager by delaying the decision to abort a transaction by a random period of time. Taking inspiration from the exponential back-off algorithm used in Ethernet collision avoidance, when two conflicting transactions occur, the transaction that detects the collision backs-off.

The rationale behind this is to give enough time for the conflicting transaction to complete. The algorithm is referred to as exponential back-off because the duration of the back-off period is $2^{n+k}$ nano-seconds, where $n$ is the number of retries until now, and $k$ is some positive constant. Implicit in this equation is the fact that a transaction that detects conflict backs-off several times. Again the rationale behind this is to give the conflicting transaction a chance to commit successfully. If after $n$ back-offs, the conflicting transaction still hasn’t committed, it’s aborted. In this paper, we make modifications to the Polite Contention Manager that attempt to give it a better ability to differentiate among transactions with different QoS parameters.

III. QoS OVERVIEW

Before we discuss the details of the algorithms developed in this paper, we first attempt to discuss the possible QoS guarantees that we can provide in STM. Note that the current work does not take into account scheduling algorithms, but only considers contention managers. Therefore, it is rather difficult to make arguments about the overall QoS guarantees that can be provided. Specifically, we cannot claim that, for example, the entire system will work towards enforcing a certain QoS policy – since we have no control over the scheduling policy. Rather, all we can claim is that when conflicts between transactions occur, these conflicts will be resolved in such a way that attempts, as much as possible, to obey the QoS of the transactions involved in the conflict.

A. System Model

In this paper, we consider a system that is divided into a set of threads, each of which performs a number of transactions. Let us denote the system as $S$, we can then say that

$$S = T_1, T_2, ..., T_n$$

(1)

Where $T_i$ is the $i^{th}$ thread in system $S$, and the system has $n$ threads.

We further assume that each of these threads, $T_i$, performs a number of transactions, $T_{a\text{i}}$. Thus we can say that

$$T_i = T_{a1_i}, T_{a2_i}, ..., T_{aij}$$

(2)

In Equation (2), $j$ represents the number of transactions in each thread. In our model, we assume that each of the threads, $T_i$ in the system have a certain priority $pr_i$, and that all the transactions within this thread have the same priority.

B. QoS Metrics

We develop two metrics for measuring whether or not our proposed algorithms meet the QoS requirements set by the user of the system. Let us again restate that we can only make claims about how conflicts are resolved, since we do not consider the underlying scheduling algorithm in any way.

The first metric we consider is the Conflict Resolve Ratio, or $CRR_i$. We define the $CRR_i$ as the number of times that conflicts involving transactions of $T_i$ will be resolved in favour of transaction belonging to thread $T_i$. It is our contention that a well designed Contention Manager that respects QoS requirements would produce $CRR_i$s that are proportional to the relative priorities of the threads in the system.

In the simplest case, if $S$ contains only two threads, $T_1$ and $T_2$, we expect that

$$\frac{pr_1}{pr_2} = \frac{CRR_1}{CRR_2}$$

(3)

We do not expect to be able to derive an explicit ratio for more than two threads per process, but, in general, the $CRR_i$s should at least have some relation to the relative priorities of the threads in the system. Section V contains a more formal statement of this fact.

The second metric is the Accumulated System Benefit (or $ASB$). This is not a per thread metric, but rather a system-wide metric. We consider that every time a transaction that belongs to a thread, $T_i$, with priority $pr_i$ commits, the system receives a benefit of $pr_i$. The total benefit accrued by the system is the
The sum of these individual benefits. We refer to this sum as the ASB. This can be mathematically expressed as

$$\text{ASB} = \sum_{i=1}^{n} \sum_{T a_{ij} \in T_i \text{ that commits}} p r_i$$

(4)

Where $n$ is the total number of threads in the system and $T a_{ij}$ is the $j^{th}$ transaction in thread $i$. It is our contention that a well-designed QoS contention manager should work to maximize, as much as possible, the value of ASB. To make the ASB of transactions comparable, we normalize the ASB by calculating it as a percentage of the total benefit that would have accrued to the system if all started transactions had successfully committed, we call this normalized value $N \text{ASB}$—for Normalized Accumulated System Benefit.

### IV. PROPOSED SOLUTION

In this section of the paper, we present our three proposed solutions to the QoS problem in contention management. As previously mentioned, we designed three different algorithms. The three algorithms can be broadly divided into two categories: aggressive algorithms, and cautious algorithms. The aggressive algorithms make immediate decisions. Once two transactions conflict, one of them is immediately terminated. No chance is given for them to resolve their conflict by having one of the transactions back off for a bit, instead, an immediate decision is taken. In the cautious category of algorithms, when conflict occurs, one transaction back off to provide the opportunity for the other transaction to complete. After it backs off, it attempts to complete its transaction again. It is then and only then that it aborts an enemy transaction if it is still in conflict. The rest of this section explains the three developed algorithms in detail.

#### A. Aggressive QoS CM

The Aggressive QoS CM algorithm is a modification of the Aggressive CM. In this modified algorithm, instead of one transaction immediately terminating the other when conflict is detected without considering any other factor, the two conflicting transactions compete for success or failure based on their relative importance. For example, assume that two transactions $T a_{12}$ and $T a_{23}$ conflict. Further, assume that $p r_1 = 20\%$ and that $p r_2 = 80\%$, the conflict between these two transactions is resolved $20\%$ of the time in favour of $T a_{12}$ and $80\%$ of the time in favour of $T a_{23}$.

In line 2 of the algorithm, the ratio of the priorities of the two transactions that are in conflict is calculated. So, to continue the example in the previous paragraph, the ratio 2:8 is computed. A random number between 0 and 9 is then generated in line 3. In line 4, this range of 0 to 9 is divided according to the ratio 2:8, which means that from 0 and 1 become one category, and from 2 to 9 becomes the second. The ranges are assigned, in line 5, to the transactions according to their ratio, so the 20\% importance transaction is assigned the range from 0 to 1, and 80\% transaction is assigned the range from 2 to 9. Since the random number is generated from an IID, independent and identical distribution, it will fall 20\% of the time in the range 0 to 1 and 80\% of the time in the range 2 to 9. The if statement in line 6 then determines in which of the two ranges the random number lies, and the transaction in whose range the number lies wins while the other is aborted. Thus, the contention between these two transactions is resolved in a ratio proportional to their respective priorities.

#### B. Greedy Aggressive QoS CM

In this subsection, we introduce the second algorithm we developed to provide QoS guarantees for CM. This algorithm belongs to the aggressive category of algorithms, in that the losing transaction is aborted immediately. In the previous algorithm, we only considered the priorities of the transactions when determining which one to abort. However, we believe that it may be possible to provide even better performance if we also take into account the remaining amount of work in each transaction. Specifically, considering that we are performing an experiment in a fixed interval of time, we believe that it would be beneficial for the algorithm, in terms of total benefit accrued to the system (as measured by $N \text{ASB}$ described in Section III), to make greedy decisions. That is, we changed the algorithm to calculate the benefit that would be accrued to the system for each unit of computation performed in each transaction and chose to favour the transaction that provided the most amount of benefit for the least amount of work.

In order to accomplish this, we need some measure of work for each transaction. We chose the number of memory references performed in each transaction as the measure of work done. When two transactions conflict, we calculate the total remaining work as the total number of memory references to be performed in a transaction minus the number of memory references that have occurred so far. We designate this as the

$$\text{Remaining Work}$$

of the transactions. Since we already have the importance of each transaction we can use this information to calculate the benefit that will accrue to the system for the time dedicated to perform each of the remaining memory references, $RB$ (Remaining Benefit), as follows

$$RB = \frac{\text{Priority}}{\text{Remaining Work}}$$

(5)
We now have a notion of the benefit that will accrue to the system for each unit of computation dedicated to making one memory reference in each transaction. We greedily favour the transaction for which the most benefit would accrue for performing a single memory reference, thus, in effect, using a “return on investment” metric to choose which transaction to favour.

**Algorithm 2 Greedy Aggressive QoS CM**

1: procedure RESOLVECONFLICT(Transaction Me, Transaction Other)
2: Calculate RemaingWork for both transactions
3: Calculate RB for both transactions (Equation (5))
4: if Me.RB < Other.RB then
5:       Me.Abort
6: else
7:       Other.Abort

As can be seen, the algorithm is nearly identical to the Aggressive QoS algorithm, but we consider the “return on investment” instead of the absolute priority when deciding which transaction to favour. We believe this will produce better results by allowing the system to minimize work while maximizing benefit.

**C. Greedy Polite Cautious QoS CM**

In this section of the paper, we introduce the Greedy Polite contention manager. Which is basically the Greedy Aggressive contention manager with a period of backoff added. The rationale behind adding this backoff period is to give the losing transaction an opportunity to commit instead of aborting it immediately. This is done by allowing the transaction to wait a bit until the cause of the conflict no longer exists. If the cause of the conflict still exists after the backoff period, then and only then is the transaction aborted. Algorithm 3 depicts this idea.

**Algorithm 3 Greedy Polite Cautious QoS CM**

1: procedure RESOLVECONFLICT(Transaction Me, Transaction Other)
2: Calculate RemaingWork for both transactions
3: Calculate RB for both transactions (Equation (5))
4: if Me.RB < Other.RB then
5:       Let Me backoff for a period of time
6:       Try to continue Me
7:       Otherwise Me.Abort
8: else
9:       Let Other backoff for a period of time
10:  Try to continue Other
11:  Otherwise Other.Abort

**V. THEORETICAL ANALYSIS**

In this section of the paper, we attempt to formalize some theoretical properties of the proposed algorithms.

**Theorem 1.** If only two threads are competing in a system using the Aggressive QoS CM, the ratio of their CRR will be equal to the ratio between their priorities as the number of conflicts between them approach infinity. That is, Equation (3) holds true for this algorithm.

**Proof.** Consider Algorithm 1, lines 2 to 5 essentially generate an IID random number in a certain range and divide the range in proportion to the relative priority of the two transactions that conflict. Depending on which range the random number falls in, line 6, one of the transactions is aborted and the other is committed. Since all numbers in a range occur with equal probability in an IID random number generator, and the size of the range is divided in proportion to the relative priority of the two transactions, the probability that a certain transaction is committed in any given conflict is exactly equal to its priority.

Therefore, as the number of times that these two transactions conflict approaches infinity, the observer will notice that the conflicts between them are resolved in favour of each one in exactly the same value as their priority. Given the definition of CRR in Section III, this concludes the proof.

**Corollary 1.** In practice, the CRR will not be exactly the same as the priority for the case of two transactions competing according to Aggressive QoS CM, but will be relatively close to it unless the experiment is run infinitely long.

**Proof.** This proof follows naturally from Theorem 1, but let us provide a counter-example to further elucidate this. Consider again the situation of two transactions with priorities 20% and 80%. Even though in the long run 20% of the conflicts will be resolved in favour of the first one and 80% in favour of the second, in the short run, say two conflicts, the result can only be either 50% for each transaction or 0% to 100%. The first case being if one of the random numbers generated falls within the range of one transaction and the other falls within the range of the other, and the second case is if the random numbers both fall into the range of only one transaction. In the long run, 20% of the random numbers will be in the range of the 20% importance transaction, while 80% will fall into the range of the 80% importance transaction, but in the short run this may not be so.

**Theorem 2.** For the Aggressive QoS CM, if the number of threads in the system is more than two, then there is no way to determine the exact percentages in which the conflicts will be resolved, but we can expect them to be at least in the same relative order as the relative priorities of the threads in the system after a sufficiently large period of time. That is, Equation (3) does not hold for this case.

**Proof.** For more than two threads, it is impossible to tell beforehand which threads will collide with which (in the case of two threads, this is trivially known as there are only two threads to collide in the first place). Since we do not know which pair of threads will collide, or if certain pairs of threads will collide at all, it is impossible to accurately compute the exact percentage by which each thread will be...
favoured during conflicts. But we can say with certainty, that if two transactions conflict, the conflict between them will be resolved in proportion to their relative importance. If enough conflicts occur, after a sufficiently large period of time, the bias that the algorithm shows to higher priority threads should show up in their CRR, even if it does not exactly match their priority.

**Theorem 3.** The Greedy Polite QoS CM and the Greedy Aggressive CM will always work to maximize, as much as possible, the value of NASB in contrast to all the other algorithms.

*Proof.* First, note that the experiments, and real-life code, run in a finite amount of time. Therefore, there is a fixed duration of time that needs to be made use of to obtain the maximum benefit, NASB, for the system. Thus, each decision should be designed to accrue the maximum benefit to the system in the smallest period of time. Since Greedy Polite QoS and Greedy Aggressive are the only algorithms that measure the benefit of dedicating the necessary computational time to perform each memory reference (line 4 of Algorithm 3 and Equation (5)), they are the only ones that will maximize the benefit accrued at each decision point. By maximizing the benefit at each decision point, the total benefit, which is the sum of the benefit of each local decision, is also maximized. Thus, Greedy Polite QoS CM and Greedy Aggressive Algorithms will always attempt to maximum NASB at each step. This concludes the proof. □

**Claim 1.** We claim that the cautious category of algorithms will always outperform the aggressive category of algorithms due to the extra diligence taken by this category to allow even losing transactions one last chance before aborting them.

*Proof.* We have no rigorous proof for this claim, but we attempt to verify it experimentally in the next section of the paper. □

### VI. EXPERIMENTAL RESULTS

This section of the paper contains an overview of the results we obtained in this work. First, a brief description of our experimental setup. We use DSTM2 [7], a flexible STM library that provides a modular way for us to implement different contention management techniques. The library uses the Java programming language, and our proposed solution was implemented in that language.

We conducted our experiments on a Windows 10 (64 bit) computer, equipped with an Intel i5 quad-core processor clocked at 2.5GHz. The open source implementation of DSTM2 [7] contains many benchmarks that can be used to measure the performance of the various parts of STM, we chose to use the Linked List benchmark to test our contention manager.

#### A. Aggressive QoS CM

In this part of the paper, we describe the experiments we performed on the Aggressive QoS CM. We performed two experiments in this section. First, we tested whether the Aggressive QoS CM performs better than the Aggressive CM of the literature in terms of CRR, and then we tested whether Theorem 1 and its corollary, Corollary 1.

For the first experiment, we ran DSTM2 with its stock aggressive contention manager and our aggressive QoS contention manager using 4 threads assigned priorities of 10%, 20%, 30% and 40% respectively with 90% update rate. We ran this experiment 50 times and recorded the average and standard deviation for the CRR for each thread. The tables I and II depict the results of the experiment.

<table>
<thead>
<tr>
<th>TABLE I AGGRESSIVE QoS CM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average</strong></td>
</tr>
<tr>
<td><strong>STD</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE II AGGRESSIVE CM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average</strong></td>
</tr>
<tr>
<td><strong>STD</strong></td>
</tr>
</tbody>
</table>

As can be seen, Table I shows that the CRR for the Aggressive QoS CM varies for each thread, and that the relative relation of these CRRs match the relative relation of the threads’ priorities. It can also be noticed that while their relative relation match, the actual values do not. This is a verification of Theorem 2. Table II also makes perfect sense, since the aggressive contention manager does not take any criteria into considerations but simply aborts the transaction that is in conflict with the current one, it makes sense the transactions’ CRRs should be about 50%. Remember that CRRs are merely an indication of the number of times a transaction will win the conflicts it is involved in. In the aggressive CM, this is basically a coin toss, so it makes sense that it is 50%. All standard deviations are low, indicating repeatable results.

Next we attempt to verify Theorem 1 and Corollary 1. In order to do this, we again perform two experiments, this time each with only two threads. In the first experiment the relative priorities are 20% and 80%, and in the second experiment the relative priorities are 40% and 60% respectively. Theorem 1 and Corollary 1 state that the CRRs should be close to the priorities of the threads, but not exactly the same since the experiments will not run for infinity (in fact, we ran each experiment for 10 seconds). Tables III and IV depict the result of the experiment.

The results clearly corroborate Theorem 1 and Corollary 1 in that the CRRs closely track the priorities of the two threads, but are not exactly equal since we do not run the experiments for an infinite amount of time.
to better system-wide benefit. We propose to further study this problem in the future in order to be able to better understand the relative roles of “return on investment” and backoff in CMs that attempt to provide QoS guarantees. We also propose to come up with more rigorous results that allow us to make stronger theoretical claims about the performance of QoS CMs – particularly in the more than two thread case in terms of CRRs. Finally, we hope to be able to test our proposed solution on a system with many more cores than we currently have available to see how well they scale.

REFERENCES


TABLE III
AGGRESSIVE QoS CM

<table>
<thead>
<tr>
<th></th>
<th>20%</th>
<th>80%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>16.6%</td>
<td>82.3%</td>
</tr>
<tr>
<td>STD</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

TABLE IV
AGGRESSIVE QoS CM

<table>
<thead>
<tr>
<th></th>
<th>40%</th>
<th>60%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>38.5%</td>
<td>60.5%</td>
</tr>
<tr>
<td>STD</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

B. Modified Polite QoS CM

In this section of the experimental results, we examine the ability of the proposed algorithms to obtain better NASB than the literature. Towards that end, we ran all algorithms using the same experimental setup previously mentioned for 50 times and recorded the average and standard deviation for the NASB for each algorithm. Table V depicts the results.

TABLE V
NASB EXPERIMENT

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>54.4%</td>
<td>76.5%</td>
<td>85.3%</td>
<td>95%</td>
<td>97.4%</td>
</tr>
<tr>
<td>STD</td>
<td>0.7</td>
<td>6.6</td>
<td>2.2</td>
<td>0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

The results clearly corroborate Theorem 3 and Claim 1. The cautious algorithms produce better results, and the Greedy Polite QoS CM produces better results than any of the other algorithms. The surprising data point is the 0% standard deviation of the Polite algorithm, it was not designed to take into account priority in anyway, but seems to consistently produce the same value for NASB. In fact it produces better results that the Greedy Aggressive algorithm, suggesting that the backoff period may be even more important than the idea of “return on investment”. However, when “return on investment” and backoff are combined in the Greedy Polite algorithm, the results are even better.

VII. CONCLUSION AND FUTURE WORK

By considering several different methods for providing QoS guarantees to CM in STM we have learned a couple of important things. First, it is impossible to make exact arguments about the value of CRR when there are more than two threads in the system. However, despite this fact, the CRR of such systems will more closely mimic their priority than if no QoS algorithms are considered. For two thread systems, it is possible to accurately predict the value of CRRs. We also learned that using the idea of “return on investment” can improve the overall performance of a system in terms of total accrued benefit, but that backing off instead of making rash decisions to abort or commit immediately also contributes