Starvation-Free Transactional Memory-System Protocols*

Mridha Mohammad Waliullah and Per Stenstrom

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96, Göteborg, Sweden
{waliulla,pers}@ce.chalmers.se

Abstract. Transactional memory systems trade ease of programming with run-time performance losses in handling transactions. This paper focuses on starvation effects that show up in systems where unordered transactions are committed on a demand-driven basis. Such simple commit arbitration policies are prone to starvation. The design issues for commit arbitration policies are analyzed and novel policies that reduce the amount of wasted computation due to roll-back and, most importantly, that avoid starvation are proposed. We analyze in detail how to incorporate them in a TCC-like transactional memory protocol. The proposed schemes have no impact on the common-case performance and add quite modest complexity to the baseline protocol.

Keywords: Multiprocessors, transactional memory, starvation.

1 Introduction

As multi-core architectures are becoming commonplace, the need to make parallel programming easier is becoming acute. Transactional memory (TM) [1,2,3,4,7] promises to reduce the programming effort by relieving the programmer from resolving complex, fine-grain, inter-thread dependences by classical synchronization primitives such as locks and event synchronizations. Instead, coarse program segments form transactions that will either execute atomically or not at all. If transactions run by different threads have no dependencies, they can run concurrently. On the other hand, if a data dependency or a conflict appears, one of the transactions is squashed and must re-execute. Therefore, transactional memory trades programming simplicity for wasted execution at run-time.

Conflicts can be detected eagerly or lazily [4]. Under lazy conflict detections, such as TCC [2], the modifications done by a transaction are isolated until the point when the transaction is to commit. When the transaction commits, other transactions that have speculatively read data modified by the committing transaction will be squashed. Squashing does not only waste useful work. We show in this paper that it can cause starvation.

* This research is sponsored by the SARC project funded by the EU under FET. The authors are members of HiPEAC - a Network of Excellence funded by the EU under FP6.

This paper makes several contributions. Firstly, it analyzes in detail how to implement feasible commit arbitration schemes for TCC-like TM protocols. Secondly, and most importantly, it contributes with two novel starvation-free commit arbitration policies. Our overall approach to detect and remedy a potential starvation problem is to track how many times a certain transaction has been squashed. At the time a thread is ready to commit, it will not be allowed to do so if there is an ongoing transaction that has been squashed more times than the committing one. Then, the committing thread is stalled until that transaction has committed. Apart from avoiding devastating starvation situations, we show experimentally, using eight applications from SPLASH-2, that our starvation-free policies have virtually no impact on common-case performance and that they can be implemented with minor modifications to a TCC-like TM protocol.

As for the rest of the paper, we first introduce the architectural framework and frame the problem in more detail in Section 2. Section 3 is devoted to the novel arbitration schemes and especially how they are incorporated in the architectural framework. We then move on to the experimental results in Sections 4-5 by first describing the methodology. Section 6 puts our work in context of the TM literature and we conclude in Section 7.

2 System Framework

In this section, we define the framework of our study including the software assumptions in Section 2.1 and the architectural framework in Section 2.2. Finally, we frame the problem addressed by this research in detail in Section 2.3.

2.1 Software Assumptions

We assume that parallel programs uses transactions only and that a transaction is annotated by the programmer using start transaction (tx_begin) and end transaction (tx_end) constructs. In case of parallel applications using critical sections and barriers transactions are formed so that the following simple rules are followed: 1) critical sections are guaranteed to be encapsulated within a transaction and 2) a transaction is terminated and a new transaction starts at a barrier. However, a transaction can be terminated and a new one can start between two barriers as long as it happens outside critical sections [6,8,9,10].

In the assumed TM system, threads can execute beyond a barrier as long as they do not conflict with a thread that has not reached the barrier. This is supported using the notion of ordered as well as unordered transactions. A *phase number* is associated with each transaction which is incremented when a barrier is passed. All transactions that are started after the dynamic invocation of a certain barrier get the same phase number. If two transactions have the same phase number they are unordered; otherwise they are ordered and must commit in the ascending order of their phase numbers. Let's next consider the system model that supports this software model.

2.2 Architectural Framework

We consider a multi-core system that consists of n nodes where each node consists of a processor core (or core for simplicity) with its private L1 (optionally L2) cache connected to a shared L2 (optionally L3) cache via a bus or other broadcast medium according to Figure 1. Each core can be optionally (simultaneously) multithreaded with k hardware threads, where k is typically a small number (four or less).

Each L1 cache is extended with meta data to keep track of which blocks have been speculatively read and written using a read (R) and a write (W) bit, respectively, by setting the corresponding bits. When a transaction is finished, it will try to commit by requesting the bus. If a block that has been speculatively written is replaced, it is placed in a victim cache (VC) attached to each L1 cache. VC overflow is treated as described in [2].

In the baseline system, multiple commit requests are arbitrated through a central arbitration unit (denoted as CAU in Figure 1). To adhere to the semantics of ordered transactions, it attempts to select a committing transaction among the ones with the lowest phase number. Among these unordered transactions, it selects a candidate using FIFO.

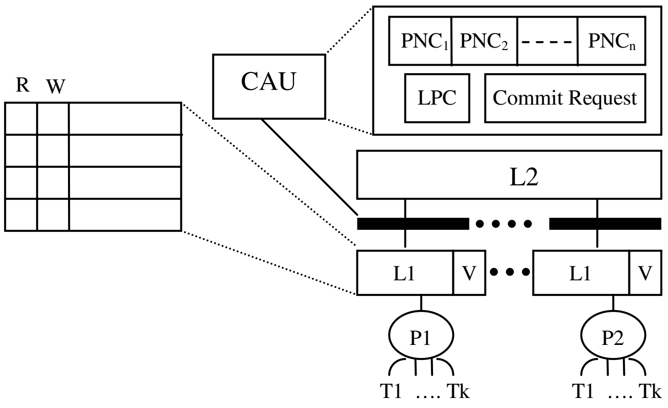


Fig. 1. Baseline architecture framework

To implement the baseline arbitration policy, the CAU uses three components: phase number counters (PNC), a lowest phase counter (LPC), and a FIFO with all commit-requests. The PNCs keep track of the current phase number of each thread using N = n x k phase-number counters, given n nodes and k hardware threads per node.

There are two types of commit requests: ordered and unordered. When a thread passes a barrier, an ordered commit request is sent. When an ordered commit request is granted by the CAU, the corresponding PNC is incremented. The LPC keeps track of the lowest phase number of any thread, i.e., min(PNCi), i=1,…,N. Finally, the FIFO simply keeps all pending commit requests on a first-come, first serve basis. Given these components, the CAU uses the LPC to filter out the requests from the FIFO that can commit, i.e., the transactions having the lowest phase numbers. It then picks the first one of these in the FIFO queue. A node with a granted commit request broadcasts its write set (the set of blocks having a W-bit set) to all L1 caches. All nodes with a block belonging to the write set and with its R-bit set will be notified to squash their ongoing transactions. Squashing a transaction involves the following steps: 1) invalidate all blocks having either the R or W-bit set; 2) gang-clear all R and W bits; and, 3) restart the transaction by reinstalling the architectural state.

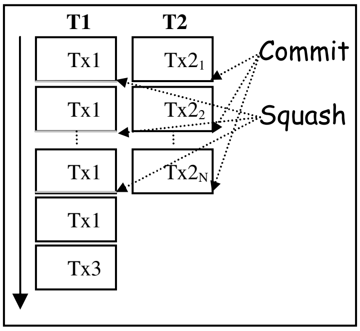
2.3 A Starvation Scenario

A major limitation of any transactional memory system is the performance lost due to squashes. More seriously, the simple arbitration policy assumed in the original TCC proposal is actually prone to starvation as the example in Figure 2 clearly demonstrates.

```
T1{
   Tx1_begin{
   ...=X;
   ...
   ...
   }
   Tx3_begin{
    Unrelated();
   }
}
```

```
T2{
for(i=1;i++;i<N){
    Tx2_begin{
    ...
    X=...;
    ...
    }
}
}
```

A



B

Fig. 2. A starvation scenario

Let's consider two threads (T1 and T2) that execute the code in Figure 2A. T1 executes a transaction (Tx1) that reads variable X followed by another transaction that does not conflict with any (Tx3). On the other hand, T2 executes a transaction (Tx2) that modifies X N times. Further, the execution time of Tx1 is assumed to be longer than that of Tx2.

Now consider the execution scenario in Figure 2B in which the execution of Tx1, Tx2, and Tx3 is tracked along the vertical time axis and where the commit and squash points are marked. As Tx1 and Tx2 obviously conflict, Tx2 will successfully commit whereas Tx1 will be squashed. As T2 will invoke Tx2 again, while T1 attempts to re-execute Tx1, the same scenario may repeat N times. As a result, the execution of all transactions will be serialized.

In another scenario, assuming the same code, Tx1 could have successfully committed before the first invocation of Tx2. Then, the execution of Tx3, and possibly subsequent transactions, could overlap the execution of the N-1 invocations of Tx2. Clearly, although the commit arbitration mechanism assumed in TCC meets fairness objectives at the level of commit requests between ordered and unordered transactions, it may result in starvation at the software level.

We have observed that a nearly as bad scenario showed up for Raytrace in SPLASH-2 (to be discussed in Section 5). The key reason for the devastating scenario of Figure 2B is that the CAU is completely unaware of the history by which transactions get squashed. In the next section, we propose novel arbitration policies that address this shortcoming.

3 Starvation-Free Arbitration Policies

To avoid starvation, the overall approach is to give priority to ongoing transactions that have already suffered from being squashed. We consider two schemes that differ in the way transactions are given priority to be selected and in implementation complexity. They are referred to as the naïve and the elaborate schemes. Both schemes extend the already existing central arbitration unit in the baseline system.

3.1 The Naïve Scheme

Recall that the baseline scheme selects a transaction to commit among the threads with the lowest phase number on a first-come-first-serve basis. In the naïve scheme, we also select a thread to commit in this set. However, we put an additional constraint on which threads to commit based on how many times their ongoing transactions have been squashed. A thread can only commit if it has been squashed at least the same number of times as any other thread with the lowest phase number. Unlike in the baseline scheme, the set of threads that can commit can be the empty set, i.e., all threads attempting to commit so far can be stalled.

To keep track of the number of squashes per thread, the CAU is extended with N miss-speculation counters (MSC), assuming N threads, where each MSC is initially cleared. When a thread suffers from a squash, the corresponding MSC is incremented. Conversely, when a thread commits its transaction, the corresponding MSC is cleared. In Figure 3, we show the extra mechanisms needed by the naïve scheme.

The CAU obviously knows when a thread can commit its ongoing transaction but lacks knowledge about when a transaction is squashed. Hence, it needs to know when to increment the MSCs. This is accomplished by requiring that each time a node selects to squash a transaction, it notifies the CAU so that the corresponding MSC can be incremented. Hence, the commit protocol involving broadcasting the write set and matching it against the read set in each node, must be extended with a final phase to notify the CAU about the hardware threads on each node that were squashed.

One solution is to let all nodes serially report to the CAU about squashed transactions. This would obviously cause a lot of overhead so we propose the following solution. All hardware threads have a SQUASH signal that is raised when a transaction has been squashed. Further, the bus is extended with N dedicated lines; one for each SQUASH-signal. These lines are used as increment-enable signals to the set of MSCs maintained by the CAU as shown in Figure 3.

In summary, in terms of structures, the naïve scheme is extended with as many miss-speculation counters as the number of hardware threads. Additionally, it needs a

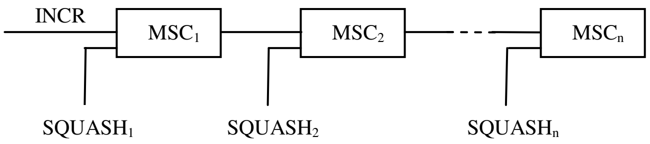


Fig. 3. Mechanisms used by the naive scheme not part of the baseline

special-purpose bus with as many lines as the number of hardware threads. Given the fairly limited number of nodes and hardware threads per core in a multi-core chip for a foreseeable future, this solution appears to be reasonable.

3.2 The Elaborate Scheme

In the naïve scheme, a committing transaction is stalled when any transaction with the lowest phase number and a higher MSC exists in the system disregarding the fact that these transactions could be data independent of each other. Intuitively, it would be unwise to stall a transaction that has no conflict with a transaction that has a higher MSC because, then, it is not the cause of a potential starvation scenario.

In the elaborate scheme, a committing transaction that has the lowest phase number and has no conflict with any transaction with the same phase number and a higher MSC is allowed to commit. Conversely, the CAU will stall a transaction either if the committing transaction has not the lowest phase number (as in the baseline) or it has the lowest phase number and it conflicts with another transaction that has the same phase number and a higher MSC.

Unfortunately, the CAU has no knowledge about what transactions would be squashed if it grants a commit request. Therefore, we need to extend the naïve scheme by breaking up a commit operation into two phases: 1) check-for-data-races 2) make-commit-decision where the first phase is not part of the naïve and the baseline scheme.

To check for data conflicts, the committing node first broadcasts the write set of its transaction. Each node checks locally whether its ongoing transaction conflicts with the committing transaction. If this is the case, it activates the SQUASH signal introduced in the naïve scheme (see Figure 3). The CAU will check the MSCs of the transactions with their SQUASH signals activated and if any of them is higher than the MSC of the committing node and they have the same (lowest) phase number, the committing node is not allowed to commit. On the other hand, if the committing transaction will be allowed to commit, the CAU will notify the nodes that have their SQUASH signals activated to squash their ongoing transactions.

In summary, in comparison with the naïve scheme, the elaborate scheme must extend the commit protocol with a phase that checks whether the transaction that requests to commit can do so without squashing a thread that has the same phase number but a higher MSC value. Note that even if a committing transaction will stall, a bus transaction that checks for conflicts is still needed. Thus, the elaborate scheme causes more traffic than the naïve scheme.

4 Experimental Methodology

In order to analyze whether the new arbitration schemes result in fewer useless cycles in terms of delays (waiting time for commit decision) or miss-speculations, we have built a simulation model of the baseline system and augmented it with the naïve and the elaborate scheme.

As we are only concerned with the number of cycles lost due to stalling a committing thread and a thread that miss-speculates, we have opted for a simple model of the memory system. Hence, we do not charge any cycles for cache misses at any level of the memory hierarchy. We model a 16-core system with a single hardware thread per core. Each node has a 128-Kbyte, 4-way L1 cache with a block size of 16 bytes. This would correspond to multi-cores with a combined L1/L2 inclusive cache hierarchy of the same capacity. Further, we maintain an infinite buffering space for speculatively modified blocks by assuming an infinite victim cache.

The system model is driven by traces generated by the eight SPLASH-2 applications [11] in Table 1. We use a trace-driven methodology to drive the system model. Each of the eight benchmarks is first run on Simics [5] with Sun Sparc as the target machine. Benchmarks are run with the original synchronization primitives and we collect statistics only in the parallel phase of each benchmark.

Table 1. Benchmarks and their data sets

Applications	Inputs	Applications	Inputs
Barnes	2048 particles	Ocean	34x34 grid
FFT	2^{16}-data points, cache line size-16, number of cache line-8K	Radix	1024 radix, 256K keys
		Raytrace	teapot.env
FMM	2048 particles	Volrend	Head
LU-Non contig.	512x512 matrix		

Transactions are marked by using the Simics magic instructions in-lined in the implementation of locks and barriers in the ANL-macros. There is a magic instruction at the beginning and at the end of these macros. All memory references between these magic instructions are filtered out of the trace.

A new transaction starts at a barrier and will terminate at the next barrier or after 1000 instructions, which of them happens first, unless that point occurs inside a critical section. If so, the transaction is continued until the corresponding unlock construct is executed. For nested locks, we flatten them out with the outermost lock, i.e., no transaction can start or terminate within the boundary of this outermost critical section.

It is well-known that a trace-driven approach that we use will not reflect the correct interleaving of events. However, as our goal is not to assess absolute performance but rather to compare the number of useless cycles using the different arbitration policies, we feel that the methodology adopted is adequate for this purpose.

5 Experimental Results

In Section 5.1, we experimentally confirm that the starvation scenario in Section 2.3 can happen. Then, we analyze the tradeoffs between the baseline, the naïve scheme, and the elaborate scheme in detail in Section 5.2

5.1 The Case for Starvation of the Baseline Protocol

Figure 4 shows the execution time of each of the 16 threads normalized to the slowest running thread when running the application Raytrace on three systems. For each thread, the three bars correspond to the baseline system, the naïve scheme, and the elaborate scheme, from left to right. Further, each bar is decomposed into three

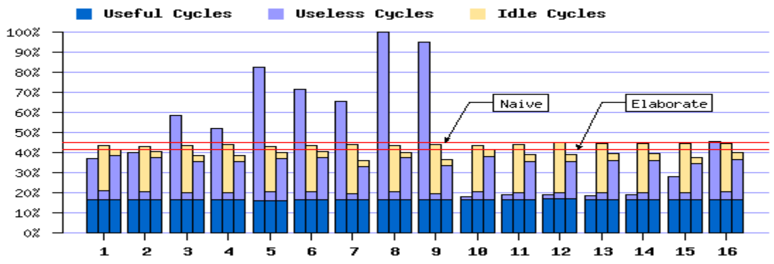


Fig. 4. Execution time of individual threads in Raytrace normalized to the slowest thread. The first, second, and third bars in each cluster correspond to the baseline, the naive and the elaborate arbitration scheme respectively.

sections that break down the execution time into useful cycles, useless cycles, and idle cycles (waiting for commit decision), from bottom to top.

As can be seen in Figure 4, there is a huge difference in execution time between different threads under the baseline arbitration scheme. For example, the baseline system execution time for thread 8 is five times longer than that of thread 10! However, the number of useful cycles across the threads is about the same so the effect is not attributed to load imbalance. The difference stems from the number of useless cycles. Trace inspection has revealed that Raytrace suffers from a similar starvation situation as described in Section 2.3 which degrades the performance.

Considering the execution times across threads for the naïve and the elaborate schemes (the two rightmost bars in each cluster), we can see that the difference between the slowest and the fastest threads is small. This suggests that these schemes successfully eliminate starvation. In fact, the execution time, which is dictated by the slowest thread, is cut down by as much as between 55% and 59% using the naïve and the elaborate schemes.

Continuing with the difference between the naïve and the elaborate schemes, it is clear from Figure 4 that the naïve scheme suffers a lot from stalling. However, it is interesting to note that the elaborate scheme also suffers from useless cycles which are attributed to miss-speculations. A close inspection of Figure 4 reveals that the naïve scheme tends to reduce the number of cycles lost for miss-speculations at the expense of cycles lost for stalling threads. However, the elaborate scheme performs slightly better than the naïve scheme – the execution time is around 4% shorter.

5.2 Tradeoffs Between the Naïve and the Elaborate Scheme

In Figure 5, we present the same results as in Figure 4 but for the ocean application. A striking observation is that the naïve scheme suffers from a huge idle time. This is caused by transactions that have to wait for unrelated transactions with a higher miss-speculation count to commit. This is unfortunate, as the waiting transactions could have been committed in the first place.

On the other hand, the elaborate scheme manages to keep the number of stall cycles low but at the expense of extending a commit transaction with a "check-for-data-races" phase. It is interesting to understand whether a simple modification of the naïve scheme could reduce the number of idle cycles.

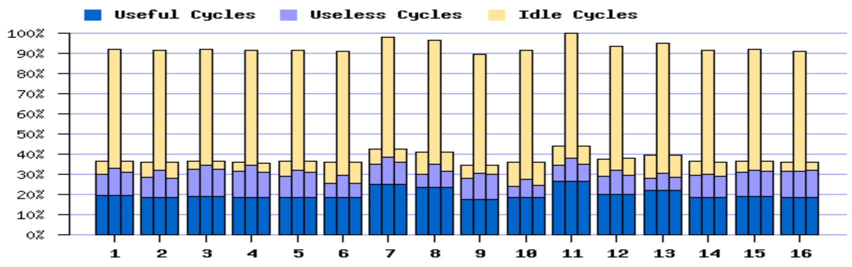
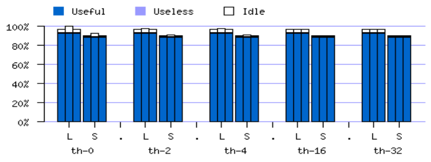
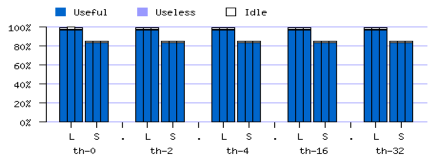


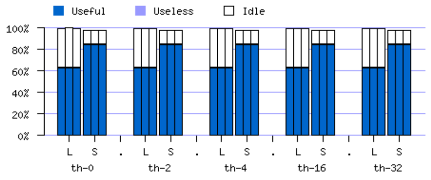
Fig. 5. Execution time of individual threads in Ocean normalized to the slowest thread. The first, second, and third bars in each cluster correspond to the baseline, the naive, and the elaborate arbitration scheme respectively.



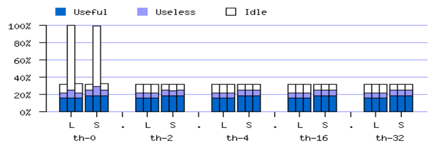
(a) Barnes



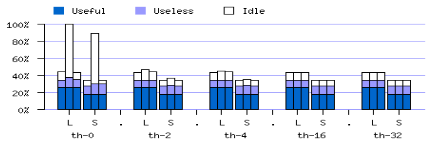
(b) FFT



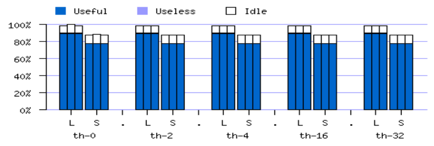
(c)FMM



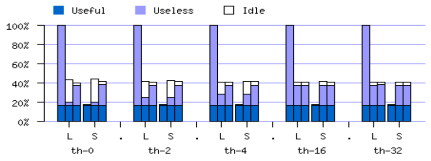
(d) LU-Nc



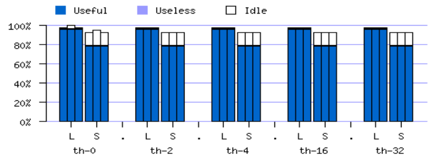
(e) Ocean



(f) Radix



(g) Raytrace



(h) Volrend

Fig. 6. Relative performance of the commit arbitration schemes for the eight applications using threshold values 0, 2, 4, 16, and 32 plotted through diagrams (a) to (h). The 1st and 2nd clusters under each threshold represent the execution time for the slowest and fastest running threads respectively. Each cluster uses three bars that represent the relative execution time for the same thread using all three arbitration schemes.

This encouraged choosing threads that have to stall under the naïve scheme more selectively. Recalling the example starvation scenario in Section 2.3, the chief observation is that a symptom of a starvation situation is that a request has been denied several times in a row. In our improved naïve scheme, we allow a transaction (with the lowest phase number) to commit if its MSC count is lower than another one (with the lowest phase number) minus an offset, which we refer to as a threshold.

Obviously, a key issue is to select an appropriate threshold. Therefore, we experimented with several threshold values. Figure 6 shows the performance of the eight applications under the naïve scheme using threshold values of 0, 2, 4, 16, and 32 for the naïve scheme which correspond to the different bar clusters (each consisting of six bars) in each diagram. In the figure, we have depicted the execution time for the longest (L) and the shortest (S) running thread for each application. This enables us to pinpoint a starvation scenario if it exists.

As we go from lower to higher threshold values, the idle time for the naïve scheme is reduced significantly. It is important to note that it becomes as low as the baseline arbitration for a threshold value of 16 for the applications which do not encounter any starvation.

When an application uses barrier synchronizations, the performance of an application that is run under a transactional memory paradigm will suffer from idle cycles lost due to ordered commits. This is because a transaction with a higher phase number must wait for a transaction with a lower phase number to commit. As a result, also the baseline scheme suffers from idle cycles in all applications that use barriers to synchronize as can be seen from Figure 6. One exception is Raytrace which does not use barriers.

A striking observation is that as we increase the threshold, the performance difference between the naïve scheme and the elaborate scheme diminishes. Therefore, our recommendation is that the naïve scheme with a fairly high threshold (16 for example) can safely eliminate starvations.

6 Related Work

We have proposed arbitration schemes for TM systems to avoid starvation. Hill et al. have classified TM systems in two groups depending on lazy/eager conflict detection [4]. While [1,4,7] are referred to as eager, [2] does lazy conflict resolution. This paper has focused on the starvation problem for TM systems using lazy conflict resolution.

Rajwar et al. [9] proposed to timestamp a transaction when it is instantiated for prioritizing transactions and in a conflicting scenario lower priority transactions rollback or wait. The policy ensures forward progress and starvation freedom in a system where conflicting scenario is visible before making a commit decision. The author also assumed a system configuration where conflict is visible at the time of the conflicting memory access. On the other hand, in a lazy conflict detecting system, conflict is visible only after a transaction commit and memory modification is revealed to others. However, we can use the timestamp based prioritization scheme to serialize all commits disregarding the fact that many transactions don't have any conflict among them (independent transactions). Nevertheless, a concern is that it would unnecessarily force independent transactions to wait.

TM systems that detect conflict lazily, know about the conflict at the commit point of any of the involving transactions. The straightforward first-come first-serve arbitration policy used in TCC may lead to a starvation as described in Section 2.3. Our proposed arbitration scheme targets a TM system that detects conflicts lazily and especially uses a central arbitration unit to select a thread to commit.

Hammond et al. [2] propose the use of pseudo-barriers to make forward progress for all processors. The idea is to arbitrarily insert a barrier implicitly so that a thread that is subject to starvation will get a chance to catch up. Unfortunately, the paper does not elaborate on how to make use of the idea. In fact, we have shown in this paper that it is important to monitor when a starvation scenario is about to happen. We do this by using miss-speculation counters. By arbitrarily inserting barriers, there is simply no such monitoring mechanism to provide feedback.

7 Concluding Remarks

In this paper, we have proposed novel commit arbitration schemes for TM systems that avoids starvation. We show how they can be implemented in a framework based on TCC. As a general approach to avoid starvation, we propose that the commit arbitration policy should be provided with feedback on squash counts of other threads. To this end, we propose the naïve scheme and the elaborate scheme. Through a detailed implementation and performance evaluation study we found the following. Both proposed schemes can avoid starvation but do it at the expense of lost cycles due to delaying the point at which a thread can commit.

We analyzed the design space of the simplest policy – the naïve policy – and found that it can eliminate most of the stall cycles by introducing a threshold value between the committing thread's and the other ongoing thread's miss-speculation count. By doing this, we have found that naïve manages to remove most of the stall cycles at the expense of some quite modest structure and protocol extensions. Overall, this paper shows that it is possible to avoid starvation of transactional memory protocols at a modest implementation cost.

References

1. Ananian, C.S., Asanovi'c, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded Transactional Memory. In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, San Franscisco, CA, February 2005, pp. 316–327 (2005)
2. Hammond, L., Wong, V., Chen, M., Hertzberg, B., Carlstrom, B., Davis, J., Prabhu, M., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional Memory Coherence and Consistency. In: Proc. of the 31st Annual International Symposium on Computer Architecture, München, Germany, June 19-23, pp. 102–113 (2004)
3. Herlihy, M., Moss, J.E.B.: Transactional Memory: architectural support for lock-free data structures. In: Proceedings of the 20th International Symposium on Computer Architecture, pp. 289–300 (1993)
4. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based Transactional Memory. In: Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12), Austin, TX (February 11-15, 2006)

5. Magnusson, P.S., Christianson, M., Eskilson, J., et al.: Simics: A full system simulation platform. IEEE Computer 35(2), 50–58 (2002)
6. Martinez, J., Torrellas, J.: Speculative synchronization: Applying thread-level speculation to parallel applications. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (October 2002)
7. Rajwar, R., Herlihy, M., Lai, L.: Virtualizing transactional memory. In: Proceedings of the 32nd International Symposium on Computer Architecture, June 2005, pp. 494–505 (2005)
8. Rajwar, R., Goodman, J.: Speculative Lock Elision: enabling highly concurrent multithreaded execution. In: MICRO 34: Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture, pp. 294–305. IEEE Computer Society Press, Los Alamitos (2001)
9. Rajwar, R., Goodman, J.: Transactional Lock-free Execution of Lock-Based Codes. In: Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (October 2002)
10. Rundberg, P., Stenstrom, P.: Reordered speculative execution of critical sections. In: Proceedings of the 2002 International Conference on Parallel Processing (February 2002)
11. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: Proceedings of the 22nd International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995, pp. 24–36 (1995)